デベロッパーガイド、バージョン 1

AWS IoT Greengrass

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS IoT Greengrass: デベロッパーガイド、バージョン 1

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスはAmazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

..... xxi
とは AWS IoT Greengrass 1
AWS IoT Greengrass コアソフトウェア 3
AWS IoT Greengrass コアソフトウェアバージョン 3
AWS IoT Greengrass グループ 14
のデバイス AWS IoT Greengrass 16
SDK 19
サポートされているプラットフォームと要件 20
AWS IoT Greengrass ダウンロード 34
AWS IoT Greengrass コアソフトウェア 34
AWS IoT Greengrass スナップソフトウェア 42
AWS IoT Greengrass Docker ソフトウェア 43
AWS IoT Greengrass コア SDK 45
サポートされている機械学習ランタイムおよびライブラリ 46
AWS IoT Greengrass ML SDK ソフトウェア 47
ご意見をお待ちしております 47
AWS IoT Greengrass Core ソフトウェアのインストール 47
tar.gz ファイルのダウンロードと抽出 48
Greengrass デバイスのセットアップスクリプトを実行する 48
APT リポジトリからのインストール 48
Docker コンテナ AWS IoT Greengrass で を実行する 51
スナップ AWS IoT Greengrass で実行する 51
コアソフトウェアのインストールのアーカイブ 63
AWS IoT Greengrass コアを設定する 65
AWS IoT Greengrass コア設定ファイル 66
サービスエンドポイントは証明書タイプと一致する必要がある 130
ポート 443 での接続またはネットワークプロキシを通じた接続 132
書き込みディレクトリの設定 142
MQTT 設定の構成 146
自動 IP 検出のアクティブ化 164
システムの起動時に Greengrass を開始する 168
関連情報 169
AWS IoT Greengrass V1 メンテナンスポリシー 170
AWS IoT Greengrass バージョニングスキーム 170

AWS IoT Greengrass Core ソフトウェアのライフサイクルフェーズ 171
AWS IoT Greengrass Core ソフトウェアのメンテナンスポリシー 171
メンテナンスフェーズのスケジュール 172
非推奨スケジュール 172
Lambda 関数のサポートポリシー 172
の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1 173
メンテナンススケジュールの終了 173
AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージのメンテナンスの終了 43
AWS IoT Greengrass Core ソフトウェア v1.x APT リポジトリのメンテナンスの終了 175
AWS IoT Greengrass Core ソフトウェア v1.11.x スナップのメンテナンスの終了 175
の開始方法 AWS IoT Greengrass 176
開始方法の選択 176
要件 179
を作成する AWS アカウント 180
にサインアップする AWS アカウント 181
管理アクセスを持つユーザーを作成する 181
クイックスタート: Greengrass デバイスのセットアップ 183
要件 183
Greengrass デバイスのセットアップを実行する 184
問題のトラブルシューティング 188
Greengrass デバイスセットアップ設定オプション 189
モジュール 1: Greengrass の環境設定 199
Raspberry Pi のセットアップ 200
Amazon EC2 インスタンスの設定 208
他のデバイスの設定 214
モジュール 2: AWS IoT Greengrass Core ソフトウェアのインストール 218
Greengrass コアとして使用する AWS IoT モノをプロビジョニングする 219
Greengrass グループを作成します 222
AWS IoT Greengrass コアデバイスにインストールして実行する 224
モジュール 3 (パート 1): での Lambda 関数 AWS IoT Greengrass 231
Lambda 関数の作成とパッケージ化 232
の Lambda 関数を設定する AWS IoT Greengrass 237
コアデバイスへのクラウド設定のデプロイ 241
Lambda 関数がコアデバイスで実行されていることを確認する 242
モジュール 3 (パート 2): での Lambda 関数 AWS IoT Greengrass 243
Lambda 関数の作成とパッケージ化 244

の存続期間の長い Lambda 関数を設定する AWS IoT Greengrass 248
存続期間の長い Lambda 関数のテスト 249
オンデマンド Lambda 関数のテスト 252
モジュール 4: AWS IoT Greengrass グループ内のクライアントデバイスを操作する 256
AWS IoT Greengrass グループにクライアントデバイスを作成する 258
サブスクリプションを設定する 261
AWS IoT Device SDK for Python のインストール 262
通信をテストする 269
モジュール 5: デバイスシャドウの操作 273
デバイスとサブスクリプションの設定 275
必要なファイルのダウンロード 278
通信をテストする (デバイス同期を無効にする) 278
通信をテストする (デバイス同期を有効にする) 282
モジュール 6: 他の AWS サービスへのアクセス 283
グループロールの設定 285
Lambda 関数の作成と設定 287
サブスクリプションを設定する 290
通信をテストする 291
モジュール 7: ハードウェアセキュリティ統合のシミュレーション 293
SoftHSM をインストールする 294
SoftHSM を設定する 294
プライベートキーをインポートする 296
Greengrass Core を設定する 297
設定をテストする 301
関連情報 301
AWS IoT Greengrass Core ソフトウェアの OTA 更新 303
要件 303
OTA 更新の IAM アクセス許可 304
考慮事項 307
Greengrass OTA Update Agent 308
init システムとの統合 309
OTA 更新による管理された再生成 309
OTA 更新の作成 311
CreateSoftwareUpdateJob API 314
AWS IoT Greengrass グループのデプロイ 318
グループのデプロイ (コンソール) 319

グループのデプロイ (API) ... 320
グループ ID の取得 ... 322
グループオブジェクトモデルの概要 ... 323
グループ ... 323
グループバージョン ... 324
グループコンポーネント ... 325
グループの更新 ... 326
関連情報 ... 327
デプロイ通知の取得 ... 328
グループデプロイステータスの変更イベント ... 329
EventBridge ルールを作成するための前提条件 ... 330
デプロイ通知の設定 (コンソール) ... 331
デプロイ通知の設定 (CLI) ... 332
デプロイ通知の設定 (AWS CloudFormation) ... 333
関連情報 ... 333
デプロイのリセット ... 333
AWS IoT コンソールからデプロイをリセットする ... 334
AWS IoT Greengrass API を使用してデプロイをリセットする ... 335
関連情報 ... 336
一括デプロイの作成 ... 336
前提条件 ... 337
一括デプロイ入力ファイルを作成してアップロードする ... 337
一括デプロイ用の IAM 実行ロールを作成および設定する ... 340
実行ロールに S3 バケットへのアクセスを許可する ... 342
グループをデプロイする ... 344
デプロイをテストする ... 346
一括デプロイのトラブルシューティング ... 348
関連情報 ... 350
ローカル Lambda 関数を実行する ... 351
SDK ... 352
クラウドベースの Lambda 関数への移行 ... 355
エイリアスまたはバージョンによる関数のリファレンス ... 356
Greengrass Lambda 関数の実行の制御 ... 357
グループ固有構成設定 ... 357
root としての Lambda 関数の実行 ... 361
Lambda 関数のコンテナ化を選択する場合の考慮事項 ... 362

グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定 366
グループ内の Lambda 関数のコンテナ化のデフォルト設定 368
通信フロー 369
MQTT メッセージを使用した通信 369
他の通信フロー 370
入力トピック (または件名) の取得 371
ライフサイクル設定 373
Lambda 実行可能ファイル 374
Lambda 実行可能ファイルを作成する 376
Docker コンテナ AWS IoT Greengrass で を実行する 377
前提条件 379
Amazon ECR から AWS IoT Greengrass コンテナイメージを取得する 380
Greengrass のグループとコアを作成して設定する 384
AWS IoT Greengrass ローカルで を実行する 384
グループの「コンテナなし」コンテナ化を設定する 388
Lambda 関数を Docker コンテナにデプロイする 388
(オプション) Docker コンテナで Greengrass を操作するクライアントデバイスをデプロイする 389
Docker AWS IoT Greengrass コンテナの停止 389
Docker コンテナ AWS IoT Greengrass でのトラブルシューティング 389
ローカルリソースへのアクセス 393
サポートされているリソースタイプ 393
要件 395
/proc ディレクトリのボリュームリソース 395
グループ所有者のファイルアクセス権限 396
関連情報 396
CLI の使用 396
ローカルリソースの作成 397
Greengrass 関数を作成する 399
グループに Lambda 関数を追加する 400
トラブルシューティング 402
コンソールを使用する 403
前提条件 404
Lambda 関数デプロイパッケージを作成する 405
Lambda 関数を作成して発行する 406
グループに Lambda 関数を追加する 409

グループにローカルリソースを追加する 410
サブスクリプションをグループに追加する 411
グループをデプロイする 412
ローカルリソースアクセスのテスト 413
機械学習の推論を実行する 416
ML AWS IoT Greengrass 推論の仕組み 416
機械学習リソース 417
サポートされているモデルソース 417
要件 420
ML 推論用のランタイムとライブラリ 420
SageMaker AI Neo 深層学習ランタイム 421
MXNet のバージョニング 421
Raspberry Pi の MXNet 421
Raspberry Pi での TensorFlow モデルの制限 422
機械学習リソースにアクセスする 422
機械学習リソースのアクセス権限 423
Lambda 関数のアクセス権限の定義 (コンソール) 425
Lambda 関数 (API) のアクセス権限の定義 426
Lambda 関数コードから機械学習リソースにアクセスする 429
トラブルシューティング 430
関連情報 432
機械学習推論を設定する方法 432
前提条件 433
Raspberry Pi を設定する 434
MXNet フレームワークをインストールする 436
モデルパッケージを作成する 437
Lambda 関数を作成して発行する 437
グループに Lambda 関数を追加する 440
グループにリソースを追加する 443
グループにサブスクリプションを追加する 445
グループをデプロイする 446
アプリケーションをテストする 447
次のステップ 451
インテル Atom の設定 451
NVIDIA Jetson TX2 の設定 455
最適化された機械学習推論を設定する方法 460

前提条件 ... 433
Raspberry Pi を設定する ... 461
Neo 深層学習ランタイムをインストールする ... 463
推論 Lambda 関数を作成する ... 464
グループに Lambda 関数を追加する ... 468
Neo 最適化モデルリソースをグループに追加する ... 470
カメラデバイスリソースをグループに追加する ... 472
サブスクリプションをグループに追加する ... 474
グループをデプロイする ... 474
例をテストする ... 476
インテル Atom の設定 ... 477
NVIDIA Jetson TX2 の設定 ... 480
ML AWS IoT Greengrass 推論のトラブルシューティング ... 448
次のステップ ... 486
データストリームの管理 ... 487
ストリーム管理ワークフロー ... 488
要件 ... 490
データセキュリティ ... 491
ローカルデータセキュリティ ... 491
クライアント承認 ... 492
関連情報 ... 493
ストリームマネージャーの設定 ... 493
ストリームマネージャーのパラメータ ... 493
設定の構成 (コンソール) ... 496
設定の設定 (CLI) ... 499
関連情報 ... 509
ストリームを操作するために StreamManagerClient を使用する ... 510
メッセージストリームの作成 ... 511
メッセージの追加 ... 515
メッセージの読み取り ... 522
ストリームの一覧表示 ... 525
メッセージストリームの説明 ... 526
メッセージストリームの更新 ... 528
メッセージストリームの削除 ... 533
関連情報 ... 534
AWS クラウド でサポートされている送信先のエクスポート設定 ... 534

データストリームのエクスポート (コンソール) ... 552
前提条件 ... 552
Lambda 関数デプロイパッケージを作成する ... 555
「Lambda 関数の作成」 ... 558
グループに関数を追加する ... 560
ストリームマネージャーを有効にする ... 561
ローカルログを設定する ... 562
グループをデプロイする ... 562
アプリケーションをテストする ... 564
関連情報 ... 565
データストリームのエクスポート (CLI) ... 565
前提条件 ... 566
Lambda 関数デプロイパッケージを作成する ... 569
「Lambda 関数の作成」 ... 573
関数の定義とバージョンを作成する ... 574
ロガー定義とバージョンを作成する ... 576
コア定義バージョンの ARN を取得する ... 578
グループバージョンを作成する ... 579
デプロイを作成する ... 579
アプリケーションをテストする ... 581
関連情報 ... 582
Core にシークレットをデプロイする ... 583
シークレットの暗号化 ... 584
要件 ... 585
シークレット暗号化用のプライベートキーを指定する ... 586
AWS IoT Greengrass シークレット値の取得を に許可する ... 588
関連情報 ... 589
シークレットリソースを使用する ... 589
シークレットの作成と管理 ... 590
ローカルシークレットの使用 ... 595
シークレットリソースを作成する方法 (コンソール) ... 598
前提条件 ... 599
Secrets Manager シークレットを作成する ... 600
グループにシークレットリソースを追加する ... 601
Lambda 関数デプロイパッケージを作成する ... 602
「Lambda 関数の作成」 ... 604

関数をグループに追加する 606
シークレットリソースを関数にアタッチする 608
サブスクリプションをグループに追加する 608
グループをデプロイする 609
Lambda 関数をテストする 611
関連情報 611
コネクタを使用してサービスおよびプロトコルと統合する 612
要件 613
Greengrass コネクタの使用 614
設定パラメータ 616
グループリソースへのアクセスに使用するパラメータ 616
コネクタのパラメータの更新 617
入力と出力 617
入力トピック 618
コンテナ化のサポート 619
コネクタのバージョンのアップグレード 619
ロギング 620
AWSが提供する Greengrass コネクタ 621
CloudWatch メトリクス 624
Device Defender 641
Docker アプリケーションのデプロイ 647
IoT Analytics 691
IoT イーサネット IP プロトコルアダプタ 708
IoT SiteWise 713
Kinesis Firehose 728
ML フィードバック 747
ML イメージ分類 764
ML オブジェクト検出 791
Modbus-RTU プロトコルアダプタ 808
Modbus-TCP プロトコルアダプタ 826
Raspberry Pi GPIO 832
シリアルストリーミング 843
ServiceNow MetricBase 統合 857
SNS 872
Splunk 統合 884
Twilio 通知 898

コネクタの使用を開始する (コンソール) ... 916
前提条件 ... 917
Secrets Manager シークレットを作成する ... 918
グループにシークレットリソースを追加する ... 919
グループにコネクタを追加する ... 920
Lambda 関数デプロイパッケージを作成する ... 921
「Lambda 関数の作成」 ... 922
グループに関数を追加する ... 925
サブスクリプションをグループに追加する ... 925
グループをデプロイする ... 926
ソリューションをテストする ... 928
関連情報 ... 929
コネクタの使用を開始する (CLI) ... 929
前提条件 ... 931
Secrets Manager シークレットを作成する ... 933
リソースの定義とバージョンを作成する ... 933
コネクタの定義とバージョンを作成する ... 934
Lambda 関数デプロイパッケージを作成する ... 936
「Lambda 関数の作成」 ... 937
関数の定義とバージョンを作成する ... 939
サブスクリプションの定義とバージョンを作成する ... 940
グループバージョンを作成する ... 941
デプロイを作成する ... 943
ソリューションをテストする ... 944
関連情報 ... 946
Greengrass Discovery RESTful API ... 947
リクエスト ... 947
レスポンス ... 948
検出の認可 ... 949
検出レスポンスドキュメントの例 ... 949
セキュリティ ... 952
AWS IoT Greengrass セキュリティの概要 ... 953
デバイス接続のワークフロー ... 954
AWS IoT Greengrass セキュリティの設定 ... 955
セキュリティプリンシパル ... 956
MQTT メッセージングワークフローにおけるマネージドサブスクリプション ... 959

TLS 暗号スイートのサポート 959
データ保護 962
データ暗号化 963
ハードウェアセキュリティ統合 966
デバイス認証と認可 987
X.509 証明書 988
AWS IoT ポリシー 990
コアデバイスの最小 AWS IoT ポリシー 993
Identity and Access Management 997
対象者 997
アイデンティティを使用した認証 998
ポリシーを使用したアクセスの管理 1001
関連情報 1004
が IAM と AWS IoT Greengrass 連携する方法 1004
Greengrass サービスロール 1013
Greengrass グループのロール 1021
サービス間の混乱した代理の防止 1032
アイデンティティベースのポリシーの例 1033
アイデンティティとアクセスの問題のトラブルシューティング 1036
コンプライアンス検証 1039
耐障害性 1041
インフラストラクチャセキュリティ 1042
設定と脆弱性の分析 1042
VPC エンドポイントAWS PrivateLink 1043
AWS IoT Greengrass VPC エンドポイントに関する考慮事項 1044
AWS IoT Greengrass コントロールプレーン操作用インターフェイス VPC エンドポイントの作成 1044
の VPC エンドポイントポリシーの作成 AWS IoT Greengrass 1045
セキュリティに関するベストプラクティス 1046
最小限のアクセス許可を付与する 1046
Lambda 関数で認証情報をハードコードしない 1046
機密情報を記録しない 1046
ターゲットを絞ったサブスクリプションの作成 1047
デバイスのクロックを同期させる 1047
Greengrass コアを使用したデバイス認証の管理 1047
関連情報 1049

ログ記録とモニタリング 1050
モニタリングツール 1050
関連情報 1051
AWS IoT Greengrass ログでのモニタリング 1051
CloudWatch Logs にアクセスする 1051
ファイルシステムログへのアクセス 1053
デフォルトのログ記録設定 1054
のログ記録を設定する AWS IoT Greengrass 1055
ログ記録の制限 1058
CloudTrail ログ 1060
を使用した AWS IoT Greengrass API コールのログ記録 AWS CloudTrail 1060
AWS IoT Greengrass CloudTrail の情報 1060
AWS IoT Greengrass ログファイルエントリについて 1061
関連情報 1065
システムヘルステレメトリデータの収集 1065
テレメトリ設定の構成 1068
テレメトリデータを受信するためのサブスクライブ 1072
AWS IoT Greengrass テレメトリのトラブルシューティング 1078
ローカルヘルスチェック API を呼び出す 1079
すべてのワーカーのヘルス情報を取得する 1079
指定されたワーカーに関するヘルス情報を取得する 1081
ワーカーのヘルス情報 1083
Greengrass リソースへのタグ付け 1086
タグの基本 1086
タグ付けのサポート (コンソール) 1086
タグ付けのサポート (API) 1087
IAM ポリシーでのタグの使用 1089
IAM ポリシーの例 1089
関連情報 1092
AWS CloudFormation のサポート AWS IoT Greengrass 1093
リソースの作成 1093
リソースのデプロイ 1094
サンプルテンプレート 1095
サポートされている AWS リージョン 1108
AWS IoT Device Tester for AWS IoT Greengrass V1 の使用 1109
AWS IoT Greengrass 認定スイート 1109

カスタムテストスイート 1110
AWS IoT Device Tester for AWS IoT Greengrass V1 でサポートされているバージョン 1110
for でサポートされていない IDT バージョン AWS IoT Greengrass 1111
IDT を使用して AWS IoT Greengrass 認定スイートを実行する 1116
テストスイートのバージョン 1117
テストグループの説明 1118
前提条件 1122
IDT テストを実行するようにデバイスを設定する 1132
IDT 設定を設定する 1155
AWS IoT Greengrass 認定スイートを実行する 1170
結果とログを理解する 1175
IDT を使用して独自のテストスイートを開発および実行する 1179
最新バージョンの IDT for AWS IoT Greengrassのダウンロード 1123
テストスイート作成ワークフロー 1180
チュートリアル: サンプル IDT テストスイートを構築して実行する 1180
チュートリアル: シンプルな IDT テストスイートの開発 1186
IDT テストスイート設定ファイルを作成する 1195
IDT ステートマシンを設定する 1203
IDT テストケース実行可能ファイルを作成する 1227
IDT コンテキストを使用する 1234
テストの実行者向けの設定の設定 1239
カスタムテストスイートのデバッグと実行 1250
IDT テストの結果とログを確認する 1253
IDT 使用状況メトリクス 1260
AWS IoT Greengrass トラブルシューティング用の IDT 1266
エラーコード 1267
AWS IoT Greengrass エラーの IDT の解決 1289
の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1 1294
トラブルシューティング 1296
AWS IoT Greengrass 主な問題 1296
エラー: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass daemon process with [pid = <pid>] died. 1298
エラー: Failed to parse /<greengrass-root>/config/config.json. 1299
エラー: Error occurred while generating TLS config: ErrUnknownURIScheme 1299
エラー: Runtime failed to start: unable to start workers: container test timed out. 1299

エラー: Failed to invoke PutLogEvents on local Cloudwatch, logGroup: /GreengrassSystem/connection_manager, error: RequestError: send request failed caused by: Post http://<path>/cloudwatch/logs/: dial tcp <address>: getsockopt: connection refused, response: { }. 1300
エラー: Unable to create server due to: failed to load group: chmod /<greengrass-root>/ggc/deployment/lambda/arn:aws:lambda:<region>:<account-id>:function:<function-name>:<version>/<file-name>: no such file or directory. 1300
AWS IoT Greengrass Core ソフトウェアは、コンテナ化なしでの実行から Greengrass コンテナでの実行に変更した後は起動しません。 1301
エラー: Spool size should be at least 262144 bytes. 1301
エラー: [ERROR]-Cloud messaging error: Error occurred while trying to publish a message. {"errorString": "operation timed out"} 1302
エラー: container_linux.go:344: starting container process caused "process_linux.go:424: container init caused \"rootfs_linux.go:64: mounting \\\"/greengrass/ggc/socket/greengrass_ipc.sock\\\" to rootfs \\\"/greengrass/ggc/packages/<version>/rootfs/merged\\\" at \\\"/greengrass_ipc.sock\\\" caused \\\"stat /greengrass/ggc/socket/greengrass_ipc.sock: permission denied\\\"\"". 1302
エラー: Greengrass daemon running with PID: <process-id>. Some system components failed to start. Check 'runtime.log' for errors. 1303
デバイスシャドウがクラウドと同期していない。 1038
エラー: unable to accept TCP connection. accept tcp [::]:8000: accept4: too many open files. 1303
エラー: Runtime execution error: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:50: preparing rootfs caused \\\"permission denied\\\"\"". 1304
警告: [WARN]-[5]GK Remote: Error retrieving public key data: ErrPrincipalNotConfigured: private key for MqttCertificate is not set. 1304
エラー: Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/core/<architecture>/greengrass-core-<distribution-version>.tar.gz. 1038
AWS IoT Greengrass コアはネットワークプロキシを使用するように設定されており、Lambda 関数は送信接続を実行できません。 1305
コアで接続と切断の無限ループが発生し、runtime.log ファイルに連続する接続と切断のエントリが含まれている。 1305

エラー: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:62: mounting \\\"proc\\\" to rootfs \\\" 1306
[ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"} 1307
[ERROR]-Deployment failed. {"deploymentId": "<deployment-id>", "errorString": "container test process with pid <pid> failed: container process state: exit status 1"} 1307
エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to create overlay fs for container: mounting overlay at /greengrass/ggc/packages/<ggc-version>/rootfs/merged failed: failed to mount with args source=\"no_source\" dest=\"/greengrass/ggc/packages/<ggc-version>/rootfs/merged\" fstype=\"overlay\" flags=\"0\" data=\"lowerdir=/greengrass/ggc/packages/<ggc-version>/dns:/,upperdir=/greengrass/ggc/packages/<ggc-version>/rootfs/upper,workdir=/greengrass/ggc/packages/<ggc-version>/rootfs/work\": too many levels of symbolic links"} 1308
エラー: [DEBUG] - ルートの取得に失敗しました。メッセージを破棄します。 1309
エラー: [Errno 24] 開いている <lambda-function> が多すぎます、[Errno 24] 開いているファイルが多すぎます 1309
エラー: ds server failed to start listening to socket: listen unix <ggc-path>/ggc/socket/greengrass_ipc.sock: bind: invalid argument 1310
[INFO] (Copier) aws.greengrass.StreamManager: stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException: Instant exceeds minimum or maximum instant 1310
GPG error: https://dnw9lb6lzp2d8.cloudfront.net stable InRelease: The following signatures were invalid: EXPKEYSIG 68D644ABD2327D47 AWS Greengrass Master Key 1311
デプロイに関する問題 1311
現在のデプロイが機能しないため、以前の正常なデプロイに戻す必要がある。 1313
デプロイに関する 403 Forbidden エラーがログに記録される。 1315
create-deployment コマンドを初めて実行すると、ConcurrentDeployment エラーが発生する。 1315
エラー: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account. 1037
エラー: デプロイのダウンロードステップを実行できません。ダウンロード中のエラー: グループ定義ファイルのダウンロード中のエラー: ... x509: 証明書の有効期限が切れているか、まだ有効ではありません 1315

デプロイが完了しない。 1316
エラー: Unable to find java or java8 executables, or the error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: worker with <worker-id> failed to initialize with reason Installed Java version must be greater than or equal to 8 1317
デプロイが完了せず、runtime.log に複数の「wait 1s for container to stop」エントリが含まれる。 1317
デプロイが完了せず、`runtime.log` に「[ERROR]-Greengrass deployment error: failed to report deployment status back to cloud {"deploymentId": "<deployment-id>", "errorString": "Failed to initiate PUT, endpoint: https://<deployment-status>, error: Put https://<deployment-status>: proxyconnect tcp: x509: certificate signed by unknown authority"}」が含まれる ... 1318
エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: <path>. 1319
エラー: <list-of-function-arns> are configured to run as root but Greengrass is not configured to run Lambda functions with root permissions. 1319
エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user: user: unknown user ggc_user. 1319
エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"} 1320
エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: process start failed: container_linux.go:259: starting container process caused "process_linux.go:250: running exec setns process for init caused \"wait: no child processes \"". 1320
エラー: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers) 1321
グループの作成と関数の作成に関する問題 1321
エラー: Your 'IsolationMode' configuration for the group is invalid. 1322
エラー: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid. ... 1322
エラー: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. 1322

エラー: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. 1323
エラー: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer. 1323
エラー: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer. 1323
エラー: Execution configuration for function with arn <function-arn> is not allowed. 1324
検出の問題 1324
エラー: デバイスがメンバーになっているグループの数が多すぎます。デバイスを 10 個を超えるグループに含めることはできません。 1324
機械学習リソースの問題 1324
InvalidMLModelOwner - GroupOwnerSetting は ML モデルリソースに提供されていますが、GroupOwner または GroupPermission がありません 431
NoContainer 関数は、機械学習リソースをアタッチするときにアクセス権限を設定できません。<function-arn> は、リソースアクセスポリシーでアクセス権限 <ro/rw> を持つ機械学習リソース <resource-id> を参照します。 431
関数 <function-arn> は機械学習リソース <resource-id> を参照しますが、ResourceAccessPolicy とリソースの OwnerSetting のどちらにもアクセス権限がありません。 431
関数 <function-arn> は \"rw\" アクセス権限で機械学習リソース <resource-id> を参照しますが、リソース所有者設定の GroupPermission で許可されているのは \"ro\" のみです。 431
NoContainer 関数 <function-arn> は、ネストされた送信先パスのリソースを参照します。 432
Lambda <function-arn> は、同じグループ所有者 ID を共有することでリソース <resource-id> にアクセスします。 432
AWS IoT Greengrass Docker のコアの問題 1327
エラー: Unknown options: -no-include-email. 389
警告: IPv4 is disabled. Networking will not work. 390
エラー: A firewall is blocking file Sharing between windows and the containers. 390
エラー: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::<account-id>:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: * 390
エラー: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use. 1328

エラー: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". 整合性を維持するには、GGC がクラッシュするため手動で再起動する必要があります。 1329
ログによるトラブルシューティング 1329
ストレージ問題のトラブルシューティング 1331
メッセージのトラブルシューティング 1331
シャドウ同期タイムアウト問題のトラブルシューティング 1332
AWS re:Post を確認する 1333
ドキュメント履歴 1334
以前の更新 1358

AWS IoT Greengrass Version 1 は 2023 年 6 月 30 日に延長ライフフェーズに参加しました。詳細については、「AWS IoT Greengrass V1 メンテナンスポリシー」を参照してください。この日以降、AWS IoT Greengrass V1 は機能、機能強化、バグ修正、またはセキュリティパッチを提供する更新をリリースしません。で実行されるデバイスは中断 AWS IoT Greengrass V1 されず、引き続き動作し、クラウドに接続します。への移行 AWS IoT Greengrass Version 2を強くお勧めします。これにより、重要な新機能が追加され、追加のプラットフォームがサポートされます。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。

とは AWS IoT Greengrass

AWS IoT Greengrass は、クラウド機能をローカルデバイスに拡張するソフトウェアです。これにより、デバイスは情報源に近いデータを収集および分析して、ローカルイベントに自動的に反応し、ローカルネットワークで互いに安全に通信することができます。ローカルデバイスは、 と安全に通信 AWS IoT Core して IoT データを にエクスポートすることもできます AWS クラウド。 AWS IoT Greengrass 開発者は、 AWS Lambda 関数と構築済みのコネクタを使用して、ローカル実行のためにデバイスにデプロイされるサーバーレスアプリケーションを作成できます。

次の図は、 の基本的なアーキテクチャを示しています AWS IoT Greengrass。

AWS IoT Greengrass を使用すると、お客様は IoT デバイスとアプリケーションロジックを構築できます。具体的には、 は、 デバイスで実行されるアプリケーションロジックのクラウドベースの管理 AWS IoT Greengrass を提供します。ローカルにデプロイした Lambda 関数とコネクタは、ローカルイベント、クラウドからのメッセージ、またはその他のソースによってトリガーされます。

では AWS IoT Greengrass、デバイスはローカルネットワーク上で安全に通信し、クラウドに接続することなく相互にメッセージを交換します。 は、接続が失われた場合にメッセージをインテリジェントにバッファリングできるローカル pub/sub メッセージマネージャー AWS IoT Greengrass を提供し、クラウドへのインバウンドメッセージとアウトバウンドメッセージを保持します。

AWS IoT Greengrass はユーザーデータを保護します。

- デバイスの安全な認証と認可を介して保護。
- ローカルネットワークの安全な接続を介して保護。

- ローカルデバイスとクラウドの間で保護。

デバイスのセキュリティ認証情報は、クラウドへの接続が中断された場合でも、失効するまでグループで機能するため、デバイスはローカルで安全に通信を続けることができます。

AWS IoT Greengrass は、Lambda 関数の安全なover-the-air更新を提供します。

AWS IoT Greengrass は以下で構成されます。

- ソフトウェアディストリビューション
  - AWS IoT Greengrass コアソフトウェア
  - AWS IoT Greengrass コア SDK
- クラウドサービス
  - AWS IoT Greengrass API
- 機能
  - Lambda ランタイム
  - シャドウの実装
  - メッセージマネージャー
  - グループ管理
  - 検出サービス
  - 無線通信経由更新エージェント
  - ストリームマネージャー
  - ローカルリソースアクセス
  - ローカル機械学習推論
  - ローカルシークレットマネージャー
  - サービス、プロトコル、およびソフトウェアとの組み込み統合を備えたコネクタ

トピック

- AWS IoT Greengrass コアソフトウェア
- AWS IoT Greengrass グループ
- のデバイス AWS IoT Greengrass
- SDK
- サポートされているプラットフォームと要件

- AWS IoT Greengrass ダウンロード
- ご意見をお待ちしております
- AWS IoT Greengrass Core ソフトウェアのインストール
- AWS IoT Greengrass コアを設定する

AWS IoT Greengrass コアソフトウェア

AWS IoT Greengrass Core ソフトウェアには、次の機能があります。

- コネクタと Lambda 関数のデプロイとローカル実行。
- への自動エクスポートを使用して、データストリームをローカルで処理します AWS クラウド。
- マネージドサブスクリプションを使用したデバイス、コネクタ、および Lambda 関数間のローカルネットワークを介した MQTT メッセージング。
- マネージドサブスクリプションを使用した と AWS IoT デバイス、コネクタ、Lambda 関数間の MQTT メッセージング。
- デバイスの認証と認可を使用したデバイスと AWS クラウド 間の安全な接続。
- デバイスのローカルシャドウ同期。シャドウは AWS クラウドと同期するように設定できます。
- ローカルデバイスとボリュームリソースへの制御されたアクセス。
- ローカル推論を実行するためにクラウドでトレーニングされた機械学習モデルのデプロイ。
- デバイスで Greengrass コアデバイスを検出するための IP アドレス自動検出。
- 新規作成または更新されたグループ設定の一元的デプロイ。設定データをダウンロードすると、コアデバイスが自動的に再起動されます。
- ユーザー定義の Lambda 関数の安全な無線通信 (OTA) によるソフトウェア更新。
- コネクタと Lambda 関数で制御される、ローカルシークレットの安全な暗号化されたストレージ。

AWS IoT Greengrass コアインスタンスは、クラウドに保存されている AWS IoT Greengrass グループ定義を作成および更新する AWS IoT Greengrass APIs を介して設定されます。

AWS IoT Greengrass コアソフトウェアバージョン

AWS IoT Greengrass には、tar.gz ダウンロードファイル、クイックスタートスクリプト、サポートされている Debian プラットフォームへの`apt`インストールなど、 AWS IoT Greengrass Core ソフトウェアをインストールするためのいくつかのオプションが用意されています。詳細については、

「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

次のタブでは、 AWS IoT Greengrass Core ソフトウェアバージョンで新しく変更される内容について説明します。

GGC v1.11

1.11.6

バグ修正と機能向上:

- デプロイ中に電源損失が突然発生した場合の復元力を改善しました。
- ストリームマネージャーのデータ破損により AWS IoT Greengrass Core ソフトウェアが起動できなくなる問題を修正しました。
- 新規クライアントデバイスが特定のシナリオでコアに接続できない問題を修正しました。
- ストリームマネージャーのストリーム名に `.log` を含めることができなかった問題を修正しました。

1.11.5

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

1.11.4

バグ修正と機能向上:

- AWS IoT Greengrass Core ソフトウェア v1.11.3 へのアップグレードを妨げるストリームマネージャーの問題を修正しました。ストリームマネージャーを使用してデータをクラウドにエクスポートする場合、OTA 更新を使用して、以前の v1.x バージョンの AWS IoT Greengrass Core ソフトウェアを v1.11.4 にアップグレードできるようになりました。
- 全般的なパフォーマンス向上とバグ修正。

1.11.3

バグ修正と機能向上:

- Ubuntu デバイスで AWS IoT Greengrass Core ソフトウェアがスナップで実行され、デバイスに突然電源が失われた後に応答しなくなる問題を修正しました。
- 存続期間の長い Lambda 関数への MQTT メッセージの配信が遅延する問題を修正しました。

- `maxWorkItemCount` 値が `1024` より大きい値に設定されていると、MQTT メッセージが適切に送信されない問題を修正しました。
- `config.json` の `keepAlive` プロパティで指定された MQTT `KeepAlive` 期間を OTA 更新エージェントが無視する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

> **Important**
>
> ストリームマネージャーを使用してデータをクラウドにエクスポートする場合は、以前の v1.x バージョンから AWS IoT Greengrass Core ソフトウェア v1.11.3 にアップグレードしないでください。ストリームマネージャーを初めて有効にする場合は、まず最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールすることを強くお勧めします。

1.11.1

バグ修正と機能向上:

- ストリームマネージャーのメモリ使用量が増加する問題を修正しました。
- Greengrass コアデバイスの電源がストリームデータの特定の有効期限 (TTL) を超えてオフになっている場合に、ストリームマネージャーがストリームのシーケンス番号を `0` にリセットする問題を修正しました。
- AWS クラウドへのデータエクスポートの再試行をストリームマネージャーが適切に停止できない問題を修正しました。

1.11.0

新機能:

- Greengrass コアのテレメトリエージェントは、ローカルテレメトリデータを収集して公開します AWS クラウド。テレメトリデータを取得してさらに処理するには、Amazon EventBridge ルールを作成して、ターゲットをサブスクライブします。詳細については、AWS IoT Greengrass 「コアデバイスからのシステムヘルステレメトリデータの収集」を参照してください。
- ローカル HTTP API は、 によって開始されたローカルワーカープロセスの現在の状態のスナップショットを返します AWS IoT Greengrass。詳細については、「ローカルヘルスチェック API を呼び出す」を参照してください。

- ストリームマネージャーは、Amazon S3 および にデータを自動的にエクスポートします AWS IoT SiteWise。

  新しいストリームマネージャーパラメータにより、既存のストリームを更新して、データのエクスポートを一時停止または再開できます。
- コア上で Python 3.8.x Lambda 関数を実行するためのサポート。
- Greengrass コア IPC ポート番号の設定に使用する、`config.json` の新しい `ggDaemonPort` プロパティ。デフォルトのポート番号は 8000 です。

  Greengrass コア IPC 認証のタイムアウト設定に使用する `config.json` の新しい `systemComponentAuthTimeout` プロパティ。デフォルトのタイムアウトは 5000 ミリ秒です。
- AWS IoT Greengrass グループあたりの AWS IoT デバイスの最大数を 200 から 2500 に増やしました。

  グループあたりのサブスクリプションの最大数を 1000 から 10000 に増やしました。

  詳細については、「AWS IoT Greengrass endpoints and quotas」を参照してください。

バグ修正と機能向上:

- Greengrass サービスプロセスのメモリ使用率を減らすことができる全体的な最適化。
- 新しいランタイム設定パラメータ (`mountAllBlockDevices`) により、Greengrass が OverlayFS の設定後、バインドマウントを使用してすべてのブロックデバイスをコンテナにマウントできるようになります。この機能により、`/usr` が `/` 階層下にない場合に、Greengrass のデプロイが失敗する問題が解決されました。
- がシンボリックリンクである場合に AWS IoT Greengrass コア障害が発生する問題を修正`/tmp`しました。
- Greengrass デプロイエージェントが機械学習の未使用モデルアーティファクトを `mlmodel_public` フォルダから削除する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

Extended life versions

1.10.5

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

1.10.4

バグ修正と機能向上:

- Ubuntu デバイスで AWS IoT Greengrass Core ソフトウェアがスナップで実行され、デバイスに突然電源が失われた後に応答しなくなる問題を修正しました。
- 存続期間の長い Lambda 関数への MQTT メッセージの配信が遅延する問題を修正しました。
- `maxWorkItemCount` 値が `1024` より大きい値に設定されていると、MQTT メッセージが適切に送信されない問題を修正しました。
- `config.json` の `keepAlive` プロパティで指定された MQTT `KeepAlive` 期間を OTA 更新エージェントが無視する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

1.10.3

バグ修正と機能向上:

- Greengrass コア IPC 認証のタイムアウト設定に使用する `config.json` の新しい `systemComponentAuthTimeout` プロパティ。デフォルトのタイムアウトは 5000 ミリ秒です。
- ストリームマネージャーのメモリ使用量が増加する問題を修正しました。

1.10.2

バグ修正と機能向上:

- config.json の新しい`mqttOperationTimeout`プロパティ。MQTT 接続でのパブリッシュ、サブスクライブ、およびサブスクライブ解除オペレーションのタイムアウトの設定に使用します AWS IoT Core。
- 全般的なパフォーマンス向上とバグ修正。

1.10.1

バグ修正と機能向上:

- ストリームマネージャーはファイルデータの破損に対する回復性が高くなっています。
- Linux カーネル 5.1 以降を使用しているデバイスで sysfs マウントエラーが発生する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

1.10.0

新機能:

- データストリームをローカルで処理し、 AWS クラウド に自動的にエクスポートするストリームマネージャー。この機能を使用するには、Greengrass Core デバイスに Java 8 が必要です。詳細については、「データストリームの管理」を参照してください。
- コアデバイスで Docker アプリケーションを実行する、新しい Greengrass Docker アプリケーションのデプロイコネクタ。詳細については、「the section called “Docker アプリケーションのデプロイ”」を参照してください。
- OPC-UA サーバーからアセットプロパティに産業用デバイスデータを送信する新しい IoT SiteWise コネクタ AWS IoT SiteWise。詳細については、「the section called “IoT SiteWise”」を参照してください。
- コンテナ化を使用せずに実行する Lambda 関数では、Greengrass グループの機械学習リソースにアクセスできます。詳細については、「the section called “機械学習リソースにアクセスする”」を参照してください。
- での MQTT 永続セッションのサポート AWS IoT。詳細については、「the section called “AWS IoT Coreを使用した MQTT 永続セッション”」を参照してください。
- ローカルの MQTT トラフィックは、デフォルトのポート 8883 以外のポートを使用できます。詳細については、「the section called “ローカルメッセージング用の MQTT ポート”」を参照してください。
- Lambda 関数から信頼性の高いメッセージを発行する、AWS IoT Greengrass Core SDK の新しい `queueFullPolicy` オプション。
- コア上で Node.js 12.x Lambda 関数を実行するためのサポート。
- ハードウェアセキュリティ統合による無線通信 (OTA) 更新は、OpenSSL 1.1 で設定できます。
- 全般的なパフォーマンス向上とバグ修正。

1.9.4

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

1.9.3

新機能:

- Armv6l. AWS IoT Greengrass Core ソフトウェア v1.9.3 以降のサポートは、Armv6l アーキテクチャの Raspbian ディストリビューション (Raspberry Pi Zero デバイスなど) にインストールできます。
- ALPN を使用したポート 443 での OTA 更新。MQTT トラフィックにポート 443 を使用する Greengrass コアはover-the-air (OTA) ソフトウェア更新をサポートするようになりました。 は、Application Layer Protocol Network (ALPN) TLS 拡張機能 AWS IoT Greengrass を使用して、これらの接続を有効にします。詳細については、 AWS IoT Greengrass Core ソフトウェアの OTA 更新およびthe section called “ポート 443 での接続またはネットワークプロキシを通じた接続”を参照してください。

バグ修正と機能向上:

- Python 2.7 Lambda 関数がバイナリペイロードを他の Lambda 関数に送信できない、v1.9.0 で発生したバグを修正。
- 全般的なパフォーマンス向上とバグ修正。

1.9.2

新機能:

- OpenWrt. AWS IoT Greengrass Core ソフトウェア v1.9.2 以降のサポートは、Armv8 (AArch64) および Armv7l アーキテクチャの OpenWrt ディストリビューションにインストールできます。現在、OpenWrt は ML 推論をサポートしていません。

1.9.1

バグ修正と機能向上:

- トピックにワイルドカード文字が含まれている `cloud` からのメッセージが削除される、v1.9.0 で発生したバグを修正しました。

1.9.0

新機能:

- Python 3.7 と Node.js 8.10 の Lambda ランタイムのサポート。Python 3.7 および Node.js 8.10 ランタイムを使用する Lambda 関数を AWS IoT Greengrass コアで実行できるようになりました (AWS IoT Greengrass 引き続き Python 2.7 および Node.js 6.10 ランタイムをサポートします) 。
- 最適化された MQTT 接続。Greengrass コアが確立する AWS IoT Coreとの接続数が少なくなりました。この変更により、接続の数に基づく料金の運用コストを削減できます。
- ローカル MQTT サーバー用楕円曲線 (EC) キー。ローカル MQTT サーバーは、RSA キーに加えて EC キーをサポートします。(MQTT サーバー証明書には、キータイプに関係な

く、SHA-256 RSA 署名が含まれています。) 詳細については、「the section called “セキュリティプリンシパル”」を参照してください。

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

1.8.4

シャドウ同期とデバイス証明書マネージャーの再接続の問題を修正しました。

全般的なパフォーマンス向上とバグ修正。

1.8.3

一般的なパフォーマンス向上とバグ修正。

1.8.2

一般的なパフォーマンス向上とバグ修正。

1.8.1

一般的なパフォーマンス向上とバグ修正。

1.8.0

新機能:

- グループ内にある Lambda 関数の設定可能なデフォルトアクセス ID。このグループレベルの設定により、Lambda 関数の実行で使用されるデフォルトのアクセス許可が決まります。ユーザー ID、グループ ID、またはその両方を設定できます。個々の Lambda 関数は、そのグループのデフォルトのアクセス ID を上書きできます。詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。
- ポート 443 経由の HTTPS トラフィック。HTTPS コミュニケーションは、デフォルトのポート 8443 ではなくポート 443 を経由するように設定できます。これにより、Application Layer Protocol Network (ALPN) TLS 拡張機能 AWS IoT Greengrass のサポートが補完され、MQTT と HTTPS の両方のすべての Greengrass メッセージングトラフィックがポート 443 を使用できるようになります。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。
- AWS IoT 接続のクライアント IDs予測できます。この変更により、 AWS IoT Device Defender および AWS IoT ライフサイクルイベントのサポートが有効になり、そのため接

続、切断、購読、および購読解除のイベントに関する通知を受け取ることができます。予測可能な命名により、接続 ID を中心としたロジックの作成も容易になります (例えば、証明書の属性に基づいてサブスクライブポリシーテンプレートを作成します。詳細については、「the section called “AWS IoTを使用した MQTT 接続用クライアント ID”」を参照してください。

バグ修正と機能向上:

- シャドウ同期とデバイス証明書マネージャーの再接続の問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

1.7.1

新機能:

- Greengrass コネクタは、ローカルインフラストラクチャ、デバイスプロトコル AWS、およびその他のクラウドサービスとの統合を組み込みます。詳細については、「コネクタを使用してサービスおよびプロトコルと統合する」を参照してください。
- AWS IoT Greengrass AWS Secrets Manager はコアデバイスに拡張され、パスワード、トークン、その他のシークレットをコネクタや Lambda 関数で使用できます。シークレットは転送中および保管時に暗号化されます。詳細については、「Core にシークレットをデプロイする」を参照してください。
- 信頼セキュリティオプションのハードウェアルートのサポート。詳細については、「the section called “ハードウェアセキュリティ統合”」を参照してください。
- 分離およびアクセス許可の設定で、Lambda 関数が Greengrass コンテナなしで実行され、指定されたユーザーとグループのアクセス許可を使用するように指定可能。詳細については、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。
- Greengrass グループをコンテナ化なしで実行するように設定することで、Docker コンテナ (Windows、macOS、または Linux) AWS IoT Greengrass で を実行できます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass で を実行する”」を参照してください。
- Application Layer Protocol Negotiation (ALPN) またはネットワークを介した接続によるポート 443 での MQTT メッセージング。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。
- SageMaker AI Neo 深層学習ランタイム。SageMaker AI Neo 深層学習コンパイラによって最適化された機械学習モデルをサポートします。Neo 深層学習ランタイムの詳細については、「the section called “ML 推論用のランタイムとライブラリ”」を参照してください。
- Raspberry Pi Core デバイスで Raspbian Stretch (2018-06-27) をサポート。

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

さらに、このリリースでは以下の機能も使用できます。

- AWS IoT Device Tester for は AWS IoT Greengrass、CPU アーキテクチャ、カーネル設定、ドライバーが動作することを検証するために使用できます AWS IoT Greengrass。詳細については、「AWS IoT Device Tester for AWS IoT Greengrass V1 の使用」を参照してください。
- AWS IoT Greengrass Core ソフトウェア、 AWS IoT Greengrass Core SDK、 AWS IoT Greengrass Machine Learning SDK パッケージは、Amazon CloudFront からダウンロードできます。詳細については、「the section called “AWS IoT Greengrass ダウンロード”」を参照してください。

1.6.1

新機能:

- Greengrass コアでバイナリ–コードを実行する Lambda 実行可能ファイル。新しい AWS IoT Greengrass Core SDK for C を使用して、C および C++ で Lambda 実行可能ファイルを書き込みます。詳細については、「the section called “Lambda 実行可能ファイル”」を参照してください。
- 再起動しても維持される、オプションのローカルストレージメッセージキャッシュ。処理のためにキュー状態にある MQTT メッセージのストレージ設定を構成できます。詳細については、「the section called “MQTT メッセージキュー”」を参照してください。
- コアデバイスが切断された場合のための設定可能な再接続の最大再試行。詳細については、「`mqttMaxConnectionRetryInterval`」の the section called “AWS IoT Greengrass コア設定ファイル” プロパティを参照してください。
- ホストの /proc ディレクトリへのローカルリソースアクセス。詳細については、「ローカルリソースへのアクセス」を参照してください。
- 設定可能な書き込みディレクトリ。 AWS IoT Greengrass Core ソフトウェアは、読み取り専用および読み取り/書き込みの場所にデプロイできます。詳細については、「the section called “書き込みディレクトリの設定”」を参照してください。

バグ修正と機能向上:

- Greengrass コア内およびデバイスとコア間のメッセージ発行のパフォーマンスの改善。
- ユーザー定義の Lambda 関数によって生成されたログを処理するために必要なコンピューティングリソースの数を減らしました。

1.5.0

新機能:

- AWS IoT Greengrass Machine Learning (ML) 推論は一般利用可能です。クラウドで構築されトレーニングされたモデルを使用して、 AWS IoT Greengrass デバイスで ML Inference をローカルで実行できます。詳細については、「機械学習の推論を実行する」を参照してください。
- Greengrass Lambda 関数は、JSON に加えてバイナリデータも入力ペイロードとしてサポートするようになりました。この機能を使用するには、 AWS IoT Greengrass Core SDK バージョン 1.1.0 にアップグレードする必要があります。これは Core AWS IoT Greengrass SDK ダウンロードページからダウンロードできます。

バグ修正と機能向上:

- 総メモリ使用量の削減。
- クラウドへのメッセージの送信パフォーマンスの向上。
- ダウンロードエージェント、Device Certificate Manager、OTA 更新エージェントのパフォーマンスと安定性の向上。
- 軽微なバグを修正。

1.3.0

新機能:

- クラウドにデプロイした Greengrass 更新ジョブを処理できる OTA (無線通信経由) 更新エージェント。このエージェントは新しい `/greengrass/ota` ディレクトリにあります。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。
- ローカルリソースアクセス機能により、Greengrass Lambda 関数は周辺デバイスやボリュームなどのローカルリソースにアクセスできます。詳細については、「Lambda 関数とコネクタを使用してローカルリソースにアクセスする」を参照してください。

1.1.0

新機能:

- デプロイされた AWS IoT Greengrass グループは、Lambda 関数、サブスクリプション、および設定を削除することでリセットできます。詳細については、「the section called “デプロイのリセット”」を参照してください。
- Python 2.7 に加えて、Node.js 6.10 と Java 8 の Lambda ランタイムをサポートしました。

以前のバージョンの AWS IoT Greengrass コアから移行するには：

- 証明書を `/greengrass/configuration/certs` フォルダから `/greengrass/certs` にコピーします。
- `/greengrass/configuration/config.json` を `/greengrass/config/config.json` にコピーします。
- `/greengrass/greengrassd` の代わりに `/greengrass/ggc/core/greengrassd` を実行します。
- 新規コアにグループをデプロイします。

1.0.0

当初のバージョン

AWS IoT Greengrass グループ

Greengrass グループは、Greengrass コア、デバイス、サブスクリプションなどのコンポーネントと設定のコレクションです。グループは、操作の範囲を定義するために使用されます。例えば、グループは建物の 1 つの階、1 台のトラック、または採掘現場全体を表します。次の図は、Greengrass グループを構成するために使用できるコンポーネントを示しています。

前の図の各オブジェクトについて説明します。

A: Greengrass グループの定義

グループの設定とコンポーネントに関する情報。

B: Greengrass グループの設定

具体的には次のとおりです。

- Greengrass グループのロール。
- 認証機関とローカル接続の設定。
- Greengrass コアの接続情報。
- デフォルトの Lambda ランタイム環境。詳細については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。
- CloudWatch とローカルログ設定。詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

C: Greengrass コア

Greengrass コアを表す AWS IoT モノ (デバイス) 。詳細については、「the section called “ AWS IoT Greengrass コアを設定する”」を参照してください。

D: Lambda 関数の定義

Core でローカルに実行されて設定データが関連付けられている Lambda 関数のリスト。詳細については、「ローカル Lambda 関数を実行する」を参照してください。

E: サブスクリプションの定義。

MQTT メッセージを使用して通信を可能にするサブスクリプションのリスト。サブスクリプションは以下を定義します。

- メッセージの送信元とメッセージターゲット。これらは、クライアントデバイス、Lambda 関数、コネクタ AWS IoT Core、ローカルシャドウサービスです。
- メッセージのフィルタリングに使用するトピックまたは件名。

詳細については、「the section called “MQTT メッセージングワークフローにおけるマネージドサブスクリプション”」を参照してください。

F: コネクタの定義

コアでローカルに実行され、関連する設定データを含むコネクタのリスト。詳細については、「コネクタを使用してサービスおよびプロトコルと統合する」を参照してください。

G: デバイスの定義

Greengrass グループのメンバーである AWS IoT モノ (クライアントデバイスまたはデバイスと呼ばれる) と、関連する設定データのリスト。詳細については、「the section called “のデバイス AWS IoT Greengrass”」を参照してください。

H: リソースの定義

Greengrass コアのローカルリソース、機械学習リソース、シークレットリソース、および関連する設定データのリスト。詳細については ローカルリソースへのアクセス、機械学習の推論を実行する、および Core にシークレットをデプロイする を参照してください。

デプロイすると、Greengrass グループの定義、Lambda 関数、コネクタ、リソース、およびサブスクリプションテーブルがコアデバイスにコピーされます。詳細については、「 AWS IoT Greengrass グループのデプロイ」を参照してください。

のデバイス AWS IoT Greengrass

Greengrass グループには、次の 2 種類の AWS IoT デバイスを含めることができます。

Greengrass コア

Greengrass コアは、 AWS IoT Greengrass Core ソフトウェアを実行するデバイスであり、AWS IoT Core および AWS IoT Greengrass サービスと直接通信できます。コアには、認証に使用される独自のデバイス証明書があります AWS IoT Core。 AWS IoT Core レジストリにデバイスシャドウとエントリがあります。Greengrass コアは、ローカル Lambda ランタイム、デプロイエージェント、および IP アドレストラッカーを実行して IP アドレス情報を AWS IoT Greengrass サービスに送信し、クライアントデバイスがグループとコアの接続情報を自動的に検出できるようにします。詳細については、「the section called “ AWS IoT Greengrass コアを設定する”」を参照してください。

> **Note**
>
> Greengrass グループに含められるコアは 1 つのみです。

クライアントデバイス

クライアントデバイス (接続されたデバイス、Greengrass デバイス、またはデバイスとも呼ばれる) は、MQTT を介して Greengrass コアに接続するデバイスのことです。 AWS IoT Core 認

証用の独自のデバイス証明書、デバイスシャドウ、 AWS IoT Core レジストリのエントリがあります。クライアントデバイスは、FreeRTOS を実行、もしくは AWS IoT Device SDK または AWS IoT Greengrass Discovery API を使用して、同じ Greengrass グループ内のコアとの接続と認証に使用される検出情報を取得できます。 AWS IoT コンソールを使用して のクライアントデバイスを作成および設定する方法については AWS IoT Greengrass、「」を参照してくださいthe section called “モジュール 4: AWS IoT Greengrass グループ内のクライアントデバイスを操作する”。または、 を使用してクライアントデバイス AWS CLI を作成および設定する方法を示す例については AWS IoT Greengrass、AWS CLI 「 コマンドリファレンス」のcreate-device-definition」を参照してください。

Greengrass グループでは、クライアントデバイスがグループ内の Lambda 関数、コネクタ、その他のクライアントデバイス、 AWS IoT Core またはローカルシャドウサービスと MQTT 経由で通信できるようにするサブスクリプションを作成できます。MQTT メッセージはコアを通じてルーティングされます。コアデバイスがクラウドから切断された場合、クライアントデバイスはローカルネットワークを介して通信を続けることができます。クライアントデバイスのサイズは、マイクロコントローラーベースの小型デバイスから大型アプライアンスまでさまざまです。現在、Greengrass グループには最大 2,500 台のクライアントデバイスを含めることができます。クライアントデバイスは、最大 10 個のグループのメンバーにすることができます。

> **Note**
>
> OPC-UA は、産業通信用の情報交換標準です。Greengrass コアで OPC-UA のサポートを実装するには、IoT SiteWise コネクタを使用できます。コネクタは、OPC-UA サーバーから のアセットプロパティに産業用デバイスデータを送信します AWS IoT SiteWise。

これらのデバイスタイプ間の関係を次の表に示します。

	Core	Device
Certificate	✓	✓
IoT Policy	✓	✓
IoT Thing	✓	✓
Device use	Gateway	Sensor and/or Actuator
Software	AWS IoT Greengrass Core Software	Amazon FreeRTOS / AWS IoT Device SDK
Group membership	✓	✓
Functions outside a Greengrass Group	✗	✓

AWS IoT Greengrass コアデバイスは証明書を 2 つの場所に保存します。

- `/greengrass-root/certs` のコアデバイス証明書。通常、コアデバイス証明書の名前は `hash.cert.pem` です (例えば、`86c84488a5.cert.pem`)。この証明書は、コアが および AWS IoT Greengrass サービスに接続する AWS IoT Core ときに、相互認証のために AWS IoT クライアントによって使用されます。
- `/greengrass-root/ggc/var/state/server` の MQTT サーバー証明書。MQTT サーバー証明書の名前は `server.crt` です。この証明書は、ローカル MQTT サーバー (Greengrass コア上) と Greengrass デバイスとの間で相互認証に使用されます。

> **Note**
>
> *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

SDK

以下の AWSが提供する SDKs は、 の操作に使用されます AWS IoT Greengrass。

AWS SDK

AWS SDK を使用して、Amazon S3、Amazon DynamoDB AWS IoT AWS IoT Greengrassなど、あらゆる AWS サービスとやり取りするアプリケーションを構築します。のコンテキストでは AWS IoT Greengrass、デプロイされた Lambda 関数で AWS SDK を使用して、任意の AWS サービスを直接呼び出すことができます。詳細については、「AWS SDK」を参照してください。

> **Note**
>
> AWS SDKs で使用できる Greengrass に固有のオペレーションは、 AWS IoT Greengrass API および でも使用できますAWS CLI。

AWS IoT デバイス SDK

AWS IoT Device SDK は、デバイスが AWS IoT Core および に接続するのに役立ちます AWS IoT Greengrass。詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT デバイス SDK」を参照してください。

クライアントデバイスは、任意の AWS IoT Device SDK v2 プラットフォームを使用して、Greengrass コアの接続情報を検出できます。接続情報には、以下が含まれます。

- クライアントデバイスが属する Greengrass グループの ID。
- 各グループに属する Greengrass コアの IP アドレス。これらは、コアエンドポイントとも呼ばれます。
- デバイスがコアとの相互認証に使用するグループ CA 証明書。詳細については、「the section called “デバイス接続のワークフロー”」を参照してください。

> **Note**
>
> AWS IoT Device SDKs の v1 では、C++ および Python プラットフォームのみが組み込みの検出サポートを提供します。

AWS IoT Greengrass コア SDK

AWS IoT Greengrass Core SDK を使用すると、Lambda 関数は Greengrass コアとやり取りし、メッセージをパブリッシュし AWS IoT、ローカルシャドウサービスとやり取りし、デプロイされた他の Lambda 関数を呼び出し、シークレットリソースにアクセスできます。この SDK は、AWS IoT Greengrass コアで実行される Lambda 関数で使用します。詳細については、「AWS IoT Greengrass コア SDK」を参照してください。

AWS IoT Greengrass Machine Learning SDK

AWS IoT Greengrass Machine Learning SDK を使用すると、Lambda 関数は Greengrass コアにデプロイされた機械学習モデルを機械学習リソースとして使用できます。この SDK は、 AWS IoT Greengrass コアで実行され、ローカル推論サービスとやり取りする Lambda 関数で使用されます。詳細については、「AWS IoT Greengrass Machine Learning SDK」を参照してください。

サポートされているプラットフォームと要件

次のタブには、 AWS IoT Greengrass Core ソフトウェアでサポートされているプラットフォームと要件が一覧表示されます。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアのダウンロードから AWS IoT Greengrass Core ソフトウェアをダウンロードできます。

GGC v1.11

サポートされているプラットフォーム:

- アーキテクチャ: Armv7l
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv8 (AArch64)
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv6l

  - OS: Linux
- アーキテクチャ: x86_64
  - OS: Linux
- Windows、macOS、Linux プラットフォームは、Docker コンテナ AWS IoT Greengrass で実行できます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass でを実行する”」を参照してください。

要件:

- AWS IoT Greengrass Core ソフトウェアで使用できるディスク容量は 128 MB 以上です。OTA 更新エージェントを使用する場合は、400 MB 以上が必要です。
- AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 128 MB の RAM。ストリームマネージャーを有効にする場合は、198 MB 以上の RAM が必要です。

> **Note**
>
> AWS IoT コンソールのデフォルトグループ作成オプションを使用して Greengrass グループを作成する場合、ストリームマネージャーはデフォルトで有効になっています。

- Linux カーネルバージョン:
  - コンテナ AWS IoT Greengrass での実行をサポートするには、Linux カーネルバージョン 4.4 以降が必要です。
  - コンテナ AWS IoT Greengrass なしでの実行をサポートするには、Linux カーネルバージョン 3.17 以降が必要です。この設定では、Greengrass グループのデフォルトの Lambda 関数コンテナ化を [No container] (コンテナなし) に設定する必要があります。手順については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。
- GNU C ライブラリ (glibc) バージョン 2.14 以降。OpenWrt ディストリビューションでは、musl C ライブラリ バージョン 1.1.16 以降が必要です。
- `/var/run` ディレクトリがデバイスに存在する必要があります。
- `/dev/stdin`、`/dev/stdout`、および `/dev/stderr` ファイルが利用可能である必要があります。
- ハードリンクとソフトリンクの保護がデバイス上で有効になっている必要があります。それ以外の場合は、 `-i`フラグを使用して安全でないモードでのみ実行 AWS IoT Greengrass できます。

- 次の Linux カーネル設定をデバイスで有効にする必要があります。
  - 名前空間:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

    カーネルは cgroup をサポートしている必要があります。コンテナ AWS IoT Greengrass でを実行する場合、次の要件が適用されます。
    - AWS IoT Greengrass が Lambda 関数のメモリ制限を設定できるように、メモリ cgroup を有効にしてマウントする必要があります。
    - ローカルリソースアクセスを持つ Lambda 関数を使用して AWS IoT Greengrass コアデバイスでファイルを開く場合は、デバイス cgroup を有効にしてマウントする必要があります。
  - その他:
    - CONFIG_POSIX_MQUEUE
    - CONFIG_OVERLAY_FS
    - CONFIG_HAVE_ARCH_SECCOMP_FILTER
    - CONFIG_SECCOMP_FILTER
    - CONFIG_KEYS
    - CONFIG_SECCOMP
    - CONFIG_SHMEM
- Amazon S3 および のルート証明書は、システムトラストストアに存在する AWS IoT 必要があります。
- ストリームマネージャーには、基本 AWS IoT Greengrass Core ソフトウェアメモリ要件に加えて、Java 8 ランタイムと 70 MB 以上の RAM が必要です。 AWS IoT コンソールでデフォルトグループ作成オプションを使用すると、ストリームマネージャーはデフォルトで有効になります。ストリームマネージャーは OpenWrt ディストリビューションではサポートされていません。

- ローカルで実行する Lambda 関数に必要な AWS Lambda ランタイムをサポートするライブラリ。必須ライブラリは、コアにインストールし、PATH 環境変数に追加する必要があります。同じコアに複数のライブラリをインストールできます。
  - Python 3.8 ランタイムを使用する関数に対応する Python バージョン 3.8。
  - Python 3.7 ランタイムを使用する関数に対応する Python バージョン 3.7。
  - Python 2.7 ランタイムを使用する関数に対応する Python バージョン 2.7。
  - Node.js 12.x ランタイムを使用する関数に対応する Node.js バージョン 12.x。
  - Java 8 ランタイムを使用する関数に対応する Java バージョン 8 以降。

  > **Note**
  >
  > OpenWrt ディストリビューションで Java を実行することは公式にはサポートされていません。ただし、OpenWrt ビルドで Java がサポートされている場合は、Java で作成された Lambda 関数を OpenWrt デバイス上で実行できる可能性があります。

  Lambda ランタイム AWS IoT Greengrass のサポートの詳細については、「」を参照してくださいローカル Lambda 関数を実行する。
- 次のシェルコマンド (BusyBox バリアントではない) は、無線 (OTA) 更新エージェントで必要です。
  - `wget`
  - `realpath`
  - `tar`
  - `readlink`
  - `basename`
  - `dirname`
  - `pidof`
  - `df`
  - `grep`
  - `umount`
  - `mv`
  - `gzip`
  - `mkdir`

- `rm`
- `ln`
- `cut`
- `cat`
- `/bin/bash`

GGC v1.10

サポートされているプラットフォーム:

- アーキテクチャ: Armv7l
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv8 (AArch64)
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv6l
  - OS: Linux
- アーキテクチャ: x86_64
  - OS: Linux
- Windows、macOS、Linux プラットフォームは、Docker コンテナ AWS IoT Greengrass で実行できます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass でを実行する”」を参照してください。

要件:

- AWS IoT Greengrass Core ソフトウェアで使用できるディスク容量は 128 MB 以上です。OTA 更新エージェントを使用する場合は、400 MB 以上が必要です。
- AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 128 MB の RAM。ストリームマネージャーを有効にする場合は、198 MB 以上の RAM が必要です。

> **Note**
>
> AWS IoT コンソールのデフォルトグループ作成オプションを使用して Greengrass グループを作成する場合、ストリームマネージャーはデフォルトで有効になっています。

- Linux カーネルバージョン:
  - コンテナ AWS IoT Greengrass での実行をサポートするには、Linux カーネルバージョン 4.4 以降が必要です。
  - コンテナ AWS IoT Greengrass なしでの実行をサポートするには、Linux カーネルバージョン 3.17 以降が必要です。この設定では、Greengrass グループのデフォルトの Lambda 関数コンテナ化を [No container] (コンテナなし) に設定する必要があります。手順については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。
- GNU C ライブラリ (glibc) バージョン 2.14 以降。OpenWrt ディストリビューションでは、musl C ライブラリ バージョン 1.1.16 以降が必要です。
- `/var/run` ディレクトリがデバイスに存在する必要があります。
- `/dev/stdin`、`/dev/stdout`、および `/dev/stderr` ファイルが利用可能である必要があります。
- ハードリンクとソフトリンクの保護がデバイス上で有効になっている必要があります。それ以外の場合は、 `-i`フラグを使用して安全でないモードでのみ実行 AWS IoT Greengrass できます。
- 次の Linux カーネル設定をデバイスで有効にする必要があります。
  - 名前空間:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

カーネルは cgroup をサポートしている必要があります。コンテナ AWS IoT Greengrass でを実行する場合、次の要件が適用されます。

- AWS IoT Greengrass が Lambda 関数のメモリ制限を設定できるように、メモリ cgroup を有効にしてマウントする必要があります。
- ローカルリソースアクセスを持つ Lambda 関数を使用して AWS IoT Greengrass コアデバイスでファイルを開く場合は、デバイス cgroup を有効にしてマウントする必要があります。

- その他:
  - CONFIG_POSIX_MQUEUE
  - CONFIG_OVERLAY_FS
  - CONFIG_HAVE_ARCH_SECCOMP_FILTER
  - CONFIG_SECCOMP_FILTER
  - CONFIG_KEYS
  - CONFIG_SECCOMP
  - CONFIG_SHMEM

- Amazon S3 および のルート証明書は、システムトラストストアに存在する AWS IoT 必要があります。
- ストリームマネージャーには、基本 AWS IoT Greengrass Core ソフトウェアメモリ要件に加えて、Java 8 ランタイムと 70 MB 以上の RAM が必要です。 AWS IoT コンソールでデフォルトグループ作成オプションを使用すると、ストリームマネージャーはデフォルトで有効になります。ストリームマネージャーは OpenWrt ディストリビューションではサポートされていません。
- ローカルで実行する Lambda 関数に必要な AWS Lambda ランタイムをサポートするライブラリ。必須ライブラリは、コアにインストールし、PATH 環境変数に追加する必要があります。同じコアに複数のライブラリをインストールできます。
  - Python 3.7 ランタイムを使用する関数に対応する Python バージョン 3.7。
  - Python 2.7 ランタイムを使用する関数に対応する Python バージョン 2.7。
  - Node.js 12.x ランタイムを使用する関数に対応する Node.js バージョン 12.x。
  - Java 8 ランタイムを使用する関数に対応する Java バージョン 8 以降。

> **Note**
>
> OpenWrt ディストリビューションで Java を実行することは公式にはサポートされていません。ただし、OpenWrt ビルドで Java がサポートされている場合は、Java で作成された Lambda 関数を OpenWrt デバイス上で実行できる可能性があります。

Lambda ランタイム AWS IoT Greengrass のサポートの詳細については、「」を参照してくださいローカル Lambda 関数を実行する。

- 次のシェルコマンド (BusyBox バリアントではない) は、無線 (OTA) 更新エージェントで必要です。
  - `wget`
  - `realpath`
  - `tar`
  - `readlink`
  - `basename`
  - `dirname`
  - `pidof`
  - `df`
  - `grep`
  - `umount`
  - `mv`
  - `gzip`
  - `mkdir`
  - `rm`
  - `ln`
  - `cut`
  - `cat`
  - `/bin/bash`

GGC v1.9

サポートされているプラットフォーム:

- アーキテクチャ: Armv7l
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv8 (AArch64)
  - OS: Linux
  - OS: Linux (OpenWrt)
- アーキテクチャ: Armv6l
  - OS: Linux
- アーキテクチャ: x86_64
  - OS: Linux
- Windows、macOS、Linux プラットフォームは、Docker コンテナ AWS IoT Greengrass で実行できます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass でを実行する”」を参照してください。

要件:

- AWS IoT Greengrass Core ソフトウェアで使用できるディスク容量は 128 MB 以上です。OTA 更新エージェントを使用する場合は、400 MB 以上が必要です。
- AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 128 MB の RAM。
- Linux カーネルバージョン:
  - コンテナ AWS IoT Greengrass での実行をサポートするには、Linux カーネルバージョン 4.4 以降が必要です。
  - コンテナ AWS IoT Greengrass なしでの実行をサポートするには、Linux カーネルバージョン 3.17 以降が必要です。この設定では、Greengrass グループのデフォルトの Lambda 関数コンテナ化を [No container] (コンテナなし) に設定する必要があります。手順については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。
- GNU C ライブラリ (glibc) バージョン 2.14 以降。OpenWrt ディストリビューションでは、musl C ライブラリ バージョン 1.1.16 以降が必要です。
- `/var/run` ディレクトリがデバイスに存在する必要があります。

- `/dev/stdin`、`/dev/stdout`、および `/dev/stderr` ファイルが利用可能である必要があります。
- ハードリンクとソフトリンクの保護がデバイス上で有効になっている必要があります。それ以外の場合は、 `-i`フラグを使用して安全でないモードでのみ実行 AWS IoT Greengrass できます。
- 次の Linux カーネル設定をデバイスで有効にする必要があります。
  - 名前空間:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

    カーネルは cgroup をサポートしている必要があります。コンテナ AWS IoT Greengrass でを実行する場合、次の要件が適用されます。
    - AWS IoT Greengrass が Lambda 関数のメモリ制限を設定できるように、メモリ cgroup を有効にしてマウントする必要があります。
    - ローカルリソースアクセスを持つ Lambda 関数を使用して AWS IoT Greengrass コアデバイスでファイルを開く場合は、デバイス cgroup を有効にしてマウントする必要があります。
  - その他:
    - CONFIG_POSIX_MQUEUE
    - CONFIG_OVERLAY_FS
    - CONFIG_HAVE_ARCH_SECCOMP_FILTER
    - CONFIG_SECCOMP_FILTER
    - CONFIG_KEYS
    - CONFIG_SECCOMP
    - CONFIG_SHMEM

- Amazon S3 および のルート証明書は、システムトラストストアに存在する AWS IoT 必要があります。
- ローカルで実行する Lambda 関数に必要な AWS Lambda ランタイムをサポートするライブラリ。必須ライブラリは、コアにインストールし、PATH 環境変数に追加する必要があります。同じコアに複数のライブラリをインストールできます。
  - Python 2.7 ランタイムを使用する関数に対応する Python バージョン 2.7。
  - Python 3.7 ランタイムを使用する関数に対応する Python バージョン 3.7。
  - Node.js 6.10 ランタイムを使用する関数に対応する Node.js バージョン 6.10 以降。
  - Node.js 8.10 ランタイムを使用する関数に対応する Node.js バージョン 8.10 以降。
  - Java 8 ランタイムを使用する関数に対応する Java バージョン 8 以降。

    > **Note**
    >
    > OpenWrt ディストリビューションで Java を実行することは公式にはサポートされていません。ただし、OpenWrt ビルドで Java がサポートされている場合は、Java で作成された Lambda 関数を OpenWrt デバイス上で実行できる可能性があります。

  Lambda ランタイム AWS IoT Greengrass のサポートの詳細については、「」を参照してくださいローカル Lambda 関数を実行する。
- 次のシェルコマンド (BusyBox バリアントではない) は、無線 (OTA) 更新エージェントで必要です。
  - `wget`
  - `realpath`
  - `tar`
  - `readlink`
  - `basename`
  - `dirname`
  - `pidof`
  - `df`
  - `grep`
  - `umount`
  - `mv`

- `gzip`
- `mkdir`
- `rm`
- `ln`
- `cut`
- `cat`

GGC v1.8

- サポートされているプラットフォーム:
  - アーキテクチャ: Armv7l、OS: Linux
  - アーキテクチャ: x86_64、OS: Linux
  - アーキテクチャ: Armv8 (AArch64)、OS: Linux
  - Windows、macOS、Linux プラットフォームは、Docker コンテナ AWS IoT Greengrass で実行できます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass で を実行する”」を参照してください。
  - Linux プラットフォームは、Snapcraft から利用可能な Greengrass スナップを使用して、機能 AWS IoT Greengrass が制限された のバージョンを実行できます。詳細については、「the section called “AWS IoT Greengrass スナップソフトウェア”」を参照してください。
- 以下のアイテムは必須です。
  - AWS IoT Greengrass Core ソフトウェアで使用できるディスク容量は 128 MB 以上です。OTA 更新エージェントを使用する場合は、400 MB 以上が必要です。
  - AWS IoT Greengrass Core ソフトウェアに割り当てられる最小 128 MB の RAM。
  - Linux カーネルバージョン:
    - コンテナ AWS IoT Greengrass での実行をサポートするには、Linux カーネルバージョン 4.4 以降が必要です。
    - コンテナ AWS IoT Greengrass なしでの実行をサポートするには、Linux カーネルバージョン 3.17 以降が必要です。この設定では、Greengrass グループのデフォルトの Lambda 関数コンテナ化を [No container] (コンテナなし) に設定する必要があります。手順については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。
  - GNU C ライブラリ (glibc) バージョン 2.14 以降。
  - `/var/run` ディレクトリがデバイスに存在する必要があります。

- `/dev/stdin`、`/dev/stdout`、および `/dev/stderr` ファイルが利用可能である必要があります。
- ハードリンクとソフトリンクの保護がデバイス上で有効になっている必要があります。それ以外の場合は、 `-i`フラグを使用して安全でないモードでのみ実行 AWS IoT Greengrass できます。
- 次の Linux カーネル設定をデバイスで有効にする必要があります。
  - 名前空間:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

    カーネルは cgroup をサポートしている必要があります。コンテナ AWS IoT Greengrass で を実行する場合、次の要件が適用されます。
    - AWS IoT Greengrass が Lambda 関数のメモリ制限を設定できるように、メモリ cgroup を有効にしてマウントする必要があります。
    - ローカルリソースアクセスを持つ Lambda 関数を使用して AWS IoT Greengrass コアデバイスでファイルを開く場合は、デバイス cgroup を有効にしてマウントする必要があります。
  - その他:
    - CONFIG_POSIX_MQUEUE
    - CONFIG_OVERLAY_FS
    - CONFIG_HAVE_ARCH_SECCOMP_FILTER
    - CONFIG_SECCOMP_FILTER
    - CONFIG_KEYS
    - CONFIG_SECCOMP
    - CONFIG_SHMEM

- Amazon S3 および のルート証明書は、システムトラストストアに存在する AWS IoT 必要があります。
- 次の項目は、条件付きで必須です。
  - ローカルで実行する Lambda 関数に必要な AWS Lambda ランタイムをサポートするライブラリ。必須ライブラリは、コアにインストールし、PATH 環境変数に追加する必要があります。同じコアに複数のライブラリをインストールできます。
    - Python 2.7 ランタイムを使用する関数に対応する Python バージョン 2.7。
    - Node.js 6.10 ランタイムを使用する関数に対応する Node.js バージョン 6.10 以降。
    - Java 8 ランタイムを使用する関数に対応する Java バージョン 8 以降。
  - 次のシェルコマンド (BusyBox バリアントではない) は、無線 (OTA) 更新エージェントで必要です。
    - `wget`
    - `realpath`
    - `tar`
    - `readlink`
    - `basename`
    - `dirname`
    - `pidof`
    - `df`
    - `grep`
    - `umount`
    - `mv`
    - `gzip`
    - `mkdir`
    - `rm`
    - `ln`
    - `cut`
    - `cat`

AWS IoT Greengrass クォータ (制限) の詳細については、『』のService Quotas」を参照してください。Amazon Web Services 全般のリファレンス。

料金に関する詳細は、「AWS IoT Greengrass 料金表」および「AWS IoT Core 料金表」を参照してください。

AWS IoT Greengrass ダウンロード

AWS IoT Greengrassで使用するソフトウェアを見つけてダウンロードするには、以下の情報を使用できます。

トピック

- AWS IoT Greengrass コアソフトウェア
- AWS IoT Greengrass スナップソフトウェア
- AWS IoT Greengrass Docker ソフトウェア
- AWS IoT Greengrass コア SDK
- サポートされている機械学習ランタイムおよびライブラリ
- AWS IoT Greengrass ML SDK ソフトウェア

AWS IoT Greengrass コアソフトウェア

AWS IoT Greengrass Core ソフトウェアは AWS 機能を AWS IoT Greengrass コアデバイスに拡張し、ローカルデバイスが生成したデータに対してローカルで動作できるようにします。

v1.11

1.11.6

バグ修正と機能向上:

- デプロイ中に電源損失が突然発生した場合の復元力を改善しました。
- ストリームマネージャーのデータ破損により AWS IoT Greengrass Core ソフトウェアが起動できなくなる問題を修正しました。
- 新規クライアントデバイスが特定のシナリオでコアに接続できない問題を修正しました。
- ストリームマネージャーのストリーム名に `.log` を含めることができなかった問題を修正しました。

1.11.5

バグ修正と機能向上:

- 全般的なパフォーマンス向上とバグ修正。

1.11.4

バグ修正と機能向上:

- AWS IoT Greengrass Core ソフトウェア v1.11.3 へのアップグレードを妨げるストリームマネージャーの問題を修正しました。ストリームマネージャーを使用してデータをクラウドにエクスポートする場合、OTA 更新を使用して、以前の v1.x バージョンの AWS IoT Greengrass Core ソフトウェアを v1.11.4 にアップグレードできるようになりました。
- 全般的なパフォーマンス向上とバグ修正。

1.11.3

バグ修正と機能向上:

- Ubuntu デバイスで AWS IoT Greengrass Core ソフトウェアがスナップで実行され、デバイスに突然電源が失われた後に応答しなくなる問題を修正しました。
- 存続期間の長い Lambda 関数への MQTT メッセージの配信が遅延する問題を修正しました。
- `maxWorkItemCount` 値が `1024` より大きい値に設定されていると、MQTT メッセージが適切に送信されない問題を修正しました。
- `config.json` の `keepAlive` プロパティで指定された MQTT `KeepAlive` 期間を OTA 更新エージェントが無視する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

> **Important**
>
> ストリームマネージャーを使用してデータをクラウドにエクスポートする場合は、以前の v1.x バージョンから AWS IoT Greengrass Core ソフトウェア v1.11.3 にアップグレードしないでください。ストリームマネージャーを初めて有効にする場合は、まず最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールすることを強くお勧めします。

1.11.1

バグ修正と機能向上:

- ストリームマネージャーのメモリ使用量が増加する問題を修正しました。

- Greengrass コアデバイスの電源がストリームデータの特定の有効期限 (TTL) を超えてオフになっている場合に、ストリームマネージャーがストリームのシーケンス番号を `0` にリセットする問題を修正しました。
- AWS クラウドへのデータエクスポートの再試行をストリームマネージャーが適切に停止できない問題を修正しました。

1.11.0

新機能:

- Greengrass コアのテレメトリエージェントは、ローカルテレメトリデータを収集して公開します AWS クラウド。テレメトリデータを取得してさらに処理するには、Amazon EventBridge ルールを作成して、ターゲットをサブスクライブします。詳細については、AWS IoT Greengrass 「コアデバイスからのシステムヘルステレメトリデータの収集」を参照してください。
- ローカル HTTP API は、 によって開始されたローカルワーカープロセスの現在の状態のスナップショットを返します AWS IoT Greengrass。詳細については、「ローカルヘルスチェック API を呼び出す」を参照してください。
- ストリームマネージャーは、Amazon S3 および にデータを自動的にエクスポートします AWS IoT SiteWise。

  新しいストリームマネージャーパラメータにより、既存のストリームを更新して、データのエクスポートを一時停止または再開できます。
- コア上で Python 3.8.x Lambda 関数を実行するためのサポート。
- Greengrass コア IPC ポート番号の設定に使用する、`config.json` の新しい `ggDaemonPort` プロパティ。デフォルトのポート番号は 8000 です。

  Greengrass コア IPC 認証のタイムアウト設定に使用する `config.json` の新しい `systemComponentAuthTimeout` プロパティ。デフォルトのタイムアウトは 5000 ミリ秒です。
- AWS IoT Greengrass グループあたりの AWS IoT デバイスの最大数を 200 から 2500 に増やしました。

  グループあたりのサブスクリプションの最大数を 1000 から 10000 に増やしました。

  詳細については、「AWS IoT Greengrass endpoints and quotas」を参照してください。

バグ修正と機能向上:

- Greengrass サービスプロセスのメモリ使用率を減らすことができる全体的な最適化。

- 新しいランタイム設定パラメータ (`mountAllBlockDevices`) により、Greengrass が OverlayFS の設定後、バインドマウントを使用してすべてのブロックデバイスをコンテナにマウントできるようになります。この機能により、`/usr` が `/` 階層下にない場合に、Greengrass のデプロイが失敗する問題が解決されました。
- がシンボリックリンクである場合に AWS IoT Greengrass コア障害が発生する問題を修正`/tmp`しました。
- Greengrass デプロイエージェントが機械学習の未使用モデルアーティファクトを `mlmodel_public` フォルダから削除する問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

Core ソフトウェアを AWS IoT Greengrass コアデバイスにインストールするには、アーキテクチャとオペレーティングシステム (OS) の パッケージをダウンロードし、「入門ガイド」のステップに従います。

> **Tip**
>
> AWS IoT Greengrass には、 AWS IoT Greengrass Core ソフトウェアをインストールするための他のオプションも用意されています。例えば、Greengrass デバイスのセットアップを使用して環境を設定し、最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールできます。または、サポートされている Debian プラットフォームでは、APT パッケージマネージャーを使用して AWS IoT Greengrass Core ソフトウェアをインストールまたはアップグレードできます。詳細については、「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

アーキテクチャ	オペレーティングシステム	リンク
Armv8 (AArch64)	リナックス	ダウンロード
Armv8 (AArch64)	Linux (OpenWrt)	ダウンロード
Armv7l	リナックス	ダウンロード
Armv7l	Linux (OpenWrt)	ダウンロード
Armv6l	リナックス	ダウンロード

アーキテクチャ	オペレーティングシステム	リンク
x86_64	リナックス	ダウンロード

Extended life versions

1.10.5

v1.10 の新機能:

- データストリームをローカルで処理し、 AWS クラウド に自動的にエクスポートするストリームマネージャー。この機能を使用するには、Greengrass Core デバイスに Java 8 が必要です。詳細については、「データストリームの管理」を参照してください。
- コアデバイスで Docker アプリケーションを実行する、新しい Greengrass Docker アプリケーションのデプロイコネクタ。詳細については、「the section called “Docker アプリケーションのデプロイ”」を参照してください。
- OPC-UA サーバーからアセットプロパティに産業用デバイスデータを送信する新しい IoT SiteWise コネクタ AWS IoT SiteWise。詳細については、「the section called “IoT SiteWise”」を参照してください。
- コンテナ化を使用せずに実行する Lambda 関数では、Greengrass グループの機械学習リソースにアクセスできます。詳細については、「the section called “機械学習リソースにアクセスする”」を参照してください。
- での MQTT 永続セッションのサポート AWS IoT。詳細については、「the section called “AWS IoT Coreを使用した MQTT 永続セッション”」を参照してください。
- ローカルの MQTT トラフィックは、デフォルトのポート 8883 以外のポートを使用できます。詳細については、「the section called “ローカルメッセージング用の MQTT ポート”」を参照してください。
- Lambda 関数から信頼性の高いメッセージを発行する、AWS IoT Greengrass Core SDK の新しい `queueFullPolicy` オプション。
- コア上で Node.js 12.x Lambda 関数を実行するためのサポート。

バグ修正と機能向上:

- ハードウェアセキュリティ統合による無線通信 (OTA) 更新は、OpenSSL 1.1 で設定できます。
- ストリームマネージャーはファイルデータの破損に対する回復性が高くなっています。

- Linux カーネル 5.1 以降を使用しているデバイスで sysfs マウントエラーが発生する問題を修正しました。
- config.json の新しいmqttOperationTimeoutプロパティ。MQTT 接続でのパブリッシュ、サブスクライブ、およびサブスクライブ解除オペレーションのタイムアウトの設定に使用します AWS IoT Core。
- ストリームマネージャーのメモリ使用量が増加する問題を修正しました。
- Greengrass コア IPC 認証のタイムアウト設定に使用する config.json の新しい systemComponentAuthTimeout プロパティ。デフォルトのタイムアウトは 5000 ミリ秒です。
- config.json の keepAlive プロパティで指定された MQTT KeepAlive 期間を OTA 更新エージェントが無視する問題を修正しました。
- maxWorkItemCount 値が 1024 より大きい値に設定されていると、MQTT メッセージが適切に送信されない問題を修正しました。
- 存続期間の長い Lambda 関数への MQTT メッセージの配信が遅延する問題を修正しました。
- Ubuntu デバイスで AWS IoT Greengrass Core ソフトウェアがスナップで実行され、デバイスに突然電源が失われた後に応答しなくなる問題を修正しました。
- 全般的なパフォーマンス向上とバグ修正。

Core ソフトウェアを AWS IoT Greengrass コアデバイスにインストールするには、アーキテクチャとオペレーティングシステム (OS) の パッケージをダウンロードし、「入門ガイド」のステップに従います。

アーキテクチャ	オペレーティングシステム	リンク
Armv8 (AArch64)	リナックス	ダウンロード
Armv8 (AArch64)	Linux (OpenWrt)	ダウンロード
Armv7l	リナックス	ダウンロード
Armv7l	Linux (OpenWrt)	ダウンロード
Armv6l	リナックス	ダウンロード
x86_64	リナックス	ダウンロード

1.9.4

v1.9 の新機能:

- Python 3.7 と Node.js 8.10 の Lambda ランタイムのサポート。Python 3.7 および Node.js 8.10 ランタイムを使用する Lambda 関数を AWS IoT Greengrass コアで実行できるようになりました (AWS IoT Greengrass 引き続き Python 2.7 および Node.js 6.10 ランタイムをサポートします) 。
- 最適化された MQTT 接続。Greengrass コアが確立する AWS IoT Coreとの接続数が少なくなりました。この変更により、接続の数に基づく料金の運用コストを削減できます。
- ローカル MQTT サーバー用楕円曲線 (EC) キー。ローカル MQTT サーバーは、RSA キーに加えて EC キーをサポートします。(MQTT サーバー証明書には、キータイプに関係なく、SHA-256 RSA 署名が含まれています。) 詳細については、「the section called “セキュリティプリンシパル”」を参照してください。
- OpenWrt. AWS IoT Greengrass Core ソフトウェア v1.9.2 以降のサポートは、Armv8 (AArch64) および Armv7l アーキテクチャの OpenWrt ディストリビューションにインストールできます。現在、OpenWrt は ML 推論をサポートしていません。
- Armv6l. AWS IoT Greengrass Core ソフトウェア v1.9.3 以降のサポートは、Armv6l アーキテクチャの Raspbian ディストリビューション (Raspberry Pi Zero デバイスなど) にインストールできます。
- ALPN を使用したポート 443 での OTA 更新。MQTT トラフィックにポート 443 を使用する Greengrass コアはover-the-air (OTA) ソフトウェア更新をサポートするようになりました。 は、Application Layer Protocol Network (ALPN) TLS 拡張機能 AWS IoT Greengrass を使用して、これらの接続を有効にします。詳細については、 AWS IoT Greengrass Core ソフトウェアの OTA 更新およびthe section called “ポート 443 での接続またはネットワークプロキシを通じた接続”を参照してください。

Core ソフトウェアを AWS IoT Greengrass コアデバイスにインストールするには、アーキテクチャとオペレーティングシステム (OS) の パッケージをダウンロードし、「入門ガイド」のステップに従います。

アーキテクチャ	オペレーティングシステム	リンク
Armv8 (AArch64)	リナックス	ダウンロード
Armv8 (AArch64)	Linux (OpenWrt)	ダウンロード
Armv7l	リナックス	ダウンロード

アーキテクチャ	オペレーティングシステム	リンク
Armv7l	Linux (OpenWrt)	ダウンロード
Armv6l	リナックス	ダウンロード
x86_64	リナックス	ダウンロード

1.8.4

- 新機能:
  - グループ内にある Lambda 関数の設定可能なデフォルトアクセス ID。このグループレベルの設定により、Lambda 関数の実行で使用されるデフォルトのアクセス許可が決まります。ユーザー ID、グループ ID、またはその両方を設定できます。個々の Lambda 関数は、そのグループのデフォルトのアクセス ID を上書きできます。詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。
  - ポート 443 経由の HTTPS トラフィック。HTTPS コミュニケーションは、デフォルトのポート 8443 ではなくポート 443 を経由するように設定できます。これにより、Application Layer Protocol Network (ALPN) TLS 拡張機能 AWS IoT Greengrass のサポートが補完され、MQTT と HTTPS の両方のすべての Greengrass メッセージングトラフィックでポート 443 を使用できます。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。
  - AWS IoT 接続のクライアント IDs予測可能。この変更により、 AWS IoT Device Defender および AWS IoT ライフサイクルイベントのサポートが有効になり、そのため接続、切断、購読、および購読解除のイベントに関する通知を受け取ることができます。予測可能な命名により、接続 ID を中心としたロジックの作成も容易になります (例えば、証明書の属性に基づいてサブスクライブポリシーテンプレートを作成します。詳細については、「the section called “AWS IoTを使用した MQTT 接続用クライアント ID”」を参照してください。

  バグ修正と機能向上:

  - シャドウ同期とデバイス証明書マネージャーの再接続の問題を修正しました。
  - 全般的なパフォーマンス向上とバグ修正。

Core ソフトウェアを AWS IoT Greengrass コアデバイスにインストールするには、アーキテクチャとオペレーティングシステム (OS) のパッケージをダウンロードし、「入門ガイド」のステップに従います。

アーキテクチャ	オペレーティングシステム	リンク
Armv8 (AArch64)	リナックス	ダウンロード
Armv7l	リナックス	ダウンロード
x86_64	リナックス	ダウンロード

このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したと見なされます。

AWS IoT Greengrass Core ソフトウェアをデバイスにインストールするためのその他のオプションについては、「」を参照してくださいthe section called “AWS IoT Greengrass Core ソフトウェアのインストール”。

AWS IoT Greengrass スナップソフトウェア

AWS IoT Greengrass snap 1.11.x を使用すると、コンテナ化された環境で、必要なすべての依存関係とともに、便利なソフトウェアパッケージ AWS IoT Greengrass を通じて の限定バージョンを実行できます。

> **Note**
>
> AWS IoT Greengrass スナップは AWS IoT Greengrass Core ソフトウェア v1.11.x. AWS IoT Greengrass で使用できます。v1.10.x にはスナップがありません。サポートされていないバージョンのバグ修正や更新プログラムは受けられません。
> AWS IoT Greengrass スナップはコネクタと機械学習 (ML) 推論をサポートしていません。

詳細については、「the section called “スナップ AWS IoT Greengrass で実行する”」を参照してください。

AWS IoT Greengrass Docker ソフトウェア

AWS には、Docker コンテナ AWS IoT Greengrass での実行を容易にする Dockerfile イメージと Docker イメージが用意されています。

Dockerfile

Dockerfiles には、カスタム AWS IoT Greengrass コンテナイメージを構築するためのソースコードが含まれています。イメージを変更して、別のプラットフォームのアーキテクチャで実行したり、イメージサイズを縮小したりできます。手順については、README ファイルを参照してください。

ターゲット AWS IoT Greengrass Core ソフトウェアバージョンをダウンロードします。

v1.11

- v1.11.6 AWS IoT Greengrass 用の Dockerfile。

Extended life versions

v1.10

v1.10.5 AWS IoT Greengrass 用の Dockerfile。

v1.9

v1.9.4 AWS IoT Greengrass 用の Dockerfile。

v1.8

v1.8.1 AWS IoT Greengrass 用の Dockerfile。

Docker イメージ

Docker イメージには、Amazon Linux 2 (x86_64) および Alpine Linux (x86_64、Armv7l、または AArch64) ベースイメージに AWS IoT Greengrass Core ソフトウェアと依存関係がインストールされています。構築済みのイメージを使用して、 AWS IoT Greengrassの試用を開始できます。

> **Important**
>
> 2022 年 6 月 30 日に、Amazon Elastic Container Registry (Amazon ECR) および Docker Hub に公開された AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージのメンテナンス AWS IoT Greengrass が終了しました。これらの Docker イメージは、メンテナ

ンス終了から 1 年後の 2023 年 6 月 30 日まで、Amazon ECR および Docker Hub から引き続きダウンロードすることができます。ただし、2022 年 6 月 30 日にメンテナンスが終了した後、 AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージはセキュリティパッチやバグ修正を受け取らなくなりました。これらの Docker イメージに依存する本稼働ワークロードを実行する場合は、 AWS IoT Greengrass が提供する Dockerfiles を使用して独自の Docker イメージを構築することをお勧めします。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

構築済みイメージは、Docker Hub または Amazon Elastic Container Registry(Amazon ECR) からダウンロードできます。

- Docker Hub の場合は、*version* タグを使用して、特定のバージョンの Greengrass Docker イメージをダウンロードします。すべての使用可能なイメージを確認するには、Docker Hub の [タグ] ページを参照してください。
- Amazon ECR の場合は、`latest` タグを使用して、利用可能な最新バージョンの Greengrass Docker イメージをダウンロードします。使用可能なイメージバージョンのリストと Amazon ECR からのイメージダウンロードの詳細については、「Docker コンテナ AWS IoT Greengrass での実行」を参照してください。

> **Warning**
>
> AWS IoT Greengrass Core ソフトウェアの v1.11.6 以降、Greengrass Docker イメージには Python 2.7 が含まれなくなりました。Python 2.7 は 2020 年にend-of-lifeとなり、セキュリティ更新プログラムを受け取らなくなったためです。これらの Docker イメージに更新する場合は、アプリケーションが新しい Docker イメージで動作することを検証した後に、アップデートを本番デバイスに展開することをお勧めします。Greengrass Docker イメージを使用するアプリケーションに Python 2.7 が必要な場合は、Greengrass Dockerfile を変更して、アプリケーションに Python 2.7 を含めることができます。

AWS IoT Greengrass は AWS IoT Greengrass Core ソフトウェア v1.11.1 の Docker イメージを提供しません。

> **Note**
>
> デフォルトでは、`alpine-aarch64` イメージと `alpine-armv7l` イメージは Arm ベースのホストでのみ実行できます。これらのイメージを x86 ホストで実行するに

は、QEMU をインストールして QEMU ライブラリをホストにマウントできます。以下に例を示します。

```
docker run --rm --privileged multiarch/qemu-user-static --reset -p yes
```

AWS IoT Greengrass コア SDK

Lambda 関数は、 AWS IoT Greengrass Core SDK を使用してローカルで AWS IoT Greengrass コアとやり取りします。これにより、デプロイされた Lambda 関数に以下を許可します。

- MQTT メッセージを と交換します AWS IoT Core。
- Greengrass グループのコネクタ、クライアントデバイス、その他の Lambda 関数で MQTT メッセージを交換します。
- ローカルシャドウサービスとやり取りを行います。
- その他のローカル Lambda 関数を呼び出します。
- シークレットリソースにアクセスします。
- ストリームマネージャーとやり取りを行います。

言語またはプラットフォームの AWS IoT Greengrass Core SDK を GitHub からダウンロードします。

- AWS IoT Greengrass Core SDK for Java
- AWS IoT Greengrass Node.js 用 Core SDK
- AWS IoT Greengrass Core SDK for Python
- AWS IoT Greengrass Core SDK for C

詳細については、「AWS IoT Greengrass コア SDK」を参照してください。

サポートされている機械学習ランタイムおよびライブラリ

Greengrass コアで推論を実行するには、ML モデルタイプ用の機械学習ランタイムまたはライブラリをインストールする必要があります。

AWS IoT Greengrass では、次の ML モデルタイプがサポートされています。モデルタイプおよびデバイスプラットフォーム用のランタイムまたはライブラリをインストールする方法については、次のリンクを参照してください。

- 深層学習ランタイム (DLR)
- MXNet
- TensorFlow

機械学習のサンプル

AWS IoT Greengrass には、サポートされている ML ランタイムとライブラリで使用できるサンプルが用意されています。これらのサンプルは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

Deep learning runtime (DLR)

デバイスプラットフォームに応じたサンプルをダウンロードしてください。

- Raspberry Pi 用の DLR サンプル
- NVIDIA Jetson TX2 用の DLR サンプル
- Intel Atom 用の DLR サンプル

DLR サンプルを使用するチュートリアルについては、「the section called “最適化された機械学習推論を設定する方法”」を参照してください。

MXNet

デバイスプラットフォームに応じたサンプルをダウンロードしてください。

- Raspberry Pi 用の MXNet サンプル
- NVIDIA Jetson TX2 用の MXNet サンプル
- Intel Atom 用の MXNet サンプル

MXNet サンプルを使用するチュートリアルについては、「the section called “機械学習推論を設定する方法”」を参照してください。

TensorFlow

デバイスプラットフォームに応じた Tensorflow サンプルをダウンロードしてください。このサンプルは、Raspberry Pi、NVIDIA Jetson TX2、Intel Atom で動作します。

AWS IoT Greengrass ML SDK ソフトウェア

AWS IoT Greengrass Machine Learning SDK により、作成済みの Lambda 関数がローカルの機械学習モデルを使用して ML フィードバックコネクタにデータを送信し、アップロードと発行を行えます。

v1.1.0

- Python 3.7

v1.0.0

- Python 2.7

ご意見をお待ちしております

ご意見をお待ちしております。お問い合わせの場合は、「AWS re:Post」にアクセスし、AWS IoT Greengrass タグを使用してください。

AWS IoT Greengrass Core ソフトウェアのインストール

AWS IoT Greengrass Core ソフトウェアは AWS 機能を AWS IoT Greengrass コアデバイスに拡張し、ローカルデバイスが生成したデータに対してローカルで動作できるようにします。

AWS IoT Greengrass には、 AWS IoT Greengrass Core ソフトウェアをインストールするためのいくつかのオプションがあります。

- tar.gz ファイルをダウンロードして解凍する。

- Greengrass デバイスのセットアップスクリプトを実行する。
- APT リポジトリからインストールする。

AWS IoT Greengrass は、 AWS IoT Greengrass Core ソフトウェアを実行するコンテナ化された環境も提供します。

- Docker コンテナ AWS IoT Greengrass で を実行します。
- スナップ AWS IoT Greengrass で を実行します。

AWS IoT Greengrass Core ソフトウェアパッケージをダウンロードして抽出する

プラットフォームの AWS IoT Greengrass Core ソフトウェアを選択して tar.gz ファイルとしてダウンロードし、デバイスで抽出します。最新バージョンのソフトウェアをダウンロードできます。詳細については、「the section called “AWS IoT Greengrass コアソフトウェア”」を参照してください。

Greengrass デバイスのセットアップスクリプトを実行する

Greengrass デバイスのセットアップを実行してデバイスを設定し、最新の AWS IoT Greengrass Core ソフトウェアバージョンをインストールし、Hello World Lambda 関数を数分でデプロイします。詳細については、「the section called “クイックスタート: Greengrass デバイスのセットアップ”」を参照してください。

APT リポジトリから AWS IoT Greengrass Core ソフトウェアをインストールする

> **Important**
>
> 2022 年 2 月 11 日をもって、APT リポジトリから AWS IoT Greengrass Core ソフトウェアをインストールまたは更新できなくなります。 AWS IoT Greengrass リポジトリを追加したデバイスでは、ソースリストからリポジトリを削除する必要があります。APT リポジトリか

らソフトウェアを実行するデバイスは、引き続き正常に動作します。tar ファイルを使用して AWS IoT Greengrass Core ソフトウェアを更新することをお勧めします。

が提供する APT リポジトリ AWS IoT Greengrass には、次のパッケージが含まれています。

- `aws-iot-greengrass-core`。 AWS IoT Greengrass Core ソフトウェアをインストールします。
- `aws-iot-greengrass-keyring`。 AWS IoT Greengrass パッケージリポジトリの署名に使用される GnuPG (GPG) キーをインストールします。

  このソフトウェアをダウンロードすると、Greengrass Core ソフトウェアのライセンス契約に同意したと見なされます。

トピック

- systemd スクリプトを使用した Greengrass デーモンのライフサイクルの管理
- APT リポジトリを使用して AWS IoT Greengrass コアソフトウェアをアンインストールする
- AWS IoT Greengrass コアソフトウェアリポジトリソースを削除する

systemd スクリプトを使用した Greengrass デーモンのライフサイクルの管理

このaws-iot-greengrass-coreパッケージでは、 AWS IoT Greengrass Core ソフトウェア (デーモン) ライフサイクルの管理に使用できるsystemdスクリプトもインストールされます。

- 起動時に Greengrass デーモンを開始するには:

```
systemctl enable greengrass.service
```

- Greengrass デーモンを開始するには:

```
systemctl start greengrass.service
```

- Greengrass デーモンを停止するには:

```
systemctl stop greengrass.service
```

- Greengrass デーモンのステータスを確認するには:

```
systemctl status greengrass.service
```

APT リポジトリを使用して AWS IoT Greengrass コアソフトウェアをアンインストールする

AWS IoT Greengrass コアソフトウェアをアンインストールするときに、デバイス証明書、グループ情報、ログファイルなど、 AWS IoT Greengrass コアソフトウェアの設定情報を保持するか削除するかを選択できます。

AWS IoT Greengrass コアソフトウェアをアンインストールし、設定情報を保持するには

- 次のコマンドを実行して、 AWS IoT Greengrass コアソフトウェアパッケージを削除し、設定情報を `/greengrass`フォルダに保存します。

  ```
  sudo apt remove aws-iot-greengrass-core aws-iot-greengrass-keyring
  ```

AWS IoT Greengrass コアソフトウェアをアンインストールし、設定情報を削除するには

1. 次のコマンドを実行して、 AWS IoT Greengrass コアソフトウェアパッケージを削除し、 から設定情報を削除します`/greengrass folder`。

   ```
   sudo apt purge aws-iot-greengrass-core aws-iot-greengrass-keyring
   ```

2. ソースリストから AWS IoT Greengrass コアソフトウェアリポジトリを削除します。詳細については、「AWS IoT Greengrass コアソフトウェアリポジトリソースを削除する」を参照してください。

AWS IoT Greengrass コアソフトウェアリポジトリソースを削除する

APT リポジトリから AWS IoT Greengrass コアソフトウェアをインストールまたは更新する必要がなくなった場合は、 AWS IoT Greengrass コアソフトウェアリポジトリソースを削除できます。2022 年 2 月 11 日以降、`apt update` の実行時にエラーが発生しないように、ソースリストからリポジトリを削除する必要があります。

APT リポジトリをソースリストから削除するには

- 次のコマンドを実行して、ソースリストから AWS IoT Greengrass コアソフトウェアリポジトリを削除します。

```
sudo rm /etc/apt/sources.list.d/greengrass.list
sudo apt update
```

Docker コンテナ AWS IoT Greengrass で を実行する

AWS IoT Greengrass には、Docker コンテナで AWS IoT Greengrass Core ソフトウェアを簡単に実行できるようにする Dockerfile イメージと Docker イメージが用意されています。詳細については、「the section called “AWS IoT Greengrass Docker ソフトウェア”」を参照してください。

> **Note**
>
> Docker アプリケーションは、Greengrass コアデバイスで実行することもできます。そのためには、Greengrass Docker アプリケーションのデプロイコネクタを使用します。

スナップ AWS IoT Greengrass で実行する

AWS IoT Greengrass snap 1.11.x を使用すると、コンテナ化された環境で、必要なすべての依存関係とともに、便利なソフトウェアパッケージ AWS IoT Greengrass を通じて の限定バージョンを実行できます。

2023 年 12 月 31 日、 AWS IoT Greengrass は、 snapcraft.io公開されている AWS IoT Greengrass コアソフトウェアバージョン 1.11.x スナップのメンテナンスを終了します。現在スナップを実行しているデバイスは、追って通知があるまで引き続き動作します。ただし、メンテナンス終了後、AWS IoT Greengrass コアスナップはセキュリティパッチやバグ修正を受信しなくなります。

スナップの概念

AWS IoT Greengrass スナップの使い方を理解しやすくするために重要なスナップの概念を以下に示します。

チャンネル

インストールされ、更新のために追跡されるスナップのバージョンを定義するスナップコンポーネント。スナップは、現在のチャネルの最新バージョンに自動的に更新されます。

インターフェイス

ネットワークやユーザーファイルなどのリソースへのアクセスを許可するスナップコンポーネント。

AWS IoT Greengrass スナップを実行するには、次のインターフェイスを接続する必要があります。最初に `greengrass-support-no-container` を接続し、切断してはならないことに注意してください。

```
    - greengrass-support-no-container
    - hardware-observe
    - home-for-hooks
    - hugepages-control
    - log-observe
    - mount-observe
    - network
    - network-bind
    - network-control
    - process-control
    - system-observe
```

その他のインターフェイスはオプションです。Lambda 関数が特定のリソースにアクセスする必要がある場合は、適切なインターフェイスに接続する必要があります。

更新

スナップは自動的に更新されます。snapd デーモンは、デフォルトで 1 日に 4 回更新をチェックするスナップパッケージマネージャーです。各更新チェックは更新と呼ばれます。更新が発生すると、デーモンが停止し、スナップが更新され、デーモンが再起動されます。

詳細については、Snapcraft のウェブサイトを参照してください。

AWS IoT Greengrass snap v1.11.x の新機能

以下では、 AWS IoT Greengrass スナップのバージョン 1.11.x の新機能と変更点について説明します。

- このバージョンでは、ユーザー ID (UID) およびグループ (GID) の 584788 として公開された snap_daemon ユーザーのみサポートします。
- このバージョンでは、コンテナ化されていない Lambda 関数のみをサポートしています。

> **Important**
>
> コンテナ化されていない Lambda 関数は同じユーザー (snap_daemon) を共有する必要があるため、Lambda 関数は互いに分離されません。詳細については、「Greengrass Lambda 関数のグループ固有の設定による実行の制御」を参照してください。

- このバージョンでは、C、C++、Java 8、Node.js 12.x、Python 2.7、Python 3.7、Python 3.8 のランタイムがサポートされます。

> **Note**
>
> 冗長な Python ランタイムを避けるために、Python 3.7 Lambda 関数は実際には Python 3.8 ランタイムを実行します。

AWS IoT Greengrass スナップの開始方法

次の手順は、デバイスに AWS IoT Greengrass スナップをインストールして設定するのに役立ちます。

要件

AWS IoT Greengrass スナップを実行するには、以下を実行する必要があります。

- Ubuntu、Linux Mint、Debian、Fedora など、サポートされている Linux ディストリビューションで AWS IoT Greengrass スナップを実行します。
- snapd デーモンをデバイスにインストールします。snap ツールを含む snapd デーモンは、デバイス上のスナップ環境を管理します。

サポートされている Linux ディストリビューションのリストとインストール手順については、スナップに関する資料の「Installing snapd」(snapd をインストールする) を参照してください。

AWS IoT Greengrass スナップをインストールして設定する

次のチュートリアルでは、デバイスに AWS IoT Greengrass スナップをインストールして設定する方法を示します。

> **Note**
>
> - このチュートリアルでは Amazon EC2 インスタンス (x86 t2.micro Ubuntu 20.04) を使用しますが、Raspberry Pi などの物理ハードウェアを使用して AWS IoT Greengrass スナップを実行できます。
> - snapd デーモンは Ubuntu にプリインストールされています。

1. 使用するデバイスのターミナルで次のコマンドを実行して、core18 スナップをインストールします。

   ```
   sudo snap install core18
   ```

   core18 スナップとは、一般的に使用されるライブラリを持つランタイム環境を提供するベーススナップです。このスナップは、Ubuntu 18.04 LTS から構築されます。
2. 次のコマンドを実行して snapd をアップグレードします。

   ```
   sudo snap install --channel=edge snapd; sudo snap refresh --channel=edge snapd
   ```
3. snap list コマンドを実行して、 AWS IoT Greengrass スナップがインストールされているかどうかを確認します。

   次のレスポンスの例では、snapd はインストールされているが、aws-iot-greengrass はインストールされていないことを示しています。

   ```
   Name              Version               Rev    Tracking         Publisher   Notes
   amazon-ssm-agent  3.0.161.0             2996   latest/stable/…  aws#        classic
   core              16-2.48               10444  latest/stable    canonical#  core
   core18            20200929              1932   latest/stable    canonical#  base
   lxd               4.0.4                 18150  4.0/stable/…     canonical#  -
   snapd             2.48+git548.g929ccfb  10526  latest/edge      canonical#  snapd
   ```
4. snap 1.11.x AWS IoT Greengrass をインストールするには、次のいずれかのオプションを選択します。

- AWS IoT Greengrass スナップをインストールするには、次のコマンドを実行します。

  ```
  sudo snap install aws-iot-greengrass
  ```

  レスポンスの例:

  ```
  aws-iot-greengrass 1.11.5 from Amazon Web Services (aws) installed
  ```

- 以前のバージョンから v1.11.x に移行するか、使用可能な最新のパッチバージョンにアップデートするには、次のコマンドを実行します。

  ```
  sudo snap refresh --channel=1.11.x aws-iot-greengrass
  ```

他のスナップと同様に、 AWS IoT Greengrass スナップはチャネルを使用してマイナーバージョンを管理します。スナップは、現在のチャネルの利用可能な最新バージョンに自動的に更新されます。たとえば、 を指定すると`--channel=1.11.x`、 AWS IoT Greengrass スナップは v1.11.5 に更新されます。

`snap info aws-iot-greengrass` コマンドを実行して、使用可能なチャネルのリストを取得できます AWS IoT Greengrass。

レスポンスの例:

```
name:      aws-iot-greengrass
summary:   AWS supported software that extends cloud capabilities to local devices.
publisher: Amazon Web Services (aws#)
store-url: https://snapcraft.io/aws-iot-greengrass
contact:   https://repost.aws/tags/TA4ckIed1sR4enZBey29rKTg/aws-io-t-greengrass
license:   Proprietary
description: |
  AWS IoT Greengrass seamlessly extends AWS onto edge devices so they can act
 locally on the data
  they generate, while still using the cloud for management, analytics, and durable
 storage.
  AWS IoT Greenrgrass snap v1.11.0 enables you to run a limited version of AWS IoT
 Greengrass with
  all necessary dependencies in a containerized environment.
  The AWS IoT Greengrass snap doesn't support connectors and machine learning (ML)
 inference.
```

```
  By downloading this software you agree to the Greengrass Core Software License
 Agreement
  (https://s3-us-west-2.amazonaws.com/greengrass-release-license/greengrass-
license-v1.pdf).
  For more information, see Run AWS IoT Greengrass in a snap
  (https://docs.aws.amazon.com/greengrass/latest/developerguide/install-
ggc.html#gg-snap-support) in
  the AWS IoT Greengrass Developer.
  If you need help, try the AWS IoT Greengrass tag on AWS re:Post
  (https://repost.aws/tags/TA4ckIed1sR4enZBey29rKTg/aws-io-t-greengrass) or connect
 with an AWS IQ expert
  (https://iq.aws.amazon.com/services/aws/greengrass).
snap-id: SRDuhPJGj4XPxFNNZQKOTvURAp0wxKnd
channels:
  latest/stable:    1.11.3 2021-06-15 (59) 111MB -
  latest/candidate: 1.11.3 2021-06-14 (59) 111MB -
  latest/beta:      1.11.3 2021-06-14 (59) 111MB -
  latest/edge:      1.11.3 2021-06-14 (59) 111MB -
  1.11.x/stable:    1.11.3 2021-06-15 (59) 111MB -
  1.11.x/candidate: 1.11.3 2021-06-15 (59) 111MB -
  1.11.x/beta:      1.11.3 2021-06-15 (59) 111MB -
  1.11.x/edge:      1.11.3 2021-06-15 (59) 111MB -
```

5. Lambda 関数が必要とする特定のリソースにアクセスするには、追加のインターフェイスに接続できます。

   次のコマンドを実行して、 AWS IoT Greengrass スナップでサポートされているインターフェイスのリストを取得します。

```
snap connections aws-iot-greengrass
```

   レスポンスの例:

```
Interface                Plug                                                Slot
               Notes
camera                   aws-iot-greengrass:camera                           -
               -
dvb                      aws-iot-greengrass:dvb                              -
               -
gpio                     aws-iot-greengrass:gpio                             -
               -
```

```
gpio-memory-control      aws-iot-greengrass:gpio-memory-control              -
               -
greengrass-support       aws-iot-greengrass:greengrass-support-no-container
  :greengrass-support  -
hardware-observe         aws-iot-greengrass:hardware-observe
  :hardware-observe    manual
hardware-random-control  aws-iot-greengrass:hardware-random-control          -
               -
home                     aws-iot-greengrass:home-for-greengrassd             -
               -
home                     aws-iot-greengrass:home-for-hooks                   :home
               manual
hugepages-control        aws-iot-greengrass:hugepages-control
  :hugepages-control   manual
i2c                      aws-iot-greengrass:i2c                              -
               -
iio                      aws-iot-greengrass:iio                              -
               -
joystick                 aws-iot-greengrass:joystick                         -
               -
log-observe              aws-iot-greengrass:log-observe                      :log-
observe         manual
mount-observe            aws-iot-greengrass:mount-observe
  :mount-observe       manual
network                  aws-iot-greengrass:network
  :network             -
network-bind             aws-iot-greengrass:network-bind
  :network-bind        -
network-control          aws-iot-greengrass:network-control
  :network-control     -
opengl                   aws-iot-greengrass:opengl
  :opengl              -
optical-drive            aws-iot-greengrass:optical-drive
  :optical-drive       -
process-control          aws-iot-greengrass:process-control
  :process-control     -
raw-usb                  aws-iot-greengrass:raw-usb                          -
               -
removable-media          aws-iot-greengrass:removable-media                  -
               -
serial-port              aws-iot-greengrass:serial-port                      -
               -
spi                      aws-iot-greengrass:spi                              -
               -
```

```
system-observe           aws-iot-greengrass:system-observe
  :system-observe      -
```

[Slot] (スロット) 列がハイフン (-) になっている場合、対応するインターフェイスは接続されていません。

6. AWS IoT Greengrass 「 Core ソフトウェアのインストール」に従って、 AWS IoT モノ、Greengrass グループ、 との安全な通信を可能にするセキュリティリソース AWS IoT、および AWS IoT Greengrass Core ソフトウェア設定ファイルを作成します。設定ファイルにはconfig.json、証明書ファイルの場所や AWS IoT デバイスデータエンドポイントなど、Greengrass コアに固有の設定が含まれています。

> **Note**
>
> ファイルを別のデバイスにダウンロードした場合は、次のステップに従ってファイルを AWS IoT Greengrass コアデバイスに転送します。

7. AWS IoT Greengrass スナップの場合は、以下に示すように config.json ファイルを更新してください。

   - *certificateId* の各インスタンスを証明書とキーファイルの名前に含まれる証明書 ID と置き換えます。
   - Amazon ルート CA 1 とは異なる Amazon ルート CA 証明書をダウンロードした場合は、*AmazonRootCA1.pem* の各インスタンスを Amazon ルート CA ファイルの名前と置き換えます。

```
{
  ...
  "crypto" : {
    "principals" : {
      "SecretsManager" : {
        "privateKeyPath" : "file:///snap/aws-iot-greengrass/current/greengrass/
certs/certificateId-private.pem.keyy"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///snap/aws-iot-greengrass/current/greengrass/
certs/certificateId-private.pem.key",
        "certificatePath" : "file:///snap/aws-iot-greengrass/current/greengrass/
certs/certificateId-certificate.pem.crt"
      }
```

```
    },
    "caPath" : "file:///snap/aws-iot-greengrass/current/greengrass/
certs/AmazonRootCA1.pem"
  },
  "writeDirectory": "/var/snap/aws-iot-greengrass/current/ggc-write-directory",
  "pidFileDirectory": "/var/snap/aws-iot-greengrass/current/pidFileDirectory"
}
```

8. 次のコマンドを実行して、 AWS IoT Greengrass 証明書と設定ファイルを追加します。

```
sudo snap set aws-iot-greengrass gg-certs=/home/ubuntu/my-certs
```

Lambda 関数のデプロイ

このセクションでは、カスタマー管理の Lambda 関数を AWS IoT Greengrass スナップにデプロイする方法を示します。

> **Important**
>
> AWS IoT Greengrass snap v1.11 は、コンテナ化されていない Lambda 関数のみをサポートします。

1. 次のコマンドを実行して、 AWS IoT Greengrass デーモンを起動します。

```
sudo snap start aws-iot-greengrass
```

レスポンスの例:

```
Started.
```

> **Note**
>
> エラーを受け取る場合は、`snap run` コマンドを使用すると、詳細なエラーメッセージを表示することができます。トラブルシューティングの詳細については、「error: cannot perform the following tasks: - Run service command "start" for services ["greengrassd"] of snap "aws-iot-greengrass" ([start snap.aws-iot-greengrass.greengrassd.service] failed with exit status 1: Job for snap.aws-iot-

greengrass.greengrassd.service failed because the control process exited with error code. See "systemctl status snap.aws-iot-greengrass.greengrassd.service" and "journalctl -xe" for details.)」を参照してください。

2. デーモンが実行中であることを確認するには、次のコマンドを実行します。

```
snap services aws-iot-greengrass.greengrassd
```

レスポンスの例:

```
Service                          Startup   Current  Notes
aws-iot-greengrass.greengrassd   disabled  active   -
```

3. 「モジュール 3 (パート 1): AWS IoT Greengrassでの Lambda 関数」に従って、Hello World Lambda 関数を作成し、デプロイします。ただし、Lambda 関数をデプロイする前に、次のステップを完了してください。

4. Lambda 関数が `snap_daemon` ユーザーとして、コンテナなしモードで実行されていることを確認します。Greengrass グループの設定を更新するには、 AWS IoT Greengrass コンソールで次の操作を行います。

   a. AWS IoT Greengrass コンソールにサインインします。

   b. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。

   c. [Greengrass groups] (グリーングラスのグループ) で、対象グループを選択します。

   d. ナビゲーションペインのグループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。

   e. [Default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境) から、[Edit] (編集) を選択して、次を行います。

      i. [Default system user and group] (デフォルトシステムユーザーとグループ) には、[Another user ID/group ID] (別のユーザー ID とグループ ID) を選択し、[System user ID (number)] (システムユーザー ID (数値)) と [System group ID (number)] (システムグループ ID (数値)) の両方に **584788** を入力します。

      ii. [Default Lambda function containerization] (デフォルト Lambda 関数のコンテナ化) には、[No container] (コンテナなし) を選択します。

      iii. [Save] を選択します。

AWS IoT Greengrass デーモンの停止

`snap stop` コマンドを使用して、サービスを停止することができます。

AWS IoT Greengrass デーモンを停止するには、次のコマンドを実行します。

```
sudo snap stop aws-iot-greengrass
```

コマンドが `Stopped.` を返します。

スナップが正常に停止したかどうかを確認するには、次のコマンドを実行します。

```
snap services aws-iot-greengrass.greengrassd
```

レスポンスの例:

```
Service                          Startup   Current   Notes
aws-iot-greengrass.greengrassd  disabled  inactive  -
```

AWS IoT Greengrass スナップのアンインストール

AWS IoT Greengrass スナップをアンインストールするには、次のコマンドを実行します。

```
sudo snap remove aws-iot-greengrass
```

レスポンスの例:

```
aws-iot-greengrass removed
```

AWS IoT Greengrass スナップのトラブルシューティング

AWS IoT Greengrass スナップに関する問題のトラブルシューティングには、次の情報を使用します。

「アクセス許可が拒否されました」というエラーを受け取った。

解決策:「アクセス許可が拒否されました」というエラーは、多くの場合、インターフェイスがないために発生します。欠落しているインターフェイスのリストと詳細なトラブルシューティング情報については、`snappy-debug` ツールを使用することができます。

次のコマンドを実行して、ツールをインストールします。

```
sudo snap install snappy-debug
```

レスポンスの例:

```
snappy-debug 0.36-snapd2.45.1 from Canonical# installed
```

別のターミナルセッションで `sudo snappy-debug` コマンドを実行します。この操作は、「アクセス許可が拒否されました」というエラーが発生するまで続行します。

例えば、Lambda 関数が `$HOME` ディレクトリ内のファイルを読み取ろうとすると、次のレスポンスが返されます。

```
INFO: Following '/var/log/syslog'. If have dropped messages, use:
INFO: $ sudo journalctl --output=short --follow --all | sudo snappy-debug
kernel.printk_ratelimit = 0
= AppArmor =
Time: Dec  6 04:48:26
Log: apparmor="DENIED" operation="mknod" profile="snap.aws-iot-greengrass.greengrassd"
 name="/home/ubuntu/my-file.txt" pid=12345 comm="touch" requested_mask="c"
 denied_mask="c" fsuid=0 ouid=0
File: /home/ubuntu/my-file.txt (write)
Suggestion:
* add 'home' to 'plugs'
```

次の例は、`/home/ubuntu/my-file.txt` ファイルを作成したら、アクセス許可エラーが返されたことを示しています。また、`home` を `plugs` に追加するように提案しています。ただし、この提案は適用されません。`home-for-greengrassd` プラグおよび `home-for-hooks` プラグには読み取り専用アクセスのみが与えられます。

詳細については、スナップに関する資料の「The snappy-debug snap」(snappy-debug スナップ) を参照してください。

error: cannot perform the following tasks: - Run service command "start" for services ["greengrassd"] of snap "aws-iot-greengrass" ([start snap.aws-iot-greengrass.greengrassd.service] failed with exit status 1: Job for snap.aws-iot-greengrass.greengrassd.service failed because the control process exited with error code. See "systemctl status snap.aws-iot-greengrass.greengrassd.service" and "journalctl -xe" for details.)

解決策: `snap start aws-iot-greengrass` コマンドが AWS IoT Greengrass Core ソフトウェアの起動に失敗すると、このエラーが表示されることがあります。

トラブルシューティングの詳細を得るには、次のコマンドを実行します。

```
sudo snap run aws-iot-greengrass.greengrassd
```

レスポンスの例:

```
Couldn't find /snap/aws-iot-greengrass/44/greengrass/config/config.json.
```

この例では、 AWS IoT Greengrass が `config.json` ファイルを見つけられなかったことを示しています。設定ファイルと証明書ファイルは確認できます。

/var/snap/aws-iot-greengrass/current/ggc-write-directory/packages/1.11.5/rootfs/merged is not an absolute path or is a symlink。

解決策: AWS IoT Greengrass スナップは、コンテナ化されていない Lambda 関数のみをサポートします。Lambda 関数がコンテナなしモードで実行されていることを確認します。詳細については、「AWS IoT Greengrass Version 1 デベロッパーガイド」の「Lambda 関数のコンテナ化を選択する場合の考慮事項」を参照してください。

sudo snap refresh snapd コマンドを実行した後、snapd デーモンの再起動に失敗した。

解決策: 「」のステップ 6 ~ 8 AWS IoT Greengrass スナップをインストールして設定するに従って、AWS IoT Greengrass 証明書と設定ファイルを AWS IoT Greengrass スナップに追加します。

AWS IoT Greengrass Core ソフトウェアのインストールをアーカイブする

AWS IoT Greengrass Core ソフトウェアの新しいバージョンにアップグレードすると、現在インストールされているバージョンをアーカイブできます。これにより現在のインストール環境が維持されるため、同じハードウェア上で新しいソフトウェアバージョンをテストできます。これにより、何らかの理由であなたのアーカイブバージョンに簡単にロールバックすることもできます。

現在のインストールをアーカイブして新しいバージョンをインストールするには

1. アップグレードする AWS IoT Greengrass Core ソフトウェアインストールパッケージをダウンロードします。
2. パッケージを宛先コアデバイスにコピーします。ファイルを転送する方法を示す手順については、こちらのステップを参照してください。

> **Note**
> 現在の証明書、キー、および設定ファイルを後で新しいインストールにコピーします。

コアデバイス端末で次のステップのコマンドを実行します。

3. コアデバイスで Greengrass デーモンが停止していることを確認します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   出力に root で実行中の /greengrass/ggc/packages/*ggc-version*/bin/daemon のエントリが含まれていれば、デーモンは実行されています。

   > **Note**
   > この手順は、 AWS IoT Greengrass Core ソフトウェアが /greengrass ディレクトリにインストールされていることを前提として記述されています。

   b. デーモンを停止するには

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd stop
   ```

4. 現在の Greengrass ルートディレクトリを別のディレクトリに移動します。

   ```
   sudo mv /greengrass /greengrass_backup
   ```

5. コアデバイス上の新しいソフトウェアを解凍します。コマンドの *os* *######*プレースホルダーと*#####*プレースホルダーを交換します。

   ```
   sudo tar -zxvf greengrass-os-architecture-version.tar.gz -C /
   ```

6. アーカイブされた証明書、キー、および設定ファイルを後で新しいインストールにコピーします。

   ```
   sudo cp /greengrass_backup/certs/* /greengrass/certs
   ```

```
sudo cp /greengrass_backup/config/* /greengrass/config
```

7. デーモンを開始します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

これで、新しいインストールをテストするためのグループデプロイを作成できます。何らかの失敗が発生した場合は、アーカイブされたインストールを復元することができます。

アーカイブされたインストールを復元するには

1. デーモンを停止します。
2. 新しい `/greengrass` ディレクトリを削除します。
3. `/greengrass_backup` ディレクトリを `/greengrass` に戻します。
4. デーモンを開始します。

AWS IoT Greengrass コアを設定する

AWS IoT Greengrass コアは、エッジ環境でハブまたはゲートウェイとして機能する AWS IoT モノ (デバイス) です。他の AWS IoT デバイスと同様に、コアはレジストリに存在し、デバイスシャドウがあり、デバイス証明書を使用して AWS IoT Core と で認証します AWS IoT Greengrass。コアデバイスは AWS IoT Greengrass Core ソフトウェアを実行します。これによって、通信、車道同期やトークン交換などの Greengrass グループのローカルプロセスを管理することができます。

AWS IoT Greengrass Core ソフトウェアには、次の機能があります。

- コネクタと Lambda 関数のデプロイとローカル実行。
- への自動エクスポートを使用して、データストリームをローカルで処理します AWS クラウド。
- マネージドサブスクリプションを使用したデバイス、コネクタ、および Lambda 関数間のローカルネットワークを介した MQTT メッセージング。
- マネージドサブスクリプションを使用した と AWS IoT デバイス、コネクタ、Lambda 関数間の MQTT メッセージング。
- デバイスの認証と認可を使用したデバイスと AWS クラウド 間の安全な接続。
- デバイスのローカルシャドウ同期。シャドウは AWS クラウドと同期するように設定できます。
- ローカルデバイスとボリュームリソースへの制御されたアクセス。

- ローカル推論を実行するためにクラウドでトレーニングされた機械学習モデルのデプロイ。
- デバイスで Greengrass コアデバイスを検出するための IP アドレス自動検出。
- 新規作成または更新されたグループ設定の一元的デプロイ。設定データをダウンロードすると、コアデバイスが自動的に再起動されます。
- ユーザー定義の Lambda 関数の安全な無線通信 (OTA) によるソフトウェア更新。
- コネクタと Lambda 関数で制御される、ローカルシークレットの安全な暗号化されたストレージ。

AWS IoT Greengrass コア設定ファイル

AWS IoT Greengrass Core ソフトウェアの設定ファイルは ですconfig.json。このファイルは、/*greengrass-root*/config ディレクトリにあります。

> **Note**
>
> *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは /greengrass ディレクトリです。
> AWS IoT Greengrass コンソールからデフォルトグループ作成オプションを使用する場合、config.jsonファイルは動作状態でコアデバイスにデプロイされます。

以下のコマンドを実行して、このファイルのコンテンツを確認できます。

```
cat /greengrass-root/config/config.json
```

次は、config.json ファイルの例です。これは、 AWS IoT Greengrass コンソールからコアを作成するときに生成されるバージョンです。

GGC v1.11

```
{
    "coreThing": {
        "caPath": "root.ca.pem",
        "certPath": "hash.cert.pem",
        "keyPath": "hash.private.key",
        "thingArn": "arn:partition:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix-ats.iot.region.amazonaws.com",
        "ggHost": "greengrass-ats.iot.region.amazonaws.com",
```

```
        "keepAlive": 600,
        "ggDaemonPort": 8000,
        "systemComponentAuthTimeout": 5000
    },
    "runtime": {
        "maxWorkItemCount": 1024,
        "maxConcurrentLimit": 25,
        "lruSize": 25,
        "mountAllBlockDevices": "no",
        "cgroup": {
            "useSystemd": "yes"
        }
    },
    "managedRespawn": false,
    "crypto": {
        "principals": {
            "SecretsManager": {
                "privateKeyPath": "file:///greengrass/certs/hash.private.key"
            },
            "IoTCertificate": {
                "privateKeyPath": "file:///greengrass/certs/hash.private.key",
                "certificatePath": "file:///greengrass/certs/hash.cert.pem"
            }
        },
        "caPath": "file:///greengrass/certs/root.ca.pem"
    },
    "writeDirectory": "/var/snap/aws-iot-greengrass/current/ggc-write-directory",
    "pidFileDirectory": "/var/snap/aws-iot-greengrass/current/pidFileDirectory"
}
```

`config.json` ファイルは以下のプロパティをサポートしています。

coreThing

フィールド	説明	メモ
`caPath`	`/greengrass-root /certs` ディレクトリに対する AWS IoT ルート CA へのパス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。

フィールド	説明	メモ
		Note エンドポイントが証明書タイプに対応していることを確認してください。
`certPath`	`/greengrass-root /certs` ディレクトリからコアデバイス証明書への相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。
`keyPath`	`/greengrass-root /certs` ディレクトリから Core プライベートキーへの相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。
`thingArn`	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	Core の下の AWS IoT Greengrass コンソールで、または CLI コマンドを実行して、コアの ARN `aws greengrass get-core-definition-version` を見つけます。

フィールド	説明	メモ
`iotHost`	AWS IoT エンドポイント。	AWS IoT コンソールの「設定」で、または `aws iot describe-endpoint --endpoint-type iot:Data-ATS` CLI コマンドを実行してエンドポイントを見つけます。 このコマンドは Amazon Trust Services (ATS) エンドポイントを返します。詳細については、「サーバー認証」のドキュメントを参照してください。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。

フィールド	説明	メモ
`ggHost`	AWS IoT Greengrass エンドポイント。	これはホストのプレフィクスが greengrass で置き換えられた `iotHost` エンドポイント (例: `greengrass-ats.iot.`*`region`*`.amazonaws.com`) です。 AWS リージョン と同じ を使用します`iotHost`。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。
`iotMqttPort`	オプション。MQTT 通信に使用するポート番号 AWS IoT。	有効な値は 8883 または 443 です。デフォルト値は 8883 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
`iotHttpPort`	オプション。 AWS IoTへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`ggMqttPort`	オプション。ローカルネットワーク経由の MQTT 通信に使用するポート番号。	有効な値は 1024 ~ 65535 です。デフォルト値は 8883 です。詳細については、「the section called “ローカルメッセージング用の MQTT ポート”」を参照してください。
`ggHttpPort`	オプション。 AWS IoT Greengrass サービスへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`keepAlive`	オプション。MQTT の `KeepAlive` 期間 (秒単位)。	有効な範囲は 30 ~ 1200 秒です。デフォルト値は 600 です。
`networkProxy`	オプション。接続先のプロキシサーバーを定義するオブジェクト。	プロキシサーバーには HTTP または HTTPS を使用できます。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
`mqttOperationTimeout`	オプション。Greengrass コアが AWS IoT Coreへの MQTT 接続で発行、サブスクリプション、またはサブスクリプション解除オペレーションを完了するまでの時間 (秒単位)。	デフォルト値は 5 です。最小値は 5 です。
`ggDaemonPort`	オプション。Greengrass Core の IPC ポート番号。	このプロパティは v1.11.0 AWS IoT Greengrass 以降で使用できます。 有効な値は 1024 ~ 65535 です。デフォルト値は 8000 です。
`systemComponentAuthTimeout`	オプション。Greengrass Core IPC が認証を完了するまでに与えられる時間 (ミリ秒)。	このプロパティは v1.11.0 AWS IoT Greengrass 以降で使用できます。 有効な値は 500 ~ 5000 です。デフォルト値は 5000 です。

runtime

フィールド	説明	メモ
`maxWorkItemCount`	オプション。Greengrass デーモンが一度に処理できる作業項目の最大数。この制限を超える作業項目は無視されます。 作業項目キューは、システムコンポーネント、ユーザー定	デフォルト値は 1024 です。最大値は、デバイスのハードウェアによって制限されます。 この値を大きくすると、 が AWS IoT Greengrass 使用するメモリが増加します。コア

フィールド	説明	メモ
	義の Lambda 関数、コネクタによって共有されます。	が大量の MQTT メッセージトラフィックを受信することが予想される場合は、この値を増やすことができます。
`maxConcurrentLimit`	オプション。Greengrass デーモンで同時接続が可能な Lambda ワーカーの最大数。このパラメータは、整数で別の値を指定して上書きできます。	デフォルト値は 25 です。最小値は `lruSize` で定義されます。
`lruSize`	Optional. Defines the minimum value for `maxConcurrentLimit` .	The default value is 25.
`mountAllBlockDevices`	Optional. Enables AWS IoT Greengrass to use bind mounts to mount all block devices into a container after setting up the OverlayFS.	このプロパティは v1.11.0 AWS IoT Greengrass 以降で使用できます。 有効な値は、`yes` および no です。デフォルト値は no です。 `/usr` ディレクトリが `/` の下にない場合は、この値を `yes` に設定してください。
`postStartHealthCheckTimeout`	Optional. The time (in milliseconds) after starting that the Greengrass daemon waits for the health check to finish.	The default timeout is 30 seconds (30000 ms).
`cgroup`		

フィールド	説明	メモ
`useSystemd`	Indicates whether your device uses `systemd`.	Valid values are ## or # ##. Run the `check_ggc_dependencies` script in モジュール 1 to see if your device uses `systemd`.

crypto

`crypto` には、PKCS#11 を使用したハードウェアセキュリティモジュール (HSM) でのプライベートキーストレージをサポートするプロパティと、ローカルでのシークレットストレージをサポートするプロパティが含まれています。詳細についてはthe section called “セキュリティプリンシパル”、the section called “ハードウェアセキュリティ統合”、およびCore にシークレットをデプロイするを参照してください。HSM またはファイルシステムでのプライベートキーストレージの設定がサポートされています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`PKCS11`		
`OpenSSLEngine`	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL 工	ファイルシステム上のファイルへのパスであることが必要です。

フィールド	説明	メモ
	ンジン `.so` ファイルへの絶対パス。	ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
`P11Provider`	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
`slotLabel`	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	

フィールド	説明	メモ
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

以下の設定プロパティもサポートされています。

フィールド	説明	メモ
`mqttMaxConnectionRetryInterval`	オプション。MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。デフォルト: `60`。
`managedRespawn`	オプション。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は `true` または `false` です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

フィールド	説明	メモ
`writeDirectory`	オプション。がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。
pidFileDirectory	Optional. は、プロセス ID (PID) をこのディレクトリに AWS IoT Greengrass 保存します。	デフォルト値は `/var/run` です。

Extended life versions

AWS IoT Greengrass Core ソフトウェアの次のバージョンは、延長寿命フェーズにあります。この情報は参照のみを目的として含まれています。

GGC v1.10

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "thingArn" : "arn:partition:iot:region:account-id:thing/core-thing-name",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600,
    "systemComponentAuthTimeout": 5000
  },
  "runtime" : {
    "maxWorkItemCount" : 1024,
    "maxConcurrentLimit" : 25,
    "lruSize": 25,
    "cgroup" : {
      "useSystemd" : "yes"
    }
  },
  "managedRespawn" : false,
  "crypto" : {
    "principals" : {
```

```
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key",
        "certificatePath" : "file:///greengrass/certs/hash.cert.pem"
      }
    },
    "caPath" : "file:///greengrass/certs/root.ca.pem"
  }
}
```

`config.json` ファイルは以下のプロパティをサポートしています。

coreThing

フィールド	説明	メモ
`caPath`	`/greengrass-root /certs` ディレクトリに対する AWS IoT ルート CA へのパス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`certPath`	`/greengrass-root /certs` ディレクトリからコアデバイス証明書への相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。

フィールド	説明	メモ
keyPath	/*greengrass-root* / certs ディレクトリから Core プライベートキーへの相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。
thingArn	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	Core の ARN は、 AWS IoT Greengrass コンソールの Core で検索するか、 aws greengrass get-core-definition-version CLI コマンドを実行して検索します。

フィールド	説明	メモ
`iotHost`	AWS IoT エンドポイント。	AWS IoT コンソールの「設定」で、または `aws iot describe-endpoint --endpoint-type iot:Data-ATS` CLI コマンドを実行してエンドポイントを見つけます。 このコマンドは Amazon Trust Services (ATS) エンドポイントを返します。詳細については、「サーバー認証」のドキュメントを参照してください。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。

フィールド	説明	メモ
`ggHost`	AWS IoT Greengrass エンドポイント。	これはホストのプレフィクスが greengrass で置き換えられた `iotHost` エンドポイント (例: `greengrass-ats.iot.`*`region`*`.amazonaws.com`) です。 AWS リージョン と同じ を使用します`iotHost`。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。
`iotMqttPort`	オプション。MQTT 通信に使用するポート番号 AWS IoT。	有効な値は 8883 または 443 です。デフォルト値は 8883 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
iotHttpPort	オプション。 AWS IoTへのHTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
ggMqttPort	オプション。ローカルネットワーク経由の MQTT 通信に使用するポート番号。	有効な値は 1024 ~ 65535 です。デフォルト値は 8883 です。詳細については、「the section called “ローカルメッセージング用の MQTT ポート”」を参照してください。
ggHttpPort	オプション。 AWS IoT Greengrass サービスへのHTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
keepAlive	オプション。MQTT の KeepAlive 期間 (秒単位)。	有効な範囲は 30 ~ 1200 秒です。デフォルト値は 600 です。
networkProxy	オプション。接続先のプロキシサーバーを定義するオブジェクト。	プロキシサーバーには HTTP または HTTPS を使用できます。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
`mqttOperationTimeout`	オプション。Greengrass コアが AWS IoT Core への MQTT 接続で発行、サブスクリプション、またはサブスクリプション解除オペレーションを完了するまでの時間 (秒単位)。	このプロパティは v1.10.2 AWS IoT Greengrass から使用できます。 デフォルト値は 5 です。最小値は 5 です。

runtime

フィールド	説明	メモ
`maxWorkItemCount`	オプション。Greengrass デーモンが一度に処理できる作業項目の最大数。この制限を超える作業項目は無視されます。 作業項目キューは、システムコンポーネント、ユーザー定義の Lambda 関数、コネクタによって共有されます。	デフォルト値は 1024 です。最大値は、デバイスのハードウェアによって制限されます。 この値を大きくすると、 が AWS IoT Greengrass 使用するメモリが増加します。コアが大量の MQTT メッセージトラフィックを受信することが予想される場合は、この値を増やすことができます。
`maxConcurrentLimit`	オプション。Greengrass デーモンで同時接続が可能な Lambda ワーカーの最大数。このパラメータは、整数で別の値を指定して上書きできます。	デフォルト値は 25 です。最小値は `lruSize` で定義されます。

フィールド	説明	メモ
`lruSize`	Optional. Defines the minimum value for `maxConcurrentLimit` .	The default value is 25.
`postStartHealthCheckTimeout`	Optional. The time (in milliseconds) after starting that the Greengrass daemon waits for the health check to finish.	The default timeout is 30 seconds (30000 ms).
`cgroup`		
`useSystemd`	Indicates whether your device uses `systemd`.	Valid values are ## or ###. Run the `check_ggc_dependencies` script in モジュール 1 to see if your device uses `systemd`.

crypto

`crypto` には、PKCS#11 を使用したハードウェアセキュリティモジュール (HSM) でのプライベートキーストレージをサポートするプロパティと、ローカルでのシークレットストレージをサポートするプロパティが含まれています。詳細については the section called “セキュリティプリンシパル”、the section called “ハードウェアセキュリティ統合”、および Core にシークレットをデプロイする を参照してください。HSM またはファイルシステムでのプライベートキーストレージの設定がサポートされています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。

フィールド	説明	メモ
		Note エンドポイントが証明書タイプに対応していることを確認してください。
PKCS11		
OpenSSLEngine	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン .so ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
P11Provider	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
slotLabel	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
slotUserPin	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
principals		

フィールド	説明	メモ
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

以下の設定プロパティもサポートされています。

フィールド	説明	メモ
`mqttMaxConnectionRetryInterval`	オプション。MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。デフォルト: `60`。
`managedRespawn`	オプション。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は `true` または `false` です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

フィールド	説明	メモ
writeDirectory	オプション。がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。

GGC v1.9

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "thingArn" : "arn:partition:iot:region:account-id:thing/core-thing-name",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
  "runtime" : {
    "cgroup" : {
      "useSystemd" : "yes"
    }
  },
  "managedRespawn" : false,
  "crypto" : {
    "principals" : {
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key",
        "certificatePath" : "file:///greengrass/certs/hash.cert.pem"
      }
    },
    "caPath" : "file:///greengrass/certs/root.ca.pem"
  }
}
```

config.json ファイルは以下のプロパティをサポートしています。

coreThing

フィールド	説明	メモ
`caPath`	`/greengrass-root /certs` ディレクトリに対する AWS IoT ルート CA へのパス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。 Note エンドポイントが証明書タイプに対応していることを確認してください。
`certPath`	`/greengrass-root /certs` ディレクトリからコアデバイス証明書への相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。
`keyPath`	`/greengrass-root /certs` ディレクトリから Core プライベートキーへの相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、`crypto` オブジェクトが存在する場合は無視されます。
`thingArn`	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	コンソールの Core AWS IoT Greengrass の下にある、または CLI コマンドを実行して、コアの ARN `aws greengrass get-core-`

フィールド	説明	メモ
		definition-version を見つけます。
iotHost	AWS IoT エンドポイント。	AWS IoT コンソールの「設定」で、または aws iot describe-endpoint --endpoint-type iot:Data-ATS CLI コマンドを実行してエンドポイントを見つけます。 このコマンドは Amazon Trust Services (ATS) エンドポイントを返します。詳細については、「サーバー認証」のドキュメントを参照してください。 Note エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。

フィールド	説明	メモ
`ggHost`	AWS IoT Greengrass エンドポイント。	これはホストのプレフィクスが greengrass で置き換えられた `iotHost` エンドポイント (例: `greengrass-ats.iot.`*`region`*`.amazonaws.com`) です。 AWS リージョン と同じ を使用します`iotHost`。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。 エンドポイントが AWS リージョンに対応していることを確認してください。
`iotMqttPort`	オプション。MQTT 通信に使用するポート番号 AWS IoT。	有効な値は 8883 または 443 です。デフォルト値は 8883 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
`iotHttpPort`	オプション。 AWS IoTへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`ggHttpPort`	オプション。 AWS IoT Greengrass サービスへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`keepAlive`	オプション。MQTT の `KeepAlive` 期間 (秒単位)。	有効な範囲は 30 ~ 1200 秒です。デフォルト値は `600` です。
`networkProxy`	オプション。接続先のプロキシサーバーを定義するオブジェクト。	プロキシサーバーには HTTP または HTTPS を使用できます。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

runtime

フィールド	説明	メモ
`maxConcurrentLimit`	オプション。Greengrass デーモンで同時接続が可能な Lambda ワーカーの最大	デフォルト値は 25 です。最小値は `lruSize` で定義されます。

フィールド	説明	メモ
	数。このパラメータは、整数で別の値を指定して上書きできます。	
`lruSize`	Optional. Defines the minimum value for `maxConcurrentLimit` .	The default value is 25.
`postStartHealthCheckTimeout`	Optional. The time (in milliseconds) after starting that the Greengrass daemon waits for the health check to finish.	The default timeout is 30 seconds (30000 ms).
`cgroup`		
`useSystemd`	Indicates whether your device uses `systemd`.	Valid values are ## or ###. Run the `check_ggc_dependencies` script in モジュール 1 to see if your device uses `systemd`.

crypto

`crypto` オブジェクトが v1.7.0 で追加されました。このオブジェクトでは、PKCS#11 を使用したハードウェアセキュリティモジュール (HSM) でのプライベートキーストレージをサポートするプロパティと、ローカルでのシークレットストレージをサポートするプロパティを導入しています。詳細についてはthe section called “セキュリティプリンシパル”、the section called “ハードウェアセキュリティ統合”、およびCore にシークレットをデプロイするを参照してください。HSM またはファイルシステムでのプライベートキーストレージの設定がサポートされています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形

フィールド	説明	メモ
		式のファイル URI である必要があります。 Note エンドポイントが証明書タイプに対応していることを確認してください。
PKCS11		
OpenSSLEngine	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン .so ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
P11Provider	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
slotLabel	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。

フィールド	説明	メモ
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

以下の設定プロパティもサポートされています。

フィールド	説明	メモ
`mqttMaxConnectionRetryInterval`	オプション。MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。デフォルト: `60`。
`managedRespawn`	オプション。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は `true` または `false` です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

フィールド	説明	メモ
writeDirectory	オプション。がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。

GGC v1.8

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "thingArn" : "arn:aws:iot:region:account-id:thing/core-thing-name",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
  "runtime" : {
    "cgroup" : {
      "useSystemd" : "yes"
    }
  },
  "managedRespawn" : false,
  "crypto" : {
    "principals" : {
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key",
        "certificatePath" : "file:///greengrass/certs/hash.cert.pem"
      }
    },
    "caPath" : "file:///greengrass/certs/root.ca.pem"
  }
}
```

config.json ファイルは以下のプロパティをサポートしています。

coreThing

フィールド	説明	メモ
caPath	/*greengrass-root* / certs ディレクトリに対する AWS IoT ルート CA へのパス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。 Note エンドポイントが証明書タイプに対応していることを確認してください。
certPath	/*greengrass-root* / certs ディレクトリからコアデバイス証明書への相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。
keyPath	/*greengrass-root* / certs ディレクトリから Core プライベートキーへの相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。
thingArn	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	Core の下の AWS IoT Greengrass コンソールで、または CLI コマンドを実行して、コアの ARN aws greengrass get-core-

フィールド	説明	メモ
		definition-version を見つけます。
iotHost	AWS IoT エンドポイント。	AWS IoT コンソールの「設定」で、または aws iot describe-endpoint --endpoint-type iot:Data-ATS CLI コマンドを実行してエンドポイントを見つけます。 このコマンドは Amazon Trust Services (ATS) エンドポイントを返します。詳細については、「サーバー認証」のドキュメントを参照してください。 Note エンドポイントが証明書タイプに対応していることを確認してください。エンドポイントが AWS リージョンに対応していることを確認してください。

フィールド	説明	メモ
`ggHost`	AWS IoT Greengrass エンドポイント。	これはホストのプレフィクスが greengrass で置き換えられた `iotHost` エンドポイント (例: `greengrass-ats.iot.`*`region`*`.amazonaws.com`) です。 AWS リージョン と同じ を使用します`iotHost`。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。エンドポイントが AWS リージョンに対応していることを確認してください。
`iotMqttPort`	オプション。MQTT 通信に使用するポート番号 AWS IoT。	有効な値は 8883 または 443 です。デフォルト値は 8883 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

フィールド	説明	メモ
`iotHttpPort`	オプション。 AWS IoTへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`ggHttpPort`	オプション。 AWS IoT Greengrass サービスへの HTTPS 接続を確立するために使用されるポート番号。	有効な値は 8443 または 443 です。デフォルト値は 8443 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
`keepAlive`	オプション。MQTT の `KeepAlive` 期間 (秒単位)。	有効な範囲は 30 ~ 1200 秒です。デフォルト値は `600` です。
`networkProxy`	オプション。接続先のプロキシサーバーを定義するオブジェクト。	プロキシサーバーには HTTP または HTTPS を使用できます。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

runtime

フィールド	説明	メモ
`cgroup`		

フィールド	説明	メモ
`useSystemd`	Indicates whether your device uses `systemd`.	Valid values are ## or # ##. Run the `check_ggc_dependencies` script in モジュール 1 to see if your device uses `systemd`.

crypto

`crypto` オブジェクトが v1.7.0 で追加されました。このオブジェクトでは、PKCS#11 を使用したハードウェアセキュリティモジュール (HSM) でのプライベートキーストレージをサポートするプロパティと、ローカルでのシークレットストレージをサポートするプロパティを導入しています。詳細についてはthe section called “セキュリティプリンシパル”、the section called “ハードウェアセキュリティ統合”、およびCore にシークレットをデプロイするを参照してください。HSM またはファイルシステムでのプライベートキーストレージの設定がサポートされています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`PKCS11`		
`OpenSSLEngine`	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL 工	ファイルシステム上のファイルへのパスであることが必要です。

フィールド	説明	メモ
	ンジン `.so` ファイルへの絶対パス。	ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
`P11Provider`	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
`slotLabel`	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	

フィールド	説明	メモ
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

以下の設定プロパティもサポートされています。

フィールド	説明	メモ
`mqttMaxConnectionRetryInterval`	オプション。MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。デフォルト: `60`。
`managedRespawn`	オプション。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は `true` または `false` です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

フィールド	説明	メモ
writeDirectory	オプション。がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。

GGC v1.7

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "thingArn" : "arn:aws:iot:region:account-id:thing/core-thing-name",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
  "runtime" : {
    "cgroup" : {
      "useSystemd" : "yes"
    }
  },
  "managedRespawn" : false,
  "crypto" : {
    "principals" : {
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key",
        "certificatePath" : "file:///greengrass/certs/hash.cert.pem"
      }
    },
    "caPath" : "file:///greengrass/certs/root.ca.pem"
  }
}
```

config.json ファイルは以下のプロパティをサポートしています。

coreThing

フィールド	説明	メモ
caPath	/*greengrass-root* / certs ディレクトリに対する AWS IoT ルート CA へのパス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。 Note エンドポイントが証明書タイプに対応していることを確認してください。
certPath	/*greengrass-root* / certs ディレクトリからコアデバイス証明書への相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。
keyPath	/*greengrass-root* / certs ディレクトリから Core プライベートキーへの相対パス。	1.7.0 より前のバージョンとの下位互換性を確保するため。このプロパティは、crypto オブジェクトが存在する場合は無視されます。
thingArn	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	Core の下の AWS IoT Greengrass コンソールで、または CLI コマンドを実行して、コアの ARN aws greengrass get-core-

フィールド	説明	メモ
		definition-version を見つけます。
iotHost	AWS IoT エンドポイント。	AWS IoT コンソールの「設定」で、または aws iot describe-endpoint --endpoint-type iot:Data-ATS CLI コマンドを実行してエンドポイントを見つけます。 このコマンドは Amazon Trust Services (ATS) エンドポイントを返します。詳細については、「サーバー認証」のドキュメントを参照してください。 Note エンドポイントが証明書タイプに対応していることを確認してください。エンドポイントが AWS リージョンに対応していることを確認してください。

フィールド	説明	メモ
ggHost	AWS IoT Greengrass エンドポイント。	これはホストのプレフィクスが greengrass で置き換えられた iotHost エンドポイント (例: greengrass-ats.iot. *region*.amazonaws.com) です。 AWS リージョン と同じ を使用しますiotHost。 Note エンドポイントが証明書タイプに対応していることを確認してください。エンドポイントが AWS リージョンに対応していることを確認してください。
iotMqttPort	オプション。MQTT 通信に使用するポート番号 AWS IoT。	有効な値は 8883 または 443 です。デフォルト値は 8883 です。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。
keepAlive	オプション。MQTT の KeepAlive 期間 (秒単位)。	有効な範囲は 30 ~ 1200 秒です。デフォルト値は 600 です。

フィールド	説明	メモ
`networkProxy`	オプション。接続先のプロキシサーバーを定義するオブジェクト。	プロキシサーバーには HTTP または HTTPS を使用できます。詳細については、「ポート 443 での接続またはネットワークプロキシを通じた接続」を参照してください。

runtime

フィールド	説明	メモ
`cgroup`		
`useSystemd`	Indicates whether your device uses `systemd`.	Valid values are ## or ###. Run the `check_ggc_dependencies` script in モジュール 1 to see if your device uses `systemd`.

crypto

v1.7.0 に追加された `crypto` オブジェクトでは、PKCS#11 を使用したハードウェアセキュリティモジュール (HSM) でのプライベートキーストレージをサポートするプロパティと、ローカルでのシークレットストレージをサポートするプロパティを導入しています。詳細については、the section called “ハードウェアセキュリティ統合”およびCore にシークレットをデプロイするを参照してください。HSM またはファイルシステムでのプライベートキーストレージの設定がサポートされています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形

フィールド	説明	メモ
		式のファイル URI である必要があります。 Note エンドポイントが証明書タイプに対応していることを確認してください。
PKCS11		
OpenSSLEngine	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン .so ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
P11Provider	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
slotLabel	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。

フィールド	説明	メモ
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

以下の設定プロパティもサポートされています。

フィールド	説明	メモ
`mqttMaxConnectionRetryInterval`	オプション。MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。デフォルト: `60`。
`managedRespawn`	オプション。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は `true` または `false` です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

フィールド	説明	メモ
writeDirectory	オプション。がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。

GGC v1.6

```
{
   "coreThing": {
       "caPath": "root-ca-pem",
       "certPath": "cloud-pem-crt",
       "keyPath": "cloud-pem-key",
       "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
       "iotHost": "host-prefix.iot.region.amazonaws.com",
       "ggHost": "greengrass.iot.region.amazonaws.com",
       "keepAlive": 600,
       "mqttMaxConnectionRetryInterval": 60
   },
   "runtime": {
       "cgroup": {
           "useSystemd": "yes|no"
       }
   },
   "managedRespawn": true,
   "writeDirectory": "/write-directory"
}
```

> **Note**
>
> AWS IoT Greengrass コンソールからデフォルトグループ作成オプションを使用する場合、config.jsonファイルはデフォルト設定を指定する動作状態でコアデバイスにデプロイされます。

config.json ファイルは以下のプロパティをサポートしています。

フィールド	説明	メモ
caPath	/*greengrass-root* / certs ディレクトリから AWS IoT ルート CA への相対パス。	/*greengrass-root* / certs にファイルを保存します。
certPath	/*greengrass-root* / certs ディレクトリに対する AWS IoT Greengrass コア証明書へのパス。	/*greengrass-root* / certs にファイルを保存します。
keyPath	/*greengrass-root* / certs ディレクトリに対する AWS IoT Greengrass コアプライベートキーへのパス。	/*greengrass-root* / certs にファイルを保存します。
thingArn	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	Core の ARN は、 AWS IoT Greengrass コンソールの Core で検索するか、 aws greengrass get-core-definition-version CLI コマンドを実行して検索します。
iotHost	AWS IoT エンドポイント。	これは、 設定の AWS IoT コンソールで、または aws iot describe-endpoint CLI コマンドを実行して検索します。
ggHost	AWS IoT Greengrass エンドポイント。	この値は greengrass.iot. *region*.amazonaws.com 形式を使用します。iotHost と同じリージョンを使用します。

フィールド	説明	メモ
keepAlive	MQTT の KeepAlive 期間 (秒単位)。	これはオプションの値です。デフォルト: 600。
mqttMaxConnectionRetryInterval	MQTT 接続間の最大間隔 (秒数) は、接続がドロップした場合に再試行します。	この値は符号なし整数として指定します。これはオプションの値です。デフォルト: 60。
useSystemd	デバイスが systemd を使用するかどうかを指定します。	有効な値は yes または no です。check_ggc_dependencies モジュール 1 で スクリプトを実行して、デバイスが systemd を使用するかどうかを確認します。
managedRespawn	オプションの無線通信 (OTA) 更新機能。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	有効な値は true または false です。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。
writeDirectory	がすべての読み取り/書き込みリソース AWS IoT Greengrass を作成する書き込みディレクトリ。	これはオプションの値です。詳細については、「の書き込みディレクトリを設定する AWS IoT Greengrass」を参照してください。

GGC v1.5

```
{
   "coreThing": {
       "caPath": "root-ca-pem",
       "certPath": "cloud-pem-crt",
       "keyPath": "cloud-pem-key",
```

```
        "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix.iot.region.amazonaws.com",
        "ggHost": "greengrass.iot.region.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes|no"
        }
    },
    "managedRespawn": true
}
```

config.json ファイルは /*greengrass-root*/config にあり、次のパラメータを含んでいます。

フィールド	説明	メモ
caPath	/*greengrass-root* /certs フォルダから AWS IoT ルート CA への相対パス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
certPath	/*greengrass-root* /certs フォルダに対する AWS IoT Greengrass コア証明書へのパス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
keyPath	/*greengrass-root* /certs フォルダに対する AWS IoT Greengrass コアプライベートキーへのパス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
thingArn	AWS IoT Greengrass コアデバイスを表す AWS IoT モノの Amazon リソースネーム (ARN)。	コンソールの Core AWS IoT Greengrass の下にある、または CLI コマンドを実行して、コアの ARN aws greengrass get-core-

フィールド	説明	メモ
		definition-version を見つけます。
iotHost	AWS IoT エンドポイント。	これは、 設定の AWS IoT コンソールで、または aws iot describe-endpoint コマンドを実行して検索します。
ggHost	AWS IoT Greengrass エンドポイント。	この値は greengrass.iot. *region*.amazonaws.com 形式を使用します。iotHost と同じリージョンを使用します。
keepAlive	MQTT の KeepAlive 期間 (秒単位)。	これはオプションの値です。デフォルト値は 600 秒です。
useSystemd	デバイスが systemd を使用するかどうかを指定します。	有効な値は yes または no です。check_ggc_dependencies モジュール 1 で スクリプトを実行して、デバイスが systemd を使用するかどうかを確認します。
managedRespawn	オプションの無線通信 (OTA) 更新機能。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

GGC v1.3

```
{
```

```
    "coreThing": {
        "caPath": "root-ca-pem",
        "certPath": "cloud-pem-crt",
        "keyPath": "cloud-pem-key",
        "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix.iot.region.amazonaws.com",
        "ggHost": "greengrass.iot.region.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes|no"
        }
    },
    "managedRespawn": true
}
```

`config.json` ファイルは `/greengrass-root/config` にあり、次のパラメータを含んでいます。

フィールド	説明	メモ
`caPath`	`/greengrass-root /certs` フォルダから AWS IoT ルート CA への相対パス。	ファイルを `/greengrass-root /certs` フォルダの下に保存します。
`certPath`	`/greengrass-root /certs` フォルダに対する AWS IoT Greengrass コア証明書へのパス。	ファイルを `/greengrass-root /certs` フォルダの下に保存します。
`keyPath`	`/greengrass-root /certs` フォルダに対する AWS IoT Greengrass コアプライベートキーへのパス。	ファイルを `/greengrass-root /certs` フォルダの下に保存します。
`thingArn`	AWS IoT Greengrass コアを表す AWS IoT モノの	この値は、 AWS IoT モノの定義の下の AWS IoT

フィールド	説明	メモ
	Amazon リソースネーム (ARN)。	Greengrass コンソールで確認できます。
iotHost	AWS IoT エンドポイント。	この値は、 AWS IoT コンソールの設定で確認できます。
ggHost	AWS IoT Greengrass エンドポイント。	この値は、コンソールのgreengrass.「 の設定 AWS IoT 」の先頭に追加されています。
keepAlive	MQTT の KeepAlive 期間 (秒単位)。	これはオプションの値です。デフォルト値は 600 秒です。
useSystemd	デバイスが systemd を使用するかどうかを示すバイナリフラグ。	有効な値は yes または no です。モジュール 1 の依存関係スクリプトを使用して、デバイスが systemd を使用しているかどうかを確認できます。
managedRespawn	オプションの無線通信 (OTA) 更新機能。OTA エージェントが更新前にカスタムコードを実行する必要があることを示します。	詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

GGC v1.1

```
{
   "coreThing": {
       "caPath": "root-ca-pem",
       "certPath": "cloud-pem-crt",
       "keyPath": "cloud-pem-key",
       "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
```

```
        "iotHost": "host-prefix.iot.region.amazonaws.com",
        "ggHost": "greengrass.iot.region.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes|no"
        }
    }
}
```

config.json ファイルは /*greengrass-root*/config にあり、次のパラメータを含んでいます。

フィールド	説明	メモ
caPath	/*greengrass-root* /certs フォルダから AWS IoT ルート CA への相対パス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
certPath	/*greengrass-root* /certs フォルダに対する AWS IoT Greengrass コア証明書へのパス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
keyPath	/*greengrass-root* /certs フォルダに対する AWS IoT Greengrass コアプライベートキーへのパス。	ファイルを /*greengrass-root* /certs フォルダの下に保存します。
thingArn	AWS IoT Greengrass コアを表す AWS IoT モノの Amazon リソースネーム (ARN)。	この値は、 AWS IoT モノの定義の下の AWS IoT Greengrass コンソールで確認できます。

フィールド	説明	メモ
iotHost	AWS IoT エンドポイント。	この値は、 AWS IoT コンソールの設定で確認できます。
ggHost	AWS IoT Greengrass エンドポイント。	この値は、 AWS IoT コンソールのgreengrass. 「 の設定」の先頭に追加されています。
keepAlive	MQTT の KeepAlive 期間 (秒単位)。	これはオプションの値です。デフォルト値は 600 秒です。
useSystemd	デバイスが systemd を使用するかどうかを示すバイナリフラグ。	有効な値は yes または no です。モジュール 1 の依存関係スクリプトを使用して、デバイスが systemd を使用しているかどうかを確認できます。

GGC v1.0

AWS IoT Greengrass Core v1.0 では、 config.jsonは にデプロイされます*greengrass-root*/configuration。

```
{
   "coreThing": {
       "caPath": "root-ca-pem",
       "certPath": "cloud-pem-crt",
       "keyPath": "cloud-pem-key",
       "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
       "iotHost": "host-prefix.iot.region.amazonaws.com",
       "ggHost": "greengrass.iot.region.amazonaws.com",
       "keepAlive": 600
   },
   "runtime": {
       "cgroup": {
```

```
            "useSystemd": "yes|no"
        }
    }
}
```

config.json ファイルは /*greengrass-root*/configuration にあり、次のパラメータを含んでいます。

フィールド	説明	メモ
caPath	/*greengrass-root* / configuration/certs フォルダから AWS IoT ルート CA への相対パス。	ファイルを /*greengrass-root* /configuration/certs フォルダの下に保存します。
certPath	/*greengrass-root* / configuration/certs フォルダに対する AWS IoT Greengrass コア証明書へのパス。	ファイルを /*greengrass-root* /configuration/certs フォルダの下に保存します。
keyPath	/*greengrass-root* / configuration/certs フォルダに対する AWS IoT Greengrass コアプライベートキーへのパス。	ファイルを /*greengrass-root* /configuration/certs フォルダの下に保存します。
thingArn	AWS IoT Greengrass コアを表す AWS IoT モノの Amazon リソースネーム (ARN)。	この値は、 AWS IoT Greengrass コンソールの hing AWS IoT の定義にあります。
iotHost	AWS IoT エンドポイント。	この値は、 AWS IoT コンソールの設定で確認できます。
ggHost	AWS IoT Greengrass エンドポイント。	この値は、 AWS IoT コンソールのgreengrass. 「

フィールド	説明	メモ
		の設定」の先頭に追加されています。
keepAlive	MQTT の KeepAlive 期間 (秒単位)。	これはオプションの値です。デフォルト値は 600 秒です。
useSystemd	デバイスが systemd を使用するかどうかを示すバイナリフラグ。	有効な値は yes または no です。モジュール 1 の依存関係スクリプトを使用して、デバイスが systemd を使用しているかどうかを確認できます。

サービスエンドポイントはルート CA 証明書タイプと一致する必要があります。

エンドポイント AWS IoT Core と AWS IoT Greengrass エンドポイントは、デバイス上のルート CA 証明書の証明書タイプに対応している必要があります。エンドポイントと証明書タイプが一致しない場合、デバイスと AWS IoT Core または の間で認証の試行は失敗します AWS IoT Greengrass。詳細については、AWS IoT デベロッパーガイドの「サーバー認証」を参照してください。

デバイスが、推奨の方法である Amazon Trust Services (ATS) ルート CA 証明書 を使用している場合、デバイス管理と検出データプレーンのオペレーションに ATS エンドポイントも使用する必要があります。ATS エンドポイントには、 AWS IoT Core エンドポイントの次の構文に示すように、atsセグメントが含まれます。

```
prefix-ats.iot.region.amazonaws.com
```

> **Note**
>
> 下位互換性のために、 AWS IoT Greengrass は現在、一部の でレガシー VeriSign ルート CA AWS リージョン証明書とエンドポイントをサポートしています。レガシーの VeriSign ルート CA 証明書を使用している場合、ATS エンドポイントを作成し、代わりに ATS ルート CA 証明書を使用することをお勧めします。それ以外の場合は、対応するレガシーエンドポイン

トを必ず使用します。詳細については、「Amazon Web Services 全般のリファレンス」の「サポートされているレガシーエンドポイント」を参照してください。

config.json のエンドポイント

Greengrass Core デバイスでは、エンドポイントは `config.json` ファイルの `coreThing` オブジェクトで指定されます。`iotHost` プロパティは AWS IoT Core エンドポイントを表します。`ggHost` プロパティは AWS IoT Greengrass エンドポイントを表します。次の例のスニペットでは、これらのプロパティは ATS エンドポイントを指定します。

```
{
  "coreThing" : {
    ...
    "iotHost" : "abcde1234uwxyz-ats.iot.us-west-2.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
    ...
  },
```

AWS IoT Core エンドポイント

適切な`--endpoint-type`パラメータを使用して aws iot describe-endpoint CLI コマンドを実行することで、 AWS IoT Core エンドポイントを取得できます。

- ATS 署名付きエンドポイントを返すには、以下を実行します。

  ```
  aws iot describe-endpoint --endpoint-type iot:Data-ATS
  ```

- 従来の VeriSign 署名付きエンドポイントを返すには、以下を実行します。

  ```
  aws iot describe-endpoint --endpoint-type iot:Data
  ```

AWS IoT Greengrass エンドポイント

AWS IoT Greengrass エンドポイントは、ホストプレフィックスが greengrass に置き換えられた`iotHost`エンドポイントです。例えば、ATS 署名付きエンドポイントは `greengrass-ats.iot.`*`region`*`.amazonaws.com` です。これにより、 AWS IoT Core エンドポイントと同じリージョンが使用されます。

ポート 443 での接続またはネットワークプロキシを通じた接続

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

Greengrass コアは、TLS クライアント認証で MQTT メッセージングプロトコル AWS IoT Core を使用して と通信します。慣例では、TLS を介した MQTT ではポート 8883 を使用します。ただし、セキュリティ対策として、制限の厳しい環境では一定範囲の TCP ポートに対するインバウンドトラフィックとアウトバウンドトラフィックを制限する場合があります。例えば、企業のファイアウォールでは HTTPS トラフィック用のポート 443 は開いても、あまり一般的ではないプロトコル用の他のポート (MQTT トラフィック用のポート 8883 など) は閉じる場合があります。他の制限の厳しい環境では、すべてのトラフィックに対して HTTP プロキシを経由してインターネットに接続することを義務付ける場合があります。

これらのシナリオで通信を有効にするために、 は次の設定 AWS IoT Greengrass を許可します。

- ポート 443 を介した TLS クライアント認証を使用する MQTT。ネットワークでポート 443 への接続を許可する場合は、デフォルトのポート 8883 ではなく ポート 443 を MQTT トラフィックに使用するように Core を設定できます。ポート 443 への直接接続またはネットワークプロキシサーバーを介した接続を使用できます。

  AWS IoT Greengrass は Application Layer Protocol Network (ALPN) TLS 拡張機能を使用して、この接続を有効にします。デフォルト設定と同様に、ポート 443 での TLS を介した MQTT では証明書ベースのクライアント認証を使用します。

  ポート 443 への直接接続を使用するように設定されている場合、コアは AWS IoT Greengrass ソフトウェアの無線通信 (OTA) 更新をサポートします。このサポートには AWS IoT Greengrass Core v1.9.3 以降が必要です。
- ポート 443 経由の HTTPS 通信。デフォルトではポート 8443 経由で HTTPS トラフィック AWS IoT Greengrass を送信しますが、ポート 443 を使用するように設定できます。
- ネットワークプロキシを介した接続。ネットワークプロキシサーバーを Greengrass コアに接続するための仲介役として設定できます。基本的な認証と HTTP/HTTPS プロキシのみがサポートされています。

  プロキシ設定は、環境変数の `http_proxy`、`https_proxy`、および `no_proxy` を通じてユーザー定義の Lambda 関数に渡されます。ユーザー定義の Lambda 関数は、渡されたこれらの設定を使用して、プロキシ経由で接続する必要があります。接続を行うために Lambda 関数によって使用される共通ライブラリ (boto3 や cURL など、および python `requests` パッケージ) は通常、

デフォルトでこれらの環境変数を使用します。Lambda 関数がこれらの同じ環境変数も指定する場合、上書き AWS IoT Greengrass しません。

> **Important**
>
> ネットワークプロキシを使用するように設定されている Greengrass Core は、OTA の更新をサポートしていません。

ポート 443 を介した MQTT を設定するには

この機能には AWS IoT Greengrass Core v1.7 以降が必要です。

この手順により、Greengrass コアが AWS IoT Coreと MQTT メッセージングにポート 443 を使用できるようになります。

1. 次のコマンドを実行して Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. coreThing オブジェクトで、iotMqttPort プロパティを追加し、値を **443** に設定します (以下の例を参照)。

```
{
    "coreThing" : {
        "caPath" : "root.ca.pem",
        "certPath" : "12345abcde.cert.pem",
        "keyPath" : "12345abcde.private.key",
        "thingArn" : "arn:aws:iot:us-west-2:123456789012:thing/core-thing-name",
        "iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
        "iotMqttPort" : 443,
        "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
        "keepAlive" : 600
    },
    ...
}
```

4. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

ポート 443 経由の HTTPS を設定するには

この機能には AWS IoT Greengrass Core v1.8 以降が必要です。

次の手順では、HTTPS 通信にポート 443 を使用するようにコアを設定します。

1. 次のコマンドを実行して Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. coreThing オブジェクトで、以下の例に示すように、iotHttpPort および ggHttpPort プロパティを追加します。

```
{
    "coreThing" : {
        "caPath" : "root.ca.pem",
        "certPath" : "12345abcde.cert.pem",
        "keyPath" : "12345abcde.private.key",
        "thingArn" : "arn:aws:iot:us-west-2:123456789012:thing/core-thing-name",
        "iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
        "iotHttpPort" : 443,
        "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
        "ggHttpPort" : 443,
        "keepAlive" : 600
    },
    ...
}
```

4. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

ネットワークプロキシを設定するには

この機能には AWS IoT Greengrass Core v1.7 以降が必要です。

この手順により、 AWS IoT Greengrass は HTTP または HTTPS ネットワークプロキシを介してインターネットに接続できます。

1. 次のコマンドを実行して Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. coreThing オブジェクトで、以下の例に示すように、networkProxy オブジェクトを追加します。

```
{
    "coreThing" : {
        "caPath" : "root.ca.pem",
        "certPath" : "12345abcde.cert.pem",
        "keyPath" : "12345abcde.private.key",
        "thingArn" : "arn:aws:iot:us-west-2:123456789012:thing/core-thing-name",
        "iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
        "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
        "keepAlive" : 600,
        "networkProxy": {
            "noProxyAddresses" : "http://128.12.34.56,www.mywebsite.com",
            "proxy" : {
                "url" : "https://my-proxy-server:1100",
                "username" : "Mary_Major",
                "password" : "pass@word1357"
            }
        }
    },
    ...
}
```

4. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
```

```
sudo ./greengrassd start
```

networkProxy オブジェクト

ネットワークプロキシに関する情報を指定するには、networkProxy オブジェクトを使用します。このオブジェクトには以下のプロパティがあります。

フィールド	説明
noProxyAddresses	オプション。プロキシの対象外となる IP アドレスやホスト名のカンマ区切りリスト。

proxy	接続先のプロキシ。プロキシには以下のプロパティがあります。
	• url。プロキシサーバーの URL (scheme://userinfo@host:port 形式)。
	  • scheme。スキーム。http または https を指定する必要があります。
	  • userinfo。オプション。ユーザー名とパスワードの情報。指定した場合、username フィールドと password フィールドは無視されます。
	  • host。プロキシサーバーのホスト名または IP アドレス。
	  • port。オプション。ポート番号。指定しない場合は、以下のデフォルト値が使用されます。
	  • http: 80
	  • https: 443
	• username。オプション。プロキシサーバーの認証に使用するユーザー名。
	• password。オプション。プロキシサーバーの認証に使用するパスワード。

エンドポイントの許可

Greengrass デバイスと AWS IoT Core または 間の通信は認証 AWS IoT Greengrass する必要があります。この認証は、登録された X.509 デバイス証明書と暗号化キーに基づきます。認証されたリクエストが追加の暗号化なしでプロキシを通過できるようにするには、次のエンドポイントを許可します。

エンドポイント	ポート	説明
`greengrass.`*`region`*`.amazonaws.com`	443	グループ管理用のコントロールプレーンオペレーションに使用されます。
`prefix``-ats.iot.`*`region`*`.amazonaws.com` or *`prefix`*`.iot.`*`region`*`.amazonaws.com`	MQTT: 8883 または 443 HTTPS: 8443 または 443	シャドウ同期など、デバイス管理用のデータプレーンオペレーションに使用されます。 コアデバイスとクライアントデバイスが Amazon Trust Services (推奨) ルート CA 証明書、レガシールート CA 証明書、またはその両方を使用しているかどうかに応

エンドポイント	ポート	説明
		じて、一方または両方のエンドポイントの使用を許可します。詳細については、「the section called “サービスエンドポイントは証明書タイプと一致する必要がある”」を参照してください。

エンドポイント	ポート	説明

`greengrass-ats.iot.`*`region`*`.amazonaws.com`
 	
or
 	
`greengrass.iot.`*`region`*`.amazonaws.com`
	8443 または 443	デバイス検出オペレーションのために使用されます。
		
		コアデバイスとクライアントデバイスが Amazon Trust Services (推奨) ルート CA 証明書、レガシールート CA 証明書、またはその両方を使用しているかどうかに応じて、一方または両方のエンドポイントの使用を許可します。詳細については、「the section called “サービスエンドポイントは証明書タイプと一致する必要がある”」を参照してください。

エンドポイント	ポート	説明
		Note ポート 443 に接続するクライアントは、Application Layer Protocol Negotiation (ALPN) の TLS 拡張機能を実装するとともに、`x-amzn-http-ca` を `ProtocolName` として

エンドポイント	ポート	説明
		`ProtocolNameList` に渡す必要があります。詳細については、「AWS IoT デベロッパーガイド」の「プロトコル」を参照してください。

エンドポイント	ポート	説明
`*.s3.amazonaws.com`	443	デプロイオペレーションと無線による更新に使用されます。エンドポイントプレフィックスは内部的に制御され、いつでも変更される可能性があるため、この形式には * 文字が含まれます。
`logs.region.amazonaws.com`	443	Greengrass グループが CloudWatch にログを書き込むように設定されている場合は必須です。

の書き込みディレクトリを設定する AWS IoT Greengrass

この機能は AWS IoT Greengrass Core v1.6 以降で使用できます。

デフォルトでは、 AWS IoT Greengrass Core ソフトウェアは がすべての読み取りおよび書き込みオペレーション AWS IoT Greengrass を実行する単一のルートディレクトリにデプロイされます。ただし、ディレクトリやファイルの作成など、すべての書き込みオペレーションに個別のディレクトリを使用する AWS IoT Greengrass ように を設定できます。この場合、 は 2 つの最上位ディレクトリ AWS IoT Greengrass を使用します。

- *greengrass-root* ディレクトリ。読み取り/書き込みのままにするか、あるいはオプションで読み取り専用にします。これには、 AWS IoT Greengrass Core ソフトウェアと、証明書や など、実行時に変更できないようにする必要があるその他の重要なコンポーネントが含まれていますconfig.json。
- 指定する書き込みディレクトリ。これには、ログ、状態情報やデプロイしたユーザー定義の Lambda 関数のような書き込み可能なコンテンツが含まれます。

この設定は次のようなディレクトリ構成になります。

Greengrass ルートディレクトリ

```
greengrass-root/
|-- certs/
|   |-- root.ca.pem
|   |-- hash.cert.pem
|   |-- hash.private.key
|   |-- hash.public.key
|-- config/
|   |-- config.json
|-- ggc/
|   |-- packages/
|       |-- package-version/
|           |-- bin/
|               |-- daemon
|           |-- greengrassd
|           |-- lambda/
|           |-- LICENSE/
|           |-- release_notes_package-version.html
|               |-- runtime/
|                   |-- java8/
|                   |-- nodejs8.10/
|                   |-- python3.8/
|   |-- core/
```

書き込みディレクトリ

```
write-directory/
|-- packages/
|   |-- package-version/
|       |-- ggc_root/
|       |-- rootfs_nosys/
```

```
|       |-- rootfs_sys/
|       |-- var/
|-- deployment/
|   |-- group/
|       |-- group.json
|   |-- lambda/
|   |-- mlmodel/
|-- var/
|   |-- log/
|   |-- state/
```

書き込みディレクトリを設定するには

1. 次のコマンドを実行して、 AWS IoT Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. 次の例に示すように、writeDirectory をパラメータとして追加し、ターゲットディレクトリへのパスを指定します。

```
{
    "coreThing": {
        "caPath": "root-CA.pem",
        "certPath": "hash.pem.crt",
        ...
    },
    ...
    "writeDirectory" : "/write-directory"
}
```

> **Note**
>
> writeDirectory 設定を必要に応じて更新することができます。設定が更新されると、 は次の起動時に新しく指定された書き込みディレクトリ AWS IoT Greengrass を使用しますが、以前の書き込みディレクトリからコンテンツを移行しません。

4. これで書き込みディレクトリは設定されため、オプションで *greengrass-root* ディレクトリを読み込み専用にできます。手順については、「Greengrass ルートディレクトリを読み取り専用にするには」を参照してください。

   それ以外の場合は、 AWS IoT Greengrass デーモンを起動します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

Greengrass ルートディレクトリを読み取り専用にするには

このステップは、Greengrass ルートディレクトリを読み取り専用にする場合にのみ実行します。書き込みディレクトリは、開始する前に設定する必要があります。

1. 必要なディレクトリにアクセス許可を付与します。

   a. `config.json` の所有者に読み取りおよび書き込み権限を与えます。

```
sudo chmod 0600 /greengrass-root/config/config.json
```

   b. ggc_user を certs とシステム Lambda ディレクトリの所有者とします。

```
sudo chown -R ggc_user:ggc_group /greengrass-root/certs/
sudo chown -R ggc_user:ggc_group /greengrass-root/ggc/packages/1.11.6/lambda/
```

   > **Note**
   >
   > デフォルトでは、ggc_user および ggc_group アカウントを使用してシステム Lambda 関数が実行されます。グループレベルのデフォルトのアクセス ID を設定して個別のアカウントを使用する場合は、代わりにそのユーザー (UID) およびグループ (GID) にアクセス権限を付与する必要があります。

2. 任意のメカニズムを使用して、*greengrass-root* ディレクトリを読み取り専用にします。

> **Note**
>
> *greengrass-root* ディレクトリを読み取り専用にする 1 つの方法は、ディレクトリを読み取り専用としてマウントすることです。ただし、無線 (OTA) による更新をマウントされたディレクトリの AWS IoT Greengrass Core ソフトウェアに適用するには、このディレクトリをまずアンマウントしてから、更新後に再度マウントする必要があります。上記の `umount` と `mount` オペレーションを `ota_pre_update` および `ota_post_update` スクリプトに追加できます。OTA 更新の詳細については、「the section called “Greengrass OTA Update Agent”」および「the section called “OTA 更新による管理された再生成”」を参照してください。

3. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

ステップ 1 のアクセス権限が正しく設定されていない場合、デーモンは起動しません。

MQTT 設定の構成

AWS IoT Greengrass 環境では、ローカルクライアントデバイス、Lambda 関数、コネクタ、およびシステムコンポーネントは相互に通信したり、 と通信したりできます AWS IoT Core。すべての通信は、エンティティ間の MQTT 通信を許可するサブスクリプションを管理するコアを通過します。

設定できる MQTT 設定の詳細については AWS IoT Greengrass、以下のセクションを参照してください。

- the section called “サービスのメッセージの品質”
- the section called “MQTT メッセージキュー”
- the section called “AWS IoT Coreを使用した MQTT 永続セッション”
- the section called “AWS IoTを使用した MQTT 接続用クライアント ID”
- ローカルメッセージング用の MQTT ポート
- the section called “との MQTT 接続でのパブリッシュ、サブスクライブ、サブスクライブ解除オペレーションのタイムアウト AWS クラウド”

> **Note**
>
> OPC-UA は、産業通信用の情報交換標準です。Greengrass コアで OPC-UA のサポートを実装するには、IoT SiteWise コネクタを使用できます。コネクタは、OPC-UA サーバーから のアセットプロパティに産業用デバイスデータを送信します AWS IoT SiteWise。

サービスのメッセージの品質

AWS IoT Greengrass は、設定と通信のターゲットと方向に応じて、サービス品質 (QoS) レベル 0 または 1 をサポートします。Greengrass コアは、 との通信用のクライアント AWS IoT Core および ローカルネットワークでの通信用のメッセージブローカーとして機能します。

MQTT および QoS の詳細については、MQTT ウェブサイトの「Getting Started」(はじめに) を参照してください。

との通信 AWS クラウド

- アウトバウンドメッセージが QoS 1 を使用

  コアは、QoS 1 を使用して AWS クラウド ターゲット宛てのメッセージを送信します。 AWS IoT Greengrass は、MQTT メッセージキューを使用してこれらのメッセージを処理します。メッセージ配信が によって確認されない場合 AWS IoT、メッセージは後で再試行されるようにスプールされます。キューがいっぱいになるとメッセージを再試行できません。メッセージ配信の確認を行うことで、断続的な接続によるデータ損失を最小限に抑えることができます。

  QoS QoS 1 AWS IoT を使用するアウトバウンドメッセージであるため、Greengrass コアがメッセージを送信できる最大レートは、コアと の間のレイテンシーによって異なります AWS IoT。コアがメッセージを送信するたびに、 がメッセージ AWS IoT を承認するまで待機して

から、次のメッセージを送信します。例えば、コアと の間のラウンドトリップ時間が 50 ミリ秒の場合、コア AWS リージョン は 1 秒あたり最大 20 個のメッセージを送信できます。コアが接続 AWS リージョン する を選択するときは、この動作を考慮してください。に大量の IoT データを取り込むには AWS クラウド、ストリームマネージャーを使用できます。

AWS クラウド ターゲット宛てのメッセージを保持できるローカルストレージキャッシュを設定する方法など、MQTT メッセージキューの詳細については、「」を参照してくださいthe section called “MQTT メッセージキュー”。

- インバウンドメッセージが QoS 0 (デフォルト) または QoS 1 を使用

  デフォルトでは、コアは AWS クラウド ソースからのメッセージに対して QoS 0 でサブスクライブします。永続セッションを有効にすると、コアは QoS 1 でサブスクライブします。これにより、断続的な接続によるデータ損失を最小限に抑えることができます。これらのサブスクリプションの QoS を管理するには、ローカルスプーラーシステムコンポーネントで永続性設定を設定します。

  コアが AWS クラウド ターゲットとの永続セッションを確立できるようにする方法などの詳細については、「」を参照してくださいthe section called “AWS IoT Coreを使用した MQTT 永続セッション”。

ローカルターゲットとの通信

すべてのローカル通信は QoS 0 を使用します。コアは、Greengrass Lambda 関数、コネクタ、またはクライアントデバイスであるローカルターゲットにメッセージを送信しようとします。メッセージの保存や、配信の確認は行われません。メッセージはコンポーネント間のどこにでもドロップできます。

> **Note**
>
> Lambda 関数間の直接通信では MQTT メッセージングは使用されませんが、動作は同じです。

クラウドターゲットの MQTT メッセージキュー

AWS クラウド ターゲット宛ての MQTT メッセージは、処理待ちのキューに入れられます。キュー状態のメッセージは先入れ先出し (FIFO) の順序で処理されます。メッセージが処理され、 に発行されると AWS IoT Core、メッセージはキューから削除されます。

デフォルトでは、Greengrass Core は AWS クラウド ターゲット宛ての未処理のメッセージをメモリに保存します。代わりにコアを設定して、未処理のメッセージをメモリあるいはローカルストレージキャッシュに保存できます。インメモリストレージとは異なり、ローカルストレージキャッシュはコア再起動の間 (グループのデプロイやデバイスの再起動後など) 保持できるため、 AWS IoT Greengrass は引き続きメッセージを処理できます。また、ストレージサイズを設定することもできます。

> **Warning**
>
> Greengrass Core は接続が失われると、オフライン状態が MQTT クライアントによって検出される前に発行操作を再試行するため、重複する MQTT メッセージがキューに作成される可能性があります。クラウドターゲットの MQTT メッセージが重複しないようにするには、コアの `keepAlive` の値を、`mqttOperationTimeout` の値の半分未満に設定します。詳細については、「AWS IoT Greengrass コア設定ファイル」を参照してください。

AWS IoT Greengrass は、スプーラーシステムコンポーネント (`GGCloudSpoolerLambda` 関数) を使用してメッセージキューを管理します。次の `GGCloudSpooler` 環境変数を使用して、ストレージ設定を設定できます。

- GG_CONFIG_STORAGE_TYPE。メッセージキューの場所。以下の値が有効です。
  - `FileSystem`。未処理のメッセージを物理コアデバイスのディスク上のローカルストレージキャッシュに保存します。コアが再起動すると、キュー状態のメッセージは処理のために維持されます。メッセージは処理された後に削除されます。
  - `Memory` (デフォルト)。未処理のメッセージをメモリに保存します。コアが再起動すると、キュー状態のメッセージは失われます。

    このオプションは、ハードウェア機能に制限があるデバイス向けに最適化されています。この設定を使用する場合を使用する場合、サービスの中断が低いときにグループのデプロイあるいはデバイスの再起動を行うことが推奨されます。

- GG_CONFIG_MAX_SIZE_BYTES。ストレージサイズ (バイト単位)。この値は、262144 (256 KB) 以上の負以外の整数にすることができます。サイズを小さくすると、 AWS IoT Greengrass Core ソフトウェアが起動できなくなります。デフォルトサイズは 2.5 MB です。サイズ制限に達すると、キューに入っている最も古いメッセージが新しいメッセージで置き換えられます。

> **Note**
>
> この機能は AWS IoT Greengrass Core v1.6 以降で使用できます。以前のバージョンでは、キューサイズが 2.5 MB のインメモリストレージを使用します。以前のバージョンでは、ストレージ設定を構成することはできません。

メッセージをローカルストレージにキャッシュするには

コアの再起動後もメッセージが保持されるように、 を設定 AWS IoT Greengrass して、メッセージをファイルシステムにキャッシュできます。これを行うには、`GGCloudSpooler` 関数がストレージタイプを `FileSystem` に設定する関数定義バージョンをデプロイします。 AWS IoT Greengrass API を使用してローカルストレージキャッシュを設定する必要があります。これをコンソールで実行することはできません。

次の手順では、`create-function-definition-version` CLI コマンドを使用して、キュー状態のメッセージをファイルシステムに保存するようにスプーラーを設定します。また、2.6 MB のキューサイズも設定します。

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   >
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

2. 出力からターゲットグループの `Id` 値と `LatestVersion` 値をコピーします。

3. 最新のグループバージョンを取得します。

   - *group-id* は、コピーした Id に置き換えます。
   - *latest-group-version-id* は、コピーした LatestVersion に置き換えます。

```
aws greengrass get-group-version \
--group-id group-id \
--group-version-id latest-group-version-id
```

4. 出力の Definition オブジェクトから、CoreDefinitionVersionArn をコピーし、FunctionDefinitionVersionArn を除く他のすべてのグループコンポーネントの ARN もコピーします。これらの値は、新しいグループバージョンを作成するときに使用します。
5. 出力の FunctionDefinitionVersionArn から、関数定義の ID をコピーします。ID は、次の例に示すように、ARN の functions セグメントに続く GUID です。

```
arn:aws:greengrass:us-west-2:123456789012:/greengrass/
definition/functions/bcfc6b49-beb0-4396-b703-6dEXAMPLEcu5/
versions/0f7337b4-922b-45c5-856f-1aEXAMPLEsf6
```

> **Note**
>
> または、create-function-definition コマンドを実行して関数定義を作成し、出力から ID をコピーすることもできます。

6. 関数定義に関数定義バージョンを追加します。

   - *function-definition-id* は、コピーした関数定義の Id に置き換えます。
   - *arbitrary-function-id* は、関数の名前 (**spooler-function** など) に置き換えます。
   - このバージョンで functions 配列に含める任意の Lambda 関数を追加します。get-function-definition-version コマンドを使用して、既存の関数定義バージョンから Greengrass Lambda 関数を取得できます。

> **Warning**
>
> GG_CONFIG_MAX_SIZE_BYTES の値が 262144 以上に指定されていることを確認します。サイズを小さくすると、 AWS IoT Greengrass Core ソフトウェアが起動できなくなります。

```
aws greengrass create-function-definition-version \
--function-definition-id function-definition-id \
--functions '[{"FunctionArn":
 "arn:aws:lambda:::function:GGCloudSpooler:1","FunctionConfiguration":
 {"Environment": {"Variables":
{"GG_CONFIG_MAX_SIZE_BYTES":"2621440","GG_CONFIG_STORAGE_TYPE":"FileSystem"}},"Executable":
 "spooler","MemorySize": 32768,"Pinned": true,"Timeout": 3},"Id": "arbitrary-
function-id"}]'
```

> **Note**
>
> で永続セッションをサポートする AWS IoT Coreように GG_CONFIG_SUBSCRIPTION_QUALITY 環境変数を以前に設定した場合は、この関数インスタンスに含めます。

7. 出力から関数定義バージョンの Arn をコピーします。
8. システムの Lambda 関数が含まれているグループバージョンを作成します。

   - *group-id* は、グループの Id に置き換えます。
   - *core-definition-version-arn* は、最新のグループバージョンからコピーした CoreDefinitionVersionArn に置き換えます。
   - *function-definition-version-arn* は、コピーした新しい関数定義バージョンの Arn に置き換えます。
   - 他のグループコンポーネントの ARN (SubscriptionDefinitionVersionArn や DeviceDefinitionVersionArn など) は、最新のグループバージョンからコピーしたものに置き換えます。
   - 使用されていないパラメータをすべて削除します。例えば、グループバージョンにリソースがない場合は、--resource-definition-version-arn を削除します。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
--function-definition-version-arn function-definition-version-arn \
--device-definition-version-arn device-definition-version-arn \
--logger-definition-version-arn logger-definition-version-arn \
--resource-definition-version-arn resource-definition-version-arn \
--subscription-definition-version-arn subscription-definition-version-arn
```

9. 出力から `Version` をコピーします。これは新しいグループバージョンの ID です。
10. 新しいグループバージョンでグループをデプロイします。

    - *`group-id`* は、コピーしたグループの `Id` に置き換えます。
    - *`group-version-id`* は、コピーした新しいグループバージョンの `Version` に置き換えます。

```
aws greengrass create-deployment \
--group-id group-id \
--group-version-id group-version-id \
--deployment-type NewDeployment
```

ストレージ設定を更新するには、 AWS IoT Greengrass API を使用して、更新された設定で関数を含む新しい`GGCloudSpooler`関数定義バージョンを作成します。次に、この関数定義バージョンを新規のグループバージョンに追加し (他のグループコンポーネントと一緒に)、このグループバージョンをデプロイします。デフォルト設定を復元する場合、`GGCloudSpooler` 関数を含まない関数定義バージョンをデプロイできます。

このシステム Lambda 関数は、コンソールでは表示されません。ただし、関数が最新のグループバージョンに追加されると、これはコンソールから行うデプロイに含まれます (API を使用して、その置き換えや削除を行う場合を除く)。

AWS IoT Coreを使用した MQTT 永続セッション

この機能は AWS IoT Greengrass Core v1.10 以降で使用できます。

Greengrass コアは、 AWS IoT メッセージブローカーとの永続セッションを確立できます。永続セッションは、コアがオフラインのときに送信されたメッセージをコアが受信できるようにする継続的な接続です。コアは、接続のクライアントです。

永続セッションでは、 AWS IoT メッセージブローカーは、接続中にコアが行うすべてのサブスクリプションを保存します。コアが切断されると、 AWS IoT メッセージブローカーは、QoS 1 として公開され、Lambda 関数やクライアントデバイスなどのローカルターゲット宛ての未確認メッセージと新しいメッセージを保存します。コアが再接続すると、永続セッションが再開され、 AWS IoT メッセージブローカーは 1 秒あたり最大 10 件のメッセージで保存されたメッセージをコアに送信します。永続セッションには、デフォルトで 1 時間の有効期限が設定されています。これは、メッセージブローカーがコアの切断を検出したときに開始されます。詳細については、「AWS IoT デベロッパーガイド」の「MQTT 永続セッション」を参照してください。

AWS IoT Greengrass は、スプーラーシステムコンポーネント (`GGCloudSpoolerLambda` 関数) を使用して、 をソース AWS IoT とするサブスクリプションを作成します。永続セッションを設定するには、次の `GGCloudSpooler` 環境変数を使用できます。

- GG_CONFIG_SUBSCRIPTION_QUALITY。をソース AWS IoT とするサブスクリプションの品質。以下の値が有効です。
  - `AtMostOnce` (デフォルト)。永続セッションを無効にします。サブスクリプションは QoS 0 を使用します。
  - `AtLeastOncePersistent`。永続セッションを有効にします。`CONNECT` メッセージの `cleanSession` フラグを `0` に設定し、QoS 1 でサブスクライブします。

    コアが受信する QoS 1 で発行されたメッセージは、Greengrass デーモンのメモリ内作業キューに到達することが保証されます。コアは、メッセージがキューに追加された後にメッセージを承認します。キューからローカルターゲット (Greengrass Lambda 関数、コネクタ、デバイスなど) への後続の通信は、QoS 0 で送信されます。この場合 AWS IoT Greengrass は、ローカルターゲットへの配信を保証しません。

    > **Note**
    >
    > maxWorkItemCount 設定プロパティを使用して、作業項目キューのサイズを制御できます。例えば、ワークロードで大量の MQTT トラフィックが必要な場合は、キューのサイズを増やすことができます。

永続セッションを有効にすると、コアは MQTT メッセージ交換のために少なくとも 1 つの追加の接続を開きます AWS IoT。詳細については、「the section called “AWS IoTを使用した MQTT 接続用クライアント ID”」を参照してください。

MQTT 永続セッションを設定するには

で永続セッションを使用する AWS IoT Greengrass ように を設定できます AWS IoT Core。これを行うには、`GGCloudSpooler` 関数がサブスクリプションの品質を `AtLeastOncePersistent` に設定する関数定義バージョンをデプロイします。この設定は、ソースとして AWS IoT Core (`cloud`) を持つすべてのサブスクリプションに適用されます。API を使用して永続セッション AWS IoT Greengrass を設定する必要があります。これをコンソールで実行することはできません。

次の手順では、`create-function-definition-version` CLI コマンドを使用して、永続セッションを使用するようにスプーラーを設定します。この手順では、既存のグループの最新グループバージョンの設定を更新しているものとします。

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   >
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

2. 出力からターゲットグループの `Id` 値と `LatestVersion` 値をコピーします。
3. 最新のグループバージョンを取得します。

- *group-id* は、コピーした Id に置き換えます。
- *latest-group-version-id* は、コピーした LatestVersion に置き換えます。

```
aws greengrass get-group-version \
--group-id group-id \
--group-version-id latest-group-version-id
```

4. 出力の Definition オブジェクトから、CoreDefinitionVersionArn をコピーし、FunctionDefinitionVersionArn を除く他のすべてのグループコンポーネントの ARN もコピーします。これらの値は、新しいグループバージョンを作成するときに使用します。
5. 出力の FunctionDefinitionVersionArn から、関数定義の ID をコピーします。ID は、次の例に示すように、ARN の functions セグメントに続く GUID です。

```
arn:aws:greengrass:us-west-2:123456789012:/greengrass/
definition/functions/bcfc6b49-beb0-4396-b703-6dEXAMPLEcu5/
versions/0f7337b4-922b-45c5-856f-1aEXAMPLEsf6
```

> **Note**
>
> または、create-function-definition コマンドを実行して関数定義を作成し、出力から ID をコピーすることもできます。

6. 関数定義に関数定義バージョンを追加します。

- *function-definition-id* は、コピーした関数定義の Id に置き換えます。
- *arbitrary-function-id* は、関数の名前 (**spooler-function** など) に置き換えます。
- このバージョンで functions 配列に含める任意の Lambda 関数を追加します。get-function-definition-version コマンドを使用して、既存の関数定義バージョンから Greengrass Lambda 関数を取得できます。

```
aws greengrass create-function-definition-version \
--function-definition-id function-definition-id \
--functions '[{"FunctionArn":
 "arn:aws:lambda:::function:GGCloudSpooler:1","FunctionConfiguration":
 {"Environment": {"Variables":
{"GG_CONFIG_SUBSCRIPTION_QUALITY":"AtLeastOncePersistent"}},"Executable":
```

```
 "spooler","MemorySize": 32768,"Pinned": true,"Timeout": 3},"Id": "arbitrary-
function-id"}]'
```

> **Note**
>
> ストレージ設定を定義するために GG_CONFIG_STORAGE_TYPE または GG_CONFIG_MAX_SIZE_BYTES 環境変数を設定していた場合は、この関数インスタンスにそれらの変数を含めます。

7. 出力から関数定義バージョンの Arn をコピーします。
8. システムの Lambda 関数が含まれているグループバージョンを作成します。
   - *group-id* は、グループの Id に置き換えます。
   - *core-definition-version-arn* は、最新のグループバージョンからコピーした CoreDefinitionVersionArn に置き換えます。
   - *function-definition-version-arn* は、コピーした新しい関数定義バージョンの Arn に置き換えます。
   - 他のグループコンポーネントの ARN (SubscriptionDefinitionVersionArn や DeviceDefinitionVersionArn など) は、最新のグループバージョンからコピーしたものに置き換えます。
   - 使用されていないパラメータをすべて削除します。例えば、グループバージョンにリソースがない場合は、--resource-definition-version-arn を削除します。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
--function-definition-version-arn function-definition-version-arn \
--device-definition-version-arn device-definition-version-arn \
--logger-definition-version-arn logger-definition-version-arn \
--resource-definition-version-arn resource-definition-version-arn \
--subscription-definition-version-arn subscription-definition-version-arn
```

9. 出力から Version をコピーします。これは新しいグループバージョンの ID です。
10. 新しいグループバージョンでグループをデプロイします。
    - *group-id* は、コピーしたグループの Id に置き換えます。

- *group-version-id* は、コピーした新しいグループバージョンの Version に置き換えます。

```
aws greengrass create-deployment \
--group-id group-id \
--group-version-id group-version-id \
--deployment-type NewDeployment
```

11. (オプション) コア設定ファイルの maxWorkItemCount プロパティを増やします。これにより、コアは増加した MQTT トラフィックとローカルターゲットとの通信を処理できます。

これらの設定変更でコアを更新するには、 AWS IoT Greengrass API を使用して、更新された設定で 関数を含む新しいGGCloudSpooler関数定義バージョンを作成します。次に、この関数定義バージョンを新規のグループバージョンに追加し (他のグループコンポーネントと一緒に)、このグループバージョンをデプロイします。デフォルト設定を復元する場合、GGCloudSpooler 関数を含まない関数定義バージョンを作成できます。

このシステム Lambda 関数は、コンソールでは表示されません。ただし、関数が最新のグループバージョンに追加されると、これはコンソールから行うデプロイに含まれます (API を使用して、その置き換えや削除を行う場合を除く)。

AWS IoTを使用した MQTT 接続用クライアント ID

この機能は AWS IoT Greengrass Core v1.8 以降で使用できます。

Greengrass コアは、シャドウ同期や証明書管理などのオペレーション AWS IoT Core のために、 との MQTT 接続を開きます。このような接続では、コアはコアのモノ名に基づいた予測可能なクライアント ID を生成します。予測可能なクライアント IDs は、 や AWS IoT Device Defender AWS IoT ライフサイクルイベントなどのモニタリング、監査、料金設定機能で使用できます。予測可能なクライアント ID を中心としたロジックも作成できます (例えば、証明書の属性に基づいたサブスクライブポリシーテンプレートなど)。

GGC v1.9 and later

2 つの Greengrass システムコンポーネントが MQTT 接続を開きます AWS IoT Core。これらのコンポーネントは次のパターンを使用して、接続用のクライアント ID を生成します。

Operation	クライアント ID のパターン
デプロイ	*core-thing-name* 例: `MyCoreThing` このクライアント ID は、接続、接続解除、ライフサイクルイベント通知へのサブスクライブあるいはサブスクライブ解除に使用します。
サブスクリプション	*core-thing-name* `-c`*n* 例: `MyCoreThing-c01` *n* は、00 からはじまり、新しい接続ごとに最大で 250 まで増加する整数です。接続数は、AWS IoT Core を使用してシャドウ状態を同期するデバイスの数 (グループごとに最大で 2,500 個のデバイス) およびグループ内のソースとしての `cloud` のサブスクリプション数 (グループごとに最大で 10,000) によって決定されます。 スプーラーシステムコンポーネントは AWS IoT Core に接続し、クラウドソースまたはターゲットとサブスクリプションのメッセージを交換します。スプーラーは、 AWS IoT Core とローカルシャドウサービスおよびデバイス証明書マネージャーとの間のメッセージ交換のプロキシとしても機能します。

1 グループあたりの MQTT 接続の数を計算するには、次の式を使用します。

```
number of MQTT connections per group = number of connections for
Deployment Agent + number of connections for Subscriptions
```

各パラメータは次のようになります。

- デプロイエージェントの接続数 = 1。
- サブスクリプションの接続数 = `(2 subscriptions for supporting certificate generation + number of MQTT topics in AWS IoT Core + number of device shadows synced) / 50`。
  - ここで、 `50` = がサポート AWS IoT Core できる接続あたりのサブスクリプションの最大数。

> **Note**
>
> でサブスクリプションの永続セッションを有効にすると AWS IoT Core、コアは永続セッションで使用する追加の接続を少なくとも 1 つ開きます。システムコンポーネントは永続セッションをサポートしていないため、その接続を共有することはできません。

MQTT 接続の数を減らしコストを削減するには、ローカル Lambda 関数を使用してデータをエッジに集約します。集約したデータを AWS クラウドに送信することで、その結果、 で使用する MQTT トピックが少なくなります AWS IoT Core。詳細については、AWS IoT Greengrass 料金を参照してください。

GGC v1.8

複数の Greengrass システムコンポーネントによって、 AWS IoT Coreを使用した MQTT 接続が開きます。これらのコンポーネントは次のパターンを使用して、接続用のクライアント ID を生成します。

Operation	クライアント ID のパターン
デプロイ	*`core-thing-name`* 例: `MyCoreThing` このクライアント ID は、接続、接続解除、ライフサイクルイベント通知へのサブスクライブあるいはサブスクライブ解除に使用します。
との MQTT メッセージ交換 AWS IoT Core	*`core-thing-name`* `-spr`

Operation	クライアント ID のパターン
	例: `MyCoreThing-spr`
シャドウ同期	*`core-thing-name`* `-s`*`nn`* 例: `MyCoreThing-s01` *`nn`* は、00 からはじまり、新しい接続ごとに最大で 03 まで増加する整数です。接続数は、シャドウ状態を と同期するデバイス (グループあたり最大 200 デバイス) の数 AWS IoT Core （ 接続あたり最大 50 サブスクリプション) によって決まります。
デバイス証明書管理	*`core-thing-name`* `-dcm` 例: `MyCoreThing-dcm`

> **Note**
>
> 同時接続で使用されるクライアント ID の重複は、無限の接続 - 切断ループを引き起こすことがあります。これは、別のデバイスが接続でクライアント ID をコアデバイス名に使用するようにハードコードされた場合に発生します。詳細については、このトラブルシューティングステップを参照してください。

また、Greengrass デバイスは AWS IoT Device Managementのフリートインデックス作成サービスと完全に統合されます。これにより、デバイス属性、シャドウ状態、およびクラウド内の接続状態に基づいてデバイスにインデックスを付けて検索することができます。例えば、Greengrass デバイスはモノの名前をクライアント ID として使用する少なくとも 1 つの接続を確立するため、デバイス接続のインデックス作成を使用して、現在 AWS IoT Coreに接続または切断されている Greengrass デバイスを検出できます。詳細については、「AWS IoT デベロッパーガイド」の「フリートインデックス作成サービス」を参照してください。

ローカルメッセージング用の MQTT ポートの設定

この機能には AWS IoT Greengrass Core v1.10 以降が必要です。

Greengrass コアは、ローカルの Lambda 関数、コネクタ、クライアントデバイス間の MQTT メッセージングのローカルメッセージブローカーとして機能します。デフォルトでは、コアはローカルネットワーク上の MQTT トラフィックにポート 8883 を使用します。ポート 8883 で動作する他のソフトウェアとの競合を避けるために、ポートを変更することもできます。

コアがローカル MQTT トラフィックに使用するポート番号を設定するには

1. 次のコマンドを実行して Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. coreThing オブジェクトに ggMqttPort プロパティを追加し、その値として使用するポート番号を設定します。有効な値は 1024 ~ 65535 です。次の例では、ポート番号を 9000 に設定します。

```
{
    "coreThing" : {
        "caPath" : "root.ca.pem",
        "certPath" : "12345abcde.cert.pem",
        "keyPath" : "12345abcde.private.key",
        "thingArn" : "arn:aws:iot:us-west-2:123456789012:thing/core-thing-name",
        "iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
        "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
        "ggMqttPort" : 9000,
        "keepAlive" : 600
    },
    ...
}
```

4. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

5. コアで自動 IP 検出が有効になっている場合、設定は完了です。

   自動 IP 検出が有効になっていない場合は、コアの接続情報を更新する必要があります。これにより、クライアントデバイスは検出操作中に正しいポート番号を受け取り、コア接続情報を取得できるようになります。 AWS IoT コンソールまたは AWS IoT Greengrass API を使用して、コ

ア接続情報を更新できます。この手順では、ポート番号のみを更新します。コアのローカル IP アドレスは同じままです。

コアの接続情報を更新するには (コンソール)

1. グループ設定ページで、Greengrass コアを選択します。
2. コアの詳細ページで、[MQTT ブローカーエンドポイント] タブを選択します。
3. [エンドポイントを管理] を選択し、続いて [エンドポイントの追加] を選択します。
4. 現在のローカル IP アドレスと新しいポート番号を入力します。次の例では、IP アドレス `192.168.1.8` のポート番号 `9000` を設定します。
5. 古いエンドポイントを削除し、[更新] を選択します。

コアの接続情報を更新するには (API)

- UpdateConnectivityInfo アクションを使用します。次の例では、 AWS CLI で `update-connectivity-info` を使用して IP アドレス `192.168.1.8` のポート番号 `9000` を設定します。

```
aws greengrass update-connectivity-info \
    --thing-name "MyGroup_Core" \
    --connectivity-info "[{\"Metadata\":\"\",\"PortNumber\":9000,
\"HostAddress\":\"192.168.1.8\",\"Id\":\"localIP_192.168.1.8\"},{\"Metadata
\":\"\",\"PortNumber\":8883,\"HostAddress\":\"127.0.0.1\",\"Id\":
\"localhost_127.0.0.1_0\"}]"
```

> **Note**
>
> また、コアが MQTT メッセージングに使用するポートを設定することもできます AWS IoT Core。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。

との MQTT 接続でのパブリッシュ、サブスクライブ、サブスクライブ解除オペレーションのタイムアウト AWS クラウド

この機能は v1.10.2 AWS IoT Greengrass 以降で使用できます。

Greengrass コアが AWS IoT Coreへの MQTT 接続で発行、サブスクリプション、またはサブスクリプション解除オペレーションを完了するまでの時間 (秒単位) を設定できます。帯域幅の制約や待ち時間が長いためオペレーションがタイムアウトする場合は、この設定を調整する必要があります。config.json ファイルでこの設定を構成するには、`coreThing` オブジェクトの `mqttOperationTimeout` プロパティを追加または変更します。例:

```
{
  "coreThing": {
    "mqttOperationTimeout": 10,
    "caPath": "root-ca.pem",
    "certPath": "hash.cert.pem",
    "keyPath": "hash.private.key",
    ...
  },
  ...
}
```

デフォルトのタイムアウトは 5 秒です。最小タイムアウトは 5 秒です。

自動 IP 検出のアクティブ化

Greengrass グループ内の AWS IoT Greengrass クライアントデバイスが Greengrass コアを自動的に検出できるように を設定できます。この検出設定を有効にした場合、コアは自身の IP アドレスの変更を監視します。アドレスが変更されると、コアは更新後のアドレスのリストを発行します。これらのアドレスは、コアと同じ Greengrass グループに属するクライアントデバイスで使用できます。

> **Note**
>
> クライアントデバイスの AWS IoT ポリシーは、デバイスがコアの接続情報を取得できるようにするアクセス`greengrass:Discover`許可を付与する必要があります。このポリシーステートメントの詳細については、「the section called “検出の認可”」を参照してください。

AWS IoT Greengrass コンソールからこの機能を有効にするには、Greengrass グループを初めてデプロイするときに自動検出を選択します。グループ設定ページでこの機能を有効または無効にするには、[Lambda 関数] タブで、[IP ディテクター] を選択します。[MQTT エンドポイントを自動的に検出して上書きする] が選択されている場合は、IP の自動検出が有効になります。

AWS IoT Greengrass API で自動検出を管理するには、`IPDetector`システム Lambda 関数を設定する必要があります。以下の手順では、create-function-definition-version CLI コマンドを使用して Greengrass Core の自動検出を設定する方法を説明しています。

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   >
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

2. 出力からターゲットグループの `Id` 値と `LatestVersion` 値をコピーします。
3. 最新のグループバージョンを取得します。

   - *`group-id`* は、コピーした `Id` に置き換えます。
   - *`latest-group-version-id`* は、コピーした `LatestVersion` に置き換えます。

   ```
   aws greengrass get-group-version \
   --group-id group-id \
   --group-version-id latest-group-version-id
   ```

4. 出力の `Definition` オブジェクトから、`CoreDefinitionVersionArn` をコピーし、`FunctionDefinitionVersionArn` を除く他のすべてのグループコンポーネントの ARN もコピーします。これらの値は、新しいグループバージョンを作成するときに使用します。
5. 出力の `FunctionDefinitionVersionArn` から、関数定義の ID と関数定義のバージョンをコピーします。

```
arn:aws:greengrass:region:account-id:/greengrass/groups/function-definition-id/
versions/function-definition-version-id
```

> **Note**
>
> 必要に応じて、create-function-definition コマンドを実行して関数定義を作成し、出力から ID をコピーすることもできます。

6. get-function-definition-version コマンドを使用して、現在の定義状態を取得します。コピーした関数定義の *function-definition-id* を使用します。例えば、*4d941bc7-92a1-4f45-8d64-EXAMPLEf76c3* です。

```
aws greengrass get-function-definition-version
--function-definition-id function-definition-id
--function-definition-version-id function-definition-version-id
```

表示された関数設定を書き留めておきます。現在の定義設定が失われるのを防ぐために、新しい関数定義バージョンを作成するときにこれらを含める必要があります。

7. 関数定義に関数定義バージョンを追加します。

   - *function-definition-id* は、コピーした関数定義の Id に置き換えます。例えば、*4d941bc7-92a1-4f45-8d64-EXAMPLEf76c3* です。
   - *arbitrary-function-id* は、関数の名前 (**auto-detection-function** など) に置き換えます。
   - このバージョンに含めるすべての Lambda 関数を functions 配列に追加します (前の手順でリストしたものなど)。

```
aws greengrass create-function-definition-version \
--function-definition-id function-definition-id \
--functions
 '[{"FunctionArn":"arn:aws:lambda:::function:GGIPDetector:1","Id":"arbitrary-
function-id","FunctionConfiguration":
{"Pinned":true,"MemorySize":32768,"Timeout":3}}]'\
--region us-west-2
```

8. 出力から関数定義バージョンの Arn をコピーします。

9. システムの Lambda 関数が含まれているグループバージョンを作成します。

   - *group-id* は、グループの Id に置き換えます。
   - *core-definition-version-arn* は、最新のグループバージョンからコピーした CoreDefinitionVersionArn に置き換えます。
   - *function-definition-version-arn* は、コピーした新しい関数定義バージョンの Arn に置き換えます。
   - 他のグループコンポーネントの ARN (SubscriptionDefinitionVersionArn や DeviceDefinitionVersionArn など) は、最新のグループバージョンからコピーしたものに置き換えます。
   - 使用されていないパラメータをすべて削除します。例えば、グループバージョンにリソースがない場合は、--resource-definition-version-arn を削除します。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
--function-definition-version-arn function-definition-version-arn \
--device-definition-version-arn device-definition-version-arn \
--logger-definition-version-arn logger-definition-version-arn \
--resource-definition-version-arn resource-definition-version-arn \
--subscription-definition-version-arn subscription-definition-version-arn
```

10. 出力から Version をコピーします。これは新しいグループバージョンの ID です。
11. 新しいグループバージョンでグループをデプロイします。

   - *group-id* は、コピーしたグループの Id に置き換えます。
   - *group-version-id* は、コピーした新しいグループバージョンの Version に置き換えます。

```
aws greengrass create-deployment \
--group-id group-id \
--group-version-id group-version-id \
--deployment-type NewDeployment
```

Greengrass コアの IP アドレスを手動で入力する場合は、`IPDetector` 関数を含まない別の関数定義を使用してこのチュートリアルを完了できます。これにより、検出関数が Greengrass コアの IP アドレスを見つけて自動的に入力することを防ぎます。

このシステム Lambda 関数は、Lambda コンソールでは表示されません。この関数が最新のグループバージョンに追加された後は、コンソールから行うデプロイに含まれます (API を使用して、その置き換えや削除を行う場合を除く)。

Init システムを設定して Greengrass デーモンを開始する

起動時に Greengrass デーモンを開始するように init システムを設定することは、特にデバイスの大規模なフリートを管理する場合に推奨されます。

> **Note**
>
> `apt` を使用して AWS IoT Greengrass Core ソフトウェアをインストールした場合は、systemd スクリプトを使用して起動時の起動を有効にできます。詳細については、「the section called “systemd スクリプトを使用した Greengrass デーモンのライフサイクルの管理”」を参照してください。

init システムにはさまざまな種類があり (initd、systemd、および SystemV など)、同様の設定パラメータが使用されます。次の例は、systemd のサービスファイルです。Greengrassd (Greengrass を開始するために使用される) は Greengrass デーモンプロセスを生成するため、`Type` パラメータは `forking` に設定されます。また、Greengrass が失敗状態になったときに systemd が Greengrass を再起動するように、`Restart` パラメータは `on-failure` に設定されます。

> **Note**
>
> お使いのデバイスが systemd を使用しているかを確認するには、モジュール 1 で説明されているように `check_ggc_dependencies` スクリプトを実行します。次に、systemd を使用するには、`useSystemd``config.json` の パラメータが `yes` に設定されていることを確認します。

```
[Unit]
Description=Greengrass Daemon
```

```
[Service]
Type=forking
PIDFile=/var/run/greengrassd.pid
Restart=on-failure
ExecStart=/greengrass/ggc/core/greengrassd start
ExecReload=/greengrass/ggc/core/greengrassd restart
ExecStop=/greengrass/ggc/core/greengrassd stop

[Install]
WantedBy=multi-user.target
```

関連情報

- とは AWS IoT Greengrass
- the section called “サポートされているプラットフォームと要件”
- の開始方法 AWS IoT Greengrass
- the section called “グループオブジェクトモデルの概要”
- the section called “ハードウェアセキュリティ統合”

AWS IoT Greengrass Version 1 メンテナンスポリシー

この AWS IoT Greengrass V1 メンテナンスポリシーを使用して、 AWS IoT Greengrass V1 サービスと AWS IoT Greengrass Core ソフトウェア v1.x のさまざまなレベルのメンテナンスと更新を理解します。

トピック

- AWS IoT Greengrass バージョニングスキーム
- AWS IoT Greengrass Core ソフトウェアのメジャーバージョンのライフサイクルフェーズ
- AWS IoT Greengrass Core ソフトウェアのメンテナンスポリシー
- 非推奨スケジュール
- Greengrass コアデバイス上の AWS Lambda 関数のサポートポリシー
- の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1
- メンテナンススケジュールの終了

AWS IoT Greengrass バージョニングスキーム

AWS IoT Greengrass は Core ソフトウェアのセマンティックバージョニング AWS IoT Greengrass を使用します。セマンティックバージョンは、major.minor.patch という番号体系に従います。メジャーバージョンでは、以前のメジャーバージョンと下位互換性がない機能および API の変更の場合に番号が大きくなります。マイナーバージョンでは、新しい下位互換機能を追加するリリースの場合に番号が大きくなります。パッチバージョンでは、セキュリティパッチまたはバグ修正の場合に番号が大きくなります。最初のメジャーリリース v1.0.0 以降、 AWS IoT Greengrass は AWS IoT Greengrass Core ソフトウェア v1.x の 11 のマイナーバージョンをリリースしました。v1.11.6 は最新のリリースです。 AWS IoT Greengrass Core ソフトウェアを利用可能な最新バージョンに更新して、新機能、機能強化、バグ修正を活用することをお勧めします。

2020 年 12 月、 は最初のメジャーバージョン更新を AWS IoT Greengrass リリースしました。この更新には、 AWS IoT Greengrass V2 サービスと AWS IoT Greengrass Core ソフトウェアのバージョン 2.0.3 が含まれていました。新しいアプリケーションでは、 AWS IoT Greengrass Version 2 と AWS IoT Greengrass Core ソフトウェア v2.x を使用することを強くお勧めします。バージョン 2 には新機能が追加され、V1 の主要機能をすべて含み、追加のプラットフォームと大規模なデバイス群に対する継続的なデプロイメントをサポートします。詳細については、「 とは AWS IoT Greengrass V2」を参照してください。

AWS IoT Greengrass Core ソフトウェアのメジャーバージョンのライフサイクルフェーズ

AWS IoT Greengrass Core ソフトウェアの各メジャーバージョンには、次の 3 つのシーケンシャルライフサイクルフェーズがあります。各ライフサイクルフェーズでは、初期リリース日以降の一定期間、さまざまなレベルのメンテナンスが提供されます。

- リリースフェーズ – 次の更新をリリース AWS IoT Greengrass できます。
  - 新機能や既存機能の拡張を提供するマイナーバージョンの更新
  - セキュリティパッチやバグ修正を提供するパッチバージョンの更新
- メンテナンスフェーズ – セキュリティパッチとバグ修正を提供するパッチバージョンの更新をリリース AWS IoT Greengrass する場合があります。メンテナンスフェーズ中に、新機能や既存の機能の強化はリリース AWS IoT Greengrass されません。
- 延長ライフフェーズ – AWS IoT Greengrass 機能、既存の機能の強化、セキュリティパッチ、バグ修正を提供する更新はリリースされません。ただし、 AWS クラウド エンドポイントと API オペレーションは引き続き利用可能であり、AWS IoT Greengrass サービスレベルアグリーメントに従って動作します。 AWS IoT Greengrass Core ソフトウェア v1.x を実行するデバイスは、引き続き に接続 AWS クラウド して動作できます。

のメジャーバージョンで延長ライフフェーズが終了すると AWS IoT Greengrass、 AWS クラウド エンドポイントと API オペレーションは廃止され、使用できなくなります。 AWS IoT Greengrass Core ソフトウェア v1.x を実行するデバイスは、運用する AWS クラウド サービスに接続できません。

AWS IoT Greengrass Core ソフトウェアのメンテナンスポリシー

AWS IoT Greengrass Core ソフトウェア v1.x は、2023 年 6 月 30 日に延長ライフフェーズに入りました。この日以降、 AWS IoT Greengrass Core ソフトウェア v1.x は、追って通知があるまで延長存続フェーズに残ります。

AWS IoT Greengrass Core ソフトウェア v2.x は現在リリースフェーズにあり、今後通知があるまでリリースフェーズに残ります。 AWS IoT Greengrass は Core ソフトウェア v2.x に新機能と機能強化を追加し AWS IoT Greengrass 続けます。例えば、 AWS IoT Greengrass Core ソフトウェアの v2.5.0 で Windows サポートを AWS IoT Greengrass リリース AWS IoT Greengrass しました。 は、リリース日から少なくとも 1 年間、すべてのマイナーバージョンの AWS IoT Greengrass Core v2.x

のセキュリティパッチとバグ修正をリリースします。詳細については、「AWS IoT Greengrass V2の新機能」を参照してください。

メンテナンスフェーズのスケジュール

2023 年 6 月 30 日、 AWS IoT Greengrass Core ソフトウェア v1.11.x のメンテナンスフェーズは終了しました。2022 年 3 月 31 日、 AWS IoT Greengrass Core ソフトウェア v1.10.x のメンテナンスフェーズは終了しました。メンテナンスフェーズは、特定の AWS IoT Greengrass Core ソフトウェア v1.x アーティファクトと機能について、これらの日付より前に終了します。詳細については、「メンテナンススケジュールの終了」を参照してください。

AWS サポート プランがある場合、 AWS IoT Greengrass Core ソフトウェア v1.x のメンテナンスフェーズは AWS サポート プランには影響しません。メンテナンスフェーズが終了した後も、 AWS サポート チケットを引き続き開くことができます。ご質問やご不明な点がある場合は、 AWS サポート の連絡先に問い合わせるか、または AWS IoT Greengrass タグを使用して AWS re: Post に質問してください。

非推奨スケジュール

現在、 AWS IoT Greengrass Core ソフトウェア v1.x のサポートを停止する予定はありません。AWS IoT Greengrass V1 エンドポイントと API オペレーションは、今後お知らせするまでに引き続き使用できます。 AWS IoT Greengrass Core ソフトウェア v1.11.6 は、2023 年 6 月 30 日に延長ライフフェーズに入りました。このフェーズでは、 AWS IoT Greengrass Core ソフトウェア v1.x を実行するデバイスは、追って通知があるまで AWS IoT Greengrass V1 サービスに接続して動作し続けることができます。

が将来サポート AWS IoT Greengrass V1 を停止する場合、 AWS IoT Greengrass はこれが発生する前に 12 か月前に通知します。これにより、 AWS IoT Greengrass V2 および AWS IoT Greengrass Core ソフトウェア v2.x を使用するアプリケーションの更新を計画できます。アプリケーションを V2 に更新する方法の詳細については、「 から V2 AWS IoT Greengrass V1 に移動する」を参照してください。

Greengrass コアデバイス上の AWS Lambda 関数のサポートポリシー

AWS IoT Greengrass を使用すると、IoT デバイスで AWS Lambda 関数を実行できます。は、Lambda ランタイムのサポートを決定するサポートポリシーとタイムライン AWS Lambda を提供します AWS IoT Greengrass。Lambda ランタイムがサポートフェーズの終了に達すると、 はその

ランタイムのサポート AWS IoT Greengrass も終了します。詳細については、「AWS Lambda デベロッパーガイド」の「ランタイムの非推奨化に関するポリシー」を参照してください。

Lambda ランタイムがサポート終了に達すると、そのランタイムを使用する Lambda 関数を作成または更新することはできません。ただし、これらの Lambda 関数を引き続き Greengrass コアデバイスにデプロイし、デプロイされた Lambda 関数を呼び出すことができます。このポリシーは、 AWS IoT Greengrass V2にも適用されます。

の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1

AWS IoT Device Tester (IDT) for AWS IoT Greengrass V1 を使用すると、 AWS IoT Greengrass Device AWS Partner Catalog に含めるデバイスを検証して認定できます。2022 年 4 月 4 日現在、の AWS IoT デバイステスター (IDT) は署名付き認定レポートを生成 AWS IoT Greengrass V1 しなくなりました。Device Qualification Program を通じて AWS Partner Device Catalog にリスト AWS IoT Greengrass V1 する新しいデバイスを認定することはできなくなりました。 AWSGreengrass V1 デバイスを認定することはできませんが、引き続き IDT for AWS IoT Greengrass V1 を使用して Greengrass V1 デバイスをテストできます。IDT for AWS IoT Greengrass V2 を使用して、Greengrass デバイスを認定し、AWS Partner Device Catalog に含めることをお勧めします。詳細については、「の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1」を参照してください。

メンテナンススケジュールの終了

次の表に、 AWS IoT Greengrass Core v1.x アーティファクトと機能のメンテナンス終了日を示します。メンテナンスのスケジュールやポリシーについてご質問がある場合は、AWS サポートにお問い合わせください。

アーティファクトまたは機能	メンテナンス終了日
Greengrass APT リポジトリのインストール	2022 年 2 月 11 日
ML イメージ分類コネクタ	2022 年 3 月 31 日
ML オブジェクト検出コネクタ	2022 年 3 月 31 日
ML フィードバックコネクタ	2022 年 3 月 31 日

アーティファクトまたは機能	メンテナンス終了日
AWS IoT Analytics コネクタ	2022 年 3 月 31 日
Twilio 通知コネクタ	2022 年 3 月 31 日
Splunk 統合コネクタ	2022 年 3 月 31 日
シリアルストリーミングコネクタ	2022 年 3 月 31 日
ServiceNow MetricBase 統合コネクタ	2022 年 3 月 31 日
Raspberry Pi GPIO コネクタ	2022 年 3 月 31 日
AWS IoT Greengrass Core ソフトウェア v1.10.x	2022 年 3 月 31 日
AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージ	2022 年 6 月 30 日
AWS IoT Greengrass Core ソフトウェア v1.11.x	2023 年 6 月 30 日
AWS IoT Greengrass Core ソフトウェア v1.11.x スナップ	2023 年 12 月 31 日

AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージのメンテナンスの終了

2022 年 6 月 30 日に、Amazon Elastic Container Registry (Amazon ECR) および Docker Hub に公開された AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージのメンテナンス AWS IoT Greengrass が終了しました。これらの Docker イメージは、メンテナンス終了から 1 年後の 2023 年 6 月 30 日まで、Amazon ECR および Docker Hub から引き続きダウンロードすることができます。ただし、2022 年 6 月 30 日にメンテナンスが終了した後、 AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージはセキュリティパッチやバグ修正を受け取らなくなりました。これらの Docker イメージに依存する本稼働ワークロードを実行する場合は、 AWS IoT Greengrass が提供する Dockerfiles を使用して独自の Docker イメージを構築することをお勧めします。詳細については、「AWS IoT Greengrass Docker ソフトウェア」を参照してください。

AWS IoT Greengrass Core ソフトウェア v1.x APT リポジトリのメンテナンスの終了

2022 年 2 月 11 日、 は APT リポジトリから AWS IoT Greengrass Core ソフトウェア v1.x をインストールするオプションのメンテナンス AWS IoT Greengrass を終了しました。この日付に APT リポジトリが削除されたため、APT リポジトリを使用して AWS IoT Greengrass Core ソフトウェアを更新したり、新しいデバイスに AWS IoT Greengrass Core ソフトウェアをインストールしたりできなくなります。 AWS IoT Greengrass リポジトリを追加したデバイスでは、ソースリストからリポジトリを削除する必要があります。tar ファイルを使用して AWS IoT Greengrass Core ソフトウェア v1.x を更新することをお勧めします。

AWS IoT Greengrass Core ソフトウェア v1.11.x スナップのメンテナンスの終了

2023 年 12 月 31 日、 AWS IoT Greengrass は、 snapcraft.io公開されている AWS IoT Greengrass コアソフトウェアバージョン 1.11.x スナップのメンテナンスを終了します。現在スナップを実行しているデバイスは、追って通知があるまで引き続き動作します。ただし、メンテナンス終了後、AWS IoT Greengrass コアスナップはセキュリティパッチやバグ修正を受信しなくなります。

の開始方法 AWS IoT Greengrass

この入門チュートリアルには、 AWS IoT Greengrass 基本を示し、使用を開始するのに役立つように設計されたいくつかのモジュールが含まれています AWS IoT Greengrass。このチュートリアルでは、次のような基本的な概念について説明します。

- AWS IoT Greengrass コアとグループの設定。
- エッジで AWS Lambda 関数を実行するためのデプロイプロセス。
- クライアント AWS IoT デバイスと呼ばれるデバイスを AWS IoT Greengrass コアに接続します。
- ローカル Lambda 関数、クライアントデバイス、 間の MQTT 通信を許可するサブスクリプションの作成 AWS IoT。

の使用を開始する方法を選択する AWS IoT Greengrass

このチュートリアルを使用してコアデバイスをセットアップする方法を選択できます。

- コアデバイスで Greengrass デバイスのセットアップを実行すると、 AWS IoT Greengrass 依存関係のインストールから Hello World Lambda 関数のテストまでを数分で実行できます。このスクリプトは、モジュール 1 から モジュール 3-1 のステップを再現します。

  - または -

- モジュール 1 からモジュール 3-1 のステップを、順を追って、Greengrass の要件とプロセスをより詳しく調べます。以下のステップでは、コアデバイスをセットアップします。Hello World Lambda 関数を含む Greengrass グループを作成して設定し、Greengrass グループをデプロイします。通常、この処理には 1 ~ 2 時間かかります。

クイックスタート

Greengrass デバイスのセットアップは、コアデバイスと Greengrass リソースを設定します。スクリプト:

- AWS IoT Greengrass 依存関係をインストールします。
- ルート CA 証明書とコアデバイスの証明書とキーをダウンロードします。
- デバイスに AWS IoT Greengrass Core ソフトウェアをダウンロード、インストール、設定します。
- コアデバイスで Greengrass デーモンプロセスを開始します。
- 必要に応じて、Greengrass サービスロールを作成または更新します。
- Greengrass グループと Greengrass コアを作成します。
- (オプション) Hello World Lambda 関数、サブスクリプション、およびローカルのログ記録設定を作成します。
- （オプション）Greengrass グループをデプロイします。

モジュール 1 および 2

モジュール 1 と モジュール 2 では、環境の設定方法について説明します。（または、Greengrass デバイスのセットアップを使用してこれらのモジュールを実行します）。

- コアデバイスを Greengrass 用に設定します。
- 依存関係チェッカーを実行します。
- Greengrass グループと Greengrass コアを作成します。
- tar.gz ファイルから最新の AWS IoT Greengrass Core ソフトウェアをダウンロードしてインストールします。

- コア上で Greengrass デーモンプロセスを開始します。

> **Note**
>
> AWS IoT Greengrass には、サポートされている Debian プラットフォームへのインストールなど、 AWS IoT Greengrass Core ソフトウェアaptをインストールするためのその他のオプションも用意されています。詳細については、「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

モジュール 3-1 および 3-2

モジュール 3-1 と モジュール 3-2 では、ローカルの Lambda 関数の使用方法について説明します。（または、Greengrass デバイスのセットアップ を使用してモジュール 3-1 を実行します）。

- で Hello World Lambda 関数を作成します AWS Lambda。
- Greengrass グループに Lambda 関数を追加します。
- Lambda 関数と 間の MQTT 通信を許可するサブスクリプションを作成します AWS IoT。
- Greengrass システムコンポーネントと Lambda 関数のローカルのログを設定します。
- Lambda 関数とサブスクリプションを含む Greengrass グループをデプロイします。
- ローカル Lambda 関数から にメッセージを送信します AWS IoT。
- からローカル Lambda 関数を呼び出します AWS IoT。
- オンデマンドおよび長期間有効な関数をテストします。

モジュール 4 およびモジュール 5

モジュール 4 は、クライアントデバイスがコアに接続し、相互に通信する方法を示しています。

モジュール 5 は、クライアントデバイスがシャドウを使用して状態を制御する方法を示しています。

- AWS IoT デバイス (コマンドラインターミナルで表されます) を登録してプロビジョニングします。
- AWS IoT Device SDK for Python をインストールします。これは、Greengrass のコアを検出するためにクライアントデバイスによって使用されます。
- クライアントデバイスを Greengrass グループに追加します。
- MQTT 通信を許可するサブスクリプションを作成します。
- クライアントデバイスを含む Greengrass グループをデプロイします。

- デバイス間通信をテストします。
- シャドウ状態の更新をテストします。

モジュール 6

モジュール 6 では、Lambda 関数が AWS クラウドにアクセスする方法を説明します。

- Amazon DynamoDB リソースへのアクセスを許可する Greengrass グループロールを作成します。
- Greengrass グループに Lambda 関数を追加します。この関数は AWS SDK for Python を使用して DynamoDB とやり取りします。
- MQTT 通信を許可するサブスクリプションを作成します。
- DynamoDB とのやりとりをテストします。

モジュール 7

モジュール 7 では、Greengrass Core 用にシミュレートされたハードウェアセキュリティモジュール (HSM) を設定する方法を示します。

> **Important**
>
> この高度なモジュールは、実験と初期テストのためにのみ提供されています。どのような種類の本番稼働用でもありません。

- ソフトウェアベースの HSM とプライベートキーをインストールして設定します。
- ハードウェアセキュリティを使用するように Greengrass コアを設定します。
- ハードウェアセキュリティ設定をテストします。

要件

このチュートリアルを完了するには、以下が必要です。

- Mac、Windows PC、または UNIX 互換システム。
- AWS アカウント。アカウントをお持ちでない場合は、「the section called “を作成する AWS アカウント”」を参照してください。
- がサポートする AWS リージョンの使用 AWS IoT Greengrass。でサポートされているリージョンのリストについては AWS IoT Greengrass、のAWS 「エンドポイントとクォータ」を参照してくださいAWS 全般のリファレンス。

> **Note**
>
> を書き留め AWS リージョン 、このチュートリアル全体で一貫して使用してください。チュートリアル AWS リージョン 中に を切り替えると、ステップの完了に問題がある可能性があります。

- Raspberry Pi 4 モデル B または Raspberry Pi 3 モデル B/B+ と 8 GB microSD カード、または Amazon EC2 インスタンス。 AWS IoT Greengrass は物理的なハードウェアで使用するのが最適であるため、Raspberry Pi を使用することをお勧めします。

> **Note**
>
> Raspberry Pi のモデルを取得するには、次のコマンドを実行します。
>
> ```
> cat /proc/cpuinfo
> ```
>
> リストの下部にある `Revision` 属性の値をメモし、「Which Pi have I got?」の表を参照してください。例えば、`Revision` の値が `a02082` である場合、Pi は「3 Model B」であることが表でわかります。
> Raspberry Pi のアーキテクチャを確認するには、次のコマンドを実行します。
>
> ```
> uname -m
> ```
>
> このチュートリアルでは、結果は `armv71` 以上である必要があります。

- Python の基本的な知識

このチュートリアルは Raspberry Pi AWS IoT Greengrass で実行することを目的としていますが、は他のプラットフォーム AWS IoT Greengrass もサポートしています。詳細については、「the section called “サポートされているプラットフォームと要件”」を参照してください。

を作成する AWS アカウント

がない場合は AWS アカウント、以下の手順に従って を作成してアクティブ化します AWS アカウント。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 のセキュリティを確保し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン 「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

> **Important**
>
> このチュートリアルでは、IAM ユーザーアカウントに管理者アクセス権があることを前提としています。

クイックスタート: Greengrass デバイスのセットアップ

Greengrass デバイスのセットアップは、 の使用を開始できるように、コアデバイスを数分でセットアップするスクリプトです AWS IoT Greengrass。このスクリプトは、次の場合に使用します。

1. デバイスを設定し、 AWS IoT Greengrass Core ソフトウェアをインストールします。
2. クラウドベースのリソースを設定します。
3. 必要に応じて、 AWS IoT Greengrass コア AWS IoT から に MQTT メッセージを送信する Hello World Lambda 関数を使用して Greengrass グループをデプロイします。このセットアップでは、次の図に示す Greengrass 環境を設定します。

要件

Greengrass デバイスのセットアップには、次の要件があります。

- コアデバイスは、サポートされているプラットフォームを使用する必要があります。デバイスには、apt、yum、または opkg の適切なパッケージマネージャーがインストールされている必要があります。

- スクリプトを実行する Linux ユーザーは、sudo として実行するアクセス許可を持っている必要があります。

- AWS アカウント 認証情報を指定する必要があります。詳細については、「the section called "AWS アカウント 認証情報の提供"」を参照してください。

> **Note**
>
> Greengrass デバイスのセットアップでは、最新バージョンの AWS IoT Greengrass Core ソフトウェアがデバイスにインストールされます。 AWS IoT Greengrass Core ソフトウェアをインストールすることで、Greengrass Core ソフトウェアライセンス契約に同意したものとみなされます。

Greengrass デバイスのセットアップを実行する

Greengrass デバイスのセットアップは、ほんの数ステップで実行できます。 AWS アカウント 認証情報を指定すると、スクリプトは Greengrass コアデバイスをプロビジョニングし、Greengrass グループを数分でデプロイします。ターゲットデバイスのターミナルウィンドウで次のコマンドを実行します。

> **Note**
>
> 次のステップでは、スクリプトを対話モードで実行する方法を示します。対話モードでは、各入力値を入力または受け入れるように求められます。スクリプトをサイレントで実行する方法については、「the section called "Greengrass デバイスのセットアップをサイレントモードで実行する"」を参照してください。

1. 認証情報を入力しないでください。この手順では、一時的なセキュリティ認証情報を環境変数として提供することを前提としています。

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

> **Note**
>
> Greengrass デバイスセットアップを Raspbian または OpenWrt プラットフォームで実行している場合は、これらのコマンドのコピーを作成してください。デバイスを再起動した後、それらを再度指定する必要があります。

2. スクリプトをダウンロードして起動します。`wget` または `curl` を使用してスクリプトをダウンロードできます。

   `wget`:

```
wget -q -O ./gg-device-setup-latest.sh https://d1onfpft10uf5o.cloudfront.net/
greengrass-device-setup/downloads/gg-device-setup-latest.sh && chmod +x ./
gg-device-setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-
greengrass-interactive
```

   `curl`:

```
curl https://d1onfpft10uf5o.cloudfront.net/greengrass-device-setup/downloads/
gg-device-setup-latest.sh > gg-device-setup-latest.sh && chmod +x ./gg-device-
setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-greengrass-
interactive
```

3. 入力値を求めるコマンドプロンプトに進みます。[Enter] キーを押してデフォルト値を使用するか、カスタム値を入力して [Enter] キーを押します。

   スクリプトは、次のようなステータスメッセージを端末に書き込みます。

```
############### Greengrass Device Setup v1.0.0 ###############
[GreengrassDeviceSetup] The Greengrass Device Setup bootstrap log is available at: /tmp/greengrass-device-setup-bootstrap-1575933831.log
[GreengrassDeviceSetup] Using package management tool: yum...
[GreengrassDeviceSetup] Using runtime: python3.7...
[GreengrassDeviceSetup] Installing a dedicated pip for Greengrass Device Setup...
[GreengrassDeviceSetup] Validating and installing required dependencies...
[GreengrassDeviceSetup] The Greengrass Device Setup configuration is complete. Starting the Greengrass environment setup...
[GreengrassDeviceSetup] Forwarding command-line parameters: bootstrap-greengrass-interactive

[GreengrassDeviceSetup] Validating the device environment...
[GreengrassDeviceSetup] Validation of the device environment is complete.

[GreengrassDeviceSetup] Running the Greengrass environment setup...
[GreengrassDeviceSetup] The Greengrass environment setup is complete.

[GreengrassDeviceSetup] Configuring cloud-based Greengrass group management...
[GreengrassDeviceSetup] The Greengrass group configuration is complete.

[GreengrassDeviceSetup] Preparing the Greengrass core software...
[GreengrassDeviceSetup] The Greengrass core software is running.

[GreengrassDeviceSetup] Configuring the group deployment...
[GreengrassDeviceSetup] The group deployment is complete.
```

4. コアデバイスで Raspbian または OpenWrt が実行されている場合は、プロンプトが表示されたらデバイスを再起動し、認証情報を入力して、スクリプトを再起動します。

   a. デバイスを再起動するように求められたら、次のいずれかのコマンドを実行します。

      Raspbian プラットフォームの場合:

      ```
      sudo reboot
      ```

      OpenWrt プラットフォームの場合:

      ```
      reboot
      ```

   b. デバイスの起動後、ターミナルを開き、環境変数として認証情報を入力します。

      ```
      export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
      export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
      export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
      ```

   c. スクリプトを再起動します。

      ```
      sudo -E ./gg-device-setup-latest.sh bootstrap-greengrass-interactive
      ```

   d. 前回のセッションからの入力値を使用するか、新規インストールを開始するかのプロンプトが表示されたら、yes と入力して入力値を再利用します。

> **Note**
>
> 再起動が必要なプラットフォームでは、認証情報を除く、前のセッションからの入力値が一時的に `GreengrassDeviceSetup.config.info` ファイルに保存されます。

セットアップが完了すると、ターミナルには、次のような成功ステータスメッセージが表示されます。

```
================================================================================

Your device is running the Greengrass core software.
Your Greengrass group and Hello World Lambda function were deployed to the core device.


Setup information:

Device info: Linux-4.14.152-127.182.amzn2.x86_64-x86_64-with-glibc2.2.5
Greengrass core software location: /
Installed Greengrass core software version: 1.10.0
Greengrass core: arn:aws:iot:us-west-2:012345678910:thing/GreengrassDeviceSetup_Core_d46a0ea4-18ae-4376-8f44-4a504cdea608
Greengrass core IoT certificate: arn:aws:iot:us-west-2:012345678910:cert/23fbf0f4b6a5ea369f2b97f1a1b558180a240faa8e059ce19dc58f4a4c0d3b77
Greengrass core IoT certificate location: /greengrass/certs/23fbf0f4b6.cert.pem
Greengrass core IoT key location: /greengrass/certs/23fbf0f4b6.private.key
Deployed Greengrass group name: GreengrassDeviceSetup_Group_ee70f777-9af0-43b6-8612-a18b418e8b4a
Deployed Greengrass group ID: 6f5c8410-f3a6-43a2-acf3-33158e10fb8e
Deployed Greengrass group version: arn:aws:greengrass:us-west-2:012345678910:/greengrass/groups/6f5c8410-f3a6-43a2-acf3-33158e10fb8e/versi
Greengrass service role: arn:aws:iam::012345678910:role/GreengrassServiceRole_mui1v
GreengrassDeviceSetup log location: GreengrassDeviceSetup-20191209-232356.log
Deployed hello-world Lambda function: arn:aws:lambda:us-west-2:012345678910:function:Greengrass_HelloWorld_uNTf2:1
Hello-world subscriber topic: hello/world

You can now use the AWS IoT Console to subscribe
to the 'hello/world' topic to receive messages published from your
Greengrass core.

================================================================================
```

5. 指定した入力値を使用してスクリプトが設定する新しい Greengrass グループを確認します。

   a. コンピュータAWS Management Consoleで にサインインし、 AWS IoT コンソールを開きます。

   > **Note**
   >
   > コンソールで AWS リージョン 選択した が、Greengrass 環境の設定に使用したものと同じであることを確認します。デフォルトでは、[Region] (リージョン) は [US West (Oregon)] (米国西部 (オレゴン)) です。

   b. ナビゲーションペインで、[Greengrass devices] (Greengrass デバイス) を展開し、[Groups (V1)] (グループ〔V1〕) を選択して、新しく作成したグループを検索します。

6. Hello World Lambda 関数を含めた場合、Greengrass デバイスセットアップは、Greengrass グループをコアデバイスにデプロイします。Lambda 関数をテストしたり、グループから Lambda 関数を削除したりする方法については、開始方法チュートリアルのモジュール 3-1 の「the section called “Lambda 関数がコアデバイスで実行されていることを確認する”」に進みます。

> **Note**
>
> コンソールで AWS リージョン 選択した が、Greengrass 環境の設定に使用したものと同じであることを確認します。デフォルトでは、[Region] (リージョン) は [US West (Oregon)] (米国西部 (オレゴン)) です。

Hello World Lambda 関数を含めなかった場合は、独自の Lambda 関数を作成するか、Greengrass の他の機能を試すことができます。例えば、Docker アプリケーションデプロイ コネクタをグループに追加し、それを使用して Docker コンテナをコアデバイスにデプロイできます。

問題のトラブルシューティング

以下の情報を使用して、 AWS IoT Greengrass デバイスのセットアップに関する問題のトラブルシューティングを行うことができます。

Error: Python (python3.7) not found。Attempting to install it..。

解決策: Amazon EC2 インスタンスを使用すると、このエラーが表示されることがあります。このエラーは、Python が `/usr/bin/python3.7` フォルダにインストールされていない場合に発生します。このエラーを解決するには、Python をインストールした後で正しいディレクトリに移動します。

```
sudo ln -s /usr/local/bin/python3.7 /usr/bin/python3.7
```

その他のトラブルシューティング

AWS IoT Greengrass デバイスのセットアップに関するその他の問題をトラブルシューティングするには、ログファイルでデバッグ情報を探します。

- Greengrass デバイスのセットアップ設定に関する問題については、`/tmp/greengrass-device-setup-bootstrap-epoch-timestamp.log` ファイルを確認してください。
- Greengrass グループまたはコア環境の設定に問題がある場合は、指定したディレクトリと同じディレクトリにある、`GreengrassDeviceSetup-date-time.log` または指定した場所にある `gg-device-setup-latest.sh` ファイルを確認してください。

トラブルシューティングの詳細については、「」を参照トラブルシューティングするか、 AWS IoT GreengrassAWS re:Post の タグを確認してください。

Greengrass デバイスセットアップ設定オプション

AWS リソースにアクセスし、Greengrass 環境をセットアップするように Greengrass デバイスのセットアップを設定します。

AWS アカウント 認証情報の提供

Greengrass デバイスのセットアップでは、 AWS アカウント 認証情報を使用して AWS リソースにアクセスします。IAM ユーザーの長期認証情報、または IAM ロールの一時的なセキュリティ認証情報をサポートします。

最初に、認証情報を取得します。

- 長期認証情報を使用するには、IAM ユーザーのアクセスキー ID とシークレットアクセスキーを指定します。長期認証情報のアクセスキーの作成については、「IAM ユーザーガイド」の「IAM ユーザーのアクセスキーの管理」を参照してください。

- 一時的なセキュリティ認証情報を使用するには (推奨)、引き受けた IAM ロールのアクセスキー ID、シークレットアクセスキー、およびセッショントークンを指定します。コマンドから一時的なセキュリティ認証情報を AWS STS `assume-role`抽出する方法については、IAM ユーザーガイドの「 での一時的なセキュリティ認証情報 AWS CLIの使用」を参照してください。

> **Note**
>
> このチュートリアルでは、IAM ユーザーまたは IAM ロールに管理者のアクセス権があることを前提としています。

次に、次の 2 つのうちいずれかの方法で Greengrass デバイスのセットアップに認証情報を入力します。

- 環境変数として。the section called “Greengrass デバイスのセットアップを実行する” のステップ 1 に示すようにスクリプトを開始する前に、`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、および `AWS_SESSION_TOKEN`（必要な場合）環境変数を設定します。

- 入力値として。スクリプトを起動した後、アクセスキー ID、シークレットアクセスキー、およびセッショントークン (必要な場合) の値をターミナルに直接入力します。

Greengrass デバイスのセットアップでは、認証情報が保存または保存されません。

入力値の指定

対話モードでは、入力値を求めるプロンプトが表示されます。[Enter] キーを押してデフォルト値を使用するか、カスタム値を入力して [Enter] キーを押します。サイレントモードでは、スクリプトの起動後に入力値を指定します。

入力値

AWS アクセスキー ID

長期的または一時的なセキュリティ認証情報のアクセスキー ID。環境変数として認証情報を指定しない場合のみ、このオプションを入力値として指定してください。詳細については、「the section called “AWS アカウント 認証情報の提供”」を参照してください。

サイレントモードのオプション名: `--aws-access-key-id`

AWS シークレットアクセスキー

長期的または一時的なセキュリティ認証情報からのシークレットアクセスキー。環境変数として認証情報を指定しない場合のみ、このオプションを入力値として指定してください。詳細については、「the section called “AWS アカウント 認証情報の提供”」を参照してください。

サイレントモードのオプション名: `--aws-secret-access-key`

AWS セッショントークン

一時的なセキュリティ認証情報からのセッショントークン。環境変数として認証情報を指定しない場合のみ、このオプションを入力値として指定してください。詳細については、「the section called “AWS アカウント 認証情報の提供”」を参照してください。

サイレントモードのオプション名: `--aws-session-token`

AWS リージョン

Greengrass グループを作成する AWS リージョン 。サポートされている AWS リージョンのリストについては、 のAWS IoT Greengrass「」を参照してくださいAmazon Web Services 全般のリファレンス。

デフォルト値: `us-west-2`

サイレントモードのオプション名: `--region`

グループ名

Greengrass グループの名前。

デフォルト値: `GreengrassDeviceSetup_Group_`*`guid`*

サイレントモードのオプション名: `--group-name`

コア名

Greengrass Core の名前。コアは、 AWS IoT Greengrass Core ソフトウェアを実行する AWS IoT デバイス (モノ) です。コアが AWS IoT レジストリと Greengrass グループに追加されます。名前を指定する場合は、 AWS アカウント および で一意である必要があります AWS リージョン。

デフォルト値: `GreengrassDeviceSetup_Core_`*`guid`*

サイレントモードのオプション名: `--core-name`

AWS IoT Greengrass コアソフトウェアのインストールパス

AWS IoT Greengrass Core ソフトウェアをインストールするデバイスファイルシステム内の場所。

デフォルト値: `/`

サイレントモードのオプション名: `--ggc-root-path`

Hello World Lambda 関数

Greengrass グループに Hello World Lambda 関数を含めるかどうかを示します。この関数は、5 秒ごとに MQTT メッセージを `hello/world` トピックに発行します。

このスクリプトは、このユーザー定義の Lambda 関数を で作成して公開 AWS Lambda し、Greengrass グループに追加します。また、このスクリプトは、関数が MQTT メッセージを AWS IoTに送信できるようにするサブスクリプションをグループ内に作成します。

> **Note**
>
> これは Python 3.7 Lambda 関数です。Python 3.7 がデバイスにインストールされておらず、スクリプトがそれをインストールできない場合、スクリプトはターミナルにエラーメッセージを出力します。Lambda 関数をグループに含めるには、Python 3.7 を手動でインストールし、スクリプトを再起動する必要があります。Lambda 関数を使用せずに Greengrass グループを作成するには、スクリプトを再起動し、関数を含めるように求められたら no を入力します。

デフォルト値: no

サイレントモードのオプション名: `--hello-world-lambda` - このオプションは値を取りません。関数を作成する場合は、コマンドに含めます。

デプロイのタイムアウト

Greengrass デバイスのセットアップが Greengrass グループのデプロイの状態のチェックを停止するまでの秒数。これは、グループに Hello World Lambda 関数が含まれている場合にのみ使用されます。それ以外の場合、グループはデプロイされません。

デプロイ時間は、ネットワークの速度によって異なります。ネットワーク速度が遅い場合は、この値を大きくすることができます。

デフォルト値: `180`

サイレントモードのオプション名: `--deployment-timeout`

ログパス

Greengrass グループおよびコアセットアップオペレーションに関する情報を含むログファイルの場所。このログを使用して、Greengrass グループとコアセットアップに関するデプロイやその他の問題のトラブルシューティングを行います。

デフォルト値: `./`

サイレントモードのオプション名: `--log-path`

詳細レベル

スクリプトの実行中にターミナルに詳細なログ情報を出力するかどうかを示します。この情報を使用して、デバイスのセットアップのトラブルシューティングを行うことができます。

デフォルト値: no

サイレントモードのオプション名: `--verbose` - このオプションは値を取りません。詳細なログ情報を出力する場合は、コマンドに含めます。

Greengrass デバイスのセットアップをサイレントモードで実行する

Greengrass デバイスのセットアップをサイレントモードで実行して、スクリプトが値の入力を要求しないようにできます。サイレントモードで実行するには、スクリプトの起動後に `bootstrap-greengrass` モードと入力値 を指定します。デフォルト値を使用する場合は、入力値を省略できます。

この手順は、スクリプトの起動前に環境変数として AWS アカウント 認証情報を指定するか、スクリプトの起動後に入力値として認証情報を指定するかによって異なります。

環境変数としての認証情報の提供

1. 環境変数として認証情報を指定します。次の例では、セッショントークンを含む一時的な認証情報をエクスポートします。

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

> **Note**
>
> Greengrass デバイスセットアップを Raspbian または OpenWrt プラットフォームで実行している場合は、これらのコマンドのコピーを作成してください。デバイスを再起動した後、それらを再度指定する必要があります。

2. スクリプトをダウンロードして起動します。必要に応じて入力値を指定します。例:

   - すべてのデフォルト値を使用するには:

```
wget -q -O ./gg-device-setup-latest.sh https://d1onfpft10uf5o.cloudfront.net/
greengrass-device-setup/downloads/gg-device-setup-latest.sh && chmod +x ./
gg-device-setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-
greengrass
```

   - カスタム値を指定するには:

```
wget -q -O ./gg-device-setup-latest.sh https://d1onfpft10uf5o.cloudfront.net/
greengrass-device-setup/downloads/gg-device-setup-latest.sh && chmod +x ./
gg-device-setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-
greengrass
--region us-east-1
--group-name Custom_Group_Name
--core-name Custom_Core_Name
--ggc-root-path /custom/ggc/root/path
--deployment-timeout 300
--log-path /customized/log/path
--hello-world-lambda
--verbose
```

> **Note**
>
> `curl` を使用してスクリプトをダウンロードするには、コマンドで `wget -q -O` を `curl` に置き換えます。

3. コアデバイスで Raspbian または OpenWrt が実行されている場合は、プロンプトが表示されたらデバイスを再起動し、認証情報を入力して、スクリプトを再起動します。

   a. デバイスを再起動するように求められたら、次のいずれかのコマンドを実行します。

      Raspbian プラットフォームの場合:

```
sudo reboot
```

      OpenWrt プラットフォームの場合:

```
reboot
```

b. デバイスの起動後、ターミナルを開き、環境変数として認証情報を入力します。

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

c. スクリプトを再起動します。

```
sudo -E ./gg-device-setup-latest.sh bootstrap-greengrass
```

d. 前回のセッションからの入力値を使用するか、新規インストールを開始するかのプロンプトが表示されたら、yes と入力して入力値を再利用します。

> **Note**
>
> 再起動が必要なプラットフォームでは、認証情報を除く、前のセッションからの入力値が一時的に GreengrassDeviceSetup.config.info ファイルに保存されます。

セットアップが完了すると、ターミナルには、次のような成功ステータスメッセージが表示されます。

```
================================================================================

Your device is running the Greengrass core software.
Your Greengrass group and Hello World Lambda function were deployed to the core device.


Setup information:

Device info: Linux-4.14.152-127.182.amzn2.x86_64-x86_64-with-glibc2.2.5
Greengrass core software location: /
Installed Greengrass core software version: 1.10.0
Greengrass core: arn:aws:iot:us-west-2:012345678910:thing/GreengrassDeviceSetup_Core_d46a0ea4-18ae-4376-8f44-4a504cdea608
Greengrass core IoT certificate: arn:aws:iot:us-west-2:012345678910:cert/23fbf0f4b6a5ea369f2b97f1a1b558180a240faa8e059ce19dc58f4a4c0d3b77
Greengrass core IoT certificate location: /greengrass/certs/23fbf0f4b6.cert.pem
Greengrass core IoT key location: /greengrass/certs/23fbf0f4b6.private.key
Deployed Greengrass group name: GreengrassDeviceSetup_Group_ee70f777-9af0-43b6-8612-a18b418e8b4a
Deployed Greengrass group ID: 6f5c8410-f3a6-43a2-acf3-33158e10fb8e
Deployed Greengrass group version: arn:aws:greengrass:us-west-2:012345678910:/greengrass/groups/6f5c8410-f3a6-43a2-acf3-33158e10fb8e/versi
Greengrass service role: arn:aws:iam::012345678910:role/GreengrassServiceRole_mui1v
GreengrassDeviceSetup log location: GreengrassDeviceSetup-20191209-232356.log
Deployed hello-world Lambda function: arn:aws:lambda:us-west-2:012345678910:function:Greengrass_HelloWorld_uNTf2:1
Hello-world subscriber topic: hello/world

You can now use the AWS IoT Console to subscribe
to the 'hello/world' topic to receive messages published from your
Greengrass core.

================================================================================
```

4. Hello World Lambda 関数を含めた場合、Greengrass デバイスセットアップは、Greengrass グループをコアデバイスにデプロイします。Lambda 関数をテストしたり、グループから Lambda 関数を削除したりする方法については、開始方法チュートリアルのモジュール 3-1 の「the section called “Lambda 関数がコアデバイスで実行されていることを確認する”」に進みます。

> **Note**
>
> コンソールで AWS リージョン 選択した が、Greengrass 環境の設定に使用したものと同じであることを確認します。デフォルトでは、[Region] (リージョン) は [US West (Oregon)] (米国西部 (オレゴン)) です。

Hello World Lambda 関数を含めなかった場合は、独自の Lambda 関数を作成するか、Greengrass の他の機能を試すことができます。例えば、Docker アプリケーションデプロイ コネクタをグループに追加し、それを使用して Docker コンテナをコアデバイスにデプロイできます。

入力値としての認証情報の提供

1. スクリプトをダウンロードして起動します。認証情報を指定し、必要に応じてその他の入力値を指定します。次の例は、セッショントークンを含む一時的な認証情報を提供する方法を示しています。

   - すべてのデフォルト値を使用するには:

```
wget -q -O ./gg-device-setup-latest.sh https://d1onfpft10uf5o.cloudfront.net/
greengrass-device-setup/downloads/gg-device-setup-latest.sh && chmod +x ./
gg-device-setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-
greengrass
--aws-access-key-id AKIAIOSFODNN7EXAMPLE
--aws-secret-access-key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
--aws-session-token AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

   - カスタム値を指定するには:

```
wget -q -O ./gg-device-setup-latest.sh https://d1onfpft10uf5o.cloudfront.net/
greengrass-device-setup/downloads/gg-device-setup-latest.sh && chmod +x ./
```

```
gg-device-setup-latest.sh && sudo -E ./gg-device-setup-latest.sh bootstrap-
greengrass
--aws-access-key-id AKIAIOSFODNN7EXAMPLE
--aws-secret-access-key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
--aws-session-token AQoDYXdzEJr1K...o5OytwEXAMPLE=
--region us-east-1
--group-name Custom_Group_Name
--core-name Custom_Core_Name
--ggc-root-path /custom/ggc/root/path
--deployment-timeout 300
--log-path /customized/log/path
--hello-world-lambda
--verbose
```

> **Note**
>
> Greengrass デバイスセットアップを Raspbian または OpenWrt プラットフォームで実行している場合は、これらの認証情報のコピーを作成してください。デバイスを再起動した後、それらを再度指定する必要があります。
> `curl` を使用してスクリプトをダウンロードするには、コマンドで `wget -q -O` を `curl` に置き換えます。

2. コアデバイスで Raspbian または OpenWrt が実行されている場合は、プロンプトが表示されたらデバイスを再起動し、認証情報を入力して、スクリプトを再起動します。

   a. デバイスを再起動するように求められたら、次のいずれかのコマンドを実行します。

      Raspbian プラットフォームの場合:

      ```
      sudo reboot
      ```

      OpenWrt プラットフォームの場合:

      ```
      reboot
      ```

   b. スクリプトを再起動します。コマンドには認証情報を含める必要がありますが、他の入力値を含めることはできません。例:

```
sudo -E ./gg-device-setup-latest.sh bootstrap-greengrass
--aws-access-key-id AKIAIOSFODNN7EXAMPLE
--aws-secret-access-key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
--aws-session-token AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

c. 前回のセッションからの入力値を使用するか、新規インストールを開始するかのプロンプトが表示されたら、yes と入力して入力値を再利用します。

> **Note**
>
> 再起動が必要なプラットフォームでは、認証情報を除く、前のセッションからの入力値が一時的に GreengrassDeviceSetup.config.info ファイルに保存されます。

セットアップが完了すると、ターミナルには、次のような成功ステータスメッセージが表示されます。

```
==============================================================================

Your device is running the Greengrass core software.
Your Greengrass group and Hello World Lambda function were deployed to the core device.


Setup information:

Device info: Linux-4.14.152-127.182.amzn2.x86_64-x86_64-with-glibc2.2.5
Greengrass core software location: /
Installed Greengrass core software version: 1.10.0
Greengrass core: arn:aws:iot:us-west-2:012345678910:thing/GreengrassDeviceSetup_Core_d46a0ea4-18ae-4376-8f44-4a504cdea608
Greengrass core IoT certificate: arn:aws:iot:us-west-2:012345678910:cert/23fbf0f4b6a5ea369f2b97f1a1b558180a240faa8e059ce19dc58f4a4c0d3b77
Greengrass core IoT certificate location: /greengrass/certs/23fbf0f4b6.cert.pem
Greengrass core IoT key location: /greengrass/certs/23fbf0f4b6.private.key
Deployed Greengrass group name: GreengrassDeviceSetup_Group_ee70f777-9af0-43b6-8612-a18b418e8b4a
Deployed Greengrass group ID: 6f5c8410-f3a6-43a2-acf3-33158e10fb8e
Deployed Greengrass group version: arn:aws:greengrass:us-west-2:012345678910:/greengrass/groups/6f5c8410-f3a6-43a2-acf3-33158e10fb8e/versi
Greengrass service role: arn:aws:iam::012345678910:role/GreengrassServiceRole_mui1v
GreengrassDeviceSetup log location: GreengrassDeviceSetup-20191209-232356.log
Deployed hello-world Lambda function: arn:aws:lambda:us-west-2:012345678910:function:Greengrass_HelloWorld_uNTf2:1
Hello-world subscriber topic: hello/world

You can now use the AWS IoT Console to subscribe
to the 'hello/world' topic to receive messages published from your
Greengrass core.

==============================================================================
```

3. Hello World Lambda 関数を含めた場合、Greengrass デバイスセットアップは、Greengrass グループをコアデバイスにデプロイします。Lambda 関数をテストしたり、グループから Lambda 関数を削除したりする方法については、開始方法チュートリアルのモジュール 3-1 の「the section called “Lambda 関数がコアデバイスで実行されていることを確認する”」に進みます。

> **Note**
>
> コンソールで AWS リージョン 選択した が、Greengrass 環境の設定に使用したものと同じであることを確認します。デフォルトでは、[Region] (リージョン) は [US West (Oregon)] (米国西部 (オレゴン)) です。

Hello World Lambda 関数を含めなかった場合は、独自の Lambda 関数を作成するか、Greengrass の他の機能を試すことができます。例えば、Docker アプリケーションデプロイ コネクタをグループに追加し、それを使用して Docker コンテナをコアデバイスにデプロイできます。

モジュール 1: Greengrass の環境設定

このモジュールでは、out-of-the-box Raspberry Pi、Amazon EC2 インスタンス、またはその他のデバイスを AWS IoT Greengrass コアデバイス AWS IoT Greengrass として で使用する準備を整える方法について説明します。

> **Tip**
>
> または、コアデバイスをセットアップするスクリプトを使用する場合は、「the section called “クイックスタート: Greengrass デバイスのセットアップ”」を参照してください。

このモジュールの所要時間は 30 分未満です。

開始する前に、このチュートリアルの要件をお読みください。次に、次のいずれかのトピックのセットアップ手順に従います。コアデバイスタイプに適用されるトピックのみを選択します。

トピック

- Raspberry Pi のセットアップ
- Amazon EC2 インスタンスの設定
- 他のデバイスの設定

> **Note**
>
> 構築済みの Docker コンテナで AWS IoT Greengrass 実行されている を使用する方法については、「」を参照してください the section called “Docker コンテナ AWS IoT Greengrass でを実行する”。

Raspberry Pi のセットアップ

このトピックの手順に従って、 を AWS IoT Greengrass コアとして使用する Raspberry Pi を設定します。

> **Tip**
>
> AWS IoT Greengrass には、 AWS IoT Greengrass Core ソフトウェアをインストールするための他のオプションも用意されています。例えば、Greengrass デバイスのセットアップを使用して環境を設定し、最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールできます。または、サポートされている Debian プラットフォームでは、APT パッケージマネージャーを使用して AWS IoT Greengrass Core ソフトウェアをインストールまたはアップグレードできます。詳細については、「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

Raspberry Pi を初めて設定する場合は、以下のすべての手順に従う必要があります。それ以外の場合は、ステップ 9 に進むことができます。ただし、ステップ 2 で推奨されているように、Raspberry Pi をオペレーティングシステムで再イメージ化することをお勧めします。

1. SD Memory Card Formatter などの SD カードフォーマットツールをダウンロードしてインストールします。SD カードをコンピュータに挿入します。プログラムを起動し、SD カードを挿入したドライブを選択します。SD カードのクイックフォーマットを実行できます。
2. Raspbian Buster オペレーティングシステムを `zip` ファイルとしてダウンロードします。
3. SD カード書き込みツール (Etcher など) を使用して、ツールの手順に従い、ダウンロードした `zip` ファイルを SD カードにフラッシュします。オペレーティングシステムイメージが大きいため、このステップに時間がかかることがあります。SD カードをコンピュータから取り出し、microSD カードを Raspberry Pi に挿入します。

4. 最初の起動では、Raspberry Pi をモニター (HDMI 使用)、キーボード、マウスに接続することをお勧めします。次に、Pi をマイクロ USB 電源と接続すると、Raspbian オペレーティングシステムが起動します。

5. 次に進む前に、Pi のキーボードレイアウトを設定できます。これを行うには、右上にある Raspberry アイコンを選択し、[Preferences (設定)]、[Mouse and Keyboard Settings (マウスとキーボードの設定)] の順に選択します。次に、[Keyboard (キーボード)] タブ、[Keyboard Layout (キーボードレイアウト)] の順に選択して適切なキーボードタイプを選択します。

6. 次に、Raspberry Pi を Wi-Fi ネットワークを介してインターネットに接続するか、イーサネットケーブルを介して接続します。

> **Note**
>
> 次に進む前に、Raspberry Pi をコンピュータの接続先と同じネットワークに接続し、コンピュータと Raspberry Pi の両方がインターネットにアクセスできることを確認します。職場の環境やファイアウォールの内側で作業している場合は、必要に応じて Pi とコンピュータをゲストネットワークに接続して両方のデバイスを同じネットワーク上に配置します。ただし、このアプローチではコンピュータがイントラネットなどのローカルネットワークリソースから切断される可能性があります。解決方法の 1 つとして、Pi をゲスト Wi-Fi ネットワークに接続し、コンピュータをゲスト Wi-Fi ネットワークに加えてイーサネットケーブル経由でローカルネットワークに接続できます。この設定では、コンピュータがゲスト Wi-Fi ネットワーク経由で Raspberry Pi に接続し、イーサネットケーブル経由でローカルネットワークリソースに接続できます。

7. リモートで接続するには、SSH を Pi にセットアップする必要があります。Raspberry Pi でターミナルウィンドウを開き、以下のコマンドを実行します。

```
sudo raspi-config
```

次のように表示されます。

下にスクロールし、[Interfacing Options] を選択して [P2 SSH] を選択します。プロンプトが表示されたら、[Yes] を選択します。(Tab キーに続けて Enter キーを使用します)。SSH が有効である必要があります。[OK] を選択します。Tab キーを使用して [完了] を選択し、Enter キーを押します。Raspberry Pi が自動的に再起動しない場合は、次のコマンドを実行します。

```
sudo reboot
```

8. Raspberry Pi のターミナルウィンドウで、以下のコマンドを実行します。

```
hostname -I
```

これは、Raspberry Pi の IP アドレスを返します。

> **Note**
>
> 以下の場合、ECDSA キーフィンガープリントのメッセージ (`Are you sure you want to continue connecting (yes/no)?`) が表示されたら、「`yes`」と入力します。Raspberry Pi のデフォルトパスワードは **`raspberry`** です。

macOS を使用している場合は、ターミナルウィンドウを開き、次のように入力します。

```
ssh pi@IP-address
```

`IP-address` は `hostname -I` コマンドで取得した Raspberry Pi の IP アドレスです。

Windows を使用している場合は、PuTTY をインストールして設定する必要があります。[Connection (接続)] を展開して [Data (データ)] を選択し、[Prompt (プロンプト)] が選択されていることを確認します。

次に、[Session (セッション)] を選択して、Raspberry Pi の IP アドレスを入力し、デフォルトの設定を使用して [Open (開く)] を選択します。

PuTTY セキュリティ警告が表示された場合は、[Yes (はい)] を選択します。

Raspberry Pi のデフォルトのログインとパスワードはそれぞれ **pi** および **raspberry** です。

> **Note**
>
> コンピュータが VPN を使用してリモートネットワークに接続されている場合は、このコンピュータから SSH を使用して Raspberry Pi に接続しにくい場合があります。

9. これで、Raspberry Pi をセットアップする準備ができました AWS IoT Greengrass。最初に、ローカル Raspberry Pi ターミナルウィンドウ、または SSH ターミナルウィンドウから以下のコマンドを実行します。

> **Tip**
>
> AWS IoT Greengrass には、 AWS IoT Greengrass Core ソフトウェアをインストールするための他のオプションも用意されています。例えば、Greengrass デバイスのセットアップを使用して環境を設定し、最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールできます。または、サポートされている Debian プラットフォームでは、APT パッケージマネージャーを使用して AWS IoT Greengrass Core ソフトウェアをインストールまたはアップグレードできます。詳細については、「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

```
sudo adduser --system ggc_user
sudo addgroup --system ggc_group
```

10. Pi デバイスでのセキュリティを向上させるには、起動時にオペレーティングシステムのハードリンクとソフトリンク (symlink) の保護を有効にします。

    a. `98-rpi.conf` ファイルに移動します。

    ```
    cd /etc/sysctl.d
    ls
    ```

    > **Note**
    >
    > `98-rpi.conf` ファイルが表示されない場合は、`README.sysctl` ファイルにある手順に従います。

    b. テキストエディタ (Leafpad、GNU nano、vi など) を使用して、次の 2 行をファイルの末尾に追加します。sudo コマンドを使用してルートとして編集することが必要な場合があります (例: `sudo nano 98-rpi.conf`)。

    ```
    fs.protected_hardlinks = 1
    fs.protected_symlinks = 1
    ```

    c. Pi を再起動します。

    ```
    sudo reboot
    ```

    約 1 分後、SSH を使用して Pi に接続し、次のコマンドを実行して変更を確認します。

    ```
    sudo sysctl -a 2> /dev/null | grep fs.protected
    ```

    `fs.protected_hardlinks = 1` および `fs.protected_symlinks = 1` と表示されるはずです。

11. コマンドラインブートファイルを編集し、メモリ cgroups を有効にしてマウントします。これにより、 AWS IoT Greengrass は Lambda 関数のメモリ制限を設定できます。デフォルトの AWS IoT Greengrass コンテナ化モードで を実行するには、 グループも必要です。

a. boot ディレクトリに移動します。

```
cd /boot/
```

b. テキストエディタを使用して、cmdline.txt を開きます。新しい行としてではなく、既存の行の末尾に以下を追加します。sudo コマンドを使用してルートとして編集することが必要な場合があります (例: sudo nano cmdline.txt)。

```
cgroup_enable=memory cgroup_memory=1
```

c. Pi を再起動します。

```
sudo reboot
```

これで、Raspberry Pi は AWS IoT Greengrassの準備ができた状態になります。

12. オプション。Java 8 ランタイムをインストールします。これはストリームマネージャーに必要です。このチュートリアルではストリームマネージャーを使用しませんが、チュートリアルで使用する [デフォルトのグループ作成] ワークフローでは、デフォルトでストリームマネージャーが有効になります。グループをデプロイする前に、以下のコマンドを使用してコアデバイスに Java 8 ランタイムをインストールするか、またはストリームマネージャーを無効にする必要があります。ストリームマネージャーを無効にする手順については、モジュール 3 を参照してください。

```
sudo apt install openjdk-8-jdk
```

13. 必要なすべての依存関係がそろっていることを確認するには、GitHub のAWS IoT Greengrass サンプルリポジトリから Greengrass 依存関係チェッカーをダウンロードして実行します。これらのコマンドにより、依存関係チェッカースクリプトは Downloads ディレクトリで解凍され、実行されます。

> **Note**
>
> Raspbian カーネルのバージョン 5.4.51 を実行している場合、依存関係チェッカーでエラーが生じる場合があります。このバージョンでは、メモリ cgroups が正常にマウントされません。これにより、コンテナモードで実行される Lambda 関数が正常に動作しない可能性があります。

カーネルの更新の詳細については、「Cgroups not loaded after kernel upgrade」(カーネルのアップグレード後に cgroups がロードされない) を参照してください。

```
cd /home/pi/Downloads
mkdir greengrass-dependency-checker-GGCv1.11.x
cd greengrass-dependency-checker-GGCv1.11.x
wget https://github.com/aws-samples/aws-greengrass-samples/raw/master/greengrass-
dependency-checker-GGCv1.11.x.zip
unzip greengrass-dependency-checker-GGCv1.11.x.zip
cd greengrass-dependency-checker-GGCv1.11.x
sudo modprobe configs
sudo ./check_ggc_dependencies | more
```

`more` が表示された場合は、Spacebar キーを押して別画面のテキストを表示します。

> **Important**
>
> このチュートリアルでは、ローカル Lambda 関数を実行するには Python 3.7 ランタイムが必要です。ストリームマネージャーが有効な場合は、Java 8 ランタイムも必要です。これらのランタイムの前提条件が不足しているという警告が `check_ggc_dependencies` スクリプトによって表示される場合は、続行する前に必ずインストールしてください。その他の欠落しているオプションのランタイム前提条件に関する警告は無視できます。

modprobe コマンドの詳細については、ターミナルで man modprobe を実行してください。

これで、Raspberry Pi の設定は完了です。「the section called “モジュール 2: AWS IoT Greengrass Core ソフトウェアのインストール”」に進みます。

Amazon EC2 インスタンスの設定

AWS IoT Greengrass コアとして使用する Amazon EC2 インスタンスをセットアップするには、このトピックのステップに従います。

> **Tip**
>
> または、環境をセットアップして AWS IoT Greengrass Core ソフトウェアをインストールするスクリプトを使用するには、「」を参照してくださいthe section called “クイックスタート: Greengrass デバイスのセットアップ”。

Amazon EC2 インスタンスを使用してこのチュートリアルを完了できますが、物理ハードウェアで使用する AWS IoT Greengrass のが理想的です。Amazon EC2 インスタンスを使用するのではなく、Raspberry Pi のセットアップをお勧めします。Raspberry Pi を使用している場合は、このトピックのステップを実行する必要はありません。

1. AWS Management Console にサインインし、Amazon Linux AMI を使用して Amazon EC2 インスタンスを起動します。Amazon EC2 インスタンスについては、「Amazon EC2 入門ガイド」を参照してください。
2. Amazon EC2 インスタンスの実行後、ポート 8883 を有効にして受信 MQTT 通信を許可し、他のデバイスが AWS IoT Greengrass コアに接続できるようにします。
   a. Amazon EC2 コンソールのナビゲーションペインで、[Security Group] (セキュリティグループ) を選択します。

b. 先ほど起動したインスタンスのセキュリティグループを選択し、[Inbound rules] (インバウンドルール) タブを選択します。

c. [インバウンドのルールを編集] を選択します。

   ポート 8883 を有効にするには、セキュリティグループにカスタム TCP ルールを追加します。詳細については、「Amazon EC2 ユーザーガイド」の「セキュリティグループへのルールの追加」を参照してください。

d. [インバウンドのルールの編集] ページで、[ルールを追加] を選択し、次の設定を入力して、[保存] を選択します。

   - [タイプ] で、[カスタム TCP ルール] を選択します。
   - [ポート範囲] に **8883** と入力します。
   - [ソース] で、[任意の場所] を選択します。
   - [説明] に **MQTT Communications** と入力します。

3. Amazon EC2 インスタンスに接続します。

   a. ナビゲーションペインで [インスタンス] を選択し、インスタンスを選択して、[接続] を選択します。

   b. [インスタンスへの接続] ページの指示に従って、SSH およびプライベートキーファイルを使用してインスタンスに接続します。

   Windows では PuTTY を、macOS ではターミナルを使用できます。詳細については、「Amazon EC2 ユーザーガイド」の「Linux インスタンスへの接続」を参照してください。

   これで、 AWS IoT Greengrassの Amazon EC2 インスタンスをセットアップする準備ができました。

4. Amazon EC2 インスタンスに接続したら、`ggc_user` アカウントと `ggc_group` アカウントを作成します。

```
sudo adduser --system ggc_user
sudo groupadd --system ggc_group
```

> **Note**
>
> システムで `adduser` コマンドを使用できない場合は、以下のコマンドを使用します。
>
> ```
> sudo useradd --system ggc_user
> ```

5. セキュリティを向上させるには、起動時に Amazon EC2 インスタンスのオペレーティングシステムでハードリンクとソフトリンク (symlink) の保護が有効になっていることを確認します。

> **Note**
>
> ハードリンクとソフトリンクの保護を有効にする手順は、オペレーティングシステムによって異なります。お使いのディストリビューションのドキュメントを参照してください。

   a. 以下のコマンドを実行して、ハードリンクとソフトリンクの保護が有効になっているかどうかを確認します。

```
sudo sysctl -a | grep fs.protected
```

ハードリンクとソフトリンクが 1 に設定されている場合、保護は正しく有効になっています。ステップ 6 に進みます。

> **Note**
>
> ソフトリンクは `fs.protected_symlinks` で表されます。

b. ハードリンクとソフトリンクが 1 に設定されていない場合は、これらの保護を有効にします。システム設定ファイルの場所に移動します。

```
cd /etc/sysctl.d
ls
```

c. 任意のテキストエディタ (Leafpad、GNU nano、vi など) を使用して、システム設定ファイルの最後に以下の 2 行を追加します。Amazon Linux 1 では、これは `00-defaults.conf` ファイルです。Amazon Linux 2 では、これは `99-amazon.conf` ファイルです。ファイルへの書き込みを行えるように (chmod コマンドを使用して) アクセス許可を変更するか、sudo コマンドを使用して root として編集する (sudo nano *`00-defaults.conf`* など) ことが必要な場合があります。

```
fs.protected_hardlinks = 1
fs.protected_symlinks = 1
```

d. Amazon EC2 インスタンスを再起動します。

```
sudo reboot
```

数分後、SSH を使用してインスタンスに接続し、次のコマンドを実行して変更を確認します。

```
sudo sysctl -a | grep fs.protected
```

ハードリンクとソフトリンクが 1 に設定されているはずです。

6. 以下のスクリプトを展開して実行し、Linux コントロールグループ (cgroups) をマウントします。これにより、 AWS IoT Greengrass は Lambda 関数のメモリ制限を設定できます。デフォルトの AWS IoT Greengrass コンテナ化モードで を実行するには、 グループも必要です。

```
curl https://raw.githubusercontent.com/tianon/cgroupfs-
mount/951c38ee8d802330454bdede20d85ec1c0f8d312/cgroupfs-mount > cgroupfs-mount.sh
chmod +x cgroupfs-mount.sh
sudo bash ./cgroupfs-mount.sh
```

これで、Amazon EC2 インスタンスは AWS IoT Greengrassの準備ができた状態になります。

7. オプション。Java 8 ランタイムをインストールします。これはストリームマネージャーに必要です。このチュートリアルではストリームマネージャーを使用しませんが、チュートリアルで使用する [デフォルトのグループ作成] ワークフローでは、デフォルトでストリームマネージャーが有効になります。グループをデプロイする前に、以下のコマンドを使用してコアデバイスに Java 8 ランタイムをインストールするか、またはストリームマネージャーを無効にする必要があります。ストリームマネージャーを無効にする手順については、モジュール 3 を参照してください。

   - Debian ベースのディストリビューションの場合:

     ```
     sudo apt install openjdk-8-jdk
     ```

   - Red Hat ベースのディストリビューションの場合:

     ```
     sudo yum install java-1.8.0-openjdk
     ```

8. 必要なすべての依存関係がそろっていることを確認するには、GitHub のAWS IoT Greengrass サンプルリポジトリから Greengrass 依存関係チェッカーをダウンロードして実行します。これらのコマンドは、Amazon EC2 インスタンスで依存関係チェッカースクリプトをダウンロード、解凍、実行します。

```
mkdir greengrass-dependency-checker-GGCv1.11.x
cd greengrass-dependency-checker-GGCv1.11.x
wget https://github.com/aws-samples/aws-greengrass-samples/raw/master/greengrass-
dependency-checker-GGCv1.11.x.zip
unzip greengrass-dependency-checker-GGCv1.11.x.zip
cd greengrass-dependency-checker-GGCv1.11.x
sudo ./check_ggc_dependencies | more
```

> **Important**
>
> このチュートリアルでは、ローカル Lambda 関数を実行するには Python 3.7 ランタイムが必要です。ストリームマネージャーが有効な場合は、Java 8 ランタイムも必要です。これらのランタイムの前提条件が不足しているという警告が `check_ggc_dependencies` スクリプトによって表示される場合は、続行する前に必ずインストールしてください。その他の欠落しているオプションのランタイム前提条件に関する警告は無視できます。

これで、Amazon EC2 インスタンスの設定は完了です。「the section called “モジュール 2: AWS IoT Greengrass Core ソフトウェアのインストール”」に進みます。

他のデバイスの設定

このトピックのステップに従って、 AWS IoT Greengrass コアとして使用するデバイス (Raspberry Pi 以外) を設定します。

> **Tip**
>
> または、環境をセットアップして AWS IoT Greengrass Core ソフトウェアをインストールするスクリプトを使用するには、「」を参照してくださいthe section called “クイックスタート: Greengrass デバイスのセットアップ”。

を初めて使用する場合は AWS IoT Greengrass、Raspberry Pi または Amazon EC2 インスタンスをコアデバイスとして使用し、デバイスに適したセットアップ手順に従うことをお勧めします。

Yocto プロジェクトを使用してカスタム Linux ベースのシステムを構築する場合は、`meta-aws`プロジェクトから AWS IoT Greengrass Bitbake レシピを使用できます。このレシピは、組み込みアプリケーション用の AWS エッジソフトウェアをサポートするソフトウェアプラットフォームの開発にも役立ちます。Bitbake ビルドにより、デバイスで AWS IoT Greengrass Core ソフトウェアのインストール、設定、自動実行を行えます。

Yocto Project

ハードウェアアーキテクチャに関係なく、組み込みアプリケーション用に Linux ベースのカスタムシステムを構築できるオープンソースのコラボレーションプロジェクト。詳細については、Yocto Project を参照してください。

`meta-aws`

Yocto レシピを提供する AWS マネージドプロジェクト。レシピを使用して、OpenEmbedded と Yocto プロジェクトで構築された Linux ベースのシステムで AWS エッジソフトウェアを開発できます。このコミュニティでサポートされている機能の詳細については、GitHub の `meta-aws` プロジェクトを参照してください。

`meta-aws-demos`

プロジェクトのデモンストレーションを含む AWS マネージド`meta-aws`プロジェクト。統合プロセスのその他の例については、GitHub の `meta-aws-demos` プロジェクトを参照してください。

別のデバイスまたはサポートされているプラットフォームを使用するには、このトピックのステップに従います。

1. コアデバイスが NVIDIA Jetson デバイスの場合、まず JetPack 4.3 インストーラを使用してファームウェアをフラッシュする必要があります。別のデバイスを設定する場合には、ステップ 2 に進みます。

   > **Note**
   >
   > 使用する JetPack インストーラのバージョンは、対象の CUDA Toolkit バージョンに基づいています。以下の手順では、JetPack 4.3 と CUDA Toolkit 10.0 を使用します。デバイスに適したバージョンの使用方法については、NVIDIA のドキュメントの「Jetpack のインストール方法」を参照してください。

   a. Ubuntu 16.04 以降を実行している物理デスクトップで、NVIDIA ドキュメントの「Download and Install JetPack」(JetPack のダウンロードとインストール) (4.3) に従って、JetPack 4.3 インストーラを使用しファームウェアをフラッシュします。

インストーラの指示に従って、Jetson ボードにすべてのパッケージと依存関係をインストールします。Jetson ボードは、Micro-B ケーブルでデスクトップに接続されている必要があります。

b. 標準モードでボードを再起動し、ディスプレイをボードに接続します。

> **Note**
>
> SSH を使用して Jetson に接続する場合は、デフォルトのユーザー名 (**nvidia**) とデフォルトのパスワード (**nvidia**) を使用します。

2. 以下のコマンドを実行して、ユーザー `ggc_user` およびグループ `ggc_group` を作成します。実行するコマンドは、コアデバイスにインストールされたディストリビューションによって異なります。

   - コアデバイスが OpenWrt を実行している場合は、次のコマンドを実行します。

   ```
   opkg install shadow-useradd
   opkg install shadow-groupadd
   useradd --system ggc_user
   groupadd --system ggc_group
   ```

   - それ以外の場合は、以下のコマンドを実行します。

   ```
   sudo adduser --system ggc_user
   sudo addgroup --system ggc_group
   ```

   > **Note**
   >
   > システムで `addgroup` コマンドを使用できない場合は、以下のコマンドを使用します。
   >
   > ```
   > sudo groupadd --system ggc_group
   > ```

3. オプション。Java 8 ランタイムをインストールします。これはストリームマネージャーに必要です。このチュートリアルではストリームマネージャーを使用しませんが、チュートリアルで使用する [デフォルトのグループ作成] ワークフローでは、デフォルトでストリームマネージャーが有効になります。グループをデプロイする前に、以下のコマンドを使用してコアデバイスに

Java 8 ランタイムをインストールするか、またはストリームマネージャーを無効にする必要があります。ストリームマネージャーを無効にする手順については、モジュール 3 を参照してください。

- Debian ベースまたは Ubuntu ベースのディストリビューションの場合:

```
sudo apt install openjdk-8-jdk
```

- Red Hat ベースのディストリビューションの場合:

```
sudo yum install java-1.8.0-openjdk
```

4. 必要なすべての依存関係がそろっていることを確認するには、GitHub のAWS IoT Greengrass サンプルリポジトリから Greengrass 依存関係チェッカーをダウンロードして実行します。これらのコマンドは、依存関係チェッカースクリプトを解凍して実行します。

```
mkdir greengrass-dependency-checker-GGCv1.11.x
cd greengrass-dependency-checker-GGCv1.11.x
wget https://github.com/aws-samples/aws-greengrass-samples/raw/master/greengrass-
dependency-checker-GGCv1.11.x.zip
unzip greengrass-dependency-checker-GGCv1.11.x.zip
cd greengrass-dependency-checker-GGCv1.11.x
sudo ./check_ggc_dependencies | more
```

> **Note**
>
> このcheck_ggc_dependenciesスクリプトは AWS IoT Greengrass 、サポートされているプラットフォームで実行され、特定の Linux システムコマンドが必要です。詳細については、依存関係チェッカーの Readme を参照してください。

5. 依存関係チェッカーの出力で示されているように、必要な依存関係をすべてデバイスにインストールします。不足しているカーネルレベルの依存関係については、カーネルの再コンパイルが必要になる場合があります。Linux コントロールグループ (cgroups) をマウントする場合は、cgroupfs-mount スクリプトを実行できます。これにより、 AWS IoT Greengrass は Lambda 関数のメモリ制限を設定できます。デフォルトの AWS IoT Greengrass コンテナ化モードで を実行するには、 グループも必要です。

   出力にエラーが表示されない場合、 AWS IoT Greengrass はデバイスで正常に実行できます。

> **Important**
>
> このチュートリアルでは、ローカル Lambda 関数を実行するには Python 3.7 ランタイムが必要です。ストリームマネージャーが有効な場合は、Java 8 ランタイムも必要です。これらのランタイムの前提条件が不足しているという警告が `check_ggc_dependencies` スクリプトによって表示される場合は、続行する前に必ずインストールしてください。その他の欠落しているオプションのランタイム前提条件に関する警告は無視できます。

AWS IoT Greengrass 要件と依存関係のリストについては、「」を参照してくださいthe section called “サポートされているプラットフォームと要件”。

モジュール 2: AWS IoT Greengrass Core ソフトウェアのインストール

このモジュールでは、選択したデバイスに AWS IoT Greengrass Core ソフトウェアをインストールする方法について説明します。このモジュールでは、まずGreengrass グループとコアを作成します。次に、コアデバイス上でソフトウェアをダウンロード、設定、および起動します。AWS IoT Greengrass Core ソフトウェアの機能の詳細については、「the section called “ AWS IoT Greengrass コアを設定する”」を参照してください。

開始する前に、選択したデバイスのモジュール 1 のセットアップ手順が完了していることを確認してください。

> **Tip**
>
> AWS IoT Greengrass には、 AWS IoT Greengrass Core ソフトウェアをインストールするための他のオプションも用意されています。例えば、Greengrass デバイスのセットアップを使用して環境を設定し、最新バージョンの AWS IoT Greengrass Core ソフトウェアをインストールできます。または、サポートされている Debian プラットフォームでは、APT パッケージマネージャーを使用して AWS IoT Greengrass Core ソフトウェアをインストールまたはアップグレードできます。詳細については、「the section called “AWS IoT Greengrass Core ソフトウェアのインストール”」を参照してください。

このモジュールの所要時間は 30 分未満です。

トピック

- Greengrass コアとして使用する AWS IoT モノをプロビジョニングする
- コアの AWS IoT Greengrass グループを作成する
- AWS IoT Greengrass コアデバイスにインストールして実行する

Greengrass コアとして使用する AWS IoT モノをプロビジョニングする

Greengrass コアは、 AWS IoT Greengrass Core ソフトウェアを実行してローカル IoT プロセスを管理するデバイスです。Greengrass コアをセットアップするには、 AWS IoTに接続するデバイスや論理要素を表す AWS IoT モノを作成します。デバイスを AWS IoT モノとして登録すると、そのデバイスはデジタル証明書とアクセスを許可するキーを使用できます AWS IoT。AWS IoT ポリシーを使用して、デバイスが AWS IoT および AWS IoT Greengrass サービスと通信できるようにします。

このセクションでは、デバイスを Greengrass コアとして使用する AWS IoT モノとして登録します。

AWS IoT モノを作成するには

1. AWS IoT コンソールに移動します。
2. [Manage] (管理)で、[All devices] (すべてのデバイス) を拡張してから、[Things] (モノ) を選択します。
3. [モノ] ページで [モノを作成する] を選択します。
4. [Creating things] (モノを作成する) ページで、[Create single thing] (単一のモノを作成する) を選択し、[Next] (次へ) を選択します。
5. [Specify thing properties] (モノのプロパティの指定) ページで、以下の作業を行います。
   a. [Thing name] (モノの名前) には、**`MyGreengrassV1Core`**など、デバイスを表す名前を入力します。
   b. [Next (次へ)] を選択します。
6. [Configure device certificate] (デバイス証明書の設定) ページで、[Next] (次へ) を選択します。
7. [Attach policies to certificate] (証明書へのポリシーのアタッチ) ページで、次のいずれかを実行します。
   - コアが必要とする権限をグラントする既存のポリシーを選択し、[Create thing] (モノを作成する) を選択します。

モーダルが開き、デバイスが AWS クラウドとの接続に使用する証明書とキーをダウンロードできます。

- クライアントデバイスにアクセス許可を付与する新しいポリシーを作成してアタッチします。以下の操作を実行します。

  a. [ポリシーを作成] を選択します。

     [ポリシーの作成] ページが新しいタブで開きます。

  b. [ポリシーの作成] ページで、次の操作を行います。

     i. [Policy name] (ポリシー名) には、**GreengrassV1CorePolicy** など、ポリシーを説明する名前を入力します。

     ii. [Policy statements] (ポリシーステートメント) タブの [Policy document] (ポリシードキュメント) で、[JSON] を選択します。

     iii. 次のポリシードキュメントを入力します。このポリシーにより、コアは サービスと通信 AWS IoT Core し、デバイスシャドウと通信し、 AWS IoT Greengrass サービスと通信できます。ユースケースに基づいて、このポリシーのアクセスを制限する方法については、「AWS IoT Greengrass コアデバイスの最小 AWS IoT ポリシー」を参照してください。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
```

```
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

iv. [作成] を選択して、ポリシーを作成します。

c. [Attach policies to certificate] (証明書にポリシーをアタッチ) ページを開き、ブラウザタブに戻ります。以下の操作を実行します。

i. [Policies] (ポリシー) 一覧で、[GreengrassV1CorePolicy] など、作成したポリシーを選択します。

新しいポリシーが表示されない場合は、更新ボタンを押します。

ii. [モノを作成する] を選択します。

モーダルが開き、コアが AWS IoTとの接続に使用する証明書とキーをダウンロードできます。

8. [Attach policies to certificate] (証明書にポリシーをアタッチ) ページを開き、ブラウザタブに戻ります。以下の操作を実行します。

a. [Policies] (ポリシー) 一覧で、[GreengrassV1CorePolicy] など、作成したポリシーを選択します。

新しいポリシーが表示されない場合は、更新ボタンを押します。

b. [モノを作成する] を選択します。

モーダルが開き、コアが AWS IoTとの接続に使用する証明書とキーをダウンロードできます。

9. [Download certificates and keys] (証明書と鍵をダウンロードする) モーダルで、デバイスの証明書をダウンロードします。

> **Important**
>
> [Done] (完了) を選択する前に、セキュリティリソースをダウンロードします。

以下の操作を実行します。

a. [デバイス証明書] で、[ダウンロード] を選択してデバイス証明書をダウンロードします。
b. [パブリックキーファイル] で、[ダウンロード] を選択して証明書のパブリックキーをダウンロードします。
c. [プライベートキーファイル] で、[ダウンロード] を選択して証明書のプライベートキーファイルをダウンロードします。
d. 「AWS IoT デベロッパーガイド」の「サーバー認証」を確認して、適切なルート CA 証明書を選択します。Amazon Trust Services (ATS) エンドポイントと ATS ルート CA 証明書の使用をお勧めします。[ルート CA 証明書] で、ルート CA 証明書の [ダウンロード] を選択します。
e. [完了] をクリックします。

デバイス証明書とキーのファイル名に含まれる共通の証明書 ID を書き留めます。これは後で必要になります。

コアの AWS IoT Greengrass グループを作成する

AWS IoT Greengrass グループには、クライアントデバイス、Lambda 関数、コネクタなど、そのコンポーネントに関する設定やその他の情報が含まれています。グループでは、コンポーネントがどのように相互に連動できるかなど、コアの設定を定義します。

このセクションでは、コアのグループを作成します。

> **Tip**
>
> AWS IoT Greengrass API を使用してグループを作成およびデプロイする例については、GitHub の gg_group_setup リポジトリを参照してください。

コアのグループを作成するには

1. AWS IoT コンソールに移動します。
2. [Manage] (管理) から、[Greengrass devices] (Greengrass デバイス) を展開し、[Groups (V1)] (グループ 〔V1〕) を選択します。

   > **Note**
   >
   > Greengrass デバイスメニューが表示されない場合は、 がサポート AWS リージョン する に変更します AWS IoT Greengrass V1。サポートされているリージョンのリストについては、「AWS 全般のリファレンス」の「AWS IoT Greengrass V1 の エンドポイントとクォータ」を参照してください。が AWS IoT Greengrass V1 利用可能なリージョンで、コアの AWS IoT モノを作成する必要があります。

3. [Greengrass groups] (Greengrass グループ) ページで、[Create group] (グループの作成) を選択します。
4. [Create Greengrass group] (Greengrass グループの作成)] ページで、以下の手順を実行します。

   a. [Greengrass group name] (Greengrass グループ名) に、グループの説明となるような名前 (**`MyGreengrassGroup`** など) を入力します。

   b. Greengrass コアでは、 など、前に作成した AWS IoT ものを選択しますMyGreengrassV1Core。

      コンソールは自動的にモノのデバイス証明書を選択します。

   c. [グループの作成] を選択してください。

AWS IoT Greengrass コアデバイスにインストールして実行する

> **Note**
>
> このチュートリアルでは、Raspberry Pi で AWS IoT Greengrass Core ソフトウェアを実行する手順について説明しますが、サポートされている任意のデバイスを使用できます。

このセクションでは、 AWS IoT Greengrass コアデバイスで Core ソフトウェアを設定、インストール、実行します。

をインストールして実行するには AWS IoT Greengrass

1. このガイドの AWS IoT Greengrass Core ソフトウェアセクションから、 AWS IoT Greengrass Core ソフトウェアインストールパッケージをダウンロードします。コアデバイスの CPU アーキテクチャ、ディストリビューション、OS に最適なパッケージを選択します。
   - Raspberry Pi で、ARMv7L アーキテクチャと Linux オペレーティングシステムに対応したパッケージをダウンロードします。
   - Amazon EC2 インスタンスの場合は、x86_64 アーキテクチャおよび Linux オペレーティングシステムに対応したパッケージをダウンロードします。
   - NVIDIA Jetson TX2 の場合は、Armv8 (AArch64) アーキテクチャおよび Linux オペレーティングシステムに対応したパッケージをダウンロードします。
   - Intel Atom の場合は、x86_64 アーキテクチャおよび Linux オペレーティングシステムに対応したパッケージをダウンロードします。
2. 前のステップでは、コンピュータに 5 つのファイルをダウンロードしました。
   - `greengrass-`*`OS-architecture`*`-1.11.6.tar.gz` – この圧縮ファイルには、 AWS IoT Greengrass コアデバイスで実行される Core ソフトウェアが含まれています。
   - *`certificateId`*`-certificate.pem.crt` - デバイス証明書ファイル。
   - *`certificateId`*`-public.pem.key` - デバイス証明書のパブリックキーファイル。
   - *`certificateId`*`-private.pem.key` - デバイス証明書のプライベートキーファイル。
   - `AmazonRootCA1.pem` - Amazon ルート認証機関 (CA) ファイル。

   このステップでは、これらのファイルをコンピュータからコアデバイスに転送します。以下の操作を実行します。

a. Greengrass コアデバイスの IP アドレスがわからない場合は、コアデバイスでターミナルを開き、次のコマンドを実行します。

> **Note**
>
> このコマンドは、一部のデバイスでは正しい IP アドレスを返さない可能性があります。デバイスの IP アドレスを取得する方法については、デバイスのドキュメントを参照してください。

```
hostname -I
```

b. これらのファイルをコンピュータからコアデバイスに転送します。ファイル転送手順はお使いのコンピュータのオペレーティングシステムによって異なります。ファイルを Raspberry Pi デバイスに転送する方法を示すステップについて、オペレーティングシステムを選択します。

> **Note**
>
> Raspberry Pi では、デフォルトのユーザー名は **pi**、デフォルトのパスワードは **raspberry** です。
> NVIDIA Jetson TX2 では、デフォルトのユーザー名は **nvidia**、デフォルトのパスワードは **nvidia** です。

Windows

圧縮ファイルをコンピュータから Raspberry Pi コアデバイスに転送するには、WinSCP などのツールまたは PuTTY の pscp コマンドを使用してください。pscp コマンドを使用するには、コンピュータでコマンドプロンプトウィンドウを開き、次のコマンドを実行します。

```
cd path-to-downloaded-files
pscp -pw Pi-password greengrass-OS-architecture-1.11.6.tar.gz pi@IP-
address:/home/pi
pscp -pw Pi-password certificateId-certificate.pem.crt pi@IP-address:/home/
pi
pscp -pw Pi-password certificateId-public.pem.key pi@IP-address:/home/pi
```

```
pscp -pw Pi-password certificateId-private.pem.key pi@IP-address:/home/pi
pscp -pw Pi-password AmazonRootCA1.pem pi@IP-address:/home/pi
```

> **Note**
>
> このコマンドのバージョン番号は、 AWS IoT Greengrass Core ソフトウェアパッケージのバージョンと一致する必要があります。

macOS

圧縮ファイルを Mac から Raspberry Pi コアデバイスに転送するには、コンピュータでターミナルウィンドウを開き、以下のコマンドを実行します。*path-to-downloaded-files* は、通常 ~/Downloads です。

> **Note**
>
> 2 つのパスワードの入力を要求される場合があります。その場合、最初のパスワードは Mac の sudo コマンド用で、2 番目は Raspberry Pi のパスワードです。

```
cd path-to-downloaded-files
scp greengrass-OS-architecture-1.11.6.tar.gz pi@IP-address:/home/pi
scp certificateId-certificate.pem.crt pi@IP-address:/home/pi
scp certificateId-public.pem.key pi@IP-address:/home/pi
scp certificateId-private.pem.key pi@IP-address:/home/pi
scp AmazonRootCA1.pem pi@IP-address:/home/pi
```

> **Note**
>
> このコマンドのバージョン番号は、 AWS IoT Greengrass Core ソフトウェアパッケージのバージョンと一致する必要があります。

UNIX-like system

コンピュータから Raspberry Pi コアデバイスに圧縮ファイルを転送するには、コンピュータでターミナルウィンドウを開き、以下のコマンドを実行します。

```
cd path-to-downloaded-files
scp greengrass-OS-architecture-1.11.6.tar.gz pi@IP-address:/home/pi
scp certificateId-certificate.pem.crt pi@IP-address:/home/pi
scp certificateId-public.pem.key pi@IP-address:/home/pi
scp certificateId-private.pem.key pi@IP-address:/home/pi
scp AmazonRootCA1.pem pi@IP-address:/home/pi
```

> **Note**
>
> このコマンドのバージョン番号は、 AWS IoT Greengrass Core ソフトウェアパッケージのバージョンと一致する必要があります。

Raspberry Pi web browser

Raspberry Pi のウェブブラウザを使用して圧縮ファイルをダウンロードした場合は、ファイルは `/home/pi/Downloads`などの Pi の `~/Downloads` フォルダにある必要があります。それ以外の場合は、圧縮ファイルは `/home/pi`などの Pi の `~` フォルダ にある必要があります。

3. Greengrass コアデバイスでターミナルを開き、 AWS IoT Greengrass Core ソフトウェアと証明書を含むフォルダに移動します。*path-to-transferred-files* は、コアデバイスでファイルを転送したパスに置き換えます。例えば、Raspberry Pi で、`cd /home/pi` を実行します。

```
cd path-to-transferred-files
```

4. AWS IoT Greengrass コアデバイスで Core ソフトウェアを解凍します。コアデバイスに転送したソフトウェアアーカイブを解凍するには、次のコマンドを実行します。このコマンドは `-C /` 引数を使用して、コアデバイスのルートフォルダに `/greengrass` フォルダを作成します。

```
sudo tar -xzvf greengrass-OS-architecture-1.11.6.tar.gz -C /
```

> **Note**
>
> このコマンドのバージョン番号は、 AWS IoT Greengrass Core ソフトウェアパッケージのバージョンと一致する必要があります。

5. 証明書とキーを AWS IoT Greengrass Core ソフトウェアフォルダに移動します。次のコマンドを実行して証明書用のフォルダを作成し、証明書とキーをそのフォルダに移動させます。*path-to-transferred-files* をコアデバイス上でファイルを転送したパスに置き換え、*certificateId* をファイル名に含まれる証明書 ID に置き換えます。例えば、Raspberry Pi では、 *path-to-transferred-files* を **/home/pi** に置き換えます。

```
sudo mv path-to-transferred-files/certificateId-certificate.pem.crt /greengrass/
certs
sudo mv path-to-transferred-files/certificateId-public.pem.key /greengrass/certs
sudo mv path-to-transferred-files/certificateId-private.pem.key /greengrass/certs
sudo mv path-to-transferred-files/AmazonRootCA1.pem /greengrass/certs
```

6. AWS IoT Greengrass Core ソフトウェアは、ソフトウェアのパラメータを指定する設定ファイルを使用します。この設定ファイルは、証明書ファイルと使用する AWS クラウド エンドポイントのファイルパスを指定します。このステップでは、 AWS IoT Greengrass コアの Core ソフトウェア設定ファイルを作成します。以下の操作を実行します。

   a. コアの AWS IoT モノの Amazon リソースネーム (ARN) を取得します。以下の操作を実行します。

      i. AWS IoT コンソールで、[Manage] (管理) の [Greengrass devices](Greengrass デバイス) で、[Groups (V1)] (グループ (V1)) を選択します。

      ii. [Greengrass groups] (Greengrassグループ) ページで、以前に作成した [Ggroup] (グループ) を選択します。

      iii. [Overview](概要) で、 [Greengrass core] (Greengrass コア) を選択します。

      iv. コアの詳細ページで、AWS IoT モノの ARN をコピーし、 AWS IoT Greengrass Core 設定ファイルで使用するように保存します。

   b. 現在のリージョン AWS アカウント の の AWS IoT デバイスデータエンドポイントを取得します。デバイスは、このエンドポイントを使用して に AWS IoT モノ AWS として接続します。以下の操作を実行します。

      i. AWS IoT コンソールで、[Settings] (設定) を選択します。

ii. [Device data endpoint](デバイス データ エンドポイント) で、 [Endpoint](エンドポイント)をコピーし、 AWS IoT Greengrass Core 設定ファイルで使用するために保存します。

c. AWS IoT Greengrass Core ソフトウェア設定ファイルを作成します。例えば、次のコマンドを実行し、GNU nano を使用してファイルを作成することができます。

```
sudo nano /greengrass/config/config.json
```

ファイルの内容を次の JSON ドキュメントに置き換えます。

```
{
  "coreThing" : {
    "caPath": "AmazonRootCA1.pem",
    "certPath": "certificateId-certificate.pem.crt",
    "keyPath": "certificateId-private.pem.key",
    "thingArn": "arn:aws:iot:region:account-id:thing/MyGreengrassV1Core",
    "iotHost": "device-data-prefix-ats.iot.region.amazonaws.com",
    "ggHost": "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes"
    }
  },
  "managedRespawn": false,
  "crypto": {
    "caPath": "file:///greengrass/certs/AmazonRootCA1.pem",
    "principals": {
      "SecretsManager": {
        "privateKeyPath": "file:///greengrass/certs/certificateId-
private.pem.key"
      },
      "IoTCertificate": {
        "privateKeyPath": "file:///greengrass/certs/certificateId-
private.pem.key",
        "certificatePath": "file:///greengrass/certs/certificateId-
certificate.pem.crt"
      }
    }
  }
```

```
}
```

次に、以下の操作を実行します。

- Amazon ルート CA 1 とは異なる Amazon ルート CA 証明書をダウンロードした場合は、*AmazonRootCA1.pem* の各インスタンスを Amazon ルート CA ファイルの名前に置き換えます。
- *certificateId* の各インスタンスを証明書とキーファイルの名前に含まれる証明書 ID に置き換えます。
- *arn:aws:iot:region:account-id:thing/MyGreengrassV1Core*を、以前に保存したコアのモノの ARN に置き換えます。
- *MyGreengrassV1core* をコアのモノの名前に置き換えます。
- *device-data-prefix-ats.iot.region.amazonaws.com* 「https://」を、前に保存した AWS IoT デバイスデータエンドポイントに置き換えます。
- *[region]* (リージョン) をお客様の AWS リージョンに置き換えてください。

設定ファイルで指定できる設定オプションの詳細については、「AWS IoT Greengrass コア設定ファイル」を参照してください。

7. コアデバイスがインターネットに接続されていることを確認します 次に、コアデバイスで AWS IoT Greengrass を起動します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

「`Greengrass successfully started`」メッセージが表示されます。PID を書き留めておきます。

> **Note**
>
> システム起動 AWS IoT Greengrass 時に起動するようにコアデバイスをセットアップするには、「」を参照してくださいthe section called “システムの起動時に Greengrass を開始する”。

次のコマンドを実行して、 AWS IoT Greengrass Core ソフトウェア (Greengrass デーモン) が機能していることを確認します。*PID-number* は、PID に置き換えます。

```
ps aux | grep PID-number
```

実行中の Greengrass デーモンへのパスを持つ PID のエントリが表示されます (例: `/greengrass/ggc/packages/1.11.6/bin/daemon`)。問題が発生した場合は AWS IoT Greengrass、「」を参照してください[トラブルシューティング](#)。

モジュール 3 (パート 1): での Lambda 関数 AWS IoT Greengrass

このモジュールでは、 AWS IoT Greengrass コアデバイスから MQTT メッセージを送信する Lambda 関数を作成してデプロイする方法を説明します。モジュールでは、Lambda 関数の設定、MQTT メッセージングを許可するために使用するサブスクリプション、およびコアデバイスへのデプロイについて説明します。

[モジュール 3 (パート 2)](#) では、 AWS IoT Greengrass コアで実行されているオンデマンドの Lambda 関数と存続期間の長い Lambda 関数の違いについて説明します。

開始する前に、[モジュール 1](#) と[モジュール 2](#) を完了し、実行中の AWS IoT Greengrass コアデバイスがあることを確認してください。

> **Tip**
>
> または、コアデバイスをセットアップするスクリプトを使用する場合は、「[the section called “クイックスタート: Greengrass デバイスのセットアップ”](#)」を参照してください。このスクリプトは、このモジュールで使用される Lambda 関数を作成してデプロイすることもできます。

このモジュールは完了までに約 30 分かかります。

トピック

- [Lambda 関数の作成とパッケージ化](#)
- [の Lambda 関数を設定する AWS IoT Greengrass](#)
- [Greengrass コアデバイスにクラウド設定をデプロイする](#)
- [Lambda 関数がコアデバイスで実行されていることを確認する](#)

Lambda 関数の作成とパッケージ化

このモジュールの Python Lambda 関数の例では、Python 用の AWS IoT Greengrass Core SDK を使用して MQTT メッセージを発行しています。

このステップでは、次の操作を行います。

- AWS IoT Greengrass Core SDK for Python をコンピュータ (AWS IoT Greengrass コアデバイスではない) にダウンロードします。
- Lambda 関数コードと依存関係を含む関数デプロイパッケージを作成します。
- Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージをアップロードします。
- Lambda 関数のバージョンを発行し、そのバージョンを指すエイリアスを作成します。

このモジュールを完了するには、Python 3.7 をコアデバイスにインストールする必要があります。

1. AWS IoT Greengrass Core SDK ダウンロードページから、 AWS IoT Greengrass Core SDK for Python をコンピュータにダウンロードします。
2. ダウンロードしたパッケージを解凍して、Lambda 関数コードおよび SDK を取得します。

   このモジュールの Lambda 関数では、以下を使用します。

   - `examples\HelloWorld` の `greengrassHelloWorld.py` ファイル。このファイルに、Lambda 関数コードが記述されています。関数は 5 秒ごとに 2 つのメッセージのいずれかを `hello/world` トピックに発行します。
   - `greengrasssdk` フォルダ。これは SDK です。
3. `greengrassHelloWorld.py` を含む `HelloWorld` フォルダに `greengrasssdk` フォルダをコピーします。
4. Lambda 関数デプロイパッケージを作成するには、`greengrassHelloWorld.py` と `greengrasssdk` フォルダを `hello_world_python_lambda.zip` という名前の zip 圧縮ファイルに保存します。py ファイルと `greengrasssdk` フォルダはディレクトリのルートにある必要があります。

UNIX 互換システム (Mac のターミナルを含む) では、次のコマンドを使用してファイルとフォルダをパッケージ化できます。

```
zip -r hello_world_python_lambda.zip greengrasssdk greengrassHelloWorld.py
```

> **Note**
>
> ディストリビューションによっては、必要に応じて最初に `zip` をインストールします。そのために、`sudo apt-get install zip` などを実行します。(インストールコマンドはお客様のディストリビューションと異なる場合があります)。

これで、Lambda 関数を作成して、デプロイパッケージをアップロードする準備ができました。

5. Lambda コンソールを開き、[関数の作成] を選択します。
6. Author from scratch (製作者を最初から) を選択します。
7. 関数に **`Greengrass_HelloWorld`** という名前を付け、残りのフィールドを以下のように設定します。

   - [ランタイム] で [Python 3.7] を選択します。
   - [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

   [Create function (関数の作成)] を選択します。
8. Lambda 関数のデプロイパッケージをアップロードします。

   a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

b. [アップロード] を選択し、`hello_world_python_lambda.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

- [ランタイム] で [Python 3.7] を選択します。
- [ハンドラ] に **`greengrassHelloWorld.function_handler`** と入力します。

d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コンソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

9. Lambda 関数を発行します。

a. ページの上部の [Actions] (アクション) メニューから、[Publish new version] (新しいバージョンを発行) を選択します。

b. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。

10. Lambda 関数のバージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

a. ページ上部の [Actions] (アクション) メニューから、[Create alias] (エイリアスの作成) を選択します。

b. エイリアスに **GG_HelloWorld** という名前を付け、バージョンを **1** (先ほど発行したバージョンに対応) に設定して、[Save] (保存) を選択します。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

の Lambda 関数を設定する AWS IoT Greengrass

これで、 AWS IoT Greengrassの Lambda 関数を設定する準備が整いました。

このステップでは、次の操作を行います。

- AWS IoT コンソールを使用して、Lambda 関数を Greengrass グループに追加します。
- Lambda 関数のグループ固有の設定を構成します。
- Lambda 関数が MQTT メッセージを AWS IoTに発行できるようにするサブスクリプションをグループに追加します。
- グループのローカルログ設定を構成します。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [Greengrass groups] (Greengrass グループ) で、モジュール 2 で作成したグループを選択します。

3. グループの設定ページで、[Lambda functions] (Lambda 関数) タブを選択してから、[My Lambda functions] (自分の Lambda 関数) セクションをスクロールして、[Add Lambda function] (Lambda 関数を追加) を選択します。
4. 前のステップで作成した Lambda 関数の名前を選択します (エイリアス名ではなく、Greengrass_HelloWorld)。
5. このバージョンでは、Alias: GG_HelloWorld を選択します。
6. [Lambda function configuration] (Lambda 関数の設定) セクションで、次のように変更します。

   - [System user and group] (システムユーザーとグループ) を [Use group default] (グループデフォルトの使用) に設定します。
   - Lambda function containerization (Lambda 関数のコンテナ化) を、[Use group default] (グループデフォルトの使用) に設定します。
   - [タイムアウト] を 25 秒に設定します。この Lambda 関数は、各呼び出しの前に 5 秒間スリープします。
   - [Pinned] (固定) で、[True] を選択します。

   > **Note**
   >
   > 存続期間の長い (または固定された) Lambda 関数は、 AWS IoT Greengrass の起動後に自動的に起動し、独自のコンテナで実行し続けます。これはオンデマンド Lambda 関数とは対照的です。この関数は呼び出されたときに開始し、実行するタスクが残っていないときに停止します。詳細については、「the section called “ライフサイクル設定”」を参照してください。

7. [Add Lambda function] (Lambda 関数の追加) を選択して、変更を保存します。Lambda 関数のプロパティについては、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。

   次に、 AWS IoT Coreに MQTT メッセージを送信することを Lambda 関数に許可するサブスクリプションを作成します。

   Greengrass Lambda 関数を使用して MQTT メッセージを以下と交換することができます。

   - Greengrass グループ内のデバイス。
   - グループのコネクタ。

- グループ内の他の Lambda 関数。
- AWS IoT Core.
- ローカルシャドウサービス。詳細については、「the section called “モジュール 5: デバイスシャドウの操作”」を参照してください。

グループはサブスクリプションを使用して、これらのエンティティが互いに通信する方法を制御します。サブスクリプションは、予測可能なやり取りとセキュリティの層を提供します。

サブスクリプションはソース、ターゲット、およびトピックで構成されます。ソースはメッセージの送信元です。ターゲットはメッセージの送信先です。このトピックでは、ソースからターゲットに送信されるデータをフィルタリングできます。ソースまたはターゲットは、Greengrass デバイス、Lambda 関数、コネクタ、デバイスシャドウ、 AWS IoT Coreのいずれかです。

> **Note**
>
> サブスクリプションは、メッセージが送信元から宛先への特定の方向に流れるという意味でコントロールされます。双方向通信を許可するには、2 つのサブスクリプションを設定する必要があります。

> **Note**
>
> 現在、サブスクリプショントピックフィルタでは、1 つのトピック内で 2 つ以上の + 文字を使用することはできません。トピックフィルタでは、トピックの末尾に加えることのできる # 文字は 1 つのみです。

`Greengrass_HelloWorld` Lambda 関数は のhello/worldトピックにのみメッセージを送信するため AWS IoT Core、Lambda 関数から 1 つのサブスクリプションを作成するだけで済みます AWS IoT Core。これは、次の手順で作成します。

8. グループの設定ページで、[Subscriptions] (サブスクリプション) タブを選択し、[Add subscription] (サブスクリプションの追加) を選択します。

   AWS CLIを使用したサブスクリプションの作成例については、「AWS CLI Command Reference」(コマンドリファレンス) の「create-subscription-definition」を参照してください。

9. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数) を選択し、[Source] (ソース) に [Greengrass_HelloWorld] を選択します。
10. [Target type] (ターゲットタイプ) には、[Service] (サービス) を選択し、[Target] (ターゲット) には [IoT Cloud] (IoT クラウド) を選択します。
11. [Topic filter] (トピックのフィルター) には、**`hello/world`** と入力し、[Create subscription] (サブスクリプションの作成) を選択します。
12. グループのログ記録設定を定義します。このチュートリアルでは、Core デバイスのファイルシステムにログを書き込むように、 AWS IoT Greengrass システムコンポーネントとユーザー定義の Lambda 関数を設定します。

    a. グループの設定ページで、[Logs] (ログ) タブを選択します。

    b. [Local logs configuration] (ローカルログ設定) セクションで、[Edit] (編集) を選択します。

    c. [Edit local logs configuration] (ローカルログ設定の編集) ダイアログボックスで、ログレベルとストレージサイズの両方についてデフォルト値を保持し、[Save] (保存) を選択します。

    このチュートリアルの実行時に発生する可能性のある問題をトラブルシューティングするには、ログを使用します。問題のトラブルシューティングを行う場合は、ログレベルを一時的に [デバッグ] に変更できます。詳細については、「the section called “ファイルシステムログへのアクセス”」を参照してください。
13. Java 8 ランタイムがコアデバイスにインストールされていない場合は、インストールするか、ストリームマネージャーを無効にする必要があります。

    > **Note**
    >
    > このチュートリアルではストリームマネージャーを使用しませんが、チュートリアルで使用する [デフォルトのグループ作成] ワークフローでは、デフォルトでストリームマネージャーが有効になります。ストリームマネージャーが有効になっていても Java 8 がインストールされていない場合、グループのデプロイは失敗します。詳細については、「ストリームマネージャーの要件」を参照してください。

    ストリームマネージャーを無効にするには:

    a. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。

    b. [System Lambda functions] (Lambda システム関数) セクションで、[Stream manager] (ストリームマネージャー)、[Edit] (編集) の順に選択します。

c. [無効化)] を選択し、[保存] を選択します。

Greengrass コアデバイスにクラウド設定をデプロイする

1. Greengrass コアデバイスがインターネットに接続されていることを確認します。例えば、ウェブページに正常に移動できるか確認します。
2. コアデバイスで Greengrass デーモンが実行されていることを確認します。コアデバイスのターミナルで以下のコマンドを実行し、デーモンが実行されているかどうかを確認します。必要ならばデーモンを起動します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   出力に `root` で実行中の `/greengrass/ggc/packages/1.11.6/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

   b. 次のようにしてデーモンを開始します。

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```

   これで、Lambda 関数とサブスクリプション設定を Greengrass Core デバイスにデプロイする準備が整いました。

3. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
4. [Greengrass groups] (Greengrass グループ) で、モジュール 2 で作成したグループを選択します。
5. グループ設定ページで、[Deploy] (デプロイ) を選択します。
6. [Lambda functions] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) セクションで、[IP detector] (IP ディテクター) を選択します。
7. [Edit] (編集) を選択し、[Automatically detect and override MQTT broker endpoints] (MQTT ブローカーのエンドポイントを自動的に検出してオーバーライドする) を選択します。これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、 AWS IoT Greengrass は手動で指定されたエンドポイントも

サポートしています。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

最初のデプロイには数分かかる場合があります。デプロイが完了すると、[Deployments] ページの [Status] 列に [Successfully completed] と表示されます。

> **Note**
>
> デプロイステータスは、ページヘッダーのグループ名の下にも表示されます。

トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

Lambda 関数がコアデバイスで実行されていることを確認する

1. AWS IoT コンソール のナビゲーションペインから、[Test] (テスト)、[MQTT test client] (MQTT テストクライアント) の順に選択します。
2. [Subscribe to topic] (トピックにサブスクライブする) タブを選択します。
3. [Topic filter] (トピックのフィルター) に**`hello/world`** を入力して [Additional configuration] (追加設定) を展開します。
4. 次の各フィールドに表示されている情報を入力します。
   - [サービスの品質] で [0] を選択します。
   - [MQTT ペイロード表示] で、[ペイロードを文字列として表示 (より正確)] を選択します。
5. [サブスクライブ] を選択します。

Lambda 関数がデバイスで実行されている場合、関数は以下のようなメッセージを `hello/world` トピックに発行します。

Lambda 関数は引き続き MQTT AWS IoT Greengrass メッセージを`hello/world`トピックに送信しますが、デーモンを停止しないでください。残りのモジュールでは、このデーモンが実行されていることを前提にしています。

グループから関数とサブスクリプションを削除することができます。

- グループ設定ページの [Lambda function] (Lambda 関数) タブで、削除する Lambda 関数を選択し、[Remove] (削除) を選択します。
- グループ設定ページの [Subscriptions] (サブスクリプション) タブからサブスクリプションを選択して、[Delete] (削除) を選択します。

関数とサブスクリプションは、次のグループデプロイ中にコアから削除されます。

モジュール 3 (パート 2): での Lambda 関数 AWS IoT Greengrass

このモジュールでは、 AWS IoT Greengrass コアで実行されているオンデマンドの Lambda 関数と存続期間の長い Lambda 関数の違いについて説明します。

開始する前に、Greengrass Device Setup スクリプトを実行するか、モジュール 1、モジュール 2、およびモジュール 3 (パート 1) を完了していることを確認します。

このモジュールは完了までに約 30 分かかります。

トピック

- Lambda 関数の作成とパッケージ化
- の存続期間の長い Lambda 関数を設定する AWS IoT Greengrass
- 存続期間の長い Lambda 関数のテスト

- オンデマンド Lambda 関数のテスト

Lambda 関数の作成とパッケージ化

このステップでは、次の操作を行います。

- Lambda 関数コードと依存関係を含む関数デプロイパッケージを作成します。
- Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージをアップロードします。
- Lambda 関数のバージョンを発行し、そのバージョンを指すエイリアスを作成します。

1. コンピュータで、モジュール 3-1 の the section called “Lambda 関数の作成とパッケージ化”でダウンロードして抽出した AWS IoT Greengrass Core SDK for Python に移動します。

   このモジュールの Lambda 関数では、以下を使用します。

   - `examples\HelloWorldCounter` の `greengrassHelloWorldCounter.py` ファイル。このファイルに、Lambda 関数コードが記述されています。
   - `greengrasssdk` フォルダ。これは SDK です。
2. Lambda 関数デプロイパッケージを作成します。

   a. `greengrassHelloWorldCounter.py` を含む `HelloWorldCounter` フォルダに `greengrasssdk` フォルダをコピーします。

   b. `greengrassHelloWorldCounter.py` と `greengrasssdk` フォルダを `hello_world_counter_python_lambda.zip` という名前の `zip` ファイルに保存します。py ファイルと `greengrasssdk` フォルダはディレクトリのルートにある必要があります。

   `zip` をインストールしたUNIX 互換システム (Mac のターミナルを含む) の場合は、次のコマンドを使用してファイルとフォルダをパッケージ化できます。

```
zip -r hello_world_counter_python_lambda.zip greengrasssdk
 greengrassHelloWorldCounter.py
```

これで、Lambda 関数を作成して、デプロイパッケージをアップロードする準備ができました。

3. Lambda コンソールを開き、[関数の作成] を選択します。
4. Author from scratch (製作者を最初から) を選択します。
5. 関数に **`Greengrass_HelloWorld_Counter`** という名前を付け、残りのフィールドを以下のように設定します。

   - [ランタイム] で [Python 3.7] を選択します。
   - [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。または、モジュール 3-1 で作成したロールを再利用することもできます。

   [Create function (関数の作成)] を選択します。

6. Lambda 関数のデプロイパッケージをアップロードします。

   a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

   b. [アップロード] を選択し、`hello_world_counter_python_lambda.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

   c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

      - [ランタイム] で [Python 3.7] を選択します。

- [ハンドラ] に **`greengrassHelloWorldCounter.function_handler`** と入力します。

d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コンソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

7. 関数の最初のバージョンを発行します。

   a. ページの上部の [Actions] (アクション) メニューから、[Publish new version] (新しいバージョンを発行) を選択します。[バージョンの説明] に「**`First version`**」と入力します。

   b. [発行] を選択します。

8. 関数バージョンのエイリアスを作成します。

   a. ページ上部の [Actions] (アクション) メニューから、[Create alias] (エイリアスの作成) を選択します。

   b. [名前] に **`GG_HW_Counter`** と入力します。

   c. [バージョン] で、[1] を選択します。

   d. [Save] を選択します。

エイリアスは、Greengrass デバイスがサブスクライブできる Lambda 関数の単一のエンティティを作成します。この方法では、関数が変更されるたびにサブスクリプションを新しい Lambda 関数バージョン番号で更新する必要がありません。

の存続期間の長い Lambda 関数を設定する AWS IoT Greengrass

これで、 AWS IoT Greengrassの Lambda 関数を設定する準備が整いました。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [Greengrass groups] (Greengrass グループ) で、モジュール 2 で作成したグループを選択します。
3. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択し、[My Lambda functions] (自分の Lambda 関数) で [Add] (追加)を選択します。
4. [Lambda function] (Lambda 関数) から[Greengrass_HelloWorld_Counter] を選択します。
5. Lambda 関数のバージョンで、公開したバージョンのエイリアスを選択します。

6. [Timeout (seconds)] (タイムアウト (秒)) には、**25** と入力します。この Lambda 関数は、各呼び出しの前に 20 秒間スリープします。
7. [Pinned] (固定) で、[True] を選択します。
8. 他のすべてのフィールドではデフォルト値を保持し、[Add Lambda function] (Lambda 関数を追加) を選択します。

存続期間の長い Lambda 関数のテスト

存続期間の長い Lambda 関数は、 AWS IoT Greengrass コアが 1 つのコンテナ (またはサンドボックス) で起動して実行されると自動的に起動します。関数ハンドラの外部で定義される変数と事前処理ロジックは、この関数ハンドラの毎回の呼び出しのために保持されます。ハンドラ関数の複数の呼び出しは、前の呼び出しが実行されるまでキューに入れられます。

このモジュールで使用される `greengrassHelloWorldCounter.py` コードは、関数ハンドラの外部で `my_counter` 変数を定義します。

> **Note**
>
> コードは、 AWS Lambda コンソールまたは GitHub の AWS IoT Greengrass Core SDK for Python で表示できます。

このステップでは、Lambda 関数と が MQTT メッセージを AWS IoT 交換できるようにするサブスクリプションを作成します。その後、グループをデプロイして関数をテストします。

1. グループの設定ページで [Subscriptions] (サブスクリプション)、[Add] (追加) の順に選択します。
2. [Source type] (ソースのタイプ) で、[Lambda function] (Lambda 機能) を選択し、次に [Greengrass_HelloWorld_Counter] を選択します。
3. [Target type] (ターゲットタイプ) から、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。
4. [トピックのフィルター] に「**`hello/world/counter`**」と入力します。
5. [Create subscription] を選択します。

   この単一のサブスクリプションは、Lambda 関数から への 1 `Greengrass_HelloWorld_Counter` つの方向にのみ移行します AWS IoT。この Lambda 関

数をクラウドから呼び出す (またはトリガーする) には、反対方向のサブスクリプションを作成する必要があります。

6. 以下の値を使用する別のサブスクリプションを追加するには、ステップ 1 ~ 5 に従ってください。このサブスクリプションにより、Lambda 関数は AWS IoTからのメッセージを受信することができます。このサブスクリプションは、 AWS IoT 関数を呼び出すコンソールからメッセージを送信するときに使用します。

   - ソースは、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。
   - ターゲットは、[Lambda function] (Lambda 関数)、[Greengrass_HelloWorld_Counter] の順に選択します。
   - [トピックのフィルター] に「**`hello/world/counter/trigger`**」と入力します。

   このトピックのフィルターで `/trigger` 拡張を使用しているのは、2 つ作成したサブスクリプション間で相互の干渉を避けるためです。

7. 「コアデバイスへのクラウド設定のデプロイ」の説明に従って Greengrass デーモンが実行されていることを確認します。
8. グループ設定ページで、[Deploy] (デプロイ) を選択します。
9. デプロイが完了したら、 AWS IoT コンソールのホームページに戻り、テストを選択します。
10. 以下のフィールドを設定します。

    - [Subscription topic (サブスクリプショントピック)] で、**`hello/world/counter`** と入力します。
    - [サービスの品質] で [0] を選択します。
    - [MQTT ペイロード表示] で、[ペイロードを文字列として表示 (より正確)] を選択します。

11. [サブスクライブ] を選択します。

    このモジュールのパート 1 とは異なり、`hello/world/counter` へのサブスクライブ後にメッセージは表示されません。これは、`hello/world/counter` トピックに発行する `greengrassHelloWorldCounter.py` コードが、関数が呼び出されたときにのみ実行される関数ハンドラの内部にあるためです。

    このモジュールでは、`hello/world/counter/trigger` トピックで MQTT メッセージを受信したときに呼び出される `Greengrass_HelloWorld_Counter` Lambda 関数を設定しました。

    [Greengrass_HelloWorld_Counter] から [IoT クラウド] へのサブスクリプションにより、この関数は `hello/world/counter` トピックで AWS IoT にメッセージを送信できます。IoT Cloud

to Greengrass_HelloWorld_Counter サブスクリプションにより AWS IoT 、 はhello/world/counter/triggerトピックの 関数にメッセージを送信できます。

12. 存続期間の長いライフサイクルをテストするには、hello/world/counter/trigger トピックにメッセージを発行することで、Lambda 関数を呼び出します。デフォルトのメッセージを使用できます。

> **Note**
>
> この Greengrass_HelloWorld_Counter 関数は、受信したメッセージの内容を無視します。function_handler のコードを実行するだけで、hello/world/counter トピックにメッセージが送信されます。このコードは、GitHub の「AWS IoT Greengrass Core SDK for Python」から確認できます。

hello/world/counter/trigger トピックにメッセージが発行されるたびに、my_counter 変数が増分されます。この呼び出し回数は、Lambda 関数から送信されたメッセージに示されています。関数ハンドラーには 20 秒のスリープサイクル (time.sleep(20)) が含まれているため、ハンドラーを繰り返しトリガーすると、 AWS IoT Greengrass コアからのレスポンスがキューに入れられます。

オンデマンド Lambda 関数のテスト

オンデマンド Lambda 関数は、クラウドベースの AWS Lambda 関数と似ています。オンデマンド Lambda 関数の複数の呼び出しは、同時に実行できます。Lambda 関数の呼び出しは、呼び出し処理に別のコンテナを作成するか、あるいは、リソースで許可される場合には、既存のコンテナを再使用します。関数ハンドラの外部で定義される変数や前処理は、作成したコンテナには保持されません。

1. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
2. [My Lambda functions] (自分の Lambda 関数)で、Greengrass_HelloWorld_Counter[Lambda function](Lambda 関数) を選択します。
3. Greengrass_HelloWorld_Counter [details](詳細) ページで、[Edit] (編集) を選択します。
4. [Pinned] (固定) で、[False]、[Save] (保存) の順に選択します。

5. グループ設定ページで、[Deploy] (デプロイ) を選択します。
6. デプロイが完了したら、 AWS IoT コンソールのホームページに戻り、テストを選択します。
7. 以下のフィールドを設定します。
   - [Subscription topic (サブスクリプショントピック)] で、**`hello/world/counter`** と入力します。
   - [サービスの品質] で [0] を選択します。
   - [MQTT ペイロード表示] で、[ペイロードを文字列として表示 (より正確)] を選択します。

8. [サブスクライブ] を選択します。

> **Note**
>
> サブスクリプション後にメッセージは表示されません。

9. オンデマンドのライフサイクルをテストするには、`hello/world/counter/trigger` トピックにメッセージを発行することで、関数を呼び出します。デフォルトのメッセージを使用できます。

   a. ボタンを押すたびに 5 秒以内に、[Publish] (発行) を 3 回すばやく選択します。

   発行ごとに、関数ハンドラが呼び出され、各呼び出しのコンテナが作成されます。各オンデマンド Lambda 関数に独自のコンテナ/サンドボックスがあるため、3 回関数をトリガーしても呼び出しカウントが増分されることはありません。

b. 約 30 秒待機してから [トピックに発行] を選択します。呼び出しカウントを 2 に増分する必要があります。これは、前の呼び出しで以前に作成されたコンテナが再利用されており、関数ハンドラの外部の前処理変数が保存されていることを示しています。

これで、 AWS IoT Greengrass コア上で実行できる 2 つのタイプの Lambda 関数を理解しました。次のモジュールである[モジュール 4](#) では、ローカル IoT デバイスが AWS IoT Greengrass グループ内でどのようにやり取りできるかを示します。

モジュール 4: AWS IoT Greengrass グループ内のクライアントデバイスを操作する

このモジュールでは、クライアントデバイスまたはデバイスと呼ばれるローカル IoT デバイスが、AWS IoT Greengrass コアデバイスに接続して通信する方法を示します。 AWS IoT Greengrass コアに接続するクライアントデバイスは AWS IoT Greengrass グループの一部であり、 AWS IoT Greengrass プログラミングパラダイムに参加できます。このモジュールでは、あるクライアントデバイスから Greengrass グループの別のクライアントデバイスに Hello World メッセージを送信します。

開始する前に、Greengrass デバイスセットアップスクリプトを実行するか、モジュール 1 とモジュール 2 を完了します。このモジュールは、2 つのシミュレートされたクライアントデバイスを作成します。他のコンポーネントやデバイスは必要ありません。

このモジュールの所要時間は 30 分未満です。

トピック

- AWS IoT Greengrass グループにクライアントデバイスを作成する
- サブスクリプションを設定する
- AWS IoT Device SDK for Python のインストール
- 通信をテストする

AWS IoT Greengrass グループにクライアントデバイスを作成する

このステップでは、Greengrass グループに 2 つのクライアントデバイスを追加します。このプロセスには、デバイスを AWS IoT モノとして登録し、接続を許可する証明書とキーを設定することが含まれます AWS IoT Greengrass。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. グループの設定ページで、[Client device] (クライアントデバイス)、[Associate] (アソシエイト) の順に選択します。
4. クライアントデバイスをこのグループモーダルに関連付けるで、新しい AWS IoT モノを作成するを選択します。

   新しいタブに [Create things] (モノの作成) ページが開きます。
5. [Creating things] (モノを作成する) ページで、[Create a single thing] (単一のモノを作成する) を選択し、[Next] (次へ) を選択します。
6. [Specify thing properties] (モノのプロパティを指定する) ページで、このクライアントデバイスを **`HelloWorld_Publisher`** として登録し、[Next] (次へ) を選択します。
7. [Configure device certificate] (デバイス証明書の設定) ページで [Next] (次へ) を選択します。
8. [Attach policies to certificate] (証明書へのポリシーのアタッチ) を選択し、次のいずれかを実行します。
   - クライアントデバイスが必要とする権限をグラントする既存のポリシーを選択し、[Create thing] (モノを作成する) を選択します。

     モーダルが開き、デバイスが とコアへの接続に使用する証明書 AWS クラウド とキーをダウンロードできます。
   - クライアントデバイスにアクセス許可を付与する新しいポリシーを作成してアタッチします。以下の操作を実行します。

     a. [ポリシーを作成] を選択します。

        [ポリシーの作成] ページが新しいタブで開きます。

     b. [ポリシーの作成] ページで、次の操作を行います。

        i. [Policy name] (ポリシー名) には、**`GreengrassV1ClientDevicePolicy`** など、ポリシーを説明する名前を入力します。

ii. [Policy statements] (ポリシーステートメント) タブの [Policy document] (ポリシードキュメント) で、[JSON] を選択します。

iii. 次のポリシードキュメントを入力します。このポリシーにより、クライアントデバイスは Greengrass コアを検出し、すべての MQTT トピックで通信できます。このポリシーのアクセスを制限する方法については、「AWS IoT Greengrassのデバイス認証と認可」を参照してください。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

iv. [作成] を選択して、ポリシーを作成します。

c. [Attach policies to certificate] (証明書にポリシーをアタッチ) ページを開き、ブラウザタブに戻ります。以下の操作を実行します。

i. [Policies] (ポリシー) 一覧で、[GreengrassV1ClientDevicePolicy] など、作成したポリシーを選択します。

新しいポリシーが表示されない場合は、更新ボタンを押します。

ii. [モノを作成する] を選択します。

モーダルが開き、デバイスが とコアへの接続に使用する証明書 AWS クラウド とキーをダウンロードできます。

9. [Download certificates and keys] (証明書と鍵をダウンロードする) モーダルで、デバイスの証明書をダウンロードします。

> **Important**
>
> [Done] (完了) を選択する前に、セキュリティリソースをダウンロードします。

以下の操作を実行します。

a. [デバイス証明書] で、[ダウンロード] を選択してデバイス証明書をダウンロードします。

b. [パブリックキーファイル] で、[ダウンロード] を選択して証明書のパブリックキーをダウンロードします。

c. [プライベートキーファイル] で、[ダウンロード] を選択して証明書のプライベートキーファイルをダウンロードします。

d. 「AWS IoT デベロッパーガイド」の「サーバー認証」を確認して、適切なルート CA 証明書を選択します。Amazon Trust Services (ATS) エンドポイントと ATS ルート CA 証明書の使用をお勧めします。[ルート CA 証明書] で、ルート CA 証明書の [ダウンロード] を選択します。

e. [完了] をクリックします。

デバイス証明書とキーのファイル名に含まれる共通の証明書 ID を書き留めます。これは後で必要になります。

10. [Associate a client device with this group] (クライアントデバイスをこのグループに関連付る) モーダルを開いたまま、ブラウザタブに戻ります。以下の操作を実行します。

a. [AWS IoT モノの名前] で、作成した [HelloWorld_Publisher] のモノを選択します。

モノが表示されない場合は、更新ボタンをクリックします。

b. [関連付ける] を選択してください。

11. ステップ 3 ~ 10 を繰り返して 2 つめのクライアントデバイスをグループに追加します。

    このクライアントデバイスに **`HelloWorld_Subscriber`** という名前を付けます。お使いのコンピュータにクライアントデバイスの証明書とキーをダウンロードします。ここでも、HelloWorld_Subscriber デバイス用の共通の証明書 ID を書き留めます。

    これで Greengrass グループに次の 2 つのクライアントデバイスがあるはずです。

    - HelloWorld_Publisher
    - HelloWorld_Subscriber

12. これらのクライアントデバイスのセキュリティ認証情報用のフォルダをコンピュータに作成します。証明書とキーをこのフォルダーにコピーします。

サブスクリプションを設定する

このステップでは、HelloWorld_Publisher クライアントデバイスから HelloWorld_Subscriber クライアントデバイスに MQTT メッセージを送信できるようにします。

1. グループの設定ページの [Subscription] (サブスクリプション) タブで、[Add] (追加) を選択します。
2. [Create a subscription] (サブスクリプションの作成) ページで、以下の操作を実行してサブスクリプションを設定します。

    a. [Source type] (ソースタイプ) は、[Client device] (クライアントデバイス)、[HelloWorld_Publisher] の順に選択します。

    b. [Target type] (ターゲットの選択) で、[Client device] (クライアントデバイス)、[HelloWorld_Subscriber] の順に選択します。

    c. [トピックのフィルター] に「**`hello/world/pubsub`**」と入力します。

    > **Note**
    >
    > 前のモジュールからサブスクリプションを削除できます。グループの [Subscriptions] (サブスクリプション) ページで削除するサブスクリプションを選び、[Delete] (削除) を選択します。

    d. [Create subscription] を選択します。

3. Greengrass コアが IP アドレスのリストを公開できるように、自動検出が有効になっていることを確認してください。クライアントデバイスはこの情報を使用してコアを検出します。以下の操作を実行します。

   a. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。

   b. [System Lambda functions] (システム Lambda 関数) から、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。

   c. [Edit IP detector settings] (IP ディテクター設定の編集) で、[Automatically detect and override MQTT broker endpoints] (MQTT ブローカーのエンドポイントを自動的に検出して上書きする)、[Save] (保存) の順に選択します。

4. 「コアデバイスへのクラウド設定のデプロイ」の説明に従って Greengrass デーモンが実行されていることを確認します。

5. グループ設定ページで、[Deploy] (デプロイ) を選択します。

デプロイステータスは、ページヘッダーのグループ名の下に表示されます。デプロイの詳細を表示するには、[Deployments] (デプロイ) タブを選択します。

AWS IoT Device SDK for Python のインストール

クライアントデバイスは、 AWS IoT Device SDK for Python を使用して、 AWS IoT および AWS IoT Greengrass コアデバイスと通信できます (Python プログラミング言語を使用) 。要件を含む詳細については、GitHub の AWS IoT Device SDK for Python Readme を参照してください。

このステップでは、SDK をインストールし、コンピュータ上のシミュレートされたクライアントデバイスによって使用される `basicDiscovery.py` サンプル関数を取得します。

1. 必要なすべてのコンポーネントとともに SDK をコンピュータにインストールするには、オペレーティングシステムを選択します。

   Windows

   1. 昇格されたコマンドプロンプトを開き、次のコマンドを実行します。

      ```
      python --version
      ```

      バージョン情報が返されない場合や、バージョン番号が 2.7 未満 (Python 2) または 3.3 未満 (Python 3) の場合は、「Python のダウンロード」の手順に従って Python 2.7 以上

または Python 3.3 以上をインストールしてください。詳細については、「Windows で Python を使う」を参照してください。

2. AWS IoT Device SDK for Python を zip ファイルとしてダウンロードし、コンピュータの適切な場所に展開します。

   setup.py ファイルを含む展開された aws-iot-device-sdk-python-master フォルダへのファイルパスを書き留めます。次のステップで、このファイルパスは *path-to-SDK-folder* として示されます。

3. 昇格されたコマンドプロンプトで、次のコマンドを実行します。

```
cd path-to-SDK-folder
python setup.py install
```

macOS

1. ターミナルウィンドウを開いて、次のコマンドを実行します。

```
python --version
```

   バージョン情報が返されない場合や、バージョン番号が 2.7 未満 (Python 2) または 3.3 未満 (Python 3) の場合は、「Python のダウンロード」の手順に従って Python 2.7 以上または Python 3.3 以上をインストールしてください。詳細については、「Macintosh で Python を使う」を参照してください。

2. ターミナルウィンドウで、次のコマンドを実行して OpenSSL のバージョンを確認します。

```
python
>>>import ssl
>>>print ssl.OPENSSL_VERSION
```

   OpenSSL バージョンの値を書き留めておきます。

> **Note**
>
> Python 3 を実行している場合は、print(ssl.OPENSSL_VERSION) を使用します。

Python シェルを閉じるには、次のコマンドを実行します。

```
>>>exit()
```

OpenSSL バージョンが 1.0.1 以降である場合、ステップ c に進んでください。そうでない場合は、以下の手順を実行します。

- ターミナルウィンドウから、次のコマンドを実行して、コンピュータが Simple Python Version Management を使用しているかどうかを確認します。

  ```
  which pyenv
  ```

ファイルパスが返された場合、[Using **pyenv**] タブを選択します。何も返されない場合、[Not using**pyenv**] タブを選択します。

Using pyenv

1. 安定している最新の Python バージョンを確認するには、「Python Releases for Max OS X」(または同様のページ) を参照してください。次の例で、この値は *latest-Python-version* として示されています。
2. ターミナルウィンドウから、以下のコマンドを実行します。

   ```
   pyenv install latest-Python-version
   pyenv global latest-Python-version
   ```

   例えば、Python 2 の最新バージョンが 2.7.14 の場合、これらのコマンドは以下のようになります。

   ```
   pyenv install 2.7.14
   pyenv global 2.7.14
   ```

3. ターミナルウィンドウを閉じてから再度開き、以下のコマンドを実行します。

   ```
   python
   >>>import ssl
   >>>print ssl.OPENSSL_VERSION
   ```

OpenSSL のバージョンは 1.0.1 以上でなければなりません。バージョンが 1.0.1 未満の場合、更新は失敗します。pyenv install コマンドと pyenv global コマンドで使用した Python バージョンの値を確認し、もう一度試してください。

4. 次のコマンドを実行して、Python シェルを終了します。

```
exit()
```

Not using pyenv

1. ターミナルウィンドウで、次のコマンドを実行して brew がインストールされているかどうかを確認します。

```
which brew
```

ファイルパスが返されない場合、次のようにして `brew` をインストールします。

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

> **Note**
> インストールプロンプトに従います。Xcode コマンドラインツールのダウンロードには多少時間がかかることがあります。

2. 以下の コマンドを実行します。

```
brew update
brew install openssl
brew install python@2
```

AWS IoT Device SDK for Python には、Python 実行可能ファイルでコンパイルされた OpenSSL バージョン 1.0.1 (またはそれ以降) が必要です。brew install python コマンドでは、この要件を満たす `python2` 実行可能ファイルがインストールされます。`python2` 実行可能ファイルは、`PATH` 環境変数の一部となっている `/usr/local/bin` ディレクトリにインストールされます。これを確認するには、次のコマンドを実行します。

```
python2 --version
```

python2 のバージョン情報が提供されている場合は、次のステップに進みます。それ以外の場合は、シェルプロファイルに次の行を付加することにより、/usr/local/bin 環境変数に PATH パスを追加します。

```
export PATH="/usr/local/bin:$PATH"
```

例えば、.bash_profile を使用している場合や、まだシェルプロファイルがない場合、ターミナルウィンドウから以下のコマンドを実行します。

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
```

次に、シェルプロファイルに source を実行し、python2 --version によりバージョン情報が提供されることを確認します。例えば、.bash_profile を使用する場合、以下のコマンドを実行します。

```
source ~/.bash_profile
python2 --version
```

python2 のバージョン情報が返されます。

3. シェルプロファイルに次の行を追加します。

```
alias python="python2"
```

例えば、.bash_profile を使用している場合や、まだシェルプロファイルがない場合、以下のコマンドを実行します。

```
echo 'alias python="python2"' >> ~/.bash_profile
```

4. 次に、シェルプロファイルに source を実行します。例えば、.bash_profile を使用する場合、以下のコマンドを実行します。

```
source ~/.bash_profile
```

python コマンドを呼び出すと、必要な OpenSSL バージョンを含む Python 実行可能ファイル (python2) が実行されます。

5. 以下の コマンドを実行します。

```
python
 import ssl
 print ssl.OPENSSL_VERSION
```

OpenSSL のバージョンは 1.0.1 以上でなければなりません。

6. Python シェルを終了するには、次のコマンドを実行します。

```
 exit()
```

3. 次のコマンドを実行して、 AWS IoT Device SDK for Python をインストールします。

```
cd ~
git clone https://github.com/aws/aws-iot-device-sdk-python.git
cd aws-iot-device-sdk-python
sudo python setup.py install
```

UNIX-like system

1. のターミナルウィンドウから、以下のコマンドを実行します。

```
python --version
```

バージョン情報が返されない場合や、バージョン番号が 2.7 未満 (Python 2) または 3.3 未満 (Python 3) の場合は、「Python のダウンロード」の手順に従って Python 2.7 以上または Python 3.3 以上をインストールしてください。詳細については、「Unix プラットフォームで Python を使う」を参照してください。

2. ターミナルで、以下のコマンドを実行して OpenSSL のバージョンを確認します。

```
python
>>>import ssl
>>>print ssl.OPENSSL_VERSION
```

OpenSSL バージョンの値を書き留めておきます。

> **Note**
>
> Python 3 を実行している場合は、print(ssl.OPENSSL_VERSION) を使用します。

Python シェルを閉じるには、次のコマンドを実行します。

```
exit()
```

OpenSSL バージョンが 1.0.1 以降である場合、次のステップに進んでください。それ以外の場合は、ディストリビューションの OpenSSL を更新するコマンド (`sudo yum update openssl`、`sudo apt-get update` など) を実行します。

次のコマンドを実行して、OpenSSL のバージョンが 1.0.1 以降であることを確認します。

```
python
>>>import ssl
>>>print ssl.OPENSSL_VERSION
>>>exit()
```

3. 次のコマンドを実行して、 AWS IoT Device SDK for Python をインストールします。

```
cd ~
git clone https://github.com/aws/aws-iot-device-sdk-python.git
cd aws-iot-device-sdk-python
sudo python setup.py install
```

2. AWS IoT Device SDK for Python をインストールしたら、 `samples`フォルダに移動して `greengrass`フォルダを開きます。

   このチュートリアルでは、「the section called “AWS IoT Greengrass グループにクライアントデバイスを作成する”」でダウンロードした証明書とキーを使用する `basicDiscovery.py` サンプル関数をコピーします。

3. HelloWorld_Publisher および HelloWorld_Subscriber デバイスの証明書とキーを含むフォルダに `basicDiscovery.py` をコピーします。

通信をテストする

1. コンピュータと AWS IoT Greengrass コアデバイスが、同じネットワークを使用してインターネットに接続されていることを確認します。

   a. AWS IoT Greengrass コアデバイスで、次のコマンドを実行して IP アドレスを見つけます。

   ```
   hostname -I
   ```

   b. コンピュータで、コアの IP アドレスを使用して次のコマンドを実行します。ping コマンドを停止するには、Ctrl + C を使用できます。

   ```
   ping IP-address
   ```

   次のような出力は、コンピュータと AWS IoT Greengrass コアデバイス間の通信が成功したことを示します (パケット損失 0%)。

   ```
   $ping 176.32.103.205
   PING 176.32.103.205 (176.32.103.205) 56(84) bytes of data.
   64 bytes from 176.32.103.205: icmp_seq=1 ttl=230 time=77.2 ms
   64 bytes from 176.32.103.205: icmp_seq=2 ttl=230 time=77.1 ms
   64 bytes from 176.32.103.205: icmp_seq=3 ttl=230 time=77.1 ms
   64 bytes from 176.32.103.205: icmp_seq=4 ttl=230 time=77.1 ms
   64 bytes from 176.32.103.205: icmp_seq=5 ttl=230 time=77.1 ms
   64 bytes from 176.32.103.205: icmp_seq=6 ttl=230 time=77.1 ms
   ^C
   --- 176.32.103.205 ping statistics ---
   6 packets transmitted, 6 received, 0% packet loss, time 5549ms
   rtt min/avg/max/mdev = 77.107/77.172/77.256/0.361 ms
   ```

   > **Note**
   >
   > 実行中の EC2 インスタンスに ping を実行できない場合は AWS IoT Greengrass、インスタンスのインバウンドセキュリティグループルールで Echo リクエストメッセージの ICMP トラフィックが許可されていることを確認してください。詳細については、「Amazon EC2 ユーザーガイド」の「セキュリティグループへのルールの追加」を参照してください。
   > Windows ホストコンピュータでは、セキュリティが強化された Windows ファイアウォールアプリケーションで、インバウンドエコーリクエストを許可するインバウ

ンドルール ([File and Printer Sharing (Echo Request - ICMPv4-In)] など) を有効にするか作成することが必要になる場合もあります。

2. AWS IoT エンドポイントを取得します。

   a. AWS IoT コンソールのナビゲーションペインから、[Settings] (設定) を選択します。

   b. [Device data endpoint] (デバイスデータエンドポイント) にある[Endpoint] (エンドポイント) の値を書き留めておきます。次の手順では、この値を使用してコマンド内の *AWS_IOT_ENDPOINT* プレースホルダーを置き換えます。

   > **Note**
   >
   > エンドポイントが証明書タイプに対応していることを確認してください。

3. コンピュータ (AWS IoT Greengrass コアデバイスではない) で、2 つのコマンドライン (ターミナルまたはコマンドプロンプト) ウィンドウを開きます。1 つのウィンドウは HelloWorld_Publisher クライアントデバイスを、もう 1 つは HelloWorld_Subscriber クライアントデバイスを表します。

   実行時に、 はエンドポイントの AWS IoT Greengrass コアの場所に関する情報を収集`basicDiscovery.py`しようとします。この情報は、クライアントデバイスがコアを検出して正常に接続すると、保存されます。これで、この先のメッセージとオペレーションがローカル (インターネット接続なし) で実行されるようになります。

   > **Note**
   >
   > MQTT 接続に使用されるクライアント ID は、クライアントデバイスのモノ名と一致する必要があります。`basicDiscovery.py` スクリプトによって、MQTT 接続のクライアント ID が、スクリプト実行時に指定したモノ名に設定されます。
   > スクリプトの詳しい使用方法を確認するには、`basicDiscovery.py` ファイルのあるフォルダから次のコマンドを実行してください。
   >
   > ```
   > python basicDiscovery.py --help
   > ```

4. HelloWorld_Publisher クライアントデバイスのウィンドウから、次のコマンドを実行します。

   - *path-to-certs-folder* を、証明書、キー、および `basicDiscovery.py` を含むフォルダへのパスに置き換えます。

- *AWS_IOT_ENDPOINT* をエンドポイントに置き換えます。
- 2 つの *publisherCertId* インスタンスを、HelloWorld_Publisher クライアントデバイスのファイル名に含まれるクライアント ID に置き換えます。

```
cd path-to-certs-folder
python basicDiscovery.py --endpoint AWS_IOT_ENDPOINT --rootCA AmazonRootCA1.pem
 --cert publisherCertId-certificate.pem.crt --key publisherCertId-private.pem.key
 --thingName HelloWorld_Publisher --topic 'hello/world/pubsub' --mode publish --
message 'Hello, World! Sent from HelloWorld_Publisher'
```

`Published topic 'hello/world/pubsub': {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}` のような、エントリを含む出力が表示されるはずです。

> **Note**
>
> スクリプトが `error: unrecognized arguments` メッセージを返した場合は、`--topic` および `--message` パラメータの一重引用符を二重引用符に変更して、コマンドを再実行します。
>
> 接続の問題のトラブルシューティングを行うには、手動で IP の検出を試すことができます。

```
Published topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 0}

2017-11-13 21:12:26,296 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [puback] event
2017-11-13 21:12:26,297 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [puback] event
2017-11-13 21:12:27,301 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}

2017-11-13 21:12:27,302 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [puback] event
2017-11-13 21:12:27,303 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [puback] event
2017-11-13 21:12:28,305 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 2}

2017-11-13 21:12:28,306 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [puback] event
2017-11-13 21:12:28,307 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [puback] event
2017-11-13 21:12:29,310 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 3}
```

5. HelloWorld_Subscriber クライアントデバイスのウィンドウから、次のコマンドを実行します。

   - *path-to-certs-folder* を、証明書、キー、および `basicDiscovery.py` を含むフォルダへのパスに置き換えます。

- *AWS_IOT_ENDPOINT* をエンドポイントに置き換えます。
- 2 つの *subscriberCertId* インスタンスを、HelloWorld_Subscriber クライアントデバイスのファイル名に含まれるクライアント ID に置き換えます。

```
cd path-to-certs-folder
python basicDiscovery.py --endpoint AWS_IOT_ENDPOINT --rootCA AmazonRootCA1.pem --
cert subscriberCertId-certificate.pem.crt --key subscriberCertId-private.pem.key --
thingName HelloWorld_Subscriber --topic 'hello/world/pubsub' --mode subscribe
```

`Received message on topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}` のような、エントリを含む出力が表示されるはずです。

```
Received message on topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 0}

2017-11-13 21:12:27,435 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [message] event
2017-11-13 21:12:27,435 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [message] event
2017-11-13 21:12:27,436 - AWSIoTPythonSDK.core.protocol.internal.clients - DEBUG - Invoking custom event callback...
Received message on topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}

2017-11-13 21:12:28,320 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [message] event
2017-11-13 21:12:28,324 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [message] event
2017-11-13 21:12:28,324 - AWSIoTPythonSDK.core.protocol.internal.clients - DEBUG - Invoking custom event callback...
Received message on topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 2}

2017-11-13 21:12:29,547 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Produced [message] event
2017-11-13 21:12:29,552 - AWSIoTPythonSDK.core.protocol.internal.workers - DEBUG - Dispatching [message] event
2017-11-13 21:12:29,552 - AWSIoTPythonSDK.core.protocol.internal.clients - DEBUG - Invoking custom event callback...
```

HelloWorld_Publisher ウィンドウを閉じて、メッセージが HelloWorld_Subscriber ウィンドウに表示されないようにします。

企業ネットワークでテストを実行すると、コアへの接続を妨げる可能性があります。回避策として、エンドポイントを手動で入力できます。これにより、`basicDiscovery.py`スクリプトは AWS IoT Greengrass コアデバイスの正しい IP アドレスに接続されます。

エンドポイントを手動で入力するには

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [Greengrass グループ] で、目的のグループを選択します。
3. MQTT ブローカーのエンドポイントを手動で管理するようにコアを設定します。以下の操作を実行します。

   a. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。

b. [System Lambda functions] (システム Lambda 関数) から、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。

c. [Edit IP detector settings] (IP ディテクター設定の編集) で、[Manually manage MQTT broker endpoints] (MQTT ブローカーのエンドポイントを手動で管理する)、[Save] (保存) の順に選択します。

4. コアの MQTT ブローカーエンドポイントを入力します。以下の操作を実行します。

a. [Overview] (概要) で、 [Greengrass core] (Greengrass コア) を選択します。

b. [MQTT broker endpoints] (MQTT ブローカーエンドポイント) で、 [Manage endpoints] (エンドポイントの管理) を選択します。

c. [Add endpoint] (エンドポイントの追加) を選択し、エンドポイントの値が 1 つだけであることを確認します。この値は、 AWS IoT Greengrass コアデバイスのポート 8883 の IP アドレスエンドポイントである必要があります (例: `192.168.1.4`) 。

d. [Update] (更新) を選択します。

モジュール 5: デバイスシャドウの操作

この高度なモジュールでは、クライアントデバイスが AWS IoT Greengrass グループ内のAWS IoT デバイスシャドウとやり取りする方法を示します。シャドウ は、モノの現在または目的の状態に関する情報を保存するための JSON ドキュメントです。このモジュールでは、1 つのクライアントデバイス (`GG_Switch`) が別のクライアントデバイス (`GG_TrafficLight`) のステータスをどのように変更できるか、またこれらのステータスをどのように AWS IoT Greengrass クラウドに同期できるかを学びます。

開始する前に、Greengrass デバイスのセットアップスクリプトを実行するか、モジュール 1 とモジュール 2 を完了していることを確認します。また、クライアントデバイスを AWS IoT Greengrass コアに接続する方法も理解しておく必要があります (モジュール 4) 。他のコンポーネントやデバイスは必要ありません。

このモジュールは完了までに約 30 分かかります。

トピック

- デバイスとサブスクリプションの設定
- 必要なファイルのダウンロード
- 通信をテストする (デバイス同期を無効にする)
- 通信をテストする (デバイス同期を有効にする)

デバイスとサブスクリプションの設定

AWS IoT Greengrass コアがインターネットに接続されている AWS IoT 場合、シャドウを に同期できます。このモジュールで、まずクラウドへのシャドウ同期を行わずにローカルシャドウを使用します。次に、クラウド同期を有効にします。

各クライアントデバイスには、独自の Shadow があります。詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT Device Shadow サービス」を参照してください。

1. グループ設定ページで、[Client devices] (クライアントデバイス) タブを選択します。
2. クライアントデバイスタブから、 AWS IoT Greengrass グループに 2 つの新しいクライアントデバイスを追加します。このプロセスの詳細なステップについては、「the section called “AWS IoT Greengrass グループにクライアントデバイスを作成する”」を参照してください。
   - クライアントデバイスにそれぞれ **GG_Switch** と **GG_TrafficLight** という名前を付けます。
   - 両クライアントデバイス用のセキュリティリソースを生成してダウンロードします。
   - クライアントデバイスのセキュリティリソースのファイル名にある証明書 ID を書き留めます。これらの値は後で使用します。
3. これらのクライアントデバイスのセキュリティ認証情報用のフォルダをコンピュータに作成します。証明書とキーをこのフォルダーにコピーします。
4. クライアントデバイスが AWS クラウドと同期せずに、ローカルシャドウを使用するように設定されていることを確認します。そうでない場合は、クライアントデバイスを選択し、[Sync shadow] (同期シャドウ)、[Disable shadow sync with cloud] (クラウドとのシャドウ同期を無効にする) の順に選択します。
5. 次の表のサブスクリプションをグループに追加します。例えば、最初のサブスクリプションを作成するには、以下のようにします。
   a. グループ設定ページの [Subscription] (サブスクリプション) タブで、[Add] (追加) を選択します。
   b. [Source type] (ソースタイプ) は、[Client device] (クライアントデバイス)、[GG_Switch] の順に選択します。
   c. [Target type] (ターゲットタイプ) は、[Service] (サービス)、[Local Shadow Service] (ローカルシャドウサービス) の順に選択します。
   d. [トピックのフィルター] に「**$aws/things/GG_TrafficLight/shadow/update**」と入力します。

e. [Create subscription] を選択します。

トピックは、表に示されているとおりに正確に入力する必要があります。ワイルドカードを使用してサブスクリプションの一部を統合することはできますが、この方法はお勧めしません。詳細については、「AWS IoT デベロッパーガイド」の「Device Shadow MQTT トピック」を参照してください。

ソース	ターゲット	トピック	メモ
GG_Switch	ローカルシャドウサービス	$aws/things/GG_TrafficLight/shadow/update	GG_Switch はトピックを更新する更新リクエストを送信します。
ローカルシャドウサービス	GG_Switch	$aws/things/GG_TrafficLight/shadow/update/accepted	GG_Switch は、更新リクエストが承認されたかどうかを知る必要があります。
ローカルシャドウサービス	GG_Switch	$aws/things/GG_TrafficLight/shadow/update/rejected	GG_Switch は、更新リクエストが拒否されたかどうかを知る必要があります。
GG_TrafficLight	ローカルシャドウサービス	$aws/things/GG_TrafficLight/shadow/update	GG_TrafficLight は、その状態の更新を更新トピックに送信します。
ローカルシャドウサービス	GG_TrafficLight	$aws/things/GG_TrafficLight/shadow/update/delta	ローカルシャドウサービスは、受信した更新をデルタトピックを通じて GG_TrafficLight に送信します。

ソース	ターゲット	トピック	メモ
ローカルシャドウサービス	GG_TrafficLight	$aws/things/GG_TrafficLight/shadow/update/accepted	GG_TrafficLight は、状態の更新が承認されたかどうかを知る必要があります。
ローカルシャドウサービス	GG_TrafficLight	$aws/things/GG_TrafficLight/shadow/update/rejected	GG_TrafficLight は、状態の更新が拒否されたかどうかを知る必要があります。

[Subscriptions] (サブスクリプション) タブに新しいサブスクリプションが表示されます。

> **Note**
>
> $ 文字の詳細については、「予約されたトピック」を参照してください。

6. Greengrass コアが IP アドレスのリストを公開できるように、自動検出が有効になっていることを確認してください。クライアントデバイスはこの情報を使用してコアを検出します。以下の操作を実行します。
   a. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
   b. [System Lambda functions] (システム Lambda 関数) から、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。
   c. [Edit IP detector settings] (IP ディテクター設定の編集) で、[Automatically detect and override MQTT broker endpoints] (MQTT ブローカーのエンドポイントを自動的に検出して上書きする)、[Save] (保存) の順に選択します。
7. 「コアデバイスへのクラウド設定のデプロイ」の説明に従って Greengrass デーモンが実行されていることを確認します。
8. グループ設定ページで、[Deploy] (デプロイ) を選択します。

必要なファイルのダウンロード

1. まだインストールしていない場合は、 AWS IoT Device SDK for Python をインストールします。手順については、the section called “AWS IoT Device SDK for Python のインストール” のステップ 1 を参照してください。

   この SDK は、クライアントデバイスがコアデバイス AWS IoT と AWS IoT Greengrass 通信するために使用されます。

2. GitHub の TrafficLight サンプルフォルダから、`trafficLight.py` ファイル `lightController.py` とファイルをコンピュータにダウンロードします。GG_Switch および GG_TrafficLight のクライアントデバイス証明書とキーを含むフォルダにそれらを保存します。

   `lightController.py` スクリプトは GG_Switch クライアントデバイスに対応し、`trafficLight.py` スクリプトは GG_TrafficLight クライアントデバイスに対応します。

   > **Note**
   >
   > サンプル Python ファイルは、便宜上 AWS IoT Greengrass Core SDK for Python リポジトリに保存されますが、 AWS IoT Greengrass Core SDK は使用しません。

通信をテストする (デバイス同期を無効にする)

1. コンピュータと AWS IoT Greengrass コアデバイスが、同じネットワークを使用してインターネットに接続されていることを確認します。

   a. AWS IoT Greengrass コアデバイスで、次のコマンドを実行して IP アドレスを見つけます。

```
hostname -I
```

b. コンピュータで、コアの IP アドレスを使用して次のコマンドを実行します。ping コマンドを停止するには、Ctrl + C を使用できます。

```
ping IP-address
```

次のような出力は、コンピュータと AWS IoT Greengrass コアデバイス間の通信が成功したことを示します (パケット損失 0%)。

```
$ping 176.32.103.205
PING 176.32.103.205 (176.32.103.205) 56(84) bytes of data.
64 bytes from 176.32.103.205: icmp_seq=1 ttl=230 time=77.2 ms
64 bytes from 176.32.103.205: icmp_seq=2 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=3 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=4 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=5 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=6 ttl=230 time=77.1 ms
^C
--- 176.32.103.205 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5549ms
rtt min/avg/max/mdev = 77.107/77.172/77.256/0.361 ms
```

> **Note**
>
> 実行中の EC2 インスタンスに ping を実行できない場合は AWS IoT Greengrass、インスタンスのインバウンドセキュリティグループルールで Echo リクエストメッセージの ICMP トラフィックが許可されていることを確認してください。詳細については、「Amazon EC2 ユーザーガイド」の「セキュリティグループへのルールの追加」を参照してください。
>
> Windows ホストコンピュータでは、セキュリティが強化された Windows ファイアウォールアプリケーションで、インバウンドエコーリクエストを許可するインバウンドルール ([File and Printer Sharing (Echo Request - ICMPv4-In)] など) を有効にするか作成することが必要になる場合もあります。

2. AWS IoT エンドポイントを取得します。

   a. AWS IoT コンソールのナビゲーションペインから、[Settings] (設定) を選択します。

b. [Device data endpoint] (デバイスデータエンドポイント) にある[Endpoint] (エンドポイント) の値を書き留めておきます。次の手順では、この値を使用してコマンド内の *AWS_IOT_ENDPOINT* プレースホルダーを置き換えます。

> **Note**
>
> エンドポイントが証明書タイプに対応していることを確認してください。

3. コンピュータ (AWS IoT Greengrass コアデバイスではない) で、2 つのコマンドライン (ターミナルまたはコマンドプロンプト) ウィンドウを開きます。1 つは、GG_Switch クライアントデバイス用のウィンドウで、もう 1 つは GG_TrafficLight クライアントデバイス用のウィンドウを表します。

   a. GG_Switch クライアントデバイスウィンドウから以下のコマンドを実行します。

      - *path-to-certs-folder* を、証明書、キー、および Python ファイルを含むフォルダへのパスに置き換えます。
      - *AWS_IOT_ENDPOINT* をエンドポイントに置き換えます。
      - 2 つの *switchCertId* インスタンスを、GG_Switch クライアントデバイスのファイル名に含まれる証明書 ID に置き換えます。

```
cd path-to-certs-folder
python lightController.py --endpoint AWS_IOT_ENDPOINT --rootCA
 AmazonRootCA1.pem --cert switchCertId-certificate.pem.crt --key switchCertId-
private.pem.key --thingName GG_TrafficLight --clientId GG_Switch
```

   b. GG_TrafficLight クライアントデバイスウィンドウから次のコマンドを実行します。

      - *path-to-certs-folder* を、証明書、キー、および Python ファイルを含むフォルダへのパスに置き換えます。
      - *AWS_IOT_ENDPOINT* をエンドポイントに置き換えます。
      - 2 つの *lightCertId* インスタンスを、GG_TrafficLight クライアントデバイスのファイル名に含まれる証明書 ID に置き換えます。

```
cd path-to-certs-folder
```

```
python trafficLight.py --endpoint AWS_IOT_ENDPOINT --rootCA AmazonRootCA1.pem
 --cert lightCertId-certificate.pem.crt --key lightCertId-private.pem.key --
thingName GG_TrafficLight --clientId GG_TrafficLight
```

20 秒ごとに、スイッチはシャドウステータスを G、Y、R に更新し、次に示されているように、ライトはその新しいステータスを表示します。

GG_Switch 出力

```
{"state":{"desired":{"property":"R"}}}
2018-12-20 12:23:01,446 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
~~~~~~~~~~~Shadow Update Accepted~~~~~~~~~~~~~
Update request with token: 3b22e27c-930d-4c6a-8562-9f86088249f4 accepted!
property: R
~~~~~~~~~~~~~~~~~~~~~~~~~
```

GG_TrafficLight 出力

```
++++++++ Received Shadow Delta ++++++++++
{u'state': {u'property': u'R'}, u'metadata': {u'property': {u'timestamp': 1545337381}}, u'version': 33, u'clientToken':
u'3b22e27c-930d-4c6a-8562-9f86088249f4'}
property: R
version: 33
++++++++++++++++++++++


Light changed to: R
{"state":{"reported":{"property":"R"}}}
2018-12-20 12:23:01,539 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
~~~~~~~~~~~ Shadow Update Accepted ~~~~~~~~~~~~~
Update request with token: f552109f-c1c2-4ae6-a841-8443506eefcb accepted!
property: R
~~~~~~~~~~~~~~~~~~~~~~~~~
```

初めて実行すると、各クライアントデバイススクリプトは AWS IoT Greengrass 検出サービスを実行して (インターネット経由で) AWS IoT Greengrass コアに接続します。クライアントデバイスが AWS IoT Greengrass コアを検出して正常に接続したら、今後のオペレーションをローカルで実行できます。

> **Note**
>
> `lightController.py` および `trafficLight.py` スクリプトは、スクリプトと同じフォルダに作成される `groupCA` フォルダに接続情報を保存します。接続エラーが表示された場合は、`ggc-host` ファイルの IP アドレスが、Core に設定した IP アドレスエンドポイントと一致することを確認してください。

4. AWS IoT コンソールで、 AWS IoT Greengrass グループを選択し、クライアントデバイスタブを選択し、GG_TrafficLight を選択してクライアントデバイスの詳細 AWS IoT ページを開きます。
5. [Device Shadows] (デバイスシャドウ) タブを選択します。GG_Switch の状態の変更後は、このシャドウは更新できません。これは、GG_TrafficLight が[Disable shadow sync with cloud] (クラウドとのシャドウ同期を無効にする) に設定されているためです。
6. GG_Switch (`lightController.py`) のクライアントデバイスウィンドウで、Ctrl + C を押します。GG_TrafficLight (`trafficLight.py`) ウィンドウでステータス変更メッセージの受信が停止されていることがわかります。

   次のセクションでコマンドを実行できるように、これらのウィンドウを開いたままにしてください。

通信をテストする (デバイス同期を有効にする)

このテストでは、 AWS IoTと同期するように GG_TrafficLight デバイスシャドウを構成します。前のテストと同じコマンドを実行しますが、今回は GG_Switch が更新リクエストを送信したときにクラウド内のシャドウ状態が更新されます。

1. AWS IoT コンソールで、 AWS IoT Greengrass グループを選択し、クライアントデバイスタブを選択します。
2. GG_TrafficLight デバイスを選択して、[Sync shadow] (シャドウの同期)を選択して [Enable shadow sync with cloud] (クラウドとのシャドウ同期を有効にする) を選択します。

   デバイスシャドウ同期が更新されたという通知を受信します。
3. グループ設定ページで、[Deploy] (デプロイ) を選択します。
4. 2 つのコマンドラインウィンドウで、GG_Switch と GG_TrafficLight クライアントデバイスについて、前のテストのコマンドを実行します。
5. 次に、 AWS IoT コンソールでシャドウの状態を確認します。 AWS IoT Greengrass グループを選択し、クライアントデバイスタブを選択し、GG_TrafficLight を選択し、デバイスシャドウタブを選択し、クラシックシャドウを選択します。

   GG_TrafficLight シャドウの への同期を有効にしたため AWS IoT、GG_Switch が更新を送信するたびにクラウド内のシャドウ状態を更新する必要があります。この機能を使用して、クライアントデバイスのステータスを AWS IoTに公開することができます。

> **Note**
>
> 必要に応じて、 AWS IoT Greengrass コアログ、特に を表示して問題をトラブルシューティングできますruntime.log。
>
> ```
> cd /greengrass/ggc/var/log
> sudo cat system/runtime.log | more
> ```
>
> GGShadowSyncManager.log と GGShadowService.log を確認することもできます。詳細については、「トラブルシューティング」を参照してください。

クライアントデバイスとサブスクリプションを設定したままにします。これらは次のモジュールで使用します。同じコマンドも実行します。

モジュール 6: 他の AWS サービスへのアクセス

この高度なモジュールでは、 AWS IoT Greengrass コアがクラウド内の他の AWS サービスとやり取りする方法を示します。これは、モジュール 5 のトラフィックライトの例を基に構築されており、シャドウステータスを処理してサマリーを Amazon DynamoDB テーブルにアップロードする、Lambda 関数を追加します。

開始する前に、Greengrass デバイスのセットアップスクリプトを実行するか、モジュール 1 とモジュール 2 を完了していることを確認します。モジュール 5 も完了しておく必要があります。他のコンポーネントやデバイスは必要ありません。

このモジュールは完了までに約 30 分かかります。

> **Note**
>
> このモジュールは、DynamoDB にテーブルを作成して更新します。このモジュールのオペレーションは小さいものが多く、ほとんどはアマゾン ウェブ サービス Free Tier 内に収まり

ますが、一部のステップを実行するとアカウントへの請求が発生する場合があります。料金に関する詳細については、「DynamoDB 料金」を参照してください。

トピック

- グループロールの設定
- Lambda 関数の作成と設定
- サブスクリプションを設定する
- 通信をテストする

グループロールの設定

グループロールは、ユーザーが作成して Greengrass グループにアタッチした IAM ロールです。このロールには、デプロイされた Lambda 関数 (およびその他の AWS IoT Greengrass 機能) が AWS サービスにアクセスするために使用するアクセス許可が含まれています。詳細については、「the section called “Greengrass グループのロール”」を参照してください。

IAM コンソールでグループロールを作成するには、次の手順 (概要) を使用します。

1. 1 つ以上のリソースに対するアクションを許可または拒否するポリシーを作成します。
2. Greengrass サービスを信頼されたエンティティとして使用するロールを作成します。
3. ポリシーをロールにアタッチします。

次に、 AWS IoT コンソールで、Greengrass グループにロールを追加します。

> **Note**
>
> Greengrass グループには 1 つのグループロールがあります。アクセス権限を追加する場合は、アタッチされたポリシーを編集するか、さらにポリシーをアタッチできます。

このステップでは、Amazon DynamoDB テーブルでアクションを記述、作成、更新する許可を与えるアクセス権限ポリシーを作成します。次に、そのポリシーを新しいロールにアタッチして、そのロールを Greengrass グループに関連付けます。

最初に、このモジュールの Lambda 関数に必要なアクセス権限を付与するカスタマー管理ポリシーを作成します。

1. IAM コンソールのナビゲーションペインで、[Policies]、[Create policy] の順に選択します。
2. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。このモジュールの Lambda 関数はこれらのアクセス権限を使用して、`CarStats` という名前の DynamoDB テーブルを作成および更新します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PermissionsForModule6",
            "Effect": "Allow",
            "Action": [
                "dynamodb:DescribeTable",
                "dynamodb:CreateTable",
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/CarStats"
        }
    ]
}
```

3. [次へ: タグ]、[次へ: 確認] の順に選択します。このチュートリアルでは、タグは使用しません。
4. [Name (名前)] に **`greengrass_CarStats_Table`** と入力し、[Create policy (ポリシーの作成)] を選択します。

次に、新しいポリシーを使用するロールを作成します。

5. ナビゲーションペインで ロール を選択してから、ロールを作成する を選択します。
6. [信頼されたエンティティタイプ] から、[AWS サービス] を選択します。
7. [ユースケース] の [その他の AWS サービスのユースケース] で、[Greengrass] を選択し、[Greengrass] を選択して、[次へ] をクリックします。
8. [Permissions policies] (アクセス許可ポリシー) で、新しい **`greengrass_CarStats_Table`** ポリシーを選択したら [Next] (次へ) をクリックします。
9. [ロール名] には、「**`Greengrass_Group_Role`**」と入力します。

10. [説明] に **`Greengrass group role for connectors and user-defined Lambda functions`** と入力します。
11. [ロールの作成] を選択します。

    次に、ロールを Greengrass グループに追加します。

12. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
13. [Greengrass グループ] で、目的のグループを選択します。
14. [Settings] (設定)、[Associate role] (ロールを関連付ける) の順に設定します。
15. ロール一覧から Greengrass_Group_Role を選択してから、[Associate role] (ロールを関連付ける) を選択します。

Lambda 関数の作成と設定

このステップでは、信号を通過する車の台数を追跡する Lambda 関数を作成します。`GG_TrafficLight` のシャドウステータスが `G` に変わるたびに、Lambda 関数は乱数化された車の数 (1〜20) の通過をシミュレートします。`G` ライトが 3 回変わるたびに、最小値や最大値などの基本的な統計情報が Lambda 関数から DynamoDB テーブルに送信されます。

1. コンピュータに `car_aggregator` という名前のフォルダを作成します。
2. GitHub の TrafficLight サンプルフォルダから、`carAggregator.py` ファイルを `car_aggregator` フォルダにダウンロードします。このファイルに、Lambda 関数コードが記述されています。

   > **Note**
   >
   > このサンプル Python ファイルは、便宜上 AWS IoT Greengrass Core SDK リポジトリに保存されますが、 AWS IoT Greengrass Core SDK は使用しません。

3. 米国東部 (バージニア北部) リージョンで作業していない場合は、 を開き`carAggregator.py`、次の行`region_name`で AWS IoT コンソールで現在選択されている AWS リージョン に変更します。サポートされている のリストについては、 AWS IoT Greengrassの AWS リージョン「」を参照してくださいAmazon Web Services 全般のリファレンス。

   ```
   dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
   ```

4. 次のコマンドをコマンドラインウィンドウで実行し、AWS SDK for Python (Boto3) パッケージとその依存関係を `car_aggregator` フォルダにインストールします。Greengrass Lambda 関数は AWS SDK を使用して他の AWS サービスにアクセスします。(Windows の場合は、昇格されたコマンドプロンプトを使用します)。

```
pip install boto3 -t path-to-car_aggregator-folder
```

この結果、ディレクトリの一覧は以下のようになります。

Name	Date modified	Type
bin	12/31/2018 2:27 PM	File folder
boto3	12/31/2018 2:27 PM	File folder
boto3-1.9.71.dist-info	12/31/2018 2:27 PM	File folder
botocore	12/31/2018 2:27 PM	File folder
botocore-1.12.71.dist-info	12/31/2018 2:27 PM	File folder
concurrent	12/31/2018 2:27 PM	File folder
dateutil	12/31/2018 2:27 PM	File folder
docutils	12/31/2018 2:27 PM	File folder
docutils-0.14.dist-info	12/31/2018 2:27 PM	File folder
futures-3.2.0.dist-info	12/31/2018 2:27 PM	File folder
jmespath	12/31/2018 2:27 PM	File folder
jmespath-0.9.3.dist-info	12/31/2018 2:27 PM	File folder
python_dateutil-2.7.5.dist-info	12/31/2018 2:27 PM	File folder
s3transfer	12/31/2018 2:27 PM	File folder
s3transfer-0.1.13.dist-info	12/31/2018 2:27 PM	File folder
six-1.12.0.dist-info	12/31/2018 2:27 PM	File folder
urllib3	12/31/2018 2:27 PM	File folder
urllib3-1.24.1.dist-info	12/31/2018 2:27 PM	File folder
carAggregator.py	12/31/2018 2:25 PM	PY File
six.py	12/31/2018 2:27 PM	PY File
six.pyc	12/31/2018 2:27 PM	Compiled Python ...

5. `car_aggregator` フォルダの内容を `.zip` ファイルに圧縮し、`car_aggregator.zip` という名前を付けます。(フォルダではなく、フォルダの内容を圧縮します。) これが Lambda 関数デプロイパッケージです。

6. Lambda コンソールで、**`GG_Car_Aggregator`** という名前の関数を作成し、残りのフィールドを以下のように設定します。

   - [ランタイム] で [Python 3.7] を選択します。

- [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

[Create function (関数の作成)] を選択します。

7. Lambda 関数のデプロイパッケージをアップロードします。

   a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

   b. [Upload] (アップロード) を選択し、`car_aggregator.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

   c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

      - [ランタイム] で [Python 3.7] を選択します。
      - [ハンドラ] に **`carAggregator.function_handler`** と入力します。

   d. [Save] を選択します。

8. Lambda 関数を発行し、**GG_CarAggregator** という名前のエイリアスを作成します。詳細な手順については、モジュール 3 (パート 1) の「Lambda 関数の発行」ステップと「エイリアスの作成」ステップを参照してください。
9. AWS IoT コンソールで、先ほど作成した Lambda 関数を AWS IoT Greengrass グループに追加します。
   a. グループ設定ページで、[Lambda functions] (Lambda 関数) を選択し、[My Lambda functions] (自分の Lambda 関数) で、[Add] (追加)を選択します。
   b. Lambda function (Lambda 関数) には、[GG_Car_Aggregator] を選択します。
   c. Lambda 関数のバージョンで、公開したバージョンのエイリアスを選択します。
   d. [メモリ制限] に **64 MB** と入力します。
   e. [Pinned] (固定) で、[True] を選択します。
   f. [Add Lambda function] (Lambda 関数の追加) を選択します。

> **Note**
> 以前のモジュールで作成した他の Lambda 関数は削除してかまいません。

サブスクリプションを設定する

このステップでは、GG_TrafficLight シャドウから更新済みの状態情報を GG_Car_Aggregator Lambda 関数に送信するためのサブスクリプションを作成します。このサブスクリプションは、モジュール 5 で作成したサブスクリプションに対する追加されたものです。すべてが、このモジュールに必須のサブスクリプションです。

1. グループ設定ページの [Subscription] (サブスクリプション) タブで、[Add] (追加) を選択します。
2. [Create a subscription] (サブスクリプションの作成) ページで、以下の操作を行います。
   a. [Source type] (ソースタイプ) は、[Service] (サービス)、[Local Shadow Service] (ローカルシャドウサービス) の順に選択します。
   b. [Target type] (ターゲットタイプ) は、[Lambda function] (Lambda 関数)、[GG_Car_Aggregator] の順に選択します。

c. [トピックのフィルター] に「**$aws/things/GG_TrafficLight/shadow/update/documents**」と入力します。

d. [Create subscription] を選択します。

このモジュールには、新しいサブスクリプションと、モジュール 5 で作成したサブスクリプションが必要です。

3. 「コアデバイスへのクラウド設定のデプロイ」の説明に従って Greengrass デーモンが実行されていることを確認します。

4. グループ設定ページで、[Deploy] (デプロイ) を選択します。

通信をテストする

1. コンピュータで 2 つのコマンドラインウィンドウを開きます。モジュール 5 と同様に、1 つのウィンドウは GG_Switch クライアントデバイス用、もう 1 つは GG_TrafficLight クライアントデバイス用です。それらを使用して、モジュール 5 で実行したのと同じコマンドを実行します。

   GG_Switch クライアントデバイスの次のコマンドを実行します。

```
cd path-to-certs-folder
python lightController.py --endpoint AWS_IOT_ENDPOINT --rootCA AmazonRootCA1.pem
 --cert switchCertId-certificate.pem.crt --key switchCertId-private.pem.key --
thingName GG_TrafficLight --clientId GG_Switch
```

   GG_TrafficLight クライアントデバイスに次のコマンドを実行します。

```
cd path-to-certs-folder
python trafficLight.py --endpoint AWS_IOT_ENDPOINT --rootCA AmazonRootCA1.pem --
cert lightCertId-certificate.pem.crt --key lightCertId-private.pem.key --thingName
 GG_TrafficLight --clientId GG_TrafficLight
```

   20 秒ごとに、スイッチはシャドウ状態を G、Y、R に更新し、その新しい状態が信号に表示されます。

2. Lambda 関数の関数ハンドラが 3 回目のグリーンライトごと (3 分ごと) にトリガーされ、新しい DynamoDB レコードが作成されます。`lightController.py` と `trafficLight.py`が 3

分間実行されたら、 に移動し AWS Management Console、DynamoDB コンソールを開きます。

3. AWS リージョン メニューで米国東部 (バージニア北部) を選択します。これは、`GG_Car_Aggregator` 関数がテーブルを作成するリージョンです。
4. ナビゲーションペインで [テーブル] を選択し、[CarStats] テーブルを選択します。
5. [View items] (項目を表示) を選択すると、テーブル内のエントリが表示されます。

   通過した車に関する基本的な統計情報のあるエントリが表示されます (3 分間ごとに 1 つのエントリ)。必要に応じて更新ボタンを選択し、テーブルの更新を確認します。
6. テストに成功しない場合、Greengrass ログでトラブルシューティング情報を検索できます。

   a. root ユーザーに切り替え、`log` ディレクトリに移動します。 AWS IoT Greengrass ログにアクセスするには、ルートアクセス許可が必要です。

   ```
   sudo su
   cd /greengrass/ggc/var/log
   ```

   b. `runtime.log` でエラーがないかどうかを確認します。

   ```
   cat system/runtime.log | grep 'ERROR'
   ```

   c. Lambda 関数によって生成されたログを確認します。

   ```
   cat user/region/account-id/GG_Car_Aggregator.log
   ```

   `lightController.py` および `trafficLight.py` スクリプトは、スクリプトと同じフォルダに作成される `groupCA` フォルダに接続情報を保存します。接続エラーが表示された場合は、`ggc-host` ファイルの IP アドレスが、Core に設定した IP アドレスエンドポイントと一致することを確認してください。

   詳細については、「トラブルシューティング」を参照してください。

これでこのチュートリアルは終了です。これで、 AWS IoT Greengrass プログラミングモデルとその基本的な概念を理解できたはずです。これには、 AWS IoT Greengrass コア、グループ、サブスクリプション、クライアントデバイス、エッジで実行される Lambda 関数のデプロイプロセスが含まれます。

DynamoDB テーブルと Greengrass Lambda 関数とサブスクリプションを削除できます。 AWS IoT Greengrass コアデバイスと AWS IoT クラウド間の通信を停止するには、コアデバイスでターミナルを開き、次のいずれかのコマンドを実行します。

- AWS IoT Greengrass コアデバイスをシャットダウンするには：

```
sudo halt
```

- AWS IoT Greengrass デーモンを停止するには：

```
cd /greengrass/ggc/core/
sudo ./greengrassd stop
```

モジュール 7: ハードウェアセキュリティ統合のシミュレーション

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

この高度なモジュールでは、Greengrass Core 用にシミュレートされたハードウェアセキュリティモジュール (HSM) を設定する方法を示します。この設定で使用している SoftHSM は、PKCS#11 アプリケーションプログラミングインターフェイス (API) を使用する純粋なソフトウェア実装です。このモジュールの目的は、PKCS#11 API のソフトウェアのみの実装について学習と初期テストを行う環境を設定できるようにすることです。このモジュールは、学習と初期テストの目的でのみ提供しています。

この設定を使用して、PKCS#11 互換のサービスによるプライベートキーの保存を試すことができます。ソフトウェアのみの実装の詳細については、「SoftHSM」を参照してください。一般的な要件など、 AWS IoT Greengrass コアでのハードウェアセキュリティの統合の詳細については、「」を参照してくださいthe section called “ハードウェアセキュリティ統合”。

> **Important**
>
> このモジュールは、実験の目的でのみ提供しています。SoftHSM は本番稼働用環境で使用しないことを強くお勧めします。使用した場合、セキュリティが強化されたような錯覚を与えますが、実際の設定におけるセキュリティ上の利点はありません。SoftHSM でのキーの保存は、Greengrass 環境での他のどのシークレット保存方法よりも安全ではありません。
> このモジュールの目的は、実際のハードウェアベース HSM の今後の使用に備えて、PKCS#11 の仕様について学習し、ソフトウェアの初期テストを実行できるようにすることです。

ハードウェア実装は、本番稼働用環境で使用する前に別個に完全にテストする必要があります。SoftHSM で提供されている PKCS#11 実装とハードウェアベースの実装は異なる場合があるためです。

サポートされているハードウェアセキュリティモジュールのオンボーディングについてサポートが必要な場合は、 AWS エンタープライズサポート担当者にお問い合わせください。

開始する前に、Greengrass Device Setup スクリプトを実行するか、入門チュートリアルのモジュール 1 とモジュール 2 を完了していることを確認します。このモジュールでは、コアがすでにプロビジョニングされ、 と通信していることを前提としています AWS。このモジュールは完了までに約 30 分かかります。

SoftHSM ソフトウェアをインストールする

このステップでは、SoftHSM をインストールし、併せて、SoftHSM インスタンスの管理に使用される pkcs11 ツールもインストールします。

- AWS IoT Greengrass コアデバイスのターミナルで、次のコマンドを実行します。

  ```
  sudo apt-get install softhsm2 libsofthsm2-dev pkcs11-dump
  ```

  これらのパッケージの詳細については、「softhsm2 のインストール」、「libsofthsm2-dev のインストール」、「pkcs11-dump のインストール」を参照してください。

  > **Note**
  > このコマンドをシステムで使用するときに問題が発生した場合は、GitHub の「SoftHSM バージョン 2」を参照してください。このサイトでは、ソースからビルドする方法を含め、より多くのインストール情報が提供されています。

SoftHSM を設定する

このステップでは、SoftHSM を設定します。

1. root ユーザーに切り替えます。

```
sudo su
```

2. 手動のページを使って、システム全体の `softhsm2.conf` の場所を見つけます。一般的な場所は `/etc/softhsm/softhsm2.conf` ですが、システムによって異なる場合があります。

```
man softhsm2.conf
```

3. システム全体の場所に softhsm2 構成ファイル用のディレクトリを作成します。この例では、場所が `/etc/softhsm/softhsm2.conf` であることを前提としています。

```
mkdir -p /etc/softhsm
```

4. トークンディレクトリを `/greengrass` ディレクトリに作成します。

> **Note**
>
> このステップを省略すると、softhsm2-util は `ERROR: Could not initialize the library` と報告します。

```
mkdir -p /greengrass/softhsm2/tokens
```

5. トークンディレクトリを設定します。

```
echo "directories.tokendir = /greengrass/softhsm2/tokens" > /etc/softhsm/
softhsm2.conf
```

6. ファイルベースのバックエンドを設定します。

```
echo "objectstore.backend = file" >> /etc/softhsm/softhsm2.conf
```

> **Note**
>
> これらの設定は、実験の目的でのみ提供しています。すべての設定オプションを確認するには、設定ファイルのマニュアルページを読みます。

```
man softhsm2.conf
```

プライベートキーを SoftHSM にインポートする

このステップでは、SoftHSM トークンを初期化し、プライベートキー形式を変換してから、プライベートキーをインポートします。

1. SoftHSM トークンを初期化します。

```
softhsm2-util --init-token --slot 0 --label greengrass --so-pin 12345 --pin 1234
```

> **Note**
>
> プロンプトが表示されたら、12345 の SO ピン、1234 のユーザーピンを入力します。AWS IoT Greengrass は SO (スーパーバイザ) ピンを使用しないため、任意の値を使用できます。
> エラー `CKR_SLOT_ID_INVALID: Slot 0 does not exist` が発生した場合、代わりに次のコマンドを試してください。
>
> ```
> softhsm2-util --init-token --free --label greengrass --so-pin 12345 --pin
> 1234
> ```

2. プライベートキーを、SoftHSM インポートツールで使用できる形式に変換します。このチュートリアルでは、モジュール 2 の入門チュートリアルで取り上げた [Default Group creation] (デフォルトグループの作成) オプションを使って取得したプライベートキーを変換します。

```
openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in hash.private.key -
out hash.private.pem
```

3. プライベートキーを SoftHSM にインポートする ご使用の softhsm2-util のバージョンに応じて、以下のコマンドを 1 つだけ実行します。

   Raspbian softhsm2-util v2.2.0 構文

```
softhsm2-util --import hash.private.pem --token greengrass --label iotkey --id
0000 --pin 12340
```

Ubuntu softhsm2-util v2.0.0 構文

```
softhsm2-util --import hash.private.pem --slot 0 --label iotkey --id 0000 --pin
 1234
```

このコマンドは、スロットを `0` と識別し、キーラベルを `iotkey` と定義します。これらの値は、次のセクションで使用します。

プライベートキーをインポートした後、オプションで `/greengrass/certs` ディレクトリから削除できます。ルート CA とデバイス証明書は必ずこのディレクトリに保存してください。

SoftHSM を使用するように Greengrass Core を設定する

このステップでは、SoftHSM を使用するように Greengrass Core 設定ファイルを変更します。

1. システム上の SoftHSM プロバイダーライブラリ (`libsofthsm2.so`) へのパスを見つけます。

   a. インストールされているライブラリのパッケージのリストを取得します。

   ```
   sudo dpkg -L libsofthsm2
   ```

   `libsofthsm2.so` ファイルは `softhsm` ディレクトリにあります。

   b. このファイルへの完全パス (`/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so` など) をコピーします。この値は後で使用します。

2. Greengrass デーモンを停止します。

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd stop
   ```

3. Greengrass 設定ファイルを開きます。これは、`/greengrass/config` ディレクトリにある `config.json` ファイルです。

> **Note**
>
> この手順の例は、`config.json` ファイルが、モジュール 2 の入門チュートリアルで取り上げた [Default Group creation] (デフォルトグループの作成) オプションを使って取得した形式を使用している前提で説明しています。

4. `crypto.principals` オブジェクトに、以下の MQTT サーバー証明書オブジェクトを挿入します。有効な JSON ファイルを作成するために必要な場所にカンマを追加します。

```
"MQTTServerCertificate": {
  "privateKeyPath": "path-to-private-key"
}
```

5. `crypto` オブジェクトに、次の PKCS11 オブジェクトを挿入します。有効な JSON ファイルを作成するために必要な場所にカンマを追加します。

```
"PKCS11": {
  "P11Provider": "/path-to-pkcs11-provider-so",
  "slotLabel": "crypto-token-name",
  "slotUserPin": "crypto-token-user-pin"
}
```

ファイルは以下のようになります。

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "thingArn" : "arn:partition:iot:region:account-id:thing/core-thing-name",
    "iotHost" : "host-prefix.iot.region.amazonaws.com",
    "ggHost" : "greengrass.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
  "runtime" : {
    "cgroup" : {
      "useSystemd" : "yes"
    }
  },
  "managedRespawn" : false,
```

```
  "crypto": {
    "PKCS11": {
      "P11Provider": "/path-to-pkcs11-provider-so",
      "slotLabel": "crypto-token-name",
      "slotUserPin": "crypto-token-user-pin"
    },
    "principals" : {
      "MQTTServerCertificate": {
        "privateKeyPath": "path-to-private-key"
      },
      "IoTCertificate" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key",
        "certificatePath" : "file:///greengrass/certs/hash.cert.pem"
      },
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      }
    },
    "caPath" : "file:///greengrass/certs/root.ca.pem"
  }
}
```

> **Note**
>
> ハードウェアセキュリティで無線 (OTA) 更新を使用するには、PKCS11 オブジェクトに OpenSSLEngine プロパティも含める必要があります。詳細については、「the section called “OTA 更新を設定する”」を参照してください。

6. crypto オブジェクトを編集します。

   a. PKCS11 オブジェクトを設定します。

      - [P11Provider] に libsofthsm2.so への完全パスを入力します。
      - slotLabel に「greengrass」と入力します。
      - slotUserPin に「1234」と入力します。

   b. principals オブジェクトでプライベートキーへのパスを設定します。certificatePath プロパティは編集しないでください。

- privateKeyPath プロパティに、以下の RFC 7512 PKCS#11 パス (キーのラベル) を入力します。IoTCertificate、SecretsManager、MQTTServerCertificate のプリンシパルに対してこの操作を行います。

```
pkcs11:object=iotkey;type=private
```

c. crypto オブジェクトを確認します。これは次のように表示されます。

```
"crypto": {
  "PKCS11": {
    "P11Provider": "/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so",
    "slotLabel": "greengrass",
    "slotUserPin": "1234"
  },
  "principals": {
    "MQTTServerCertificate": {
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    },
    "SecretsManager": {
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    },
    "IoTCertificate": {
      "certificatePath": "file://certs/core.crt",
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    }
  },
  "caPath": "file://certs/root.ca.pem"
}
```

7. coreThing オブジェクトから caPath、certPath、keyPath 値を削除します これは次のように表示されます。

```
"coreThing" : {
  "thingArn" : "arn:partition:iot:region:account-id:thing/core-thing-name",
  "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
  "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
  "keepAlive" : 600
}
```

> **Note**
>
> このチュートリアルでは、すべてのプリンシパルに対して同じプライベートキーを指定します。ローカル MQTT サーバーのプライベートキーの選択の詳細については、「パフォーマンス」を参照してください。ローカルシークレットマネージャーの詳細については、「Core にシークレットをデプロイする」を参照してください。

設定をテストする

- Greengrass デーモンを開始します。

  ```
  cd /greengrass/ggc/core/
  sudo ./greengrassd start
  ```

  デーモンが正常に起動すると、Core が正しく設定されます。

  これで、PKCS#11 の仕様について学習し、SoftHSM 実装で提供される PKCS#11 API で初期テストを行う準備ができました。

  > **Important**
  >
  > 繰り返しますが、このモジュールは学習およびテスト専用である点を忘れないでください。このモジュールにより、実際に Greengrass 環境のセキュリティ体制は強化されません。
  > 代わりに、このモジュールの目的は、実際のハードウェアベース HSM の今後の使用に備えて学習とテストを開始できるようにすることです。その時点で、ソフトウェアを本番稼働に使用する前にハードウェアベース HSM に対して個別に完全にテストする必要があります。これは、SoftHSM で提供されている PKCS#11 実装とハードウェアベース実装には違いがあるためです。

関連情報

- PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40 John Leiseboer および Robert Griffin 編集。2014 年 11 月 16 日。「OASIS Committee Note 02」http://docs.oasis-

open.org/pkcs11/pkcs11-ug/v2.40/cn02/pkcs11-ug-v2.40-cn02.html 最新バージョン: http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html

- RFC 7512

AWS IoT Greengrass Core ソフトウェアの OTA 更新

AWS IoT Greengrass Core ソフトウェアパッケージには、 AWS IoT Greengrass ソフトウェアの over-the-air (OTA) 更新を実行できる更新エージェントが含まれています。OTA 更新を使用して、最新バージョンの AWS IoT Greengrass Core ソフトウェアまたは OTA 更新エージェントソフトウェアを 1 つ以上の コアにインストールできます。OTA 更新では、コアデバイスが物理的に存在する必要はありません。

可能であれば、OTA 更新を使用することをお勧めします。これらは、更新ステータスと更新履歴を追跡するために使用できるメカニズムを提供します。更新に失敗すると、OTA 更新エージェントは以前のソフトウェアバージョンにロールバックします。

> **Note**
>
> `apt` を使用して AWS IoT Greengrass Core ソフトウェアをインストールする場合、OTA 更新はサポートされていません。このようなインストールでは、`apt` を使用してソフトウェアをアップグレードすることをお勧めします。詳細については、「the section called “APT リポジトリからのインストール”」を参照してください。

OTA 更新により、以下の作業の効率が向上します。

- セキュリティの脆弱性を修正する。
- ソフトウェアの安定性の問題に対処する。
- 新しい機能や改良された機能をデプロイする。

この機能は、AWS IoT ジョブと統合されています。

要件

AWS IoT Greengrass ソフトウェアの OTA 更新には、以下の要件が適用されます。

- Greengrass Core はローカルストレージに 400 MB 以上の空き容量が必要です。OTA 更新エージェントには、 AWS IoT Greengrass Core ソフトウェアのランタイム使用要件の約 3 倍が必要です。詳細については、「Amazon Web Services 全般のリファレンス」の Greengrass core の「Service Quotas」を参照してください。
- Greengrass Core では、 AWS クラウドへの接続が必要です。

- Greengrass コアは、 および で認証するための証明書とキーで正しく設定 AWS IoT Core およびプロビジョニングする必要があります AWS IoT Greengrass。詳細については、「the section called “X.509 証明書”」を参照してください。
- Greengrass コアは、ネットワークプロキシを使用するように設定することはできません。

> **Note**
>
> AWS IoT Greengrass v1.9.3 以降、OTA 更新は、デフォルトのポート 8883 ではなくポート 443 を使用するように MQTT トラフィックを設定するコアでサポートされています。ただし、OTA 更新エージェントはネットワークプロキシを介した更新をサポートしていません。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。

- AWS IoT Greengrass Core ソフトウェアを含むパーティションでは、信頼されたブートを有効にすることはできません。

> **Note**
>
> Core AWS IoT Greengrass ソフトウェアは、信頼できるブートが有効になっているパーティションにインストールして実行できますが、OTA 更新はサポートされていません。

- AWS IoT Greengrass には、 Core ソフトウェアを含む AWS IoT Greengrass パーティションに対する読み取り/書き込みアクセス許可が必要です。
- Greengrass コアを管理するために init システムを使用する場合、OTA 更新を init システムと統合するように設定する必要があります。詳細については、「the section called “init システムとの統合”」を参照してください。
- Amazon S3 URL を AWS IoT Greengrass ソフトウェア更新アーティファクトに事前署名するために使用されるロールを作成する必要があります。 URLs この署名者ロールにより AWS IoT Core 、 はユーザーに代わって Amazon S3 に保存されているソフトウェア更新アーティファクトにアクセスできます。詳細については、「the section called “OTA 更新の IAM アクセス許可”」を参照してください。

OTA 更新の IAM アクセス許可

が AWS IoT Greengrass Core ソフトウェアの新しいバージョンを AWS IoT Greengrass リリースすると、 は OTA 更新に使用される Amazon S3 に保存されているソフトウェアアーティファクト AWS IoT Greengrass を更新します。

には、これらのアーティファクトへのアクセスに使用できる Amazon S3 URL 署名者ロールが含まれている AWS アカウント 必要があります。ロールには、ターゲット のバケットに対する `s3:GetObject`アクションを許可するアクセス許可ポリシーが必要です AWS リージョン。ロールには、信頼されたエンティティとしてロールを引き受けることを `iot.amazonaws.com` に許可する信頼ポリシーも必要です。

アクセス許可ポリシー

ロールのアクセス許可については、 AWS 管理ポリシーを使用するか、カスタムポリシーを作成できます。

- AWS 管理ポリシーを使用する

  GreengrassOTAUpdateArtifactAccess 管理ポリシーは、 AWS IoT Greengrassによって提供されています。現在と将来の両方の でサポートされているすべての Amazon Web Services リージョンでアクセスを許可する場合は AWS IoT Greengrass、このポリシーを使用します。

- カスタムポリシーの作成

  コアがデプロイされるアマゾン ウェブ サービスリージョンを明示的に指定する場合は、カスタムポリシーを作成する必要があります。次のポリシー例では、6 つのリージョンの AWS IoT Greengrass ソフトウェア更新へのアクセスを許可します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToGreengrassOTAUpdateArtifacts",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::us-east-1-greengrass-updates/*",
                "arn:aws:s3:::us-west-2-greengrass-updates/*",
                "arn:aws:s3:::ap-northeast-1-greengrass-updates/*",
                "arn:aws:s3:::ap-southeast-2-greengrass-updates/*",
                "arn:aws:s3:::eu-central-1-greengrass-updates/*",
                "arn:aws:s3:::eu-west-1-greengrass-updates/*"
            ]
        }
    ]
```

```
}
```

信頼ポリシー

ロールにアタッチされた信頼ポリシーは、`sts:AssumeRole` アクションを許可し、`iot.amazonaws.com` をプリンシパルとして定義する必要があります。これにより、 AWS IoT Core は信頼されたエンティティとしてロールを引き受けることができます。ポリシードキュメントの例を次に示します。

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Sid": "AllowIotToAssumeRole",
             "Action": "sts:AssumeRole",
             "Principal": {
                  "Service": "iot.amazonaws.com"
             },
             "Effect": "Allow"
         }
    ]
}
```

さらに、OTA 更新を開始するユーザーには、`greengrass:CreateSoftwareUpdateJob` および `iot:CreateJob` を使用するアクセス許可と、`iam:PassRole` を使用して署名者ロールのアクセス許可を渡すアクセス許可が必要です。IAM ポリシーの例を次に示します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "greengrass:CreateSoftwareUpdateJob"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:CreateJob"
```

```
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn-of-s3-url-signer-role"
        }
    ]
}
```

考慮事項

Greengrass Core ソフトウェアの OTA 更新を開始する前に、Core デバイスと、その Core にローカルに接続されたクライアントデバイスの両方で、Greengrass グループのデバイスへの影響に注意します。

- Core は更新中にシャットダウンされます。
- Core で実行されている Lambda 関数はすべてシャットダウンされます。これらの関数がローカルリソースに書き込むと、それらのリソースは適切にシャットダウンされない限り正しい状態にならない場合があります。
- コアのダウンタイム中、 との接続はすべて失われ AWS クラウド ます。クライアントデバイスによって Core 経由でルーティングされたメッセージは失われます。
- 認証情報キャッシュは失われます。
- Lambda 関数の作業を保留中のキューは失われます。
- 存続期間の長い Lambda 関数は動的な状態情報を失い、保留中の作業はすべて破棄されます。

OTA 更新の実行中、以下の状態情報は保持されます。

- Core 設定
- Greengrass グループ設定
- ローカルシャドウ
- Greengrass のログ
- OTA 更新エージェントログ

Greengrass OTA Update Agent

Greengrass OTA 更新エージェントは、クラウドで作成およびデプロイされた更新ジョブを処理する、デバイス上のソフトウェアコンポーネントです。OTA 更新エージェントは、 AWS IoT Greengrass Core ソフトウェアと同じソフトウェアパッケージに配布されます。エージェントは `/greengrass-root/ota/ota_agent/ggc-ota` にあります。ログを `/var/log/greengrass/ota/ggc_ota.txt` に書き込みます。

> **Note**
>
> *`greengrass-root`* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

OTA 更新エージェントを起動するには、バイナリを手動で実行するか、systemd サービスファイルなどの init スクリプトの一部として統合します。バイナリを手動で実行する場合は、ルートとして実行する必要があります。起動すると、OTA 更新エージェントは からソフトウェア更新ジョブを AWS IoT Greengrass リッスン AWS IoT Core し、順番に実行します。OTA 更新エージェントは、他のすべての AWS IoT ジョブタイプを無視します。

次の抜粋は、OTA 更新エージェントを起動、停止、および再起動する systemd サービスファイルの例を示したものです。

```
[Unit]
Description=Greengrass OTA Daemon

[Service]
Type=forking
Restart=on-failure
ExecStart=/greengrass/ota/ota_agent/ggc-ota

[Install]
WantedBy=multi-user.target
```

更新のターゲットであるコアは、OTA 更新エージェントの 2 つのインスタンスを実行することはできません。実行すると、2 つのエージェントが同じジョブを処理し、それにより競合が発生します。

init システムとの統合

OTA 更新中に、OTA 更新エージェントは Core デバイスのバイナリを再起動します。バイナリが実行されている場合、更新中に init システムが AWS IoT Greengrass Core ソフトウェアまたはエージェントの状態をモニタリングしているときに競合が発生する可能性があります。OTA 更新メカニズムを init モニタリング戦略と統合するために、更新の前後に実行するシェルスクリプトを記述できます。例えば、データをバックアップしたり、デバイスをシャットダウンする前にプロセスを停止したりする `ggc_pre_update.sh` スクリプトを記述できます。

これらのシェルスクリプトを実行するように OTA 更新エージェントに指示するには、config.json ファイルに `"managedRespawn" : true` フラグを含める必要があります。この設定は、次の抜粋に示されています。

```
{
   "coreThing": {
       …
   },
   "runtime": {
       …
   },
   "managedRespawn": true
       …
}
```

OTA 更新による管理された再生成

`managedRespawn` を `true` に設定した OTA 更新では、以下の要件が適用されます。

- 以下のシェルスクリプトが `/greengrass-root/usr/scripts` ディレクトリ内に必要です。
  - `ggc_pre_update.sh`
  - `ggc_post_update.sh`
  - `ota_pre_update.sh`
  - `ota_post_update.sh`
- スクリプトは、成功のリターンコードを返す必要があります。
- スクリプトは root が所有し、root のみが実行できることが必要です。
- `ggc_pre_update.sh` スクリプトで Greengrass デーモンを停止させる必要があります。
- `ggc_post_update.sh` スクリプトで Greengrass デーモンを開始しておく必要があります。

> **Note**
>
> OTA 更新エージェントは自身でプロセスを管理しているため、`ota_pre_update.sh` および `ota_post_update.sh` スクリプトが OTA サービスの停止と開始に対応する必要はありません。

OTA 更新エージェントは `/greengrass-root/usr/scripts` にあるスクリプトを実行します。ディレクトリツリーは次のようになります。

```
<greengrass_root>
|-- certs
|-- config
|   |-- config.json
|-- ggc
|-- usr/scripts
|   |-- ggc_pre_update.sh
|   |-- ggc_post_update.sh
|   |-- ota_pre_update.sh
|   |-- ota_post_update.sh
|-- ota
```

`managedRespawn` が `true` に設定されている場合、OTA 更新エージェントは、ソフトウェア更新の前後に `/greengrass-root/usr/scripts` ディレクトリでスクリプトをチェックします。スクリプトがない場合は、更新は失敗します。 AWS IoT Greengrass はスクリプトの内容については検証しません。ベストプラクティスとして、スクリプトが正しく機能することを確認した上で、エラーの発生時には適切な終了コードを発行するようにしてください。

AWS IoT Greengrass Core ソフトウェアの OTA 更新の場合：

- 更新を開始する前に、エージェントは `ggc_pre_update.sh` スクリプトを実行します。このスクリプトは、OTA 更新エージェントが AWS IoT Greengrass Core ソフトウェアの更新を開始する前に実行する必要があるコマンドに使用します。たとえば、データのバックアップや実行中のプロセス停止などに使用します。次の例は、Greengrass デーモンを停止させる単純なスクリプトです。

  ```
  #!/bin/bash
  set -euo pipefail
  systemctl stop greengrass
  ```

- 更新が完了すると、エージェントは `ggc_post_update.sh` スクリプトを実行します。このスクリプトは、OTA 更新エージェントが AWS IoT Greengrass Core ソフトウェアの更新を開始した後に実行する必要があるコマンドに使用します。たとえば、プロセスを再起動する場合などです。次の例は、Greengrass デーモンを起動する単純なスクリプトです。

```
#!/bin/bash
set -euo pipefail
systemctl start greengrass
```

OTA 更新エージェントの OTA 更新の場合

- 更新を開始する前に、エージェントは `ota_pre_update.sh` スクリプトを実行します。OTA 更新エージェントが自身を更新する前に必要なコマンド (データのバックアップや実行中のプロセスの停止など) を実行するために、このスクリプトを使用します。
- 更新が完了すると、エージェントは `ota_post_update.sh` スクリプトを実行します。OTA 更新エージェントが自身の更新を開始した後に必要なコマンド (プロセスの再起動など) を実行するために、このスクリプトを使用します。

> **Note**
>
> `managedRespawn` が `false` に設定されている場合、OTA 更新エージェントはスクリプトを実行しません。

OTA 更新の作成

1 つ以上のコアで AWS IoT Greengrass ソフトウェアの OTA 更新を実行するには、次の手順に従います。

1. コアが OTA 更新の要件を満たしていることを確認します。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアまたは OTA 更新エージェントを管理するように init システムを設定した場合は、コアで以下を確認します。
>
> - config.json ファイルは `"managedRespawn" : true` を指定します。

- /*greengrass-root*/usr/scripts ディレクトリには、次のスクリプトが含まれています。
  - ggc_pre_update.sh
  - ggc_post_update.sh
  - ota_pre_update.sh
  - ota_post_update.sh

詳細については、「the section called “init システムとの統合”」を参照してください。

2. Core デバイスターミナルで、OTA 更新エージェントを起動します。

```
cd /greengrass-root/ota/ota_agent
sudo ./ggc-ota
```

> **Note**
>
> *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

競合の原因となる可能性があるため、コアで OTA 更新エージェントの複数のインスタンスを起動しないでください。

3. AWS IoT Greengrass API を使用してソフトウェア更新ジョブを作成します。

   a. CreateSoftwareUpdateJob API を呼び出します。この例では、 AWS CLI コマンドを使用します。

   次のコマンドは、1 つのコアで AWS IoT Greengrass Core ソフトウェアを更新するジョブを作成します。サンプル値を置き換えて、コマンドを実行します。

   Linux or macOS terminal

```
aws greengrass create-software-update-job \
--update-targets-architecture x86_64 \
--update-targets [\"arn:aws:iot:region:123456789012:thing/myCoreDevice\"] \
--update-targets-operating-system ubuntu \
--software-to-update core \
```

```
--s3-url-signer-role arn:aws:iam::123456789012:role/myS3UrlSignerRole \
--update-agent-log-level WARN \
--amzn-client-token myClientToken1
```

Windows command prompt

```
aws greengrass create-software-update-job ^
--update-targets-architecture x86_64 ^
--update-targets [\"arn:aws:iot:region:123456789012:thing/myCoreDevice\"] ^
--update-targets-operating-system ubuntu ^
--software-to-update core ^
--s3-url-signer-role arn:aws:iam::123456789012:role/myS3UrlSignerRole ^
--update-agent-log-level WARN ^
--amzn-client-token myClientToken1
```

このコマンドは、次のレスポンスを返します。

```
{
    "IotJobId": "GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-a1da0EXAMPLE",
    "IotJobArn": "arn:aws:iot:region:123456789012:job/
GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-a1da0EXAMPLE",
    "PlatformSoftwareVersion": "1.10.1"
}
```

b. レスポンスから `IoTJobId` をコピーします。

c. AWS IoT Core API で DescribeJob を呼び出して、ジョブのステータスを確認します。サンプル値をジョブ ID に置き換えて、コマンドを実行します。

```
aws iot describe-job --job-id GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-
a1da0EXAMPLE
```

コマンドは、ジョブに関する情報 (`status` や `jobProcessDetails` など) を含むレスポンスオブジェクトを返します。

```
{
    "job": {
        "jobArn": "arn:aws:iot:region:123456789012:job/
GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-a1da0EXAMPLE",
        "jobId": "GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-a1da0EXAMPLE",
        "targetSelection": "SNAPSHOT",
```

```
        "status": "IN_PROGRESS",
        "targets": [
            "arn:aws:iot:region:123456789012:thing/myCoreDevice"
        ],
        "description": "This job was created by Greengrass to update the
 Greengrass Cores in the targets with version 1.10.1 of the core software
 running on x86_64 architecture.",
        "presignedUrlConfig": {
            "roleArn": "arn:aws::iam::123456789012:role/myS3UrlSignerRole",
            "expiresInSec": 3600
        },
        "jobExecutionsRolloutConfig": {},
        "createdAt": 1588718249.079,
        "lastUpdatedAt": 1588718253.419,
        "jobProcessDetails": {
            "numberOfCanceledThings": 0,
            "numberOfSucceededThings": 0,
            "numberOfFailedThings": 0,
            "numberOfRejectedThings": 0,
            "numberOfQueuedThings": 1,
            "numberOfInProgressThings": 0,
            "numberOfRemovedThings": 0,
            "numberOfTimedOutThings": 0
        },
        "timeoutConfig": {}
    }
}
```

トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

CreateSoftwareUpdateJob API

`CreateSoftwareUpdateJob` API を使用して、コアデバイスの AWS IoT Greengrass Core ソフトウェアまたは OTA 更新エージェントソフトウェアを更新できます。この API は、更新が利用可能になったときにデバイスに通知する AWS IoT スナップショットジョブを作成します。を呼び出すと`CreateSoftwareUpdateJob`、他の AWS IoT ジョブコマンドを使用してソフトウェアの更新を追跡できます。詳細については、AWS IoT デベロッパーガイドのジョブを参照してください。

次の例は、 を使用して AWS CLI 、コアデバイスで Core AWS IoT Greengrass ソフトウェアを更新するジョブを作成する方法を示しています。

```
aws greengrass create-software-update-job \
--update-targets-architecture x86_64 \
--update-targets [\"arn:aws:iot:region:123456789012:thing/myCoreDevice\"] \
--update-targets-operating-system ubuntu \
--software-to-update core \
--s3-url-signer-role arn:aws:iam::123456789012:role/myS3UrlSignerRole \
--update-agent-log-level WARN \
--amzn-client-token myClientToken1
```

`create-software-update-job` コマンドは、更新によってインストールされたジョブ ID、ジョブ ARN、およびソフトウェアバージョンを含む JSON レスポンスを返します。

```
{
    "IotJobId": "GreengrassUpdateJob_c3bd7f36-ee80-4d42-8321-a1da0EXAMPLE",
    "IotJobArn": "arn:aws:iot:region:123456789012:job/GreengrassUpdateJob_c3bd7f36-
ee80-4d42-8321-a1da0EXAMPLE",
    "PlatformSoftwareVersion": "1.9.2"
}
```

`create-software-update-job` を使用してコアデバイスを更新する方法を示す手順については、「the section called “OTA 更新の作成”」を参照してください。

`create-software-update-job` コマンドでは、以下のパラメータを使用します。

`--update-targets-architecture`

Core デバイスのアーキテクチャ。

有効な値: `armv7l`、`armv6l`、`x86_64`、または `aarch64`

`--update-targets`

更新するコア。リストには、個々のコアの ARN と、メンバーがコアであるモノのグループの ARN を含めることができます。モノのグループの詳細については、「AWS IoT デベロッパーガイド」の「モノの静的グループ」を参照してください。

`--update-targets-operating-system`

Core デバイスのオペレーティングシステム。

有効な値: `ubuntu`、`amazon_linux`、`raspbian`、または `openwrt`

`--software-to-update`

更新するのが Core ソフトウェアであるか OTA 更新エージェントソフトウェアであるかを指定します。

有効な値: `core` または `ota_agent`

`--s3-url-signer-role`

AWS IoT Greengrass ソフトウェア更新アーティファクトにリンクする Amazon S3 URL の事前署名に使用される IAM ロールの ARN。ロールのアタッチされたアクセス許可ポリシーは、ターゲット AWS リージョン内のバケットに対する `s3:GetObject` アクションを許可する必要があります。ロールは、ロールを信頼されたエンティティとして引き受けることを `iot.amazonaws.com` に許可する必要があります。詳細については、「the section called “OTA 更新の IAM アクセス許可”」を参照してください。

`--amzn-client-token`

(オプション) べき等リクエストを行うために使用されるクライアントトークン。内部再試行のため重複した更新が作成されないようにする一意のトークンを指定します。

`--update-agent-log-level`

(オプション) OTA 更新エージェントによって生成されるログステートメントのログ記録レベル。デフォルト: `ERROR`。

有効な値: `NONE`、`TRACE`、`DEBUG`、`VERBOSE`、`INFO`、`WARN`、`ERROR`、または `FATAL`

> **Note**
>
> `CreateSoftwareUpdateJob` は、次のサポートされているアーキテクチャとオペレーティングシステムの組み合わせに対するリクエストのみを受け入れます。
>
> - ubuntu/x86_64
> - ubuntu/aarch64
> - amazon_linux/x86_64
> - raspbian/armv7l
> - raspbian/armv6l
> - openwrt/aarch64

- openwrt/armv7l

AWS IoT Greengrass コアに AWS IoT Greengrass グループをデプロイする

AWS IoT Greengrass グループを使用して、エッジ環境でエンティティを整理します。また、グループ内のエンティティ同士のやり取りや、 AWS クラウドとのやり取りを制御する用途にも使用します。例えば、グループ内の Lambda 関数のみをローカルでの実行用にデプロイし、グループ内のデバイスのみがローカル MQTT サーバーを使用して通信できます。

グループには、 Core ソフトウェアを実行する AWS IoT デバイスである AWS IoT Greengrass Core を含める必要があります。コアはエッジゲートウェイとして機能し、エッジ環境で AWS IoT Core 機能を提供します。ビジネスニーズに応じて、次のエンティティをグループに追加することもできます。

- クライアントデバイス。 AWS IoT レジストリでモノとして表されます。Greengrass デバイスは FreeRTOS を実行するか、AWS IoT Device SDK または AWS IoT Greengrass Discovery API を使用して、コアの接続情報を取得します。グループのメンバーであるクライアントデバイスのみが Core に接続できます。
- Lambda 関数。コアでコードを実行する、ユーザー定義のサーバーレスアプリケーションです。Lambda 関数は Greengrass グループで作成 AWS Lambda され、Greengrass グループから参照されます。詳細については、「ローカル Lambda 関数を実行する」を参照してください。
- コネクタ。コアでコードを実行する事前定義されたサーバーレスアプリケーションです。コネクタは、ローカルインフラストラクチャ、デバイスプロトコル AWS、およびその他のクラウドサービスとの統合を組み込みます。詳細については、「コネクタを使用してサービスおよびプロトコルと統合する」を参照してください。
- サブスクリプション。MQTT 通信が承認されている発行者、受信者、および MQTT トピック (またはサブジェクトなど) を定義します。
- リソース。ローカルのデバイスとボリューム、機械学習モデル、シークレットを参照し、Greengrass Lambda 関数およびコネクタによるアクセス制御に使用されます。
- ログ。 AWS IoT Greengrass システムコンポーネントと Lambda 関数の設定のログ記録。詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

で Greengrass グループを管理し AWS クラウド 、コアにデプロイします。デプロイは、グループ設定を Core デバイスの `group.json` ファイルにコピーします。このファイルは *`greengrass-root`*`/ggc/deployments/group` にあります。

> **Note**
>
> デプロイ中、コアデバイスの Greengrass デーモンプロセスは停止し、再起動します。

AWS IoT コンソールからのグループのデプロイ

AWS IoT コンソールのグループの設定ページから、グループをデプロイし、そのデプロイを管理できます。

> **Note**
>
> コンソールでこのページを開くには、[Greengrass]を選択してから、[Groups (V1)] (グループ〔V1〕) を選択し、[Greengrass groups] (Greengrass グループ) からグループを選択します。

グループの現在のバージョンをデプロイするには

- グループ設定ページで、[Deploy] (デプロイ) を選択します。

グループのデプロイ履歴を表示するには

グループのデプロイ履歴には、各デプロイの試行の日時、グループバージョン、ステータスが含まれます。

1. グループ設定ページで、[Deployment] (デプロイ) タブを選択します。
2. エラーメッセージなど、デプロイに関する詳細を表示するには、Greengrass デバイスの AWS IoT コンソールからデプロイを選択します。

グループのデプロイを再デプロイするには

現在のデプロイが失敗した場合、または別のグループバージョンに戻した場合は、デプロイを再デプロイできます。

1. AWS IoT コンソールから Greengrass デバイスを選択し、グループ (V1) を選択します。
2. [Deployment] (デプロイ) タブを選択します。
3. 再デプロイするデプロイを選択し、[Redeploy] (再デプロイ) を選択します。

グループのデプロイをリセットするには

グループデプロイをリセットして、グループを移動または削除したり、デプロイ情報を削除したりできます。詳細については、「the section called “デプロイのリセット”」を参照してください。

1. AWS IoT コンソールから Greengrass デバイスを選択し、グループ (V1) を選択します。
2. [Deployment] (デプロイ) タブを選択します。
3. リセットするデプロイを選択し、[Reset deployments] (デプロイのリセット) を選択します。

AWS IoT Greengrass API を使用したグループのデプロイ

AWS IoT Greengrass API には、 AWS IoT Greengrass グループをデプロイし、グループデプロイを管理するための以下のアクションが用意されています。これらのアクションは、、 AWS IoT Greengrass API AWS CLI、または AWS SDK から呼び出すことができます。

アクション	説明
CreateDeployment	`NewDeployment` または `Redeployment` デプロイを作成します。 現在のデプロイが失敗した場合は、デプロイを再デプロイできます。または、再デプロイして別のグループバージョンに戻すこともできます。
GetDeploymentStatus	デプロイのステータス、`Building`、`InProgress` 、`Success`、または `Failure` を返します。 デプロイ通知を受信するように Amazon EventBridge イベントを設定できます。詳細については、「the section called “デプロイ通知の取得”」を参照してください。
ListDeployments	グループのデプロイ履歴を返します。
ResetDeployments	グループのデプロイをリセットします。 グループデプロイをリセットして、グループを移動または削除したり、デプロイ情報を削除したりできます。詳細については、「the section called “デプロイのリセット”」を参照してください。

> **Note**
> 一括デプロイオペレーションの詳細については、「the section called “一括デプロイの作成”」を参照してください。

グループ ID の取得

グループ ID は、API アクションで共通して使用されます。 ListGroups アクションを使用して、グループのリストからターゲットグループの ID を検索できます。たとえば、 で AWS CLIコマンドを使用しますlist-groups。

```
aws greengrass list-groups
```

結果をフィルタリングする query オプションを含めることもできます。例:

- 最後に作成されたグループを取得するには、次の操作を行います。

  ```
  aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))[0]"
  ```

- 名前によりグループを取得するには:

  ```
  aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
  ```

  グループ名は一意である必要はないため、複数のグループが返されることがあります。

以下に、list-groups 応答の例を示します。各グループの情報には、グループの ID (Id プロパティ) と最新のグループバージョンの ID (LatestVersion プロパティ) が含まれます。グループの他のバージョン ID を取得するには、ListGroupVersions とともにグループ ID を使用します。

> **Note**
>
> これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

```
{
    "Groups": [
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/00dedaaa-ac16-484d-ad77-c3eedEXAMPLE/versions/4cbc3f07-fc5e-48c4-
a50e-7d356EXAMPLE",
            "Name": "MyFirstGroup",
```

```
            "LastUpdatedTimestamp": "2019-11-11T05:47:31.435Z",
            "LatestVersion": "4cbc3f07-fc5e-48c4-a50e-7d356EXAMPLE",
            "CreationTimestamp": "2019-11-11T05:47:31.435Z",
            "Id": "00dedaaa-ac16-484d-ad77-c3eedEXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/groups/00dedaaa-
ac16-484d-ad77-c3eedEXAMPLE"
        },
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE/versions/8fe9e8ec-64d1-4647-
b0b0-01dc8EXAMPLE",
            "Name": "GreenhouseSensors",
            "LastUpdatedTimestamp": "2020-01-07T19:58:36.774Z",
            "LatestVersion": "8fe9e8ec-64d1-4647-b0b0-01dc8EXAMPLE",
            "CreationTimestamp": "2020-01-07T19:58:36.774Z",
            "Id": "036ceaf9-9319-4716-ba2a-237f9EXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE"
        },
        ...
    ]
}
```

を指定しない場合 AWS リージョン、 AWS CLI コマンドはプロファイルのデフォルトリージョンを使用します。別のリージョンのグループを返すには、#####オプションを含めます。以下に例を示します。

```
aws greengrass list-groups --region us-east-1
```

AWS IoT Greengrass グループオブジェクトモデルの概要

AWS IoT Greengrass API を使用してプログラミングする場合、Greengrass グループオブジェクトモデルを理解すると便利です。

グループ

AWS IoT Greengrass API では、最上位`Group`オブジェクトはメタデータと`GroupVersion`オブジェクトのリストで構成されます。 `GroupVersion` オブジェクトは ID `Group`によって に関連付けられます。

グループバージョン

GroupVersion オブジェクトは、グループメンバーシップを定義します。各 GroupVersion は、ARN によって CoreDefinitionVersion およびその他のコンポーネントバージョンを参照します。これらの参照は、グループに含めるエンティティを決定します。

例えば、グループに 3 つの Lambda 関数、1 つのデバイス、2 つのサブスクリプションを含めるには、GroupVersion で次を参照します。

- 必要なコアを含む CoreDefinitionVersion。
- 3 つの関数を含む FunctionDefinitionVersion。
- クライアントデバイスを含む DeviceDefinitionVersion デバイス。

- 2 つのサブスクリプションを含む SubscriptionDefinitionVersion。

コアデバイスにデプロイされた GroupVersion によって、ローカル環境で使用できるエンティティと、それらのエンティティがどのようにやり取りできるかが決まります。

グループコンポーネント

グループに追加するコンポーネントには、3 つのレベルの階層があります。

- 定義は特定のタイプの DefinitionVersion オブジェクトのリストを参照します。例えば、DeviceDefinition は DeviceDefinitionVersion オブジェクトのリストを参照します。
- 特定のタイプのエンティティのセットを含む DefinitionVersion。例えば、DeviceDefinitionVersion には、Device オブジェクトのリストが含まれています。
- プロパティと動作を定義する個々のエンティティ。たとえば、 は、 AWS IoT レジストリ内の対応するクライアントデバイスの ARN、デバイス証明書の ARN、およびローカルシャドウがクラウドと自動的に同期するかどうかDeviceを定義します。

  グループには、次のタイプのエンティティを追加できます。

  - コネクタ
  - コア
  - デバイス
  - 関数
  - ロガー
  - リソース
  - サブスクリプション

次の例では、DeviceDefinition は、それぞれに複数の Device オブジェクトを含む 3 つの DeviceDefinitionVersion オブジェクトを参照します。グループでは、一度に 1 つの DeviceDefinitionVersion のみが使用されます。

グループの更新

AWS IoT Greengrass API では、バージョンを使用してグループの設定を更新します。バージョンは変更不可であるため、グループコンポーネントを追加、削除、または変更するには、新規または更新されたエンティティを含む DefinitionVersion オブジェクトを作成する必要があります。

新しい DefinitionVersions オブジェクトを新規または既存の定義オブジェクトに関連付けることができます。例えば、 `CreateFunctionDefinition` アクションを使用して、`FunctionDefinitionVersion` を初期バージョンとして含む `FunctionDefinition` を作成したり、 `CreateFunctionDefinitionVersion` アクションを使用して既存の `FunctionDefinition` を参照したりできます。

グループコンポーネントを作成したら、そのグループのメンバーにする、すべての DefinitionVersion オブジェクトを含む `GroupVersion` を作成します。次に、`GroupVersion` をデプロイします。

`GroupVersion` をデプロイするには、1 つだけ `Core` を含む `CoreDefinitionVersion` を参照する必要があります。参照されるエンティティはすべて、グループのメンバーである必要があります。また、Greengrass サービスロールは、 をデプロイ AWS リージョン する AWS アカウント の に関連付ける必要があります`GroupVersion`。

> **Note**
>
> API の `Update` アクションは、 `Group` またはコンポーネント定義オブジェクトの名前を変更するために使用されます。

AWS リソースを参照するエンティティの更新

Greengrass Lambda 関数とシークレットリソースは、Greengrass 固有のプロパティを定義し、対応する AWS リソースも参照します。これらのエンティティを更新するには、Greengrass オブジェクトではなく、対応する AWS リソースに変更を加えることができます。たとえば、Lambda 関数は で関数を参照 AWS Lambda し、Greengrass グループに固有のライフサイクルやその他のプロパティも定義します。

- Lambda 関数コードまたはパッケージ化された依存関係を更新するには、 AWS Lambdaで変更を加えます。次のグループのデプロイ時に、これらの変更は から取得 AWS Lambda され、ローカル環境にコピーされます。
- Greengrass 固有のプロパティを更新するには、更新された `Function` プロパティを含む `FunctionDefinitionVersion` を作成します。

> **Note**
>
> Greengrass Lambda 関数は、エイリアス ARN またはバージョン ARN で Lambda 関数を参照できます。エイリアス ARN を参照する場合 (推奨)、 AWS Lambdaで新しい関数バージョンを作成するときに `FunctionDefinitionVersion` (または `SubscriptionDefinitionVersion`) を更新する必要はありません。詳細については、「the section called “エイリアスまたはバージョンによる関数のリファレンス”」を参照してください。

関連情報

- the section called “デプロイ通知の取得”
- the section called “デプロイのリセット”
- the section called “一括デプロイの作成”
- デプロイの問題のトラブルシューティング

- AWS IoT Greengrass Version 1 API リファレンス
- コマンドリファレンスAWS IoT Greengrassの コマンド AWS CLI

デプロイ通知の取得

Amazon EventBridge イベントルールを使用して、Greengrass グループのデプロイの状態変更に関する通知を受け取ることができます。EventBridge は、 AWS リソースの変更を説明するシステムイベントのほぼリアルタイムのストリームを配信します。 は、これらのイベントを少なくとも 1 回 EventBridge AWS IoT Greengrass に送信します。つまり、 は、配信を確保するために、特定のイベントの複数のコピーを送信する AWS IoT Greengrass 可能性があります。さらに、イベントリスナーは、イベントが発生した順序でイベントを受信しない場合があります。

> **Note**
>
> Amazon EventBridge は、アプリケーションを Greengrass Core デバイスやデプロイ通知などのさまざまなソースのデータに接続するために使用できるイベントバスサービスです。詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge とは」を参照してください。

AWS IoT Greengrass グループデプロイの状態が変わると、 はイベントを発行します。すべての状態遷移または指定した状態への移行に対して実行される EventBridge ルールを作成できます。デプロイがルールを開始する状態になると、EventBridge はルールで定義されたターゲットアクションを呼び出します。これにより、通知を送信したり、イベント情報をキャプチャしたり、修正アクションを実行したり、状態の変更に応じて他のイベントを開始したりできます。例えば、次のユースケースのルールを作成できます。

- アセットのダウンロードや担当者の通知など、デプロイ後のオペレーションを開始します。
- デプロイの成功または失敗時に通知を送信します。
- デプロイイベントに関するカスタムメトリクスを発行します。

AWS IoT Greengrass は、デプロイが `Building`、`InProgress`、、`Success`および の状態になるとイベントを発行します`Failure`。

> **Note**
>
> 一括デプロイ オペレーションのステータスのモニタリングは、現在サポートされていません。ただし、 は、一括デプロイの一部である個々のグループデプロイの状態変更イベント AWS IoT Greengrass を発行します。

グループデプロイステータスの変更イベント

デプロイ状態変更のイベントでは、次の形式を使用します。

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass Deployment Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "time":"2018-03-22T00:38:11Z",
    "region":"us-west-2",
    "resources":[],
    "detail":{
        "group-id": "284dcd4e-24bc-4c8c-a770-EXAMPLEf03b8",
        "deployment-id": "4f38f1a7-3dd0-42a1-af48-EXAMPLE09681",
        "deployment-type": "NewDeployment|Redeployment|ResetDeployment|
ForceResetDeployment",
        "status": "Building|InProgress|Success|Failure"
    }
}
```

1 つ以上のグループに適用するルールを作成できます。次の 1 つ以上のデプロイタイプとデプロイの状態でルールをフィルタリングできます。

デプロイタイプ

- `NewDeployment`。グループバージョンの最初のデプロイ。
- `ReDeployment`。グループバージョンの再デプロイ。
- `ResetDeployment`。 AWS クラウド および AWS IoT Greengrass コアに保存されているデプロイ情報を削除します。詳細については、「the section called “デプロイのリセット”」を参照してください。

- `ForceResetDeployment`。 AWS クラウド に保存されているデプロイ情報を削除し、コアが応答するのを待たずに成功を報告します。また、コアが接続されている場合、または次に接続するときに、コアに保存されているデプロイ情報も削除します。

デプロイの状態

- `Building`. AWS IoT Greengrass は、グループ設定を検証し、デプロイアーティファクトを構築しています。
- `InProgress`。 デプロイは AWS IoT Greengrass コアで進行中です。
- `Success`。デプロイに成功しました。
- `Failure`。デプロイに失敗しました。

イベントが重複したり、順序が順不同である可能性があります。イベントの順序を特定するには、`time` プロパティを使用します。

> **Note**
>
> AWS IoT Greengrass は `resources`プロパティを使用しないため、常に空です。

EventBridge ルールを作成するための前提条件

の EventBridge ルールを作成する前に AWS IoT Greengrass、次の操作を行います。

- EventBridge のイベント、ルール、ターゲットに精通しておいてください。
- EventBridge ルールによって呼び出されるターゲットを作成して設定します。ルールは、以下のようなさまざまなタイプのターゲットを呼び出すことができます。
  - Amazon Simple Notification Service (Amazon SNS)
  - AWS Lambda 関数
  - Amazon Kinesis Video Streams
  - Amazon Simple Queue Service Amazon SQS キュー

詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge とは」および「Amazon EventBridge の開始方法」を参照してください。

デプロイ通知の設定 (コンソール)

グループのデプロイ状態が変更されたときに Amazon SNS トピックを発行する EventBridge ルールを作成するには、次のステップを使用します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。詳細については、「Amazon EventBridge ユーザーガイド」の「 AWS リソースからイベントでトリガーする EventBridge ルールの作成」を参照してください。 EventBridge

1. Amazon EventBridge コンソールを開きます。
2. ナビゲーションペインで [ルール] を選択します。
3. [ルールの作成] を選択します。
4. ルールの名前と説明を入力します。

   ルールには同じリージョン内および同じイベントバス上の別のルールと同じ名前を付けることはできません。
5. [イベントバス] で、このルールに関連付けるイベントバスを選択します。このルールをアカウントからのイベントと一致させるには、AWS のデフォルトのイベントバスを選択します。アカウントの AWS サービスがイベントを発行すると、常にアカウントのデフォルトのイベントバスに送られます。
6. [ルールタイプ] で、[イベントパターンを持つルール] を選択してください。
7. [Next (次へ)] を選択します。
8. [イベントソース] で、[AWS のサービス] を選択します。
9. [イベントパターン] で、[AWS のサービス] を選択します。
10. [AWS のサービス] で、[Greengrass] を選択します。
11. [イベントタイプ] で、[Greengrass デプロイステータスの変更] を選択します。

> **Note**
>
> AWS CloudTrail イベントタイプを介した API コールは、 と AWS IoT Greengrass の統合に基づいています AWS CloudTrail。このオプションを使用して、 AWS IoT Greengrass API への読み取りまたは書き込み呼び出しによって開始されるルールを作成できます。詳細については、「the section called “を使用した AWS IoT Greengrass API コールのログ記録 AWS CloudTrail”」を参照してください。

12. 通知を開始するデプロイ状態を選択します。

- すべての状態変更イベントの通知を受信するには、[Any state (すべての状態)] を選択します。
- 一部の状態変更イベントの通知のみを受信するには、[特定の状態] を選択してから、対象とする状態を選択します。

13. 通知を開始するデプロイタイプを選択します。

    - すべてのデプロイタイプの通知を受信するには、[すべての状態] を選択します。
    - 一部のデプロイタイプの通知のみを受信するには、[特定の状態] を選択してから、対象とするデプロイタイプを選択します。

14. [Next (次へ)] を選択します。
15. [ターゲットタイプ] で、[AWS のサービス] を選択します。
16. [ターゲットの選択] で、ターゲットを設定します。この例では Amazon SNS トピックを使用していますが、通知を送信するターゲットタイプには他のトピックも設定できます。

    a. [Target (ターゲット)] で [SNS topic (SNS トピック)] を選択します。
    b. [トピック] で、ターゲットトピックを選択します。
    c. [Next (次へ)] を選択します。

17. [Tags] (タグ) で、ルールのタグを定義するか、フィールドを空のままにします。
18. [次へ] を選択します。
19. ルールの詳細を確認し、ルールの作成 を選択します。

デプロイ通知の設定 (CLI)

グループのデプロイ状態が変更されたときに Amazon SNS トピックを発行する EventBridge ルールを作成するには、次のステップを使用します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。

1. ルールを作成します。

    - *group-id* を AWS IoT Greengrass グループの ID に置き換えます。

```
aws events put-rule \
  --name TestRule \
  --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"group-id\":
 [\"group-id\"]}}"
```

パターンで省略されたプロパティは無視されます。

2. トピックをルールターゲットとして追加します。

   - *topic-arn* を Amazon SNS トピックの ARN に置き換えます。

```
aws events put-targets \
  --rule TestRule \
  --targets "Id"="1","Arn"="topic-arn"
```

> **Note**
>
> Amazon EventBridge にターゲットトピックの呼び出しを許可するには、トピックにリソースベースのポリシーを追加する必要があります。詳細については、Amazon EventBridge ユーザーガイドの「Amazon SNS のアクセス許可」を参照してください。

詳細については、「Amazon EventBridge ユーザーガイド」の「EventBridge のイベントとイベントパターン」を参照してください。

デプロイ通知の設定 (AWS CloudFormation)

AWS CloudFormation テンプレートを使用して、Greengrass グループデプロイの状態変更に関する通知を送信する EventBridge ルールを作成します。詳細については、「AWS CloudFormation ユーザーガイド」の「Amazon EventBridge リソースタイプのリファレンス」を参照してください。

関連情報

- AWS IoT Greengrass グループのデプロイ
- 「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge とは」

デプロイのリセット

この機能は AWS IoT Greengrass Core v1.1 以降で使用できます。

グループのデプロイを次のようにリセットできます。

- グループのコアを別のグループに移動する場合や、グループのコアのイメージが再作成されている場合などに、グループを削除します。別の Greengrass グループでコアを使用するには、グループを削除する前に、グループのデプロイをリセットする必要があります。
- グループのコアを別のグループに移動する。
- すべてのデプロイ前の状態にグループを戻す。
- コアデバイスからデプロイの設定を削除する。
- コアデバイスあるいはクラウドから機密データを削除する。
- 現在のグループのコアを別のものに置き換えることなく、新規グループの設定をコアにデプロイする。

> **Note**
>
> デプロイのリセット機能は、 AWS IoT Greengrass Core ソフトウェア v1.0.0 では使用できません。v1.0.0 を使用してデプロイされたグループを削除することはできません。

デプロイのリセットオペレーションは、まず、特定のグループのクラウドに保存されているすべてのデプロイ情報をクリーンアップします。そして、グループのコアデバイスにデプロイ関連のすべての情報を消去するよう指示します (ユーザー定義の `config.json` や Greengrass コア証明書を除く、Lambda 関数、ユーザーログ、シャドウデータベース、サーバー証明書)。グループに現在 `In Progress` あるいは `Building` 状態のデプロイがある場合には、このグループのデプロイのリセットを開始できません。

AWS IoT コンソールからデプロイをリセットする

AWS IoT コンソールのグループ設定ページからグループデプロイをリセットできます。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. [Deployments] (デプロイ) タブから、[Reset deployments] (デプロイのリセット) を選択します。
4. [Reset deployments for this Greengrass Group] (この Greengrass グループのデプロイをリセットする) ダイアログボックスで、**`confirm`** と入力して同意し、[Reset deployment] (デプロイのリセット) を選択します。

AWS IoT Greengrass API を使用してデプロイをリセットする

、 AWS IoT Greengrass API AWS CLI、または AWS SDK の `ResetDeployments`アクションを使用して、デプロイをリセットできます。このトピックの例では、CLI を使用しています。

```
aws greengrass reset-deployments --group-id GroupId [--force]
```

CLI コマンド **`reset-deployments`** の引数

`--group-id`

グループ ID。この値を取得するには、`list-groups` コマンドを使用します。

`--force`

オプション。グループのコアデバイスが紛失、盗難または破壊された場合には、このパラメータを使用します。このオプションでは、コアデバイスのレスポンスを待たずに、すべてのデプロイ情報がクラウドから消去された後でデプロイリセットプロセスから成功の報告を行います。ただし、コアデバイスがアクティブであるか、アクティブになった場合は、クリーンアップオペレーションも実行します。

CLI コマンド `reset-deployments` の出力は次のようになります。

```
{
    "DeploymentId": "4db95ef8-9309-4774-95a4-eea580b6ceef",
    "DeploymentArn": "arn:aws:greengrass:us-west-2:106511594199:/greengrass/groups/
b744ed45-a7df-4227-860a-8d4492caa412/deployments/4db95ef8-9309-4774-95a4-eea580b6ceef"
}
```

デプロイのリセット状態は、CLI コマンド `get-deployment-status` で確認できます。

```
aws greengrass get-deployment-status --deployment-id DeploymentId --group-id GroupId
```

CLI コマンド **`get-deployment-status`** の引数

`--deployment-id`

デプロイ ID。

`--group-id`

グループ ID。

CLI コマンド `get-deployment-status` の出力は次のようになります。

```
{
    "DeploymentStatus": "Success",
    "UpdatedAt": "2017-04-04T00:00:00.000Z"
}
```

デプロイのリセット準備中は、`DeploymentStatus` が `Building` に設定されます。リセットデプロイの準備ができたが、 AWS IoT Greengrass コアがリセットデプロイを取得していない場合、`DeploymentStatus`は ですInProgress。

リセットオペレーションが失敗すると、エラー情報がレスポンスで返されます。

関連情報

- AWS IoT Greengrass グループのデプロイ
- 「AWS IoT Greengrass Version 1 API リファレンス」の「ResetDeployments」
- 「AWS IoT Greengrass Version 1 API リファレンス」の「GetDeploymentStatus」

グループの一括デプロイを作成する

シンプルな API コールを使用して、多数の Greengrass グループを一度にデプロイできます。これらのデプロイは、上限が固定された適応レートでトリガーされます。

このチュートリアルでは、 を使用して AWS CLI で一括グループデプロイを作成およびモニタリングする方法について説明します AWS IoT Greengrass。このチュートリアルの一括デプロイ例には、複数のグループが含まれています。この例をお客様の実装に使用して、必要な数のグループを追加できます。

このチュートリアルのおおまかな手順は以下のとおりです。

1. 一括デプロイ入力ファイルを作成してアップロードする
2. 一括デプロイ用の IAM 実行ロールを作成および設定する
3. 実行ロールに S3 バケットへのアクセスを許可する
4. グループをデプロイする
5. デプロイをテストする

前提条件

このチュートリアルを完了するには、以下が必要です。

- 1 つ以上のデプロイ可能な Greengrass グループ。 AWS IoT Greengrass のグループと Core の作成の詳細については、「の開始方法 AWS IoT Greengrass」を参照してください。
- マシンに AWS CLI インストールおよび設定されている 。詳細については、「AWS CLI ユーザーガイド」を参照してください。
- AWS リージョン と同じ で作成された S3 バケット AWS IoT Greengrass。新しいバケットの作成の詳細については、「Amazon Simple Storage Service ユーザーガイド」の「Amazon S3 バケットの作成、設定、操作」を参照してください。

> **Note**
>
> 現在、SSE KMS 対応バケットはサポートされていません。

ステップ 1: 一括デプロイ入力ファイルを作成してアップロードする

このステップでは、デプロイ入力ファイルを作成し、Amazon S3 バケットにアップロードします。このファイルは、一括デプロイの各グループに関する情報を含む、シリアル化された行区切りの JSON ファイルです。 AWS IoT Greengrass は、一括グループのデプロイを初期化するときに、この情報を使用してユーザーに代わって各グループをデプロイします。

1. 以下のコマンドを実行して、デプロイするグループごとに `groupId` を取得します。デプロイする各グループが AWS IoT Greengrass によって識別されるように、`groupId` を一括デプロイ入力ファイルに入力します。

   > **Note**
   >
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

   ```
   aws greengrass list-groups
   ```

レスポンスには、 AWS IoT Greengrass アカウント内の各グループに関する情報が含まれます。

```
{
  "Groups": [
    {
      "Name": "string",
      "Id": "string",
      "Arn": "string",
      "LastUpdatedTimestamp": "string",
      "CreationTimestamp": "string",
      "LatestVersion": "string",
      "LatestVersionArn": "string"
    }
  ],
  "NextToken": "string"
}
```

以下のコマンドを実行して、デプロイするグループごとに `groupVersionId` を取得します。

```
list-group-versions --group-id groupId
```

レスポンスには、グループのすべてのバージョンに関する情報が含まれています。使用するグループバージョンの `Version` の値を記録します。

```
{
  "Versions": [
    {
      "Arn": "string",
      "Id": "string",
      "Version": "string",
      "CreationTimestamp": "string"
    }
  ],
```

```
  "NextToken": "string"
}
```

2. 使用しているコンピュータのターミナルまたはエディタで、以下の例の *MyBulkDeploymentInputFile* というファイルを作成します。このファイルには、一括デプロイに含める各 AWS IoT Greengrass グループに関する情報が含まれています。この例では複数のグループを定義していますが、このチュートリアルでは 1 つのグループを定義するだけでもかまいません。

> **Note**
>
> このファイルのサイズは 100 MB 未満であることが必要です。

```
{"GroupId":"groupId1", "GroupVersionId":"groupVersionId1",
 "DeploymentType":"NewDeployment"}
{"GroupId":"groupId2", "GroupVersionId":"groupVersionId2",
 "DeploymentType":"NewDeployment"}
{"GroupId":"groupId3", "GroupVersionId":"groupVersionId3",
 "DeploymentType":"NewDeployment"}
...
```

各レコード (行) には、グループオブジェクトが含まれています。各グループオブジェクトには、対応する `GroupId` と `GroupVersionId`、および `DeploymentType` が含まれています。現在、 はNewDeployment一括デプロイタイプのみ AWS IoT Greengrass をサポートしています。

このファイルを保存して閉じます。ファイルの場所を記録します。

3. ターミナルで以下のコマンドを使用して、入力ファイルを Amazon S3 バケットにアップロードします。ファイルパスを先ほどのファイルの場所と名前に置き換えます。詳細については、「バケットへのオブジェクトの追加」を参照してください。

```
aws s3 cp path/MyBulkDeploymentInputFile s3://amzn-s3-demo-bucket/
```

ステップ 2: IAM 実行ロールを作成して設定する

このステップでは、IAM コンソールを使用してスタンドアロンの実行ロールを作成します。次に、ロールと の間に信頼関係を確立 AWS IoT Greengrass し、IAM ユーザーに実行ロールのPassRole権限があることを確認します。これにより、 AWS IoT Greengrass は実行ロールを引き受け、ユーザーに代わってデプロイを作成できます。

1. 以下のポリシーを使用して、実行ロールを作成します。このポリシードキュメントでは、お客様に代わって各デプロイを作成するときに一括デプロイ入力ファイルにアクセスすることを AWS IoT Greengrass に許可します。

   IAM ロールの作成とアクセス許可の委任の詳細については、「IAM ロールの作成」を参照してください。

```
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Sid": "VisualEditor0",
           "Effect": "Allow",
           "Action": "greengrass:CreateDeployment",
           "Resource": [
               "arn:aws:greengrass:region:accountId:/greengrass/groups/groupId1",
               "arn:aws:greengrass:region:accountId:/greengrass/groups/groupId2",
               "arn:aws:greengrass:region:accountId:/greengrass/groups/groupId3",
               ...
           ]
       }
   ]
}
```

> **Note**
>
> このポリシーでは、 AWS IoT Greengrassによってデプロイされる一括デプロイ入力ファイル内の各グループまたはグループバージョン用にリソースが必要です。すべてのグループにアクセスを許可するには、Resource でアスタリスクを指定します。

```
"Resource": ["*"]
```

2. AWS IoT Greengrassを含めるように実行ロールの信頼関係を変更します。これにより、 AWS IoT Greengrass は実行ロールとそれにアタッチされたアクセス許可を使用できます。詳細については、「既存のロールの信頼関係の編集」を参照してください。

   また、aws:SourceArn と aws:SourceAccount のグローバル条件コンテキストキーを信頼ポリシーに加えることで、「混乱した代理」によるセキュリティ上の問題への対策にすることをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。「混乱した代理」問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        }
      }
    }
  ]
}
```

3. 実行ロールの IAM PassRole アクセス許可を IAM ユーザーに付与します。この IAM ユーザーは、一括デプロイを開始するために使用されます。PassRole アクセス許可により、IAM ユーザーは実行ロールを AWS IoT Greengrass に渡せるようになります。詳細については、AWS「 サービスにロールを渡すアクセス許可をユーザーに付与する」を参照してください。

次の例を参考にして、実行ロールにアタッチされた IAM ポリシーを更新します。必要に応じて、この例を変更します。

```
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Sid": "Stmt1508193814000",
           "Effect": "Allow",
           "Action": [
               "iam:PassRole"
           ],
           "Resource": [
               "arn:aws:iam::account-id:user/executionRoleArn"
           ]
           "Condition": {
               "StringEquals": {
                   "iam:PassedToService": "greengrass.amazonaws.com"
               }
           }
       }
   ]
}
```

ステップ 3: 実行ロールに S3 バケットへのアクセスを許可する

一括デプロイを開始するには、実行ロールが Amazon S3 バケットから一括デプロイ入力ファイルを読み取ることができる必要があります。以下のポリシー例を Amazon S3 バケットにアタッチして、そのバケットに対する `GetObject` アクセス許可が実行ロールに付与されるようにします。

詳細については、「S3 バケットポリシーを追加する方法」を参照してください。

```
{
    "Version": "2008-10-17",
    "Id": "examplePolicy",
    "Statement": [
        {
            "Sid": "Stmt1535408982966",
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "executionRoleArn"
                ]
            },
            "Action": "s3:GetObject",
            "Resource":
            "arn:aws:s3:::amzn-s3-demo-bucket/objectKey"
        }
    ]
}
```

ターミナルで以下のコマンドを使用して、バケットのポリシーを確認できます。

```
aws s3api get-bucket-policy --bucket amzn-s3-demo-bucket
```

> **Note**
>
> 代わりに、Amazon S3 バケットに対する `GetObject` アクセス許可が実行ロールに付与されるように、そのロールを直接変更することもできます。そのためには、以下のポリシー例を実行ロールにアタッチします。
>
> ```
> {
>  "Version": "2012-10-17",
>  "Statement": [
>  {
>  "Sid": "VisualEditor0",
>  "Effect": "Allow",
>  "Action": "s3:GetObject",
>  "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/objectKey"
>  }
>  ]
> }
> ```

ステップ 4: グループをデプロイする

このステップでは、一括デプロイ入力ファイルで設定したすべてのグループバージョンに対して一括デプロイオペレーションを開始します。各グループバージョンに対するデプロイアクションのタイプは `NewDeploymentType` です。

> **Note**
> 同じアカウントの別の一括デプロイがまだ実行されている間は、StartBulkDeployment を呼び出すことはできません。リクエストは却下されました。

1. 以下のコマンドを使用して、一括デプロイを開始します。

   各 StartBulkDeployment リクエストに `X-Amzn-Client-Token` トークンを含めることをお勧めします。これらのリクエストは、トークンおよびリクエストパラメータに対してべき等です。このトークンは、最大 64 バイトの ASCII 文字で、大文字と小文字を区別する一意の文字列です。

   ```
   aws greengrass start-bulk-deployment --cli-input-json "{
             "InputFileUri":"URI of file in S3 bucket",
             "ExecutionRoleArn":"ARN of execution role",
             "AmznClientToken":"your Amazon client token"
             }"
   ```

   このコマンドを実行した結果、成功ステータスコード `200` が以下のレスポンスと共に返されます。

   ```
   {
     "bulkDeploymentId": UUID
   }
   ```

   一括デプロイ ID を記録します。この ID は、一括デプロイのステータスの確認に使用します。

> **Note**
>
> 一括デプロイオペレーションは現在サポートされていませんが、Amazon EventBridge イベントルールを作成して、個々のグループのデプロイステータスの変更通知を受け取ることができます。詳細については、「the section called “デプロイ通知の取得”」を参照してください。

2. 以下のコマンドを使用して、一括デプロイのステータスを確認します。

```
aws greengrass get-bulk-deployment-status --bulk-deployment-id 1234567
```

このコマンドを実行した結果、成功ステータスコード 200 が情報の JSON ペイロードと共に返されます。

```
{
 "BulkDeploymentStatus": Running,
 "Statistics": {
    "RecordsProcessed": integer,
    "InvalidInputRecords": integer,
    "RetryAttempts": integer
 },
 "CreatedAt": "string",
 "ErrorMessage": "string",
 "ErrorDetails": [
   {
     "DetailedErrorCode": "string",
     "DetailedErrorMessage": "string"
   }
 ]
}
```

BulkDeploymentStatus には、一括実行の現在のステータスが含まれています。実行は 以下の 6 つの異なるステータスのいずれかになります。

- Initializing。一括デプロイリクエストが受け取られ、デプロイの実行を開始する準備中です。
- Running。一括デプロイの実行が開始されました。

- Completed。一括デプロイの実行により、すべてのレコードの処理が完了しました。
- Stopping。一括デプロイの実行は停止コマンドを受け取り、間もなく終了します。前のデプロイが Stopping 状態になっている間は、新しい一括デプロイを開始できません。
- Stopped。一括デプロイの実行は手動で停止されました。
- Failed。一括デプロイの実行中にエラーが発生し、デプロイの実行は終了しました。エラーの詳細は、ErrorDetails フィールドで確認できます。

JSON ペイロードには、一括デプロイの進行状況に関する統計情報も含まれています。この情報を使用して、処理されたグループの数と失敗したグループの数を判断できます。統計情報は以下のとおりです。

- RecordsProcessed: 試行されたグループレコードの数。
- InvalidInputRecords: 再試行不可のエラーを返したレコードの合計数。例えば、入力ファイルのグループレコードで無効な形式を使用している場合や、存在しないグループバージョンを指定している場合、デプロイの実行によりグループまたはグループバージョンをデプロイするアクセス許可が付与されない場合に、このエラーが発生します。
- RetryAttempts: 再試行可能なエラーを返したデプロイ試行の回数。例えば、グループをデプロイしようとしてスロットリングエラーが返された場合は、再試行がトリガーされます。グループデプロイは 5 回まで再試行できます。

一括デプロイの実行に失敗した場合、このペイロードには、トラブルシューティングに使用できる ErrorDetails セクションも含まれています。このセクションに、実行失敗の原因に関する情報が含まれています。

一括デプロイの状態を定期的にチェックして、デプロイが想定どおりに進行していることを確認できます。デプロイが完了したら、RecordsProcessed は、一括デプロイの入力ファイルで指定したデプロイグループの数と一致しています。これは、各レコードが処理されたことを示します。

ステップ 5: デプロイをテストする

ListBulkDeployments コマンドを使用して、一括デプロイの ID を見つけます。

```
aws greengrass list-bulk-deployments
```

このコマンドは、最も新しいものから古いものまですべての一括デプロイのリストを BulkDeploymentId も含めて返します。

```
{
  "BulkDeployments": [
    {
      "BulkDeploymentId": 1234567,
      "BulkDeploymentArn": "string",
      "CreatedAt": "string"
    }
  ],
  "NextToken": "string"
}
```

次に、ListBulkDeploymentDetailedReports コマンドを呼び出して、各デプロイに関する詳細情報を収集します。

```
aws greengrass list-bulk-deployment-detailed-reports --bulk-deployment-id 1234567
```

このコマンドを実行した結果、成功ステータスコード 200 が情報の JSON ペイロードと共に返されます。

```
{
  "BulkDeploymentResults": [
    {
      "DeploymentId": "string",
      "GroupVersionedArn": "string",
      "CreatedAt": "string",
      "DeploymentStatus": "string",
      "ErrorMessage": "string",
      "ErrorDetails": [
        {
          "DetailedErrorCode": "string",
          "DetailedErrorMessage": "string"
        }
```

```
      ]
    }
  ],
  "NextToken": "string"
}
```

このペイロードには通常、最も新しいものから古いものまで各デプロイとそのデプロイステータスのページ分割されたリストが含まれています。また、一括デプロイの実行に失敗した場合の詳細情報も含まれています。先ほど説明したように、リストに示されたデプロイの合計数は、一括デプロイの入力ファイルで指定したグループの数と一致しています。

返される情報は、デプロイが終了状態 (成功または失敗) になるまで変わります。それまでは、このコマンドを定期的に呼び出すことができます。

一括デプロイのトラブルシューティング

一括デプロイに成功しなかった場合は、以下のトラブルシューティング手順を試すことができます。ターミナルで以下のコマンドを実行します。

入力ファイルのエラーのトラブルシューティングを行う

一括デプロイ入力ファイルに構文エラーがある場合、一括デプロイは失敗する可能性があります。これにより、一括デプロイステータス Failed が、最初の検証エラーの行番号を示すエラーメッセージと共に返されます。以下の 4 つのエラーが考えられます。

- ```
  InvalidInputFile: Missing GroupId at line number: line number
  ```

  このエラーは、指定された入力ファイル行で、指定されたパラメータを登録できないことを示します。パラメータ GroupId と GroupVersionId が指定されていない可能性があります。

- ```
  InvalidInputFile: Invalid deployment type at line number : line number. Only valid
   type is 'NewDeployment'.
  ```

  このエラーは、指定された入力ファイル行で無効なデプロイタイプが指定されていることを示します。現時点でサポートされているデプロイタイプは NewDeployment のみです。

- ```
  Line %s is too long in S3 File. Valid line is less than 256 chars.
  ```
  ```

このエラーは、指定された入力ファイルの行が長すぎるため、短くする必要があることを示します。

- ```
  Failed to parse input file at line number: line number
  ```

  このエラーは、指定された入力ファイル行が有効な JSON とみなされないことを示します。

同時一括デプロイがないことを確認する

新しい一括デプロイは、別の一括デプロイが実行中など終了以外の状態では開始できません。この場合、`Concurrent Deployment Error` になります。ListBulkDeployments コマンドを使用して、一括デプロイが現在実行中ではないことを確認できます。このコマンドは、最も新しいものから古いものまで一括デプロイを一覧表示します。

```
{
  "BulkDeployments": [
    {
      "BulkDeploymentId": BulkDeploymentId,
      "BulkDeploymentArn": "string",
      "CreatedAt": "string"
    }
  ],
  "NextToken": "string"
}
```

GetBulkDeploymentStatus コマンドを実行するには、最初に一覧表示された一括デプロイの `BulkDeploymentId` を使用します。最も新しい一括デプロイが実行状態 (`Initializing` または `Running`) である場合は、以下のコマンドを使用して一括デプロイを停止します。

```
aws greengrass stop-bulk-deployment --bulk-deployment-id BulkDeploymentId
```

このアクションを実行した結果、デプロイのステータスは `Stopped` になるまで `Stopping` になります。デプロイのステータスが `Stopped` になったら、新しい一括デプロイを開始できます。
```

ErrorDetails を確認する

`GetBulkDeploymentStatus` コマンドを実行すると、一括デプロイの実行の失敗に関する情報を含む JSON ペイロードが返されます。

```
"Message": "string",
"ErrorDetails": [
  {
    "DetailedErrorCode": "string",
    "DetailedErrorMessage": "string"
  }
]
```

エラーで終了すると、この呼び出しによって返された `ErrorDetails` JSON ペイロードには、一括デプロイの実行の失敗に関する詳細が含まれています。例えば、`400` 番台のエラーステータスコードは、入力パラメータまたは呼び出し元依存関係の入力エラーを示します。

AWS IoT Greengrass コアログを確認する

AWS IoT Greengrass コアログを表示することで、問題をトラブルシューティングできます。`runtime.log` を表示するには、以下のコマンドを使用します。

```
cd /greengrass/ggc/var/log
sudo cat system/runtime.log | more
```

AWS IoT Greengrass ログ記録の詳細については、「」を参照してください[AWS IoT Greengrass ログによるモニタリング](#)。

関連情報

詳細については、以下のリソースを参照してください。

- [AWS IoT Greengrass グループのデプロイ](#)
- 「AWS CLI コマンドリファレンス」の [Amazon S3 API コマンド](#)
- コマンドリファレンス[AWS IoT Greengrass](#)の コマンド AWS CLI

AWS IoT Greengrass コアで Lambda 関数を実行する

AWS IoT Greengrass は、ユーザーが作成したユーザー定義コード用のコンテナ化された Lambda ランタイム環境を提供します AWS Lambda。 AWS IoT Greengrass コアにデプロイされた Lambda 関数は、コアのローカル Lambda ランタイムで実行されます。ローカル Lambda 関数は、ローカルイベント、クラウドからのメッセージやその他のリソースによってトリガーされ、クライアントデバイスにローカルコンピューティングの機能性をもたらします。例えば、データをクラウドに送信する前にデバイスデータをフィルタリングするために、Greengrass Lambda 関数を使用できます。

Lambda 関数をコアにデプロイするには、この関数を Greengrass グループに追加し (既存の Lambda 関数を参照して行います)、この関数に対してグループ固有の設定を行い、グループをデプロイします。関数が AWS サービスにアクセスする場合は、Greengrass グループロールに必要なアクセス許可も追加する必要があります。

Lambda 関数の実行方法 (アクセス許可、分離、メモリ制限など) を指定するパラメータを設定できます。詳細については、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。

> **Note**
>
> これらの設定により、Docker コンテナ AWS IoT Greengrass で を実行することもできます。詳細については、「the section called “Docker コンテナ AWS IoT Greengrass で を実行する”」を参照してください。

次の表にAWS Lambda 、サポートされているランタイムと、それらが実行できる AWS IoT Greengrass Core ソフトウェアのバージョンを示します。

言語あるいはプラットフォーム	GGC のバージョン
Python 3.8	1.11
Python 3.7	1.9 以降
Python 2.7 *	1.0 以降
Java 8	1.1 以降

言語あるいはプラットフォーム	GGC のバージョン
Node.js 12.x *	1.10 以降
Node.js 8.10 *	1.9 以降
Node.js 6.10 *	1.1 以降
C、C++	1.6 以降

* これらのランタイムを使用する Lambda 関数は、サポートされている のバージョンで実行できますが AWS IoT Greengrass、 で作成することはできません AWS Lambda。デバイスのランタイムがその関数のために指定された AWS Lambda ランタイムと異なる場合、`FunctionDefintionVersion` の `FunctionRuntimeOverride` を使用して独自のランタイムを選択できます。詳細については、「CreateFunctionDefinition」を参照してください。サポートされているランタイムの詳細については、「AWS Lambda デベロッパーガイド」 の「Runtime support policy」(ランタイムのサポートポリシー) を参照してください。

Greengrass Lambda 関数の SDK

AWS には、 AWS IoT Greengrass コアで実行されている Greengrass Lambda 関数で使用できる 3 SDKs が用意されています。これらの SDK は別々のパッケージに含まれているため、関数で同時に使用できます。Greengrass Lambda 関数で SDK を使用するには、 AWS Lambdaにアップロードする Lambda 関数デプロイパッケージに SDK を含めます。

AWS IoT Greengrass コア SDK

コアを操作するローカル Lambda 関数を有効にします。

- MQTT メッセージを と交換します AWS IoT Core。
- Greengrass グループのコネクタ、クライアントデバイス、その他の Lambda 関数で MQTT メッセージを交換します。
- ローカルシャドウサービスとやり取りを行います。
- その他のローカル Lambda 関数を呼び出します。
- シークレットリソースにアクセスします。
- ストリームマネージャーとやり取りを行います。

AWS IoT Greengrass は、GitHub で以下の言語とプラットフォームで AWS IoT Greengrass Core SDK を提供します。

- AWS IoT Greengrass Core SDK for Java
- AWS IoT Greengrass Node.js 用 Core SDK
- AWS IoT Greengrass Core SDK for Python
- AWS IoT Greengrass Core SDK for C

Lambda 関数デプロイパッケージに AWS IoT Greengrass Core SDK 依存関係を含めるには：

1. Lambda 関数のランタイムに一致する AWS IoT Greengrass Core SDK パッケージの言語またはプラットフォームをダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
3. 関数コードを含む Lambda 関数デプロイパッケージに `greengrasssdk` を含めます。これは、Lambda 関数の作成 AWS Lambda 時に にアップロードするパッケージです。

StreamManagerClient

ストリームマネージャーオペレーションに使用できる AWS IoT Greengrass Core SDKsは、次のとおりです。

- Java SDK (v1.4.0 以降)
- Python SDK (v1.5.0 以降)
- Node.js SDK (v1.6.0 以降)

AWS IoT Greengrass Core SDK for Python を使用してストリームマネージャーを操作するには、Python 3.7 以降をインストールする必要があります。また、Python Lambda 関数のデプロイパッケージに含める依存関係もインストールする必要があります。

1. `requirements.txt` ファイルのある SDK ディレクトリに移動します。このファイルには依存関係の一覧が記載されています。
2. SDK の依存関係をインストールします。例えば、次の `pip` コマンドを実行して、現在のディレクトリにインストールします。

```
pip install --target . -r requirements.txt
```

AWS IoT Greengrass Core SDK for Python をコアデバイスにインストールする

Python Lambda 関数を実行している場合は、 `pip`を使用して AWS IoT Greengrass Core SDK for Python をコアデバイスにインストールすることもできます。そうすれば、Lambda 関数デプロイパッケージに SDK を含めずに関数をデプロイできます。詳細については、「greengrasssdk」を参照してください。

このサポートは、サイズ制限のあるコアを対象としています。可能な場合は、Lambda 関数デプロイパッケージに SDK を含めることをお勧めします。

AWS IoT Greengrass Machine Learning SDK

ローカル Lambda 関数は Greengrass コアに機械学習 (ML) リソースとしてデプロイされる ML モデルを使用できます。Lambda 関数は、この SDK を使用してコネクタとしてコアにデプロイされているローカル推論サービスを呼び出し、このサービスと対話できます。Lambda 関数と ML コネクタも、この SDK を使用してデータを ML フィードバックコネクタへ送信し、アップロードおよび発行を行うことができます。SDK を使用するコード例などの詳細については、「the section called “ML イメージ分類”」、「the section called “ML オブジェクト検出”」、および「the section called “ML フィードバック”」を参照してください。

次の表に、 SDK バージョンと、それらが実行できる AWS IoT Greengrass Core ソフトウェアでサポートされている言語またはプラットフォームを示します。

SDK のバージョン	言語あるいはプラットフォーム	必要な GGC バージョン	Changelog
1.1.0	Python 3.7 または 2.7	1.9.3 以降	Python 3.7 のサポートと新しい `feedback` クライアントを追加しました。

SDK のバージョン	言語あるいはプラットフォーム	必要な GGC バージョン	Changelog
1.0.0	Python 2.7	1.7 以降	初回リリース。

ダウンロード情報については、「the section called “AWS IoT Greengrass ML SDK ソフトウェア”」を参照してください。

AWS SDK

ローカル Lambda 関数が Amazon S3、DynamoDB、 AWS IoTなどの AWS サービスに直接呼び出しできるようにします AWS IoT Greengrass。 AWS SDK を Greengrass Lambda 関数で使用するには、デプロイパッケージに含める必要があります。 AWS IoT Greengrass Core AWS SDK と同じパッケージで SDK を使用する場合は、Lambda 関数が正しい名前空間を使用していることを確認してください。Greengrass Lambda 関数は、このコアがオフラインの場合にクラウドサービスと通信できません。

入門リソースセンターから AWS SDKs をダウンロードします。

デプロイパッケージの作成の詳細については、入門チュートリアルの「the section called “Lambda 関数の作成とパッケージ化”」あるいは「AWS Lambda デベロッパーガイド」の「デプロイパッケージの作成」を参照してください。

クラウドベースの Lambda 関数への移行

AWS IoT Greengrass Core SDK は AWS SDK プログラミングモデルに従っており、クラウド用に開発された Lambda 関数を AWS IoT Greengrass コアで実行される Lambda 関数に簡単に移植できます。

たとえば、次の Python Lambda 関数は AWS SDK for Python (Boto3) を使用して、クラウド`some/topic`内のトピックにメッセージを発行します。

```
import boto3

iot_client = boto3.client("iot-data")
response = iot_client.publish(
    topic="some/topic", qos=0, payload="Some payload".encode()
```

```
)
```

AWS IoT Greengrass コアの 関数を移植するには、次の例`greengrasssdk`に示すように、`import`ステートメントと`client`初期化で`boto3`モジュール名を に変更します。

```
import greengrasssdk

iot_client = greengrasssdk.client("iot-data")
iot_client.publish(topic="some/topic", qos=0, payload="Some payload".encode())
```

> **Note**
>
> AWS IoT Greengrass Core SDK は、QoS = 0 の MQTT メッセージの送信のみをサポートします。詳細については、「the section called “サービスのメッセージの品質”」を参照してください。

プログラミングモデル間の類似性により、最小限の労力 AWS IoT Greengrass で Lambda 関数をクラウドで開発し、 に移行することもできます。Lambda 実行可能ファイルはクラウドでは実行されないため、デプロイ前に AWS SDK を使用してクラウドで開発することはできません。

エイリアスまたはバージョンによる Lambda 関数のリファレンス

Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。エイリアスは、グループデプロイ中にバージョン番号を解決します。エイリアスを使用すると、解決されたバージョンはデプロイ時にエイリアスが示すバージョンに更新されます。

AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。$LATEST バージョンは、イミュータブルで発行された関数バージョンにバインドされず、いつでも変更できます。これは、 AWS IoT Greengrass のバージョン普遍性の原則とは異なります。

Greengrass Lambda 関数がコード変更によって常に更新されるようにするための一般的な方法は、Greengrass グループとサブスクリプションで **PRODUCTION** という名前のエイリアスを使用することです。Lambda 関数の新しいバージョンを本稼働環境に移行すると、エイリアスが最新の安定バージョンを示し、グループを再デプロイします。また、このメソッドを使用して以前のバージョンにロールバックすることもできます。

グループ固有の設定による Greengrass Lambda 関数の実行の制御

AWS IoT Greengrass は、Greengrass Lambda 関数のクラウドベースの管理を提供します。Lambda 関数のコードと依存関係は を使用して管理されますが AWS Lambda、Lambda 関数が Greengrass グループで実行された場合の動作を設定できます。

グループ固有構成設定

AWS IoT Greengrass には、Greengrass Lambda 関数の以下のグループ固有の設定が用意されています。

[System user and group] (システムユーザーとグループ)

各 Lambda 関数を実行するために使用されるアクセス ID。デフォルトでは、Lambda 関数はグループのデフォルトのアクセス ID として実行されます。通常の場合、これはスタンダードの AWS IoT Greengrass システムアカウント (ggc_user および ggc_group) です。デフォルト設定を変更し、Lambda 関数を実行するために必要なアクセス許可を持つユーザー ID とグループ ID を選択できます。UID と GID の両方をオーバーライドするか、他方のフィールドを空のままにして一方だけをオーバーライドすることもできます。この設定により、デバイスリソースへのアクセスをより詳細に制御できます。Greengrass ハードウェアの設定では、適切なリソース制限、適切なファイルへのアクセス許可、および Lambda 関数を実行するために使用されるアクセス許可を持つユーザーとグループの適切なディスククォータを使用することをお勧めします。

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

> **Important**
>
> やむを得ない場合を除き、Lambda 関数を root として実行することは避けてください。root として実行すると、次のリスクが増加します。
>
> - 重要なファイルを誤って削除するなど、意図しない変更が行われるリスク。
> - 悪意のあるユーザーがデータやデバイスにアクセスするリスク。
> - Docker コンテナが `--net=host` や `UID=EUID=0` で実行されると、コンテナがエスケープするリスク。
>
> root として を実行する必要がある場合は、 AWS IoT Greengrass 設定を更新して有効にする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

[System user ID (number)] (システムユーザー ID (数値))

Lambda 関数を実行するために必要なアクセス許可を持つユーザーのユーザー ID。この設定を利用できるのは、[Another user ID/group ID] (別のユーザー ID/グループ ID) として実行した場合に限ります。 AWS IoT Greengrass コアデバイスで getent passwd コマンドを使用して、Lambda 関数の実行に使用するユーザー ID を検索できます。

同じ UID を使用してプロセスと Lambda 関数を Greengrass コアデバイスで実行する場合、Greengrass グループロールはプロセスに一時的な認証情報を付与することができます。プロセスは Greengrass コアデプロイメント全体で一時的な認証情報を使用できます。

[System group ID (number)] (システムグループ ID (番号))

Lambda 関数を実行するために必要なアクセス許可を持つグループのグループ ID。この設定を利用できるのは、[Another user ID/group ID] (別のユーザー ID/グループ ID) として実行した場合に限ります。 AWS IoT Greengrass コアデバイスで getent group コマンドを使用して、Lambda 関数の実行に使用するグループ ID を検索できます。

[Lambda function containerization] (Lambda 関数のコンテナ化)

Lambda 関数を実行する際に、グループのデフォルトのコンテナ化を使用するかどうかを選択します。または、この Lambda 関数で常に使用するコンテナ化を指定します。

Lambda 関数のコンテナ化モードは、その分離レベルを決定します。

- コンテナ化された Lambda 関数は、Greengrass コンテナモードで実行されます。Lambda 関数は、 AWS IoT Greengrass コンテナ内の分離されたランタイム環境 (または名前空間) で実行されます。
- コンテナ化されていない Lambda 関数は、コンテナなしモードで実行されます。この Lambda 関数は、分離することなく、通常の Linux プロセスとして実行されます。

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

コンテナ化を使用せずに Lambda 関数を実行する必要があるユースケースを除き、Greengrass コンテナで関数を実行することをお勧めします。Lambda 関数を Greengrass コンテナで実行する場合、アタッチされたローカルおよびデバイスリソースを使用することで、分離の利点とセキュリティの向上を得られます。コンテナ化を変更する場合は、事前に「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

> **Note**
> デバイスのカーネル名前空間および cgroup を有効化せずに実行するには、コンテナ化を使用しないですべての Lambda 関数を実行する必要があります。これを簡単に行うには、グループのデフォルトのコンテナ化を設定できます。詳細については、the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定” を参照してください。

メモリ制限

関数のメモリ割り当て。デフォルトでは 16 MB です。

> **Note**
> コンテナ化を使用しないで実行するように Lambda 関数を変更すると、メモリ制限の設定は使用できなくなります。コンテナ化を使用せずに実行される Lambda 関数には、メモリ制限を設定することができません。コンテナ化を使用しないで実行するように Lambda 関数またはグループのデフォルトのコンテナ化設定を変更すると、メモリ制限の設定は破棄されます。

タイムアウト

関数あるいはリクエストが終了するまでの時間数。デフォルト値は 3 秒です。

[Pinned] (固定)

Lambda 関数のライフサイクルは、オンデマンドあるいは長い存続期間とすることができます。デフォルトはオンデマンドです。

オンデマンド Lambda 関数は、呼び出されたときに新規または再利用されるコンテナで開始します。関数に対するリクエストは、任意の利用可能なコンテナで処理される可能性があります。存続期間の長い、または固定された Lambda 関数は、 AWS IoT Greengrass 起動後に自動的に起動し、独自のコンテナ (またはサンドボックス) で実行し続けます。関数へのすべてのリクエストは、同じコンテナで処理されます。詳細については、「the section called “ライフサイクル設定”」を参照してください。

/sys ディレクトリへの読み込みアクセス

関数がホストの /sys フォルダにアクセスできるかどうかを設定します。関数が /sys からデバイス情報を読み取る必要があるときに使用します。デフォルトは False です。

> **Note**
>
> この設定は、コンテナ化を使用しないで Lambda 関数を実行する場合は使用できません。コンテナ化を使用しないで実行するように Lambda 関数を変更すると、この設定の値は破棄されます。

[Encoding type] (エンコードタイプ)

関数の入力ペイロードの予期されるエンコードタイプ (JSON あるいは バイナリ)。デフォルトは JSON です。

バイナリエンコーディングタイプのサポートは、 AWS IoT Greengrass Core Software v1.5.0 および AWS IoT Greengrass Core SDK v1.1.0 から利用できます。デバイスのハードウェア機能が制限されていているため、JSON データ型を構築することが難しいかできないことが多いので、バイナリ入力データを受け入れることは、関数がデバイスデータとやり取りするために便利です。

> **Note**
>
> Lambda 実行可能ファイル は、バイナリエンコードタイプのみ (JSON ではなく) をサポートします。

[Process arguments] (プロセスの引数)

コマンドライン引数は、Lambda 関数の実行時に渡されます。

環境変数

関数コードとライブラリに設定を動的に渡すことができるキー値ペア。ローカル環境変数は AWS Lambda 関数環境変数と同様に動作しますが、Core 環境で利用可能です。

[リソースアクセスポリシー]

Lambda 関数がアクセスを許可される、最大で 10 個までのローカルリソース、シークレットリソース、および機械学習リソースと、対応する `read-only` または `read-write` アクセス許可のリスト。コンソールで、これらのアフィリエイトリソースは、[Resources] (リソース) タブのグループ設定ページに掲載されています。

コンテナ化モードは、Lambda 関数がローカルデバイスとボリュームリソース、機械学習リソースにアクセスする方法に影響します。

- コンテナ化されていない Lambda 関数は、コアデバイス上のファイルシステムを介してローカルデバイスおよびボリュームリソースに直接アクセスする必要があります。
- コンテナ化されていない Lambda 関数が Greengrass グループ内の機械学習リソースにアクセスできるようにするには、機械学習リソースでリソースの所有者とアクセス権限のプロパティを設定する必要があります。詳細については、「the section called “機械学習リソースにアクセスする”」を参照してください。

AWS IoT Greengrass API を使用してユーザー定義の Lambda 関数のグループ固有の設定を行う方法については、 AWS IoT Greengrass Version 1 API リファレンスの CreateFunctionDefinition または AWS CLI コマンドリファレンスの create-function-definition を参照してください。Greengrass コアに Lambda 関数をデプロイするには、関数を含む関数定義バージョンを作成し、関数定義バージョンと他のグループコンポーネントを参照するグループバージョンを作成してから、グループをデプロイします。

root としての Lambda 関数の実行

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

1 つ以上の Lambda 関数をルートとして実行する前に、まず AWS IoT Greengrass 設定を更新してサポートを有効にする必要があります。デフォルトでは、root として Lambda 関数を実行するためのサポートはオフになっています。Lambda 関数をデプロイしてルート (UID と GID 0) として実行しようとし、 AWS IoT Greengrass 設定を更新していない場合、デプロイは失敗します。ランタイムログ (*greengrass_root*/ggc/var/log/system/runtime.log) に以下のようなエラーが表示されます。

```
lambda(s)
[list of function arns] are configured to run as root while Greengrass is not
 configured to run lambdas with root permissions
```

> **Important**
>
> やむを得ない場合を除き、Lambda 関数を root として実行することは避けてください。root として実行すると、次のリスクが増加します。
>
> - 重要なファイルを誤って削除するなど、意図しない変更が行われるリスク。
> - 悪意のあるユーザーがデータやデバイスにアクセスするリスク。
> - Docker コンテナが `--net=host` や `UID=EUID=0` で実行されると、コンテナがエスケープするリスク。

root として実行することを Lambda 関数に許可するには

1. AWS IoT Greengrass デバイスで、*greengrass-root*/config フォルダに移動します。

   > **Note**
   >
   > デフォルトでは、*greengrass-root* は /greengrass ディレクトリです。

2. config.json ファイルを編集し、`"allowFunctionsToRunAsRoot" : "yes"` を `runtime` フィールドに追加します。以下に例を示します。

```
{
  "coreThing" : {
    ...
  },
  "runtime" : {
    ...
    "allowFunctionsToRunAsRoot" : "yes"
  },
  ...
}
```

3. 次のコマンドを使用して再起動します AWS IoT Greengrass。

```
cd /greengrass/ggc/core
sudo ./greengrassd restart
```

   これで Lambda 関数のユーザー ID とグループ ID (UID/GID) を 0 に設定し、root として Lambda 関数を実行できます。

Lambda 関数をルートとして実行することを許可 AWS IoT Greengrass しない場合は、 の値を `"allowFunctionsToRunAsRoot"`に変更`"no"`して再起動できます。

Lambda 関数のコンテナ化を選択する場合の考慮事項

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

デフォルトでは、Lambda 関数は AWS IoT Greengrass コンテナ内で実行されます。このコンテナにより、関数とホストが分離され、ホストとコンテナ内の関数の両方でセキュリティが向上します。

コンテナ化を使用せずに Lambda 関数を実行する必要があるユースケースを除き、Greengrass コンテナで関数を実行することをお勧めします。Greengrass コンテナで Lambda 関数を実行することで、リソースへのアクセスを制限する方法をより細かく制御できます。

コンテナ化を使用しないで実行するユースケースの例:

- コンテナモードをサポートしていない AWS IoT Greengrass デバイスで を実行する場合 (例えば、特別な Linux ディストリビューションを使用しているか、カーネルバージョンが古すぎるため) 。
- 独自の OverlayFS がある別のコンテナ環境の Lambda 関数を、Greengrass コンテナで実行すると、OverlayFS の競合が発生する場合。
- アクセス先のローカルリソースのパスがデプロイ時に決定できないか、デプロイ後に変わることがある場合 (プラガブルデバイスなど)。
- プロセスとして記述されたレガシーアプリケーションがあり、これをコンテナ化された Lambda 関数として実行するときに問題が発生する場合。

コンテナ化の相違点

コンテナ化	メモ
Greengrass コンテナ	• Greengrass コンテナで Lambda 関数を実行すると、すべての AWS IoT Greengrass 機能を使用できます。 • Greengrass コンテナで実行する Lambda 関数は、他の Lambda 関数のデプロイ済みコードにアクセスできません (同じグループ ID を使用している場合でも)。つまり、Lambda 関数相互は分離されて実行されます。 • AWS IoT Greengrass コンテナで実行される Lambda 関数では、すべての子プロセスが Lambda 関数と同じコンテナで実行されるため、子プロセスは Lambda 関数が終了すると終了します。
コンテナなし	• 以下の機能は、コンテナ化されていない Lambda 関数では使用できません。 • Lambda 関数のメモリ制限。

コンテナ化	メモ
	• ローカルデバイスおよびボリュームリソース。Greengrass グループのメンバーとしてアクセスする代わりに、コアデバイス上のこれらのリソースに直接アクセスする必要があります。 • コンテナ化されていない Lambda 関数が機械学習リソースにアクセスする場合、リソース所有者を指定し、Lambda 関数ではなくリソースにアクセス権限を設定する必要があります。これには AWS IoT Greengrass Core ソフトウェア v1.10 以降が必要です。詳細については、「the section called “機械学習リソースにアクセスする”」を参照してください。 • Lambda 関数は、同じグループ ID で実行されている他の Lambda 関数のデプロイ済みコードに対して読み取り専用アクセス許可があります。 • 別のプロセスセッションで子プロセスを生成する Lambda 関数、または、 ノードアップユーティリティを使用するなど、オーバーライドされた SIGHUP (シグナルハングアップ) ハンドラーを使用して子プロセスを生成する Lambda 関数は、親 Lambda 関数が終了した AWS IoT Greengrass ときに によって自動的に終了されません。

> **Note**
>
> Greengrass グループの既定のコンテナ化設定は、コネクタには適用されません。

Lambda 関数のコンテナ化を変更すると、デプロイ時に問題が発生する場合があります。ローカルリソースを割り当てた Lambda 関数が新しいコンテナ化の設定で使用できなくなった場合、デプロイは失敗します。

- Greengrass コンテナでの実行からコンテナ化を使用しない実行へと Lambda 関数を変更すると、関数のメモリ制限は破棄されます。アタッチ済みのローカルリソースを使用する代わりに、ファイルシステムに直接アクセスする必要があります。デプロイする前に、すべてのアタッチ済みリソースを削除する必要があります。
- コンテナ化を使用しない実行からコンテナでの実行へと Lambda 関数を変更すると、Lambda 関数はファイルシステムに直接アクセスできなくなります。関数ごとにメモリ制限を定義するか、デフォルトの 16 MB を受け入れる必要があります。これらの設定は、デプロイ前に Lambda 関数ごとに構成できます。

Lambda 関数のコンテナ化の設定を変更するには

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. 設定を変更する Lambda 関数が含まれているグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. 変更する Lambda 関数で、省略記号 (…) を選択し、次に [Edit configuration] (設定の編集) を選択します。
5. コンテナ化の設定を変更します。Greengrass コンテナで実行するように Lambda 関数を設定する場合は、[Memory limit] (メモリ制限) と [Read access to /sys directory] (/sys ディレクトリへの読み込みアクセス) も設定する必要があります。
6. [Save] (保存) を選択してから [Confirm] (確認) を選択して Lambda 関数への変更を保存します。

変更は、グループのデプロイ時に反映されます。

また、「AWS IoT Greengrass API リファレンス」の「CreateFunctionDefinition」および「CreateFunctionDefinitionVersion」を使用することもできます。コンテナ化の設定を変更する場合は、他のパラメータも必ず更新してください。例えば、Lambda 関数を Greengrass コンテナでの実行からコンテナ化を使用しない実行へと変更する場合は、`MemorySize` パラメータを必ずクリアします。

Greengrass デバイスでサポートされている分離モードの確認

AWS IoT Greengrass 依存関係チェッカーを使用して、Greengrass デバイスでサポートされている分離モード (Greengrass コンテナ/コンテナなし) を判断できます。

AWS IoT Greengrass 依存関係チェッカーを実行するには

1. GitHub リポジトリから AWS IoT Greengrass 依存関係チェッカーをダウンロードして実行します。

```
wget https://github.com/aws-samples/aws-greengrass-samples/raw/master/greengrass-
dependency-checker-GGCv1.11.x.zip
unzip greengrass-dependency-checker-GGCv1.11.x.zip
cd greengrass-dependency-checker-GGCv1.11.x
sudo modprobe configs
sudo ./check_ggc_dependencies | more
```

2. 「`more`」と表示された場合は、Spacebar キーを押して別のページのテキストを表示します。

modprobe コマンドの詳細については、ターミナルで man modprobe を実行してください。

グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定

この機能は AWS IoT Greengrass Core v1.8 以降で使用できます。

デバイスリソースへのアクセスをより細かく制御するために、グループ内で Lambda 関数を実行するために使用されるデフォルトのアクセス ID を設定できます。この設定は、Lambda 関数がコアデバイスで実行されたときに付与されるデフォルトのアクセス許可を決定します。グループ内の個々の関数の設定を上書きするには、その関数の [Run as (として実行)] プロパティを使用できます。詳細については、「Run as」を参照してください。

このグループレベルの設定は、基盤となる AWS IoT Greengrass Core ソフトウェアの実行にも使用されます。これは、メッセージルーティング、ローカルシャドウ同期、および自動 IP アドレス検出などのオペレーションを管理するシステム Lambda 関数で構成されます。

デフォルトのアクセスアイデンティティは、標準 AWS IoT Greengrass システムアカウント (ggc_user および ggc_group) として実行するように設定することも、別のユーザーまたはグループのアクセス許可を使用するように設定することもできます。Greengrass ハードウェアの設定では、適切なリソース制限、適切なファイルへのアクセス許可、およびユーザー定義またはシステム

Lambda 関数を実行するために使用されるアクセス許可を持つユーザーとグループの適切なディスククォータを使用することをお勧めします。

AWS IoT Greengrass グループのデフォルトのアクセスアイデンティティを変更するには

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. 設定を変更するグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択し、[Default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境)セクションで [Edit] (編集) を選択します。
4. [Edit default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境の編集) ページの [Default system user and group] (デフォルトシステムユーザーおよびグループ) で、[Another user ID/group ID] (別のユーザー ID とグループ ID) を選択します。

   このオプションを選択すると、[System user ID (number)] (システムユーザー ID (数値)) および [System group ID (number)] (システムグループ ID (数値)) フィールドが表示されます。
5. ユーザー ID、グループ ID、またはその両方を入力します。フィールドを空白のままにすると、それぞれの Greengrass システムアカウント (ggc_user または ggc_group) が使用されます。

   - [System user ID (number)] (システムユーザー ID (数値)) で、グループ内で Lambda 関数を実行するためにデフォルトで使用するアクセス許可を持つユーザーのユーザー ID を入力します。 AWS IoT Greengrass デバイスで getent passwd コマンドを使用して、ユーザー ID を検索できます。
   - [System group ID (number)] (システムグループ ID (数値)) で、グループ内で Lambda 関数を実行するためにデフォルトで使用するアクセス許可を持つグループのグループ ID を入力します。 AWS IoT Greengrass デバイスで getent group コマンドを使用して、グループ ID を検索できます。

> **Important**
>
> root ユーザーとして実行すると、データとデバイスに対するリスクが増大します。ビジネスケースで要求されている場合を除き、root (UID/GID=0) として実行しないでください。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

変更は、グループのデプロイ時に反映されます。

グループ内の Lambda 関数のコンテナ化のデフォルト設定

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

Greengrass グループのコンテナ化設定によって、グループ内の Lambda 関数のデフォルトのコンテナ化が決定されます。

- Greengrass コンテナモードでは、Lambda 関数はデフォルトで AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。
- コンテナなしモードでは、Lambda 関数はデフォルトで通常の Linux プロセスとして実行されます。

グループ設定を変更して、グループ内の Lambda 関数のデフォルトのコンテナ化を指定できます。グループのデフォルトとは異なるコンテナ化を使用して Lambda 関数を実行する場合は、グループ内の 1 つ以上の Lambda 関数について、この設定を上書きできます。コンテナ化の設定を変更する場合は、事前に「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

> **Important**
>
> グループのデフォルトのコンテナ化を変更する際に、一部の関数で別のコンテナ化を使用するという場合は、グループの設定を変更する前に Lambda 関数の設定を変更します。グループのコンテナ化の設定を最初に変更すると、[メモリ制限] と [/sys ディレクトリへの読み込みアクセス] に設定した値は破棄されます。

AWS IoT Greengrass グループのコンテナ化設定を変更するには

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. 設定を変更するグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. [Default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境) から、[Edit] (編集) を選択します。

5. [Edit default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境を編集する) ページの、[Default Lambda function containerization] (デフォルトの Lambda 関数のコンテナ化) でコンテナ化の設定を変更します。
6. [Save] を選択します。

変更は、グループのデプロイ時に反映されます。

Greengrass Lambda 関数のコミュニケーションフロー

Greengrass Lambda 関数は、 AWS IoT Greengrass グループの他のメンバー、ローカルサービス、クラウドサービス (AWS サービスを含む) と通信するいくつかの方法をサポートしています。

MQTT メッセージを使用した通信

Lambda 関数は、サブスクリプションによって制御される発行 - サブスクリプションパターンを使用して MQTT メッセージを送受信できます。

このコミュニケーションフローにより、Lambda 関数は以下のエンティティとメッセージを交換することができる。

- グループ内のクライアントデバイス。
- グループのコネクタ。
- グループ内の他の Lambda 関数。
- AWS IoT.
- ローカルデバイスシャドウサービス。

サブスクリプションは、メッセージ送信元、メッセージターゲット、および送信元からターゲットへのメッセージのルーティングに使用されるトピック (または件名) を定義します。Lambda 関数に発行されるメッセージは、関数に登録されたハンドラに渡されます。サブスクリプションはより高度なセキュリティを可能にし、予測可能なやり取りを提供します。詳細については、「the section called “MQTT メッセージングワークフローにおけるマネージドサブスクリプション”」を参照してください。

> **Note**
>
> Greengrass Lambda 関数は、クライアントデバイス、コネクタ、他の関数、およびコアがオフラインのときにローカルシャドウとメッセージを交換できますが、 AWS IoT へのメッ

セージはキュー状態になります。詳細については、「the section called “MQTT メッセージキュー”」を参照してください。

他の通信フロー

- コアデバイスでローカルデバイスおよびボリュームリソース、機械学習モデルとやり取りするため、Greengrass Lambda 関数はプラットフォーム固有のオペレーティングシステムインターフェイスを使用します。例えば、Python 関数で os モジュールの open メソッドを使用できます。関数がリソースにアクセスすることを許可するには、この関数がリソースに関連し、`read-only` あるいは `read-write` アクセス許可を付与されていることが必要です。 AWS IoT Greengrass コアバージョンの可用性などの詳細については、ローカルリソースへのアクセス「」および「」を参照してくださいthe section called “Lambda 関数コードから機械学習リソースにアクセスする”。

  > **Note**
  > コンテナ化せずに Lambda 関数を実行する場合、アタッチされたローカルデバイスおよびボリュームリソースを使用することはできず、直接それらのリソースにアクセスする必要があります。

- Lambda 関数は、 AWS IoT Greengrass Core SDK の Lambdaクライアントを使用して、Greengrass グループ内の他の Lambda 関数を呼び出すことができます。
- Lambda 関数は AWS SDK を使用して AWS サービスと通信できます。詳細については、「AWS SDK」をご参照ください。
- Lambda 関数は、クラウドベースの Lambda 関数のように、サードパーティーのインターフェイスを使用して外部のクラウドサービスと通信できます。

> **Note**
> Greengrass Lambda 関数は、コアがオフラインの場合、 AWS や他のクラウドサービスと通信できません。

入力 MQTT トピック (または件名) の取得

AWS IoT Greengrass はサブスクリプションを使用して、グループ内のクライアントデバイス、Lambda 関数、コネクタ間、および AWS IoT またはローカルシャドウサービスとの MQTT メッセージの交換を制御します。サブスクリプションは、メッセージソース、メッセージターゲット、およびメッセージのルーティングに使用される MQTT トピックを定義します。ターゲットが Lambda 関数の場合、ソースがメッセージを発行すると、関数のハンドラが呼び出されます。詳細については、「the section called “MQTT メッセージを使用した通信”」を参照してください。

次の例は、Lambda 関数がハンドラに渡された `context` から入力トピックを取得する方法を示しています。これを行うには、コンテキスト階層から `subject` キーにアクセスします (`context.client_context.custom['subject']`)。この例では、入力 JSON メッセージも解析してから、解析したトピックとメッセージを発行します。

> **Note**
>
> AWS IoT Greengrass API では、サブスクリプションのトピックは `subject`プロパティで表されます。

```
import greengrasssdk
import logging

client = greengrasssdk.client('iot-data')

OUTPUT_TOPIC = 'test/topic_results'

def get_input_topic(context):
    try:
        topic = context.client_context.custom['subject']
    except Exception as e:
        logging.error('Topic could not be parsed. ' + repr(e))
    return topic

def get_input_message(event):
    try:
        message = event['test-key']
    except Exception as e:
        logging.error('Message could not be parsed. ' + repr(e))
    return message
```

```
def function_handler(event, context):
    try:
        input_topic = get_input_topic(context)
        input_message = get_input_message(event)
        response = 'Invoked on topic "%s" with message "%s"' % (input_topic,
 input_message)
        logging.info(response)
    except Exception as e:
        logging.error(e)

    client.publish(topic=OUTPUT_TOPIC, payload=response)

    return
```

関数をテストするには、デフォルトの設定を使用してグループに追加します。次に、以下のサブスクリプションを追加し、グループをデプロイします。手順については、the section called “モジュール 3 (パート 1): での Lambda 関数 AWS IoT Greengrass” を参照してください。

次のフィルター	IoT/test/input
関数のコード	get_input_message
次のフィルター	IoT/test/output
関数のコード	get_results

デプロイが完了したら、関数を呼び出します。

1. AWS IoT コンソールで、MQTT テストクライアントページを開きます。
2. [Subscribe to a topic] (トピックへのサブスクライブ) タブを選択して、`test/topic_results` をサブスクライブします。
3. [Publish to a topic] (トピックへの発行) タブを選択して `test/input_message` トピックを発行します。この例では、JSON メッセージに `test-key` プロパティを含める必要があります。

   ```
   {
     "test-key": "Some string value"
   }
   ```

   成功した場合、関数は入力トピックとメッセージ文字列を `test/topic_results` トピックに発行します。

Greengrass Lambda 関数のライフサイクル設定

Greengrass Lambda 関数ライフサイクルは、関数が開始する時期とどのようにコンテナを作成して使用するかを定義します。また、ライフサイクルは関数ハンドラの外部にある変数および処理中のロジックが保持されるかを定義します。

AWS IoT Greengrass は、オンデマンド (デフォルト) または存続期間の長いライフサイクルをサポートします。

- オンデマンド 関数は、呼び出されたときに起動し、実行するタスクが残っていないときに停止します。関数の呼び出しは、再使用できる既存のコンテナが利用可能な場合を除き、呼び出し処理に別のコンテナ (またはサンドボックス) を作成します。関数に送信されたデータは、いずれかのコンテナによってプルされる可能性があります。

  関数の複数の呼び出しは、同時に実行できます。

  関数ハンドラの外部で定義される変数や前処理ロジックは、新しいコンテナが作成されるときに保持されません。

- 存続期間の長い (または固定された) 関数は、 AWS IoT Greengrass コアが 1 つのコンテナで起動および実行されると自動的に開始されます。関数に送信されたすべてのデータは、同じコンテナによってプルされます。

複数の呼び出しは、前の呼び出しが実行されるまでキュー状態になります。

関数ハンドラの外部で定義される変数と事前処理ロジックは、このハンドラの毎回の呼び出しのために保持されます。

長い存続期間の Lambda 関数は、初期の入力が全くない実行を開始する必要がある場合に便利です。例えば、存続期間が長い関数は、関数がデバイスデータの受信を開始するときに備えて、ML モデルをロードして処理を開始できます。

> **Note**
>
> 長い存続期間の関数には、そのハンドラの呼び出しに関連付けられたタイムアウトがあることに注意してください。実行中のコードを無期限で実行する場合には、ハンドラ外でこれを開始する必要があります。関数の初期化の完了を妨害するようなハンドラ外のブロックコードがないことを確認します。
> これらの関数は、コアが停止する (グループのデプロイ中やデバイスの再起動中など) か、関数がエラー状態 (ハンドラのタイムアウト、キャッチされない例外、またはメモリ制限を超えたときなど) にならない限り実行されます。

コンテナの再利用の詳細については、 AWS コンピューティングブログの「 でのコンテナの再利用について AWS Lambda」を参照してください。

Lambda 実行可能ファイル

この機能は AWS IoT Greengrass Core v1.6 以降で使用できます。

Lambda 実行可能ファイルは、コア環境でバイナリコードを実行するために使用できる Greengrass Lambda 関数の一種です。これによって、デバイス固有の機能を実行することができ、コンパイルされたコードの小さなフットプリントの利点を活用できます。Lambda 実行可能ファイルは、イベントによる呼び出し、他の関数の呼び出し、そしてローカルリソースにアクセスが可能です。

Lambda 実行可能ファイルはバイナリエンコードタイプのみ (JSON ではなく) をサポートします。それ以外では、Greengrass グループで管理し、他の Greengrass Lambda 関数のようにデプロイできます。ただし、Lambda 実行可能ファイルを作成するプロセスは、Python、Java、および Node.js の Lambda 関数とは異なります。

- AWS Lambda コンソールを使用して Lambda 実行可能ファイルを作成 (または管理) することはできません。Lambda 実行可能ファイルは、 AWS Lambda API を使用してのみ作成できます。
- 関数コードを Core AWS IoT Greengrass SDK for C を含むコンパイルされた実行可能ファイル AWS Lambda として にアップロードします。
- 実行可能ファイル名を関数のハンドラとして指定します。

Lambda 実行可能ファイルでは、その関数コードに特定の呼び出しおよびプログラミングパターンが実装されている必要があります。例えば、`main` メソッドは次を満たす必要があります。

- Greengrass 内部グローバル変数を初期化する `gg_global_init` 呼び出し。この関数は、スレッドを作成する前、および他の AWS IoT Greengrass Core SDK 関数を呼び出す前に呼び出す必要があります。
- Greengrass Lambda ランタイムに関数ハンドラを登録する `gg_runtime_start` の呼び出し。この関数は初期化中に呼び出す必要があります。この関数を呼び出すと、現在のスレッドはランタイムで使用されます。オプションの `GG_RT_OPT_ASYNC` パラメータはこの関数をブロックしませんが、代わりにランタイムに新規のスレッドを作成します。この関数は `SIGTERM` ハンドラを使用します。

次のスニペットは、GitHub の `mainsimple_handler.c` コード例の メソッドです。

```
int main() {
    gg_error err = GGE_SUCCESS;

    err = gg_global_init(0);
    if(err) {
        gg_log(GG_LOG_ERROR, "gg_global_init failed %d", err);
        goto cleanup;
    }

    gg_runtime_start(handler, 0);

cleanup:
    return -1;
}
```

実装の要件や制限などの詳細については、「AWS IoT Greengrass Core SDK for C」を参照してください。

Lambda 実行可能ファイルを作成する

SDK とともにコードをコンパイルしたら、 AWS Lambda API を使用して Lambda 関数を作成し、コンパイルされた実行可能ファイルをアップロードします。

> **Note**
>
> 関数は、C89 互換性のあるコンパイラーでコンパイルする必要があります。

次の例では、create-function CLI コマンドを使用して、Lambda 実行可能ファイルを作成します。このコマンドは以下を指定します。

- ハンドラの実行可能ファイルの名前。これは、コンパイルされた実行可能ファイルの正確な名前である必要があります。
- コンパイルされた実行可能ファイルを含む `.zip` ファイルへのパス。
- ランタイムの `arn:aws:greengrass:::runtime/function/executable`。これはすべて Lambda 実行可能ファイルのランタイムです。

> **Note**
>
> `role` では、任意の Lambda 実行ロールの ARN を指定できます。 AWS IoT Greengrass はこのロールを使用しませんが、関数を作成するにはパラメータが必要です。Lambda 実行可能ファイルのロールの詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda アクセス権限モデル」を参照してください。

```
aws lambda create-function \
--region aws-region \
--function-name function-name \
--handler executable-name \
--role role-arn \
--zip-file fileb://file-name.zip \
--runtime arn:aws:greengrass:::runtime/function/executable
```

次に、 AWS Lambda API を使用してバージョンを公開し、エイリアスを作成します。

- publish-version を使用して、関数バージョンを発行します。

```
aws lambda publish-version \
--function-name function-name \
--region aws-region
```

- create-alias を使用して、先ほど発行したバージョンを指すエイリアスを作成します。Greengrass グループ に Lambda 関数を追加する場合、この関数をエイリアスで参照することが推奨されます。

```
aws lambda create-alias \
--function-name function-name \
--name alias-name \
--function-version version-number \
--region aws-region
```

> **Note**
>
> AWS Lambda コンソールには Lambda 実行可能ファイルは表示されません。関数コードを更新するには、 AWS Lambda API を使用する必要があります。

次に、Lambda 実行可能ファイルを Greengrass グループに追加し、このグループ固有の設定でバイナリ入力データを受け入れるように設定して、グループをデプロイします。これは、 AWS IoT Greengrass コンソールまたは AWS IoT Greengrass API を使用して実行できます。

Docker コンテナ AWS IoT Greengrass での実行

AWS IoT Greengrass は、Docker コンテナで実行するように設定できます。

AWS IoT Greengrass Core ソフトウェアと依存関係がインストールされている Dockerfile は、Amazon CloudFront からダウンロードできます。Docker イメージを変更してさまざまなプラットフォームアーキテクチャで実行するか、Docker イメージのサイズを小さくするには、Docker パッケージダウンロードの `README` ファイルを参照してください。

の実験を開始できるように AWS IoT Greengrass、 には AWS IoT Greengrass Core ソフトウェアと依存関係がインストールされている構築済みの Docker イメージ AWS も用意されています。イメージは、Docker Hub または Amazon Elastic Container Registry(Amazon ECR) からダウンロードできます。これらの構築済みのイメージでは、Amazon Linux 2 (x86_64) および Alpine Linux (x86_64、Armv7l、または AArch64) のベースイメージを使用します。

> **Important**
>
> 2022 年 6 月 30 日に、Amazon Elastic Container Registry (Amazon ECR) および Docker Hub に公開された AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージのメンテナンス AWS IoT Greengrass が終了しました。これらの Docker イメージは、メンテナンス終了から 1 年後の 2023 年 6 月 30 日まで、Amazon ECR および Docker Hub から引き続きダウンロードすることができます。ただし、2022 年 6 月 30 日にメンテナンスが終了した後、AWS IoT Greengrass Core ソフトウェア v1.x Docker イメージはセキュリティパッチやバグ修正を受け取らなくなりました。これらの Docker イメージに依存する本稼働ワークロードを実行する場合は、 AWS IoT Greengrass が提供する Dockerfiles を使用して独自の Docker イメージを構築することをお勧めします。詳細については、「AWS IoT Greengrass Docker ソフトウェア」を参照してください。

このトピックでは、Amazon ECR から AWS IoT Greengrass Docker イメージをダウンロードし、Windows、macOS、または Linux (x86_64) プラットフォームで実行する方法について説明します。以下の各ステップを示します。

1. Amazon ECR から AWS IoT Greengrass コンテナイメージを取得する
2. Greengrass のグループとコアを作成して設定する
3. AWS IoT Greengrass ローカルで を実行する
4. グループの「コンテナなし」コンテナ化を設定する
5. Lambda 関数を Docker コンテナにデプロイする
6. (オプション) Docker コンテナで Greengrass を操作するクライアントデバイスをデプロイする

Docker コンテナ AWS IoT Greengrass で を実行する場合、以下の機能はサポートされていません。

- [Greengrass container] (Greengrass コンテナ) モードで実行されるコネクタ。Docker コンテナでコネクタを実行するには、コネクタをコンテナなしモードで実行する必要があります。コンテナなしモードをサポートするコネクタを検索するには、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。これらのコネクタの一部では、分離モードパラメータを使用されており、[コンテナなし] に設定する必要があります。
- ローカルデバイスおよびボリュームリソース。Docker コンテナで実行されるユーザー定義 Lambda 関数は、コア上のデバイスとボリュームに直接アクセスする必要があります。

これらの機能は、Greengrass グループの Lambda ランタイム環境が、Docker コンテナで実行するために必要なコンテナなしに設定されている場合はサポートされていません。 AWS IoT Greengrass

前提条件

このチュートリアルを開始する前に、以下を実行する必要があります。

- 選択した AWS Command Line Interface （ AWS CLI) バージョンに基づいて、ホストコンピュータに次のソフトウェアとバージョンをインストールする必要があります。

  AWS CLI version 2

  - Docker バージョン 18.09 以降。以前のバージョンでも動作する可能性がありますが、18.09 以降を推奨します。
  - AWS CLI バージョン 2.0.0 以降。
    - AWS CLI バージョン 2 をインストールするには、「バージョン 2 AWS CLIのインストール」を参照してください。
    - を設定するには AWS CLI、「 の設定 AWS CLI」を参照してください。

    > **Note**
    > Windows コンピュータで新しい AWS CLI バージョン 2 にアップグレードするには、MSI のインストールプロセスを繰り返す必要があります。

  AWS CLI version 1

  - Docker バージョン 18.09 以降。以前のバージョンでも動作する可能性がありますが、18.09 以降を推奨します。
  - Python バージョン 3.6 以降。
  - pip バージョン 18.1 以降。
  - AWS CLI バージョン 1.17.10 以降
    - AWS CLI バージョン 1 をインストールするには、「バージョン 1 AWS CLIのインストール」を参照してください。
    - を設定するには AWS CLI、「 の設定 AWS CLI」を参照してください。
    - バージョン 1 AWS CLI の最新バージョンにアップグレードするには、次のコマンドを実行します。

      ```
      pip install awscli --upgrade --user
      ```

> **Note**
>
> Windows で AWS CLI バージョン 1 の MSI インストールを使用する場合は、次の点に注意してください。
>
> - AWS CLI バージョン 1 のインストールで botocore のインストールに失敗する場合は、Python と pip のインストールを試してください。
> - 新しい AWS CLI バージョン 1 にアップグレードするには、MSI のインストールプロセスを繰り返す必要があります。

- ユーザーが Amazon Elastic Container Registry (Amazon ECR) のリソースにアクセスできるようにするには、次のアクセス権限を付与する必要があります。
  - Amazon ECR では、レジストリに対して認証し、Amazon ECR リポジトリからイメージをプッシュまたはプルする前に、 AWS Identity and Access Management (IAM) ポリシーを通じてアクセス`ecr:GetAuthorizationToken`許可を付与する必要があります。詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR Repository Policy Examples」(Amazon ECR リポジトリポリシーの例) および「1 つの Amazon ECR リポジトリにアクセスする」を参照してください。

ステップ 1: Amazon ECR から AWS IoT Greengrass コンテナイメージを取得する

AWS は、 AWS IoT Greengrass Core ソフトウェアがインストールされている Docker イメージを提供します。

> **Warning**
>
> AWS IoT Greengrass Core ソフトウェアの v1.11.6 以降、Greengrass Docker イメージには Python 2.7 が含まれなくなりました。Python 2.7 は 2020 年にend-of-lifeとなり、セキュリティ更新プログラムを受け取らなくなったためです。これらの Docker イメージに更新する場合は、アプリケーションが新しい Docker イメージで動作することを検証した後に、アップデートを本番デバイスに展開することをお勧めします。Greengrass Docker イメージを使用するアプリケーションに Python 2.7 が必要な場合は、Greengrass Dockerfile を変更して、アプリケーションに Python 2.7 を含めることができます。

Amazon ECR から `latest` イメージをプルする方法については、お使いのオペレーティングシステムを選択してください。

コンテナイメージをプルする (Linux)

コンピュータのターミナルで以下のコマンドを実行します。

1. Amazon ECR の AWS IoT Greengrass レジストリにログインします。

```
aws ecr get-login-password --region  us-west-2 | docker login --username AWS --
password-stdin https://216483018798.dkr.ecr.us-west-2.amazonaws.com
```

   成功すると、`Login Succeeded` が出力されます。

2. AWS IoT Greengrass コンテナイメージを取得します。

```
docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

> **Note**
>
> このlatestイメージには、Amazon Linux 2 ベースイメージにインストールされている AWS IoT Greengrass Core ソフトウェアの最新の安定バージョンが含まれています。他のイメージをリポジトリからプルすることもできます。使用可能なすべてのイメージを確認するには、Docker Hub の [Tags] (タグ) ページを確認するか、aws ecr list-images コマンドを使用してください。例:
>
> ```
> aws ecr list-images --region us-west-2 --registry-id 216483018798 --
> repository-name aws-iot-greengrass
> ```

3. シンボリックリンクとハードリンクの保護を有効にします。コンテナ AWS IoT Greengrass での実行を試している場合は、現在のブートの設定のみを有効にできます。

> **Note**
>
> これらのコマンドを実行するには、sudo を使用することが必要な場合があります。

   - 現在のブートに対してのみ設定を有効にするには、以下のコマンドを使用します。

```
echo 1 > /proc/sys/fs/protected_hardlinks
echo 1 > /proc/sys/fs/protected_symlinks
```

- 再起動しても維持される設定を有効にするには、以下のコマンドを使用します。

```
echo '# AWS IoT Greengrass' >> /etc/sysctl.conf
echo 'fs.protected_hardlinks = 1' >> /etc/sysctl.conf
echo 'fs.protected_symlinks = 1' >> /etc/sysctl.conf

sysctl -p
```

4. AWS IoT Greengrass クラウドデプロイと MQTT 通信が Linux で動作するために必要な IPv4 ネットワーク転送を有効にします。`/etc/sysctl.conf` ファイルで、`net.ipv4.ip_forward` を 1 に設定して、`sysctls` を再ロードします。

```
sudo nano /etc/sysctl.conf
# set this net.ipv4.ip_forward = 1
sudo sysctl -p
```

> **Note**
>
> nano の代わりに任意のエディタを使用できます。

コンテナイメージをプルする (macOS)

コンピュータのターミナルで以下のコマンドを実行します。

1. Amazon ECR の AWS IoT Greengrass レジストリにログインします。

```
aws ecr get-login-password --region  us-west-2 | docker login --username AWS --
password-stdin https://216483018798.dkr.ecr.us-west-2.amazonaws.com
```

成功すると、`Login Succeeded` が出力されます。

2. AWS IoT Greengrass コンテナイメージを取得します。

```
docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

> **Note**
>
> このlatestイメージには、Amazon Linux 2 ベースイメージにインストールされている AWS IoT Greengrass Core ソフトウェアの最新の安定バージョンが含まれています。他のイメージをリポジトリからプルすることもできます。使用可能なすべてのイメージを確認するには、Docker Hub の [Tags] (タグ) ページを確認するか、aws ecr list-images コマンドを使用してください。例:
>
> ```
> aws ecr list-images --region us-west-2 --registry-id 216483018798 --
> repository-name aws-iot-greengrass
> ```

コンテナイメージをプルする (Windows)

コマンドプロンプトで次のコマンドを実行します。Windows で Docker コマンドを使用する前に、Docker デスクトップが実行されている必要があります。

1. Amazon ECR の AWS IoT Greengrass レジストリにログインします。

   ```
   aws ecr get-login-password --region  us-west-2 | docker login --username AWS --
   password-stdin https://216483018798.dkr.ecr.us-west-2.amazonaws.com
   ```

   成功すると、`Login Succeeded` が出力されます。

2. AWS IoT Greengrass コンテナイメージを取得します。

   ```
   docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
   ```

   > **Note**
   >
   > このlatestイメージには、Amazon Linux 2 ベースイメージにインストールされている AWS IoT Greengrass Core ソフトウェアの最新の安定バージョンが含まれています。他のイメージをリポジトリからプルすることもできます。使用可能なすべてのイメージを確認するには、Docker Hub の [Tags] (タグ) ページを確認するか、aws ecr list-images コマンドを使用してください。例:

```
aws ecr list-images --region us-west-2 --registry-id 216483018798 --
repository-name aws-iot-greengrass
```

ステップ 2: Greengrass のグループとコアを作成して設定する

Docker イメージには AWS IoT Greengrass Core ソフトウェアがインストールされています が、Greengrass グループと Core を作成する必要があります。これには、証明書とコア設定ファイルのダウンロードが含まれます。

- 「the section called “モジュール 2: AWS IoT Greengrass Core ソフトウェアのインストール”」のステップを実行してください。 AWS IoT Greengrass Core ソフトウェアをダウンロードして実行するステップをスキップします。このソフトウェアとその実行時の依存関係は、Docker イメージにセットアップ済みです。

ステップ 3: AWS IoT Greengrass ローカルで を実行する

グループの設定が完了したら、コアを設定して開始する準備ができました。これを実行する方法を示す手順については、以下でオペレーティングシステムを選択します。

Greengrass をローカルで実行する (Linux)

コンピュータのターミナルで以下のコマンドを実行します。

1. デバイスのセキュリティリソース用のフォルダを作成し、証明書とキーをそのフォルダに移動します。以下のコマンドを実行します。*path-to-security-files* (セキュリティファイルへのパス) をセキュリティリソースへのパスに置き換え、*certificateId* をファイル名に含まれる証明書 ID に置き換えます。

   ```
   mkdir /tmp/certs
   mv path-to-security-files/certificateId-certificate.pem.crt /tmp/certs
   mv path-to-security-files/certificateId-public.pem.key /tmp/certs
   mv path-to-security-files/certificateId-private.pem.key /tmp/certs
   mv path-to-security-files/AmazonRootCA1.pem /tmp/certs
   ```

2. デバイスの設定用のフォルダを作成し、 AWS IoT Greengrass Core 設定ファイルをそのフォルダに移動します。以下のコマンドを実行します。*configuration-file-path* を、設定ファイルへのパスに置き換えます。

```
mkdir /tmp/config
mv path-to-config-file/config.json /tmp/config
```

3. Docker コンテナで証明書と設定ファイルを起動 AWS IoT Greengrass してバインドマウントします。

   /tmp は、証明書と設定ファイルを解凍したパスに置き換えてください。

```
docker run --rm --init -it --name aws-iot-greengrass \
--entrypoint /greengrass-entrypoint.sh \
-v /tmp/certs:/greengrass/certs \
-v /tmp/config:/greengrass/config \
-p 8883:8883 \
216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

   出力は、次の例のようになります。

```
Setting up greengrass daemon
Validating hardlink/softlink protection
Waiting for up to 30s for Daemon to start

Greengrass successfully started with PID: 10
```

Greengrass をローカルで実行する (macOS)

コンピュータのターミナルで以下のコマンドを実行します。

1. デバイスのセキュリティリソース用のフォルダを作成し、証明書とキーをそのフォルダに移動します。以下のコマンドを実行します。*path-to-security-files* (セキュリティファイルへのパス) をセキュリティリソースへのパスに置き換え、*certificateId* をファイル名に含まれる証明書 ID に置き換えます。

```
mkdir /tmp/certs
mv path-to-security-files/certificateId-certificate.pem.crt /tmp/certs
mv path-to-security-files/certificateId-public.pem.key /tmp/certs
mv path-to-security-files/certificateId-private.pem.key /tmp/certs
mv path-to-security-files/AmazonRootCA1.pem /tmp/certs
```

2. デバイスの設定用のフォルダを作成し、 AWS IoT Greengrass Core 設定ファイルをそのフォルダに移動します。以下のコマンドを実行します。*configuration-file-path* を、設定ファイルへのパスに置き換えます。

```
mkdir /tmp/config
mv path-to-config-file/config.json /tmp/config
```

3. Docker コンテナで証明書と設定ファイルを起動 AWS IoT Greengrass してバインドマウントします。

   /tmp は、証明書と設定ファイルを解凍したパスに置き換えてください。

```
docker run --rm --init -it --name aws-iot-greengrass \
--entrypoint /greengrass-entrypoint.sh \
-v /tmp/certs:/greengrass/certs \
-v /tmp/config:/greengrass/config \
-p 8883:8883 \
216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

   出力は、次の例のようになります。

```
Setting up greengrass daemon
Validating hardlink/softlink protection
Waiting for up to 30s for Daemon to start

Greengrass successfully started with PID: 10
```

Greengrass をローカルで実行する (Windows)

1. デバイスのセキュリティリソース用のフォルダを作成し、証明書とキーをそのフォルダに移動します。コマンドプロンプトで次のコマンドを実行します。*path-to-security-files* (セキュリティファイルへのパス) をセキュリティリソースへのパスに置き換え、*certificateId* をファイル名に含まれる証明書 ID に置き換えます。

```
mkdir C:\Users\%USERNAME%\Downloads\certs
move path-to-security-files\certificateId-certificate.pem.crt C:\Users\%USERNAME%
\Downloads\certs
move path-to-security-files\certificateId-public.pem.key C:\Users\%USERNAME%
\Downloads\certs
```

```
move path-to-security-files\certificateId-private.pem.key C:\Users\%USERNAME%
\Downloads\certs
move path-to-security-files\AmazonRootCA1.pem C:\Users\%USERNAME%\Downloads\certs
```

2. デバイスの設定用のフォルダを作成し、 AWS IoT Greengrass Core 設定ファイルをそのフォルダに移動します。コマンドプロンプトで次のコマンドを実行します。*configuration-file-path* を、設定ファイルへのパスに置き換えます。

```
mkdir C:\Users\%USERNAME%\Downloads\config
move path-to-config-file\config.json C:\Users\%USERNAME%\Downloads\config
```

3. Docker コンテナで証明書と設定ファイルを起動 AWS IoT Greengrass してバインドマウントします。コマンドプロンプトで次のコマンドを実行します。

```
docker run --rm --init -it --name aws-iot-greengrass --entrypoint /greengrass-
entrypoint.sh -v c:/Users/%USERNAME%/Downloads/certs:/greengrass/certs
 -v c:/Users/%USERNAME%/Downloads/config:/greengrass/config -p 8883:8883
 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

Docker で `C:\` ドライブを Docker デーモンと共有するように要求されたら、Docker コンテナ内で `C:\` ディレクトリをバインドマウントすることを許可します。詳細については、Docker ドキュメントの「共有ドライブ」を参照してください。

出力は、次の例のようになります。

```
Setting up greengrass daemon
Validating hardlink/softlink protection
Waiting for up to 30s for Daemon to start

Greengrass successfully started with PID: 10
```

> **Note**
>
> コンテナがシェルを開かずにすぐに終了する場合は、イメージを起動したときに Greengrass ランタイムログをバインドマウントすることで問題をデバッグできます。詳細については、「the section called “Docker コンテナの外部で Greengrass ランタイムログを永続化する”」を参照してください。

ステップ 4: Greengrass グループの「コンテナなし」コンテナ化を設定する

Docker コンテナ AWS IoT Greengrass で を実行する場合、すべての Lambda 関数はコンテナ化なしで実行する必要があります。このステップでは、グループのデフォルトのコンテナ化を [No container (コンテナなし)] に設定します。グループを初めてデプロイする前に行う必要があります。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. 設定を変更するグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. [Default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境) から、[Edit] (編集) を選択します。
5. [Edit default Lambda function runtime environment] (デフォルトの Lambda 関数ランタイム環境を編集する) の、[Default Lambda function containerization] (デフォルトの Lambda 関数のコンテナ化) でコンテナ化の設定を変更します。
6. [Save] を選択します。

変更は、グループのデプロイ時に反映されます。

詳細については、「the section called “グループ内の Lambda 関数のコンテナ化のデフォルト設定”」を参照してください。

> **Note**
>
> デフォルトでは、Lambda 関数はグループコンテナ化設定を使用します。 AWS IoT Greengrass を Docker コンテナで実行しているときに、Lambda 関数の [No container] (コンテナなし) 設定を上書きすると、デプロイは失敗します。

ステップ 5: Lambda 関数を Docker AWS IoT Greengrass コンテナにデプロイする

存続期間の長い Lambda 関数を Greengrass Docker コンテナにデプロイできます。

- 「the section called “モジュール 3 (パート 1): での Lambda 関数 AWS IoT Greengrass”」の手順に従って、存続期間の長い Hello World Lambda 関数をコンテナにデプロイします。

ステップ 6: (オプション) Docker コンテナで実行中の Greengrass を操作するクライアントデバイスをデプロイする

Docker コンテナで実行されている AWS IoT Greengrass ときに とやり取りするクライアントデバイスをデプロイすることもできます。

- 「the section called “モジュール 4: AWS IoT Greengrass グループ内のクライアントデバイスを操作する”」の手順に従って、コアに接続するクライアントデバイスをデプロイして、MQTT メッセージを送信します。

Docker AWS IoT Greengrass コンテナの停止

Docker AWS IoT Greengrass コンテナを停止するには、ターミナルまたはコマンドプロンプトで Ctrl+C を押します。このアクションにより、`SIGTERM` が Greengrass デーモンプロセスに送信され、Greengrass デーモンプロセスとデーモンプロセスで開始されたすべての Lambda プロセスが破棄されます。Docker コンテナは `/dev/init` プロセスで PID 1 として初期化されます。これにより、残存するゾンビ状態のプロセスが削除されます。詳細については、Docker run リファレンスを参照してください。

Docker コンテナ AWS IoT Greengrass でのトラブルシューティング

以下の情報は、Docker コンテナ AWS IoT Greengrass で を実行する際の問題のトラブルシューティングに役立ちます。

次のエラーが発生する。Cannot perform an interactive login from a non TTY device。

解決策: `aws ecr get-login-password` コマンドを実行すると、このエラーが発生することがあります。最新の AWS CLI バージョン 2 またはバージョン 1 がインストールされていることを確認してください。 AWS CLI バージョン 2 を使用することをお勧めします。詳細については、AWS CLI ユーザーガイドのAWS Command Line Interface のインストールを参照してください。

エラー「Unknown options: -no-include-email」が発生する。

解決策: `aws ecr get-login` コマンドを実行すると、このエラーが発生することがあります。AWS CLI 最新バージョンがインストールされていることを確認します (例: run: `pip install`

`awscli --upgrade --user`)。Windows を使用していて、MSI インストーラを使用して CLI をインストールした場合は、インストールプロセスを繰り返す必要があります。詳細については、「AWS Command Line Interface ユーザーガイド」の「Microsoft Windows に AWS Command Line Interface をインストールする」を参照してください。

警告: IPv4 is disabled. Networking will not work.

解決策： Linux コンピュータ AWS IoT Greengrass で を実行すると、この警告または同様のメッセージが表示されることがあります。このステップで説明されているようにIPv4 ネットワーク転送を有効にします。IPv4 転送が有効になっていない場合、 AWS IoT Greengrass クラウドデプロイと MQTT 通信は機能しません。詳細については、Docker ドキュメントの「Configure namespaced kernel parameters (sysctls) at runtime」を参照してください。

エラー: A firewall is blocking file Sharing between windows and the containers.

解決策: Windows コンピュータで Docker を実行すると、このエラーまたは `Firewall Detected` メッセージが表示されることがあります。このエラーは、仮想プライベートネットワーク (VPN) にサインインしていて、ネットワーク設定が原因で共有ドライブをマウントできない場合にも発生することがあります。このような場合は、VPN をオフにし、Docker コンテナを再実行します。

エラー: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::<account-id>:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: *

このエラーは、Amazon ECR リポジトリにアクセスするための十分な権限がない場合、`aws ecr get-login-password` コマンドの実行時に表示されることがあります。詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR リポジトリポリシーの例」および「1 つの Amazon ECR リポジトリにアクセスする」を参照してください。

一般的な AWS IoT Greengrass トラブルシューティングのヘルプについては、「」を参照してくださいトラブルシューティング。

Docker コンテナ AWS IoT Greengrass でのデバッグ

Docker コンテナの問題をデバッグするには、Greengrass ランタイムログを維持するか、Docker コンテナにインタラクティブシェルをアタッチすることができます。

Docker コンテナの外部で Greengrass ランタイムログを永続化する

`/greengrass/ggc/var/log` ディレクトリをバインドマウントした後、 AWS IoT Greengrass Docker コンテナを実行できます。ログは、コンテナが終了した後または削除された後も保持されます。

Linux または macOS の場合

ホスト上で実行されている Greengrass Docker コンテナを停止してから、ターミナルで次のコマンドを実行します。これは Greengrass の `log` ディレクトリをバインドマウントして Docker イメージを起動します。

`/tmp` は、証明書と設定ファイルを解凍したパスに置き換えてください。

```
docker run --rm --init -it --name aws-iot-greengrass \
      --entrypoint /greengrass-entrypoint.sh \
      -v /tmp/certs:/greengrass/certs \
      -v /tmp/config:/greengrass/config \
      -v /tmp/log:/greengrass/ggc/var/log \
      -p 8883:8883 \
      216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

次に、ホストの `/tmp/log` でログを確認して、Greengrass が Docker コンテナ内で実行されている間に何が起こったのかを確認できます。

Windows の場合

ホスト上で実行されている Greengrass Docker コンテナを停止してから、コマンドプロンプトで次のコマンドを実行します。これは Greengrass の `log` ディレクトリをバインドマウントして Docker イメージを起動します。

```
cd C:\Users\%USERNAME%\Downloads
mkdir log
docker run --rm --init -it --name aws-iot-greengrass --entrypoint /greengrass-
entrypoint.sh -v c:/Users/%USERNAME%/Downloads/certs:/greengrass/certs -v c:/
Users/%USERNAME%/Downloads/config:/greengrass/config -v c:/Users/%USERNAME%/
Downloads/log:/greengrass/ggc/var/log -p 8883:8883 216483018798.dkr.ecr.us-
west-2.amazonaws.com/aws-iot-greengrass:latest
```

次に、ホストの `C:/Users/%USERNAME%/Downloads/log` でログを確認して、Greengrass が Docker コンテナ内で実行されている間に何が起こったのかを確認できます。

インタラクティブシェルを Docker コンテナにアタッチするには

実行中の AWS IoT Greengrass Docker コンテナにインタラクティブシェルをアタッチできます。これは、Greengrass Docker コンテナの状態を調査するのに役立ちます。

Linux または macOS の場合

Greengrass Docker コンテナの実行中に、別の端末で次のコマンドを実行してください。

```
docker exec -it $(docker ps -a -q -f "name=aws-iot-greengrass") /bin/bash
```

Windows の場合

Greengrass Docker コンテナの実行中に、別のコマンドプロンプトで次のコマンドを実行してください。

```
docker ps -a -q -f "name=aws-iot-greengrass"
```

gg-container-id を、前のコマンドから得られた `container_id` の結果に置き換えます。

```
docker exec -it gg-container-id /bin/bash
```

Lambda 関数とコネクタを使用してローカルリソースにアクセスする

この機能は AWS IoT Greengrass Core v1.3 以降で使用できます。

を使用すると AWS IoT Greengrass、 クラウドで AWS Lambda 関数を作成し、コネクタを設定し、ローカル実行のためにコアデバイスにデプロイできます。Linux を実行する Greengrass Core で、これらのローカルにデプロイされた Lambda 関数およびコネクタは、Greengrass Core デバイスに物理的に存在するローカルリソースにアクセスできます。例えば、Modbus または CANbus を介して接続されているデバイスと通信するには、Lambda 関数を有効にして、Core デバイスのシリアルポートにアクセスします。ローカルリソースへの安全なアクセスを設定するには、物理的なハードウェアおよび Greengrass コアデバイス OS のセキュリティを保証する必要があります。

ローカルリソースへのアクセスを開始するには、次のチュートリアルを参照してください。

- AWS コマンドラインインターフェイスを使用してローカルリソースアクセスを設定する方法
- を使用してローカルリソースアクセスを設定する方法 AWS Management Console

サポートされているリソースタイプ

ボリュームリソースとデバイスリソースの 2 種類のローカルリソースにアクセスできます。

ボリュームリソース

ルートファイルシステム上のファイルまたはディレクトリ (`/sys`、`/dev` または `/var` を除く)。具体的には次のとおりです。

- Greengrass Lambda 関数 (例えば、`/usr/lib/python2.x/site-packages/local`) を介して情報を読み書きするためのフォルダまたはファイル。
- ホストの /proc ファイルシステムのフォルダまたはファイル (`/proc/net` または `/proc/stat` など)。v1.6 以降でサポートされています。追加の要件については、the section called “/proc ディレクトリのボリュームリソース”を参照してください。

> **Tip**
>
> /var、/var/run、および /var/lib ディレクトリをボリュームリソースとして設定するには、最初にディレクトリを別のフォルダにマウントし、フォルダをボリュームリソースとして設定します。

ボリュームリソースを設定するときは、ソースパスと送信先パスを指定します。ソースパスはホスト上のリソースの絶対パスです。送信先パスは Lambda 名前空間環境内のリソースの絶対パスです。これは、Greengrass Lambda 関数またはコネクタが実行されるコンテナです。送信先パスの変更はホストファイルシステムのソースパスに反映されます。

> **Note**
>
> 送信先パスのファイルは Lambda 名前空間でのみ表示されます。通常の Linux 名前空間では表示できません。

デバイスリソース

/dev の下にあるファイル。/dev の下のキャラクターデバイスまたはブロックデバイスだけがデバイスリソースに使用できます。具体的には次のとおりです。

- シリアルポート (/dev/ttyS0 や /dev/ttyS1 など) を介して接続されたデバイスとの通信に使用されるシリアルポート。
- USB 周辺機器 (/dev/ttyUSB0 または /dev/bus/usb など) を接続するために使用される USB。
- GPIO (/dev/gpiomem など) を介してセンサーとアクチュエーターに使用される GPIO。
- オンボード GPU (/dev/nvidia0 など) を使用して機械学習を加速するために使用される GPU。
- 画像や動画のキャプチャに使用されるカメラ (/dev/video0 など)。

> **Note**
>
> /dev/shm は例外です。これは、ボリュームリソースとしてのみ設定できます。/dev/shm リソースには rw アクセス許可が付与されている必要があります。

AWS IoT Greengrass は、機械学習推論の実行に使用されるリソースタイプもサポートしています。詳細については、「機械学習の推論を実行する」を参照してください。

要件

ローカルリソースへのセキュアなアクセスの設定には、以下の要件が適用されます。

- AWS IoT Greengrass Core Software v1.3 以降を使用している必要があります。ホストの /proc ディレクトリにリソースを作成するには、v1.6 以降を使用する必要があります。
- ローカルリソース (必要なドライバーとライブラリのすべてを含む) は、Greengrass コアデバイスに正しくインストールされ、使用中も常にアクセス可能であることが必要です。
- 必要になるリソースの操作とリソースへのアクセスに root 権限は不要です。
- `read` または `read and write` アクセス許可のみが使用可能です。Lambda 関数は、リソースに対して特別な権限が必要なオペレーションを実行することはできません。
- Greengrass コアデバイスのオペレーティングシステム上のローカルリソースの完全パスを指定することが必要です。
- リソース名または ID の最大長は 128 文字で、パターン `[a-zA-Z0-9:_-]+` を使用する必要があります。

/proc ディレクトリのボリュームリソース

ホストの /proc ディレクトリにあるボリュームリソースには、以下の考慮事項が適用されます。

- AWS IoT Greengrass Core Software v1.6 以降を使用している必要があります。
- Lambda 関数には読み取り専用アクセスを許可できますが、読み取り/書き込みアクセスは許可できません。このレベルのアクセスは によって管理されます AWS IoT Greengrass。
- また、ファイルシステムに読み取りアクセスを可能にする OS グループ権限を付与することが必要になる場合があります。例えば、リソースディレクトリあるいはファイルに 660 ファイルアクセス許可があるとします。これは、このグループの所有者あるいはユーザーのみに読み取り (あるいは書き込み) アクセスがあることを意味します。この場合、このリソースに OS グループ所有者の権限を追加する必要があります。詳細については、「the section called “グループ所有者のファイルアクセス権限”」を参照してください。
- ホスト環境および Lambda 名前空間には、その両方に /proc ディレクトリがあるため、配置先パスを指定するときに名前の競合を回避するように注意します。例えば、/proc がソースパスの場合、/host-proc を配置先パスとして指定できます (または、「/proc」を除く任意のパス名)。

グループ所有者のファイルアクセス権限

AWS IoT Greengrass Lambda 関数プロセスは通常、 `ggc_user`および として実行されます`ggc_group`。ただし、次のように、ローカルリソース定義の Lambda 関数プロセスに追加のファイルアクセス許可を与えることができます。

- リソースを所有する Linux グループのアクセス権限を追加するには、`GroupOwnerSetting#AutoAddGroupOwner` パラメータまたは [Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのアクセス権限をファイルシステムに自動的に追加) コンソールオプションを使用します。
- 別の Linux グループのアクセス権限を追加するには、`GroupOwnerSetting#GroupOwner` パラメータまたは [Specify another system group to add file system permissions] (ファイルシステムの許可を追加するために、別のシステムグループを指定) コンソールオプションを使用します。`GroupOwnerSetting#AutoAddGroupOwner` が true の場合、`GroupOwner` 値は無視されます。

AWS IoT Greengrass Lambda 関数プロセスは、`ggc_user`、、`ggc_group`および Linux グループのすべてのファイルシステムアクセス許可を継承します (追加した場合) 。Lambda 関数がリソースにアクセスするためには、このリソースに要求されるアクセス許可が Lambda 関数プロセスにあることが必要です。必要に応じて、`chmod(1)` コマンドを使用して、リソースへのアクセス許可を変更できます。

関連情報

- Amazon Web Services 全般のリファレンス 内のリソースの Service Quotas

AWS コマンドラインインターフェイスを使用してローカルリソースアクセスを設定する方法

この機能は AWS IoT Greengrass Core v1.3 以降で使用できます。

ローカルリソースを使用するには、Greengrass Core デバイスにデプロイされたグループ定義にリソース定義を追加する必要があります。グループ定義には、Lambda 関数にローカルリソースに対するアクセス権限を付与する Lambda 関数定義も含める必要があります。要件と制約を含む詳細については、「Lambda 関数とコネクタを使用してローカルリソースにアクセスする」を参照してください。

このチュートリアルでは、ローカルリソースを作成し、 AWS Command Line Interface (CLI) を使用してローカルリソースへのアクセスを設定するプロセスについて説明します。このチュートリアルのステップを実行するには、「の開始方法 AWS IoT Greengrass」で説明するように、Greengrass グループをすでに作成している必要があります。

を使用するチュートリアルについては AWS Management Console、「」を参照してください を使用してローカルリソースアクセスを設定する方法 AWS Management Console。

ローカルリソースの作成

まず、`CreateResourceDefinition` コマンドを使用して、アクセス先のリソースを指定するリソース定義を作成します。この例では、2 つのリソースとして `TestDirectory` と `TestCamera` を作成します。

```
aws greengrass create-resource-definition  --cli-input-json '{
    "Name": "MyLocalVolumeResource",
    "InitialVersion": {
        "Resources": [
            {
                "Id": "data-volume",
                "Name": "TestDirectory",
                "ResourceDataContainer": {
                    "LocalVolumeResourceData": {
                        "SourcePath": "/src/LRAtest",
                        "DestinationPath": "/dest/LRAtest",
                        "GroupOwnerSetting": {
                            "AutoAddGroupOwner": true,
                            "GroupOwner": ""
                        }
                    }
                }
            },
            {
                "Id": "data-device",
                "Name": "TestCamera",
                "ResourceDataContainer": {
                    "LocalDeviceResourceData": {
                        "SourcePath": "/dev/video0",
                        "GroupOwnerSetting": {
                            "AutoAddGroupOwner": true,
                            "GroupOwner": ""
```

```
                        }
                    }
                }
            }
        ]
    }
}'
```

Resources: Greengrass グループの `Resource` オブジェクトのリスト。1 つの Greengrass グループには、最大 50 個のリソースを含めることができます。

Resource#Id: リソースの一意の識別子。この ID は、Lambda 関数の設定でリソースを参照するために使用されます。最大長: 128 文字。パターン: [a-zA-Z0-9:_-]+。

Resource#Name: リソースの論理名。このリソース名は、Greengrass コンソールに表示されます。最大長: 128 文字。パターン: [a-zA-Z0-9:_-]+。

LocalDeviceResourceData#SourcePath: デバイスリソースのローカル絶対パス。デバイスリソースのソースパスは、`/dev` の文字デバイスまたはブロックデバイスのみを参照できます。

LocalVolumeResourceData#SourcePath: Greengrass コアデバイス上のボリュームリソースのローカル絶対パス。この場所は、関数が実行されるコンテナの外側です。ボリュームリソースタイプのソースパスは、`/sys` で始めることはできません。

LocalVolumeResourceData#DestinationPath: Lambda 環境内のボリュームリソースの絶対パス。この場所は、関数が実行されるコンテナの内側です。

GroupOwnerSetting: Lambda プロセスの追加のグループ権限を設定できます。このフィールドはオプションです。詳細については、「グループ所有者のファイルアクセス権限」を参照してください。

GroupOwnerSetting#AutoAddGroupOwner: true の場合は、リソースの指定した Linux OS グループ所有者の権限が Lambda プロセスに Greengrass によって自動的に追加されます。つまり、Lambda プロセスには、追加された Linux グループのファイルアクセス権限があります。

GroupOwnerSetting#GroupOwner: Lambda プロセスに権限が追加される Linux OS グループの名前を指定します。このフィールドはオプションです。

`CreateResourceDefinition` によって、リソース定義バージョン ARN が返されます。この ARN はグループ定義の更新時に使用する必要があります。

```
{
```

```
    "LatestVersionArn": "arn:aws:greengrass:us-west-2:012345678901:/greengrass/
definition/resources/ab14d0b5-116e-4951-a322-9cde24a30373/versions/a4d9b882-
d025-4760-9cfe-9d4fada5390d",
    "Name": "MyLocalVolumeResource",
    "LastUpdatedTimestamp": "2017-11-15T01:18:42.153Z",
    "LatestVersion": "a4d9b882-d025-4760-9cfe-9d4fada5390d",
    "CreationTimestamp": "2017-11-15T01:18:42.153Z",
    "Id": "ab14d0b5-116e-4951-a322-9cde24a30373",
    "Arn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/resources/
ab14d0b5-116e-4951-a322-9cde24a30373"
}
```

Greengrass 関数を作成する

リソースが作成されたら、`CreateFunctionDefinition` コマンドを使用して Greengrass 関数を作成し、その関数にリソースへのアクセス許可を付与します。

```
aws greengrass create-function-definition --cli-input-json '{
    "Name": "MyFunctionDefinition",
    "InitialVersion": {
        "Functions": [
            {
                "Id": "greengrassLraTest",
                "FunctionArn": "arn:aws:lambda:us-
west-2:012345678901:function:lraTest:1",
                "FunctionConfiguration": {
                    "Pinned": false,
                    "MemorySize": 16384,
                    "Timeout": 30,
                    "Environment": {
                        "ResourceAccessPolicies": [
                            {
                                "ResourceId": "data-volume",
                                "Permission": "rw"
                            },
                            {
                                "ResourceId": "data-device",
                                "Permission": "ro"
                            }
                        ],
                        "AccessSysfs": true
                    }
                }
```

```
            }
        ]
    }
}'
```

ResourceAccessPolicies: Lambda 関数にリソースへのアクセスを許可する `resourceId` と `permission` が含まれます。Lambda 関数は最大 20 のリソースにアクセスできます。

ResourceAccessPolicy#Permission: Lambda 関数のリソースに対するアクセス権限を指定します。使用可能なオプションは、`rw` (読み取り/書き込み) または `ro` (読み取り専用) です。

AccessSysfs: true の場合、Lambda プロセスには Greengrass コアデバイスの `/sys` フォルダに対する読み取りアクセス権限が付与されます。これは、Greengrass Lambda 関数が `/sys` からデバイス情報を読み取る必要がある場合に使用されます。

ここでも、`CreateFunctionDefinition` によって関数定義バージョン ARN が返されます。この ARN はグループ定義バージョンで使用する必要があります。

```
{
    "LatestVersionArn": "arn:aws:greengrass:us-west-2:012345678901:/greengrass/
definition/functions/3c9b1685-634f-4592-8dfd-7ae1183c28ad/versions/37f0d50e-ef50-4faf-
b125-ade8ed12336e",
    "Name": "MyFunctionDefinition",
    "LastUpdatedTimestamp": "2017-11-22T02:28:02.325Z",
    "LatestVersion": "37f0d50e-ef50-4faf-b125-ade8ed12336e",
    "CreationTimestamp": "2017-11-22T02:28:02.325Z",
    "Id": "3c9b1685-634f-4592-8dfd-7ae1183c28ad",
    "Arn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/
functions/3c9b1685-634f-4592-8dfd-7ae1183c28ad"
}
```

グループに Lambda 関数を追加する

最後に、`CreateGroupVersion` を使用して関数をグループに追加します。例:

```
aws greengrass create-group-version --group-id "b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5" \
--resource-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/
greengrass/definition/resources/db6bf40b-29d3-4c4e-9574-21ab7d74316c/versions/31d0010f-
e19a-4c4c-8098-68b79906fb87" \
--core-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/
greengrass/definition/cores/adbf3475-f6f3-48e1-84d6-502f02729067/
versions/297c419a-9deb-46dd-8ccc-341fc670138b" \
```

```
--function-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/
greengrass/definition/functions/d1123830-da38-4c4c-a4b7-e92eec7b6d3e/versions/a2e90400-
caae-4ffd-b23a-db1892a33c78" \
--subscription-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/
greengrass/definition/subscriptions/7a8ef3d8-1de3-426c-9554-5b55a32fbcb6/
versions/470c858c-7eb3-4abd-9d48-230236bfbf6a"
```

> **Note**
>
> このコマンドで使用するグループ ID を取得する方法については、「the section called “グループ ID の取得”」を参照してください。

新しいグループのバージョンが返されます。

```
{
    "Arn": "arn:aws:greengrass:us-west-2:012345678901:/greengrass/groups/
b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5/versions/291917fb-ec54-4895-823e-27b52da25481",
    "Version": "291917fb-ec54-4895-823e-27b52da25481",
    "CreationTimestamp": "2017-11-22T01:47:22.487Z",
    "Id": "b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5"
}
```

Greengrass グループには、TestDirectory と TestCamera の 2 つのリソースにアクセスできる lraTest Lambda 関数が含まれています。

Python で書かれたこのサンプル Lambda 関数 `lraTest.py` は、ローカルボリュームリソースを書き込みます。

```
# Demonstrates a simple use case of local resource access.
# This Lambda function writes a file test to a volume mounted inside
# the Lambda environment under destLRAtest. Then it reads the file and
# publishes the content to the AWS IoT LRAtest topic.

import sys
import greengrasssdk
import platform
import os
import logging

# Setup logging to stdout
logger = logging.getLogger(__name__)
```

```
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

# Create a Greengrass Core SDK client.
client = greengrasssdk.client('iot-data')
volumePath = '/dest/LRAtest'

def function_handler(event, context):
    try:
        client.publish(topic='LRA/test', payload='Sent from AWS IoT Greengrass Core.')
        volumeInfo = os.stat(volumePath)
        client.publish(topic='LRA/test', payload=str(volumeInfo))
        with open(volumePath + '/test', 'a') as output:
            output.write('Successfully write to a file.')
        with open(volumePath + '/test', 'r') as myfile:
            data = myfile.read()
        client.publish(topic='LRA/test', payload=data)
    except Exception as e:
        logger.error('Failed to publish message: ' + repr(e))
    return
```

以下のコマンドは Greengrass API に用意されており、リソース定義とリソース定義バージョンの作成と管理に使用します。

- CreateResourceDefinition
- CreateResourceDefinitionVersion
- DeleteResourceDefinition
- GetResourceDefinition
- GetResourceDefinitionVersion
- ListResourceDefinitions
- ListResourceDefinitionVersions
- UpdateResourceDefinition

トラブルシューティング

- Q: Greengrass グループのデプロイに失敗するのはなぜですか?

```
group config is invalid:
    ggc_user or [ggc_group root tty] don't have ro permission on the file: /dev/tty0
```

A: このエラーは、指定したリソースに対するアクセス権限が Lambda プロセスにないことを示します。解決策は、Lambda がアクセスできるようにリソースのファイルアクセス権限を変更することです。(詳細については、「グループ所有者のファイルアクセス権限」を参照してください)。

- Q: ボリュームリソースとして `/var/run` を設定すると、runtime.log にエラーメッセージが記録され、Lambda 関数が開始されないのはなぜですか?

```
[ERROR]-container_process.go:39,Runtime execution error: unable to start lambda
 container.
container_linux.go:259: starting container process caused "process_linux.go:345:
container init caused \"rootfs_linux.go:62: mounting \\\"/var/run\\\" to rootfs \\\"/
greengrass/ggc/packages/1.3.0/rootfs_sys\\\" at \\\"/greengrass/ggc/packages/1.3.0/
rootfs_sys/run\\\"
caused \\\"invalid argument\\\"\""
```

A: AWS IoT Greengrass core は現在、ボリュームリソース`/var/lib`としての `/var`、`/var/run`、および の設定をサポートしていません。1 つの回避策は、最初に `/var`、`/var/run`、`/var/lib` を別のフォルダーにマウントし、そのフォルダーをボリュームリソースとして設定することです。

- Q: `/dev/shm` を読み取り専用アクセス権限のあるボリュームリソースとして設定すると、runtime.log にエラーが記録されて、Lambda 関数が開始されないのはなぜですか?

```
[ERROR]-container_process.go:39,Runtime execution error: unable to start lambda
 container.
container_linux.go:259: starting container process caused "process_linux.go:345:
container init caused \"rootfs_linux.go:62: mounting \\\"/dev/shm\\\" to rootfs \\\"/
greengrass/ggc/packages/1.3.0/rootfs_sys\\\" at \\\"/greengrass/ggc/packages/1.3.0/
rootfs_sys/dev/shm\\\"
caused \\\"operation not permitted\\\"\"”
```

A: `/dev/shm` は読み取り/書き込みとしてのみ設定できます。リソースに対するアクセス許可を `rw` に変更して、問題を解決します。

を使用してローカルリソースアクセスを設定する方法 AWS Management Console

この機能は AWS IoT Greengrass Core v1.3 以降で使用できます。

ホストの Greengrass コアデバイス上のローカルリソースに安全にアクセスするように Lambda 関数を設定することができます。ローカルリソースとは、物理的にホスト上にあるバスや周辺機器、またはホスト OS 上のファイルシステムボリュームを指します。要件と制約を含む詳細については、「Lambda 関数とコネクタを使用してローカルリソースにアクセスする」を参照してください。

このチュートリアルでは、 を使用して AWS Management Console 、 AWS IoT Greengrass コアデバイスに存在するローカルリソースへのアクセスを設定する方法について説明します。これには、次のような手順が含まれています。

1. Lambda 関数デプロイパッケージを作成する
2. Lambda 関数を作成して発行する
3. グループに Lambda 関数を追加する
4. グループにローカルリソースを追加する
5. サブスクリプションをグループに追加する
6. グループをデプロイする

を使用するチュートリアルについては AWS Command Line Interface、「」を参照してください AWS コマンドラインインターフェイスを使用してローカルリソースアクセスを設定する方法。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.3 以降)。Greengrass グループまたはコアを作成するには、「の開始方法 AWS IoT Greengrass」を参照してください。
- Greengrass コアデバイス上にある以下のディレクトリでは、次のとおりです。
  - /src/LRAtest
  - /dest/LRAtest

  これらのディレクトリの所有者グループは、そのディレクトリへの読み取りと書き込みアクセスが許可されている必要があります。例えば、次のコマンドを使用して、以下のアクセスを許可することができます。

  ```
  sudo chmod 0775 /src/LRAtest
  ```

ステップ 1: Lambda 関数デプロイパッケージを作成する

このステップでは、Lambda 関数のデプロイパッケージを作成します。これは、関数のコードと依存関係を含む ZIP ファイルです。また、 AWS IoT Greengrass Core SDK をダウンロードして、依存関係としてパッケージに を含めます。

1. コンピュータで、以下の Python スクリプトを `lraTest.py` という名前のローカルファイルにコピーします。これは、Lambda 関数のアプリケーションロジックです。

```
# Demonstrates a simple use case of local resource access.
# This Lambda function writes a file test to a volume mounted inside
# the Lambda environment under destLRAtest. Then it reads the file and
# publishes the content to the AWS IoT LRAtest topic.

import sys
import greengrasssdk
import platform
import os
import logging

# Setup logging to stdout
logger = logging.getLogger(__name__)
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

# Create a Greengrass Core SDK client.
client = greengrasssdk.client('iot-data')
volumePath = '/dest/LRAtest'

def function_handler(event, context):
    try:
        client.publish(topic='LRA/test', payload='Sent from AWS IoT Greengrass
 Core.')
        volumeInfo = os.stat(volumePath)
        client.publish(topic='LRA/test', payload=str(volumeInfo))
        with open(volumePath + '/test', 'a') as output:
            output.write('Successfully write to a file.')
        with open(volumePath + '/test', 'r') as myfile:
            data = myfile.read()
        client.publish(topic='LRA/test', payload=data)
    except Exception as e:
        logger.error('Failed to publish message: ' + repr(e))
    return
```

2. AWS IoT Greengrass Core SDK ダウンロードページから、 AWS IoT Greengrass Core SDK for Python をコンピュータにダウンロードします。
3. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
4. 以下の項目を `lraTestLambda.zip` という名前のファイルに圧縮します。
   - `lraTest.py`。アプリケーションロジック。
   - `greengrasssdk`。すべての Python Lambda 関数に必要なライブラリ。

   `lraTestLambda.zip` ファイルは Lambda 関数デプロイパッケージです。これで、Lambda 関数を作成して、デプロイパッケージをアップロードする準備ができました。

ステップ 2: Lambda 関数を作成して発行する

このステップでは、 AWS Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージを使用するように設定します。次に、関数のバージョンを公開し、エイリアスを作成します。

最初に Lambda 関数を作成します。

1. で AWS Management Console、サービスを選択し、コンソールを開きます AWS Lambda 。
2. 関数を選択します。
3. [Create function] (関数の作成) を選択し、[Author from scratch] (一から作成) を選択します。
4. [Basic information] セクションで、以下の値を指定します。
   a. [関数名] に「**`TestLRA`**」と入力します。
   b. [ランタイム] で [Python 3.7] を選択します。
   c. [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。
5. [Create function (関数の作成)] を選択します。

6. Lambda 関数デプロイパッケージをアップロードし、ハンドラを登録します。

   a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

   b. [アップロード] を選択し、`lraTestLambda.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

   c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

      - [ランタイム] で [Python 3.7] を選択します。
      - [ハンドラ] で、[lraTest.function_handler] と入力します。

   d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コン

ソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

次に、Lambda 関数の最初のバージョンを発行します。その後、バージョンのエイリアスを作成します。

Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

7. [アクション] メニューから、[新しいバージョンを発行] を選択します。
8. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。
9. [TestLRA:1] 設定ページで、[Actions] から、[Create alias] を選択します。
10. [エイリアスの作成] ページの [名前] に、「**`test`**」と入力します。[バージョン] に、[1] と選択します。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

11. [Create] (作成) を選択します。

Greengrass グループに Lambda 関数を追加できるようになりました。

ステップ 3: Lambda 関数を Greengrass グループに追加する

このステップでは、この関数をグループに追加し、その関数のライフサイクルを設定します。

まず、Lambda 関数を Greengrass グループに追加します。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. Lambda 関数を追加する Greengrass グループを選択します。
3. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
4. [My Lambda functions] (自分の Lambda 関数) セクションで、 [Add] (追加) を選択します。
5. [Add Lambda function] (Lambda 関数の追加) ページで、 [Lambda function](Lambda 関数)を選択します。**`TestLRA`** を選択します。
6. [Lambda 関数のバージョン] を選択します。
7. [Lambda function configuration] (Lambda 関数の設定) セクションで、[System user and group] (システムユーザーとグループ)、[Lambda function containerization] (Lambda 関数のコンテナ化) の順に選択します。

次に、Lambda 関数のライフサイクルを設定します。

8. [タイムアウト] で、[30 秒] を選択します。

> **Important**
>
> この手順で説明されているように、ローカルリソースを使用する Lambda 関数は、Greengrass コンテナ内で実行する必要があります。それ以外の場合は、関数をデプロイしようとすると、デプロイが失敗します。詳細については、「コンテナ化」を参照してください。

9. ページの下部で、[Add Lambda function] (Lambda 関数の追加) を選択します。

ステップ 4: Greengrass グループにローカルリソースを追加する

このステップでは、Greengrass グループにローカルボリュームリソースを追加し、リソースへの読み取りと書き込みアクセスを許可します。ローカルリソースにはグループレベルのスコープがあります。グループ内の任意の Lambda 関数に、リソースにアクセスするためのアクセス権限を付与できます。

1. グループ設定ページで、[Resources] (リソース) タブを選択します。
2. [Local resources] (ローカルリソース) セクションから [Add] (追加)を選択します。
3. [Add a local resource] (ローカルリソースの追加) ページで、次の値を使用します。

   a. [リソース名] に **`testDirectory`** と入力します。

   b. [リソースタイプ] で、[ボリューム] を選択します。

   c. [Local device path] (ローカルデバイスパス) には、「**`/src/LRAtest`**」と入力します。このパスはホスト OS 上に存在している必要があります。

      ローカルデバイスは、コアデバイスのファイルシステム上のリソースのローカル絶対パスです。この場所は、関数が実行されるコンテナの外側です。このパスの先頭を `/sys` にすることはできません。

   d. [送信先パス] に **`/dest/LRAtest`** と入力します。このパスはホスト OS 上に存在している必要があります。

送信先パスは、Lambda 名前空間にあるリソースの絶対パスです。この場所は、関数が実行されるコンテナの内側です。

e. [System group owner and file access permission] (システムグループ所有者のファイルアクセス許可) で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有する システムグループのファイルシステム権限を自動的に追加する) を選択します。

   [System group owner and file access permission] (システムグループ所有者のファイルアクセス権限) オプションを使用すると、Lambda プロセスに追加のファイルアクセス権限をグラントできます。詳細については、「グループ所有者のファイルアクセス権限」を参照してください。

4. [リソースを追加] を選択します。[リソース] ページで、新しい testDirectory リソースが表示されます。

ステップ 5: サブスクリプションを Greengrass グループに追加する

このステップでは、2 つのサブスクリプションを Greengrass グループに追加します。これらのサブスクリプションにより、Lambda 関数と 間の双方向通信が可能になります AWS IoT。

まず、 AWS IoTにメッセージを送信する Lambda 関数のサブスクリプションを作成します。

1. グループ設定ページで、[Subscriptions] (サブスクリプション) タブを選択します。
2. [追加] を選択します。
3. [Create a subscription] (サブスクリプションの作成) ページで、ソースおよびターゲットを次のように設定します。

   a. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数)、[TestLRA] の順に選択します。

   b. [Target type] (ターゲットタイプ)で、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。

   c. [Topic filter] (トピックのフィルター) で、**`LRA/test`** と入力し、 [Create subscription] (サブスクリプションの作成) を選択します。

4. [Subscriptions] ページに新しいサブスクリプションが表示されます。

次に、 関数を呼び出すサブスクリプションを設定します AWS IoT。

5. [Subscriptions] ページで [Add Subscription] を選択します。
6. [ソースとターゲットの選択] ページで、ソースおよびターゲットを次のように設定します。
   a. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数)、[IoT Cloud] (IoT クラウド) の順に選択します。
   b. [Target type] (ターゲットタイプ) で、[Service] (サービス)、[TestLRA]の順に選択します。
   c. [Next (次へ)] を選択します。
7. [トピックでデータをフィルタリングする] ページの [トピックフィルター] に「**`invoke/LRAFunction`**」と入力し、[Next (次へ)] を選択します。
8. [Finish] を選択してください。[Subscriptions] ページに両方のサブスクリプションが表示されます。

ステップ 6: AWS IoT Greengrass グループをデプロイする

このステップでは、グループ定義の現在のバージョンをデプロイします。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。
   a. デーモンが実行中かどうかを確認するには、以下を実行します。

      ```
      ps aux | grep -E 'greengrass.*daemon'
      ```

      出力に `root` で実行中の `/greengrass/ggc/packages/1.11.6/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

      > **Note**
      > パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

   b. デーモンを開始するには、以下を実行します。

      ```
      cd /greengrass/ggc/core/
      sudo ./greengrassd start
      ```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。

> **Note**
> コンテナ化を使用せずに Lambda 関数を実行し、アタッチ済みのローカルリソースにアクセスしようとすると、デプロイは失敗します。

3. プロンプトが表示されたら、[Lambda function] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) から、[IP detector] (IP ディテクター)、[Edit] (編集)、[Automatically detect] (自動的に検出) の順に選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイステータスは、[Completed] (完了) になります。

   トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

ローカルリソースアクセスのテスト

これで、ローカルリソースアクセスが正しく設定されているかどうかを確認できます。テストするには、`LRA/test` トピックにサブスクライブし、`invoke/LRAFunction` トピックに発行します。Lambda 関数が想定ペイロードを に送信すると、テストは成功です AWS IoT。

1. AWS IoT コンソールナビゲーションメニューから、テストで MQTT テストクライアントを選択します。

2. [Subscribe to a topic] (トピックへのサブスクライブ) で、[Topic filter] (トピックのフィルター) に、**`LRA/test`**と入力します。
3. [Additional information] (追加情報) から、[MQTT payload display] (MQTT ペイロード表示) で、[Display payloads as strings] (文字列としてペイロードを表示) を選択します。
4. [サブスクライブ] を選択します。Lambda 関数は LRA/test トピックに発行します。

5. [Publish to a topic] (トピックに発行) から、[Topic name] (トピック名) で、**`invoke/LRAFunction`**と入力し、[Publish] (発行) を選択して、Lambda 関数を呼び出します。ページに関数の 3 つのメッセージペイロードが表示されている場合、テストは成功です。

Lambda 関数によって作成されたテストファイルは、Greengrass コアデバイスの `/src/LRAtest` ディレクトリにあります。Lambda 関数によって、`/dest/LRAtest` ディレクトリのファイルに書き込まれ、そのファイルは Lambda 名前空間でのみ表示されます。通常の Linux 名前空間には表示されません。送信先パスの変更は、ファイルシステムのソースパスに反映されます。

トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

機械学習の推論を実行する

この機能は AWS IoT Greengrass Core v1.6 以降で使用できます。

を使用すると AWS IoT Greengrass、クラウドでトレーニングされたモデルを使用して、ローカルに生成されたデータのエッジで機械学習 (ML) 推論を実行できます。ローカル推論の実行により低レイテンシーとコスト節約のメリットを得ながら、モデルのトレーニングと複雑な処理にクラウドコンピューティングの処理能力を活用できます。

ローカル推論の実行を開始するには、「the section called “機械学習推論を設定する方法”」を参照してください。

ML AWS IoT Greengrass 推論の仕組み

推論モデルは、任意の場所でトレーニングし、Greengrass グループに機械学習リソースとしてローカルにデプロイした後、Greengrass Lambda 関数からアクセスできるようになります。たとえば、SageMaker AI で深層学習モデルを構築およびトレーニングし、Greengrass コアにデプロイできます。その後、Lambda 関数はローカルモデルを使用して、接続されたデバイスで推論を実行し、新しいトレーニングデータをクラウドに送り返すことができます。

次の図は、ML AWS IoT Greengrass 推論ワークフローを示しています。

AWS IoT Greengrass ML 推論は、以下を含む ML ワークフローの各ステップを簡素化します。

- ML フレームワークのプロトタイプを構築し、デプロイする。
- クラウドトレーニング済みモデルにアクセスし、それらのモデルを Greengrass コアデバイスにデプロイする。
- ローカルリソースであるハードウェアアクセラレーター (GPU や FPGA など) にアクセスできる推論アプリケーションを作成する。

機械学習リソース

機械学習リソースは、 AWS IoT Greengrass コアにデプロイされるクラウドトレーニングされた推論モデルを表します。機械学習リソースをデプロイするには、まず Greengrass グループに追加し、グループ内の Lambda 関数がそれらのリソースにアクセスできる方法を定義します。グループデプロイ中、 はクラウドからソースモデルパッケージ AWS IoT Greengrass を取得し、Lambda ランタイム名前空間内のディレクトリに抽出します。その後、Greengrass Lambda 関数によって、ローカルにデプロイしたモデルを使用して推論が実行されるようになります。

ローカルにデプロイしたモデルを更新するには、まず機械学習リソースに対応するソースモデル (クラウド内) を更新してから、そのグループをデプロイします。デプロイ中、 AWS IoT Greengrass によってソースの変更が確認されます。変更が検出されると、 はローカルモデル AWS IoT Greengrass を更新します。

サポートされているモデルソース

AWS IoT Greengrass は、機械学習リソースの SageMaker AI および Amazon S3 モデルソースをサポートしています。

モデルソースには、以下の要件が適用されます。

- SageMaker AI および Amazon S3 モデルソースを保存する S3 バケットは、SSE-C を使用して暗号化しないでください。 Amazon S3 サーバー側の暗号化を使用するバケットの場合、 AWS IoT Greengrass ML 推論は現在 SSE-S3 または SSE-KMS 暗号化オプションのみをサポートしています。サーバー側の暗号化の詳細については、「Amazon Simple Storage Service ユーザーガイド」の「サーバー側の暗号化を使用したデータ保護」を参照してください。
- SageMaker AI および Amazon S3 Amazon S3 バケットの名前にピリオド () を含めることはできません.。 SageMaker 詳細については、「Amazon Simple Storage Service ユーザーガイド」のバケット命名規則で、SSL で仮想ホスト型バケットの使用に関するルールを参照してください。

- サービスレベルの AWS リージョン サポートは、 AWS IoT Greengrassと SageMaker AI の両方で利用できる必要があります。現在、 AWS IoT Greengrass は次のリージョンで SageMaker AI モデルをサポートしています。
  - 米国東部(オハイオ)
  - 米国東部 (バージニア北部)
  - 米国西部 (オレゴン)
  - アジアパシフィック (ムンバイ)
  - アジアパシフィック (ソウル)
  - アジアパシフィック (シンガポール)
  - アジアパシフィック (シドニー)
  - アジアパシフィック (東京)
  - 欧州 (フランクフルト)
  - 欧州 (アイルランド)
  - 欧州 (ロンドン)
- AWS IoT Greengrass 次のセクションで説明するように、 にはモデルソースに対する`read`アクセス許可が必要です。

SageMaker AI

AWS IoT Greengrass は、SageMaker AI トレーニングジョブとして保存されるモデルをサポートします。SageMaker AI は、組み込みアルゴリズムまたはカスタムアルゴリズムを使用してモデルを構築およびトレーニングするために使用できるフルマネージド ML サービスです。詳細については、SageMaker AI デベロッパーガイド」の「SageMaker AI とは」を参照してください。SageMaker

という名前のバケットを作成して SageMaker AI 環境を設定した場合`sagemaker`、 AWS IoT Greengrass には SageMaker AI トレーニングジョブにアクセスするための十分なアクセス許可があります。`AWSGreengrassResourceAccessRolePolicy` 管理ポリシーは、名前に文字列 `sagemaker` が含まれるバケットへのアクセスを許可します。このポリシーは Greengrass サービスのロールにアタッチされます。

それ以外の場合は、トレーニングジョブが保存されているバケットにアクセス許可を付与 AWS IoT Greengrass `read`する必要があります。そのためには、サービスロールに以下のインラインポリシーを埋め込みます。複数のバケット ARN を含めることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
            ]
        }
    ]
}
```

Amazon S3

AWS IoT Greengrass は、Amazon S3 に `tar.gz`または `.zip` ファイルとして保存されているモデルをサポートします。

AWS IoT Greengrass が Amazon S3 バケットに保存されているモデルにアクセスできるようにするには、次のいずれかを実行してバケットにアクセスするアクセス AWS IoT Greengrass `read`許可を付与する必要があります。

- 名前に `greengrass` が含まれるバケットにモデルを保存します。

  `AWSGreengrassResourceAccessRolePolicy` 管理ポリシーは、名前に文字列 `greengrass` が含まれるバケットへのアクセスを許可します。このポリシーは Greengrass サービスのロールにアタッチされます。

- Greengrass サービスロールにインラインポリシーを埋め込みます。

  バケット名に `greengrass` が含まれない場合は、サービスロールに以下のインラインポリシーを追加します。複数のバケット ARN を含めることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
            ]
        }
    ]
}
```

詳細については、「IAM ユーザーガイド」の「インラインポリシーの埋め込み」を参照してください。

要件

機械学習リソースの作成と使用には、以下の要件が適用されます。

- AWS IoT Greengrass Core v1.6 以降を使用している必要があります。
- ユーザー定義 Lambda 関数は、リソースに対して `read` または `read and write` オペレーションを実行できます。他のオペレーションに対するアクセス許可はありません。関連する Lambda 関数のコンテナ化モードによって、アクセス権限の設定方法が決まります。詳細については、「the section called “機械学習リソースにアクセスする”」を参照してください。
- コアデバイスのオペレーティングシステム上のリソースの完全パスを指定することが必要です。
- リソース名または ID の最大長は 128 文字で、パターン `[a-zA-Z0-9:_-]+` を使用する必要があります。

ML 推論用のランタイムとライブラリ

では、次の ML ランタイムとライブラリを使用できます AWS IoT Greengrass。

- Amazon SageMaker Neo Deep Learning ランタイム
- Apache MXNet
- TensorFlow

これらのランタイムとライブラリは、NVIDIA Jetson TX2、Intel Atom、および Raspberry Pi プラットフォームにインストールできます。ダウンロード情報については、「the section called “サポート

されている機械学習ランタイムおよびライブラリ”」を参照してください。コアデバイスに直接インストールできます。

互換性と制限については、必ず以下の情報を参照してください。

SageMaker AI Neo 深層学習ランタイム

SageMaker AI Neo 深層学習ランタイムを使用して、 AWS IoT Greengrass デバイスで最適化された機械学習モデルで推論を実行できます。これらのモデルは、SageMaker AI Neo 深層学習コンパイラを使用して最適化され、機械学習の推論予測速度が向上します。SageMaker AI でのモデル最適化の詳細については、SageMaker AI Neo ドキュメントを参照してください。

> **Note**
>
> 現在、特定のアマゾン ウェブ サービスリージョンでのみ Neo 深層学習コンパイラを使用して機械学習モデルを最適化できます。ただし、 AWS リージョン AWS IoT Greengrass コアがサポートされている各 で最適化されたモデルで Neo 深層学習ランタイムを使用できます。詳細については、「最適化された機械学習推論を設定する方法」を参照してください。

MXNet のバージョニング

Apache MXNet は現在、下位互換性を保証していないため、フレームワークの新しいバージョンを使用してトレーニングするモデルは、フレームワークの以前のバージョンでは正しく動作しないことがあります。モデルトレーニング段階とモデル提供段階との間で競合を回避し、一貫したエンドツーエンドエクスペリエンスを実現するには、両方の段階で同じバージョンの MXNet フレームワークを使用します。

Raspberry Pi の MXNet

ローカル MXNet モデルにアクセスする Greengrass Lambda 関数では、以下の環境変数が設定される必要があります。

```
MXNET_ENGINE_TYPE=NativeEngine
```

関数コードで環境変数を設定することも、関数のグループ固有設定に追加することもできます。構成設定として環境変数を追加する例については、このステップを参照してください。

> **Note**
>
> サードパーティのコード例を実行するなどの MXNet フレームワークの一般使用に対しては、環境変数を Raspberry Pi で設定する必要があります。

Raspberry Pi での TensorFlow モデルの制限

推論結果を向上させるための推奨事項は、Raspberry Pi プラットフォームで TensorFlow 32-bit Arm ライブラリを使用したテストに基づいています。これらの推奨事項は、上級ユーザーのみを対象としており、いかなる種類の保証もありません。

- チェックポイント形式を使用してトレーニングされたモデルは、提供する前にプロトコルバッファ形式で "圧縮" する必要があります。例については、「TensorFlow-Slim イメージ分類モデルライブラリ」を参照してください。
- TF-Estimator および TF-Slim ライブラリはトレーニングコードまたは推論コードで使用しないでください。代わりに、以下の例に示している .pb ファイルモデルロードパターンを使用してください。

```
graph = tf.Graph()
graph_def = tf.GraphDef()
graph_def.ParseFromString(pb_file.read())
with graph.as_default():
  tf.import_graph_def(graph_def)
```

> **Note**
>
> TensorFlow でサポートされているプラットフォームの詳細については、TensorFlow のドキュメントの「TensorFlow のインストール」を参照してください。

Lambda 関数から機械学習リソースにアクセスする

ユーザー定義の Lambda 関数は、機械学習リソースにアクセスして、 AWS IoT Greengrass コアでローカル推論を実行できます。機械学習リソースは、トレーニングを受けたモデルと、コアデバイスにダウンロードされるその他のアーティファクトで構成されます。

Lambda 関数がコアの機械学習リソースにアクセスできるようにするには、リソースを Lambda 関数にアタッチし、アクセス権限を定義する必要があります。その際の手順は、関連付けられた (またはアタッチされた) Lambda 関数のコンテナ化モードによって決まります。

機械学習リソースのアクセス権限

AWS IoT Greengrass Core v1.10.0 以降では、機械学習リソースのリソース所有者を定義できます。リソース所有者は、 がリソースアーティファクトのダウンロード AWS IoT Greengrass に使用する OS グループとアクセス許可を表します。リソース所有者が定義されていない場合、ダウンロードされたリソースアーティファクトはルートにのみアクセスできます。

- コンテナ化されていない Lambda 関数が機械学習リソースにアクセスする場合、コンテナからのアクセス権限制御がないため、リソースの所有者を定義する必要があります。コンテナ化されていない Lambda 関数は、リソース所有者のアクセス権限を継承し、それらを使用してリソースにアクセスできます。

- コンテナ化された Lambda 関数だけがリソースにアクセスする場合は、リソース所有者を定義するのではなく、関数レベルのアクセス権限を使用することをお勧めします。

リソース所有者のプロパティ

リソース所有者は、グループ所有者とグループ所有者の権限を指定します。

グループの所有者。コアデバイス上の既存の Linux OS グループのグループ ID (GID)。グループの権限が Lambda プロセスに追加されます。具体的には、Lambda 関数の補足グループ ID に GID が追加されます。

Greengrass グループの Lambda 関数が、機械学習リソースのリソース所有者と同じ OS グループとして実行されるように設定されている場合は、そのリソースを Lambda 関数にアタッチする必要があります。それ以外の場合、この設定は Lambda 関数が AWS IoT Greengrass 認可なしでリソースにアクセスするために使用できる暗黙的なアクセス許可を付与するため、デプロイは失敗します。Lambda 関数がルート (UID=0) として実行されている場合、デプロイ検証チェックはスキップされます。

Greengrass Core の他のリソース、Lambda 関数、ファイルで使用されていない OS グループを使用することをお勧めします。共有 OS グループを使用すると、アタッチされた Lambda 関数が

必要以上に多くのアクセス権限を付与できます。共有 OS グループを使用する場合は、アタッチされた Lambda 関数も、その共有 OS グループを使用するすべての機械学習リソースにアタッチする必要があります。それ以外の場合、デプロイは失敗します。

グループ所有者の権限。Lambda プロセスに追加する読み取り専用権限、または読み取り/書き込み権限。

コンテナ化されていない Lambda 関数は、リソースに対するこれらのアクセス権限を継承する必要があります。コンテナ化された Lambda 関数は、これらのリソースレベルのアクセス権限を継承するか、関数レベルのアクセス権限を定義できます。関数レベルのアクセス権限を定義する場合、アクセス権限はリソースレベルのアクセス権限と同じか、制限がより大きいものである必要があります。

次のテーブルに、サポートされているアクセス権限の設定を示します。

GGC v1.10 or later

プロパティ	コンテナ化された Lambda 関数のみがリソースにアクセスする場合	コンテナ化されていない Lambda 関数がリソースにアクセスする場合
関数レベルのプロパティ		
権限 (読み取り/書き込み)	リソースがリソース所有者を定義していない場合は必須です。リソース所有者が定義されている場合、関数レベルのアクセス権限は、リソース所有者のアクセス権限と同じか、制限がより大きいものである必要があります。 コンテナ化された Lambda 関数だけがリソースにアクセスする場合は、リソースの所有者を定義しないことをお勧めします。	コンテナ化されていない Lambda 関数: サポート外。コンテナ化されていない Lambda 関数は、リソースレベルのアクセス権限を継承する必要があります。 コンテナ化された Lambda 関数: オプション。ただし、リソースレベルのアクセス権限と同じか制限がより大きいものである必要があります。

プロパティ	コンテナ化された Lambda 関数のみがリソースにアクセスする場合	コンテナ化されていない Lambda 関数がリソースにアクセスする場合
リソースレベルのプロパティ		
リソース所有者	オプション (推奨しません)。	必須。
権限 (読み取り/書き込み)	オプション (推奨しません)。	必須。

GGC v1.9 or earlier

プロパティ	コンテナ化された Lambda 関数のみがリソースにアクセスする場合	コンテナ化されていない Lambda 関数がリソースにアクセスする場合
関数レベルのプロパティ		
権限 (読み取り/書き込み)	必須。	サポート外。
リソースレベルのプロパティ		
リソース所有者	サポート外。	サポート外。
権限 (読み取り/書き込み)	サポート外。	サポート外。

> **Note**
>
> AWS IoT Greengrass API を使用して Lambda 関数とリソースを設定する場合、関数レベルの`ResourceId`プロパティも必要です。`ResourceId` プロパティは、機械学習リソースを Lambda 関数にアタッチします。

Lambda 関数のアクセス権限の定義 (コンソール)

AWS IoT コンソールでは、機械学習リソースを設定するとき、または Lambda 関数にアタッチするときにアクセス許可を定義します。

コンテナ化された Lambda 関数

コンテナ化された Lambda 関数だけが機械学習リソースにアタッチされている場合:

- 機械学習リソースのリソース所有者として [No system group] (システムグループなし) を選択します。これは、コンテナ化された Lambda 関数だけが機械学習リソースにアクセスする場合に推奨される設定です。そうしないと、アタッチされた Lambda 関数が必要以上に多くのアクセス権限を与える可能性があります。

コンテナ化されていない Lambda 関数 (GGC v1.10 以降が必要)

コンテナ化されていない Lambda 関数が機械学習リソースにアタッチされている場合:

- 機械学習リソースのリソース所有者として使用する System group ID (GID) (システムグループ ID (GID)) を指定します。[Specify system group and permissions] (システムグループと権限を指定) を選択し、GID を入力します。コアデバイスで `getent group` コマンドを使用して、システムグループ ID を検索できます。

- [System group permissions] (システムグループの権限) には [Read-only access] (読み取り専用アクセス) または [Read and write access] (読み取り/書き込みアクセス)] を選択します。

Lambda 関数 (API) のアクセス権限の定義

AWS IoT Greengrass API では、Lambda 関数の `ResourceAccessPolicy`プロパティまたはリソースの `OwnerSetting`プロパティで、機械学習リソースへのアクセス許可を定義します。

コンテナ化された Lambda 関数

コンテナ化された Lambda 関数だけが機械学習リソースにアタッチされている場合:

- コンテナ化された Lambda 関数の場合は、`ResourceAccessPolicies` プロパティの `Permission` プロパティにアクセス権限を定義します。例:

```
"Functions": [
    {
        "Id": "my-containerized-function",
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:function-name:alias-or-version",
```

```
        "FunctionConfiguration": {
            "Environment": {
                "ResourceAccessPolicies": [
                    {
                        "ResourceId": "my-resource-id",
                        "Permission": "ro-or-rw"
                    }
                ]
            },
            "MemorySize": 512,
            "Pinned": true,
            "Timeout": 5
        }
    }
]
```

- 機械学習リソースの場合は、`OwnerSetting` プロパティを省略します。例:

```
"Resources": [
    {
        "Id": "my-resource-id",
        "Name": "my-resource-name",
        "ResourceDataContainer": {
            "S3MachineLearningModelResourceData": {
                "DestinationPath": "/local-destination-path",
                "S3Uri": "s3://uri-to-resource-package"
            }
        }
    }
]
```

これは、コンテナ化された Lambda 関数だけが機械学習リソースにアクセスする場合に推奨される設定です。そうしないと、アタッチされた Lambda 関数が必要以上に多くのアクセス権限を与える可能性があります。

コンテナ化されていない Lambda 関数 (GGC v1.10 以降が必要)

コンテナ化されていない Lambda 関数が機械学習リソースにアタッチされている場合:

- コンテナ化されていない Lambda 関数の場合は、`ResourceAccessPolicies` の `Permission` プロパティを省略します。この設定は必須で、関数がリソースレベルのアクセス権限を継承できるようにします。例:

```
"Functions": [
    {
        "Id": "my-non-containerized-function",
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:function-
name:alias-or-version",
        "FunctionConfiguration": {
            "Environment": {
                "Execution": {
                    "IsolationMode": "NoContainer",
                },
                "ResourceAccessPolicies": [
                    {
                        "ResourceId": "my-resource-id"
                    }
                ]
            },
            "Pinned": true,
            "Timeout": 5
        }
    }
]
```

- 機械学習リソースにもアクセスするコンテナ化された Lambda 関数の場合は、ResourceAccessPolicies で Permission プロパティを省略するか、リソースレベルのアクセス権限と同じ、または制限のより厳しいアクセス権限を定義します。例:

```
"Functions": [
    {
        "Id": "my-containerized-function",
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:function-
name:alias-or-version",
        "FunctionConfiguration": {
            "Environment": {
                "ResourceAccessPolicies": [
                    {
                        "ResourceId": "my-resource-id",
                        "Permission": "ro-or-rw" // Optional, but cannot exceed
 the GroupPermission defined for the resource.
                    }
                ]
            },
            "MemorySize": 512,
```

```
            "Pinned": true,
            "Timeout": 5
        }
    }
]
```

- 機械学習リソースの場合は、子 `GroupOwner` および `GroupPermission` プロパティを含む `OwnerSetting` プロパティを定義します。例:

```
"Resources": [
    {
        "Id": "my-resource-id",
        "Name": "my-resource-name",
        "ResourceDataContainer": {
            "S3MachineLearningModelResourceData": {
                "DestinationPath": "/local-destination-path",
                "S3Uri": "s3://uri-to-resource-package",
                "OwnerSetting": {
                    "GroupOwner": "os-group-id",
                    "GroupPermission": "ro-or-rw"
                }
            }
        }
    }
]
```

Lambda 関数コードから機械学習リソースにアクセスする

ユーザー定義の Lambda 関数は、プラットフォーム固有の OS インターフェイスを使用して、Core デバイスの機械学習リソースにアクセスします。

GGC v1.10 or later

コンテナ化された Lambda 関数の場合、リソースは Greengrass コンテナ内にマウントされ、リソースに対して定義されたローカルの送信先パスで利用できます。コンテナ化されていない Lambda 関数の場合、リソースは Lambda 専用の作業ディレクトリにシンボリックリンクされ、Lambda プロセス内の `AWS_GG_RESOURCE_PREFIX` 環境変数に渡されます。

機械学習リソースのダウンロードされたアーティファクトへのパスを取得するために、Lambda 関数は、リソースに対して定義されたローカルの送信先パスに `AWS_GG_RESOURCE_PREFIX` 環

境変数を追加します。コンテナ化された Lambda 関数の場合、返される値は 1 つのスラッシュ (/) です。

```
resourcePath = os.getenv("AWS_GG_RESOURCE_PREFIX") + "/destination-path"
with open(resourcePath, 'r') as f:
    # load_model(f)
```

GGC v1.9 or earlier

機械学習リソースのダウンロードされたアーティファクトは、リソースに対して定義されたローカルの送信先パスにあります。 AWS IoT Greengrass Core v1.9 以前の機械学習リソースにアクセスできるのは、コンテナ化された Lambda 関数だけです。

```
resourcePath = "/local-destination-path"
with open(resourcePath, 'r') as f:
    # load_model(f)
```

モデルのロード実装は、ML ライブラリによって異なります。

トラブルシューティング

次の情報を使用して、機械学習リソースへのアクセスに関する問題のトラブルシューティングに役立ててください。

トピック

- InvalidMLModelOwner - GroupOwnerSetting は ML モデルリソースに提供されていますが、GroupOwner または GroupPermission がありません
- NoContainer 関数は、機械学習リソースをアタッチするときにアクセス権限を設定できません。<function-arn> は、リソースアクセスポリシーでアクセス権限 <ro/rw> を持つ機械学習リソース <resource-id> を参照します。
- 関数 <function-arn> は機械学習リソース <resource-id> を参照しますが、ResourceAccessPolicy とリソースの OwnerSetting のどちらにもアクセス権限がありません。
- 関数 <function-arn> は \"rw\" アクセス権限で機械学習リソース <resource-id> を参照しますが、リソース所有者設定の GroupPermission で許可されているのは \"ro\" のみです。
- NoContainer 関数 <function-arn> は、ネストされた送信先パスのリソースを参照します。
- Lambda <function-arn> は、同じグループ所有者 ID を共有することでリソース <resource-id> にアクセスします。

InvalidMLModelOwner - GroupOwnerSetting は ML モデルリソースに提供されていますが、GroupOwner または GroupPermission がありません

解決策: このエラーは、機械学習リソースに ResourceDownloadOwnerSetting オブジェクトが含まれていても、必須の `GroupOwner` または `GroupPermission` プロパティが定義されていない場合に表示されます。この問題を解決するには、不足しているプロパティを定義します。

NoContainer 関数は、機械学習リソースをアタッチするときにアクセス権限を設定できません。<function-arn> は、リソースアクセスポリシーでアクセス権限 <ro/rw> を持つ機械学習リソース <resource-id> を参照します。

解決策: コンテナ化されていない Lambda 関数が機械学習リソースに対する関数レベルのアクセス権限を指定した場合、このエラーが表示されます。コンテナ化されていない関数は、機械学習リソースに定義されているリソース所有者のアクセス権限からアクセス権限を継承する必要があります。この問題を解決するには、コンソールから [inherit resource owner permissions] (リソース所有者のアクセス許可を継承する) を選択するか、API を使用して Lambda 関数のリソースアクセスポリシーからアクセス権限を削除するかを選択します。

関数 <function-arn> は機械学習リソース <resource-id> を参照しますが、ResourceAccessPolicy とリソースの OwnerSetting のどちらにもアクセス権限がありません。

解決策: このエラーは、機械学習リソースへのアクセス権限が、アタッチされた Lambda 関数またはリソースに対して設定されていない場合に表示されます。この問題を解決するには、Lambda 関数の ResourceAccessPolicy プロパティか、またはリソースの OwnerSetting プロパティにアクセス権限を設定します。

関数 <function-arn> は \"rw\" アクセス権限で機械学習リソース <resource-id> を参照しますが、リソース所有者設定の GroupPermission で許可されているのは \"ro\" のみです。

解決策: このエラーは、アタッチされた Lambda 関数に定義されたアクセス権限が、機械学習リソースに対して定義されたリソース所有者のアクセス権限を超えた場合に表示されます。この問題を解決

するには、Lambda 関数に対して制限のより厳しいアクセス権限を設定するか、リソース所有者の制限がより低いアクセス権限を設定します。

NoContainer 関数 <function-arn> は、ネストされた送信先パスのリソースを参照します。

解決策: コンテナ化されていない Lambda 関数にアタッチされた複数の機械学習リソースが同じ送信先パスまたはネストされた送信先パスを使用している場合に、このエラーが表示されます。この問題を解決するには、リソースに別の送信先パスを指定します。

Lambda <function-arn> は、同じグループ所有者 ID を共有することでリソース <resource-id> にアクセスします。

解決策: このエラーは、Lambda 関数の実行者 ID と、機械学習リソースのリソース所有者に同じ OS グループを指定しながら、リソースが Lambda 関数にアタッチされていない場合に `runtime.log` に記録されます。この設定では、Lambda 関数に暗黙のアクセス権限が付与されます。このアクセス権限は、 AWS IoT Greengrass の認可なしでリソースにアクセスするために使用できます。

この問題を解決するには、プロパティの 1 つに別の OS グループを使用するか、機械学習リソースを Lambda 関数にアタッチします。

関連情報

- 機械学習の推論を実行する
- the section called “機械学習推論を設定する方法”
- the section called “最適化された機械学習推論を設定する方法”
- AWS IoT Greengrass Version 1 API リファレンス

AWS Management Consoleを使用して機械学習推論を設定する方法

このチュートリアルのステップを実行するには、 AWS IoT Greengrass Core v1.10 以降が必要です。

ローカルに生成されたデータを使用して、Greengrass コアデバイスで機械学習 (ML) 推論をローカルで実行できます。要件と制約を含め、情報については、「機械学習の推論を実行する」を参照してください。

このチュートリアルでは、 を使用して Greengrass グループを設定し、クラウド AWS Management Console にデータを送信せずに、カメラからローカルで画像を認識する Lambda 推論アプリケーションを実行する方法について説明します。推論アプリケーションは、Raspberry Pi 上のカメラモジュールにアクセスし、オープンソースの SqueezeNet モデルを使用して推論を実行します。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Raspberry Pi を設定する
2. MXNet フレームワークをインストールする
3. モデルパッケージを作成する
4. Lambda 関数を作成して発行する
5. グループに Lambda 関数を追加する
6. グループにリソースを追加する
7. グループにサブスクリプションを追加する
8. グループをデプロイする
9. アプリケーションをテストする

前提条件

このチュートリアルを完了するには、以下が必要です。

- Raspberry Pi 4 モデル B、または Raspberry Pi 3 モデル B/B+ は、 で使用するようにセットアップおよび設定されています AWS IoT Greengrass。Raspberry Pi を AWS IoT Greengrassと共にセットアップする前に、Greengrass Device Setup スクリプトを実行するか、の開始方法 AWS IoT Greengrass のモジュール 1 とモジュール 2 を完了していることを確認します。

> **Note**
>
> Raspberry Pi では、イメージ分類に一般的に使用される深層学習のフレームワークを動かすために、2.5A の電源が必要な場合があります。定格の低い電源を使用すると、デバイスが再起動する場合があります。

- Raspberry Pi カメラモジュール V2 - 8 メガピクセル、1080p。カメラの設定方法については、Raspberry Pi ドキュメントのカメラの接続に関するトピックを参照してください。
- Greengrass グループと Greengrass コア。Greengrass グループまたはコアを作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。

> **Note**
>
> このチュートリアルでは Raspberry Pi を使用しますが、Intel Atom や NVIDIA Jetson TX2 などの他のプラットフォーム AWS IoT Greengrass をサポートしています。Jetson TX2 の例は、カメラからストリームされた画像の代わりに静的画像を使用できます。Jetson TX2 の例を使用する場合は、Python 3.7 ではなく Python 3.6 をインストールする必要があります。
> AWS IoT Greengrass Core ソフトウェアをインストールできるようにデバイスを設定する方法については、「」を参照してくださいthe section called “他のデバイスの設定”。
> AWS IoT Greengrass がサポートしていないサードパーティープラットフォームの場合、Lambda 関数をコンテナ化されていないモードで実行する必要があります。非コンテナ化モードで実行するには、Lambda 関数を root として実行する必要があります。詳細については、the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”およびthe section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”を参照してください。

ステップ 1: Raspberry Pi を設定する

このステップでは、Raspbian オペレーティングシステムの更新プログラムをインストールし、カメラモジュールソフトウェアと Python の依存関係をインストールして、カメラインターフェイスを有効にします。

Raspberry Pi のターミナルで以下のコマンドを実行します。

1. Raspbian に更新プログラムをインストールします。

```
sudo apt-get update
sudo apt-get dist-upgrade
```

2. カメラモジュールの `picamera` インターフェイス、およびこのチュートリアルに必要なその他の Python ライブラリをインストールします。

```
sudo apt-get install -y python3-dev python3-setuptools python3-pip python3-picamera
```

インストールを検証します。

- Python 3.7 のインストールに pip が含まれていることを確認します。

  ```
  python3 -m pip
  ```

  pip がインストールされていない場合は、pip ウェブサイトからダウンロードし、次のコマンドを実行します。

  ```
  python3 get-pip.py
  ```

- Python のバージョンが 3.7 以上であることを確認します。

  ```
  python3 --version
  ```

  出力に以前のバージョンが表示されている場合は、次のコマンドを実行します。

  ```
  sudo apt-get install -y python3.7-dev
  ```

- Setuptools と Picamera が正常にインストールされたことを確認します。

  ```
  sudo -u ggc_user bash -c 'python3 -c "import setuptools"'
  sudo -u ggc_user bash -c 'python3 -c "import picamera"'
  ```

  出力にエラーが含まれていない場合、検証は成功です。

> **Note**
>
> デバイスにインストールされている Python 実行可能ファイルが `python3.7` である場合は、このチュートリアルのコマンドに `python3` ではなく、`python3.7` を使用します。依存関係のエラーを回避するために、pip インストールが正しい `python3` バージョンまたは `python3.7` バージョンにマップされていることを確認してください。

3. Raspberry Pi を再起動します。

```
sudo reboot
```

4. Raspberry Pi 設定ツールを開きます。

```
sudo raspi-config
```

5. 矢印キーを使用して [インターフェイスオプション] を開き、カメラインターフェイスを有効にします。プロンプトが表示されたら、デバイスを再起動します。
6. 以下のコマンドを使用して、カメラの設定をテストします。

```
raspistill -v -o test.jpg
```

これにより、Raspberry Pi のプレビューウィンドウが開き、`test.jpg` という写真が現在のディレクトリに保存されて、カメラに関する情報が Raspberry Pi のターミナルに表示されます。

ステップ 2: MXNet フレームワークをインストールする

このステップでは、Raspberry Pi に MXNet ライブラリをインストールします。

1. リモートで Raspberry Pi にサインインします。

```
ssh pi@your-device-ip-address
```

2. MXNet のドキュメントを開き、MXNet のインストールを開いて、指示に従って MXNet をデバイスにインストールします。

> **Note**
> このチュートリアルでは、デバイスの競合を回避するために、バージョン 1.5.0 をインストールし、ソースから MXNet をビルドすることをお勧めします。

3. MXNet をインストールしたら、次の設定を検証します。
   - `ggc_user` システムアカウントが MXNet フレームワークを使用できることを確認します。

```
sudo -u ggc_user bash -c 'python3 -c "import mxnet"'
```

- NumPy がインストールされていることを確認します。

```
sudo -u ggc_user bash -c 'python3 -c "import numpy"'
```

ステップ 3: MXNet モデルパッケージを作成する

このステップでは、Amazon Simple Storage Service (Amazon S3) にアップロードする事前トレーニング済みのサンプル MXNet モデルを含むモデルパッケージを作成します。tar.gz または zip 形式を使用する場合に限り、 AWS IoT Greengrass は Amazon S3 のモデルパッケージを使用できます。Amazon S3

1. コンピュータで、the section called “機械学習のサンプル” から Raspberry Pi 用の MXNet サンプルをダウンロードします。
2. ダウンロードした `mxnet-py3-armv7l.tar.gz` ファイルを解凍します。
3. `squeezenet` ディレクトリに移動します。

   ```
   cd path-to-downloaded-sample/mxnet-py3-armv7l/models/squeezenet
   ```

   このディレクトリ内の `squeezenet.zip` ファイルはモデルパッケージです。これには、イメージ分類モデルの SqueezeNet オープンソースモデルアーティファクトが含まれています。後で、このモデルパッケージを Amazon S3 にアップロードします。

ステップ 4: Lambda 関数を作成して発行する

この手順では、Lambda 関数デプロイパッケージと Lambda 関数を作成します。次に、関数のバージョンを公開し、エイリアスを作成します。

まず、Lambda 関数デプロイパッケージを作成します。

1. コンピュータで、「the section called “モデルパッケージを作成する”」で解凍したサンプルパッケージ内の `examples` ディレクトリに移動します。

   ```
   cd path-to-downloaded-sample/mxnet-py3-armv7l/examples
   ```

   `examples` ディレクトリには、関数コードと依存関係が含まれています。

- `greengrassObjectClassification.py` は、このチュートリアルで使用される推論コードです。このコードをテンプレートとして使用して、独自の推論関数を作成できます。
- `greengrasssdk` は、 AWS IoT Greengrass Core SDK for Python のバージョン 1.5.0 です。

> **Note**
>
> 新しいバージョンが利用できる場合は、そのバージョンをダウンロードし、デプロイパッケージ内の SDK バージョンをアップグレードできます。詳細については、GitHub の「AWS IoT Greengrass Core SDK for Python」を参照してください。

2. `examples` ディレクトリの内容を `greengrassObjectClassification.zip` という名前のファイルに圧縮します。このファイルがデプロイパッケージです。

```
zip -r greengrassObjectClassification.zip .
```

> **Note**
>
> `.py` ファイルと依存関係がディレクトリのルートにあることを確認します。

次に、Lambda 関数を作成します。

3. AWS IoT コンソールから、関数 と 関数の作成 を選択します。
4. [一から作成] を選択し、以下の値を使用して関数を作成します。
   - [関数名] に「**`greengrassObjectClassification`**」と入力します。
   - [ランタイム] で [Python 3.7] を選択します。

   [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。
5. [Create function (関数の作成)] を選択します。

今度は、Lambda 関数デプロイパッケージをアップロードし、ハンドラを登録します。

6. [Lambda function](Lambda 関数) を選択し、Lambda 関数のデプロイパッケージをアップロードします。

   a. [コード] タブの [コードソース] で、[アップロード元] を選択します。ドロップダウンから [.zip ファイル] を選択します。

   b. [アップロード] を選択し、`greengrassObjectClassification.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

   c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

      - [ランタイム] で [Python 3.7] を選択します。
      - [ハンドラ] に **`greengrassObjectClassification.function_handler`** と入力します。

      [保存] を選択します。

次に、Lambda 関数の最初のバージョンを発行します。その後、バージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

7. [アクション] メニューから、[新しいバージョンを発行] を選択します。

8. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。
9. [greengrassObjectClassification: 1] 設定ページで、[Actions] (アクション) メニューの [エイリアスの作成] を選択します。

10. [新しいエイリアスの作成] ページで、次の値を使用します。
    - [名前] に **`mlTest`** と入力します。
    - [バージョン] に **`1`** と入力します。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

11. [Save] を選択します。

次に、Lambda 関数を Greengrass グループに追加します。

ステップ 5: Lambda 関数を Greengrass グループに追加する

このステップでは、Lambda 関数をグループに追加してから、そのライフサイクルと環境変数を設定します。

まず、Lambda 関数を Greengrass グループに追加します。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。

3. [My Lambda functions] (自分の Lambda 関数) セクションから、[Add] (追加) を選択します。
4. [Lambda 関数] で、[greengrassObjectClassification] を選択します。
5. Lambda 関数のバージョンで、Alias:mlTest を選択します。

次に、Lambda 関数のライフサイクルと環境変数を設定します。

6. [Lambda function configuration] (Lambda 関数の設定) セクションで次のように更新します。

> **Note**
>
> ビジネスケースで要求される場合を除き、Lambda 関数はコンテナ化を使用せずに実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバイスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  >
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [タイムアウト] に **10 seconds** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [System user and group] (システムユーザーとグループ) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [メモリ制限] に **96 MB** と入力します。
- [タイムアウト] に **10 seconds** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。

7. [環境変数] で、キーと値のペアを作成します。キーと値のペアは、Raspberry Pi 上で MXNet モデルとやり取りする関数で必要となります。

   キーには、MXNET_ENGINE_TYPE を使用します。値には、NaiveEngine を使用します。

> **Note**
> 独自のユーザー定義 Lambda 関数では、必要に応じて、関数コードで環境変数を設定できます。

8. 他のすべてのプロパティではデフォルト値を保持し、[Add Lambda function] (Lambda 関数の追加) を選択します。

ステップ 6: Greengrass グループにリソースを追加する

このステップでは、カメラモジュールと ML 推論モデルのリソースを作成し、リソースを Lambda 関数に関連付けます。これにより、Lambda 関数がコアデバイス上のリソースにアクセスできるようになります。

> **Note**
>
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はこれらのデバイスリソースを設定せずにデバイス GPU とカメラにアクセスできます。

まず、カメラ用に 2 つのローカルデバイスリソースを作成します。1 つは共有メモリ用、もう 1 つはデバイスインターフェイス用です。ローカルリソースアクセスの詳細については、「Lambda 関数とコネクタを使用してローカルリソースにアクセスする」を参照してください。

1. グループ設定ページで、[Resources] (リソース) タブを選択します。
2. [Local resources] (ローカルリソース) タブで、[Add local resource] (ローカルリソースの追加) を選択します。
3. [Add a local resource] (ローカルリソースの追加) ページで、次の値を使用します。
   - [リソース名] に **`videoCoreSharedMemory`** と入力します。
   - [リソースタイプ] で、[デバイス] を選択します。
   - [Local device path] (ローカルデバイスパス) には、「**`/dev/vcsm`**」と入力します。

     デバイスパスはデバイスリソースのローカル絶対パスです。このパスでは、`/dev` 以下のキャラクターデバイスまたはブロックデバイスのみを参照できます。
   - [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

     [System group owner and file access permissions] (システムグループ所有者のファイルアクセス権限) オプションを使用すると、Lambda プロセスに追加のファイルアクセス権限をグラントできます。詳細については、「グループ所有者のファイルアクセス権限」を参照してください。
4. 次に、カメラインターフェイス用にローカルデバイスリソースを追加します。

5. [Add local resource] (ローカルリソースの追加) を選択します。
6. [Add a local resource] (ローカルリソースの追加) ページで、次の値を使用します。
   - [リソース名] に **`videoCoreInterface`** と入力します。
   - [リソースタイプ] で、[デバイス] を選択します。
   - [Local device path] (ローカルデバイスパス) には、「**`/dev/vchiq`**」と入力します。
   - [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。
7. ページの下部で、[Add resource] (リソースの追加) を選択します。

ここで、推論モデルを機械学習リソースとして追加します。このステップでは、`squeezenet.zip` モデルパッケージを Amazon S3 にアップロードします。

1. グループの [Resources] (リソース) タブの、[Machine Learning] (機械学習) で、[Add machine learning resour] (機械学習リソースの追加) を選択します。
2. [Add a machine learning resource (機械学習リソースの追加) ページで、[Resource name] (リソース名) に、「 **`squeezenet_model`**」と入力します。
3. [Model source] (モデルソース) で、[Use a model stored in S3, such as a model optimized through Deep Learning Compiler] (深層学習コンパイラで最適化されたモデルなど、S3 に保存されているモデルを使用する) を選択します。
4. [S3 URI]には、S3 バケットが保存されているパスを入力します。
5. [S3 を参照] を選択します。これにより、新しいタブで Amazon S3 コンソールが開きます。
6. Amazon S3 コンソールタブで、`squeezenet.zip` ファイルを S3 バケットにアップロードします。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットにファイルとフォルダをアップロードする方法」を参照してください。

> **Note**
>
> S3 バケットにアクセスできるようにするには、バケット名に文字列 **`greengrass`** が含まれている必要があります。バケットは、 AWS IoT Greengrassに使用するリージョンと同じリージョンに存在する必要があります。一意の名前 (**`greengrass-`**

bucket-*user-id*-*epoch-time* など) を選択します。バケット名にピリオド (.) を使用しないでください。

7. AWS IoT Greengrass コンソールタブで、S3 バケットを見つけて選択します。アップロードした `squeezenet.zip` ファイルを見つけ、[選択] を選択します。利用可能なバケットとファイルのリストを更新するために [更新] を選択する必要がある場合があります。
8. [送信先パス] に **`/greengrass-machine-learning/mxnet/squeezenet`** と入力します。

   これは、Lambda ランタイム名前空間内のローカルモデルのターゲットです。グループをデプロイすると、 はソースモデルパッケージ AWS IoT Greengrass を取得し、指定したディレクトリにコンテンツを抽出します。このチュートリアルのサンプル Lambda 関数は、このパス (`model_path` 変数) を使用するように既に設定されています。
9. [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可) で、[No system group] (システムグループなし) を選択します。
10. [リソースを追加] を選択します。

SageMaker AI トレーニング済みモデルの使用

このチュートリアルでは、Amazon S3 に保存されているモデルを使用しますが、SageMaker AI モデルも簡単に使用できます。 AWS IoT Greengrass コンソールには SageMaker AI 統合が組み込まれているため、これらのモデルを Amazon S3 に手動でアップロードする必要はありません。SageMaker AI モデルを使用する際の要件と制限については、「」を参照してくださいthe section called “サポートされているモデルソース”。

SageMaker AI モデルを使用するには：

- モデルソースで、SageMaker AI で AWS トレーニングされたモデルを使用し、モデルのトレーニングジョブの名前を選択します。
- [Destination path] (送信先) に、Lambda 関数によってモデルが検索されるディレクトリへのパスを入力します。

ステップ 7: Greengrass グループにサブスクリプションを追加する

このステップでは、グループにサブスクリプションを追加します。このサブスクリプションにより、Lambda 関数は MQTT トピックに発行 AWS IoT することで、予測結果を に送信できます。

1. グループ設定ページで、[Subscriptions] (サブスクリプション) タブ、[Add Subscription] (サブスクリプションの追加) の順に選択します。
2. [Subscription details] (サブスクリプションの詳細) ページで、ソースおよびターゲットを次のように設定します。
   a. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数)、[greengrassObjectClassification] の順に選択します。
   b. [Target type] (ターゲットタイプ)で、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。
3. [Topic filter] (トピックのフィルター) で、「**`hello/world`**」と入力し、[Create subscription] (サブスクリプションの作成) を選択します。

ステップ 8: Greengrass グループをデプロイする

このステップでは、グループ定義の現在のバージョンを Greengrass コアデバイスにデプロイします。この定義には、追加した Lambda 関数、リソース、サブスクリプション設定が含まれます。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。
   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   出力に `root` で実行中の `/greengrass/ggc/packages/1.11.6/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

   > **Note**
   > パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

   b. デーモンを開始するには、以下を実行します。

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。

3. [Lambda functions] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) セクションで、[IP detector] (IP ディテクター)、[Edit} (編集) の順に選択します。
4. [IP ディテクターの設定を編集] ダイアログボックスで、[MQTT ブローカーエンドポイントを自動的に検出して上書きする] を選択します。
5. [Save] を選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   >
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイのステータスが [Completed] (完了) と表示されます。

   デプロイの詳細については、「AWS IoT Greengrass グループのデプロイ」を参照してください。トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

ステップ 9: 推論アプリケーションをテストする

これで、デプロイが正しく設定されているかどうかを確認できます。テストするには、`hello/world` トピックにサブスクライブし、Lambda 関数によって発行された予測結果を表示します。

> **Note**
>
> モニターが Raspberry Pi に接続されている場合、ライブカメラのフィードがプレビューウィンドウに表示されます。

1. AWS IoT コンソールのテストで、MQTT テストクライアントを選択します。
2. [サブスクリプション] で、以下の値を使用します。
   - サブスクリプションのトピックには、hello/world を使用します。
   - [Additional configuration] (追加設定) から、[MQTT payload display] (MQTT ペイロード表示) で、[Display payloads as strings] (文字列としてペイロードを表示) を選択します。
3. [サブスクライブ] を選択します。

   テストに成功すると、Lambda 関数からのメッセージがページの下部に表示されます。各メッセージには、確率、予測クラス ID、対応するクラス名の形式を使用して、イメージの上位 5 つの予測結果が含まれます。

ML 推論 AWS IoT Greengrass のトラブルシューティング

テストに成功しなかった場合は、以下のトラブルシューティング手順を実行できます。Raspberry Pi のターミナルで以下のコマンドを実行します。

エラーログを確認する

1. root ユーザーに切り替え、log ディレクトリに移動します。 AWS IoT Greengrass ログにアクセスするには、ルートアクセス許可が必要です。

   ```
   sudo su
   cd /greengrass/ggc/var/log
   ```

2. system ディレクトリで、runtime.log または python_runtime.log を確認します。

   user/*region*/*account-id* ディレクトリで、greengrassObjectClassification.log を確認します。

   詳細については、「the section called “ログによるトラブルシューティング”」を参照してください。

runtime.log 内の 解凍エラー

runtime.log に以下のようなエラーが含まれる場合は、tar.gz ソースモデルパッケージに親ディレクトリがあることを確認します。

```
Greengrass deployment error: unable to download the artifact model-arn: Error while
 processing.
Error while unpacking the file from /tmp/greengrass/artifacts/model-arn/path to /
greengrass/ggc/deployment/path/model-arn,
error: open /greengrass/ggc/deployment/path/model-arn/squeezenet/
squeezenet_v1.1-0000.params: no such file or directory
```

モデルファイルを格納する親ディレクトリがパッケージに含まれていない場合は、以下のコマンドを使用してモデルを再パッケージ化します。

```
tar -zcvf model.tar.gz ./model
```

例:

```
#$ tar -zcvf test.tar.gz ./test
./test
./test/some.file
./test/some.file2
./test/some.file3
```

> **Note**
>
> このコマンドでは、末尾に /* 文字を含めないでください。

Lambda 関数が正常にデプロイされていることを確認する

1. /lambda ディレクトリ内のデプロイされた Lambda の内容を一覧表示します。コマンドを実行する前に、プレースホルダーの値を置き換えてください。

```
cd /greengrass/ggc/deployment/lambda/
arn:aws:lambda:region:account:function:function-name:function-version
ls -la
```

2. ディレクトリに、ステップ 4: Lambda 関数を作成して発行する でアップロードした greengrassObjectClassification.zip デプロイパッケージと同じ内容が含まれることを確認します。

   .py ファイルと依存関係がディレクトリのルートにあることを確認します。

推論モデルが正常にデプロイされていることを確認する

1. Lambda ランタイムプロセスのプロセス識別番号 (PID) を見つけます。

```
ps aux | grep 'lambda-function-name*'
```

   出力では、Lambda ランタイムプロセスの行の 2 列目に PID が表示されます。

2. Lambda ランタイム名前空間を入力します。コマンドを実行する前に、*pid* プレースホルダーの値を置き換えてください。

> **Note**
>
> このディレクトリとその内容は、Lambda ランタイム名前空間にあるため、通常の Linux 名前空間には表示されません。

```
sudo nsenter -t pid -m /bin/bash
```

3. ML リソース用に指定したローカルディレクトリの内容を一覧表示します。

```
cd /greengrass-machine-learning/mxnet/squeezenet/
ls -ls
```

以下のファイルが表示されます。

```
32 -rw-r--r-- 1 ggc_user ggc_group   31675 Nov 18 15:19 synset.txt
32 -rw-r--r-- 1 ggc_user ggc_group   28707 Nov 18 15:19 squeezenet_v1.1-symbol.json
4832 -rw-r--r-- 1 ggc_user ggc_group 4945062 Nov 18 15:19
 squeezenet_v1.1-0000.params
```

次のステップ

次に、他の推論アプリを調べます。 は、ローカル推論を試すために使用できる他の Lambda 関数 AWS IoT Greengrass を提供します。サンプルパッケージは、「the section called “MXNet フレームワークをインストールする”」でダウンロードしたプリコンパイル済みライブラリフォルダにあります。

インテル Atom の設定

Intel Atom デバイスでこのチュートリアルを実行するには、ソースイメージを指定し、Lambda 関数を設定して、別のローカルデバイスリソースを追加する必要があります。GPU を推論に使用するには、デバイスに次のソフトウェアがインストールされていることを確認します。

- OpenCL バージョン 1.0 以降
- Python 3.7 と pip

> **Note**
>
> デバイスが Python 3.6 で構築済みの場合は、代わりに Python 3.7 へのシンボリックリンクを作成することができます。詳細については、「Step 2」を参照してください。

- NumPy

- ホイール上の OpenCV

1. Lambda 関数用に静的な PNG あるいは JPG 画像をダウンロードして、イメージ分類に使用します。この例は小さいイメージファイルで最適に動作します。

   `greengrassObjectClassification.py` ファイルがあるディレクトリ (あるいは、このディレクトリのサブディレクトリ) に画像ファイルを保存します。これは、the section called "Lambda 関数を作成して発行する" でアップロードした Lambda 関数デプロイパッケージにあります。

   > **Note**
   >
   > を使用している場合は AWS DeepLens、オンボードカメラを使用するか、独自のカメラをマウントして、静的イメージの代わりにキャプチャされたイメージに対して推論を実行できます。ただし、最初に静的イメージから開始することを強くお勧めします。
   > カメラを使用する場合は、`awscam` APT パッケージがインストールされていて、最新の状態であることを確認してください。詳細については、「AWS DeepLens デベロッパーガイド」の「AWS DeepLens デバイスの更新」を参照してください。

2. Python 3.7 を使用していない場合は、Python 3.x から Python 3.7 へのシンボリックリンクを必ず作成します。これにより、 で Python 3 を使用するようにデバイスが設定されます AWS IoT Greengrass。次のコマンドを実行して、Python のインストールを検索します。

   ```
   which python3
   ```

   次のコマンドを実行して、シンボリックリンクを作成します。

   ```
   sudo ln -s path-to-python-3.x/python3.x path-to-python-3.7/python3.7
   ```

   デバイスを再起動します。

3. Lambda 関数の設定を編集します。「the section called "グループに Lambda 関数を追加する"」の手順に従います。

   > **Note**
   >
   > ビジネスケースで要求される場合を除き、Lambda 関数を、コンテナ化を使用しないで実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバ

イスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  >
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。詳しい手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

  コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [Timeout (タイムアウト)] 値を 5 秒に更新します。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned](固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。
- [Lambda のライフサイクル] で、[存続期間が長く無制限に稼働する関数にする] を選択します。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
>
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [Timeout (タイムアウト)] 値を 5 秒に更新します。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned] (固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

4. コンテナ化モードで実行している場合、必要なローカルデバイスリソースを追加して、デバイス GPU へのアクセスを付与します。

> **Note**
>
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はデバイスリソースを設定せずにデバイス GPU にアクセスできます。

a. グループ設定ページで、[Resources] (リソース) タブを選択します。
b. [Add local resource] (ローカルリソースの追加) を選択します。
c. リソースを定義します。
   - [リソース名] に **`renderD128`** と入力します。
   - [[Resource type] (リソースタイプ) で、[Local device] (ローカルデバイス) を選択します。
   - [デバイスパス] に「**`/dev/dri/renderD128`**」と入力します。
   - [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。
   - [Lambda function affiliations] (Lambda 関数の所属) で、Lambda 関数への [Read and write access] (読み取りと書き込みアクセス) を許可します。

NVIDIA Jetson TX2 の設定

NVIDIA Jetson TX2 でこのチュートリアルを実行するには、ソースイメージを指定して、Lambda 関数を設定します。GPU を使用している場合、さらにローカルデバイスリソースを追加する必要があります。

1. AWS IoT Greengrass Core ソフトウェアをインストールできるように、Jetson デバイスが設定されていることを確認します。デバイスの設定の詳細については、「the section called “他のデバイスの設定”」を参照してください。
2. MXNet のドキュメントを開き、Jetson への MXNet のインストールに移動し、指示に従って Jetson デバイスに MXNet をインストールします。

   > **Note**
   >
   > ソースから MXNet をビルドする場合は、指示に従って共有ライブラリをビルドします。`config.mk` ファイルの次の設定を編集して、Jetson TX2 デバイスを操作します。
   >
   > - `-gencode arch=compute-62, code=sm_62` を `CUDA_ARCH` 設定に追加します。
   > - CUDA を有効にします。
   >
   > ```
   > USE_CUDA = 1
   > ```

3. Lambda 関数用に静的な PNG あるいは JPG 画像をダウンロードして、イメージ分類に使用します。このアプリは小さな画像ファイルで最適に動作します。また、Jetson ボードにカメラを設置して、ソースイメージをキャプチャできます。

   `greengrassObjectClassification.py` ファイルを含むディレクトリにイメージファイルを保存します。このディレクトリのサブディレクトリに保存することもできます。このディレクトリは、the section called “Lambda 関数を作成して発行する” でアップロードした Lambda 関数デプロイパッケージにあります。

4. Python 3.7 から Python 3.6 へのシンボリックリンクを作成して、 で Python 3 を使用します AWS IoT Greengrass。次のコマンドを実行して、Python のインストールを検索します。

   ```
   which python3
   ```

   次のコマンドを実行して、シンボリックリンクを作成します。

```
sudo ln -s path-to-python-3.6/python3.6 path-to-python-3.7/python3.7
```

デバイスを再起動します。

5. `ggc_user` システムアカウントが MXNet フレームワークを使用できることを確認します。

```
“sudo -u ggc_user bash -c 'python3 -c "import mxnet"'
```

6. Lambda 関数の設定を編集します。「the section called “グループに Lambda 関数を追加する”」の手順に従います。

> **Note**
>
> ビジネスケースで要求される場合を除き、Lambda 関数を、コンテナ化を使用しないで実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバイスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  >
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。
- [Environment variables] (環境変数) で、次のキーと値のペアを Lambda 関数に追加します。これにより、MXNet フレームワークを使用する AWS IoT Greengrass ように が設定されます。

キー	値
PATH	/usr/local/cuda/bin:$PATH
MXNET_HOME	$HOME/mxnet/
PYTHONPATH	$MXNET_HOME/python:$PYTHONPATH
CUDA_HOME	/usr/local/cuda
LD_LIBRARY_PATH	$LD_LIBRARY_PATH:${CUDA_HOME}/lib64

b. 代わりにコンテナ化モードで実行するには:

> **Note**
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [メモリ制限] の値を増やします。CPU の場合は 500 MB、GPU の場合は少なくとも 2000 MB を使用します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。
- [Environment variables] (環境変数) で、次のキーと値のペアを Lambda 関数に追加します。これにより、MXNet フレームワークを使用する AWS IoT Greengrass ように が設定されます。

キー	値
PATH	/usr/local/cuda/bin:$PATH
MXNET_HOME	$HOME/mxnet/
PYTHONPATH	$MXNET_HOME/python:$PYTHONPATH
CUDA_HOME	/usr/local/cuda
LD_LIBRARY_PATH	$LD_LIBRARY_PATH:${CUDA_HOME}/lib64

7. コンテナ化モードで実行している場合、次のローカルデバイスリソースを追加して、デバイス GPU へのアクセスを許可します。「the section called “グループにリソースを追加する”」の手順に従います。

> **Note**
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はデバイスリソースを設定せずにデバイス GPU にアクセスできます。

リソースごとに:

- [リソースタイプ] で、[デバイス] を選択します。
- [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

名前	デバイスパス
nvhost-ctrl	/dev/nvhost-ctrl
nvhost-gpu	/dev/nvhost-gpu

名前	デバイスパス
nvhost-ctrl-gpu	/dev/nvhost-ctrl-gpu
nvhost-dbg-gpu	/dev/nvhost-dbg-gpu
nvhost-prof-gpu	/dev/nvhost-prof-gpu
nvmap	/dev/nvmap
nvhost-vic	/dev/nvhost-vic
tegra_dc_ctrl	/dev/tegra_dc_ctrl

8. コンテナ化モードで実行している場合で、次のローカルボリュームリソースを追加して、デバイスカメラへのアクセスを許可します。「the section called “グループにリソースを追加する”」の手順に従います。

> **Note**
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はボリュームリソースを設定せずにデバイスカメラにアクセスできます。

- [リソースタイプ] で、[ボリューム] を選択します。
- [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

名前	ソースパス	送信先パス
shm	/dev/shm	/dev/shm
tmp	/tmp	/tmp

最適化された機械学習推論を AWS Management Consoleを使用して設定する方法

このチュートリアルのステップに従うには、 AWS IoT Greengrass Core v1.10 以降を使用する必要があります。

SageMaker AI Neo 深層学習コンパイラを使用すると、Tensorflow、Apache MXNet、PyTorch、ONNX、XGBoost フレームワークのネイティブ機械学習推論モデルの予測効率を最適化して、フットプリントを小さくし、パフォーマンスを向上させることができます。その後、最適化されたモデルをダウンロードして SageMaker AI Neo 深層学習ランタイムをインストールし、AWS IoT Greengrass デバイスにデプロイして推論を高速化できます。

このチュートリアルでは、 を使用して Greengrass グループを設定し、クラウド AWS Management Console にデータを送信せずにカメラからのイメージをローカルで認識する Lambda 推論の例を実行する方法について説明します。推論例では、Raspberry Pi のカメラモジュールにアクセスします。このチュートリアルでは、Resnet-50 によってトレーニングされて Neo 深層学習コンパイラで最適化された、事前にパッケージ化されたモデルをダウンロードします。次に、モデルを使用して、AWS IoT Greengrass デバイスでローカルイメージ分類を実行します。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Raspberry Pi を設定する
2. Neo 深層学習ランタイムをインストールする
3. 推論 Lambda 関数を作成する
4. グループに Lambda 関数を追加する
5. Neo 最適化モデルリソースをグループに追加する
6. カメラデバイスリソースをグループに追加する
7. サブスクリプションをグループに追加する
8. グループをデプロイする
9. 例をテストする

前提条件

このチュートリアルを完了するには、以下が必要です。

- Raspberry Pi 4 モデル B、または Raspberry Pi 3 モデル B/B+ は、 で使用するようにセットアップおよび設定されています AWS IoT Greengrass。Raspberry Pi を AWS IoT Greengrassと共にセットアップする前に、Greengrass Device Setup スクリプトを実行するか、の開始方法 AWS IoT Greengrass のモジュール 1 とモジュール 2 を完了していることを確認します。

> **Note**
>
> Raspberry Pi では、イメージ分類に一般的に使用される深層学習のフレームワークを動かすために、2.5A の電源が必要な場合があります。定格の低い電源を使用すると、デバイスが再起動する場合があります。

- Raspberry Pi カメラモジュール V2 - 8 メガピクセル、1080p。カメラの設定方法については、Raspberry Pi ドキュメントで「カメラの接続」を参照してください。
- Greengrass グループと Greengrass コア。Greengrass グループまたはコアを作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。

> **Note**
>
> このチュートリアルでは Raspberry Pi を使用しますが、Intel Atom や NVIDIA Jetson TX2 などの他のプラットフォーム AWS IoT Greengrass をサポートしています。Intel Atom の例を使用する場合は、Python 3.7 ではなく Python 3.6 のインストールが必要な場合があります。AWS IoT Greengrass Core ソフトウェアをインストールできるようにデバイスを設定する方法については、「」を参照してくださいthe section called “他のデバイスの設定”。
> AWS IoT Greengrass がサポートしていないサードパーティープラットフォームの場合、Lambda 関数をコンテナ化されていないモードで実行する必要があります。非コンテナ化モードで実行するには、Lambda 関数を root として実行する必要があります。詳細については、the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”およびthe section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”を参照してください。

ステップ 1: Raspberry Pi を設定する

このステップでは、Raspbian オペレーティングシステムの更新プログラムをインストールし、カメラモジュールソフトウェアと Python の依存関係をインストールして、カメラインターフェイスを有効にします。

Raspberry Pi のターミナルで以下のコマンドを実行します。

1. Raspbian に更新プログラムをインストールします。

```
sudo apt-get update
sudo apt-get dist-upgrade
```

2. カメラモジュールの picamera インターフェイス、およびこのチュートリアルに必要なその他の Python ライブラリをインストールします。

```
sudo apt-get install -y python3-dev python3-setuptools python3-pip python3-picamera
```

インストールを検証します。

- Python 3.7 のインストールに pip が含まれていることを確認します。

```
python3 -m pip
```

pip がインストールされていない場合は、pip ウェブサイトからダウンロードし、次のコマンドを実行します。

```
python3 get-pip.py
```

- Python のバージョンが 3.7 以上であることを確認します。

```
python3 --version
```

出力に以前のバージョンが表示されている場合は、次のコマンドを実行します。

```
sudo apt-get install -y python3.7-dev
```

- Setuptools と Picamera が正常にインストールされたことを確認します。

```
sudo -u ggc_user bash -c 'python3 -c "import setuptools"'
sudo -u ggc_user bash -c 'python3 -c "import picamera"'
```

出力にエラーが含まれていない場合、検証は成功です。

> **Note**
>
> デバイスにインストールされている Python 実行可能ファイルが `python3.7` である場合は、このチュートリアルのコマンドに `python3` ではなく、`python3.7` を使用します。依存関係のエラーを回避するために、pip インストールが正しい `python3` バージョンまたは `python3.7` バージョンにマップされていることを確認してください。

3. Raspberry Pi を再起動します。

```
sudo reboot
```

4. Raspberry Pi 設定ツールを開きます。

```
sudo raspi-config
```

5. 矢印キーを使用して [インターフェイスオプション] を開き、カメラインターフェイスを有効にします。プロンプトが表示されたら、デバイスを再起動します。
6. 以下のコマンドを使用して、カメラの設定をテストします。

```
raspistill -v -o test.jpg
```

これにより、Raspberry Pi のプレビューウィンドウが開き、`test.jpg` という写真が現在のディレクトリに保存されて、カメラに関する情報が Raspberry Pi のターミナルに表示されます。

ステップ 2: Amazon SageMaker Neo 深層学習ランタイムをインストールする

このステップでは、Neo 深層学習ランタイム (DLR) を Raspberry Pi にインストールドします。

> **Note**
>
> このチュートリアルでは、バージョン 1.1.0 をインストールすることをお勧めします。

1. リモートで Raspberry Pi にサインインします。

```
ssh pi@your-device-ip-address
```

2. DLR のドキュメントを開き、DLR のインストールを開き、Raspberry Pi デバイスのホイール URL を見つけます。次に、指示に従って、デバイスに DLR をインストールします。例えば、pip を使用できます。

```
pip3 install rasp3b-wheel-url
```

3. DLR をインストールした後、次の設定を検証します。

   - `ggc_user` システムアカウントが DLR ライブラリを使用できることを確認します。

     ```
     sudo -u ggc_user bash -c 'python3 -c "import dlr"'
     ```

   - NumPy がインストールされていることを確認します。

     ```
     sudo -u ggc_user bash -c 'python3 -c "import numpy"'
     ```

ステップ 3: 推論 Lambda 関数を作成する

この手順では、Lambda 関数デプロイパッケージと Lambda 関数を作成します。次に、関数のバージョンを公開し、エイリアスを作成します。

1. コンピュータで、the section called “機械学習のサンプル” から Raspberry Pi の DLR サンプルをダウンロードします。
2. ダウンロードした `dlr-py3-armv7l.tar.gz` ファイルを解凍します。

   ```
   cd path-to-downloaded-sample
   tar -xvzf dlr-py3-armv7l.tar.gz
   ```

   抽出されたサンプルパッケージの `examples` ディレクトリには、関数コードと依存関係が含まれています。

   - `inference.py` は、このチュートリアルで使用される推論コードです。このコードをテンプレートとして使用して、独自の推論関数を作成できます。
   - `greengrasssdk` は、 AWS IoT Greengrass Core SDK for Python のバージョン 1.5.0 です。

> **Note**
>
> 新しいバージョンが利用できる場合は、そのバージョンをダウンロードし、デプロイパッケージ内の SDK バージョンをアップグレードできます。詳細については、GitHub の「AWS IoT Greengrass Core SDK for Python」を参照してください。

3. `examples` ディレクトリの内容を `optimizedImageClassification.zip` という名前のファイルに圧縮します。このファイルがデプロイパッケージです。

```
cd path-to-downloaded-sample/dlr-py3-armv7l/examples
zip -r optimizedImageClassification.zip .
```

デプロイメントパッケージには、関数コードと依存関係が含まれています。これには、Neo 深層学習ランタイム Python API を呼び出して、Neo 深層学習コンパイラモデルで推論を実行するコードが含まれます。

> **Note**
>
> `.py` ファイルと依存関係がディレクトリのルートにあることを確認します。

4. 次に、Lambda 関数を Greengrass グループに追加します。

   [Lambda console](Lambda コンソール) ページから、[Functions] (関数)、[Create function] (関数の作成) の順に選択します。

5. [一から作成] を選択し、以下の値を使用して関数を作成します。

   - [関数名] に「**`optimizedImageClassification`**」と入力します。
   - [ランタイム] で [Python 3.7] を選択します。

   [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

6. [Create function (関数の作成)] を選択します。

今度は、Lambda 関数デプロイパッケージをアップロードし、ハンドラを登録します。

1. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

2. `optimizedImageClassification.zip` デプロイパッケージを選択し、[Save] (保存) を選択します。
3. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

   - [ランタイム] で [Python 3.7] を選択します。
   - [ハンドラ] に **`inference.handler`** と入力します。

   [Save] を選択します。

次に、Lambda 関数の最初のバージョンを発行します。その後、バージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

1. [アクション] メニューから、[新しいバージョンを発行] を選択します。

2. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。
3. [optimizedImageClassification: 1] 設定ページで、[Actions (アクション)] メニューの [エイリアスの作成] を選択します。

4. [新しいエイリアスの作成] ページで、次の値を使用します。
   - [名前] に **`mlTestOpt`** と入力します。
   - [バージョン] に **`1`** と入力します。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

5. [Create] (作成) を選択します。

   ここで、Greengrass グループに Lambda 関数を追加します。

ステップ 4: Lambda 関数を Greengrass グループに追加する

このステップでは、Lambda 関数をグループに追加し、そのライフサイクルを設定します。

まず、Lambda 関数を Greengrass グループに追加します。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. グループ設定ページで、[Lambda functions] (Lambda 関数) タブ、[Add] (追加) の順に選択します。
3. [Lambda function] (Lambda 関数) と[optimizedImageClassification]を選択します。
4. [Lambda function version] (Lambda 関数のバージョン) で、公開したバージョンのエイリアスを選択します。

次に、Lambda 関数のライフサイクルを設定します。

1. [Lambda function configuration] (Lambda 関数の設定) セクションで次のように更新します。

> **Note**
>
> ビジネスケースで要求される場合を除き、Lambda 関数はコンテナ化を使用せずに実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバイスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  >
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

  コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [タイムアウト] に **`10 seconds`** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。

- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
>
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [System user and group] (システムユーザーとグループ) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [メモリ制限] に **1024 MB** と入力します。
- [タイムアウト] に **10 seconds** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

2. [Add Lambda function] (Lambda 関数の追加) を選択します。

ステップ 5: Greengrass グループに SageMaker AI Neo 最適化モデルリソースを追加する

このステップでは、最適化された ML 推論モデルのリソースを作成して Amazon S3 バケットにアップロードします。次に、 AWS IoT Greengrass コンソールで Amazon S3 アップロードされたモデルを見つけ、新しく作成されたリソースを Lambda 関数に関連付けます。これにより、関数が Core デバイス上のリソースにアクセスできるようになります。

1. コンピュータで、「the section called “推論 Lambda 関数を作成する”」で解凍したサンプルパッケージ内の `resnet50` ディレクトリに移動します。

> **Note**
>
> NVIDIA Jetson の例を使用する場合は、代わりにサンプルパッケージの `resnet18` ディレクトリを使用する必要があります。詳細については、「the section called “NVIDIA Jetson TX2 の設定”」を参照してください。

```
cd path-to-downloaded-sample/dlr-py3-armv7l/models/resnet50
```

このディレクトリには、Resnet-50 でトレーニングされたイメージ分類モデルのプリコンパイルされたモデルアーティファクトが含まれています。

2. `resnet50` ディレクトリ内のファイルを `resnet50.zip` という名前のファイルに圧縮します。

```
zip -r resnet50.zip .
```

3. グループのグループ設定ページで AWS IoT Greengrass 、リソースタブを選択します。[Machine Learning (機械学習)] セクションに移動し、[機械学習リソースの追加] を選択します。[Create a machine learning resource (機械学習リソースの作成)] ページで、[Resource name (リソース名)] に**`resnet50_model`** と入力します。
4. [Model source] (モデルソース) で、[Use a model stored in S3, such as a model optimized through Deep Learning Compiler] (深層学習コンパイラで最適化されたモデルなど、S3 に保存されているモデルを使用する) を選択します。
5. [S3 URI]から、選択 [Browse S3] (S3 の閲覧) を選択します。

> **Note**
>
> 現在、最適化された SageMaker AI モデルは Amazon S3 に自動的に保存されます。このオプションを使用して、Amazon S3 バケット内の最適化されたモデルを見つけることができます。SageMaker AI でのモデル最適化の詳細については、SageMaker AI Neo ドキュメントを参照してください。

6. [モデルをアップロードする] を選択します。
7. Amazon S3 コンソールタブで、zip ファイルを Amazon S3 バケットにアップロードします。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットにファイルとフォルダをアップロードする方法」を参照してください。

> **Note**
>
> バケット名には文字列 **`greengrass`** が含まれている必要があります。一意の名前 (**`greengrass-dlr-bucket-`***`user-id`***`-`***`epoch-time`** など) を選択します。バケット名にピリオド (`.`) を使用しないでください。

8. AWS IoT Greengrass コンソールタブで、Amazon S3 バケットを見つけて選択します。アップロードした `resnet50.zip` ファイルを見つけ、[選択] を選択します。必要に応じてページを更新し、使用可能なバケットとファイルのリストを更新します。

9. [Destination path] (送信先) に「**`/ml_model`**」と入力します。

これは、Lambda ランタイム名前空間内のローカルモデルのターゲットです。グループをデプロイすると、 はソースモデルパッケージ AWS IoT Greengrass を取得し、コンテンツを指定されたディレクトリに抽出します。

> **Note**
>
> ローカルパスに指定されている正確なパスを使用することを強くお勧めします。このステップで別のローカルモデルのターゲットパスを使用すると、このチュートリアルで示しているいくつかのトラブルシューティングコマンドが正確でなくなります。別のパスを使用する場合は、ここで指定する正確なパスを `MODEL_PATH` 環境変数に設定する必要があります。環境変数については、「AWS Lambda 環境変数」を参照してください。

10. コンテナ化モードで実行している場合:

    a. [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可) から、[Specify system group and permissions] (システムグループと権限を指定する) を選択します。

    b. [Read-only access] (読み取り専用アクセス)、[Add resources] (リソースの追加) の順に選択します。

ステップ 6: Greengrass グループにカメラデバイスリソースを追加する

このステップでは、カメラモジュールのリソースを作成し、Lambda 関数に関連付けます。これにより、Lambda 関数がコアデバイス上のリソースにアクセスできるようになります。

> **Note**
>
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はこのデバイスリソースを設定せずにデバイス GPU とカメラにアクセスできます。

1. グループ設定ページで、[Resources] (リソース) タブを選択します。
2. [Local resources] (ローカルリソース) タブで、[Add local resource] (ローカルリソースの追加) を選択します。
3. [Add a local resource] (ローカルリソースの追加) ページで、次の値を使用します。
   - [リソース名] に **`videoCoreSharedMemory`** と入力します。
   - [リソースタイプ] で、[デバイス] を選択します。
   - [Local device path] (ローカルデバイスパス) には、「**`/dev/vcsm`**」と入力します。

     デバイスパスはデバイスリソースのローカル絶対パスです。このパスは、`/dev` 下の文字デバイスまたはブロックデバイスのみを参照できます。
   - [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

     [Group owner file access permission] (グループ所有者のファイルアクセス権限) オプションを使用すると、Lambda プロセスに追加のファイルアクセス権限を付与できます。詳細については、「グループ所有者のファイルアクセス権限」を参照してください。
4. ページの下部で、[Add resource] (リソースの追加) を選択します。
5. [Resources] (リソース) タブから、[Add] (追加) を選択して、別のローカルリソースを作成し、次の値を使用します。
   - [リソース名] に **`videoCoreInterface`** と入力します。
   - [リソースタイプ] で、[デバイス] を選択します。
   - [Local device path] (ローカルデバイスパス) には、「**`/dev/vchiq`**」と入力します。
   - [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。
6. [リソースを追加] を選択します。

ステップ 7: サブスクリプションを Greengrass グループに追加する

このステップでは、グループにサブスクリプションを追加します。これらのサブスクリプションにより、Lambda 関数は MQTT トピックに発行 AWS IoT することで、予測結果を に送信できます。

1. グループ設定ページで、[Subscriptions] (サブスクリプション) タブ、[Add subscription] (サブスクリプションの追加) の順に選択します。
2. [Create a subscription] (サブスクリプションの作成) ページで、ソースおよびターゲットを次のように設定します。
   a. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数)、[optimizedImageClassification] の順に選択します。
   b. [Target type] (ターゲットタイプ)で、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。
   c. [Topic filter] (トピックのフィルター) で、「**`/resnet-50/predictions`**」と入力し、[Create subscription] (サブスクリプションの作成) を選択します。
3. 2 つ目のサブスクリプションを追加します。[Subscriptions] (サブスクリプション) タブ、[Add subscription] (サブスクリプションの追加) の順に選択し、ソースとターゲットを次のように設定します。
   a. [Source type] (ソースタイプ) で、[Service] (サービス)、[IoT Cloud] (IoT クラウド) の順に選択します。
   b. [Target type] (ターゲットタイプ) で、[Lambda function] (Lambda 関数)、[optimizedImageClassification] の順に選択します。
   c. [Topic filter] (トピックのフィルター) で、「**`/resnet-50/test`**」と入力し、[Create subscription] (サブスクリプションの作成) を選択します。

ステップ 8: Greengrass グループをデプロイする

このステップでは、グループ定義の現在のバージョンを Greengrass コアデバイスにデプロイします。この定義には、追加した Lambda 関数、リソース、サブスクリプション設定が含まれます。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。
   a. デーモンが実行中かどうかを確認するには、以下を実行します。

```
ps aux | grep -E 'greengrass.*daemon'
```

出力に root で実行中の /greengrass/ggc/packages/*latest-core-version*/bin/daemon のエントリが含まれていれば、デーモンは実行されています。

b. デーモンを開始するには、以下を実行します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。
3. [Lambda functions] (Lambda 関数) タブで、[IP detector] (IP ディテクター)と [Edit] (編集) を選択します。
4. [Edit IP detector settings] (IP ディテクター設定の編集) のダイアログボックスで、[Automatically detect and override MQTT broker endpoints] (MQTT ブローカーのエンドポイントを自動的に検出して上書きする) と、Save (保存) を選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイのステータスが [Completed] (完了) と表示されます。

   デプロイの詳細については、「AWS IoT Greengrass グループのデプロイ」を参照してください。トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

推論例をテストする

これで、デプロイが正しく設定されているかどうかを確認できます。テストするには、`/resnet-50/predictions` トピックにサブスクライブし、`/resnet-50/test` トピックにメッセージを発行します。これにより、Raspberry Pi で写真を撮影してキャプチャしたイメージの推論を実行する Lambda 関数がトリガーされます。

> **Note**
>
> NVIDIA Jetson の例を使用する場合は、代わりに `resnet-18/predictions` トピックおよび `resnet-18/test` トピックを使用してください。

> **Note**
>
> モニターが Raspberry Pi に接続されている場合、ライブカメラのフィードがプレビューウィンドウに表示されます。

1. AWS IoT コンソールのホームページのテストで、MQTT テストクライアントを選択します。
2. [Subscriptions] (サブスクリプション) で、[Subscribe to a Topic] (トピックへのサブスクライブ) を選択します。以下の値を使用します。残りのオプションはデフォルトのままにします。
   - [Subscription topic (サブスクリプショントピック)] で、**`/resnet-50/predictions`** と入力します。
   - [Additional configuration] (追加設定) から、[MQTT payload display] (MQTT ペイロード表示) で、[Display payloads as strings] (文字列としてペイロードを表示) を選択します。
3. [サブスクライブ] を選択します。
4. [Publish to a topic] (トピックに公開) を選択して、[Topic name] (トピック名) として「**`/resnet-50/test`**」を入力し、[Publish] (発行) を選択します。
5. テストが成功すると、発行されたメッセージによって Raspberry Pi カメラがイメージをキャプチャします。Lambda 関数からのメッセージがページの下部に表示されます。このメッセージには、予測クラス名、確率、最大メモリ使用量の形式で、イメージの予測結果が含まれています。

インテル Atom の設定

Intel Atom デバイスでこのチュートリアルを実行するには、ソースイメージを指定し、Lambda 関数を設定して、別のローカルデバイスリソースを追加する必要があります。GPU を推論に使用するには、デバイスに次のソフトウェアがインストールされていることを確認します。

- OpenCL バージョン 1.0 以降
- Python 3.7 と pip
- NumPy
- ホイール上の OpenCV

1. Lambda 関数用に静的な PNG あるいは JPG 画像をダウンロードして、イメージ分類に使用します。この例は小さいイメージファイルで最適に動作します。

   `inference.py` ファイルがあるディレクトリ (あるいは、このディレクトリのサブディレクトリ) に画像ファイルを保存します。これは、the section called “推論 Lambda 関数を作成する” でアップロードした Lambda 関数デプロイパッケージにあります。

   > **Note**
   >
   > を使用している場合は AWS DeepLens、オンボードカメラを使用するか、独自のカメラをマウントして、静的イメージの代わりにキャプチャされたイメージに対して推論を実行できます。ただし、最初に静的イメージから開始することを強くお勧めします。
   > カメラを使用する場合は、`awscam` APT パッケージがインストールされていて、最新の状態であることを確認してください。詳細については、「AWS DeepLens デベロッパーガイド」の「AWS DeepLens デバイスの更新」を参照してください。

2. Lambda 関数の設定を編集します。「the section called “グループに Lambda 関数を追加する”」の手順に従います。

   > **Note**
   >
   > ビジネスケースで要求される場合を除き、Lambda 関数を、コンテナ化を使用しないで実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバイスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する

場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

  コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [タイムアウト] の値を 2 分に増やします。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned] (固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [メモリ制限] の値を 3000 MB に増やします。
- [タイムアウト] の値を 2 分に増やします。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned] (固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

3. Neo 最適化モデルリソースをグループに追加します。the section called “推論 Lambda 関数を作成する” で解凍したサンプルパッケージの `resnet50` ディレクトリにモデルリソースをアップロードします。このディレクトリには、Resnet-50 でトレーニングされたイメージ分類モデルのプリコンパイルされたモデルアーティファクトが含まれています。次の更新で、the section called “Neo 最適化モデルリソースをグループに追加する” の手順に従います。

   - `resnet50` ディレクトリ内のファイルを `resnet50.zip` という名前のファイルに圧縮します。
   - [Create a machine learning resource (機械学習リソースの作成)] ページで、[Resource name (リソース名)] に**`resnet50_model`** と入力します。
   - `resnet50.zip` ファイルをアップロードします。

4. コンテナ化モードで実行している場合、必要なローカルデバイスリソースを追加して、デバイス GPU へのアクセスを付与します。

   > **Note**
   >
   > コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はデバイスリソースを設定せずにデバイス GPU にアクセスできます。

   a. グループ設定ページで、[Resources] (リソース) タブを選択します。

   b. [Local resources] (ローカルリソース) タブで、[Add local resource] (ローカルリソースの追加) を選択します。

   c. リソースを定義します。

      - [リソース名] に **`renderD128`** と入力します。
      - [リソースタイプ] で、[デバイス] を選択します。

- [Local device path] (ローカルデバイスパス) には、「**`/dev/dri/renderD128`**」と入力します。
- [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

NVIDIA Jetson TX2 の設定

NVIDIA Jetson TX2 でこのチュートリアルを実行するには、ソースイメージを指定して、Lambda 関数を設定し、さらにローカルデバイスリソースを追加します。

1. Core AWS IoT Greengrass ソフトウェアをインストールし、推論に GPU を使用できるように、Jetson デバイスが設定されていることを確認します。デバイスの設定の詳細については、「the section called “他のデバイスの設定”」を参照してください。NVIDIA Jetson TX2 で推論に GPU を使用するには、Jetpack 4.3 でボードのイメージを作成するときに、デバイスに CUDA 10.0 と cuDNN 7.0 をインストールする必要があります。
2. Lambda 関数用に静的な PNG あるいは JPG 画像をダウンロードして、イメージ分類に使用します。この例は小さいイメージファイルで最適に動作します。

   `inference.py` ファイルを含むディレクトリにイメージファイルを保存します。このディレクトリのサブディレクトリに保存することもできます。このディレクトリは、the section called “推論 Lambda 関数を作成する” でアップロードした Lambda 関数デプロイパッケージにあります。

   > **Note**
   >
   > 代わりに、Jetson ボードにカメラを設置して、ソースイメージをキャプチャすることもできます。ただし、最初に静的イメージから開始することを強くお勧めします。

3. Lambda 関数の設定を編集します。「the section called “グループに Lambda 関数を追加する”」の手順に従います。

   > **Note**
   >
   > ビジネスケースで要求される場合を除き、Lambda 関数を、コンテナ化を使用しないで実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバ

イスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [Run as] (として実行) に、「**Another user ID/group ID**」を選択します。[UID] に、「**0**」と入力します。[GUID] に、「**0**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called "グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定"」を参照してください。

  > **Tip**
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called "root としての Lambda 関数の実行"」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

  コンテナ化を使用しない実行の詳細については、「the section called "Lambda 関数のコンテナ化を選択する場合の考慮事項"」を参照してください。

- [タイムアウト] の値を 5 分に増やします。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned] (固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [メモリ制限] の値を増やします。指定されたモデルを GPU モードで使用するには、少なくとも 2000 MB を使用します。
- [タイムアウト] の値を 5 分に増やします。これにより、リクエストの早過ぎるタイムアウトがなくなります。セットアップ後、推論の実行には数分かかります。
- [Pinned] (固定)で、[True] を選択します。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

4. Neo 最適化モデルリソースをグループに追加します。the section called “推論 Lambda 関数を作成する” で解凍したサンプルパッケージの `resnet18` ディレクトリにモデルリソースをアップロードします。このディレクトリには、Resnet-18 でトレーニングされたイメージ分類モデルのプリコンパイル済みのモデルアーティファクトが含まれています。次の更新で、the section called “Neo 最適化モデルリソースをグループに追加する” の手順に従います。

   - `resnet18` ディレクトリ内のファイルを `resnet18.zip` という名前のファイルに圧縮します。
   - [Create a machine learning resource (機械学習リソースの作成)] ページで、[Resource name (リソース名)] に**`resnet18_model`** と入力します。
   - `resnet18.zip` ファイルをアップロードします。

5. コンテナ化モードで実行している場合、必要なローカルデバイスリソースを追加して、デバイス GPU へのアクセスを付与します。

   > **Note**
   >
   > コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はデバイスリソースを設定せずにデバイス GPU にアクセスできます。

   a. グループ設定ページで、[Resources] (リソース) タブを選択します。
   b. [Local resources] (ローカルリソース) タブで、[Add local resource] (ローカルリソースの追加) を選択します。
   c. 各リソースを定義します。

      - [リソース名] と [デバイスパス] には、次の表の値を使用します。テーブルの行ごとに 1 つのデバイスリソースを作成します。

- [リソースタイプ] で、[デバイス] を選択します。
- [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

名前	デバイスパス
nvhost-ctrl	/dev/nvhost-ctrl
nvhost-gpu	/dev/nvhost-gpu
nvhost-ctrl-gpu	/dev/nvhost-ctrl-gpu
nvhost-dbg-gpu	/dev/nvhost-dbg-gpu
nvhost-prof-gpu	/dev/nvhost-prof-gpu
nvmap	/dev/nvmap
nvhost-vic	/dev/nvhost-vic
tegra_dc_ctrl	/dev/tegra_dc_ctrl

6. コンテナ化モードで実行している場合で、次のローカルボリュームリソースを追加して、デバイスカメラへのアクセスを許可します。「the section called “Neo 最適化モデルリソースをグループに追加する”」の手順に従います。

> **Note**
> コンテナ化されていないモードで を実行する場合、 AWS IoT Greengrass はデバイスリソースを設定せずにデバイスカメラにアクセスできます。

- [リソースタイプ] で、[ボリューム] を選択します。
- [System group owner and file access permissions] (システムグループ所有者のファイルアクセス許可)で、[Automatically add file system permissions of the system group that owns the

resource] (リソースを所有するシステムグループのファイルシステム権限を自動的に追加する) を選択します。

名前	ソースパス	送信先パス
shm	/dev/shm	/dev/shm
tmp	/tmp	/tmp

7. 正しいディレクトリを使用するようにグループサブスクリプションを更新します。次の更新で、the section called “サブスクリプションをグループに追加する” の手順に従います。
   - 最初のトピックフィルタには、**/resnet-18/predictions** と入力します。
   - 2 番目のトピックフィルタには、**/resnet-18/test** と入力します。
8. 正しいディレクトリを使用するようにテストサブスクリプションを更新します。次の更新で、the section called “例をテストする” の手順に従います。
   - [Subscriptions] (サブスクリプション) で、[Subscribe to a Topic] (トピックへのサブスクライブ) を選択します。[Subscription topic (サブスクリプショントピック)] で、**/resnet-18/predictions** と入力します。
   - `/resnet-18/predictions` ページで、発行先の `/resnet-18/test` トピックを指定します。

ML AWS IoT Greengrass 推論のトラブルシューティング

テストに成功しなかった場合は、以下のトラブルシューティング手順を実行できます。Raspberry Pi のターミナルで以下のコマンドを実行します。

エラーログを確認する

1. root ユーザーに切り替え、`log` ディレクトリに移動します。 AWS IoT Greengrass ログにアクセスするには、ルートアクセス許可が必要です。

```
sudo su
cd /greengrass/ggc/var/log
```

2. `runtime.log` でエラーがないかどうかを確認します。

```
cat system/runtime.log | grep 'ERROR'
```

ユーザー定義の Lambda 関数ログでもエラーがないかどうかを確認できます。

```
cat user/your-region/your-account-id/lambda-function-name.log | grep 'ERROR'
```

詳細については、「the section called “ログによるトラブルシューティング”」を参照してください。

Lambda 関数が正常にデプロイされていることを確認する

1. `/lambda` ディレクトリ内のデプロイされた Lambda の内容を一覧表示します。コマンドを実行する前に、プレースホルダーの値を置き換えてください。

   ```
   cd /greengrass/ggc/deployment/lambda/
   arn:aws:lambda:region:account:function:function-name:function-version
   ls -la
   ```

2. ディレクトリに、ステップ 3: 推論 Lambda 関数を作成する でアップロードした `optimizedImageClassification.zip` デプロイパッケージと同じ内容が含まれることを確認します。

   `.py` ファイルと依存関係がディレクトリのルートにあることを確認します。

推論モデルが正常にデプロイされていることを確認する

1. Lambda ランタイムプロセスのプロセス識別番号 (PID) を見つけます。

   ```
   ps aux | grep lambda-function-name
   ```

   出力では、Lambda ランタイムプロセスの行の 2 列目に PID が表示されます。

2. Lambda ランタイム名前空間を入力します。コマンドを実行する前に、*pid* プレースホルダーの値を置き換えてください。

> **Note**
>
> このディレクトリとその内容は、Lambda ランタイム名前空間にあるため、通常の Linux 名前空間には表示されません。

```
sudo nsenter -t pid -m /bin/bash
```

3. ML リソース用に指定したローカルディレクトリの内容を一覧表示します。

> **Note**
>
> ML リソースのパスが `ml_model` 以外の場合は、ここで置き換えてください。

```
cd /ml_model
ls -ls
```

以下のファイルが表示されます。

```
    56 -rw-r--r-- 1 ggc_user ggc_group     56703 Oct 29 20:07 model.json
196152 -rw-r--r-- 1 ggc_user ggc_group 200855043 Oct 29 20:08 model.params
   256 -rw-r--r-- 1 ggc_user ggc_group    261848 Oct 29 20:07 model.so
    32 -rw-r--r-- 1 ggc_user ggc_group     30564 Oct 29 20:08 synset.txt
```

Lambda 関数で **/dev/dri/renderD128** が見つからない

このエラーは、OpenCL から必要な GPU デバイスに接続できない場合に発生します。Lambda 関数に必要なデバイスでデバイスリソースを作成する必要があります。

次のステップ

次は、最適化された他のモデルを試します。詳細については、SageMaker AI Neo ドキュメントを参照してください。

AWS IoT Greengrass コアでデータストリームを管理する

AWS IoT Greengrass ストリームマネージャーを使用すると、大量の IoT データを に簡単かつ確実に転送できます AWS クラウド。ストリームマネージャーは、データストリームをローカルで処理し、 AWS クラウド に自動的にエクスポートします。この機能は、機械学習 (ML) 推論などの一般的なエッジシナリオと統合され、データをローカルで処理して分析してから、 AWS クラウド または ローカルストレージの送信先にエクスポートします。

ストリームマネージャーは、アプリケーション開発を簡素化します。IoT アプリケーションは、カスタムストリーム管理機能を構築する代わりに、標準化されたメカニズムを使用して大量のストリームを処理し、ローカルデータ保持ポリシーを管理できます。IoT アプリケーションは、ストリームの読み書きが可能です。ストレージタイプ、サイズ、データ保持に関するポリシーをストリームごとに定義して、ストリームマネージャーがストリームを処理および エクスポートする方法を制御できます。

ストリームマネージャーは、断続的または制限された接続環境で動作するように設計されています。帯域幅の使用、タイムアウト動作、コアが接続または切断されたときのストリームデータの処理方法を定義できます。重要なデータの場合は、優先順位を設定して、ストリームを AWS クラウドに エクスポートする順序を制御できます。

への自動エクスポートを設定して、 AWS クラウド ストレージや、さらに処理および分析を行うことができます。ストリームマネージャーは、次の AWS クラウド 宛先への エクスポートをサポートしています。

- のチャネル AWS IoT Analytics。 AWS IoT Analytics を使用すると、データに対して高度な分析を実行して、ビジネス上の意思決定を行い、機械学習モデルを改善できます。詳細については、「 AWS IoT Analytics ユーザーガイド」の「What is AWS IoT Analytics？」を参照してください。
- Kinesis Data Streams のストリーム Kinesis Data Streams は、一般的に、大量のデータを集約して、データウェアハウスまたは MapReduce クラスターに読み込むために使用されます。詳細については、「Amazon Kinesis デベロッパーガイド」の「Amazon Kinesis Data Streams とは」を参照してください。
- のアセットプロパティ AWS IoT SiteWise。 AWS IoT SiteWise を使用すると、産業機器からデータを大規模に収集、整理、分析できます。詳細については、「 AWS IoT SiteWise ユーザーガイド」の「What is AWS IoT SiteWise？」を参照してください。
- Amazon S3 のオブジェクト。Amazon S3 を使用すると、膨大なデータの保存と取得を行えます。詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「Amazon S3 とは」を参照してください。

ストリーム管理ワークフロー

IoT アプリケーションは、 AWS IoT Greengrass Core SDK を介してストリームマネージャーとやり取りします。単純なワークフローの場合、Greengrass コア上で実行されるユーザー定義 Lambda 関数は、時系列温度や圧力メトリクスなどの IoT データを消費します。Lambda 関数は、データをフィルタリングまたは圧縮し、 AWS IoT Greengrass Core SDK を呼び出してストリームマネージャーのストリームにデータを書き込む場合があります。ストリームマネージャーは、ストリームに定義されたポリシーに基づいて、ストリームを自動的に AWS クラウド にエクスポートします。ユーザー定義 Lambda 関数は、ローカルデータベースまたはストレージリポジトリにデータを直接送信することもできます。

IoT アプリケーションには、ストリームの読み書きを行うユーザー定義 Lambda 関数を複数含めることができます。これらのローカル Lambda 関数は、ストリームに対して読み書きを行い、データをローカルでフィルタリング、集約、分析できます。これにより、コアからクラウドまたはローカルの送信先にデータを転送する前に、ローカルイベントに迅速に対応し、貴重な情報を抽出することができます。

次の図に、ワークフローの例を示します。

ストリームマネージャーを使用するには、まずストリームマネージャーパラメータを設定して、Greengrass Core 上のすべてのストリームに適用するグループレベルのランタイム設定を定義します。これらのカスタマイズ可能な設定を使用すると、ビジネスニーズと環境の制約に基づいて、ストリームマネージャーがストリームを保存、処理、エクスポートする方法を制御できます。詳細については、「the section called “ストリームマネージャーの設定”」を参照してください。

ストリームマネージャーを設定したら、IoT アプリケーションを作成してデプロイできます。これらは通常、 AWS IoT Greengrass Core SDK `StreamManagerClient`で を使用してストリームを作成および操作するユーザー定義の Lambda 関数です。この Lambda 関数は、ストリームの作成時に、エクスポート先、優先度、永続性といった、ストリームごとのポリシーを定義します。`StreamManagerClient` 操作用コードスニペットなどの詳細については、「the section called “ストリームを操作するために StreamManagerClient を使用する”」を参照してください。

単純なワークフローを設定するチュートリアルについては、「the section called “データストリームのエクスポート (コンソール)”」または「the section called “データストリームのエクスポート (CLI)”」を参照してください。

要件

ストリームマネージャーを使用する際に次の要件が適用されます。

- Core AWS IoT Greengrass ソフトウェア v1.10 以降を使用し、ストリームマネージャーを有効にする必要があります。詳細については、「the section called “ストリームマネージャーの設定”」を参照してください。

  ストリームマネージャーは OpenWrt ディストリビューションではサポートされていません。
- Java 8 ランタイム (JDK 8) をコアにインストールする必要があります。
  - Debian ベースのディストリビューション (Raspbian を含む) または Ubuntu ベースのディストリビューションの場合は、次のコマンドを実行します。

    ```
    sudo apt install openjdk-8-jdk
    ```
  - Red Hat ベースのディストリビューション (Amazon Linux を含む) の場合は、次のコマンドを実行します。

    ```
    sudo yum install java-1.8.0-openjdk
    ```

    詳細については、OpenJDK ドキュメントの「How to download and install prebuilt OpenJDK packages」を参照してください。
- ストリームマネージャーには、基本 AWS IoT Greengrass Core ソフトウェアに加えて、70 MB 以上の RAM が必要です。合計メモリ要件は、ワークロードによって異なります。
- ユーザー定義の Lambda 関数は、AWS IoT Greengrass Core SDK を使用してストリームマネージャーと対話する必要があります。 AWS IoT Greengrass Core SDK は複数の言語で利用できますが、ストリームマネージャーオペレーションをサポートしているのは次のバージョンのみです。
  - Java SDK (v1.4.0 以降)
  - Python SDK (v1.5.0 以降)

- Node.js SDK (v1.6.0 以降)

Lambda 関数ランタイムに対応する SDK のバージョンをダウンロードし、Lambda 関数デプロイパッケージに含めます。

> **Note**
>
> AWS IoT Greengrass Core SDK for Python には Python 3.7 以降が必要で、その他のパッケージ依存関係があります。詳細については、「Lambda 関数のデプロイパッケージを作成する (コンソール)」または「Lambda 関数のデプロイパッケージを作成する (CLI)」を参照してください。

- ストリームの AWS クラウド エクスポート先を定義する場合は、エクスポートターゲットを作成し、Greengrass グループロールでアクセス許可を付与する必要があります。送信先により、他の要件も適用される場合があります。詳細については、以下を参照してください。
  - the section called “AWS IoT Analytics チャネル”
  - the section called “Amazon Kinesis Data Streams”
  - the section called “AWS IoT SiteWise アセットプロパティ”
  - the section called “Amazon S3 オブジェクト”

  これらの AWS クラウド リソースを維持するのはお客様の責任です。

データセキュリティ

ストリームマネージャーを使用する場合は、次のセキュリティ上の考慮事項に注意してください。

ローカルデータセキュリティ

AWS IoT Greengrass は、コアデバイスのコンポーネント間で保管中または転送中のストリームデータをローカルで暗号化しません。

- 保管時のデータ。ストリームデータは、Greengrass コアのストレージディレクトリにローカルに保存されます。データセキュリティのために、 は、Unix ファイルのアクセス許可とフルディスク暗号化が有効になっている場合 AWS IoT Greengrass に依存します。オプションの STREAM_MANAGER_STORE_ROOT_DIR パラメータを使用して、ストレージディレクトリを指定できます。このパラメータを後で変更して別のストレージディレクトリを使用する場合、 AWS IoT Greengrass は以前のストレージディレクトリまたはそのコンテンツを削除しません。

- ローカルに送信されるデータ。 AWS IoT Greengrass は、データソース、Lambda 関数、 AWS IoT Greengrass Core SDK、ストリームマネージャー間でコアでローカルに送信されるストリームデータを暗号化しません。

- に転送中のデータ AWS クラウド。ストリームマネージャーによって にエクスポートされたデータストリームは、Transport Layer Security (TLS) による標準 AWS サービスクライアント暗号化 AWS クラウド を使用します。

詳細については、「the section called “データ暗号化”」を参照してください。

クライアント承認

ストリームマネージャークライアントは AWS IoT Greengrass Core SDK を使用してストリームマネージャーと通信します。クライアント認証が有効になっている場合、Greengrass グループの Lambda 関数だけがストリームマネージャーのストリームと対話できます。クライアント認証が無効になっている場合、Greengrass コアで実行されているプロセス (Docker コンテナなど) は、ストリームマネージャーのストリームと対話できます。ビジネスケースで要求される場合にのみ、認証を無効にする必要があります。

クライアント認証モードを設定するには、STREAM_MANAGER_AUTHENTICATE_CLIENT パラメータを使用します。このパラメータは、 コンソールまたは AWS IoT Greengrass API から設定できます。変更は、グループがデプロイされた後に有効になります。

	有効	無効
パラメータ値	`true` (デフォルトおよび推奨)	`false`
許可されるクライアント	Greengrass グループのユーザー定義 Lambda 関数	Greengrass グループのユーザー定義 Lambda 関数 Greengrass コアデバイスで実行されているその他のプロセス

関連情報

- the section called “ストリームマネージャーの設定”
- the section called “ストリームを操作するために StreamManagerClient を使用する”
- the section called “AWS クラウド でサポートされている送信先のエクスポート設定”
- the section called “データストリームのエクスポート (コンソール)”
- the section called “データストリームのエクスポート (CLI)”

AWS IoT Greengrass ストリームマネージャーを設定する

ストリームマネージャーは、 AWS IoT Greengrass コアで IoT デバイスデータを保存、処理、エクスポートできます。ストリームマネージャーは、グループレベルのランタイム設定を設定するために使用するパラメータを提供します。これらの設定は、Greengrass コアのすべてのストリームに適用されます。 AWS IoT コンソールまたは AWS IoT Greengrass API を使用して、ストリームマネージャーの設定を行うことができます。変更は、グループがデプロイされた後に有効になります。

> **Note**
>
> ストリームマネージャーを設定したら、作成した IoT アプリケーションをデプロイし、Greengrass Core で実行してストリームマネージャーとやり取りすることができます。通常このような IoT アプリケーションは、ユーザー定義の Lambda 関数です。詳細については、「the section called “ストリームを操作するために StreamManagerClient を使用する”」を参照してください。

ストリームマネージャーのパラメータ

ストリームマネージャーは、グループレベルの設定を定義するための次のパラメータを提供します。すべてのパラメータは省略可能です。

[Storage directory] (ストレージディレクトリ)

パラメータ名: `STREAM_MANAGER_STORE_ROOT_DIR`

ストリームを保存するために使用されるローカルディレクトリの絶対パス。この値は、スラッシュ (`/data` など) で開始する必要があります。

ストリームデータのセキュリティ保護については、「the section called “ローカルデータセキュリティ”」を参照してください。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

Server port

パラメータ名: `STREAM_MANAGER_SERVER_PORT`

ストリームマネージャーとの通信に使用されるローカルポート番号。デフォルト: `8088`。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

クライアントを認証する

パラメータ名: `STREAM_MANAGER_AUTHENTICATE_CLIENT`

ストリームマネージャーと対話するためにクライアントを認証する必要があるかどうかを示します。クライアントとストリームマネージャー間のすべてのやり取りは、 AWS IoT Greengrass Core SDK によって制御されます。このパラメータは、ストリームを操作するために AWS IoT Greengrass Core SDK を呼び出すことができるクライアントを決定します。詳細については、「the section called “クライアント承認”」を参照してください。

有効な値は `true` または `false` です。デフォルトは `true` (推奨) です。

- `true`。Greengrass Lambda 関数のみをクライアントとして許可します。Lambda 関数クライアントは、内部 AWS IoT Greengrass コアプロトコルを使用して AWS IoT Greengrass Core SDK で認証します。
- `false`。 AWS IoT Greengrass コアで実行されるすべてのプロセスをクライアントに許可します。ビジネスケースで要求されない限り、`false` に設定しないでください。例えば、Core デバイスの非 Lambda プロセスがストリームマネージャと直接通信する必要がある (コアで動作する Docker コンテナのような) 場合にのみ、この値を `false` に設定します。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

最大帯域幅

パラメータ名: `STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH`

データのエクスポートに使用できる平均最大帯域幅 (キロビット/秒)。デフォルトでは、使用可能な帯域幅を無制限に使用することができます。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

スレッドプールサイズ

パラメータ名: `STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE`

データのエクスポートに使用できるアクティブなスレッドの最大数。デフォルト: 5。

最適なサイズは、ハードウェア、ストリームボリューム、予定されているエクスポートストリームの数によって異なります。エクスポート速度が遅い場合は、この設定を調整して、ハードウェアとビジネスケースに最適なサイズを見つけることができます。コアデバイスハードウェアの CPU とメモリは、制限要因です。開始するには、この値をデバイスのプロセッサコアの数と同じ値に設定してみてください。

ハードウェアがサポートできるサイズよりも大きいサイズを設定しないように注意してください。各ストリームはハードウェアリソースを消費するため、制約のあるデバイス上のエクスポートストリームの数を制限する必要があります。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

JVM の引数

パラメータ名: `JVM_ARGS`

起動時にストリームマネージャーに渡すカスタム Java 仮想マシン引数。複数の引数はスペースで区切る必要があります。

このパラメータは、JVM で使用されるデフォルト設定を上書きする必要がある場合にのみ使用します。例えば、大量のストリームをエクスポートする場合は、デフォルトのヒープサイズを大きくする必要があります。

最小 AWS IoT Greengrass Core バージョン: 1.10.0

[Read-only input file directories] (読み取り専用入力ファイルディレクトリ)

パラメータ名: `STREAM_MANAGER_READ_ONLY_DIRS`

入力ファイルを格納するルートファイルシステム外のディレクトリを指す絶対パスを、カンマで区切ったリスト。ストリームマネージャーは、読み込んだファイルを Amazon S3 にアップロードし、ディレクトリを読み取り専用でマウントします。Amazon S3 へのエクスポートの詳細については、「the section called “Amazon S3 オブジェクト”」を参照してください。

このパラメータは、次の条件に当てはまる場合にのみ使用してください。

- ストリームを介して Amazon S3 にエクスポートする入力ファイルディレクトリが、次のいずれかの場所にある。

  - ルートファイルシステム以外のパーティション。
  - ルートファイルシステム配下の `/tmp`。
- Greengrass グループのデフォルトのコンテナ化が [Greengrass container] (Greengrass コンテナ) である。

値の例: `/mnt/directory-1,/mnt/directory-2,/tmp`

最小 AWS IoT Greengrass Core バージョン: 1.11.0

マルチパートアップロードの最小サイズ

パラメータ名: `STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

Amazon S3 へのマルチパートアップロードにおけるパートの最小サイズ (バイト単位)。ストリームマネージャーはこの設定と入力ファイルのサイズを基に、マルチパート PUT リクエストのデータをバッチ処理する方法を決定します。デフォルトの最小値は `5242880` バイト (5 MB) です。

> **Note**
>
> ストリームマネージャーはストリームの `sizeThresholdForMultipartUploadBytes` プロパティを基に、Amazon S3 へのエクスポートをシングルアップロードで行うか、マルチパートアップロードで行うかを決定します。ユーザー定義の Lambda 関数は、Amazon S3 にエクスポートするストリームを作成する際にこのしきい値を設定します。デフォルトのしきい値は 5 MB です。

最小 AWS IoT Greengrass Core バージョン: 1.11.0

ストリームマネージャーの設定 (コンソール)

AWS IoT コンソールは、以下の管理タスクに使用できます。

- ストリームマネージャーが有効になっているかどうかを確認する
- グループ作成時のストリームマネージャーを有効または無効にする
- 既存のグループのストリームマネージャーを有効または無効にする
- ストリームマネージャー設定の変更

変更は、Greengrass グループがデプロイされた後に有効になります。ストリームマネージャーとの通信に使用する Lambda 関数が格納された Greengrass グループを、チュートリアル形式でデプロイする方法については、「the section called “データストリームのエクスポート (コンソール)”」を参照してください。

> **Note**
>
> コンソールを使用してストリームマネージャーを有効にし、グループをデプロイすると、ストリームマネージャーのメモリサイズはデフォルトで 4194304 KB (4 GB) に設定されます。メモリのサイズは 128000 KB 以上に設定することをお勧めします。

ストリームマネージャーが有効になっているかどうかを確認するには (コンソール)

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. [System Lambda functions] (システム Lambda 関数) で、[Stream manager] (ストリームマネージャー)、[Edit] (編集) の順に選択します。
5. 有効または無効のステータスを確認します。設定されているカスタムストリームマネージャー設定も表示されます。

グループの作成中にストリームマネージャーを有効または無効にするには (コンソール)

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [グループを作成] を選択します。次のページで選択すると、グループのストリームマネージャーの設定方法が決まります。
3. [Name your Group] (グループに名前を付ける) から [Greengrass core] (Greengrass コア) ページを選択します。

4. [グループの作成] を選択してください。
5. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択し、[Stream manager] (ストリームマネージャー)、[Edit] (編集) の順に選択します。

   - デフォルト設定でストリームマネージャーを有効にするには、[Use defaults] (デフォルトを使用) を選択します。

   - カスタム設定でストリームマネージャーを有効にするには、[Customize settings (設定をカスタマイズ)] を選択します。

     1. [Configure stream manager] (ストリームマネージャーの設定) ページで、[Enable with custom settings] (カスタム設定で有効にする) を選択します。
     2. [Custom settings (カスタム設定)] で、ストリームマネージャーパラメータの値を入力します。詳細については、「the section called “ストリームマネージャーのパラメータ”」を参照してください。がデフォルト値を使用できるようにするには、フィールド AWS IoT Greengrass を空のままにします。

   - ストリームマネージャーを無効にするには、[Disable] (無効化) を選択します。

     1. [Configure stream manager (ストリームマネージャーの設定)] ページで、[Disable (無効にする)] を選択します。

6. [Save] を選択します。
7. 残りのページに進み、グループを作成します。
8. [Client devices] (クライアントデバイス) ページで、セキュリティリソースをダウンロードし、情報を確認して、[Finish] (完了) を選択します。

> **Note**
>
> ストリームマネージャーが有効になっている場合、グループをデプロイする前に、コアデバイスに Java 8 ランタイムをインストールする必要があります。

既存のグループのストリームマネージャーを有効または無効にするには (コンソール)

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. [System Lambda functions] (システム Lambda 関数) で、[Stream manager] (ストリームマネージャー)、[Edit] (編集) の順に選択します。
5. 有効または無効のステータスを確認します。設定されているカスタムストリームマネージャー設定も表示されます。

ストリームマネージャーの設定を変更するには (コンソール)

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. [Lambda functions] (Lambda 関数) タブを選択します。
4. [System Lambda functions] (システム Lambda 関数) で、[Stream manager] (ストリームマネージャー)、[Edit] (編集) の順に選択します。
5. 有効または無効のステータスを確認します。設定されているカスタムストリームマネージャー設定も表示されます。
6. [Save] を選択します。

ストリームマネージャーの設定 (CLI)

で AWS CLI、システム `GGStreamManager` Lambda 関数を使用してストリームマネージャーを設定します。System Lambda 関数は AWS IoT Greengrass Core ソフトウェアのコンポーネントです。ストリームマネージャーおよびその他のシステム Lambda 関数の一部では、Greengrass の各種機能に対応する Greengrass グループの `Function` オブジェクトや `FunctionDefinitionVersion` オブジェクトを管理することで、Greengrass の機能を設定できます。詳細については、「the section called “グループオブジェクトモデルの概要”」を参照してください。

API は以下の管理タスクに使用できます。このセクションの例では、 の使用方法を示していますが AWS CLI、 AWS IoT Greengrass API を直接呼び出すことも、 AWS SDK を使用することもできます。

- ストリームマネージャーが有効になっているかどうかを確認する
- ストリームマネージャーの有効化、無効化、設定

変更は、グループがデプロイされた後に有効になります。ストリームマネージャーとのやり取りに使用する Lambda 関数が格納された Greengrass グループを、チュートリアル形式でデプロイする方法については、「the section called “データストリームのエクスポート (CLI)”」を参照してください。

> **Tip**
>
> ストリームマネージャーが有効で実行されているかどうかを Core デバイスから確認するには、そのデバイスのターミナルで次のコマンドを実行します。
>
> ```
> ps aux | grep -i 'streammanager'
> ```

ストリームマネージャーが有効になっているかどうかを確認するには (CLI)

デプロイされた関数定義バージョンにシステム `GGStreamManager` Lambda 関数が含まれている場合、ストリームマネージャーが有効になります。確認するには、次の手順を実行します。

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

> **Note**
>
> これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

2. 出力からターゲットグループの `Id` 値と `LatestVersion` 値をコピーします。
3. 最新のグループバージョンを取得します。
   - *`group-id`* は、コピーした `Id` に置き換えます。
   - *`latest-group-version-id`* は、コピーした `LatestVersion` に置き換えます。

```
aws greengrass get-group-version \
--group-id group-id \
--group-version-id latest-group-version-id
```

4. 出力の `FunctionDefinitionVersionArn` で、関数定義の ID と関数定義のバージョンを取得します。
   - 関数定義 ID は、Amazon リソースネーム (ARN) の `functions` セグメントに続く GUID です。
   - 関数定義のバージョン ID は、ARN の `versions` セグメントに続く GUID です。

```
arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/
functions/function-definition-id/versions/function-definition-version-id
```

5. 関数定義バージョンの ID。
   - *`function-definition-id`* を関数定義 ID に置き換えます。
   - *`function-definition-version-id`* ID を関数定義バージョン ID に置き換えます。

```
aws greengrass get-function-definition-version \
--function-definition-id function-definition-id \
--function-definition-version-id function-definition-version-id
```

出力内の functions 配列に GGStreamManager 関数が含まれている場合、ストリームマネージャーが有効になります。関数に定義された環境変数は、ストリームマネージャーのカスタム設定を表します。

ストリームマネージャー (CLI) を有効化、無効化、設定するには

で AWS CLI、システム GGStreamManager Lambda 関数を使用してストリームマネージャーを設定します。変更は、グループをデプロイした後に有効になります。

- ストリームマネージャーを有効にするには、関数定義バージョンの functions 配列に GGStreamManager を含めます。カスタム設定を設定するには、対応するストリームマネージャーのパラメータの環境変数を定義します。
- ストリームマネージャーを無効にするには、関数定義バージョンの functions 配列から GGStreamManager を削除します。

デフォルト設定のストリームマネージャー

次の設定例では、デフォルト設定でストリームマネージャーを有効にします。任意の関数 ID を streamManager に設定します。

```
{
    "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
    "FunctionConfiguration": {
        "MemorySize": 4194304,
        "Pinned": true,
        "Timeout": 3
    },
    "Id": "streamManager"
}
```

> **Note**
>
> FunctionConfiguration プロパティについては、次の点に留意してください。
>
> - MemorySize はデフォルト設定では 4194304 KB (4 GB) に設定されています。この値は、いつでも変更できます。MemorySize は 128000 KB 以上に設定することをお勧めします。
> - Pinned を true に設定する必要があります。
> - Timeout は関数定義バージョンで必要ですが、GGStreamManager は使用しません。

カスタム設定のストリームマネージャー

以下の設定例では、ストレージディレクトリ、サーバーポート、スレッドプールサイズの各種パラメータにカスタム値を使用した上で、ストリームマネージャーを有効にします。

```
{
    "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
    "FunctionConfiguration": {
        "Environment": {
            "Variables": {
                "STREAM_MANAGER_STORE_ROOT_DIR": "/data",
                "STREAM_MANAGER_SERVER_PORT": "1234",
                "STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE": "4"
            }
        },
        "MemorySize": 4194304,
        "Pinned": true,
        "Timeout": 3
    },
    "Id": "streamManager"
}
```

AWS IoT Greengrass は、環境変数として指定されていないストリームマネージャーパラメータのデフォルト値を使用します。

ストリームマネージャーに Amazon S3 のエクスポート用カスタム設定を適用する

以下の設定例では、アップロードディレクトリやマルチポートアップロードの最小サイズにカスタム値を使用した上で、ストリームマネージャーを有効にします。

```
{
    "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
    "FunctionConfiguration": {
        "Environment": {
            "Variables": {
                "STREAM_MANAGER_READ_ONLY_DIRS": "/mnt/directory-1,/mnt/
directory-2,/tmp",

 "STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES":
 "10485760"
            }
        },
        "MemorySize": 4194304,
```

```
        "Pinned": true,
        "Timeout": 3
    },
    "Id": "streamManager"
}
```

ストリームマネージャー (CLI) を有効化、無効化、設定するには

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [デプロイ] タブに表示されます。

2. 出力からターゲットグループの `Id` 値と `LatestVersion` 値をコピーします。
3. 最新のグループバージョンを取得します。

   - *`group-id`* は、コピーした `Id` に置き換えます。
   - *`latest-group-version-id`* は、コピーした `LatestVersion` に置き換えます。

   ```
   aws greengrass get-group-version \
   --group-id group-id \
   --group-version-id latest-group-version-id
   ```

4. CoreDefinitionVersionArn と、その他の出力されたすべてのバージョン ARN をコピーします。ただし、FunctionDefinitionVersionArn は対象外です。上記の値は、後ほどグループバージョンを作成する時に使用します。
5. 出力の FunctionDefinitionVersionArn から、関数定義の ID をコピーします。ID は、次の例に示すように、ARN の functions セグメントに続く GUID です。

```
arn:aws:greengrass:us-west-2:123456789012:/greengrass/
definition/functions/bcfc6b49-beb0-4396-b703-6dEXAMPLEcu5/
versions/0f7337b4-922b-45c5-856f-1aEXAMPLEsf6
```

> **Note**
>
> または、create-function-definition コマンドを実行して関数定義を作成し、出力から ID をコピーすることもできます。

6. 関数定義に関数定義バージョンを追加します。

   - *function-definition-id* は、コピーした関数定義の Id に置き換えます。
   - functions 配列には、Greengrass Core で使用できるようにする他のすべての関数を含めます。get-function-definition-version コマンドを使用して、既存の関数のリストを取得できます。

デフォルト設定でストリームマネージャーを有効にする

次の例では、functions 配列に GGStreamManager 関数を含めることで、ストリームマネージャーを有効にします。この例では、ストリームマネージャーのパラメータにデフォルト値を使用します。

```
aws greengrass create-function-definition-version \
--function-definition-id function-definition-id \
--functions '[
        {
            "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
            "FunctionConfiguration": {
                "MemorySize":  4194304,
                "Pinned": true,
                "Timeout": 3
```

```
            },
            "Id": "streamManager"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:MyLambdaFunction:MyAlias",
            "FunctionConfiguration": {
                "Executable": "myLambdaFunction.function_handler",
                "MemorySize": 16000,
                "Pinned": true,
                "Timeout": 5
            },
            "Id": "myLambdaFunction"
        },
        ... more user-defined functions
    ]
}'
```

> **Note**
>
> 例の myLambdaFunction 関数は、ユーザー定義 Lambda 関数の 1 つを表します。

カスタム設定でストリームマネージャーを有効にする

次の例では、functions 配列に GGStreamManager 関数を含めることで、ストリームマネージャーを有効にします。デフォルト値を変更しない限り、すべてのストリームマネージャー設定はオプションです。この例では、環境変数を使用してカスタム値を設定する方法を示します。

```
aws greengrass create-function-definition-version \
--function-definition-id function-definition-id \
--functions '[
        {
            "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
            "FunctionConfiguration": {
                "Environment": {
                    "Variables": {
                        "STREAM_MANAGER_STORE_ROOT_DIR": "/data",
                        "STREAM_MANAGER_SERVER_PORT": "1234",
                        "STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE": "4"
```

```
                    }
                },
                "MemorySize":  4194304,
                "Pinned": true,
                "Timeout": 3
            },
            "Id": "streamManager"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:MyLambdaFunction:MyAlias",
            "FunctionConfiguration": {
                "Executable": "myLambdaFunction.function_handler",
                "MemorySize": 16000,
                "Pinned": true,
                "Timeout": 5
            },
            "Id": "myLambdaFunction"
        },
        ... more user-defined functions
    ]
}'
```

> **Note**
>
> FunctionConfiguration プロパティについては、次の点に留意してください。
>
> - MemorySize はデフォルト設定では 4194304 KB (4 GB) に設定されています。この値は、いつでも変更できます。MemorySize は 128000 KB 以上に設定することをお勧めします。
> - Pinned を true に設定する必要があります。
> - Timeout は関数定義バージョンで必要ですが、GGStreamManager は使用しません。

ストリームマネージャーを無効にする

次の例では、ストリームマネージャーを無効にする GGStreamManager 関数を省略しています。

```
aws greengrass create-function-definition-version \
```

```
--function-definition-id function-definition-id \
--functions '[
        {
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:MyLambdaFunction:MyAlias",
            "FunctionConfiguration": {
                "Executable": "myLambdaFunction.function_handler",
                "MemorySize": 16000,
                "Pinned": true,
                "Timeout": 5
            },
            "Id": "myLambdaFunction"
        },
        ... more user-defined functions
    ]
}'
```

> **Note**
>
> Lambda 関数をデプロイしない場合は、関数定義バージョンを完全に省略できます。

7. 出力から関数定義バージョンの `Arn` をコピーします。
8. システムの Lambda 関数が含まれているグループバージョンを作成します。

   - *`group-id`* は、グループの `Id` に置き換えます。
   - *`core-definition-version-arn`* は、最新のグループバージョンからコピーした `CoreDefinitionVersionArn` に置き換えます。
   - *`function-definition-version-arn`* は、コピーした新しい関数定義バージョンの `Arn` に置き換えます。
   - 他のグループコンポーネントの ARN (`SubscriptionDefinitionVersionArn` や `DeviceDefinitionVersionArn` など) は、最新のグループバージョンからコピーしたものに置き換えます。
   - 使用されていないパラメータをすべて削除します。例えば、グループバージョンにリソースがない場合は、`--resource-definition-version-arn` を削除します。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
```

```
--function-definition-version-arn function-definition-version-arn \
--device-definition-version-arn device-definition-version-arn \
--logger-definition-version-arn logger-definition-version-arn \
--resource-definition-version-arn resource-definition-version-arn \
--subscription-definition-version-arn subscription-definition-version-arn
```

9. 出力から Version をコピーします。これは新しいグループバージョンの ID です。
10. 新しいグループバージョンでグループをデプロイします。

    - *group-id* は、コピーしたグループの Id に置き換えます。
    - *group-version-id* は、コピーした新しいグループバージョンの Version に置き換えます。

```
aws greengrass create-deployment \
--group-id group-id \
--group-version-id group-version-id \
--deployment-type NewDeployment
```

ストリームマネージャーの設定を後でもう一度編集する場合は、この手順に沿って行ってください。これらの設定を更新するには、更新された設定の GGStreamManager 関数を含む関数定義バージョンを作成します。グループのバージョンは、コアにデプロイするすべてのコンポーネントバージョン ARN を参照する必要があります。変更は、グループがデプロイされた後に有効になります。

関連情報

- データストリームの管理
- the section called “ストリームを操作するために StreamManagerClient を使用する”
- the section called “AWS クラウド でサポートされている送信先のエクスポート設定”
- the section called “データストリームのエクスポート (コンソール)”
- the section called “データストリームのエクスポート (CLI)”

ストリームを操作するために StreamManagerClient を使用する

AWS IoT Greengrass コアで実行されるユーザー定義の Lambda 関数は、 AWS IoT Greengrass Core SDK の `StreamManagerClient` オブジェクトを使用してストリームマネージャーでストリームを作成し、ストリームとやり取りできます。Lambda 関数は、ストリームを作成するときに、ストリームの送信 AWS クラウド 先、優先順位付け、その他のエクスポートポリシーとデータ保持ポリシーを定義します。また、ストリームマネージャーにデータを送信するために、データをストリームに追加します。ストリームにエクスポート先が定義されている場合、ストリームマネージャーは自動的にストリームをエクスポートします。

> **Note**
>
> 通常、ストリームマネージャーのクライアントはユーザー定義の Lambda 関数です。ビジネスケースで必要な場合は、Greengrass コア (Docker コンテナなど) で実行されている非 Lambda プロセスがストリームマネージャーと対話できるようにすることもできます。詳細については、「the section called “クライアント承認”」を参照してください。

このトピックのスニペットは、クライアントが `StreamManagerClient` 手法を使用してストリームを操作する方法を示しています。メソッドとその引数の実装の詳細については、各スニペットの下に記載されている SDK リファレンスへのリンクを使用してください。完全な Python Lambda 関数も使用可能なチュートリアルについては、the section called “データストリームのエクスポート (コンソール)” または the section called “データストリームのエクスポート (CLI)” を参照してください。

Lambda 関数では、関数ハンドラの外部で `StreamManagerClient` をインスタンス化する必要があります。ハンドラでインスタンス化されると、関数は呼び出されるたびに `client` およびストリームマネージャへの接続を作成します。

> **Note**
>
> ハンドラで `StreamManagerClient` のインスタンス化を行う場合は、`client` が作業を完了したときに、`close()` メソッドを明示的に呼び出す必要があります。それ以外の場合、`client` は接続を開いたままにし、スクリプトが終了するまで別のスレッドを実行します。

`StreamManagerClient` では次の操作がサポートされています。

- the section called “メッセージストリームの作成”
- the section called “メッセージの追加”
- the section called “メッセージの読み取り”
- the section called “ストリームの一覧表示”
- the section called “メッセージストリームの説明”
- the section called “メッセージストリームの更新”
- the section called “メッセージストリームの削除”

メッセージストリームの作成

ストリームを作成するには、ユーザー定義の Lambda 関数で create メソッドを呼び出し、`MessageStreamDefinition` オブジェクトを渡します。このオブジェクトによって、ストリームの一意の名前を指定すると共に、最大ストリームサイズに達したときにストリームマネージャーが新しいデータを処理する方法を定義します。`MessageStreamDefinition` とそのデータ型 (`ExportDefinition`、`StrategyOnFull`、`Persistence` など) を使用して、他のストリームプロパティを定義できます。具体的には次のとおりです。

- 自動エクスポートのターゲット AWS IoT Analytics、Kinesis Data Streams AWS IoT SiteWise、Amazon S3 の送信先。詳細については、「the section called “AWS クラウド でサポートされている送信先のエクスポート設定”」を参照してください。
- エクスポートの優先度。ストリームマネージャーは、プライオリティの低いストリームよりも先にプライオリティの高いストリームをエクスポートします。
- 、Kinesis Data Streams AWS IoT Analytics、および送信先の最大バッチサイズと AWS IoT SiteWise バッチ間隔。ストリームマネージャーは、いずれかの条件が満たされたときにメッセージをエクスポートします。
- 有効期限 (TTL) ストリームデータが処理可能であることを保証する時間。この期間内にデータを消費できることを確認する必要があります。これは削除ポリシーではありません。TTL 期間の直後にデータが削除されない場合があります。
- ストリームの永続性。ストリームをファイルシステムに保存して、コアを再起動してもデータを保持するか、ストリームをメモリに保存するかを選択します。
- 開始するシーケンス番号。エクスポートの開始メッセージとして使用するメッセージのシーケンス番号を指定します。

`MessageStreamDefinition` の詳細については、対象言語の SDK リファレンスを参照してください。

- Java SDK の MessageStreamDefinition
- Node.js SDK の MessageStreamDefinition
- Python SDK の MessageStreamDefinition

> **Note**
>
> `StreamManagerClient` は、ストリームを HTTP サーバーにエクスポートするため、ターゲットの送信先も提供します。このターゲットは、テストのみを目的としています。また、安定しておらず、実稼働環境での使用はサポートされていません。

ストリームが作成されると、Lambda 関数はストリームにメッセージを追加してエクスポート用データを送信します。また、ローカル処理用にストリームからメッセージを読み取ります。作成するストリームの数は、ハードウェアの機能とビジネスケースによって異なります。1 つの戦略は、 AWS IoT Analytics または Kinesis データストリームの各ターゲットチャネルにストリームを作成することですが、ストリームに複数のターゲットを定義できます。ストリームは寿命に耐久性があります。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

> **Note**
>
> AWS IoT SiteWise または Amazon S3 エクスポート先を使用してストリームを作成する場合、次の要件があります。
>
> - 最小 AWS IoT Greengrass Core バージョン: 1.11.0
> - AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.6.0 | Java: 1.5.0 | Node.js: 1.7.0

例

次のスニペットでは、StreamName という名前のストリームが作成されます。MessageStreamDefinition のストリームプロパティとと下位のデータタイプを定義します。

Python

```
client = StreamManagerClient()

try:
    client.create_message_stream(MessageStreamDefinition(
        name="StreamName",  # Required.
        max_size=268435456,  # Default is 256 MB.
        stream_segment_size=16777216,  # Default is 16 MB.
        time_to_live_millis=None,  # By default, no TTL is enabled.
        strategy_on_full=StrategyOnFull.OverwriteOldestData,  # Required.
        persistence=Persistence.File,  # Default is File.
        flush_on_write=False,  # Default is false.
        export_definition=ExportDefinition(  # Optional. Choose where/how the stream
 is exported to the AWS ####.
            kinesis=None,
            iot_analytics=None,
            iot_sitewise=None,
            s3_task_executor=None
        )
    ))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: create_message_stream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    client.createMessageStream(
            new MessageStreamDefinition()
                    .withName("StreamName") // Required.
```

```
                    .withMaxSize(268435456L)  // Default is 256 MB.
                    .withStreamSegmentSize(16777216L)  // Default is 16 MB.
                    .withTimeToLiveMillis(null)  // By default, no TTL is enabled.
                    .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)  //
 Required.
                    .withPersistence(Persistence.File)  // Default is File.
                    .withFlushOnWrite(false)  // Default is false.
                    .withExportDefinition(  // Optional. Choose where/how the stream
 is exported to the AWS ####.
                            new ExportDefinition()
                                    .withKinesis(null)
                                    .withIotAnalytics(null)
                                    .withIotSitewise(null)
                                    .withS3TaskExecutor(null)
                    )

    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: [createMessageStream](#) | [MessageStreamDefinition](#)

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        await client.createMessageStream(
            new MessageStreamDefinition()
                .withName("StreamName") // Required.
                .withMaxSize(268435456)  // Default is 256 MB.
                .withStreamSegmentSize(16777216)  // Default is 16 MB.
                .withTimeToLiveMillis(null)  // By default, no TTL is enabled.
                .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)  //
 Required.
                .withPersistence(Persistence.File)  // Default is File.
                .withFlushOnWrite(false)  // Default is false.
                .withExportDefinition(  // Optional. Choose where/how the stream is
 exported to the AWS ####.
                    new ExportDefinition()
                        .withKinesis(null)
                        .withIotAnalytics(null)
                        .withIotSitewise(null)
```

```
                        .withS3TaskExecutor(null)
                )
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: createMessageStream | MessageStreamDefinition

エクスポート先設定の詳細については、「the section called “AWS クラウド でサポートされている送信先のエクスポート設定”」を参照してください。

メッセージの追加

ストリームマネージャーにエクスポート用データを送信するには、Lambda 関数でそのデータをターゲットストリームに追加します。このエクスポート先によって、このメソッドに渡すデータ型が決まります。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

> **Note**
>
> AWS IoT SiteWise または Amazon S3 のエクスポート先でメッセージを追加する場合、次の要件があります。
>
> - 最小 AWS IoT Greengrass Core バージョン: 1.11.0

- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.6.0 | Java: 1.5.0 | Node.js: 1.7.0

例

AWS IoT Analytics または Kinesis Data Streams のエクスポート先

次のスニペットは、`StreamName` という名前のストリームにメッセージを追加します。 AWS IoT Analytics または Kinesis Data Streams の送信先の場合、Lambda 関数はデータの BLOB を追加します。

このスニペットには次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

Python

```
client = StreamManagerClient()

try:
    sequence_number = client.append_message(stream_name="StreamName",
 data=b'Arbitrary bytes data')
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    long sequenceNumber = client.appendMessage("StreamName", "Arbitrary byte
 array".getBytes());
} catch (StreamManagerException e) {
```

```
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const sequenceNumber = await client.appendMessage("StreamName",
 Buffer.from("Arbitrary byte array"));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage

AWS IoT SiteWise エクスポート先

次のスニペットは、`StreamName` という名前のストリームにメッセージを追加します。 AWS IoT SiteWise 送信先の場合、Lambda 関数はシリアル化された`PutAssetPropertyValueEntry`オブジェクトを追加します。詳細については、「the section called “への エクスポート AWS IoT SiteWise”」を参照してください。

> **Note**
>
> にデータを送信する場合 AWS IoT SiteWise、データは `BatchPutAssetPropertyValue`アクションの要件を満たしている必要があります。詳細については、「AWS IoT SiteWise API リファレンス」の「BatchPutAssetPropertyValue」を参照してください。

このスニペットには次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.11.0

- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.6.0 | Java: 1.5.0 | Node.js: 1.7.0

Python

```
client = StreamManagerClient()

try:
    # SiteWise requires unique timestamps in all messages. Add some randomness to
 time and offset.

    # Note: To create a new asset property data, you should use the classes defined
 in the
    # greengrasssdk.stream_manager module.

    time_in_nanos = TimeInNanos(
        time_in_seconds=calendar.timegm(time.gmtime()) - random.randint(0, 60),
 offset_in_nanos=random.randint(0, 10000)
    )
    variant = Variant(double_value=random.random())
    asset = [AssetPropertyValue(value=variant, quality=Quality.GOOD,
 timestamp=time_in_nanos)]
    putAssetPropertyValueEntry =
 PutAssetPropertyValueEntry(entry_id=str(uuid.uuid4()),
 property_alias="PropertyAlias", property_values=asset)
    sequence_number = client.append_message(stream_name="StreamName",
 data=Util.validate_and_serialize_to_json_bytes(putAssetPropertyValueEntry))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message | PutAssetPropertyValueEntry

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    Random rand = new Random();
    // Note: To create a new asset property data, you should use the classes defined
 in the
    // com.amazonaws.greengrass.streammanager.model.sitewise package.
```

```
    List<AssetPropertyValue> entries = new ArrayList<>() ;

    // IoTSiteWise requires unique timestamps in all messages. Add some randomness
 to time and offset.
    final int maxTimeRandomness = 60;
    final int maxOffsetRandomness = 10000;
    double randomValue = rand.nextDouble();
    TimeInNanos timestamp = new TimeInNanos()
            .withTimeInSeconds(Instant.now().getEpochSecond() -
 rand.nextInt(maxTimeRandomness))
            .withOffsetInNanos((long) (rand.nextInt(maxOffsetRandomness)));
    AssetPropertyValue entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);
    entries.add(entry);

    PutAssetPropertyValueEntry putAssetPropertyValueEntry = new
 PutAssetPropertyValueEntry()
            .withEntryId(UUID.randomUUID().toString())
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues(entries);
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage | PutAssetPropertyValueEntry

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const maxTimeRandomness = 60;
        const maxOffsetRandomness = 10000;
        const randomValue = Math.random();
        // Note: To create a new asset property data, you should use the classes
 defined in the
        // aws-greengrass-core-sdk StreamManager module.
        const timestamp = new TimeInNanos()
            .withTimeInSeconds(Math.round(Date.now() / 1000) -
 Math.floor(Math.random() - maxTimeRandomness))
```

```
            .withOffsetInNanos(Math.floor(Math.random() * maxOffsetRandomness));
        const entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);

        const putAssetPropertyValueEntry =  new PutAssetPropertyValueEntry()
            .withEntryId(`${ENTRY_ID_PREFIX}${i}`)
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues([entry]);
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage | PutAssetPropertyValueEntry

Amazon S3 のエクスポート先

次のスニペットは、`StreamName` という名前のストリームにエクスポートタスクを追加します。Amazon S3 を送信先とする場合、Lambda 関数は、シリアル化された `S3ExportTaskDefinition` オブジェクトを追加します。これには、ソース入力ファイルとターゲット Amazon S3 オブジェクトに関する情報が含まれています。指定されたオブジェクトが存在しない場合、ストリームマネージャーがそのオブジェクトを作成します。詳細については、「the section called “Amazon S3 へのエクスポート”」を参照してください。

このスニペットには次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.11.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.6.0 | Java: 1.5.0 | Node.js: 1.7.0

Python

```
client = StreamManagerClient()
```

```
try:
    # Append an Amazon S3 Task definition and print the sequence number.
    s3_export_task_definition = S3ExportTaskDefinition(input_url="URLToFile",
 bucket="BucketName", key="KeyName")
    sequence_number = client.append_message(stream_name="StreamName",
 data=Util.validate_and_serialize_to_json_bytes(s3_export_task_definition))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: append_message | S3ExportTaskDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    // Append an Amazon S3 export task definition and print the sequence number.
    S3ExportTaskDefinition s3ExportTaskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(s3ExportTaskDefinition));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: appendMessage | S3ExportTaskDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
     // Append an Amazon S3 export task definition and print the sequence number.
     const taskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(taskDefinition)));
```

```
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: appendMessage | S3ExportTaskDefinition

メッセージの読み取り

ストリームのメッセージを読み取ります。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

例

次のスニペットは、`StreamName` という名前のストリームからメッセージを読み取ります。読み取りメソッドは、読み取りを開始するシーケンス番号、読み取る最小値と最大値、メッセージを読み取るためのタイムアウトを指定するオプションの `ReadMessagesOptions` オブジェクトを受け取ります。

Python

```
client = StreamManagerClient()

try:
    message_list = client.read_messages(
        stream_name="StreamName",
        # By default, if no options are specified, it tries to read one message from
 the beginning of the stream.
        options=ReadMessagesOptions(
```

```
            desired_start_sequence_number=100,
            # Try to read from sequence number 100 or greater. By default, this is
 0.
            min_message_count=10,
            # Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is raised. By default, this is 1.
            max_message_count=100,  # Accept up to 100 messages. By default this is
 1.
            read_timeout_millis=5000
            # Try to wait at most 5 seconds for the min_messsage_count to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
        )
    )
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: read_messages | ReadMessagesOptions

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    List<Message> messages = client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                    // Try to read from sequence number 100 or greater. By default
 this is 0.
                    .withDesiredStartSequenceNumber(100L)
                    // Try to read 10 messages. If 10 messages are not available,
 then NotEnoughMessagesException is raised. By default, this is 1.
                    .withMinMessageCount(10L)
                    // Accept up to 100 messages. By default this is 1.
                    .withMaxMessageCount(100L)
                    // Try to wait at most 5 seconds for the min_messsage_count to
 be fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                    .withReadTimeoutMillis(Duration.ofSeconds(5L).toMillis())
    );
```

```
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: readMessages | ReadMessagesOptions

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messages = await client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                // Try to read from sequence number 100 or greater. By default this
 is 0.
                .withDesiredStartSequenceNumber(100)
                // Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is thrown. By default, this is 1.
                .withMinMessageCount(10)
                // Accept up to 100 messages. By default this is 1.
                .withMaxMessageCount(100)
                // Try to wait at most 5 seconds for the minMessageCount to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                .withReadTimeoutMillis(5 * 1000)
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: readMessages | ReadMessagesOptions

ストリームの一覧表示

ストリームマネージャーでストリームのリストを取得します。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

例

次のスニペットは、ストリームマネージャーのストリームのリストを (名前で) 取得します。

Python

```
client = StreamManagerClient()

try:
    stream_names = client.list_streams()
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: list_streams

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    List<String> streamNames = client.listStreams();
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: listStreams

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const streams = await client.listStreams();
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: listStreams

メッセージストリームの説明

ストリーム定義、サイズ、エクスポート状態を含め、ストリームに関するメタデータを取得します。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

例

次のスニペットは、ストリームの定義、サイズ、エクスポーターのステータスなど、`StreamName` という名前のストリームに関するメタデータを取得します。

Python

```
client = StreamManagerClient()
```

```
try:
    stream_description = client.describe_message_stream(stream_name="StreamName")
    if stream_description.export_statuses[0].error_message:
        # The last export of export destination 0 failed with some error
        # Here is the last sequence number that was successfully exported
        stream_description.export_statuses[0].last_exported_sequence_number

    if (stream_description.storage_status.newest_sequence_number >
            stream_description.export_statuses[0].last_exported_sequence_number):
        pass
        # The end of the stream is ahead of the last exported sequence number
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: describe_message_stream

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    MessageStreamInfo description = client.describeMessageStream("StreamName");
    String lastErrorMessage =
 description.getExportStatuses().get(0).getErrorMessage();
    if (lastErrorMessage != null && !lastErrorMessage.equals("")) {
        // The last export of export destination 0 failed with some error.
        // Here is the last sequence number that was successfully exported.
        description.getExportStatuses().get(0).getLastExportedSequenceNumber();
    }

    if (description.getStorageStatus().getNewestSequenceNumber() >
            description.getExportStatuses().get(0).getLastExportedSequenceNumber())
 {
        // The end of the stream is ahead of the last exported sequence number.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: describeMessageStream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const description = await client.describeMessageStream("StreamName");
        const lastErrorMessage = description.exportStatuses[0].errorMessage;
        if (lastErrorMessage) {
            // The last export of export destination 0 failed with some error.
            // Here is the last sequence number that was successfully exported.
            description.exportStatuses[0].lastExportedSequenceNumber;
        }

        if (description.storageStatus.newestSequenceNumber >
            description.exportStatuses[0].lastExportedSequenceNumber) {
            // The end of the stream is ahead of the last exported sequence number.
        }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: describeMessageStream

メッセージストリームの更新

既存ストリームのプロパティを更新します。ストリームを作成した後に要件が変更された場合、ストリームの更新が必要となる場合があります。以下に例を示します。

- AWS クラウド 送信先の新しいエクスポート設定を追加します。
- ストリームの最大サイズを増やして、データのエクスポートまたは保持方法を変更します。例えば、ストリームサイズとフル設定の戦略を組み合わせると、ストリームマネージャーが処理する前に、データが削除または拒否されることがあります。
- エクスポートを一時停止して再開します。例えば、エクスポートタスクが長時間実行されているため、アップロードするデータ量を抑える場合などです。

Lambda 関数は、次の高レベルプロセスに従ってストリームを更新します。

1. ストリームを説明する情報を取得します。
2. 対応する `MessageStreamDefinition` と下位オブジェクトのターゲットプロパティを更新します。
3. 更新済み `MessageStreamDefinition` として渡します。更新済みストリームの完全なオブジェクト定義を必ず含めてください。未定義のプロパティはデフォルト値に戻ります。

   エクスポートで開始メッセージとして使用するメッセージのシーケンス番号を指定できます。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.11.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.6.0 | Java: 1.5.0 | Node.js: 1.7.0

例

次のスニペットは、`StreamName` という名前のストリームを更新します。Kinesis データストリームにエクスポートするストリームの複数プロパティを更新します。

Python

```
client = StreamManagerClient()

try:
    message_stream_info = client.describe_message_stream(STREAM_NAME)
    message_stream_info.definition.max_size=536870912
    message_stream_info.definition.stream_segment_size=33554432
    message_stream_info.definition.time_to_live_millis=3600000
    message_stream_info.definition.strategy_on_full=StrategyOnFull.RejectNewData
    message_stream_info.definition.persistence=Persistence.Memory
    message_stream_info.definition.flush_on_write=False
    message_stream_info.definition.export_definition.kinesis=
        [KinesisConfig(
            # Updating Export definition to add a Kinesis Stream configuration.
            identifier=str(uuid.uuid4()), kinesis_stream_name=str(uuid.uuid4()))]
    client.update_message_stream(message_stream_info.definition)
except StreamManagerException:
```

```
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: updateMessageStream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    MessageStreamInfo messageStreamInfo = client.describeMessageStream(STREAM_NAME);
    // Update the message stream with new values.
    client.updateMessageStream(
        messageStreamInfo.getDefinition()
            .withStrategyOnFull(StrategyOnFull.RejectNewData) // Required. Updating
 Strategy on full to reject new data.
            // Max Size update should be greater than initial Max Size defined in
 Create Message Stream request
            .withMaxSize(536870912L) // Update Max Size to 512 MB.
            .withStreamSegmentSize(33554432L) // Update Segment Size to 32 MB.
            .withFlushOnWrite(true) // Update flush on write to true.
            .withPersistence(Persistence.Memory) // Update the persistence to
 Memory.
            .withTimeToLiveMillis(3600000L)  // Update TTL to 1 hour.
            .withExportDefinition(
                // Optional. Choose where/how the stream is exported to the AWS ###
#.
                messageStreamInfo.getDefinition().getExportDefinition().
                    // Updating Export definition to add a Kinesis Stream
 configuration.
                    .withKinesis(new ArrayList<KinesisConfig>() {{
                        add(new KinesisConfig()
                            .withIdentifier(EXPORT_IDENTIFIER)
                            .withKinesisStreamName("test"));
                        }})
            );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: update_message_stream | MessageStreamDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messageStreamInfo = await c.describeMessageStream(STREAM_NAME);
        await client.updateMessageStream(
            messageStreamInfo.definition
                // Max Size update should be greater than initial Max Size defined
 in Create Message Stream request
                .withMaxSize(536870912)  // Default is 256 MB. Updating Max Size to
 512 MB.
                .withStreamSegmentSize(33554432)  // Default is 16 MB. Updating
 Segment Size to 32 MB.
                .withTimeToLiveMillis(3600000)  // By default, no TTL is enabled.
 Update TTL to 1 hour.
                .withStrategyOnFull(StrategyOnFull.RejectNewData)  // Required.
 Updating Strategy on full to reject new data.
                .withPersistence(Persistence.Memory)  // Default is File. Update the
 persistence to Memory
                .withFlushOnWrite(true)  // Default is false. Updating to true.
                .withExportDefinition(
                    // Optional. Choose where/how the stream is exported to the AWS
 ####.
                    messageStreamInfo.definition.exportDefinition
                        // Updating Export definition to add a Kinesis Stream
 configuration.
                        .withKinesis([new
 KinesisConfig().withIdentifier(uuidv4()).withKinesisStreamName(uuidv4())])
                )
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: updateMessageStream | MessageStreamDefinition

ストリーム更新の制約

ストリームを更新する際に次の制約が適用されます。次のリストに記載がない限り、更新は直ちに有効になります。

- ストリームの永続性を更新することはできません。この動作を変更するには、ストリームを削除して、新しい永続性ポリシーを定義するストリームを作成します。
- ストリームの最大サイズは、次の条件を満たしている場合に限って更新できます:
  - 最大サイズは、現在のストリームサイズ以上である必要があります。この情報を見つけるには、ストリームを記述して、返された `MessageStreamInfo` オブジェクトのストレージ状態を確認します。
  - 最大サイズは、ストリームのセグメントサイズ以上である必要があります。
- ストリーム セグメント サイズを、ストリームの最大サイズよりも小さい値に更新できます。更新した設定は、新しいセグメントに適用されます。
- 有効期限 (TTL) プロパティの更新は、新しい追加操作に適用されます。この値を減らす場合、ストリームマネージャーは TTL を超える既存のセグメントを削除することもあります。
- フルプロパティの戦略の更新は、新しい追加操作に適用されます。最も古いデータを上書きするように戦略を設定する場合、ストリームマネージャーは新しい設定に基づいて既存のセグメントも上書きすることがあります。
- 書き込み時のフラッシュのプロパティ更新は、新しいメッセージに適用されます。
- エクスポート設定の更新は、新しいエクスポートに適用されます。更新リクエストは、サポートするすべてのエクスポート設定を含める必要があります。それ以外の場合、ストリームマネージャーが削除します。
  - エクスポート設定を更新するとき、ターゲットのエクスポート設定の識別子を指定します。
  - エクスポート設定を追加するには、新しいエクスポート設定に対して一意の識別子を指定します。
  - エクスポート設定を削除するには、エクスポート設定を省略します。
- ストリームのエクスポート設定の開始シーケンス番号を更新するには、最後のシーケンス番号よりも小さい値を指定する必要があります。この情報を見つけるには、ストリームを記述して、返された `MessageStreamInfo` オブジェクトのストレージ状態を確認します。

メッセージストリームの削除

ストリームを削除します。ストリームを削除すると、ストリームに保存されているすべてのデータがディスクから削除されます。

要件

この操作には次の要件があります:

- 最小 AWS IoT Greengrass Core バージョン: 1.10.0
- AWS IoT Greengrass Core SDK の最小バージョン: Python: 1.5.0 | Java: 1.4.0 | Node.js: 1.6.0

例

次のスニペットは、`StreamName` という名前のストリームを削除します。

Python

```
client = StreamManagerClient()

try:
    client.delete_message_stream(stream_name="StreamName")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK リファレンス: deleteMessageStream

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    client.deleteMessageStream("StreamName");
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: delete_message_stream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        await client.deleteMessageStream("StreamName");
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: deleteMessageStream

関連情報

- データストリームの管理
- the section called “ストリームマネージャーの設定”
- the section called “AWS クラウド でサポートされている送信先のエクスポート設定”
- the section called “データストリームのエクスポート (コンソール)”
- the section called “データストリームのエクスポート (CLI)”
- `StreamManagerClient` AWS IoT Greengrass Core SDK リファレンスの：
  - Python
  - Java
  - Node.js

AWS クラウド でサポートされている送信先のエクスポート設定

ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Core SDK `StreamManagerClient`で を使用してストリームマネージャーとやり取りします。Lambda 関数は、ストリームの作成またはストリームの更新を行う際に、`MessageStreamDefinition` オブジェクト (エクスポート定義などのストリームプロパティ) を渡します。`ExportDefinition` オブジェクトには、そのストリームに定義

されたエクスポート設定が含まれています。ストリームマネージャーは、これらのエクスポート設定を使用して、ストリームをエクスポートする場所と方法を決定します。

1 つのストリームに 0 または 1 つ以上のエクスポートを定義できます。この場合、1 つの送信先タイプに複数のエクスポートを定義することも可能です。例えば、ストリームを 2 つの AWS IoT Analytics チャネルと 1 つの Kinesis データストリームにエクスポートできます。

エクスポートに失敗した場合、ストリームマネージャーは最大 5 分間隔で AWS クラウド へのデータのエクスポートを継続的に再試行します。再試行回数に上限はありません。

> **Note**
>
> `StreamManagerClient` を利用すると、ターゲットの送信先を使用して、ストリームを HTTP サーバーにエクスポートできます。このターゲットは、テストのみを目的としています。本場環境での使用は安定しておらず、サポートされていません。

サポートされている AWS クラウド 送信先

- AWS IoT Analytics チャネル
- Amazon Kinesis Data Streams
- AWS IoT SiteWise アセットプロパティ
- Amazon S3 オブジェクト

これらの AWS クラウド リソースは、お客様が管理する必要があります。

AWS IoT Analytics チャネル

ストリームマネージャーは への自動エクスポートをサポートしています AWS IoT Analytics。 AWS IoT Analytics では、データに対して高度な分析を実行して、ビジネス上の意思決定や機械学習モデルの改善に役立てることができます。詳細については、AWS IoT Analytics 「 ユーザーガイド」の「What is AWS IoT Analytics？」を参照してください。

AWS IoT Greengrass Core SDK では、Lambda 関数は `IoTAnalyticsConfig` を使用して、この送信先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の IoTAnalyticsConfig
- Java SDK の IoTAnalyticsConfig
- Node.js SDK の IoTAnalyticsConfig

要件

このエクスポート先には以下の要件があります。

- のターゲットチャネルは、Greengrass グループと同じ AWS アカウント および AWS リージョンにある AWS IoT Analytics 必要があります。
- the section called “Greengrass グループのロール” は、ターゲットのチャネルに `iotanalytics:BatchPutMessage` 権限を許可する必要があります。例:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iotanalytics:BatchPutMessage"
            ],
            "Resource": [
                "arn:aws:iotanalytics:region:account-id:channel/channel_1_name",
                "arn:aws:iotanalytics:region:account-id:channel/channel_2_name"
            ]
        }
    ]
```

```
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

へのエクスポート AWS IoT Analytics

にエクスポートするストリームを作成するには AWS IoT Analytics、Lambda 関数は 1 つ以上のオブジェクトを含むエクスポート定義を使用してストリームを作成します。 `IoTAnalyticsConfig`このオブジェクトによって、ターゲットチャネル、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Lambda 関数は、デバイスからデータを受信した際に、データの BLOB を含むメッセージをターゲットストリームに追加します。

その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

Amazon Kinesis Data Streams

ストリームマネージャーは、Amazon Kinesis Data Streams への自動エクスポートをサポートしています。Kinesis Data Streams は、一般的に、大量のデータを集約して、データウェアハウスまたは MapReduce クラスターに読み込むために使用されます。詳細については、「Amazon Kinesis デベロッパーガイド」の「Amazon Kinesis Data Streams とは」を参照してください。

AWS IoT Greengrass Core SDK では、Lambda 関数は `KinesisConfig` を使用して、この送信先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の KinesisConfig
- Java SDK の KinesisConfig
- Node.js SDK の KinesisConfig

要件

このエクスポート先には以下の要件があります。

- Kinesis Data Streams のターゲットストリームは、Greengrass グループと同じ AWS アカウントおよび AWS リージョン に存在する必要があります。
- the section called “Greengrass グループのロール” は、ターゲットデータストリームに `kinesis:PutRecords` 権限を許可する必要があります。例:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecords"
            ],
            "Resource": [
                "arn:aws:kinesis:region:account-id:stream/stream_1_name",
                "arn:aws:kinesis:region:account-id:stream/stream_2_name"
            ]
        }
    ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Kinesis Data Streams へのエクスポート

Kinesis Data Streams にエクスポートするストリームを作成するには、Lambda 関数で 1 つ以上の `KinesisConfig` オブジェクトを含むエクスポート定義を使用してストリームを作成します。このオブジェクトによって、ターゲットデータストリーム、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Lambda 関数は、デバイスからデータを受信した際に、データの BLOB を含むメッセージをターゲットストリームに追加します。その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

ストリームマネージャーは、Amazon Kinesis にアップロードされた各レコードのパーティションキーとして、一意のランダムな UUID を生成します。

AWS IoT SiteWise アセットプロパティ

ストリームマネージャーは への自動エクスポートをサポートしています AWS IoT SiteWise。 AWS IoT SiteWise を使用すると、産業機器からデータを大規模に収集、整理、分析できます。詳細については、AWS IoT SiteWise 「 ユーザーガイド」の「What is AWS IoT SiteWise？」を参照してください。

AWS IoT Greengrass Core SDK では、Lambda 関数は `IoTSiteWiseConfig` を使用して、この送信先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の IoTSiteWiseConfig
- Java SDK の IoTSiteWiseConfig
- Node.js SDK の IoTSiteWiseConfig

> **Note**
>
> AWS はthe section called “IoT SiteWise”、OPC-UA ソースで使用できる構築済みのソリューションである も提供します。

要件

このエクスポート先には以下の要件があります。

- のターゲットアセットプロパティは、Greengrass グループと同じ AWS アカウント および AWS リージョン にある AWS IoT SiteWise 必要があります。

  > **Note**
  >
  > が AWS IoT SiteWise サポートするリージョンのリストについては、 AWS 全般のリファレンスのAWS IoT SiteWise 「エンドポイントとクォータ」を参照してください。

- the section called “Greengrass グループのロール” は、ターゲットのアセットプロパティに `iotsitewise:BatchPutAssetPropertyValue` 権限を許可する必要があります。次の例では、ポリシーで `iotsitewise:assetHierarchyPath` 条件キーを使用して、ターゲットルートアセットとその子へのアクセスを許可しています。ポリシーから `Condition` を削除して、すべて

の AWS IoT SiteWise アセットへのアクセスを許可したり、個々のアセットの ARN を指定したりできます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
             }
        }
    ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

重要なセキュリティ情報については、AWS IoT SiteWise ユーザーガイドの「BatchPutAssetPropertyValue 認証」を参照してください。

へのエクスポート AWS IoT SiteWise

にエクスポートするストリームを作成するには AWS IoT SiteWise、Lambda 関数は 1 つ以上のオブジェクトを含むエクスポート定義を使用してストリームを作成します。 `IoTSiteWiseConfig`このオブジェクトによって、バッチサイズ、バッチ間隔、優先度などのエクスポート設定を定義します。

Lambda 関数は、デバイスからアセットプロパティデータを受信した際に、そのデータを含むメッセージをターゲットストリームに追加します。メッセージは、JSON シリアル化された `PutAssetPropertyValueEntry` オブジェクトであり、これには、1 つ以上のアセットプロパティに対するプロパティ値が含まれています。詳細については、 AWS IoT SiteWise のエクスポート先に関する「メッセージの追加」を参照してください。

> **Note**
>
> にデータを送信する場合 AWS IoT SiteWise、データは `BatchPutAssetPropertyValue`アクションの要件を満たしている必要があります。詳細については、「AWS IoT SiteWise API リファレンス」の「BatchPutAssetPropertyValue」を参照してください。

その後、ストリームマネージャーは、ストリームのエクスポート設定で定義されたバッチ設定と優先度に基づいてデータをエクスポートします。

ストリームマネージャーの設定と Lambda 関数ロジックを調整して、エクスポート対策を設計できます。例:

- ほぼリアルタイムのエクスポートでは、少ないバッチサイズと短い間隔を設定して、受信したデータをストリームに追加します。
- バッチの最適化、帯域幅の制約軽減、コスト最小化を行うには、Lambda 関数を使用して、受信した、単一アセットプロパティのタイムスタンプ品質値 (TQV) データポイントをプールし、その後、データをストリームに追加します。対策を 1 つ挙げるとすれば、それは、同じプロパティのエントリを複数送信するのではなく、最大 10 の異なるプロパティとアセットの組み合わせ、またはプロパティエイリアスのエントリを 1 つのメッセージでバッチ処理することです。これにより、ストリームマネージャーは AWS IoT SiteWise クォータ内にとどまることができます。

Amazon S3 オブジェクト

ストリームマネージャーは、Amazon S3 への自動エクスポートをサポートしています。Amazon S3 を使用すると、膨大なデータの保存と取得を行えます。詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「Amazon S3 とは」を参照してください。

AWS IoT Greengrass Core SDK では、Lambda 関数は `S3ExportTaskExecutorConfig` を使用して、この送信先タイプのエクスポート設定を定義します。詳細については、ターゲット言語の SDK リファレンスを参照してください。

- Python SDK の S3ExportTaskExecutorConfig
- Java SDK の S3ExportTaskExecutorConfig

- Node.js SDK の S3ExportTaskExecutorConfig

要件

このエクスポート先には以下の要件があります。

- ターゲット Amazon S3 バケットは Greengrass グループ AWS アカウント と同じ にある必要があります。
- Greengrass グループのデフォルトのコンテナ化が Greengrass コンテナの場合に、/tmp 下にある入力ファイルディレクトリ、またはルートファイルシステム上にない入力ファイルディレクトリを使用するには、STREAM_MANAGER_READ_ONLY_DIRS パラメータを設定する必要があります。
- Greengrass コンテナモードで実行している Lambda 関数によって入力ファイルを入力ファイルディレクトリに書き込む場合は、そのディレクトリ用にローカルボリュームリソースを作成し、書き込み権限でディレクトリをコンテナにマウントする必要があります。これにより、ファイルがルートファイルシステムに書き込まれ、コンテナ外に表示されるようになります。詳細については、「ローカルリソースへのアクセス」を参照してください。
- the section called “Greengrass グループのロール” で、ターゲットのバケットに次の権限を許可する必要があります。例:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:AbortMultipartUpload",
                "s3:ListMultipartUploadParts"
            ],
            "Resource": [
                "arn:aws:s3:::bucket-1-name/*",
                "arn:aws:s3:::bucket-2-name/*"
            ]
        }
    ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Amazon S3 へのエクスポート

Amazon S3 にエクスポートするストリームを作成するには、Lambda 関数で `S3ExportTaskExecutorConfig` オブジェクトを使用してエクスポートポリシーを設定します。このポリシーによって、マルチパートアップロードのしきい値や優先度といったエクスポート設定を定義します。Amazon S3 エクスポートでは、ストリームマネージャーが、コアデバイス上のローカルファイルから読み取るデータをアップロードします。アップロードを開始するには、Lambda 関数でエクスポートタスクをターゲットストリームに追加します。エクスポートタスクには、入力ファイルとターゲット Amazon S3 オブジェクトの情報が含まれます。ストリームマネージャーは、ストリームに追加された順にタスクを実行します。

> **Note**
>
> ターゲットバケットは に既に存在している必要があります AWS アカウント。指定したキーのオブジェクトが存在しない場合は、ストリームマネージャーによってオブジェクトが作成されます。

以下の図に、この高レベルのワークフローを示します。

ストリームマネージャーは、マルチパートアップロードしきい値プロパティ、最小パーツサイズ設定、入力ファイルサイズを使用して、データのアップロード方法を決定します。マルチパートアップロードのしきい値は、最小パートサイズ以上でなければなりません。データを並行してアップロードする場合は、複数のストリームを作成できます。

ターゲット Amazon S3 オブジェクトを指定するキーには、有効な Java DateTimeFormatter 文字列を `!{timestamp:value}` プレースホルダーに含めることができます。これらのタイムスタンププレースホルダーを使用すると、入力ファイルデータがアップロードされた時刻に基づいて Amazon S3 のデータを分割できます。例えば、次のキー名は、`my-key/2020/12/31/data.txt` などの値に解決されます。

```
my-key/!{timestamp:YYYY}/!{timestamp:MM}/!{timestamp:dd}/data.txt
```

> **Note**
>
> ストリームのエクスポートステータスを監視する場合は、まずステータスストリームを作成して、そのストリームを使用するようにエクスポートストリームを設定します。詳細については、「the section called “エクスポートタスクの監視”」を参照してください。

入力データの管理

IoT アプリケーションが入力データのライフサイクル管理に使用するコードを作成できます。次のワークフロー例は、Lambda 関数を使用してこのデータを管理する方法を示しています。

1. ローカルプロセスは、デバイスまたは周辺機器からデータを受信し、コアデバイスのディレクトリ内にあるファイルにデータを書き込みます。これらが、ストリームマネージャーの入力ファイルとなります。

   > **Note**
   >
   > 入力ファイルディレクトリへのアクセスを設定する必要があるかどうかを判断するには、STREAM_MANAGER_READ_ONLY_DIRS パラメータを参照してください。
   > ストリームマネージャーが実行されるプロセスは、グループのデフォルトアクセス ID のファイルシステム権限をすべて継承します。ストリームマネージャーには、入力ファイルへのアクセス許可が必要です。必要に応じて、`chmod(1)` コマンドを使用して、ファイルへのアクセス許可を変更できます。

2. Lambda 関数はディレクトリをスキャンし、新しいファイルが作成されると、ターゲットストリームにエクスポートタスクを追加します。このタスクは、JSON シリアル化された `S3ExportTaskDefinition` オブジェクトであり、これによって、入力ファイルの URL、ターゲットの Amazon S3 バケットとキー、オプションのユーザーメタデータが指定されます。
3. ストリームマネージャーは、入力ファイルを読み取り、追加されたタスクの順に Amazon S3 にデータをエクスポートします。ターゲットバケットは に既に存在している必要があります AWS アカウント。指定したキーのオブジェクトが存在しない場合は、ストリームマネージャーによってオブジェクトが作成されます。
4. Lambda 関数は、ステータスストリームからメッセージを読み取り、エクスポートステータスを監視します。エクスポートタスクが完了すると、Lambda 関数は対応する入力ファイルを削除します。詳細については、「the section called “エクスポートタスクの監視”」を参照してください。

エクスポートタスクの監視

IoT アプリケーションが Amazon S3 エクスポートのステータス監視に使用するコードを作成できます。Lambda 関数は、ステータスストリームを作成して、そのストリームにステータス更新を書き込むようにエクスポートストリームを設定する必要があります。1 つのステータスストリームは、Amazon S3 にエクスポートする複数のストリームからステータスの更新を受け取ることができます。

まず、ステータスストリームとして使用するストリームを作成します。ストリームのサイズとリテンションポリシーを設定して、ステータスメッセージのライフスパンを制御できます。例:

- ステータスメッセージを保存しない場合は、`Persistence` を `Memory` に設定します。
- 新しいステータスメッセージが失われないようにするには、`StrategyOnFull` を `OverwriteOldestData` に設定します。

次に、ステータスストリームを使用するようにエクスポートストリームを作成または更新します。具体的には、ストリームの `S3ExportTaskExecutorConfig` エクスポート設定のステータス構成プロパティを設定します。これにより、エクスポートタスクに関するステータスメッセージをステータスストリームに書き込むようにストリームマネージャーに指示します。`StatusConfig` オブジェクトで、ステータスストリームの名前と冗長性のレベルを指定します。サポート対象の値を次に示します。最も冗長でないもの (`ERROR`) から最も冗長なもの (`TRACE`) を表しています。デフォルト: `INFO`。

- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

次のワークフロー例は、Lambda 関数がステータスストリームを使用してエクスポートステータスを監視する方法を示しています。

1. 前のワークフローで説明したように、Lambda 関数は、エクスポートタスクに関するステータスメッセージをステータスストリームに書き込むように設定されたストリームにエクスポートタスクを追加します。この追加の操作によって、タスク ID を表すシーケンス番号が返ります。

2. Lambda 関数は、ステータスストリームからメッセージを順番に読み取ります。その後、ストリーム名とタスク ID に基づいて、またはメッセージコンテキストからのエクスポートタスクプロパティに基づいてメッセージをフィルタリングします。例えば、Lambda 関数は、エクスポートタスクの入力ファイル URL でフィルタリングできます。このタスクは、メッセージコンテキストの `S3ExportTaskDefinition` オブジェクトで表されます。

   次のステータスコードは、エクスポートタスクが完了の状態になったことを示します。

   - `Success`。アップロードは正常に完了しました。
   - `Failure`。ストリームマネージャーでエラー (例: 指定したバケットが存在しないなど) が発生しました。問題の解決後に、エクスポートタスクをストリームに再度追加できます。
   - `Canceled`。ストリームまたはエクスポートの定義が削除された、もしくはタスクの存続期間 (TTL) の有効期限が切れたため、タスクは中止されました。

   > **Note**
   >
   > タスクのステータスは `InProgress` または `Warning` の場合もあります。ストリームマネージャーは、タスクの実行に影響しないエラーがイベントから返ったときに警告を発行します。例えば、中断された部分的アップロードのクリーンアップが失敗すると、警告を返します。

3. エクスポートタスクが完了すると、Lambda 関数は対応する入力ファイルを削除します。

次の例は、Lambda 関数がステータスメッセージを読み取り、処理する方法を示しています。

Python

```
import time
from greengrasssdk.stream_manager import (
    ReadMessagesOptions,
    Status,
    StatusConfig,
    StatusLevel,
    StatusMessage,
    StreamManagerClient,
)
from greengrasssdk.stream_manager.util import Util

client = StreamManagerClient()
```

```
try:
    # Read the statuses from the export status stream
    is_file_uploaded_to_s3 = False
    while not is_file_uploaded_to_s3:
        try:
            messages_list = client.read_messages(
                "StatusStreamName", ReadMessagesOptions(min_message_count=1,
 read_timeout_millis=1000)
            )
            for message in messages_list:
                # Deserialize the status message first.
                status_message = Util.deserialize_json_bytes_to_obj(message.payload,
 StatusMessage)

                # Check the status of the status message. If the status is
 "Success",
                # the file was successfully uploaded to S3.
                # If the status was either "Failure" or "Cancelled", the server was
 unable to upload the file to S3.
                # We will print the message for why the upload to S3 failed from the
 status message.
                # If the status was "InProgress", the status indicates that the
 server has started uploading
                # the S3 task.
                if status_message.status == Status.Success:
                    logger.info("Successfully uploaded file at path " + file_url + "
 to S3.")
                    is_file_uploaded_to_s3 = True
                elif status_message.status == Status.Failure or
 status_message.status == Status.Canceled:
                    logger.info(
                        "Unable to upload file at path " + file_url + " to S3.
 Message: " + status_message.message
                    )
                    is_file_uploaded_to_s3 = True
            time.sleep(5)
        except StreamManagerException:
            logger.exception("Exception while running")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
```

```
    # Properly handle errors.
```

Python SDK リファレンス: read_messages | StatusMessage

Java

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;
import com.amazonaws.greengrass.streammanager.client.utils.ValidateAndSerialize;
import com.amazonaws.greengrass.streammanager.model.ReadMessagesOptions;
import com.amazonaws.greengrass.streammanager.model.Status;
import com.amazonaws.greengrass.streammanager.model.StatusConfig;
import com.amazonaws.greengrass.streammanager.model.StatusLevel;
import com.amazonaws.greengrass.streammanager.model.StatusMessage;

try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    try {
        boolean isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                List<Message> messages = client.readMessages("StatusStreamName",
                    new
 ReadMessagesOptions().withMinMessageCount(1L).withReadTimeoutMillis(1000L));
                for (Message message : messages) {
                    // Deserialize the status message first.
                    StatusMessage statusMessage =
 ValidateAndSerialize.deserializeJsonBytesToObj(message.getPayload(),
 StatusMessage.class);
                    // Check the status of the status message. If the status is
 "Success", the file was successfully uploaded to S3.
                    // If the status was either "Failure" or "Canceled", the server
 was unable to upload the file to S3.
                    // We will print the message for why the upload to S3 failed
 from the status message.
                    // If the status was "InProgress", the status indicates that the
 server has started uploading the S3 task.
                    if (Status.Success.equals(statusMessage.getStatus())) {
                        System.out.println("Successfully uploaded file at path " +
 FILE_URL + " to S3.");
                        isS3UploadComplete = true;
                     } else if (Status.Failure.equals(statusMessage.getStatus()) ||
 Status.Canceled.equals(statusMessage.getStatus())) {
```

```
                        System.out.println(String.format("Unable to upload file at
 path %s to S3. Message %s",

 statusMessage.getStatusContext().getS3ExportTaskDefinition().getInputUrl(),
                            statusMessage.getMessage()));
                        sS3UploadComplete = true;
                    }
                }
            } catch (StreamManagerException ignored) {
            } finally {
                // Sleep for sometime for the S3 upload task to complete before
 trying to read the status message.
                Thread.sleep(5000);
            }
        } catch (e) {
        // Properly handle errors.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK リファレンス: readMessages | StatusMessage

Node.js

```
const {
    StreamManagerClient, ReadMessagesOptions,
    Status, StatusConfig, StatusLevel, StatusMessage,
    util,
} = require('aws-greengrass-core-sdk').StreamManager;

const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        let isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                const messages = await c.readMessages("StatusStreamName",
                    new ReadMessagesOptions()
                        .withMinMessageCount(1)
                        .withReadTimeoutMillis(1000));

                messages.forEach((message) => {
```

```
                    // Deserialize the status message first.
                    const statusMessage =
 util.deserializeJsonBytesToObj(message.payload, StatusMessage);
                    // Check the status of the status message. If the status is
 'Success', the file was successfully uploaded to S3.
                    // If the status was either 'Failure' or 'Cancelled', the server
 was unable to upload the file to S3.
                    // We will print the message for why the upload to S3 failed
 from the status message.
                    // If the status was "InProgress", the status indicates that the
 server has started uploading the S3 task.
                    if (statusMessage.status === Status.Success) {
                        console.log(`Successfully uploaded file at path ${FILE_URL}
 to S3.`);
                        isS3UploadComplete = true;
                    } else if (statusMessage.status === Status.Failure ||
 statusMessage.status === Status.Canceled) {
                        console.log(`Unable to upload file at path ${FILE_URL} to
 S3. Message: ${statusMessage.message}`);
                        isS3UploadComplete = true;
                    }
                });
                // Sleep for sometime for the S3 upload task to complete before
 trying to read the status message.
                await new Promise((r) => setTimeout(r, 5000));
            } catch (e) {
                // Ignored
            }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK リファレンス: readMessages | StatusMessage

AWS クラウド へのデータストリームエクスポート (コンソール)

このチュートリアルでは、 AWS IoT コンソールを使用して、ストリームマネージャーを有効にした AWS IoT Greengrass グループを設定およびデプロイする方法を示します。グループには、ストリームマネージャーのストリームに書き込むユーザー定義 Lambda 関数が含まれています。ストリームマネージャーは、自動的に AWS クラウドにエクスポートされます。

ストリームマネージャーにより、大量のデータストリームの取り込み、処理、エクスポートが効率化され、信頼性が向上します。このチュートリアルでは、IoT データを消費する `TransferStream` Lambda 関数を作成します。Lambda 関数は AWS IoT Greengrass Core SDK を使用してストリームマネージャーでストリームを作成し、それに読み書きします。ストリームマネージャーは、ストリームを Kinesis Data Streams にエクスポートします。以下の図は、このワークフローを示しています。

このチュートリアルでは、ユーザー定義の Lambda 関数が AWS IoT Greengrass Core SDK の `StreamManagerClient` オブジェクトを使用してストリームマネージャーとやり取りする方法を示します。簡単にするために、このチュートリアルで作成する Python Lambda 関数は、シミュレートされたデバイスデータを生成します。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.10 以降)。Greengrass のグループまたはコアを作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。入門チュートリアルには、 AWS IoT Greengrass Core ソフトウェアをインストールする手順も含まれています。

> **Note**
>
> ストリームマネージャーは OpenWrt ディストリビューションではサポートされていません。

- コアデバイスにインストールされている Java 8 ランタイム (JDK 8)。
  - Debian ベースのディストリビューション (Raspbian を含む) または Ubuntu ベースのディストリビューションの場合は、次のコマンドを実行します。

    ```
    sudo apt install openjdk-8-jdk
    ```

  - Red Hat ベースのディストリビューション (Amazon Linux を含む) の場合は、次のコマンドを実行します。

    ```
    sudo yum install java-1.8.0-openjdk
    ```

    詳細については、OpenJDK ドキュメントの「How to download and install prebuilt OpenJDK packages」を参照してください。
- AWS IoT Greengrass Core SDK for Python v1.5.0 以降。 AWS IoT Greengrass Core SDK for Python で `StreamManagerClient` を使用するには、以下を行う必要があります。
  - Python 3.7 以降をコアデバイスにインストールします。
  - SDK とその依存関係を Lambda 関数デプロイパッケージに含めます。このチュートリアルでは、手順を説明しています。

> **Tip**
>
> `StreamManagerClient` を Java または NodeJS で使用できます。コードの例については、GitHub の 「AWS IoT Greengrass Core SDK for Java」と「AWS IoT Greengrass Core SDK for Node.js」を参照してください。

- Greengrass グループ AWS リージョン と同じ の Amazon Kinesis Data Streams で**`MyKinesisStream`**作成された という名前の送信先ストリーム。詳細については、「Amazon Kinesis デベロッパーガイド」の「ストリームを作成する」を参照してください。

> **Note**
>
> このチュートリアルでは、ストリームマネージャーが Kinesis Data Streams にデータをエクスポートするため、 AWS アカウントに課金されます。料金の詳細については、「Amazon Kinesis Data Streams の料金」を参照してください。
> 料金が発生しないようにするには、Kinesis データストリームを作成せずにこのチュートリアルを実行します。この場合、ログを確認して、ストリームマネージャーがストリームを Kinesis Data Streams にエクスポートしようとしたことを確認します。

- 次の例に示すように、ターゲットデータストリームで `kinesis:PutRecords` アクションを許可する the section called “Greengrass グループのロール” に IAM ポリシーが追加されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecords"
            ],
            "Resource": [
                "arn:aws:kinesis:region:account-id:stream/MyKinesisStream"
            ]
        }
    ]
}
```

このチュートリアルのおおまかな手順は以下のとおりです。

1. Lambda 関数デプロイパッケージを作成する
2. 「Lambda 関数の作成」
3. グループに関数を追加する
4. ストリームマネージャーを有効にする
5. ローカルログを設定する
6. グループをデプロイする
7. アプリケーションをテストする

このチュートリアルは完了までに約 20 分かかります。

ステップ 1: Lambda 関数デプロイパッケージを作成する

この手順では、Python 関数コードと依存関係を含む Lambda 関数デプロイパッケージを作成します。このパッケージは、後で AWS Lambdaで Lambda 関数を作成するときにアップロードします。Lambda 関数は AWS IoT Greengrass Core SDK を使用してローカルストリームを作成し、操作します。

> **Note**
>
> ユーザー定義の Lambda 関数は、AWS IoT Greengrass Core SDK を使用してストリームマネージャーと対話する必要があります。Greengrass ストリームマネージャーの要件の詳細については、「Greengrass ストリームマネージャーの要件」を参照してください。

1. v1.5.0 以降の AWS IoT Greengrass Core SDK for Python をダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
3. パッケージ依存関係をインストールして、Lambda 関数デプロイパッケージに SDK を含めます。
   1. `requirements.txt` ファイルのある SDK ディレクトリに移動します。このファイルには依存関係の一覧が記載されています。
   2. SDK の依存関係をインストールします。例えば、次の `pip` コマンドを実行して、現在のディレクトリにインストールします。

      ```
      pip install --target . -r requirements.txt
      ```
4. 以下の Python コード関数を `transfer_stream.py` というローカルファイルに保存します。

> **Tip**
>
> Java と NodeJS を使用するコードの例については、GitHub の「AWS IoT Greengrass Core SDK for Java」と「AWS IoT Greengrass Core SDK for Node.js」を参照してください。

```
import asyncio
import logging
import random
import time

from greengrasssdk.stream_manager import (
    ExportDefinition,
    KinesisConfig,
    MessageStreamDefinition,
    ReadMessagesOptions,
    ResourceNotFoundException,
    StrategyOnFull,
    StreamManagerClient,
)


# This example creates a local stream named "SomeStream".
# It starts writing data into that stream and then stream manager automatically
 exports
# the data to a customer-created Kinesis data stream named "MyKinesisStream".
# This example runs forever until the program is stopped.

# The size of the local stream on disk will not exceed the default (which is 256
 MB).
# Any data appended after the stream reaches the size limit continues to be
 appended, and
# stream manager deletes the oldest data until the total stream size is back under
 256 MB.
# The Kinesis data stream in the cloud has no such bound, so all the data from this
 script is
# uploaded to Kinesis and you will be charged for that usage.


def main(logger):
    try:
        stream_name = "SomeStream"
        kinesis_stream_name = "MyKinesisStream"

        # Create a client for the StreamManager
        client = StreamManagerClient()

        # Try deleting the stream (if it exists) so that we have a fresh start
```

```
        try:
            client.delete_message_stream(stream_name=stream_name)
        except ResourceNotFoundException:
            pass

        exports = ExportDefinition(
            kinesis=[KinesisConfig(identifier="KinesisExport" + stream_name,
kinesis_stream_name=kinesis_stream_name)]
        )
        client.create_message_stream(
            MessageStreamDefinition(
                name=stream_name,
strategy_on_full=StrategyOnFull.OverwriteOldestData, export_definition=exports
            )
        )

        # Append two messages and print their sequence numbers
        logger.info(
            "Successfully appended message to stream with sequence number %d",
            client.append_message(stream_name, "ABCDEFGHIJKLMNO".encode("utf-8")),
        )
        logger.info(
            "Successfully appended message to stream with sequence number %d",
            client.append_message(stream_name, "PQRSTUVWXYZ".encode("utf-8")),
        )

        # Try reading the two messages we just appended and print them out
        logger.info(
            "Successfully read 2 messages: %s",
            client.read_messages(stream_name,
ReadMessagesOptions(min_message_count=2, read_timeout_millis=1000)),
        )

        logger.info("Now going to start writing random integers between 0 and 1000
to the stream")
        # Now start putting in random data between 0 and 1000 to emulate device
sensor input
        while True:
            logger.debug("Appending new random integer to stream")
            client.append_message(stream_name, random.randint(0,
1000).to_bytes(length=4, signed=True, byteorder="big"))
            time.sleep(1)

    except asyncio.TimeoutError:
```

```
        logger.exception("Timed out while executing")
    except Exception:
        logger.exception("Exception while running")


def function_handler(event, context):
    return


logging.basicConfig(level=logging.INFO)
# Start up this sample code
main(logger=logging.getLogger())
```

5. 以下の項目を `transfer_stream_python.zip` という名前のファイルに圧縮します。これが Lambda 関数デプロイパッケージです。

   - transfer_stream.py。アプリケーションロジック。
   - greengrasssdk。MQTT メッセージを発行する Python Greengrass Lambda 関数で必須のライブラリ。

     ストリームマネージャーオペレーションは、 AWS IoT Greengrass Core SDK for Python のバージョン 1.5.0 以降で使用できます。
   - AWS IoT Greengrass Core SDK for Python にインストールした依存関係 (`cbor2`ディレクトリなど) 。

   `zip` ファイルを作成するときは、これらの項目のみを含み、それらが含まれているフォルダは含みません。

ステップ 2: Lambda 関数を作成する

このステップでは、 AWS Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージを使用するように設定します。次に、関数のバージョンを公開し、エイリアスを作成します。

1. 最初に Lambda 関数を作成します。

   a. で AWS Management Console、サービスを選択し、 AWS Lambda コンソールを開きます。

   b. [Create function] (関数の作成) を選択し、[Author from scratch] (一から作成) を選択します。

c. [基本的な情報] セクションで、以下の値を使用します。

- [関数名] に「**`TransferStream`**」と入力します。
- [ランタイム] で [Python 3.7] を選択します。
- [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

d. ページの下部で、[関数の作成] を選択します。

2. 今度は、ハンドラを登録し、Lambda 関数デプロイパッケージをアップロードします。

a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

b. [アップロード] を選択し、`transfer_stream_python.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

- [ランタイム] で [Python 3.7] を選択します。
- [ハンドラ] に **`transfer_stream.function_handler`** と入力します。

d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コンソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

3. ここで、Lambda 関数の最初のバージョンを公開し、バージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

a. [アクション] メニューから、[新しいバージョンを発行] を選択します。

b. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。

c. [TransferStream: 1] 設定ページで、[アクション] メニューの [エイリアスを作成] を選択します。

d. [新しいエイリアスの作成] ページで、次の値を使用します。

   - [名前] に **`GG_TransferStream`** と入力します。
   - [バージョン] で、[1] を選択します。

   > **Note**
   >
   > AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

e. [Create] (作成) を選択します。

これで、Greengrass グループに Lambda 関数を追加する準備ができました。

ステップ 3: Greengrass グループに Lambda 関数を追加する

このステップでは、Lambda 関数をグループに追加し、そのライフサイクルと環境変数を設定します。詳細については、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。

3. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
4. [My Lambda functions] (自分の Lambda 関数) で、[Add] (追加)を選択します。
5. [Lambda 関数の追加] ページで、Lambda 関数用の [Lambda 関数] を選択します。
6. Lambda バージョンでは、Alias:GG_TransferStream を選択します。

   ここで、Greengrass グループの Lambda 関数の動作を決定するプロパティを設定します。

7. [Lambda function configuration] (Lambda 関数の設定) セクションで、次のように変更します。
   - [メモリ制限] を 32 MB に設定します。
   - [Pinned] (固定)で、[True] を選択します。

   > **Note**
   >
   > 存続期間の長い (または固定された) Lambda 関数は、 AWS IoT Greengrass の起動後に自動的に起動し、独自のコンテナで実行し続けます。これはオンデマンド Lambda 関数とは対照的です。この関数は呼び出されたときに開始し、実行するタスクが残っていないときに停止します。詳細については、「the section called “ライフサイクル設定”」を参照してください。

8. [Add Lambda function] (Lambda 関数の追加) を選択します。

ステップ 4: ストリームマネージャーを有効にする

この手順では、ストリームマネージャーが有効になっていることを確認します。

1. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
2. [System Lambda functions] (システム Lambda 関数) で、 [Stream manager] (ストリームマネージャー) を選択し、ステータスを確認します。無効になっている場合は、[編集] を選択します。次に、[有効化] を選択し、[保存] を選択します。このチュートリアルでは、デフォルトのパラメータ設定を使用できます。詳細については、「the section called “ストリームマネージャーの設定”」を参照してください。

> **Note**
>
> コンソールを使用してストリームマネージャーを有効にし、グループをデプロイすると、ストリームマネージャーのメモリサイズはデフォルトで 4194304 KB (4 GB) に設定されます。メモリのサイズは 128000 KB 以上に設定することをお勧めします。

ステップ 5: ローカルログを設定する

このステップでは、コアデバイスのファイルシステムにログを書き込むように、 グループ内の AWS IoT Greengrass システムコンポーネント、ユーザー定義の Lambda 関数、コネクタを設定します。ログを使用して、発生する可能性のある問題のトラブルシューティングを行うことができます。詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

1. [ローカルログ設定] で、ローカルログが設定されているかどうかを確認します。
2. Greengrass システムコンポーネントやユーザー定義 Lambda 関数のログが設定されていない場合は、[編集] を選択します。
3. [User Lambda functions log level] (ユーザー Lambda 関数のログレベル) と [Greengrass system log level] (Greengrass システムのログレベル) を選択します。
4. ログレベルとディスク容量制限のデフォルト値はそのままにし、[Save (保存)] を選択します。

ステップ 6: Greengrass グループをデプロイする

Core デバイスにグループをデプロイします。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   出力に `root` で実行中の `/greengrass/ggc/packages/`*`ggc-version`*`/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

> **Note**
>
> パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

b. デーモンを開始するには、以下を実行します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。
3. a. [Lambda functions] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) セクションで、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。

   b. [IP ディテクターの設定を編集] ダイアログボックスで、[MQTT ブローカーエンドポイントを自動的に検出して上書きする] を選択します。

   c. [Save] を選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   >
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイのステータスが [完了] と表示されます。

   トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

ステップ 7: アプリケーションをテストする

この `TransferStream` Lambda 関数は、シミュレートされたデバイスデータを生成します。ストリームマネージャーがターゲットの Kinesis データストリームにエクスポートするストリームにデータを書き込みます。

1. Amazon Kinesis コンソールの [Kinesis data streams] (Kinesis データストリーム) で、[MyKinesisStream] を選択します。

   > **Note**
   >
   > ターゲットの Kinesis データストリームを使用せずにチュートリアルを実行した場合は、ストリームマネージャーのログファイルを確認します (`GGStreamManager`)。エラーメッセージに `export stream MyKinesisStream doesn't exist` が含まれている場合、テストは成功します。このエラーは、サービスがストリームにエクスポートしようとしましたが、ストリームが存在しないことを意味します。

2. [MyKinesisStream] ページで、[Monitoring (モニタリング)] を選択します。テストが成功すると、Put Records (レコードの配置) グラフにデータが表示されます。接続によっては、データが表示されるまでに 1 分かかることがあります。

   > **Important**
   >
   > テストが終了したら、Kinesis データストリームを削除して、それ以上の料金が発生しないようにします。
   > または、次のコマンドを実行して Greengrass デーモンを停止します。これにより、テストを続行する準備が整うまで、コアがメッセージを送信できなくなります。
   >
   > ```
   > cd /greengrass/ggc/core/
   > sudo ./greengrassd stop
   > ```

3. コアから TransferStream Lambda 関数を削除します。

   a. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。

   b. [Greengrass groups] (Greengrass グループ) で、対象グループを選択します。

   c. [Lambdas] ページで、[TransferStream] 関数の省略記号 (...)、[Remove function] (関数の削除) の順に選択します。

d. [アクション] で、[デプロイ] を選択します。

ロギング情報を表示したり、ストリームに関する問題をトラブルシューティングしたりするには、`TransferStream` および `GGStreamManager` 関数のログを確認します。ファイルシステムの AWS IoT Greengrass ログを読み取るための`root`アクセス許可が必要です。

- `TransferStream` は、ログエントリを *`greengrass-root`*`/ggc/var/log/user/`*`region`*`/`*`account-id`*`/TransferStream.log` に書き込みます。
- `GGStreamManager` は、ログエントリを *`greengrass-root`*`/ggc/var/log/system/GGStreamManager.log` に書き込みます。

トラブルシューティングの詳細が必要な場合は、[User Lambda logs] (ユーザー Lambda ログ) のログレベルを [Debug logs] (デバッグログ) に設定してから、グループを再度デプロイできます。

関連情報

- データストリームの管理
- the section called “ストリームマネージャーの設定”
- the section called “ストリームを操作するために StreamManagerClient を使用する”
- the section called “AWS クラウド でサポートされている送信先のエクスポート設定”
- the section called “データストリームのエクスポート (CLI)”

データストリームを にエクスポートする AWS クラウド (CLI)

このチュートリアルでは、 を使用して AWS CLI 、ストリームマネージャーを有効にした AWS IoT Greengrass グループを設定およびデプロイする方法を示します。グループには、ストリームマネージャーのストリームに書き込むユーザー定義 Lambda 関数が含まれています。ストリームマネージャーは、自動的に AWS クラウドにエクスポートされます。

ストリームマネージャーにより、大量のデータストリームの取り込み、処理、エクスポートが効率化され、信頼性が向上します。このチュートリアルでは、IoT データを消費する `TransferStream` Lambda 関数を作成します。Lambda 関数は AWS IoT Greengrass Core SDK を使用してストリームマネージャーでストリームを作成し、それに読み書きします。ストリームマネージャーは、ストリームを Kinesis Data Streams にエクスポートします。以下の図は、このワークフローを示しています。

このチュートリアルでは、ユーザー定義の Lambda 関数が AWS IoT Greengrass Core SDK の `StreamManagerClient` オブジェクトを使用してストリームマネージャーとやり取りする方法を示します。簡単にするために、このチュートリアルで作成する Python Lambda 関数は、シミュレートされたデバイスデータを生成します。

に Greengrass コマンドを含む AWS IoT Greengrass API を使用してグループ AWS CLIを作成すると、ストリームマネージャーはデフォルトで無効になります。コアでストリームマネージャーを有効にするには、システム `GGStreamManager` Lambda 関数に加え、新しい関数定義バージョンを参照するグループバージョンを含む関数定義バージョンを作成します。次に、フローをデプロイします。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.10 以降)。Greengrass のグループまたはコアを作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。入門チュートリアルには、 AWS IoT Greengrass Core ソフトウェアをインストールする手順も含まれています。

> **Note**
>
> ストリームマネージャーは OpenWrt ディストリビューションではサポートされていません。

- コアデバイスにインストールされている Java 8 ランタイム (JDK 8)。
  - Debian ベースのディストリビューション (Raspbian を含む) または Ubuntu ベースのディストリビューションの場合は、次のコマンドを実行します。

    ```
    sudo apt install openjdk-8-jdk
    ```

  - Red Hat ベースのディストリビューション (Amazon Linux を含む) の場合は、次のコマンドを実行します。

    ```
    sudo yum install java-1.8.0-openjdk
    ```

    詳細については、OpenJDK ドキュメントの「How to download and install prebuilt OpenJDK packages」を参照してください。
- AWS IoT Greengrass Core SDK for Python v1.5.0 以降。 AWS IoT Greengrass Core SDK for Python で `StreamManagerClient` を使用するには、以下を行う必要があります。
  - Python 3.7 以降をコアデバイスにインストールします。
  - SDK とその依存関係を Lambda 関数デプロイパッケージに含めます。このチュートリアルでは、手順を説明しています。

> **Tip**
>
> `StreamManagerClient` を Java または NodeJS で使用できます。コードの例については、GitHub の 「AWS IoT Greengrass Core SDK for Java」と「AWS IoT Greengrass Core SDK for Node.js」を参照してください。

- Greengrass グループ AWS リージョン と同じ の Amazon Kinesis Data Streams で**`MyKinesisStream`**作成された という名前の送信先ストリーム。詳細については、「Amazon Kinesis デベロッパーガイド」の「ストリームを作成する」を参照してください。

> **Note**
>
> このチュートリアルでは、ストリームマネージャーが Kinesis Data Streams にデータをエクスポートするため、 AWS アカウントに課金されます。料金の詳細については、「Amazon Kinesis Data Streams の料金」を参照してください。

料金が発生しないようにするには、Kinesis データストリームを作成せずにこのチュートリアルを実行します。この場合、ログを確認して、ストリームマネージャーがストリームを Kinesis Data Streams にエクスポートしようとしたことを確認します。

- 次の例に示すように、ターゲットデータストリームで `kinesis:PutRecords` アクションを許可する the section called “Greengrass グループのロール” に IAM ポリシーが追加されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecords"
            ],
            "Resource": [
                "arn:aws:kinesis:region:account-id:stream/MyKinesisStream"
            ]
        }
    ]
}
```

- コンピュータに AWS CLI インストールされ、設定されている 。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS Command Line Interfaceのインストール」および「AWS CLIの設定」を参照してください。

このチュートリアルのコマンドの例は、Linux やその他の Unix ベースのシステム向けに書かれています。Windows を使用している場合は、構文の違いの詳細については、AWS 「コマンドラインインターフェイスのパラメータ値の指定」を参照してください。

コマンドに JSON 文字列が含まれている場合、このチュートリアルでは、1 行形式の JSON の例を示しています。システムによっては、この形式を使用したほうが、コマンドの編集と実行を効率化できる場合があります。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Lambda 関数デプロイパッケージを作成する
2. 「Lambda 関数の作成」
3. 関数の定義とバージョンを作成する
4. ロガー定義とバージョンを作成する
5. コア定義バージョンの ARN を取得する
6. グループバージョンを作成する
7. デプロイを作成する
8. アプリケーションをテストする

このチュートリアルは完了までに約 30 分かかります。

ステップ 1: Lambda 関数デプロイパッケージを作成する

この手順では、Python 関数コードと依存関係を含む Lambda 関数デプロイパッケージを作成します。このパッケージは、後で AWS Lambdaで Lambda 関数を作成するときにアップロードします。Lambda 関数は AWS IoT Greengrass Core SDK を使用してローカルストリームを作成し、操作します。

> **Note**
>
> ユーザー定義の Lambda 関数は、AWS IoT Greengrass Core SDK を使用してストリームマネージャーと対話する必要があります。Greengrass ストリームマネージャーの要件の詳細については、「Greengrass ストリームマネージャーの要件」を参照してください。

1. v1.5.0 以降の AWS IoT Greengrass Core SDK for Python をダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
3. パッケージ依存関係をインストールして、Lambda 関数デプロイパッケージに SDK を含めます。

   1. `requirements.txt` ファイルのある SDK ディレクトリに移動します。このファイルには依存関係の一覧が記載されています。
   2. SDK の依存関係をインストールします。例えば、次の `pip` コマンドを実行して、現在のディレクトリにインストールします。

```
pip install --target . -r requirements.txt
```

4. 以下の Python コード関数を `transfer_stream.py` というローカルファイルに保存します。

> **Tip**
>
> Java と NodeJS を使用するコードの例については、GitHub の「AWS IoT Greengrass Core SDK for Java」と「AWS IoT Greengrass Core SDK for Node.js」を参照してください。

```
import asyncio
import logging
import random
import time

from greengrasssdk.stream_manager import (
    ExportDefinition,
    KinesisConfig,
    MessageStreamDefinition,
    ReadMessagesOptions,
    ResourceNotFoundException,
    StrategyOnFull,
    StreamManagerClient,
)


# This example creates a local stream named "SomeStream".
# It starts writing data into that stream and then stream manager automatically
 exports
# the data to a customer-created Kinesis data stream named "MyKinesisStream".
# This example runs forever until the program is stopped.

# The size of the local stream on disk will not exceed the default (which is 256
 MB).
# Any data appended after the stream reaches the size limit continues to be
 appended, and
# stream manager deletes the oldest data until the total stream size is back under
 256 MB.
# The Kinesis data stream in the cloud has no such bound, so all the data from this
 script is
```

```
# uploaded to Kinesis and you will be charged for that usage.


def main(logger):
    try:
        stream_name = "SomeStream"
        kinesis_stream_name = "MyKinesisStream"

        # Create a client for the StreamManager
        client = StreamManagerClient()

        # Try deleting the stream (if it exists) so that we have a fresh start
        try:
            client.delete_message_stream(stream_name=stream_name)
        except ResourceNotFoundException:
            pass

        exports = ExportDefinition(
            kinesis=[KinesisConfig(identifier="KinesisExport" + stream_name,
 kinesis_stream_name=kinesis_stream_name)]
        )
        client.create_message_stream(
            MessageStreamDefinition(
                name=stream_name,
 strategy_on_full=StrategyOnFull.OverwriteOldestData, export_definition=exports
            )
        )

        # Append two messages and print their sequence numbers
        logger.info(
            "Successfully appended message to stream with sequence number %d",
            client.append_message(stream_name, "ABCDEFGHIJKLMNO".encode("utf-8")),
        )
        logger.info(
            "Successfully appended message to stream with sequence number %d",
            client.append_message(stream_name, "PQRSTUVWXYZ".encode("utf-8")),
        )

        # Try reading the two messages we just appended and print them out
        logger.info(
            "Successfully read 2 messages: %s",
            client.read_messages(stream_name,
 ReadMessagesOptions(min_message_count=2, read_timeout_millis=1000)),
        )
```

```
        logger.info("Now going to start writing random integers between 0 and 1000
 to the stream")
        # Now start putting in random data between 0 and 1000 to emulate device
 sensor input
        while True:
            logger.debug("Appending new random integer to stream")
            client.append_message(stream_name, random.randint(0,
 1000).to_bytes(length=4, signed=True, byteorder="big"))
            time.sleep(1)

    except asyncio.TimeoutError:
        logger.exception("Timed out while executing")
    except Exception:
        logger.exception("Exception while running")


def function_handler(event, context):
    return


logging.basicConfig(level=logging.INFO)
# Start up this sample code
main(logger=logging.getLogger())
```

5. 以下の項目を `transfer_stream_python.zip` という名前のファイルに圧縮します。これが Lambda 関数デプロイパッケージです。

   - transfer_stream.py。アプリケーションロジック。
   - greengrasssdk。MQTT メッセージを発行する Python Greengrass Lambda 関数で必須のライブラリ。

     ストリームマネージャーオペレーションは、 AWS IoT Greengrass Core SDK for Python のバージョン 1.5.0 以降で使用できます。
   - AWS IoT Greengrass Core SDK for Python にインストールした依存関係 (`cbor2`ディレクトリなど) 。

   `zip` ファイルを作成するときは、これらの項目のみを含み、それらが含まれているフォルダは含みません。

ステップ 2: Lambda 関数を作成する

1. 関数の作成時に ARN を渡せるように、IAM ロールを作成します。

   JSON Expanded

   ```
   aws iam create-role --role-name Lambda_empty --assume-role-policy '{
       "Version": "2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Principal": {
                   "Service": "lambda.amazonaws.com"
               },
              "Action": "sts:AssumeRole"
           }
       ]
   }'
   ```

   JSON Single-line

   ```
   aws iam create-role --role-name Lambda_empty --assume-role-policy '{"Version":
    "2012-10-17", "Statement": [{"Effect": "Allow", "Principal": {"Service":
    "lambda.amazonaws.com"},"Action": "sts:AssumeRole"}]}'
   ```

   > **Note**
   >
   > AWS IoT Greengrass Greengrass Lambda 関数のアクセス許可は Greengrass グループロールで指定されているため、 はこのロールを使用しません。このチュートリアルでは、空のロールを作成します。

2. 出力から `Arn` をコピーします。
3. AWS Lambda API を使用して `TransferStream`関数を作成します。以下のコマンドでは、zip ファイルが現在のディレクトリにあるとします。

   - *`role-arn`* を、コピーした `Arn` に置き換えます。

   ```
   aws lambda create-function \
   ```

```
--function-name TransferStream \
--zip-file fileb://transfer_stream_python.zip \
--role role-arn \
--handler transfer_stream.function_handler \
--runtime python3.7
```

4. 関数のバージョンを発行します。

```
aws lambda publish-version --function-name TransferStream --description 'First
 version'
```

5. 発行されるバージョンのエイリアスを作成します。

   Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

```
aws lambda create-alias --function-name TransferStream --name GG_TransferStream --
function-version 1
```

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

6. 出力から `AliasArn` をコピーします。この値は、 関数を設定するときに使用します AWS IoT Greengrass。

これで、 関数を設定する準備が整いました AWS IoT Greengrass。

ステップ 3: 関数の定義とバージョンを作成する

この手順では、システム `GGStreamManager` Lambda 関数とユーザー定義の `TransferStream` Lambda 関数を参照する関数定義バージョンを作成します。 AWS IoT Greengrass API の使用時にストリームマネージャーを有効にするには、関数定義バージョンに `GGStreamManager`関数が含まれている必要があります。

1. システムとユーザー定義の Lambda 関数を含む初期バージョンを使用して関数定義を作成します。

次の定義バージョンにより、ストリームマネージャーがデフォルトのパラメータ設定で有効になります。カスタム設定を構成するには、対応するストリームマネージャーのパラメータの環境変数を定義する必要があります。例については、「. AWS IoT Greengrass uses default settings for parameters that are omitted」を参照してくださいthe section called “ストリームマネージャーの有効化、無効化、設定”。 `MemorySize` は少なくとも である必要があります`128000`。 は に設定`Pinned`する必要がありますtrue。

> **Note**
>
> 存続期間の長い (または固定された) Lambda 関数は、 AWS IoT Greengrass の起動後に自動的に起動し、独自のコンテナで実行し続けます。これはオンデマンド Lambda 関数とは対照的です。この関数は呼び出されたときに開始し、実行するタスクが残っていないときに停止します。詳細については、「the section called “ライフサイクル設定”」を参照してください。

- *`arbitrary-function-id`* は、関数の名前 (**`stream-manager`** など) に置き換えます。
- *`alias-arn`* を、`TransferStream` Lambda 関数のエイリアスの作成時にコピーした `AliasArn` に置き換えます。

JSON expanded

```
aws greengrass create-function-definition --name MyGreengrassFunctions --
initial-version '{
    "Functions": [
        {
            "Id": "arbitrary-function-id",
            "FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
            "FunctionConfiguration": {
                "MemorySize": 128000,
                "Pinned": true,
                "Timeout": 3
            }
        },
        {
            "Id": "TransferStreamFunction",
```

```
            "FunctionArn": "alias-arn",
            "FunctionConfiguration": {
                "Executable": "transfer_stream.function_handler",
                "MemorySize": 16000,
                "Pinned": true,
                "Timeout": 5
            }
        }
    ]
}'
```

JSON single

```
aws greengrass create-function-definition \
--name MyGreengrassFunctions \
--initial-version '{"Functions": [{"Id": "arbitrary-function-
id","FunctionArn": "arn:aws:lambda:::function:GGStreamManager:1",
 "FunctionConfiguration": {"Environment": {"Variables":
{"STREAM_MANAGER_STORE_ROOT_DIR": "/data","STREAM_MANAGER_SERVER_PORT":
 "1234","STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH": "20000"}},"MemorySize":
 128000,"Pinned": true,"Timeout": 3}},{"Id": "TransferStreamFunction",
 "FunctionArn": "alias-arn", "FunctionConfiguration": {"Executable":
 "transfer_stream.function_handler", "MemorySize": 16000,"Pinned":
 true,"Timeout": 5}}]}'
```

> **Note**
>
> `Timeout` は関数定義バージョンで必要ですが、`GGStreamManager` は使用しません。`Timeout` および、その他のグループレベルの設定については、「the section called "Greengrass Lambda 関数の実行の制御"」を参照してください。

2. 出力から `LatestVersionArn` をコピーします。この値を使用して、Core にデプロイするグループバージョンに、関数定義バージョンを追加します。

ステップ 4: ロガー定義とバージョンを作成する

グループのログ記録設定を定義します。このチュートリアルでは、コアデバイスのファイルシステムにログを書き込むように、 AWS IoT Greengrass システムコンポーネント、ユーザー定義の Lambda 関数、コネクタを設定します。ログを使用して、発生する可能性のある問題のトラブルシューティン

グを行うことができます。詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

1. 初期バージョンを含む関数定義を作成します。

   JSON Expanded

```
aws greengrass create-logger-definition --name "LoggingConfigs" --initial-
version '{
    "Loggers": [
        {
            "Id": "1",
            "Component": "GreengrassSystem",
            "Level": "INFO",
            "Space": 10240,
            "Type": "FileSystem"
        },
        {
            "Id": "2",
            "Component": "Lambda",
            "Level": "INFO",
            "Space": 10240,
            "Type": "FileSystem"
        }
    ]
}'
```

   JSON Single-line

```
aws greengrass create-logger-definition \
    --name "LoggingConfigs" \
    --initial-version '{"Loggers":
[{"Id":"1","Component":"GreengrassSystem","Level":"INFO","Space":10240,"Type":"FileSyste
{"Id":"2","Component":"Lambda","Level":"INFO","Space":10240,"Type":"FileSystem"}]}'
```

2. 出力からロガー定義の `LatestVersionArn` をコピーします。この値を使用して、コアにデプロイするグループバージョンに、ロガー定義のバージョンを追加します。

ステップ 5: コア定義バージョンの ARN を取得する

新しいグループバージョンに追加するコア定義バージョンの ARN を取得します。グループバージョンをデプロイするには、グループバージョンが、1 つのコアが含まれているコア定義バージョンを参照する必要があります。

1. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))
   [0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   >
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [Deployments] (デプロイ) タブに表示されます。

2. 出力からターゲットグループの `Id` をコピーします。この情報は、コア定義バージョンを取得するときとグループをデプロイするときに使用します。
3. 出力から `LatestVersion` をコピーします。これは、グループに追加された最後のバージョンの ID です。この情報は、コア定義バージョンを取得するときに使用します。
4. コア定義バージョンの ARN を取得します。

   a. グループバージョンを取得します。

      - *group-id* は、コピーしたグループの `Id` に置き換えます。
      - *group-version-id* は、コピーしたグループの `LatestVersion` に置き換えます。

      ```
      aws greengrass get-group-version \
      --group-id group-id \
      ```

```
--group-version-id group-version-id
```

b. 出力から CoreDefinitionVersionArn をコピーします。この値を使用して、コアにデプロイするグループバージョンにコア定義バージョンを追加します。

ステップ 6: グループバージョンを作成する

デプロイするすべてのエンティティを含むグループバージョンを作成する準備ができました。ここでは、各コンポーネントタイプのターゲットバージョンを参照するグループバージョンを作成します。このチュートリアルでは、コア定義バージョン、関数定義バージョン、ロガー定義バージョンを含めます。

1. グループバージョンを作成します。

   - *group-id* は、コピーしたグループの Id に置き換えます。
   - *core-definition-version-arn* は、コピーしたコア定義バージョンの CoreDefinitionVersionArn に置き換えます。
   - *function-definition-version-arn* は、コピーした新しい関数定義バージョンの LatestVersionArn に置き換えます。
   - *logger-definition-version-arn* は、コピーした新しいロガー定義バージョンの LatestVersionArn に置き換えます。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
--function-definition-version-arn function-definition-version-arn \
--logger-definition-version-arn logger-definition-version-arn
```

2. 出力から Version をコピーします。これは新しいグループバージョンの ID です。

ステップ 7: デプロイを作成する

Core デバイスにグループをデプロイします。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

```
ps aux | grep -E 'greengrass.*daemon'
```

出力に root で実行中の /greengrass/ggc/packages/*ggc-version*/bin/daemon のエントリが含まれていれば、デーモンは実行されています。

> **Note**
>
> パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

b. デーモンを開始するには、以下を実行します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. デプロイを作成します。

   - *group-id* は、コピーしたグループの Id に置き換えます。
   - *group-version-id* は、コピーした新しいグループバージョンの Version に置き換えます。

```
aws greengrass create-deployment \
--deployment-type NewDeployment \
--group-id group-id \
--group-version-id group-version-id
```

3. 出力から DeploymentId をコピーします。
4. デプロイのステータスを取得します。

   - *group-id* は、コピーしたグループの Id に置き換えます。
   - *deployment-id* は、コピーしたデプロイの DeploymentId に置き換えます。

```
aws greengrass get-deployment-status \
--group-id group-id \
--deployment-id deployment-id
```

ステータスが Success の場合、デプロイは成功しています。トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

ステップ 8: アプリケーションをテストする

この TransferStream Lambda 関数は、シミュレートされたデバイスデータを生成します。ストリームマネージャーがターゲットの Kinesis データストリームにエクスポートするストリームにデータを書き込みます。

1. Amazon Kinesis コンソールの [Kinesis data streams] (Kinesis データストリーム) で、[MyKinesisStream] を選択します。

   > **Note**
   >
   > ターゲットの Kinesis データストリームを使用せずにチュートリアルを実行した場合は、ストリームマネージャーのログファイルを確認します (GGStreamManager)。エラーメッセージに export stream MyKinesisStream doesn't exist が含まれている場合、テストは成功します。このエラーは、サービスがストリームにエクスポートしようとしましたが、ストリームが存在しないことを意味します。

2. [MyKinesisStream] ページで、[Monitoring (モニタリング)] を選択します。テストが成功すると、Put Records (レコードの配置) グラフにデータが表示されます。接続によっては、データが表示されるまでに 1 分かかることがあります。

   > **Important**
   >
   > テストが終了したら、Kinesis データストリームを削除して、それ以上の料金が発生しないようにします。
   > または、次のコマンドを実行して Greengrass デーモンを停止します。これにより、テストを続行する準備が整うまで、コアがメッセージを送信できなくなります。
   >
   > ```
   > cd /greengrass/ggc/core/
   > sudo ./greengrassd stop
   > ```

3. コアから TransferStream Lambda 関数を削除します。

a. 新しいグループバージョンを作成するには、the section called “グループバージョンを作成する” に従います。ただし、`create-group-version` コマンド内の `--function-definition-version-arn` オプションを削除します。または、TransferStream Lambda 関数を含まない関数定義バージョンを作成します。

> **Note**
>
> デプロイされたグループのバージョンからシステム `GGStreamManager` Lambda 関数を省略すると、コアでのストリーム管理が無効になります。

b. 新しいグループバージョンをデプロイするには、the section called “デプロイを作成する” に従います。

ロギング情報を表示したり、ストリームに関する問題をトラブルシューティングしたりするには、`TransferStream` および `GGStreamManager` 関数のログを確認します。ファイルシステムの AWS IoT Greengrass ログを読み取るためのrootアクセス許可が必要です。

- `TransferStream` は、ログエントリを *`greengrass-root`*`/ggc/var/log/user/`*`region`*`/`*`account-id`*`/TransferStream.log` に書き込みます。
- `GGStreamManager` は、ログエントリを *`greengrass-root`*`/ggc/var/log/system/GGStreamManager.log` に書き込みます。

さらにトラブルシューティング情報が必要な場合は、Lambda ログレベルを `DEBUG` に設定し、新しいグループバージョンを作成してデプロイできます。

関連情報

- データストリームの管理
- the section called “ストリームを操作するために StreamManagerClient を使用する”
- the section called “AWS クラウド でサポートされている送信先のエクスポート設定”
- the section called “ストリームマネージャーの設定”
- the section called “データストリームのエクスポート (コンソール)”
- コマンドリファレンスの AWS Identity and Access Management (IAM) コマンド AWS CLI
- コマンドリファレンスAWS Lambdaの コマンド AWS CLI
- コマンドリファレンスAWS IoT Greengrassの コマンド AWS CLI

シークレットを AWS IoT Greengrass コアにデプロイする

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

AWS IoT Greengrass では、パスワード、トークン、その他のシークレットをハードコーディングすることなく、Greengrass デバイスからのサービスとアプリケーションで認証できます。

AWS Secrets Manager は、シークレットをクラウドに安全に保存および管理するために使用できるサービスです。 は Secrets Manager を Greengrass コアデバイスに AWS IoT Greengrass 拡張するため、コネクタと Lambda 関数はローカルシークレットを使用してサービスやアプリケーションを操作できます。たとえば、Twilio 通知コネクタは、ローカルに保存された認証トークンを使用します。

シークレットを Greengrass グループ内に統合するには、Secrets Manager シークレットを参照するグループリソースを作成します。このシークレットリソースは ARN に基づいてクラウドシークレットを参照します。シークレットリソースを作成、管理、および使用する方法については、「the section called “シークレットリソースを使用する”」を参照してください。

AWS IoT Greengrass は、転送中および保管中にシークレットを暗号化します。グループのデプロイ中に、 は Secrets Manager からシークレット AWS IoT Greengrass を取得し、Greengrass コアに暗号化されたローカルコピーを作成します。Secrets Manager でクラウドシークレットを更新した後、グループを再デプロイして、更新された値をコアに伝達します。

以下の図では、シークレットを Core にデプロイする大まかなプロセスを示しています。シークレットは転送中および保管時に暗号化されます。

AWS IoT Greengrass を使用してシークレットをローカルに保存すると、次の利点があります。

- コードから分離 (ハードコードされない)。これにより、一元管理された認証情報がサポートされ、機密データを侵害のリスクから守ることができます。
- オフラインシナリオで利用可能。コネクタと関数はインターネットから切断された状態でローカルサービスとソフトウェアに安全にアクセスできます。
- シークレットへのアクセスをコントロール。グループの承認されたコネクタと関数のみがシークレットにアクセスできます。 AWS IoT Greengrass はプライベートキー暗号を使用してシークレットを保護します。シークレットは転送中および保管時に暗号化されます。詳細については、「the section called “シークレットの暗号化”」を参照してください。
- 更新をコントロール。Secrets Manager でシークレットを更新した後、Greengrass グループを再デプロイして、シークレットのローカルコピーを更新します。詳細については、「the section called “シークレットの作成と管理”」を参照してください。

> **Important**
>
> AWS IoT Greengrass は、クラウドバージョンがローテーションされた後、ローカルシークレットの値を自動的に更新しません。ローカル値を更新するには、グループを再デプロイする必要があります。

シークレットの暗号化

AWS IoT Greengrass は、転送中および保管中のシークレットを暗号化します。

> **Important**
>
> ユーザー定義の Lambda 関数がシークレットを安全に処理することと、シークレットに格納されている機密データをログに記録しないことを確認してください。詳細については、「AWS Secrets Manager ユーザーガイド」の「Lambda 関数のログ記録とデバッグのリスクを軽減する」を参照してください。このドキュメントでは特に回転関数について言及していますが、こうした推奨事項は Greengrass Lambda 関数にも適用されます。

転送時の暗号化

AWS IoT Greengrass は Transport Layer Security (TLS) を使用して、インターネットとローカルネットワーク経由のすべての通信を暗号化します。これにより、転送中のシークレットが保護されます。この保護は、シークレットが Secrets Manager から取得されてコアにデプロイされると

きに行われます。サポートされている TLS 暗号スイートについては、「the section called “TLS 暗号スイートのサポート”」を参照してください。

保管時の暗号化

AWS IoT Greengrass は、 で指定されたプライベートキー`config.json`を使用して、コアに保存されているシークレットを暗号化します。このため、ローカルシークレットの保護には、プライベートキーの安全な保存が重要です。責任 AWS 共有モデルでは、コアデバイス上のプライベートキーの安全なストレージを保証するのはお客様の責任です。

AWS IoT Greengrass は、プライベートキーストレージの 2 つのモードをサポートしています。

- ハードウェアセキュリティモジュールの使用。詳細については、「the section called “ハードウェアセキュリティ統合”」を参照してください。

> **Note**
>
> 現在、 は、ハードウェアベースのプライベートキーを使用する場合のローカルシークレットの暗号化と復号のための PKCS#1 v1.5 パディングメカニズムのみ AWS IoT Greengrass をサポートしています。ベンダーが提供する指示に従ってハードウェアベースのプライベートキーを手動で生成する場合は、PKCS#1 v1.5. AWS IoT Greengrass を選択し、最適な非対称暗号化パディング (OAEP) をサポートしていないことを確認してください。

- ファイルシステムアクセス許可の使用 (デフォルト)。

プライベートキーは、ローカルシークレットの暗号化用のデータキーを保護するために使用されます。データキーは、グループデプロイごとに更新されます。

AWS IoT Greengrass コアは、プライベートキーにアクセスできる唯一のエンティティです。シークレットリソースに関連付けられている Greengrass コネクタまたは Lambda 関数はコアからシークレットを取得します。

要件

以下に示しているのは、ローカルシークレットをサポートするための要件です。

- AWS IoT Greengrass Core v1.7 以降を使用している必要があります。
- ローカルシークレットの値を取得するには、ユーザー定義の Lambda 関数で AWS IoT Greengrass Core SDK v1.3.0 以降を使用する必要があります。

- ローカルシークレットの暗号化に使用されるプライベートキーは、Greengrass 設定ファイルで指定する必要があります。デフォルトでは、 はファイルシステムに保存されているコアプライベートキー AWS IoT Greengrass を使用します。独自のプライベートキーを指定する場合は、「the section called “シークレット暗号化用のプライベートキーを指定する”」を参照してください。RSA キータイプのみがサポートされています。

> **Note**
>
> 現在、 は、ハードウェアベースのプライベートキーを使用する場合のローカルシークレットの暗号化と復号のための PKCS#1 v1.5 パディングメカニズムのみ AWS IoT Greengrass をサポートしています。ベンダーが提供する指示に従ってハードウェアベースのプライベートキーを手動で生成する場合は、PKCS#1 v1.5. AWS IoT Greengrass を選択し、最適な非対称暗号化パディング (OAEP) をサポートしていないことを確認してください。

- AWS IoT Greengrass シークレット値を取得するアクセス許可を付与する必要があります。これにより、 AWS IoT Greengrass はグループのデプロイ中に値を取得できます。デフォルトの Greengrass サービスロールを使用している場合、 は greengrass- で始まる名前のシークレットに AWS IoT Greengrass 既にアクセスできます。アクセス権限をカスタマイズするには、「the section called “AWS IoT Greengrass シークレット値の取得を に許可する”」を参照してください。

> **Note**
>
> この命名規則を使用して、アクセス許可をカスタマイズした場合でも、アクセスが AWS IoT Greengrass 許可されているシークレットを特定することをお勧めします。コンソールはシークレットを読み取るために異なるアクセス許可を使用するため、取得するアクセス許可 AWS IoT Greengrass を持たないシークレットをコンソールで選択できる可能性があります。命名規則を使用すると、デプロイエラーにつながるアクセス許可の競合を回避するのに役立ちます。

シークレット暗号化用のプライベートキーを指定する

この手順では、ローカルシークレット暗号化に使用されるプライベートキーのパスを提供します。ここでは、最少長の 2048 ビットの RSA キーを使用する必要があります。 AWS IoT Greengrass コアで使用されるプライベートキーの詳細については、「」を参照してくださいthe section called “セキュリティプリンシパル”。

AWS IoT Greengrass は、ハードウェアベースまたはファイルシステムベース (デフォルト) の 2 つのプライベートキーストレージモードをサポートしています。詳細については、「the section called "シークレットの暗号化"」を参照してください。

ファイルシステムのコアプライベートキーを使用するデフォルト設定を変更する場合にのみ以下の手順を実行します。これらの手順は、開始方法チュートリアルのモジュール 2 で説明したようにグループと Core を作成したとします。

1. `/greengrass-root/config` ディレクトリにある `config.json` ファイルを開きます。

   > **Note**
   >
   > *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

2. `crypto.principals.SecretsManager` オブジェクトの `privateKeyPath` プロパティで、プライベートキーのパスを入力します。

   - プライベートキーがファイルシステムに保存されている場合は、キーへの絶対パスを指定します。例:

     ```
     "SecretsManager" : {
       "privateKeyPath" : "file:///somepath/hash.private.key"
     }
     ```

   - プライベートキーがハードウェアセキュリティモジュール (HSM) に保存されている場合は、RFC 7512 PKCS#11 URI スキームを使用してパスを指定します。例:

     ```
     "SecretsManager" : {
       "privateKeyPath" : "pkcs11:object=private-key-label;type=private"
     }
     ```

     詳細については、「the section called "ハードウェアセキュリティ設定"」を参照してください。

     > **Note**
     >
     > 現在、 は、ハードウェアベースのプライベートキーを使用する場合のローカルシークレットの暗号化と復号のための PKCS#1 v1.5 パディングメカニズムのみ AWS IoT Greengrass をサポートしています。ベンダーが提供する指示に従ってハード

ウェアベースのプライベートキーを手動で生成する場合は、PKCS#1 v1.5. AWS IoT Greengrass を選択し、最適な非対称暗号化パディング (OAEP) をサポートしていないことを確認してください。

AWS IoT Greengrass シークレット値の取得を に許可する

この手順では、 がシークレットの値 AWS IoT Greengrass を取得できるようにするインラインポリシーを Greengrass サービスロールに追加します。

シークレットに AWS IoT Greengrass カスタムアクセス許可を付与する場合、または Greengrass サービスロールに `AWSGreengrassResourceAccessRolePolicy`管理ポリシーが含まれていない場合のみ、この手順を実行します。 は greengrass- で始まる名前のシークレットへのアクセス`AWSGreengrassResourceAccessRolePolicy`を許可します。

1. 以下の CLI コマンドを実行して、Greengrass サービスロールの ARN を取得します。

   ```
   aws greengrass get-service-role-for-account --region region
   ```

   返された ARN にはロール名が含まれています。

   ```
   {
     "AssociatedAt": "time-stamp",
     "RoleArn": "arn:aws:iam::account-id:role/service-role/role-name"
   }
   ```

   以下の手順で ARN または名前を使用します。

2. `secretsmanager:GetSecretValue` アクションを許可するインラインポリシーを追加します。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

   シークレットを明示的に指定するかワイルドカード * 命名規則を使用することで、詳細なアクセスを許可できます。バージョニングまたはタグ付けされたシークレットへの条件付きアクセスを許可することもできます。たとえば、次のポリシーでは、 AWS IoT Greengrass が指定されたシークレットのみを読み取ることを許可します。

   ```
   {
       "Version": "2012-10-17",
   ```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:region:account-id:secret:greengrass-
SecretA-abc",
                "arn:aws:secretsmanager:region:account-id:secret:greengrass-
SecretB-xyz"
            ]
        }
    ]
}
```

> **Note**
>
> カスタマーマネージド AWS KMS キーを使用してシークレットを暗号化する場合、Greengrass サービスロールも `kms:Decrypt`アクションを許可する必要があります。

Secrets Manager の IAM ポリシーの詳細については、「AWS Secrets Manager ユーザーガイド」の「AWS Secrets Managerの認証とアクセスコントロール」および「AWS Secrets Managerの IAM ポリシーまたはシークレットポリシーで使用できるアクション、リソース、コンテキストキー」を参照してください。

関連情報

- AWS Secrets Manager ユーザーガイドの「 AWS Secrets Managerとは」
- PKCS #1: RSA Encryption Version 1.5

シークレットリソースを使用する

AWS IoT Greengrass はシークレットリソースを使用して、 のシークレット AWS Secrets Manager を Greengrass グループに統合します。シークレットリソースとは、Secrets Manager シークレット

への参照を意味します。詳細については、「Core にシークレットをデプロイする」を参照してください。

AWS IoT Greengrass コアデバイスでは、コネクタと Lambda 関数はシークレットリソースを使用して、パスワード、トークン、その他の認証情報をハードコーディングすることなく、 サービスやアプリケーションで認証できます。

シークレットの作成と管理

Greengrass グループで、シークレットリソースは Secrets Manager シークレットの ARN を参照します。シークレットリソースをコアにデプロイすると、シークレットの値は暗号化され、関連付けられているコネクタおよび Lambda 関数で使用できるようになります。詳細については、「the section called “シークレットの暗号化”」を参照してください。

Secrets Manager を使用して、シークレットのクラウドバージョンを作成および管理します。 AWS IoT Greengrass を使用して、シークレットリソースを作成、管理、デプロイします。

> **Important**
>
> Secrets Manager でシークレットを更新するためのベストプラクティスに従うことをお勧めします。次に、Greengrass グループをデプロイして、シークレットのローカルコピーを更新します。詳細については、「 AWS Secrets Manager ユーザーガイド」の「シーAWS Secrets Manager クレットのローテーション」を参照してください。

Greengrass Core でシークレットを使用可能にするには

1. Secrets Manager でシークレットを作成します。これは、Secrets Manager で一元的に保存および管理されるシークレットのクラウドバージョンです。管理タスクには、シークレットの値の更新とリソースポリシーの適用が含まれます。
2. でシークレットリソースを作成します AWS IoT Greengrass。これは、ARN に基づいてクラウドシークレットを参照するグループリソースの一種です。シークレットはグループごとに 1 回のみ参照できます。
3. コネクタまたは Lambda 関数を設定する。リソースをコネクタまたは関数に関連付けるには、対応するパラメータまたはプロパティを指定する必要があります。これにより、ローカルにデプロイされたシークレットリソースの値を取得できます。詳細については、「the section called “ローカルシークレットの使用”」を参照してください。

4. Greengrass グループをデプロイする。デプロイ中、 AWS IoT Greengrass はクラウドシークレットの値を取り出し、Core のローカルシークレットを作成 (または更新) します。

Secrets Manager は、 が AWS IoT Greengrass シークレット値を取得する AWS CloudTrail たびにイベントを記録します。 AWS IoT Greengrass は、ローカルシークレットのデプロイまたは使用に関連するイベントを記録しません。Secrets Manager のログ記録の詳細については、AWS Secrets Manager 「 ユーザーガイド」の「シーク AWS Secrets Manager レットの使用を監視する」を参照してください。

シークレットリソースにステージングラベルを含める

Secrets Manager はステージングラベルを使用して、シークレットの値の特定バージョンを識別します。ステージングラベルは、システム定義またはユーザー定義とすることができます。Secrets Manager は、`AWSCURRENT` ラベルをシークレット値の最新バージョンに割り当てます。ステージングラベルは一般に、シークレットの更新を管理するために使用されます。Secrets Manager のバージョニングの詳細については、「AWS Secrets Manager ユーザーガイド」の「Key terms and concepts for AWS Secrets Manager」(の主な用語と概念) を参照してください。

シークレットリソースには必ず `AWSCURRENT` ステージングラベルを含めます。また、Lambda 関数やコネクタに必要な場合、他のステージングラベルを含めることもできます。グループのデプロイ中に、 はグループ内で参照されているステージングラベルの値 AWS IoT Greengrass を取得し、コアで対応する値を作成または更新します。

シークレットリソースの作成と管理 (コンソール)

シークレットリソースの作成 (コンソール)

AWS IoT Greengrass コンソールでは、グループのリソースページのシークレットタブからシークレットリソースを作成および管理します。シークレットリソースを作成してグループに追加するチュートリアルについては、「the section called “シークレットリソースを作成する方法 (コンソール)”」と「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

> **Note**
>
> 別の方法として、コネクタや Lambda 関数を設定するときに、コンソールでシークレットとシークレットリソースを作成することもできます。これはコネクタの [Configure parameters] (パラメータの設定) ページか、Lambda 関数の [Resources] (リソース) ページから実行できます。

シークレットリソースの管理 (コンソール)

Greengrass グループのシークレットリソース管理タスクとして、グループへのシークレットリソースの追加、グループからのシークレットリソースの削除、シークレットリソース内にあるステージングラベルセットの変更が挙げられます。

Secrets Manager とは異なるシークレットを参照する場合は、そのシークレットを使用するコネクタをすべて編集する必要があります。

1. グループの設定ページで、[コネクタ] を選択します。
2. コネクタのコンテキストメニューから [Edit] (編集) を選択します。
3. [パラメータの編集] ページで、シークレット ARN が変更されたことを知らせるメッセージが表示されます。変更を確定するには、[Save (保存)] を選択します。

Secrets Manager でシークレットを削除する場合は、グループ、およびそれを参照するコネクタと Lambda 関数から、対応するシークレットリソースを削除します。その操作を行わないと、グループのデプロイ中に、 AWS IoT Greengrass から、シークレットが見つからない旨のエラーが返ります。また、必要に応じて Lambda 関数コードを更新します。

シークレットリソースの作成と管理 (CLI)

シークレットリソースの作成 (CLI)

AWS IoT Greengrass API では、シークレットはグループリソースの一種です。以下の例では、`MySecretResource` というシークレットリソースを含む初期バージョンでリソース定義を作成します。シークレットリソースを作成してグループバージョンに追加するチュートリアルについては、「the section called “コネクタの使用を開始する (CLI)”」を参照してください。

シークレットリソースは、対応する Secrets Manager シークレットの ARN を参照し、常に含まれる `AWSCURRENT` に加えて 2 つのステージングラベルを含みます。

```
aws greengrass create-resource-definition --name MyGreengrassResources --initial-
version '{
    "Resources": [
        {
            "Id": "my-resource-id",
            "Name": "MySecretResource",
            "ResourceDataContainer": {
                "SecretsManagerSecretResourceData": {
                    "ARN": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:greengrass-SomeSecret-KUj89s",
                    "AdditionalStagingLabelsToDownload": [
                        "Label1",
                        "Label2"
                    ]
                }
            }
        }
    ]
}'
```

シークレットリソースの管理 (CLI)

Greengrass グループのシークレットリソース管理タスクとして、グループへのシークレットリソースの追加、グループからのシークレットリソースの削除、シークレットリソース内にあるステージングラベルセットの変更が挙げられます。

AWS IoT Greengrass API では、これらの変更は バージョンを使用して実装されます。

AWS IoT Greengrass API はバージョンを使用してグループを管理します。バージョンは変更不可であるため、グループのクライアントデバイス、関数、リソースといったグループコンポーネントを

追加または変更するには、新規または更新済みコンポーネントのバージョンを作成する必要があります。その後、各コンポーネントのターゲットバージョンを含むグループバージョンを作成およびデプロイします。グループの詳細については、「the section called “AWS IoT Greengrass グループ”」を参照してください。

例えば、シークレットリソースの一連のステージングラベルを変更するには、以下の操作を実行します。

1. 更新されたシークレットリソースを含むリソース定義バージョンを作成します。以下の例では、前のセクションのシークレットリソースに 3 つ目のステージングラベルを追加します。

> **Note**
>
> バージョンにさらにリソースを追加するには、それらのリソースを `Resources` 配列に含めます。

```
aws greengrass create-resource-definition --name MyGreengrassResources --initial-
version '{
    "Resources": [
        {
            "Id": "my-resource-id",
            "Name": "MySecretResource",
            "ResourceDataContainer": {
                "SecretsManagerSecretResourceData": {
                    "ARN": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:greengrass-SomeSecret-KUj89s",
                    "AdditionalStagingLabelsToDownload": [
                        "Label1",
                        "Label2",
                        "Label3"
                    ]
                }
            }
        }
    ]
}'
```

2. シークレットリソースの ID が変更された場合は、そのシークレットリソースを使用するコネクタと関数を更新します。新しいバージョンでは、リソース ID に対応するパラメータまたはプロパ

ティを更新します。シークレットの ARN が変更された場合は、シークレットを使用するすべてのコネクタに対応するパラメータも更新する必要があります。

> **Note**
>
> リソース ID は、お客様が提供する任意の識別子です。

3. Core に送信する各コンポーネントのターゲットバージョンを含むグループバージョンを作成します。
4. グループバージョンをデプロイします。

シークレットリソース、コネクタ、関数を作成してデプロイする方法を示すチュートリアルについては、「the section called “コネクタの使用を開始する (CLI)”」を参照してください。

Secrets Manager でシークレットを削除する場合は、グループ、およびそれを参照するコネクタと Lambda 関数から、対応するシークレットリソースを削除します。その操作を行わないと、グループのデプロイ中に、 AWS IoT Greengrass から、シークレットが見つからない旨のエラーが返ります。また、必要に応じて Lambda 関数コードを更新します。ローカルシークレットを削除するには、対応するシークレットリソースを含まないリソース定義バージョンをデプロイします。

コネクタと Lambda 関数でのローカルシークレットの使用

Greengrass コネクタと Lambda 関数はローカルシークレットを使用して、サービスやアプリケーションとやり取りします。デフォルトでは `AWSCURRENT` 値が使用されますが、シークレットリソースに含まれている他のステージングラベルの値も使用できます。

コネクタと関数はローカルシークレットにアクセスする前に設定する必要があります。これにより、シークレットリソースがコネクタまたは関数に関連付けられます。

Connector

コネクタがローカルシークレットにアクセスする必要がある場合、シークレットにアクセスするためのパラメータはお客様が設定します。

- AWS IoT Greengrass コンソールでこれを行う方法については、「」を参照してくださいthe section called “コネクタの使用を開始する (コンソール)”。
- CLI でこれを行う方法については、 AWS IoT Greengrass 「」を参照してくださいthe section called “コネクタの使用を開始する (CLI)”。

個々のコネクタの要件については、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

シークレットにアクセスして使用するためのロジックは、コネクタに組み込まれています。

Lambda 関数

Greengrass Lambda 関数にローカルシークレットへのアクセスを許可するには、その関数のプロパティを設定します。

- AWS IoT Greengrass コンソールでこれを行う方法については、「」を参照してくださいthe section called “シークレットリソースを作成する方法 (コンソール)”。
- AWS IoT Greengrass API でこれを行うには、 `ResourceAccessPolicies`プロパティで次の情報を指定します。
  - `ResourceId`: Greengrass グループのシークレットリソースの ID。これは、対応する Secrets Manager シークレットの ARN を参照するリソースです。
  - `Permission`: 関数がリソースに対して付与されるアクセス許可のタイプ。シークレットリソースに対しては `ro` (読み取り専用) アクセス許可のみがサポートされています。

以下の例では、`MyApiKey` シークレットリソースにアクセスできる Lambda 関数を作成します。

```
aws greengrass create-function-definition --name MyGreengrassFunctions --initial-
version '{
    "Functions": [
        {
            "Id": "MyLambdaFunction",
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1",
            "FunctionConfiguration": {
                "Pinned": false,
                "MemorySize": 16384,
                "Timeout": 10,
                "Environment": {
                    "ResourceAccessPolicies": [
                        {
                            "ResourceId": "MyApiKey",
                            "Permission": "ro"
                        }
                    ],
                    "AccessSysfs": true
                }
```

```
            }
        }
    ]
}'
```

実行時にローカルシークレットにアクセスするには、Greengrass Lambda 関数は AWS IoT Greengrass Core SDK (v1.3.0 以降) の`secretsmanager`クライアントから`get_secret_value`関数を呼び出します。

次の例は、 AWS IoT Greengrass Core SDK for Python を使用してシークレットを取得する方法を示しています。シークレットの名前を `get_secret_value` 関数に渡します。`SecretId` は、Secrets Manager シークレット (シークレットリソースではなく) の名前または ARN です。

```
import greengrasssdk

secrets_client = greengrasssdk.client("secretsmanager")
secret_name = "greengrass-MySecret-abc"


def function_handler(event, context):
    response = secrets_client.get_secret_value(SecretId=secret_name)
    secret = response.get("SecretString")
```

テキストタイプのシークレットの場合、`get_secret_value` 関数は文字列を返します。バイナリタイプのシークレットの場合は、Base 64 でエンコードされた文字列を返します。

> **Important**
>
> ユーザー定義の Lambda 関数がシークレットを安全に処理することと、シークレットに格納されている機密データをログに記録しないことを確認してください。詳細については、「AWS Secrets Manager ユーザーガイド」の「Lambda 関数のログ記録とデバッグのリスクを軽減する」を参照してください。このドキュメントでは特に回転関数について言及していますが、こうした推奨事項は Greengrass Lambda 関数にも適用されます。

デフォルトでは、シークレットの現在の値が返されます。これは AWSCURRENT ステージングラベルのアタッチ先のバージョンです。別のバージョンにアクセスするには、対応するステージングラベルの名前をオプションの VersionStage 引数に渡します。例:

```
import greengrasssdk

secrets_client = greengrasssdk.client("secretsmanager")
secret_name = "greengrass-TestSecret"
secret_version = "MyTargetLabel"


# Get the value of a specific secret version
def function_handler(event, context):
    response = secrets_client.get_secret_value(
        SecretId=secret_name, VersionStage=secret_version
    )
    secret = response.get("SecretString")
```

get_secret_value を呼び出す別の関数の例については、「Lambda 関数デプロイパッケージを作成する」を参照してください。

シークレットリソースを作成する方法 (コンソール)

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

このチュートリアルでは、 AWS Management Console を使用して Greengrass グループにシークレットリソースを追加する方法を示します。シークレットリソースは、 からのシークレットへの参照です AWS Secrets Manager。詳細については、「Core にシークレットをデプロイする」を参照してください。

AWS IoT Greengrass コアデバイスでは、コネクタと Lambda 関数はシークレットリソースを使用して、パスワード、トークン、その他の認証情報をハードコーディングすることなく、 サービスやアプリケーションで認証できます。

このチュートリアルでは、まず AWS Secrets Manager コンソールでシークレットを作成します。次に、 AWS IoT Greengrass コンソールで、グループの [Resources] (リソース) ページからシークレットリソースを Greengrass グループに追加します。このシークレットリソースは、Secrets Manager

シークレットを参照します。後で、シークレットリソースを Lambda 関数にアタッチします。これにより、関数はローカルシークレットの値を取得できます。

> **Note**
>
> 別の方法として、コネクタや Lambda 関数を設定するときに、コンソールでシークレットとシークレットリソースを作成することもできます。これはコネクタの [Configure parameters] (パラメータの設定) ページか、Lambda 関数の [Resources] (リソース) ページから実行できます。
>
> シークレットのパラメータを含むコネクタのみがシークレットにアクセスできます。ローカルに保存された認証トークンを Twilio 通知コネクタがどのように使用するかを示すチュートリアルについては、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Secrets Manager シークレットを作成する
2. グループにシークレットリソースを追加する
3. Lambda 関数デプロイパッケージを作成する
4. 「Lambda 関数の作成」
5. 関数をグループに追加する
6. シークレットリソースを関数にアタッチする
7. サブスクリプションをグループに追加する
8. グループをデプロイする
9. the section called “Lambda 関数をテストする”

このチュートリアルは完了までに約 20 分かかります。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.7 以降)。Greengrass のグループまたは Core を作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。入門チュートリアルには、 AWS IoT Greengrass Core ソフトウェアをインストールする手順も含まれています。

- AWS IoT Greengrass は、ローカルシークレットをサポートするように設定する必要があります。詳細については、「シークレットの要件」を参照してください。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- ローカルシークレットの値を取得するには、ユーザー定義の Lambda 関数で AWS IoT Greengrass Core SDK v1.3.0 以降を使用する必要があります。

ステップ 1: Secrets Manager シークレットを作成する

このステップでは、 AWS Secrets Manager コンソールを使用してシークレットを作成します。

1. AWS Secrets Manager コンソール にサインインします。

> **Note**
>
> このプロセスの詳細については、「AWS Secrets Manager ユーザーガイド」の「ステップ 1: AWS Secrets Managerでシークレットを作成および保存する」を参照してください。

2. [新しいシークレットを保存] を選択します。
3. [Choose secret type] (シークレットの種類を選択) で、[Other type of secrets] (他の種類のシークレット) を選択します。
4. [このシークレットに保存するキーと値のペアを指定します] で以下の操作を行います。
   - [キー] に「**`test`**」と入力します。
   - [Value (値)] に「**`abcdefghi`**」と入力します。
5. 暗号化には aws/secretsmanager を暗号キーとして選択した状態で、[Next] (次へ) を選択します。

> **Note**
>
> Secrets Manager がアカウントで作成するデフォルトの AWS マネージドキー AWS KMS を使用する場合、 は課金されません。

6. [シークレット名] に「**`greengrass-TestSecret`**」と入力し、[次へ] を選択します。

> **Note**
>
> デフォルトでは、Greengrass サービスロールにより AWS IoT Greengrass 、 は greengrass- で始まる名前のシークレットの値を取得できます。詳細については、「シークレットの要件」を参照してください。

7. このチュートリアルではローテーションは不要であるため、[disable automatic rotation] (自動ローテーションを無効化)、Next (次へ) の順に選択します。
8. [確認] ページで、設定を確認し、[保存] を選択します。

次に、シークレットを参照する Greengrass グループにシークレットリソースを作成します。

ステップ 2: Greengrass グループにローカルシークレットリソースを追加する

この手順では、Secrets Manager シークレットを参照するグループリソースを設定します。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. シークレットリソースを追加するグループを選択します。
3. グループ設定ページの [Resources] (リソース) タブから、[Secrets] (シークレット) セクションを選択します。[Secrets] (シークレット) セクションには、グループに属するシークレットリソースが表示されます。このセクションからシークレットリソースを追加、編集、削除できます。

> **Note**
>
> 別の方法として、コネクタや Lambda 関数を設定するときに、コンソールでシークレットとシークレットリソースを作成することもできます。これはコネクタの [Configure

parameters] (パラメータの設定) ページか、Lambda 関数の [Resources] (リソース) ページから実行できます。

4. [Secrets] (シークレット) セクションから [Add] (追加) を選択します。
5. [Add a secret resource] (シークレットリソースに追加) ページで、[Resource name] (リソース名) に **`MyTestSecret`** を入力します。
6. [Secret] (シークレット) から[greengrass-testsecret]を選択します。
7. [Select labels (Optional)] (ラベルを選択 〔オプション〕) セクションの、AWSCURRENT ステージングラベルはシークレットの最新バージョンを表します。このラベルはシークレットリソースに常に含まれています。

> **Note**
>
> このチュートリアルでは、AWSCURRENT ラベルのみが必要です。オプションで、Lambda 関数またはコネクタで必要になるラベルを含めることができます。

8. [Add resource] (リソースを追加) を選択します。

ステップ 3: Lambda 関数デプロイパッケージを作成する

Lambda 関数を作成するには、関数のコードと依存関係を含む Lambda 関数デプロイパッケージを最初に作成する必要があります。Greengrass Lambda 関数には、コア環境での MQTT メッセージとの通信やローカルシークレットへのアクセスなどのタスクに使用する、AWS IoT Greengrass Core SDK が必要です。このチュートリアルでは Python 関数を作成するため、デプロイパッケージには Python 版の SDK を使用します。

> **Note**
>
> ローカルシークレットの値を取得するには、ユーザー定義の Lambda 関数で AWS IoT Greengrass Core SDK v1.3.0 以降を使用する必要があります。

1. AWS IoT Greengrass Core SDK ダウンロードページから、 AWS IoT Greengrass Core SDK for Python をコンピュータにダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。

3. 以下の Python コード関数を `secret_test.py` というローカルファイルに保存します。

```
import greengrasssdk

secrets_client = greengrasssdk.client("secretsmanager")
iot_client = greengrasssdk.client("iot-data")
secret_name = "greengrass-TestSecret"
send_topic = "secrets/output"


def function_handler(event, context):
    """
    Gets a secret and publishes a message to indicate whether the secret was
    successfully retrieved.
    """
    response = secrets_client.get_secret_value(SecretId=secret_name)
    secret_value = response.get("SecretString")
    message = (
        f"Failed to retrieve secret {secret_name}."
        if secret_value is None
        else f"Successfully retrieved secret {secret_name}."
    )
    iot_client.publish(topic=send_topic, payload=message)
    print("Published: " + message)
```

`get_secret_value` 関数は、`SecretId` 値として Secrets Manager シークレットの名前または ARN をサポートしています。この例では、シークレット名を使用しています。このシークレットの例では、 はキーと値のペア AWS IoT Greengrass を返します`{"test":"abcdefghi"}`。

> **Important**
>
> ユーザー定義の Lambda 関数がシークレットを安全に処理することと、シークレットに格納されている機密データをログに記録しないことを確認してください。詳細については、「AWS Secrets Manager ユーザーガイド」の「Lambda 関数のログ記録とデバッグのリスクを軽減する」を参照してください。このドキュメントでは特に回転関数について言及していますが、こうした推奨事項は Greengrass Lambda 関数にも適用されます。

4. 以下の項目を `secret_test_python.zip` という名前のファイルに圧縮します。ZIP ファイルを作成するときに、コードと依存関係のみを含め、フォルダは含めません。

   - secret_test.py アプリケーションロジック。
   - greengrasssdk。すべての Python Greengrass Lambda 関数に必要なライブラリ。

   これが Lambda 関数デプロイパッケージです。

ステップ 4: Lambda 関数を作成する

このステップでは、 AWS Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージを使用するように設定します。次に、関数のバージョンを公開し、エイリアスを作成します。

1. 最初に Lambda 関数を作成します。

   a. で AWS Management Console、サービスを選択し、 AWS Lambda コンソールを開きます。

   b. [Create function] (関数の作成) を選択し、[Author from scratch] (一から作成) を選択します。

   c. [基本的な情報] セクションで、以下の値を使用します。

      - [関数名] に「**`SecretTest`**」と入力します。
      - [ランタイム] で [Python 3.7] を選択します。
      - [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

   d. ページの下部で、[関数の作成] を選択します。

2. 今度は、ハンドラを登録し、Lambda 関数デプロイパッケージをアップロードします。

   a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

b. [Upload] (アップロード) を選択し、`secret_test_python.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

- [ランタイム] で [Python 3.7] を選択します。
- [ハンドラ] に **`secret_test.function_handler`** と入力します。

d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コンソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

3. ここで、Lambda 関数の最初のバージョンを公開し、バージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

a. [アクション] メニューから、[新しいバージョンを発行] を選択します。

b. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。

c. [SecretTest: 1] 設定ページで、[Actions (アクション)] メニューの [エイリアスの作成] を選択します。

d. [新しいエイリアスの作成] ページで、次の値を使用します。

- [名前] に **`GG_SecretTest`** と入力します。
- [バージョン] で、[1] を選択します。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

e. [Create] (作成) を選択します。

これで、Greengrass グループに Lambda 関数を追加し、シークレットリソースをアタッチする準備ができました。

ステップ 5: Lambda 関数を Greengrass グループに追加する

このステップでは、Lambda 関数を AWS IoT コンソールの Greengrass グループに追加します。

1. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
2. [My Lambda functions] (自分の Lambda 関数) セクションで、[Add] (追加) を選択します。
3. [Lambda function] (Lambda 関数) で、[SecretTest]を選択します。
4. [Lambda function version] (Lambda 関数のバージョン) で、公開したバージョンのエイリアスを選択します。

次に、Lambda 関数のライフサイクルを設定します。

1. [Lambda function configuration] (Lambda 関数の設定) セクションで次のように更新します。

> **Note**
>
> ビジネスケースで要求される場合を除き、Lambda 関数はコンテナ化を使用せずに実行することをお勧めします。これにより、デバイスリソースを設定しなくても、デバイスの GPU とカメラにアクセスできるようになります。コンテナ化なしで を実行する場

合は、 AWS IoT Greengrass Lambda 関数へのルートアクセスも許可する必要があります。

a. コンテナ化を使用せずに実行するには:

- [System user and group] (システムユーザーとグループ) で、**`Another user ID/group ID`**を選択します。[System user ID] (システムユーザ ID) には、「**`0`**」と入力します。[System group ID] (システムグループ ID) には、「**`0`**」と入力します。

  これにより、Lambda 関数を root として実行できます。root として実行の詳細については、「the section called “グループ内の Lambda 関数に対するデフォルトのアクセス ID の設定”」を参照してください。

  > **Tip**
  > また、ルートアクセスを Lambda 関数に付与するように `config.json` ファイルを更新する必要があります。手順については、「the section called “root としての Lambda 関数の実行”」を参照してください。

- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[No container] (コンテナなし) を選択します。

  コンテナ化を使用しない実行の詳細については、「the section called “Lambda 関数のコンテナ化を選択する場合の考慮事項”」を参照してください。

- [タイムアウト] に **`10 seconds`** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。

- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

b. 代わりにコンテナ化モードで実行するには:

> **Note**
> ビジネスケースで要求されない限り、コンテナ化モードでの実行はお勧めしていません。

- [System user and group] (システムユーザーとグループ) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [Lambda function containerization] (Lambda 関数のコンテナ化) で、[Use group default] (グループのデフォルトを使用) を選択します。
- [メモリ制限] に **1024 MB** と入力します。
- [タイムアウト] に **10 seconds** と入力します。
- [固定] で、[True] を選択します。

  詳細については、「the section called “ライフサイクル設定”」を参照してください。
- [Additional Parameter] (追加のパラメータ) の、[Read access to /sys directory] (/sys ディレクトリへの読み取りアクセス) で、[Enabled] (有効) を選択します。

2. [Add Lambda function] (Lambda 関数の追加) を選択します。

次に、シークレットリソースを関数に関連付けます。

ステップ 6: シークレットリソースを Lambda 関数にアタッチする

このステップでは、シークレットリソースを Greengrass グループの Lambda 関数に関連付けます。これで、リソースは関数に関連付けられて、ローカルシークレットの値を取得できるようになります。

1. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
2. [SecretTest] 関数を選択します。
3. 関数の詳細ページで、[Resources] (リソース) を選択します。
4. シークレット セクションまでスクロールして、[Associate] (関連付け) を選択します。
5. [MyTestSecret]、[Associate] (関連付け) の順に選択します。

ステップ 7: サブスクリプションを Greengrass グループに追加する

このステップでは、 AWS IoT と Lambda 関数がメッセージを交換できるようにするサブスクリプションを追加します。1 つのサブスクリプションでは AWS IoT が関数を呼び出し、もう 1 つのサブスクリプションでは関数が出力データを送信できるようにします AWS IoT。

1. グループ設定ページの [Subscriptions] (サブスクリプション) タブで、[Add Subscription] (サブスクリプションの追加) を選択します。
2. が 関数 AWS IoT にメッセージを発行できるようにするサブスクリプションを作成します。

   グループ設定ページで、[Subscriptions] (サブスクリプション) タブ、[Add subscription] (サブスクリプションの追加) の順に選択します。
3. [Create a subscription] (サブスクリプションの作成) ページで、ソースおよびターゲットを次のように設定します。

   a. [Source type] (ソースタイプ) で、[Lambda function] (Lambda 関数)、[IoT Cloud] (IoT クラウド) の順に選択します。

   b. [Target type] (ターゲットタイプ) で、[Service] (サービス)、[SecretTest]の順に選択します。

   c. [Topic filter] (トピックのフィルター) で、「**`secrets/input`**」と入力し、[Create subscription] (サブスクリプションの作成) を選択します。
4. 2 つ目のサブスクリプションを追加します。[Subscriptions] (サブスクリプション) タブ、[Add subscription] (サブスクリプションの追加) の順に選択し、ソースとターゲットを次のように設定します。

   a. [Source type] (ソースタイプ) で、[Services] (サービス)、[SecretTest]の順に選択します。

   b. [Target type] (ターゲットタイプ) で、[Lambda function] (Lambda 関数)、 [IoT Cloud] (IoT クラウド) の順に選択します。

   c. [Topic filter] (トピックのフィルター) で、「**`secrets/output`**」と入力し、[Create subscription] (サブスクリプションの作成) を選択します。

ステップ 8: Greengrass グループをデプロイする

Core デバイスにグループをデプロイします。デプロイ中、 は Secrets Manager からシークレットの値 AWS IoT Greengrass を取得し、ローカルで暗号化されたコピーをコアに作成します。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

出力に `root` で実行中の `/greengrass/ggc/packages/ggc-version/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

> **Note**
>
> パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

b. デーモンを開始するには、以下を実行します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。

3. a. [Lambda functions] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) セクションで、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。

   b. [IP ディテクターの設定を編集] ダイアログボックスで、[MQTT ブローカーエンドポイントを自動的に検出して上書きする] を選択します。

   c. [Save] を選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   >
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイのステータスが [完了] と表示されます。

トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

Lambda 関数をテストする

1. AWS IoT コンソールのホームページで、テストを選択します。
2. [Subscribe to topic] (トピックへのサブスクライブ) で、以下の値を使用し、[Subscribe] (サブスクリプション) を選択します。

プロパティ	値
トピックのサブスクリプション	シークレット/出力
MQTT ペイロード表示	文字列としてペイロードを表示

3. [Publish to topic] (トピックに発行) で、以下の値を使用し、[Publish] (発行) を選択して関数を呼び出します。

プロパティ	値
トピック	シークレット/入力
メッセージ	デフォルトのメッセージを保持します。メッセージの発行では Lambda 関数を呼び出しますが、このチュートリアルの関数はメッセージ本文を処理しません。

成功した場合、関数は成功のメッセージを発行します。

関連情報

- Core にシークレットをデプロイする

Greengrass コネクタを使用したサービスおよびプロトコルとの統合

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

のコネクタ AWS IoT Greengrass は、ローカルインフラストラクチャ、デバイスプロトコル AWS、その他のクラウドサービスとやり取りする効率を高める構築済みのモジュールです。コネクタを使用すれば、新しいプロトコルや API を学習する時間を短縮し、ビジネスにとって重要なロジックに注力できます。

次の図は、 AWS IoT Greengrass 環境でコネクタが役立つ場所を示しています。

多くのコネクタは、MQTT メッセージを使用して、グループ内のクライアントデバイスと Greengrass Lambda 関数、または AWS IoT およびローカルシャドウサービスと通信します。次の例では、Twilio Notifications コネクタはユーザー定義の Lambda 関数から MQTT メッセージを受信し、 からシークレットのローカルリファレンスを使用し AWS Secrets Manager、Twilio API を呼び出します。

このソリューションを作成するチュートリアルについては、「the section called “コネクタの使用を開始する (コンソール)”」および「the section called “コネクタの使用を開始する (CLI)”」を参照してください。

Greengrass コネクタは、デバイスの機能を拡張したり、専用デバイスを作成したりするのに役立ちます。コネクタを使用すると次のことができます。

- 再利用可能なビジネスロジックを実装する。
- やサードパーティーのサービスなど、クラウド AWS およびローカルサービスとやり取りします。
- デバイスのデータを収集して処理する。
- MQTT トピックのサブスクリプションとユーザー定義の Lambda 関数を使用して、デバイス間呼び出しを可能にする。

AWS は、一般的な サービスやデータソースとのやり取りを簡素化する Greengrass コネクタのセットを提供します。これらの事前構築済みモジュールにより、ログ記録と診断、補充、産業データ処理、アラームとメッセージングのシナリオに対応できます。詳細については、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

要件

コネクタを使用する際は次の点に注意してください。

- 使用するコネクタにはそれぞれに要件があり、それを満たす必要があります。これらの要件には、最小 AWS IoT Greengrass Core ソフトウェアバージョン、デバイスの前提条件、必要なアクセス

許可、制限が含まれる場合があります。詳細については、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

- Greengrass グループに含めることができる、コネクタの設定済みインスタンスは 1 つのみですが、このインスタンスは複数のサブスクリプションで使用することができます。詳細については、「the section called “設定パラメータ”」を参照してください。
- Greengrass グループのデフォルトのコンテナ化が [コンテナなし] に設定されている場合、グループのコネクタはコンテナ化なしで実行する必要があります。コンテナなしモードをサポートするコネクタを検索するには、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

Greengrass コネクタの使用

コネクタはグループコンポーネントの一種です。クライアントデバイスやユーザー定義の Lambda 関数など、他のグループコンポーネントと同様に、グループにコネクタを追加し、設定を設定し、AWS IoT Greengrass コアにデプロイします。コネクタは、コア環境で実行されます。

一部のコネクタは、シンプルなスタンドアロンアプリケーションとしてデプロイすることができます。例えば、Device Defender コネクタは Core デバイスからシステムメトリクスを読み取り、解析のためにそれらを AWS IoT Device Defender に送信します。

それ以外のコネクタは、さらに大きいソリューションの構成要素として追加することができます。以下のサンプルソリューションでは、Modbus-RTU プロトコルアダプタコネクタはセンサーからのメッセージを処理し、Twilio 通知コネクタは Twilio メッセージを開始します。

ソリューションには、多くの場合、コネクタの横に配置され、コネクタが送受信するデータを処理するユーザー定義の Lambda 関数が含まれています。この例でその役割にあたる TempMonitor 関数は、Modbus-RTU プロトコルアダプタからデータを受信し、ビジネスロジックを実行して、データを Twilio 通知に送信します。

ソリューションを作成してデプロイするには、以下の一般的なプロセスに従います。

1. データフローの概要を設計します。作業に必要なデータソース、データチャネル、サービス、プロトコル、リソースを特定します。このサンプルソリューションでは、Modbus RTU プロトコル、物理 Modbus シリアルポート、Twilio でのデータが含まれています。
2. ソリューションに含めるコネクタを特定し、グループに追加します。このサンプルソリューションでは、Modbus-RTU プロトコルアダプタと Twilio 通知を使用します。シナリオに該当するコネクタを見つけ、個々の要件を知るには、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。
3. ユーザー定義の Lambda 関数、クライアントデバイス、またはリソースが必要かどうかを特定し、それらを作成してグループに追加します。例えば、ビジネスロジックを含む関数や、ソリューション内の別のエンティティに必要な形式にデータを加工する関数を作成して追加します。このサンプルソリューションでは、Modbus RTU リクエストを送信して Twilio 通知を開始する関数を使用しています。また、Modbus RTU シリアルポートのローカルデバイスリソースと、Twilio 認証トークンのシークレットリソースも含まれています。

   > **Note**
   >
   > シークレットリソースは、 AWS Secrets Managerからのパスワード、トークン、および他のシークレットを参照します。シークレットは、コネクタおよび Lambda 関数でサービスやアプリケーションの認証に使用できます。デフォルトでは、 AWS IoT Greengrass は「greengrass-」で始まる名前のシークレットにアクセスできます。詳細については、「Core にシークレットをデプロイする」を参照してください。

4. ソリューション内のエンティティに MQTT メッセージの交換を許可するサブスクリプションを作成します。サブスクリプションでコネクタを使用する場合、コネクタ、およびメッセージのソースまたはターゲットでは、コネクタによってサポートされている事前定義済みのトピック構文を使用する必要があります。詳細については、「the section called “入力と出力”」を参照してください。
5. Greengrass Core にグループをデプロイします。

コネクタを作成してデプロイする方法については、次のチュートリアルを参照してください。

- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

設定パラメータ

多くのコネクタには、動作や出力をカスタマイズするためのパラメータが用意されています。これらのパラメータは、コネクタのライフサイクルでの初期化時、実行時などのタイミングで使用します。

パラメータのタイプと使用方法はコネクタごとに異なります。例えば、SNS コネクタにはデフォルトの SNS トピックを設定するパラメータがあり、Device Defender にはデータサンプリングレートを設定するパラメータがあります。

グループバージョンには、複数のコネクタを含めることができますが、一度に使用されるコネクタのインスタンスは 1 つのみです。これは、グループの各コネクタのアクティブな設定は 1 つだけであることを意味します。ただし、コネクタインスタンスはグループ内の複数のサブスクリプションで使用できます。例えば、複数のデバイスに Kinesis Firehose コネクタへのデータの送信を許可するサブスクリプションを作成できます。

グループリソースへのアクセスに使用するパラメータ

Greengrass コネクタでは、グループリソースを使用して、Core デバイス上のファイルシステム、ポート、周辺機器、その他のローカルリソースにアクセスします。コネクタには、グループリソースへのアクセスを必要とする場合に使用する設定パラメータがあります。

グループリソースには以下のものが含まれます。

- ローカルリソース。Greengrass Core デバイス上に存在するディレクトリ、ファイル、ポート、ピン、周辺機器。
- 機械学習リソース。クラウドでトレーニングされ、ローカル推論のために Core にデプロイされる機械学習モデル。
- シークレットリソース。パスワード、キー、トークン、または任意のテキストの暗号化されたローカルコピー AWS Secrets Manager。コネクタは、これらのローカルシークレットに安全にアクセスし、それらを使用してサービスまたはローカルインフラストラクチャを認証できます。

例えば、Device Defender のパラメータにより、ホストの `/proc` ディレクトリのシステムメトリクスへのアクセスが可能になり、Twilio 通知のパラメータにより、ローカルに保存された Twilio 認証トークンへのアクセスが可能になります。

コネクタのパラメータの更新

コネクタを Greengrass グループに追加すると、パラメータが設定されます。コネクタを追加したら、パラメータ値を変更することができます。

- コンソールの場合: グループ設定ページで [コネクタ] を開き、コネクタのコンテキストメニューから [編集] を選択します。

> **Note**
>
> コネクタで使用されているシークレットリソースを、別のシークレットを参照するように後で変更する場合は、コネクタのパラメータを編集し、変更を確認する必要があります。

- API の場合: 新しい設定を定義する別のバージョンのコネクタを作成します。

  AWS IoT Greengrass API はバージョンを使用してグループを管理します。バージョンは変更不可であるため、グループのクライアントデバイス、関数、リソースといったグループコンポーネントを追加または変更するには、新規または更新済みコンポーネントのバージョンを作成する必要があります。その後、各コンポーネントのターゲットバージョンを含むグループバージョンを作成およびデプロイします。

コネクタの設定に変更を加えたら、これらの変更をコアに反映するためにグループをデプロイする必要があります。

入力と出力

多くの Greengrass コネクタは、MQTT メッセージを送受信することで、他のエンティティと通信できます。MQTT 通信は、コネクタが Greengrass グループ内の Lambda 関数、クライアントデバイス、およびその他のコネクタ、または AWS IoT およびローカルシャドウサービスとデータを交換できるようにするサブスクリプションによって制御されます。この通信を許可するには、コネクタが属するグループにサブスクリプションを作成する必要があります。詳細については、「the section called “MQTT メッセージングワークフローにおけるマネージドサブスクリプション”」を参照してください。

コネクタは、メッセージ受信者、メッセージ発行者、またはその両方になります。各コネクタは、発行またはサブスクライブする MQTT トピックを定義します。メッセージソースまたはメッセージターゲットがコネクタとなるサブスクリプションでは、これらの事前定義済みのトピックを使用する必要があります。コネクタのサブスクリプションを設定する手順が含まれているチュートリアルにつ

いては、「the section called “コネクタの使用を開始する (コンソール)”」と「the section called “コネクタの使用を開始する (CLI)”」を参照してください。

> **Note**
>
> 多くのコネクタには、クラウドサービスまたはローカルサービスとやり取りするための組み込みの通信モードもあります。これらのモードはコネクタごとに異なり、パラメータの設定やグループロールへのアクセス許可の追加が必要になる場合があります。コネクタの要件については、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

入力トピック

コネクタには通常、MQTT トピックに関する入力データが送信されます。一部のコネクタは、入力データ用に複数のトピックにサブスクライブします。例えば、シリアルストリーミングコネクタは以下の 2 つのトピックをサポートしています。

- `serial/+/read/#`
- `serial/+/write/#`

このコネクタでは、読み取りリクエストと書き込みリクエストを対応するトピックに送信します。サブスクリプションを作成するときは、実装に合ったトピックを使用してください。

前の例の + 文字と # 文字はワイルドカードです。これらのワイルドカードを使用すると、受信者は複数のトピックに関するメッセージを受信できます。発行者は発行先のトピックをカスタマイズできます。

- + ワイルドカードは、トピック階層の任意の場所に使用できます。このワイルドカードは、1 つの階層項目に置き換わります。

  例えば、`sensor/+/input` トピックの場合、メッセージを `sensor/id-123/input` トピックには発行できますが、`sensor/group-a/id-123/input` トピックには発行できません。

- # ワイルドカードは、トピック階層の末尾にのみ使用できます。0 以上の階層項目で置き換えることができます。

例えば、`sensor/#` トピックの場合、メッセージを `sensor/`、`sensor/id-123`、`sensor/group-a/id-123` には発行できますが、`sensor` には発行できません。

ワイルドカード文字は、トピックにサブスクライブするときにのみ有効です。ワイルドカードを含むトピックにメッセージを発行することはできません。入力または出力トピックの要件の詳細については、コネクタのドキュメントを参照してください。詳細については、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。

コンテナ化のサポート

デフォルトでは、ほとんどのコネクタは、 AWS IoT Greengrassによって管理されている分離されたランタイム環境の Greengrass コア上で実行されます。コンテナと呼ばれるこれらのランタイム環境は、コネクタとホストシステム間の分離を実現します。これにより、ホストとコネクタのセキュリティが強化されます。

ただし、この Greengrass コンテナ化は、Docker コンテナや cgroup を使用しない古い Linux カーネル AWS IoT Greengrass で を実行する場合など、一部の環境ではサポートされていません。これらの環境では、コネクタはコンテナなしモードで実行する必要があります。コンテナなしモードをサポートするコネクタを検索するには、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。一部のコネクタはこのモードでネイティブに動作し、一部のコネクタでは分離モードを設定できます。

Greengrass コンテナ化をサポートする環境では、分離モードを [コンテナなし] に設定することもできますが、可能であれば [Greengrass コンテナ] モードを使用することをお勧めします。

> **Note**
>
> Greengrass グループの既定のコンテナ化設定は、コネクタには適用されません。

コネクタのバージョンのアップグレード

コネクタプロバイダーは、機能の追加、問題の修正、パフォーマンスの向上を行うコネクタの新しいバージョンをリリースすることがあります。使用可能なバージョンおよび関連する変更については、各コネクタのドキュメントを参照してください。

AWS IoT コンソールでは、Greengrass グループのコネクタの新しいバージョンを確認できます。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [Greengrass グループ] で、目的のグループを選択します。
3. [コネクタ] を選択して、グループ内のコネクタを表示します。

   新しいバージョンのコネクタがある場合は、[Available] (利用可能) ボタンが [Upgrade] (アップグレード) 列に表示されます。
4. コネクタのバージョンをアップグレードするには、以下を実行します。

   a. [コネクタ] ページの [アップグレード] 列で、[使用可能] を選択します。[コネクタのアップグレード] ページが開き、現在のパラメータ設定が表示されます (該当する場合)。

      新しいコネクタのバージョンを選択し、必要に応じてパラメータを定義して、[アップグレード] を選択します。

   b. [サブスクリプション] ページで、新しいサブスクリプションをグループに追加して、コネクタをソースまたはターゲットとして使用しているサブスクリプションを置き換えます。次に、古いサブスクリプションを削除します。

      サブスクリプションはバージョンごとにコネクタを参照するため、グループのコネクタバージョンを変更すると無効になります。

   c. [アクション] メニューから [デプロイ] を選択し、変更をコアにデプロイします。

AWS IoT Greengrass API からコネクタをアップグレードするには、更新されたコネクタとサブスクリプションを含むグループバージョンを作成してデプロイします。コネクタをグループに追加する場合と同じプロセスを使用します。を使用して Twilio Notifications コネクタの例 AWS CLI を設定およびデプロイする方法を示す詳細な手順については、「」を参照してくださいthe section called “コネクタの使用を開始する (CLI)”。

コネクタのログ記録

Greengrass コネクタには、Greengrass ログにイベントとエラーを書き込む Lambda 関数が含まれています。グループ設定に応じて、ログは CloudWatch Logs、ローカルファイルシステム、またはその両方に書き込まれます。コネクタからのログには、対応する関数の ARN が含まれています。以下のサンプル ARN は、Kinesis Firehose コネクタのものです。

```
arn:aws:lambda:aws-region:account-id:function:KinesisFirehoseClient:1
```

ログ記録のデフォルト設定では、情報レベルのログが以下のディレクトリ構造でファイルシステムに書き込まれます。

```
greengrass-root/ggc/var/log/user/region/aws/function-name.log
```

Greengrass のログ記録の詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

AWSが提供する Greengrass コネクタ

AWS には、一般的な AWS IoT Greengrass シナリオをサポートする以下のコネクタが用意されています。コネクタの動作の詳細については、以下のドキュメントを参照してください。

- コネクタを使用してサービスおよびプロトコルと統合する
- 「コネクタの使用を開始する (コンソール)」または「コネクタの使用を開始する (CLI)」

コネクタ	説明	サポートされる Lambda ランタイム	コンテナなしモードのサポート
CloudWatch メトリクス	カスタムメトリクスを Amazon CloudWatch に発行します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい
Device Defender	システムメトリクスを に送信します AWS IoT Device Defender。	• Python 3.8 * • Python 3.7 • Python 2.7	いいえ
Docker アプリケーションのデプロイ	Docker Compose ファイルを実行して、コアデバイス上で Docker アプリケーションを起動します。	• Python 3.8 • Python 3.7	はい
IoT Analytics	デバイスとセンサーから にデータを送信します AWS IoT Analytics。	• Python 3.8 *	はい

コネクタ	説明	サポートされる Lambda ランタイム	コンテナなしモードのサポート
		• Python 3.7 • Python 2.7	
IoT イーサネット IP プロトコルアダプタ	イーサネットデバイスおよび IP デバイスからデータを収集します。	• Java 8	はい
IoT SiteWise	デバイスやセンサーから AWS IoT SiteWiseのアセットプロパティにデータを送信 します。	• Java 8	はい
Kinesis Firehose	Amazon Data Firehose 配信ストリーミングにデータを送信します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい
ML フィードバック	機械学習モデルの入力をクラウドに発行し、出力を MQTT トピックに発行します。	• Python 3.8 • Python 3.7	いいえ
ML イメージ分類	ローカルイメージ分類推論サービスを実行します。このコネクタには、複数のプラットフォーム用のバージョンが用意されています。	• Python 3.8 * • Python 3.7 • Python 2.7	なし
ML オブジェクト検出	ローカルオブジェクトの検出推論サービスを実行します。このコネクタには、複数のプラットフォーム用のバージョンが用意されています。	• Python 3.8 • Python 3.7	いいえ
Modbus-RTU プロトコルアダプタ	Modbus RTU デバイスにリクエストを送信します。	• Python 3.8 * • Python 3.7 • Python 2.7	いいえ

コネクタ	説明	サポートされる Lambda ランタイム	コンテナなしモードのサポート
Modbus-TCP プロトコルアダプタ	ModbusTCP デバイスからデータを収集します。	• Java 8	あり
Raspberry Pi GPIO	Raspberry Pi Core デバイスの GPIO ピンをコントロールします。	• Python 3.8 * • Python 3.7 • Python 2.7	いいえ
シリアルストリーミング	Core デバイスのシリアルポートに対する読み込みと書き込みを行います。	• Python 3.8 * • Python 3.7 • Python 2.7	いいえ
ServiceNow MetricBase 統合	ServiceNow MetricBase に時系列メトリクスを発行します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい
SNS	Amazon SNS トピックにメッセージを送信します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい
Splunk 統合	データを Splunk HEC に発行します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい

コネクタ	説明	サポートされる Lambda ランタイム	コンテナなしモードのサポート
Twilio 通知	Twilio のテキストまたは音声メッセージを開始します。	• Python 3.8 * • Python 3.7 • Python 2.7	はい

* Python 3.8 ランタイムを使用するには、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成する必要があります。詳細については、コネクタ固有の要件を参照してください。

> **Note**
>
> コネクタのバージョンをアップグレードし、Python 2.7 から Python 3.7 にすることをお勧めします。Python 2.7 コネクタの継続的なサポートは AWS Lambda 、ランタイムのサポートによって異なります。詳細については、「AWS Lambda デベロッパーガイド」の「ランタイムの非推奨化に関するポリシー」を参照してください。

CloudWatch Metrics コネクタ

CloudWatch Metrics コネクタは、Greengrass デバイスのカスタムメトリクスを Amazon CloudWatch に発行します。このコネクタでは、CloudWatch メトリクスの発行に一元管理型のインフラストラクチャを使用して、Greengrass Core 環境をモニタリングおよび分析し、ローカルのイベントに対処できます。詳細については、「Amazon CloudWatch ユーザーガイド」の「Amazon CloudWatch メトリクスを使用する」を参照してください。

このコネクタには MQTT メッセージとしてメトリクスデータが送信されます。また、コネクタは、同じ名前空間にあるメトリクスをバッチにまとめて、定期的に CloudWatch に発行します。

このコネクタには、次のバージョンがあります。

バージョン	ARN
5	`arn:aws:greengrass:  region::/connectors/CloudWatchMetrics/versions/5`
4	`arn:aws:greengrass:  region::/connectors/CloudWatchMetrics/versions/4`
3	`arn:aws:greengrass:  region::/connectors/CloudWatchMetrics/versions/3`
2	`arn:aws:greengrass:  region::/connectors/CloudWatchMetrics/versions/2`
1	`arn:aws:greengrass:  region::/connectors/CloudWatchMetrics/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3 - 5

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- 以下の例に示すように、`cloudwatch:PutMetricData` アクションを許可する Greengrass グループロールに AWS Identity and Access Management (IAM) ポリシーが追加されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1528133056761",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

Amazon CloudWatch のアクセス許可の詳細については「IAM ユーザーガイド」の「Amazon CloudWatch の許可リファレンス」を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- 以下の例に示すように、`cloudwatch:PutMetricData` アクションを許可する Greengrass グループロールに AWS Identity and Access Management (IAM) ポリシーが追加されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1528133056761",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

Amazon CloudWatch のアクセス許可の詳細については「IAM ユーザーガイド」の「Amazon CloudWatch の許可リファレンス」を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Versions 4 - 5

`PublishInterval`

特定の名前空間のメトリクスのバッチを発行するまでに待機する最大秒数。最大値は 900 です。メトリクスを受信時にバッチにまとめないで発行するようにコネクタを設定するには、0 を指定します。

同じ名前空間内で、または指定した間隔後に 20 個のメトリクスを受信したら、コネクタから CloudWatch に発行されます。

> **Note**
> コネクタでは、発行イベントの順序は保証されません。

AWS IoT コンソールの表示名: 発行間隔

必須: `true`

型: `string`

有効な値: `0 - 900`

有効なパターン: `[0-9]|[1-9]\d|[1-9]\d\d|900`

`PublishRegion`

CloudWatch メトリクス AWS リージョン を投稿する 。この値は、デフォルトの Greengrass メトリクスリージョンを上書きします。クロスリージョンメトリクスをポストする場合にのみ必要です。

AWS IoT コンソールの表示名: リージョンの公開

必須: `false`

型: `string`

有効なパターン: `^$|([a-z]{2}-[a-z]+-\d{1})`

`MemorySize`

コネクタに割り当てるメモリ (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`MaxMetricsToRetain`

新しいメトリクスで置き換えられるまでにメモリに保存する、名前空間全体のメトリクスの最大数。最小値は 2000 です。

この制限は、インターネットへの接続がなく、コネクタが後で発行するメトリクスをバッファし始めるときに適用されます。バッファが満杯になると、最も古いメトリクスが新しいメトリクスに置き換えられます。特定の名前空間内のメトリクスは、同じ名前空間内のメトリクスでのみ置き換えられます。

> **Note**
>
> コネクタのホストプロセスが中断された場合、メトリクスは保存されません。この中断は、例えばグループデプロイ中やデバイスの再起動時などに発生する可能性があります。

AWS IoT コンソールの表示名: 保持する最大メトリクス

必須: `true`

型: `string`

有効なパターン: `^([2-9]\d{3}|[1-9]\d{4,})$`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは `GreengrassContainer` です。つまり、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
>
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Versions 1 - 3

`PublishInterval`

特定の名前空間のメトリクスのバッチを発行するまでに待機する最大秒数。最大値は 900 です。メトリクスを受信時にバッチにまとめないで発行するようにコネクタを設定するには、0 を指定します。

同じ名前空間内で、または指定した間隔後に 20 個のメトリクスを受信したら、コネクタから CloudWatch に発行されます。

> **Note**
> コネクタでは、発行イベントの順序は保証されません。

AWS IoT コンソールの表示名: 発行間隔

必須: `true`

型: `string`

有効な値: `0 - 900`

有効なパターン: `[0-9]|[1-9]\d|[1-9]\d\d|900`

`PublishRegion`

CloudWatch メトリクス AWS リージョン を投稿する 。この値は、デフォルトの Greengrass メトリクスリージョンを上書きします。クロスリージョンメトリクスをポストする場合にのみ必要です。

AWS IoT コンソールの表示名: リージョンの公開

必須: `false`

型: `string`

有効なパターン: `^$|([a-z]{2}-[a-z]+-\d{1})`

`MemorySize`

コネクタに割り当てるメモリ (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`MaxMetricsToRetain`

新しいメトリクスで置き換えられるまでにメモリに保存する、名前空間全体のメトリクスの最大数。最小値は 2000 です。

この制限は、インターネットへの接続がなく、コネクタが後で発行するメトリクスをバッファし始めるときに適用されます。バッファが満杯になると、最も古いメトリクスが新しいメトリクスに置き換えられます。特定の名前空間内のメトリクスは、同じ名前空間内のメトリクスでのみ置き換えられます。

> **Note**
>
> コネクタのホストプロセスが中断された場合、メトリクスは保存されません。この中断は、例えばグループデプロイ中やデバイスの再起動時などに発生する可能性があります。

AWS IoT コンソールの表示名: 保持する最大メトリクス

必須: `true`

型: `string`

有効なパターン: `^([2-9]\d{3}|[1-9]\d{4,})$`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、CloudWatch Metrics コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyCloudWatchMetricsConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/CloudWatchMetrics/
versions/4",
            "Parameters": {
                "PublishInterval" : "600",
                "PublishRegion" : "us-west-2",
                "MemorySize" : "16",
                "MaxMetricsToRetain" : "2500",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、MQTT トピックのメトリクスを受け取り、それを CloudWatch に発行します。入力メッセージは JSON 形式である必要があります。

サブスクリプションのトピックフィルター

`cloudwatch/metric/put`

メッセージのプロパティ

`request`

このメッセージのメトリクスに関する情報。

リクエストオブジェクトには、CloudWatch に発行するメトリクスデータが含まれています。メトリクス値は、`PutMetricData` API の仕様を満たしている必要があります

す。namespace、metricData.metricName、および metricData.value プロパティのみが必須です。

必須: true

型: 以下のプロパティを含む object。

namespace

このリクエスト内のメトリクスデータのユーザー定義による名前空間。CloudWatch はメトリクスデータポイントのコンテナとして名前空間を使用します。

> **Note**
>
> 予約された文字列 AWS/ で始まる名前空間を指定することはできません。

必須: true

型: string

有効なパターン: [^:].*

metricData

メトリクスのデータ。

必須: true

型: 以下のプロパティを含む object。

metricName

メトリクスの名前。

必須: true

型: string

dimensions

メトリクスに関連付けられたディメンション。ディメンションは、メトリクスとそのデータに関する追加情報を提供します。1 つのメトリクスでは、最大 10 個のディメンションを定義できます。

このコネクタには coreName という名前のディメンションが必ず含まれ、そのコアの名前を値に持ちます。

必須: false

タイプ: 以下のプロパティを含むディメンションオブジェクトの array。

name

ディメンション名。

必須: false

型: string

value

ディメンション値

必須: false

型: string

timestamp

メトリクスデータが受信された時刻。Jan 1, 1970 00:00:00 UTC からの秒数で表されます。この値を省略すると、コネクタはメッセージを受信した時刻を使用します。

必須: false

型: timestamp

> **Note**
>
> このコネクタのバージョン 1 から 4 を使用する場合は、1 つのソースから複数のメトリックを送信するときに、メトリックごとに個別にタイムスタンプを取得することをお勧めします。変数を使用してタイムスタンプを保存しないでください。

value

メトリクスの値。

> **Note**
>
> CloudWatch は小さすぎる値または大きすぎる値を拒否します。値は `8.515920e-109` から `1.174271e+108` (基数 10)、または `2e-360` から `2e360` (基数 2) の範囲内であることが必要です。特殊な値 (`NaN`、`+Infinity`、`-Infinity` など) はサポートされていません。

必須: `true`

型: `double`

`unit`

メトリクスの単位。

必須: `false`

型: `string`

有効な値: `Seconds`, `Microseconds`, `Milliseconds`, `Bytes`, `Kilobytes`, `Megabytes`, `Gigabytes`, `Terabytes`, `Bits`, `Kilobits`, `Megabits`, `Gigabits`, `Terabits`, `Percent`, `Count`, `Bytes/Second`, `Kilobytes/Second`, `Megabytes/Second`, `Gigabytes/Second`, `Terabytes/Second`, `Bits/Second`, `Kilobits/Second`, `Megabits/Second`, `Gigabits/Second`, `Terabits/Second`, `Count/Second`, `None`

制限

このコネクタを使用すると、CloudWatch `PutMetricData` API によって課せられるすべての制限がメトリクスに適用されます。以下の制限が特に重要です。

- API ペイロードに適用される 40 KB の制限
- API リクエストごとに 20 個のメトリクス
- `PutMetricData` API の 150 トランザクション/秒 (TPS)

詳細については、「Amazon CloudWatch ユーザーガイド」の「CloudWatch の制限」を参照してください。

入力例

```
{
```

```
   "request": {
       "namespace": "Greengrass",
       "metricData":
           {
               "metricName": "latency",
               "dimensions": [
                   {
                       "name": "hostname",
                       "value": "test_hostname"
                   }
               ],
               "timestamp": 1539027324,
               "value": 123.0,
               "unit": "Seconds"
            }
    }
}
```

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。

サブスクリプションのトピックフィルター

```
cloudwatch/metric/put/status
```

出力例: 成功

レスポンスには、メトリクスデータの名前空間と CloudWatch レスポンスの `RequestId` フィールドが含まれています。

```
{
   "response": {
        "cloudwatch_rid":"70573243-d723-11e8-b095-75ff2EXAMPLE",
        "namespace": "Greengrass",
        "status":"success"
    }
}
```

出力例: 失敗

```
{
```

```
    "response" : {
         "namespace": "Greengrass",
         "error": "InvalidInputException",
         "error_message":"cw metric is invalid",
         "status":"fail"
    }
}
```

> **Note**
>
> コネクタが再試行可能なエラー (接続エラーなど) を検出した場合は、次のバッチ処理で再発行を試します。

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。

   グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォル

ダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

   b. コネクタを追加し、そのパラメータを設定します。

   c. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。

5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'cloudwatch/metric/put'

def create_request_with_all_fields():
```

```
    return  {
        "request": {
            "namespace": "Greengrass_CW_Connector",
            "metricData": {
                "metricName": "Count1",
                "dimensions": [
                    {
                        "name": "test",
                        "value": "test"
                    }
                ],
                "value": 1,
                "unit": "Seconds",
                "timestamp": time.time()
            }
        }
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=send_topic,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

CloudWatch Metrics コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0

- urllib3/MIT ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
5	入力データでの重複するタイムスタンプに対応するように修正。
4	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- 「Amazon CloudWatch ユーザーガイド」の「Amazon CloudWatch メトリクスの使用」
- 「Amazon CloudWatch API Reference」(Amazon CloudWatch API リファレンス) の「PutMetricData」

Device Defender コネクタ

Device Defender コネクタは、Greengrass Core デバイスの状態の変化を管理者に通知します。これは、侵害されたデバイスを示す可能性のある異常な動作を特定するのに役立ちます。

このコネクタは、コアデバイスの `/proc` ディレクトリからシステムメトリクスを読み取り、メトリクスを に発行します AWS IoT Device Defender。メトリクスレポートの詳細については、「AWS IoT デベロッパーガイド」の「デバイスメトリクスドキュメントの仕様」を参照してください。

このコネクタには、次のバージョンがあります。

バージョン	ARN
3	`arn:aws:greengrass:  region::/connectors/DeviceDefender/versions/3`
2	`arn:aws:greengrass:  region::/connectors/DeviceDefender/versions/2`
1	`arn:aws:greengrass:  region::/connectors/DeviceDefender/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- AWS IoT Device Defender Detect 機能を使用して違反を追跡するように設定されています。詳細については、「AWS IoT デベロッパーガイド」の「検出」を参照してください。
- `/proc` ディレクトリを参照する Greengrass グループのローカルボリュームリソース。リソースは以下のプロパティを使用する必要があります。
  - ソースパス: `/proc`
  - ターゲットパス: `/host_proc` (または有効なパターンと一致する値)
  - AutoAddGroupOwner: `true`
- Greengrass コアにインストールされた psutil ライブラリ。バージョン 5.7.0 は、コネクタで動作することが確認された最新バージョンです。
- Greengrass コアにインストールされた cbor ライブラリ。バージョン 1.0.0 は、コネクタで動作することが確認された最新バージョンです。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- AWS IoT Device Defender Detect 機能を使用して違反を追跡するように設定されています。詳細については、「AWS IoT デベロッパーガイド」の「検出」を参照してください。
- `/proc` ディレクトリを参照する Greengrass グループのローカルボリュームリソース。リソースは以下のプロパティを使用する必要があります。
  - ソースパス: `/proc`
  - ターゲットパス: `/host_proc` (または有効なパターンと一致する値)

- AutoAddGroupOwner: `true`
- Greengrass コアにインストールされた psutil ライブラリ。
- Greengrass コアにインストールされた cbor ライブラリ。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

`SampleIntervalSeconds`

メトリクスの収集およびレポートの各サイクル間の秒数。最小値は 300 秒 (5 分) です。

AWS IoT コンソールの表示名: メトリクスレポート間隔

必須: `true`

型: `string`

有効なパターン: `^[0-9]*(?:3[0-9][0-9]|[4-9][0-9]{2}|[1-9][0-9]{3,})$`

`ProcDestinationPath-ResourceId`

`/proc` ボリュームリソースの ID。

> **Note**
> このコネクタには、リソースへの読み取り専用アクセス許可が付与されています。

AWS IoT コンソールの表示名: /proc ディレクトリのリソース

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9_-]+`

`ProcDestinationPath`

`/proc` ボリュームリソースのターゲットパス。

AWS IoT コンソールの表示名: /proc リソースの送信先パス

必須: `true`

型: `string`

有効なパターン: `\/[a-zA-Z0-9_-]+`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Device Defender コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyDeviceDefenderConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/DeviceDefender/
versions/3",
            "Parameters": {
                "SampleIntervalSeconds": "600",
                "ProcDestinationPath": "/host_proc",
                "ProcDestinationPath-ResourceId": "my-proc-resource"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

このコネクタは、セキュリティメトリクスを出力データ AWS IoT Device Defender として に発行します。

サブスクリプションのトピックフィルター

```
$aws/things/+/defender/metrics/json
```

> **Note**
>
> これは、 が AWS IoT Device Defender 想定するトピック構文です。コネクタはワイルドカード + をデバイス名 ($aws/things/*thing-name*/defender/metrics/json など) に置き換えます。

出力例

メトリクスレポートの詳細については、「AWS IoT デベロッパーガイド」の「デバイスメトリクスドキュメントの仕様」を参照してください。

```
{
    "header": {
        "report_id": 1529963534,
        "version": "1.0"
    },
    "metrics": {
        "listening_tcp_ports": {
            "ports": [
                {
                    "interface": "eth0",
                    "port": 24800
                },
                {
                    "interface": "eth0",
                    "port": 22
                },
                {
                    "interface": "eth0",
                    "port": 53
                }
            ],
```

```
            "total": 3
        },
        "listening_udp_ports": {
            "ports": [
                {
                    "interface": "eth0",
                    "port": 5353
                },
                {
                    "interface": "eth0",
                    "port": 67
                }
            ],
            "total": 2
        },
        "network_stats": {
            "bytes_in": 1157864729406,
            "bytes_out": 1170821865,
            "packets_in": 693092175031,
            "packets_out": 738917180
        },
        "tcp_connections": {
            "established_connections":{
                "connections": [
                    {
                    "local_interface": "eth0",
                    "local_port": 80,
                    "remote_addr": "192.168.0.1:8000"
                    },
                    {
                    "local_interface": "eth0",
                    "local_port": 80,
                    "remote_addr": "192.168.0.1:8000"
                    }
                ],
                "total": 2
            }
        }
    }
}
```

ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- 「AWS IoT デベロッパーガイド」の「Device Defender」

Docker アプリケーションのデプロイコネクタ

Greengrass Docker アプリケーションデプロイコネクタを使用すると、 AWS IoT Greengrass コアで Docker イメージを簡単に実行できます。コネクタは、Docker Compose を使用して、`docker-compose.yml` ファイルからマルチコンテナ Docker アプリケーションを起動します。具体的には、コネクタは、単一のコアデバイス上の Docker コンテナを管理する `docker-compose` コマンドを実行します。詳細については、Docker ドキュメントの「Docker 作成の概要」を参照してください。コネクタは、Amazon Elastic Container Registry (Amazon ECR)、Docker Hub、プライベート Docker

の信頼できるレジストリなど、Docker コンテナレジストリに格納された Docker イメージにアクセスできます。

Greengrass グループをデプロイすると、コネクタによって最新のイメージがプルされて Docker コンテナが起動します。実行されるコマンドは `docker-compose pull` と `docker-compose up` です。コネクタは次にそのコマンドのステータスを、出力 MQTT トピックに発行します。また、Docker コンテナの実行に関するステータス情報をログに記録します。これにより、Amazon CloudWatch のアプリケーションのログインを監視することができます。詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。このコネクタは、Greengrass デーモンが再起動するたびに Docker コンテナも起動します。コアで実行できる Docker コンテナの数は、ハードウェアによって異なります。

Docker コンテナはコアデバイス上の Greengrass ドメインの外部で実行されるため、コアのプロセス間通信 (IPC) にアクセスできません。ただし、ローカル Lambda 関数など、一部の通信チャネルを Greengrass コンポーネントで構成できます。詳細については、「the section called “Docker コンテナとの通信”」を参照してください。

コネクタは、コアデバイスで Web サーバーや MySQL サーバーをホストするなどのシナリオに使用できます。Docker アプリケーションのローカルサービスは、相互に、ローカル環境内の他のプロセス、およびクラウドサービスと通信できます。例えば、Lambda 関数からクラウド内の Web サービスにリクエストを送信する Web サーバーをコアで実行できます。

このコネクタは、コンテナなしモードで実行されるため、Greengrass コンテナ化なしで実行される Greengrass グループにデプロイできます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
7	`arn:aws:greengrass:  region::/connectors/DockerApplicationDeployment/versions/7`
6	`arn:aws:greengrass:  region::/connectors/DockerApplicationDeployment/versions/6`

バージョン	ARN
5	`arn:aws:greengrass:region::/connectors/DockerApplicationDeployment/versions/5`
4	`arn:aws:greengrass:region::/connectors/DockerApplicationDeployment/versions/4`
3	`arn:aws:greengrass:region::/connectors/DockerApplicationDeployment/versions/3`
2	`arn:aws:greengrass:region::/connectors/DockerApplicationDeployment/versions/2`
1	`arn:aws:greengrass:region::/connectors/DockerApplicationDeployment/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

- AWS IoT Greengrass Core ソフトウェア v1.10 以降。

> **Note**
> このコネクタは OpenWrt ディストリビューションではサポートされていません。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- Docker コンテナの実行を監視するためのコネクタ用の Greengrass コアには、最低 36 MB の RAM が必要です。総メモリ要件は、コアで実行される Docker コンテナの数によって異なります。
- Docker Engine 1.9.1 以降は Greengrass コアにインストールされています。バージョン 19.0.3 は、コネクタで動作することが確認された最新バージョンです。

  `docker` 実行可能ファイルは、`/usr/bin` ディレクトリまたは `/usr/local/bin` ディレクトリにある必要があります。

  > **Important**
  >
  > Docker 認証情報のローカルコピーを保護するために、認証情報ストアをインストールすることをお勧めします。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

  Amazon Linux ディストリビューションに Docker をインストールする方法については、「Amazon Elastic Container Service デベロッパーガイド」の「Amazon ECS で使用するコンテナイメージの作成」を参照してください。
- Docker Compose は、Greengrass のコアにインストールされています。`docker-compose` 実行可能ファイルは、`/usr/bin` ディレクトリまたは `/usr/local/bin` ディレクトリにある必要があります。

  次の Docker Compose のバージョンは、コネクタで動作することが確認されています。

コネクタのバージョン	検証済みの Docker Compose バージョン
7	1.25.4
6	1.25.4
5	1.25.4
4	1.25.4
3	1.25.4
2	1.25.1
1	1.24.1

- 1 つの Docker Compose ファイル (例えば、`docker-compose.yml`) が、Amazon Simple Storage Service (Amazon S3) に格納されている。フォーマットは、コアにインストールされている Docker Compose のバージョンと互換性がある必要があります。コアで使用する前に、ファイルをテストする必要があります。Greengrass グループのデプロイ後にファイルを編集する場合は、グループを再デプロイして、コア上のローカルコピーを更新する必要があります。
- ローカルの Docker デーモンを呼び出し、Compose ファイルのローカルコピーを格納するディレクトリに書き込む権限を持つ Linux ユーザー。詳細については、「コアでの Docker ユーザーの設定」を参照してください。
- Compose ファイルを含む S3 バケットでの `s3:GetObject` アクションを許可する Greengrass グループロール。このアクセス許可は、次の IAM ポリシーの例に示されています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToComposeFileS3Bucket",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:s3:::bucket-name/*"
        }
```

```
    ]
}
```

> **Note**
>
> S3 バケットでバージョニングが有効になっている場合は、ロールについても `s3:GetObjectVersion` アクションを許可するように設定されている必要があります。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「Using versioning」(バージョニングの使用) を参照してください。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)”またはthe section called “グループロールの管理 (CLI)”を参照してください。

- Docker Compose ファイルが Amazon ECR に保存されている Docker イメージを参照する場合、Greengrass グループロールは次のことを許可するように設定されています。
  - `ecr:GetDownloadUrlForLayer` および `ecr:BatchGetImage` は、Docker イメージを含む Amazon ECR リポジトリで実行します。
  - `ecr:GetAuthorizationToken` は、リソースを実行します。

  リポジトリは、コネクタと同じ AWS アカウント および AWS リージョン にある必要があります。

> **Important**
>
> グループロールのアクセス許可は、Greengrass グループ内のすべての Lambda 関数とコネクタによって引き継ぐことができます。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

これらのアクセス許可は、次のポリシーの例に示されています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowGetEcrRepositories",
```

```
            "Effect": "Allow",
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage"
            ],
            "Resource": [
                "arn:aws:ecr:region:account-id:repository/repository-name"
            ]
        },
        {
            "Sid": "AllowGetEcrAuthToken",
            "Effect": "Allow",
            "Action": "ecr:GetAuthorizationToken",
            "Resource": "*"
        }
    ]
}
```

詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR Repository Policy Examples」(Amazon ECR リポジトリポリシーの例) を参照してください。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)”またはthe section called “グループロールの管理 (CLI)”を参照してください。

- Docker Compose ファイルが AWS Marketplace から Docker イメージを参照する場合、コネクタには次の要件もあります。
  - AWS Marketplace コンテナ製品をサブスクライブする必要があります。詳細については、「AWS Marketplace 購入者ガイド」の「Finding and subscribing to container products」(コンテナ製品の検索とサブスクライブ) を参照してください。
  - AWS IoT Greengrass は、「シークレットの要件」で説明されているように、ローカルシークレットをサポートするように設定する必要があります。コネクタは、シークレットの保存ではなく AWS Secrets Manager、シークレットの取得にのみこの機能を使用します。
  - Compose ファイルで参照される Docker イメージを保存する AWS Marketplace レジストリごとに、Secrets Manager でシークレットを作成する必要があります。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。
- Docker Compose ファイルが Amazon ECR レジストリ以外のレジストリ (Docker Hub など) のプライベートリポジトリから Docker イメージを参照する場合、コネクタには次の要件もあります。

- AWS IoT Greengrass は、「シークレットの要件」で説明されているように、ローカルシークレットをサポートするように設定する必要があります。コネクタは、シークレットの保存ではなく AWS Secrets Manager、シークレットの取得にのみこの機能を使用します。
- Compose ファイルで参照される Docker イメージの格納先プライベートリポジトリごとに、Secrets Manager にシークレットを作成する必要があります。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。
- このコネクタを含む Greengrass グループをデプロイするときは、Docker デーモンが実行されている必要があります。

プライベートリポジトリからの Docker イメージへのアクセス

認証情報を使用して Docker イメージにアクセスする場合は、コネクタがイメージにアクセスできるようにする必要があります。これを行う方法は、Docker イメージの場所によって異なります。

Amazon ECR に保存された Docker イメージの場合 、Greengrass グループロールで認可トークンを取得する権限を付与します。詳細については、「the section called “要件”」を参照してください。

他のプライベートリポジトリまたはレジストリに保存されている Docker イメージの場合、ログイン情報を保存する AWS Secrets Manager には にシークレットを作成する必要があります。これには、サブスクライブした Docker イメージが含まれます AWS Marketplace。リポジトリごとに 1 つのシークレットを作成します。Secrets Manager でシークレットを更新すると、その変更は次にグループをデプロイするときにコアに伝達されます。

> **Note**
>
> Secrets Manager は、認証情報、キー、およびその他のシークレットを AWS クラウドに安全に保存および管理するために使用できるサービスです。詳細については、「 AWS Secrets Manager ユーザーガイド」の「What is AWS Secrets Manager？」を参照してください。

各シークレットには、次のキーが含まれている必要があります。

キー	値
`username`	リポジトリまたはレジストリへのアクセスに使用するユーザー名。

キー	値
`password`	リポジトリまたはレジストリへのアクセスに使用するパスワード。
`registryUrl`	レジストリのエンドポイント。これは、Compose ファイル内の対応するレジストリ URL と一致する必要があります。

> **Note**
> AWS IoT Greengrass がデフォルトでシークレットにアクセスできるようにするには、シークレットの名前が greengrass- で始まる必要があります。それ以外の場合、Greengrass サービスロールはアクセスを許可する必要があります。詳細については、「the section called "AWS IoT Greengrass シークレット値の取得を に許可する"」を参照してください。

Docker イメージのログイン情報を取得するには AWS Marketplace

1. `aws ecr get-login-password` コマンド AWS Marketplace を使用して、 から Docker イメージのパスワードを取得します。詳細については、「AWS CLI コマンドリファレンス」の「get-login-password」を参照してください。

   ```
   aws ecr get-login-password
   ```

2. Docker イメージのレジストリ URL を取得します。 AWS Marketplace ウェブサイトを開き、コンテナ製品の起動ページを開きます。[Container Images] (コンテナイメージ) で、[View container image details] (コンテナイメージの詳細を表示) をクリックして、ユーザー名とレジストリ URL を検索します。

取得したユーザー名、パスワード、レジストリ URL を使用して、Compose ファイルで参照される Docker イメージを保存する各 AWS Marketplace レジストリのシークレットを作成します。

シークレットを作成するには (コンソール)

AWS Secrets Manager コンソールで、その他のタイプのシークレットを選択します。[このシークレットに保存するキーと値のペアを指定します] で以下の操作を行い、`username`、`password`

および `registryUrl` の行を追加します。詳細については、「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。

シークレットを作成するには (CLI)

で AWS CLI、次の例に示すように Secrets Manager `create-secret` コマンドを使用します。詳細については、「AWS CLI コマンドリファレンス」の「create-secret」を参照してください。

```
aws secretsmanager create-secret --name greengrass-MySecret --secret-string
 [{"username":"Mary_Major"},{"password":"abc123xyz456"},{"registryUrl":"https://
docker.io"}]
```

> **Important**
>
> Docker Compose ファイルを格納する `DockerComposeFileDestinationPath` ディレクトリと、Docker イメージの資格情報をプライベートリポジトリから保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

パラメータ

このコネクタには、以下のパラメータが用意されています。

Version 7

`DockerComposeFileS3Bucket`

Docker Compose ファイルを含む S3 バケットの名前。バケットの作成時は、「Amazon Simple Storage Service ユーザーガイド」の「バケットの名前付け」に従ってください。

AWS IoT コンソールでの名前の表示: S3 の Docker Compose ファイル

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `[a-zA-Z0-9\\-\\.]{3,63}`

`DockerComposeFileS3Key`

Amazon S3 の Docker Compose ファイルのオブジェクトキー。オブジェクトの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `.+`

`DockerComposeFileS3Version`

Amazon S3 の Docker Compose ファイルのオブジェクトバージョン。オブジェクトキーの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットでのバージョニングの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `false`

型: `string`

有効なパターン `.+`

`DockerComposeFileDestinationPath`

Docker Compose ファイルのコピーを格納するために使用されるローカルディレクトリの絶対パス。これは既存のディレクトリでなければなりません。`DockerUserId` に指定したユーザーには、このディレクトリにファイルを作成する権限が必要です。詳細については、「the section called “コアでの Docker ユーザーの設定”」を参照してください。

> **Important**
> このディレクトリには、Docker Compose ファイルとプライベートリポジトリからの Docker イメージの資格情報が格納されます。このディレクトリを保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

AWS IoT コンソールの表示名: ローカル Compose ファイルのディレクトリパス

必須: `true`

型: `string`

有効なパターン `\/.*\/?`

例: `/home/username/myCompose`

`DockerUserId`

コネクタが実行される Linux ユーザーの UID。このユーザーは、コアデバイスの `docker` Linux グループに属し、`DockerComposeFileDestinationPath` ディレクトリへの書き込み権限を持っている必要があります。詳細については、「コアでの Docker ユーザーの設定」を参照してください。

> **Note**
>
> やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

AWS IoT コンソールの表示名: Docker ユーザー ID

必須: `false`

型: `string`

有効なパターン: `^[0-9]{1,5}$`

`AWSSecretsArnList`

プライベートリポジトリ内の Docker イメージへのアクセスに使用されるログイン情報を含む AWS Secrets Manager のシークレットの Amazon リソースネーム (ARNs)。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。

AWS IoT コンソールでの名前の表示: プライベートリポジトリの認証情報

必須: `false` このパラメータは、プライベートリポジトリに格納された Docker イメージにアクセスするために必要です。

タイプ: `string` の `array`

有効なパターン: `[( ?,? ?"(arn:(aws(-[a-z]+)):secretsmanager:[a-z0-9-]+:[0-9]{12}:secret:([a-zA-Z0-9\]+/)[a-zA-Z0-9/_+=,.@-]+-[a-zA-Z0-9]+)")]`

`DockerContainerStatusLogFrequency`

コネクタがコアで実行されている Docker コンテナに関するステータス情報をログに記録する頻度 (秒単位)。デフォルトは 300 秒 (5 分) です。

AWS IoT コンソールの表示名: ログ記録の頻度

必須: `false`

型: `string`

有効なパターン: `^[1-9]{1}[0-9]{0,3}$`

`ForceDeploy`

最後のデプロイの不適切なクリーンアップが原因で Docker デプロイが失敗する場合に、Docker デプロイを強制するかどうかを示します。デフォルト値は `False` です。

AWS IoT コンソールの表示名: 強制デプロイ

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`DockerPullBeforeUp`

デプロイヤがプルダウンアップ動作で `docker-compose up` の実行前に `docker-compose pull` を実行する必要があるかどうかを示します。デフォルト値は `True` です。

AWS IoT コンソールの表示名: Docker Pull Before Up

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`StopContainersOnNewDeployment`

GGC が停止した時に、Docker デプロイヤが管理する Docker コンテナをコネクタが停止させるかどうかを示します (GGC が停止するのは新しいグループがデプロイされた時、またはカーネルがシャットダウンされた時です)。デフォルト値は `True` です。

AWS IoT コンソールの表示名: 新しいデプロイで Docker が停止する

> **Note**
>
> このパラメータはデフォルトの `True` のままにしておくことをお勧めします。パラメータを `False`に設定すると、 AWS IoT Greengrass コアを終了した後や新しいデプロイを開始した後でも、Docker コンテナの実行が続行されます。このパラメータを `False` に設定する場合は、`docker-compose` サービス名の変更や追加があった時などの必要に応じて、Docker コンテナが正常に管理されていることを確認する必要があります。
> 詳細については、`docker-compose` Compose ファイルのドキュメントを参照してください。

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`DockerOfflineMode`

AWS IoT Greengrass をオフラインで起動する時に、既存の Docker Compose ファイルを使用するかどうかを示します。デフォルト値は `False` です。

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

Version 6

`DockerComposeFileS3Bucket`

Docker Compose ファイルを含む S3 バケットの名前。バケットの作成時は、「Amazon Simple Storage Service ユーザーガイド」の「バケットの名前付け」に従ってください。

AWS IoT コンソールでの名前の表示: S3 の Docker Compose ファイル

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `[a-zA-Z0-9\\-\\.]{3,63}`

`DockerComposeFileS3Key`

Amazon S3 の Docker Compose ファイルのオブジェクトキー。オブジェクトの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `.+`

`DockerComposeFileS3Version`

Amazon S3 の Docker Compose ファイルのオブジェクトバージョン。オブジェクトキーの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットでのバージョニングの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `false`

型: `string`

有効なパターン `.+`

`DockerComposeFileDestinationPath`

Docker Compose ファイルのコピーを格納するために使用されるローカルディレクトリの絶対パス。これは既存のディレクトリでなければなりません。`DockerUserId` に指定したユーザーには、このディレクトリにファイルを作成する権限が必要です。詳細については、「the section called “コアでの Docker ユーザーの設定”」を参照してください。

> **Important**
> このディレクトリには、Docker Compose ファイルとプライベートリポジトリからの Docker イメージの資格情報が格納されます。このディレクトリを保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

AWS IoT コンソールの表示名: ローカル Compose ファイルのディレクトリパス

必須: `true`

型: `string`

有効なパターン `\/.*\/?`

例: `/home/username/myCompose`

`DockerUserId`

コネクタが実行される Linux ユーザーの UID。このユーザーは、コアデバイスの `docker` Linux グループに属し、`DockerComposeFileDestinationPath` ディレクトリへの書き込

み権限を持っている必要があります。詳細については、「コアでの Docker ユーザーの設定」を参照してください。

> **Note**
>
> やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

AWS IoT コンソールの表示名: Docker ユーザー ID

必須: `false`

型: `string`

有効なパターン: `^[0-9]{1,5}$`

`AWSSecretsArnList`

プライベートリポジトリ内の Docker イメージへのアクセスに使用されるログイン情報を含む AWS Secrets Manager のシークレットの Amazon リソースネーム (ARNs)。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。

AWS IoT コンソールでの名前の表示: プライベートリポジトリの認証情報

必須: `false` このパラメータは、プライベートリポジトリに格納された Docker イメージにアクセスするために必要です。

タイプ: `string` の `array`

有効なパターン: `[( ?,? ?"(arn:(aws(-[a-z]+)):secretsmanager:[a-z0-9-]+:[0-9]{12}:secret:([a-zA-Z0-9\]+/)[a-zA-Z0-9/_+=,.@-]+-[a-zA-Z0-9]+)")]`

`DockerContainerStatusLogFrequency`

コネクタがコアで実行されている Docker コンテナに関するステータス情報をログに記録する頻度 (秒単位)。デフォルトは 300 秒 (5 分) です。

AWS IoT コンソールの表示名: ログ記録の頻度

必須: `false`

型: `string`

有効なパターン: `^[1-9]{1}[0-9]{0,3}$`

`ForceDeploy`

最後のデプロイの不適切なクリーンアップが原因で Docker デプロイが失敗する場合に、Docker デプロイを強制するかどうかを示します。デフォルト値は `False` です。

AWS IoT コンソールの表示名: 強制デプロイ

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`DockerPullBeforeUp`

デプロイヤがプルダウンアップ動作で `docker-compose up` の実行前に `docker-compose pull` を実行する必要があるかどうかを示します。デフォルト値は `True` です。

AWS IoT コンソールの表示名: Docker Pull Before Up

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`StopContainersOnNewDeployment`

GGC が停止した時に、Docker デプロイヤが管理する Docker コンテナを、コネクタが停止させるかどうかを示します (新しいグループがデプロイされた時、またはカーネルがシャットダウンされた時)。デフォルト値は `True` です。

AWS IoT コンソールの表示名: 新しいデプロイで Docker が停止する

> **Note**
>
> このパラメータはデフォルトの `True` のままにしておくことをお勧めします。パラメータを `False`に設定すると、 AWS IoT Greengrass コアを終了した後や新しいデプ

ロイを開始した後でも、Docker コンテナの実行が続行されます。このパラメータを `False` に設定する場合は、`docker-compose` サービス名の変更や追加があった時などの必要に応じて、Docker コンテナが正常に管理されていることを確認する必要があります。
詳細については、`docker-compose` Compose ファイルのドキュメントを参照してください。

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

Version 5

`DockerComposeFileS3Bucket`

Docker Compose ファイルを含む S3 バケットの名前。バケットの作成時は、「Amazon Simple Storage Service ユーザーガイド」の「バケットの名前付け」に従ってください。

AWS IoT コンソールでの名前の表示: S3 の Docker Compose ファイル

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `[a-zA-Z0-9\\-\\.]{3,63}`

`DockerComposeFileS3Key`

Amazon S3 の Docker Compose ファイルのオブジェクトキー。オブジェクトの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `.+`

`DockerComposeFileS3Version`

Amazon S3 の Docker Compose ファイルのオブジェクトバージョン。オブジェクトキーの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットでのバージョニングの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `false`

型: `string`

有効なパターン `.+`

`DockerComposeFileDestinationPath`

Docker Compose ファイルのコピーを格納するために使用されるローカルディレクトリの絶対パス。これは既存のディレクトリでなければなりません。`DockerUserId` に指定したユーザーには、このディレクトリにファイルを作成する権限が必要です。詳細については、「the section called “コアでの Docker ユーザーの設定”」を参照してください。

> **Important**
> このディレクトリには、Docker Compose ファイルとプライベートリポジトリからの Docker イメージの資格情報が格納されます。このディレクトリを保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

AWS IoT コンソールの表示名: ローカル Compose ファイルのディレクトリパス

必須: `true`

型: `string`

有効なパターン `\/.*\/?`

例: `/home/username/myCompose`

`DockerUserId`

コネクタが実行される Linux ユーザーの UID。このユーザーは、コアデバイスの `docker` Linux グループに属し、`DockerComposeFileDestinationPath` ディレクトリへの書き込み権限を持っている必要があります。詳細については、「コアでの Docker ユーザーの設定」を参照してください。

> **Note**
> やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

AWS IoT コンソールの表示名: Docker ユーザー ID

必須: `false`

型: `string`

有効なパターン: `^[0-9]{1,5}$`

`AWSSecretsArnList`

プライベートリポジトリ内の Docker イメージへのアクセスに使用されるログイン情報を含む AWS Secrets Manager のシークレットの Amazon リソースネーム (ARNs)。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。

AWS IoT コンソールでの名前の表示: プライベートリポジトリの認証情報

必須: `false` このパラメータは、プライベートリポジトリに格納された Docker イメージにアクセスするために必要です。

タイプ: `string` の `array`

有効なパターン: `[( ?,? ?"(arn:(aws(-[a-z]+)):secretsmanager:[a-z0-9-]+:[0-9]{12}:secret:([a-zA-Z0-9\]+/)[a-zA-Z0-9/_+=,.@-]+-[a-zA-Z0-9]+)")]`

`DockerContainerStatusLogFrequency`

コネクタがコアで実行されている Docker コンテナに関するステータス情報をログに記録する頻度 (秒単位)。デフォルトは 300 秒 (5 分) です。

AWS IoT コンソールでの名前の表示: ログ記録の頻度

必須: `false`

型: `string`

有効なパターン: `^[1-9]{1}[0-9]{0,3}$`

`ForceDeploy`

最後のデプロイの不適切なクリーンアップが原因で Docker デプロイが失敗する場合に、Docker デプロイを強制するかどうかを示します。デフォルト値は `False` です。

AWS IoT コンソールの表示名: 強制デプロイ

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

`DockerPullBeforeUp`

デプロイヤがプルダウンアップ動作で `docker-compose up` の実行前に `docker-compose pull` を実行する必要があるかどうかを示します。デフォルト値は `True` です。

AWS IoT コンソールの表示名: Docker Pull Before Up

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

Versions 2 - 4

`DockerComposeFileS3Bucket`

Docker Compose ファイルを含む S3 バケットの名前。バケットの作成時は、「Amazon Simple Storage Service ユーザーガイド」の「バケットの名前付け」に従ってください。

AWS IoT コンソールでの名前の表示: S3 の Docker Compose ファイル

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `[a-zA-Z0-9\\-\\.]{3,63}`

`DockerComposeFileS3Key`

Amazon S3 の Docker Compose ファイルのオブジェクトキー。オブジェクトの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `.+`

`DockerComposeFileS3Version`

Amazon S3 の Docker Compose ファイルのオブジェクトバージョン。オブジェクトキーの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットでのバージョニングの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `false`

型: `string`

有効なパターン `.+`

`DockerComposeFileDestinationPath`

Docker Compose ファイルのコピーを格納するために使用されるローカルディレクトリの絶対パス。これは既存のディレクトリでなければなりません。`DockerUserId` に指定したユーザーには、このディレクトリにファイルを作成する権限が必要です。詳細については、「the section called “コアでの Docker ユーザーの設定”」を参照してください。

> **Important**
> このディレクトリには、Docker Compose ファイルとプライベートリポジトリからの Docker イメージの資格情報が格納されます。このディレクトリを保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

AWS IoT コンソールの表示名: ローカル Compose ファイルのディレクトリパス

必須: `true`

型: `string`

有効なパターン `\/.*\/?`

例: `/home/username/myCompose`

`DockerUserId`

コネクタが実行される Linux ユーザーの UID。このユーザーは、コアデバイスの `docker` Linux グループに属し、`DockerComposeFileDestinationPath` ディレクトリへの書き込み権限を持っている必要があります。詳細については、「コアでの Docker ユーザーの設定」を参照してください。

> **Note**
> やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

AWS IoT コンソールの表示名: Docker ユーザー ID

必須: `false`

型: `string`

有効なパターン: `^[0-9]{1,5}$`

`AWSSecretsArnList`

プライベートリポジトリ内の Docker イメージへのアクセスに使用されるログイン情報を含む AWS Secrets Manager のシークレットの Amazon リソースネーム (ARNs)。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。

AWS IoT コンソールでの名前の表示: プライベートリポジトリの認証情報

必須: `false` このパラメータは、プライベートリポジトリに格納された Docker イメージにアクセスするために必要です。

タイプ: `string` の `array`

有効なパターン: `[( ?,? ?"(arn:(aws(-[a-z]+)):secretsmanager:[a-z0-9-]+:[0-9]{12}:secret:([a-zA-Z0-9\]+/)[a-zA-Z0-9/_+=,.@-]+-[a-zA-Z0-9]+)")]`

`DockerContainerStatusLogFrequency`

コネクタがコアで実行されている Docker コンテナに関するステータス情報をログに記録する頻度 (秒単位)。デフォルトは 300 秒 (5 分) です。

AWS IoT コンソールの表示名: ログ記録の頻度

必須: `false`

型: `string`

有効なパターン: `^[1-9]{1}[0-9]{0,3}$`

`ForceDeploy`

最後のデプロイの不適切なクリーンアップが原因で Docker デプロイが失敗する場合に、Docker デプロイを強制するかどうかを示します。デフォルト値は `False` です。

AWS IoT コンソールの表示名: 強制デプロイ

必須: `false`

型: `string`

有効なパターン: `^(true|false)$`

Version 1

`DockerComposeFileS3Bucket`

Docker Compose ファイルを含む S3 バケットの名前。バケットの作成時は、「Amazon Simple Storage Service ユーザーガイド」の「バケットの名前付け」に従ってください。

AWS IoT コンソールでの名前の表示: S3 の Docker Compose ファイル

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `[a-zA-Z0-9\\-\\.]{3,63}`

`DockerComposeFileS3Key`

Amazon S3 の Docker Compose ファイルのオブジェクトキー。オブジェクトの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

> **Note**
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `true`

型: `string`

有効なパターン `.+`

`DockerComposeFileS3Version`

Amazon S3 の Docker Compose ファイルのオブジェクトバージョン。オブジェクトキーの命名ガイドラインなどの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「S3 バケットでのバージョニングの使用」を参照してください。

> **Note**
>
> コンソールでは、S3 プロパティの Docker Compose ファイルは、`DockerComposeFileS3Bucket`、`DockerComposeFileS3Key`、および `DockerComposeFileS3Version` の各パラメータを結合します。

必須: `false`

型: `string`

有効なパターン `.+`

`DockerComposeFileDestinationPath`

Docker Compose ファイルのコピーを格納するために使用されるローカルディレクトリの絶対パス。これは既存のディレクトリでなければなりません。`DockerUserId` に指定したユーザーには、このディレクトリにファイルを作成する権限が必要です。詳細については、「the section called “コアでの Docker ユーザーの設定”」を参照してください。

> **Important**
>
> このディレクトリには、Docker Compose ファイルとプライベートリポジトリからの Docker イメージの資格情報が格納されます。このディレクトリを保護するのはユーザーの責任です。詳細については、「the section called “セキュリティ上の考慮事項”」を参照してください。

AWS IoT コンソールの表示名: ローカル Compose ファイルのディレクトリパス

必須: `true`

型: `string`

有効なパターン `\/.*\/?`

例: `/home/username/myCompose`

`DockerUserId`

コネクタが実行される Linux ユーザーの UID。このユーザーは、コアデバイスの `docker` Linux グループに属し、`DockerComposeFileDestinationPath` ディレクトリへの書き込み権限を持っている必要があります。詳細については、「コアでの Docker ユーザーの設定」を参照してください。

> **Note**
>
> やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

AWS IoT コンソールの表示名: Docker ユーザー ID

必須: `false`

型: `string`

有効なパターン: `^[0-9]{1,5}$`

`AWSSecretsArnList`

プライベートリポジトリ内の Docker イメージへのアクセスに使用されるログイン情報を含む AWS Secrets Manager のシークレットの Amazon リソースネーム (ARNs)。詳細については、「the section called “プライベートリポジトリからの Docker イメージへのアクセス”」を参照してください。

AWS IoT コンソールでの名前の表示: プライベートリポジトリの認証情報

必須: `false` このパラメータは、プライベートリポジトリに格納された Docker イメージにアクセスするために必要です。

タイプ: `string` の `array`

有効なパターン: `[( ?,? ?"(arn:(aws(-[a-z]+)):secretsmanager:[a-z0-9-]+:[0-9]{12}:secret:([a-zA-Z0-9\]+/)[a-zA-Z0-9/_+=,.@-]+-[a-zA-Z0-9]+)")]`

`DockerContainerStatusLogFrequency`

コネクタがコアで実行されている Docker コンテナに関するステータス情報をログに記録する頻度 (秒単位)。デフォルトは 300 秒 (5 分) です。

AWS IoT コンソールでの表示名: ログ記録の頻度

必須: `false`

型: `string`

有効なパターン: `^[1-9]{1}[0-9]{0,3}$`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Greengrass Docker アプリケーションデプロイコネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyDockerAppplicationDeploymentConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
DockerApplicationDeployment/versions/5",
            "Parameters": {
                "DockerComposeFileS3Bucket": "amzn-s3-demo-bucket",
                "DockerComposeFileS3Key": "production-docker-compose.yml",
                "DockerComposeFileS3Version": "123",
                "DockerComposeFileDestinationPath": "/home/username/myCompose",
                "DockerUserId": "1000",
                "AWSSecretsArnList": "[\"arn:aws:secretsmanager:region:account-
id:secret:greengrass-secret1-hash\",\"arn:aws:secretsmanager:region:account-
id:secret:greengrass-secret2-hash\"]",
                "DockerContainerStatusLogFrequency": "30",
                "ForceDeploy": "True",
                "DockerPullBeforeUp": "True"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

入力データ

このコネクタは入力データを必要としません。

出力データ

このコネクタは、`docker-compose up` コマンドのステータスを出力データとして公開します。

サブスクリプションのトピックフィルター

```
dockerapplicationdeploymentconnector/message/status
```

出力例: 成功

```
{
  "status":"success",
  "GreengrassDockerApplicationDeploymentStatus":"Successfully triggered docker-
compose up",
  "S3Bucket":"amzn-s3-demo-bucket",
  "ComposeFileName":"production-docker-compose.yml",
  "ComposeFileVersion":"123"
}
```

出力例: 失敗

```
{
  "status":"fail",
  "error_message":"description of error",
  "error":"InvalidParameter"
}
```

エラーの種類は `InvalidParameter` または `InternalError` です。

AWS IoT Greengrass コアでの Docker ユーザーの設定

Greengrass Docker アプリケーションデプロイコネクタは、`DockerUserId` パラメータで指定したユーザーを実行者として実行されます。値を指定しない場合、コネクタは `ggc_user` として実行されます。これは、デフォルトの Greengrass アクセス ID です。

コネクタが Docker デーモンと対話できるようにするには、Docker ユーザーがコアの `docker` Linux グループに属している必要があります。Docker ユーザーには、`DockerComposeFileDestinationPath` ディレクトリへの書き込み権限も必要です。これは、コネクタがローカル `docker-compose.yml` ファイルと Docker の資格情報を格納する場所です。

> **Note**
>
> - デフォルト `ggc_user` を使用する代わりに、Linux ユーザーを作成することをお勧めします。それ以外の場合は、Greengrass グループの任意の Lambda 関数が Compose ファイルと Docker の認証情報にアクセスできます。
> - やむを得ない場合を除き、root として実行することは避けてください。ルートユーザーを指定する場合は、 AWS IoT Greengrass Core で Lambda 関数をルートとして実行できるようにする必要があります。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

1. ユーザーを作成します。`useradd` コマンドを実行し、UID を割り当てる任意の `-u` オプションを含めることができます。例:

   ```
   sudo useradd -u 1234 user-name
   ```

2. コア上の `docker` グループにユーザーを追加します。例:

   ```
   sudo usermod -aG docker user-name
   ```

   `docker` グループの作成方法などの詳細については、Docker のドキュメントの「Docker を非ルートユーザーとして管理する」を参照してください。

3. `DockerComposeFileDestinationPath` パラメータに指定されたディレクトリへの書き込み権限をユーザーに付与します。例:

a. ユーザーをディレクトリの所有者として設定します。この例では、手順 1 の UID を使用します。

```
chown 1234 docker-compose-file-destination-path
```

b. 所有者に読み取りおよび書き込み権限を与えます。

```
chmod 700 docker-compose-file-destination-path
```

詳細については、Linux Foundation のドキュメントの「Linux でファイルとフォルダのアクセス許可を管理する方法」を参照してください。

c. ユーザーの作成時に UID を割り当てなかった場合、または既存のユーザーを使用している場合は、id コマンドを実行して UID を検索します。

```
id -u user-name
```

UID を使用して、コネクタの DockerUserId パラメータを構成します。

使用状況の情報

Greengrass Docker アプリケーションデプロイコネクタを使用する時は、次の実装固有の使用状況の情報に注意する必要があります。

- プロジェクト名の固定プレフィックス。コネクタは、開始する Docker コンテナの名前の前に greengrassdockerapplicationdeployment プレフィックスを追加します。コネクタは、実行する docker-compose コマンドで、このプレフィックスをプロジェクト名として使用します。
- ログ記録の動作。コネクタは、ステータス情報とトラブルシューティング情報をログファイルに書き込みます。CloudWatch Logs にログを送信し、ログをローカルに書き込むように AWS IoT Greengrass を設定できます。詳細については、「the section called “ロギング”」を参照してください。これは、コネクタのローカルログへのパスです。

```
/greengrass-root/ggc/var/log/user/region/aws/DockerApplicationDeployment.log
```

ローカルログにアクセスするには、root 権限が必要です。

- Docker イメージの更新。Docker はコアデバイスにイメージをキャッシュします。Docker イメージを更新し、変更をコアデバイスに伝播する場合は、Compose ファイルでイメージのタグを変更してください。変更は、Greengrass グループがデプロイされた後に有効になります。
- クリーンアップ操作の 10 分タイムアウト。Greengrass デーモンが再起動中に停止すると、`docker-compose down` コマンドが開始されます。すべての Docker コンテナには、`docker-compose down` が開始されてからクリーンアップ操作を実行するために 最大 10 分の時間が与えられます。10 分以内にクリーンアップが完了しない場合は、残りのコンテナを手動でクリーンアップする必要があります。詳細については、Docker CLI ドキュメントの「docker rm」を参照してください。
- Docker コマンドの実行。問題のトラブルシューティングを行うには、コアデバイスのターミナルウィンドウで Docker コマンドを実行します。例えば、次のコマンドを実行して、コネクタによって起動された Docker コンテナを表示します。

```
docker ps --filter name="greengrassdockerapplicationdeployment"
```

- 予約済みリソース ID。コネクタは、Greengrass グループで作成した Greengrass リソースの `DOCKER_DEPLOYER_SECRET_RESOURCE_RESERVED_ID_`*`index`* ID を使用します。リソース ID はグループ内で一意である必要があるため、この予約済みリソース ID と競合する可能性のあるリソース ID を割り当てないでください。
- オフラインモード。`DockerOfflineMode` 設定パラメータを `True` に設定すると、Docker コネクタはオフラインモードで動作します。これは、コアデバイスがオフラインの時に Greengrass グループのデプロイが再起動し、コネクタが Amazon S3 または Amazon ECR への接続を確立できず Docker Compose ファイルを取得できない場合に発生します。

  オフラインモードを有効にすると、コネクタは Compose ファイルのダウンロードを試み、通常の再起動の時と同じように `docker login` コマンドを実行します。ダウンロードに失敗すると、コネクタは `DockerComposeFileDestinationPath` パラメータで指定されたローカルのフォルダに保存されている Compose ファイルを検索します。ローカルの Compose ファイルが存在する場合、コネクタは `docker-compose` コマンドの通常処理に沿って動作し、ローカルのイメージからプルを実行します。Compose ファイルまたはローカルイメージが存在しない場合は、コネクタは失敗します。`ForceDeploy` と `StopContainersOnNewDeployment` パラメータの動作はオフラインモードでも変わりません。

Docker コンテナとの通信

AWS IoT Greengrass は、Greengrass コンポーネントと Docker コンテナ間の次の通信チャネルをサポートしています。

- Greengrass Lambda 関数は、REST API を使用して Docker コンテナ内のプロセスと通信できます。Docker コンテナ内に、ポートを開くサーバーを設定できます。Lambda 関数はこのポートを介してコンテナと通信できます。
- Docker コンテナ内のプロセスは、ローカルの Greengrass メッセージブローカーを介して MQTT メッセージを交換することができます。Docker コンテナを Greengrass グループのクライアントデバイスとして設定し、サブスクリプションを作成して、コンテナがグループ内の Greengrass Lambda 関数、クライアントデバイス、およびその他のコネクタ、または AWS IoT およびローカルシャドウサービスと通信できるようにします。詳細については、「the section called “Docker コンテナを使用した MQTT 通信の設定”」を参照してください。
- Greengrass Lambda 関数は、共有ファイルを更新して Docker コンテナに情報を渡すことができます。Compose ファイルを使用して、Docker コンテナの共有ファイルパスをバインドできます。

Docker コンテナを使用した MQTT 通信の設定

Docker コンテナを Greengrass デバイスとして設定し、クライアントグループに追加することができます。次に、Docker コンテナと Greengrass コンポーネントまたは AWS IoTの間の MQTT 通信を許可するサブスクリプションを作成できます。次の手順では、Docker コンテナデバイスがローカルのシャドウサービスからシャドウ更新メッセージを受信できるようにするサブスクリプションを作成します。このパターンに従って、他のサブスクリプションを作成できます。

> **Note**
>
> この手順では、Greengrass グループと Greengrass Core (v1.10 以降) が既に作成されていることを前提としています。Greengrass グループと Core の作成の詳細については、「の開始方法 AWS IoT Greengrass」を参照してください。

Docker コンテナをクライアントデバイスとして設定し、それを Greengrass グループに追加するには

1. コアデバイス上にフォルダを作成し、Greengrass デバイスの認証に使用する証明書とキーを保存します。

   ファイルパスは、起動する Docker コンテナにマウントする必要があります。次のスニペットは、Compose ファイルにファイルパスをマウントする方法を示しています。この例では、*path-to-device-certs* は、この手順で作成したフォルダを表します。

   ```
   version: '3.3'
   ```

```
services:
  myService:
    image: user-name/repo:image-tag
    volumes:
      -  /path-to-device-certs/:/path-accessible-in-container
```

2. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
3. ターゲットグループを選択します。
4. グループの設定ページで、[Client device] (クライアントデバイス)、[Associate] (アソシエイト) の順に選択します。
5. クライアントデバイスをこのグループモーダルに関連付けるで、新しい AWS IoT モノを作成するを選択します。

   新しいタブに [Create things] (モノの作成) ページが開きます。
6. [Creating things] (モノを作成する) ページで、[Create a single thing] (単一のモノを作成する) を選択し、[Next] (次へ) を選択します。
7. [Specify thing properties] (モノのプロパティを指定する) ページで、デバイスの名前を入力し、[Next] (次へ) を選択します。
8. [Configure device certificate] (デバイス証明書の設定) ページで [Next] (次へ) を選択します。
9. [Attach policies to certificate] (証明書へのポリシーのアタッチ) を選択し、次のいずれかを実行します。
   - クライアントデバイスが必要とする権限をグラントする既存のポリシーを選択し、[Create thing] (モノを作成する) を選択します。

     モーダルが開き、デバイスが とコアへの接続に使用する証明書 AWS クラウド とキーをダウンロードできます。
   - クライアントデバイスにアクセス許可を付与する新しいポリシーを作成してアタッチします。以下の操作を実行します。

     a. [ポリシーを作成] を選択します。

        [ポリシーの作成] ページが新しいタブで開きます。

     b. [ポリシーの作成] ページで、次の操作を行います。

        i. [Policy name] (ポリシー名) には、**GreengrassV1ClientDevicePolicy** など、ポリシーを説明する名前を入力します。

ii. [Policy statements] (ポリシーステートメント) タブの [Policy document] (ポリシードキュメント) で、[JSON] を選択します。

iii. 次のポリシードキュメントを入力します。このポリシーにより、クライアントデバイスは Greengrass コアを検出し、すべての MQTT トピックで通信できます。このポリシーのアクセスを制限する方法については、「AWS IoT Greengrassのデバイス認証と認可」を参照してください。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

iv. [作成] を選択して、ポリシーを作成します。

c. [Attach policies to certificate] (証明書にポリシーをアタッチ) ページを開き、ブラウザタブに戻ります。以下の操作を実行します。

i. [Policies] (ポリシー) 一覧で、[GreengrassV1ClientDevicePolicy] など、作成したポリシーを選択します。

新しいポリシーが表示されない場合は、更新ボタンを押します。

ii. [モノを作成する] を選択します。

モーダルが開き、デバイスが とコアへの接続に使用する証明書 AWS クラウド とキーをダウンロードできます。

10. [Download certificates and keys] (証明書と鍵をダウンロードする) モーダルで、デバイスの証明書をダウンロードします。

> **Important**
>
> [Done] (完了) を選択する前に、セキュリティリソースをダウンロードします。

以下の操作を実行します。

a. [デバイス証明書] で、[ダウンロード] を選択してデバイス証明書をダウンロードします。

b. [パブリックキーファイル] で、[ダウンロード] を選択して証明書のパブリックキーをダウンロードします。

c. [プライベートキーファイル] で、[ダウンロード] を選択して証明書のプライベートキーファイルをダウンロードします。

d. 「AWS IoT デベロッパーガイド」の「サーバー認証」を確認して、適切なルート CA 証明書を選択します。Amazon Trust Services (ATS) エンドポイントと ATS ルート CA 証明書の使用をお勧めします。[ルート CA 証明書] で、ルート CA 証明書の [ダウンロード] を選択します。

e. [完了] をクリックします。

デバイス証明書とキーのファイル名に含まれる共通の証明書 ID を書き留めます。これは後で必要になります。

11. ステップ 1 で作成したフォルダに、証明書とキーをコピーします。

次に、グループにサブスクリプションを作成します。この例では、Docker コンテナデバイスがローカルシャドウサービスから MQTT メッセージを受信できるようにするサブスクリプションを作成します。

> **Note**
>
> シャドウドキュメントの最大サイズは 8 KB です。詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT のクォータ」を参照してください。

Docker コンテナデバイスがローカルシャドウサービスから MQTT メッセージを受信できるようにするサブスクリプションを作成するには

1. グループ設定ページの [Subscriptions] (サブスクリプション) タブで、[Add Subscription] (サブスクリプションの追加) を選択します。
2. [ソースとターゲットの選択] ページで、ソースおよびターゲットを次のように設定します。
   a. [ソースの選択] で、[サービス]、[Local Shadow Service (ローカルシャドウサービス)] の順に選択します。
   b. [ターゲットの選択] で [デバイス] を選択し、あなたのデバイスを選択します。
   c. [次へ] を選択します。
   d. [Filter your data with a topic] (トピックでデータをフィルタリングする) ページで、[Topic filter] (トピックのフィルター) に「**`$aws/things/`*`MyDockerDevice`*`/shadow/update/accepted`**」と入力し、[Next] (次へ) を選択します。*`MyDockerDevice`* は、先ほど作成したデータベースの名前に置き換えます。
   e. [Finish] を選択してください。

Compose ファイルで参照する Docker イメージに次のコードスニペットを含めます。これは Greengrass のデバイスコードです。また、コンテナ内で Greengrass デバイスを起動する Docker コンテナにコードを追加します。イメージ内の別のプロセスとして、または別のスレッドで実行できます。

```
import os
import sys
import time
import uuid

from AWSIoTPythonSDK.core.greengrass.discovery.providers import DiscoveryInfoProvider
from AWSIoTPythonSDK.exception.AWSIoTExceptions import DiscoveryInvalidRequestException
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient

# Replace thingName with the name you registered for the Docker device.
```

```
thingName = "MyDockerDevice"
clientId = thingName

# Replace host with the IoT endpoint for your &AWS-account;.
host = "myPrefix.iot.region.amazonaws.com"

# Replace topic with the topic where the Docker container subscribes.
topic = "$aws/things/MyDockerDevice/shadow/update/accepted"

# Replace these paths based on the download location of the certificates for the Docker
 container.
rootCAPath = "/path-accessible-in-container/AmazonRootCA1.pem"
certificatePath = "/path-accessible-in-container/certId-certificate.pem.crt"
privateKeyPath = "/path-accessible-in-container/certId-private.pem.key"


# Discover Greengrass cores.
discoveryInfoProvider = DiscoveryInfoProvider()
discoveryInfoProvider.configureEndpoint(host)
discoveryInfoProvider.configureCredentials(rootCAPath, certificatePath, privateKeyPath)
discoveryInfoProvider.configureTimeout(10)  # 10 seconds.

GROUP_CA_PATH = "./groupCA/"
MQTT_QOS = 1

discovered = False
groupCA = None
coreInfo = None

try:
    # Get discovery info from AWS IoT.
    discoveryInfo = discoveryInfoProvider.discover(thingName)
    caList = discoveryInfo.getAllCas()
    coreList = discoveryInfo.getAllCores()

    # Use first discovery result.
    groupId, ca = caList[0]
    coreInfo = coreList[0]

    # Save the group CA to a local file.
    groupCA = GROUP_CA_PATH + groupId + "_CA_" + str(uuid.uuid4()) + ".crt"
    if not os.path.exists(GROUP_CA_PATH):
        os.makedirs(GROUP_CA_PATH)
    groupCAFile = open(groupCA, "w")
```

```
    groupCAFile.write(ca)
    groupCAFile.close()
    discovered = True
except DiscoveryInvalidRequestException as e:
    print("Invalid discovery request detected!")
    print("Type: %s" % str(type(e)))
    print("Error message: %s" % str(e))
    print("Stopping...")
except BaseException as e:
    print("Error in discovery!")
    print("Type: %s" % str(type(e)))
    print("Error message: %s" % str(e))
    print("Stopping...")

myAWSIoTMQTTClient = AWSIoTMQTTClient(clientId)
myAWSIoTMQTTClient.configureCredentials(groupCA, privateKeyPath, certificatePath)


# Try to connect to the Greengrass core.
connected = False
for connectivityInfo in coreInfo.connectivityInfoList:
    currentHost = connectivityInfo.host
    currentPort = connectivityInfo.port
    myAWSIoTMQTTClient.configureEndpoint(currentHost, currentPort)
    try:
        myAWSIoTMQTTClient.connect()
        connected = True
    except BaseException as e:
        print("Error in connect!")
        print("Type: %s" % str(type(e)))
        print("Error message: %s" % str(e))
    if connected:
        break

if not connected:
    print("Cannot connect to core %s. Exiting..." % coreInfo.coreThingArn)
    sys.exit(-2)

# Handle the MQTT message received from GGShadowService.
def customCallback(client, userdata, message):
    print("Received an MQTT message")
    print(message)

# Subscribe to the MQTT topic.
```

```
myAWSIoTMQTTClient.subscribe(topic, MQTT_QOS, customCallback)

# Keep the process alive to listen for messages.
while True:
    time.sleep(1)
```

セキュリティ上の考慮事項

Greengrass Docker アプリケーションデプロイコネクタを使用する場合は、次のセキュリティ上の考慮事項に注意してください。

Docker の Compose ファイルのローカルストレージ

コネクタは、`DockerComposeFileDestinationPath` パラメータに指定されたディレクトリに Compose ファイルのコピーを格納します。

このディレクトリを保護するのはあなたの責任です。ディレクトリへのアクセスを制限するには、ファイルシステムの権限を使用する必要があります。

Docker 認証情報のローカルストレージ

Docker イメージがプライベートリポジトリに格納されている場合、コネクタは、`DockerComposeFileDestinationPath` パラメータに指定されたディレクトリに Docker 認証情報を格納します。

これらの認証情報を保護するのはお客様の責任です。例えば、Docker Engine をインストールするときは、コアデバイスで資格情報ヘルパーを使用する必要があります。

信頼できるソースから Docker Engine をインストールする

信頼できるソースから Docker Engine をインストールするのはあなたの責任です。このコネクタは、コアデバイス上の Docker デーモンを使用して、Docker アセットにアクセスし、Docker コンテナを管理します。

Greengrass グループの役割権限のスコープ

Greengrass グループロールに追加したアクセス許可は、Greengrass グループのすべての Lambda 関数とコネクタによって引き継ぐことができます。このコネクタには、S3 バケットに格納されている Docker Compose ファイルへのアクセスが必要です。また、Docker イメージが Amazon ECR のプライベートリポジトリに格納されている場合は、Amazon ECR 認可トークンにアクセスする必要があります。

ライセンス

Greengrass Docker アプリケーションデプロイコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
7	がオフライン AWS IoT Greengrass で起動するときに既存の Docker Compose ファイルを使用する`DockerOfflineMode` ように を追加しました。`docker login` コマンドの再試行を実装しました。32 ビット UID に対応しました。
6	新しいデプロイが行われた時、または GGC が停止した時にコンテナのクリーンアップを上書きする `StopContainersOnNewDeployment` を追加しました。より安全なシャットダウンと起動の仕組み。YAML 検証でのバグを修正しました。
5	`docker-compose down` 実行前にイメージをプルしました。
4	Docker イメージを更新する起動前プルの動作を追加しました。
3	環境変数の検索に関する問題が修正されました。

バージョン	変更
2	`ForceDeploy` パラメータが追加されました。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

IoT Analytics コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

IoT Analytics コネクタは、ローカルデバイスデータを に送信します AWS IoT Analytics。このコネクタを中央ハブとして使用して、Greengrass コアデバイスのセンサーおよび接続されたクライアントデバイスからデータを収集できます。コネクタは、現在の AWS アカウント とリージョンの AWS IoT Analytics チャネルにデータを送信します。デフォルトの送信先チャネルおよび動的に指定されたチャネルにデータを送信できます。

> **Note**
>
> AWS IoT Analytics は、IoT データの収集、保存、処理、クエリを可能にするフルマネージドサービスです。では AWS IoT Analytics、データをさらに分析して処理できます。例えば、

マシンの健全性をモニタリングするために ML モデルをトレーニングしたり、新しいモデリング戦略をテストしたりするために使用できます。詳細については、「 AWS IoT Analytics ユーザーガイド」の「What is AWS IoT Analytics ?」を参照してください。

コネクタは、MQTT トピックの書式設定されたデータおよび書式設定されていないデータを受け入れます。送信先チャネルがインラインで指定された、2 つの事前定義されたトピックがサポートされます。また、サブスクリプションで設定された、お客様定義のトピックでメッセージを受信できます。これは、固定トピックに発行するクライアントデバイスからメッセージをルーティングしたり、リソースに制約のあるデバイスから、構造化されていないデータやスタックに依存したデータを処理したりするために使用できます。

このコネクタは、`BatchPutMessage` API を使用して、データ (JSON または base64 エンコードされた文字列) を送信先チャネルに送信します。コネクタは、raw データを API の要件に準拠した形式に処理できます。コネクタは入力メッセージをチャネルごとのキューにバッファし、非同期的にバッチを処理します。また、クエリおよびバッチ動作を制御し、メモリ使用を制限するパラメータを提供します。例えば、最大キューサイズ、バッチ間隔、メモリサイズ、アクティブなチャネルの数を設定できます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
4	`arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/4`
3	`arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/3`
2	`arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/2`
1	`arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3 - 4

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- このコネクタは、AWS IoT Greengrass および AWS IoT Analytics の両方がサポートされているアマゾン ウェブ サービスリージョンでのみ使用できます。
- 関連するすべての AWS IoT Analytics エンティティとワークフローが作成され、設定されます。エンティティには、チャネル、パイプライン、データストア、およびデータセットが含まれます。詳細については、「AWS IoT Analytics ユーザーガイド」の AWS CLI またはコンソールの手順を参照してください。

> **Note**
>
> 送信先 AWS IoT Analytics チャネルは、同じアカウントを使用し、このコネクタ AWS リージョン と同じ に存在する必要があります。

- 以下の IAM ポリシーの例に示すように、送信先チャネルに対する `iotanalytics:BatchPutMessage` アクションを許可するために Greengrass グループロールが設定されている。このチャネルは、現在の AWS アカウント とリージョンに存在する必要があります。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1528133056761",
            "Action": [
                "iotanalytics:BatchPutMessage"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:iotanalytics:region:account-id:channel/channel_1_name",
                "arn:aws:iotanalytics:region:account-id:channel/channel_2_name"
            ]
        }
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass および AWS IoT Analytics の両方がサポートされているアマゾン ウェブ サービスリージョンでのみ使用できます。
- 関連するすべての AWS IoT Analytics エンティティとワークフローが作成され、設定されます。エンティティには、チャネル、パイプライン、データストア、およびデータセットが含まれます。詳細については、「AWS IoT Analytics ユーザーガイド」の AWS CLI またはコンソールの手順を参照してください。

> **Note**
>
> 送信先 AWS IoT Analytics チャネルは、同じアカウントを使用し、このコネクタ AWS リージョン と同じ に存在する必要があります。

- 以下の IAM ポリシーの例に示すように、送信先チャネルに対する `iotanalytics:BatchPutMessage` アクションを許可するために Greengrass グループロールが設定されている。このチャネルは、現在の AWS アカウント とリージョンに存在する必要があります。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1528133056761",
            "Action": [
                "iotanalytics:BatchPutMessage"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:iotanalytics:region:account-id:channel/channel_1_name",
                "arn:aws:iotanalytics:region:account-id:channel/channel_2_name"
            ]
        }
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

パラメータ

`MemorySize`

このコネクタに割り当てるメモリ量 (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`PublishRegion`

AWS IoT Analytics チャネル AWS リージョン が作成される 。コネクタと同じリージョンを使用します。

> **Note**
> これは、グループロールで指定されたチャネルのリージョンに一致する必要があります。

AWS IoT コンソールの表示名: リージョンの公開

必須: `false`

型: `string`

有効なパターン: `^$|([a-z]{2}-[a-z]+-\\d{1})`

`PublishInterval`

受信したデータのバッチをパブリッシュする間隔 (秒単位) AWS IoT Analytics。

AWS IoT コンソールの表示名: 発行間隔

必須: `false`

型: `string`

デフォルト値: `1`

有効なパターン: `$|^[0-9]+$`

`IotAnalyticsMaxActiveChannels`

コネクタがアクティブに監視する AWS IoT Analytics チャネルの最大数。これは 0 より大きい必要があり、少なくとも、コネクタが同時に発行することが予期されるチャネル数と同じである必要があります。

このパラメータを使用すると、コネクタが同時に管理できるキューの合計数を限定して、メモリ消費量を制限することができます。キューに入れられたすべてのメッセージが送信されると、キューは削除されます。

AWS IoT コンソールの表示名: アクティブなチャネルの最大数

必須: `false`

型: `string`

デフォルト値: `50`

有効なパターン: `^$|^[1-9][0-9]*$`

`IotAnalyticsQueueDropBehavior`

キューがいっぱいであるときに、チャネルキューからメッセージを削除する動作。

AWS IoT コンソールでの名前の表示: キューのドロップ動作

必須: `false`

型: `string`

有効な値: `DROP_NEWEST` または `DROP_OLDEST`

デフォルト値: `DROP_NEWEST`

有効なパターン: `^DROP_NEWEST$|^DROP_OLDEST$`

`IotAnalyticsQueueSizePerChannel`

メッセージが送信または削除される前に、メモリに保持するされるメッセージの最大数 (1 チャネルあたり)。0 より大きくする必要があります。

AWS IoT コンソールの表示名: チャネルあたりの最大キューサイズ

必須: `false`

型: `string`

デフォルト値: `2048`

有効なパターン: `^$|^[1-9][0-9]*$`

`IotAnalyticsBatchSizePerChannel`

1 つのバッチリクエストで AWS IoT Analytics チャネルに送信するメッセージの最大数。0 より大きくする必要があります。

AWS IoT コンソールの表示名: チャネルごとにバッチ処理するメッセージの最大数

必須: `false`

型: `string`

デフォルト値: 5

有効なパターン: `^$|^[1-9][0-9]*$`

`IotAnalyticsDefaultChannelName`

このコネクタがカスタマー定義の入力トピックに送信されるメッセージに使用する AWS IoT Analytics チャネルの名前。

AWS IoT コンソールの表示名: デフォルトのチャネル名

必須: `false`

型: `string`

有効なパターン: `^[a-zA-Z0-9_]$`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: ^NoContainer$|^GreengrassContainer$

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、IoT Analytics コネクタを含む初期バージョンで ConnectorDefinition を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyIoTAnalyticsApplication",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/IoTAnalytics/
versions/3",
            "Parameters": {
                "MemorySize": "65535",
                "PublishRegion": "us-west-1",
                "PublishInterval": "2",
                "IotAnalyticsMaxActiveChannels": "25",
                "IotAnalyticsQueueDropBehavior": "DROP_OLDEST",
                "IotAnalyticsQueueSizePerChannel": "1028",
                "IotAnalyticsBatchSizePerChannel": "5",
                "IotAnalyticsDefaultChannelName": "my_channel"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、事前定義またはお客様定義の MQTT トピックでデータを受け入れます。発行者は、クライアントデバイス、Lambda 関数、またはその他のコネクタとすることができます。

事前定義されたトピック

コネクタは、発行者がチャネル名をインラインで定義できる、次の 2 つの構造化された MQTT トピックをサポートします。

- `iotanalytics/channels/+/messages/put` トピックのフォーマットされたメッセージ これらの入力メッセージの IoT データは、JSON または base64 でエンコードされた文字列としてフォーマットされる必要があります。
- `iotanalytics/channels/+/messages/binary/put` トピックのフォーマットされていないメッセージ このトピックで受信された入力メッセージはバイナリデータとして扱われ、任意のデータ型を含めることができます。

  事前定義されたトピックに発行するには、+ ワイルドカードをチャンネル名で置き換えます。例:

  ```
  iotanalytics/channels/my_channel/messages/put
  ```

お客様定義のトピック

コネクタは # トピック構文をサポートします。これにより、サブスクリプションで設定する任意の MQTT トピックで入力メッセージを受け入れることができます。サブスクリプションでは、# ワイルドカードのみを使用する代わりに、トピックパスを指定することをお勧めします。これらのメッセージは、コネクタに指定したデフォルトのチャネルに送信されます。

お客様定義のトピックの入力メッセージはバイナリデータとして処理されます。任意のメッセージ形式を使用し、任意のデータ型を含めることができます。お客様定義のトピックを使用すると、固定されたトピックに発行するデバイスからメッセージをルーティングできます。また、それらのトピックを使用して、データを処理できないクライアントデバイスから入力データをフォーマットされたメッセージに受け入れ、コネクタに送信することもできます。

サブスクリプションと MQTT トピックの詳細については、「the section called “入力と出力”」を参照してください。

グループロールは、すべての送信先チャネルで `iotanalytics:BatchPutMessage` アクションを許可する必要があります。詳細については、「the section called “要件”」を参照してください。

トピックのフィルター: `iotanalytics/channels/+/messages/put`

このトピックを使用して、フォーマットされたメッセージをコネクタに送信し、動的に送信先チャネルを指定します。このトピックでは、応答出力で返される ID を指定することもできま

す。コネクタは、IDs が送信先のアウトバウンドBatchPutMessageリクエスト内のメッセージごとに一意であることを確認します AWS IoT Analytics。ID が重複しているメッセージは削除されます。

このトピックに送信される入力データは、次のメッセージ形式を使用します。

メッセージのプロパティ

`request`

指定されたチャンネルに送信するデータ。

必須: `true`

型: 以下のプロパティを含む `object`。

`message`

デバイスまたはセンサーデータ (JSON または base64 でエンコードされた文字列)。

必須: `true`

型: `string`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。指定すると、レスポンスオブジェクトの `id` プロパティがこの値に設定されます。このプロパティを省略する場合は、コネクタによって ID が生成されます。

必須: `false`

型: `string`

有効なパターン: `.*`

入力例

```
{
    "request": {
        "message" : "{\"temp\":23.33}"
    },
```

```
    "id" : "req123"
}
```

トピックのフィルター: `iotanalytics/channels/+/messages/binary/put`

このトピックを使用して、フォーマットされていないメッセージをコネクタに送信し、動的に送信先チャネルを指定します。

コネクタのデータにより、このトピックで受信された入力メッセージは解析されません。これは、バイナリデータとして扱われます。メッセージを送信する前に AWS IoT Analytics、コネクタは `BatchPutMessage` API 要件に準拠するようにメッセージをエンコードしてフォーマットします。

- コネクタは base64 で raw データをエンコードし、エンコードされたペイロードをアウトバウンド `BatchPutMessage` リクエストに含めます。
- コネクタは ID を生成し、各入力メッセージに割り当てます。

> **Note**
> コネクタのレスポンス出力には、これらの入力メッセージの ID 相関は含まれません。

メッセージのプロパティ

なし。

トピックのフィルター: `#`

このトピックを使用して、任意のメッセージ形式をデフォルトのチャネルに送信します。これが特に役立つのは、クライアントデバイスが固定されたトピックに発行するときや、コネクタでサポートされているメッセージ形式にデータを処理できないクライアントデバイスからデフォルトのチャネルにデータを送信する場合です。

トピック構文は、このコネクタをデータソースに接続するために作成するサブスクリプションで定義します。サブスクリプションでは、# ワイルドカードのみを使用する代わりに、トピックパスを指定することをお勧めします。

コネクタのデータにより、この入力トピックに公開されるメッセージは解析されません。すべての入力メッセージはバイナリデータとして扱われます。メッセージを送信する前に AWS IoT Analytics、コネクタは `BatchPutMessage` API 要件に準拠するようにメッセージをエンコードしてフォーマットします。

- コネクタは base64 で raw データをエンコードし、エンコードされたペイロードをアウトバウンド `BatchPutMessage` リクエストに含めます。
- コネクタは ID を生成し、各入力メッセージに割り当てます。

> **Note**
> コネクタのレスポンス出力には、これらの入力メッセージの ID 相関は含まれません。

メッセージのプロパティ

なし。

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。この情報には、 が受信して送信する入力メッセージ AWS IoT Analytics ごとに から返されるレスポンスが含まれます AWS IoT Analytics。

サブスクリプションのトピックフィルター

```
iotanalytics/messages/put/status
```

出力例: 成功

```
{
    "response" : {
        "status" : "success"
    },
    "id" : "req123"
}
```

出力例: 失敗

```
{
    "response" : {
        "status" : "fail",
        "error" : "ResourceNotFoundException",
        "error_message" : "A resource with the specified name could not be found."
    },
    "id" : "req123"
}
```

> **Note**
>
> コネクタが再試行可能なエラー (接続エラーなど) を検出した場合は、次のバッチ処理で再発行を試します。エクスポネンシャルバックオフは AWS SDK によって処理されます。再試行可能なエラーが発生したリクエストは、チャネルキューに再度追加され、`IotAnalyticsQueueDropBehavior` パラメータに従ってさらに発行できます。

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。

   グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

b. コネクタを追加し、そのパラメータを設定します。

c. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

- Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
- コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'iotanalytics/channels/my_channel/messages/put'

def create_request_with_all_fields():
    return  {
        "request": {
            "message" : "{\"temp\":23.33}"
        },
        "id" : "req_123"
    }
```

```
def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=send_topic,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

制限

このコネクタには、次の制限が適用されます。

- `batch_put_message` アクション AWS SDK for Python (Boto3) に対して によって課される AWS IoT Analytics すべての制限。
- AWS IoT Analytics BatchPutMessage API によって課されるすべてのクォータ。詳細については、 AWS IoT Analytics のService Quotas」を参照してくださいAWS 全般のリファレンス。
  - チャネルごとに 1 秒あたり 100,000 件のメッセージ。
  - バッチごとに 100 件のメッセージ。
  - メッセージごとに 128 KB。

  この API では、チャネル名 (チャネル ARN ではありません) を使用するため、クロスリージョンまたはクロスアカウントチャネルへのデータの送信はサポートされません。
- AWS IoT Greengrass Core によって課されるすべてのクォータ。詳細については、 の AWS IoT Greengrass 「コアの Service Quotas」を参照してくださいAWS 全般のリファレンス。

  特に、以下のクォータが適用されます。
  - デバイスによって送信されるメッセージの最大サイズは 128 KB です。
  - Greengrass コアルーターのメッセージキューの最大サイズは、2.5 MB です。
  - トピック文字列の最大長は、UTF-8 エンコード文字で 256 バイトです。

ライセンス

IoT Analytics コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
4	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”

- the section called “コネクタの使用を開始する (CLI)”
- AWS IoT Analytics ユーザーガイドの「 AWS IoT Analyticsとは」

IoT イーサネット IP プロトコルアダプタコネクタ

IoT イーサネット IP プロトコルアダプタコネクタは、イーサネットプロトコルや IP プロトコルを使用してローカルデバイスからデータを収集します。このコネクタを使用して、複数のデバイスからデータを収集し、`StreamManager` メッセージストリームに発行できます。

このコネクタは、IoT SiteWise コネクタや IoT SiteWise ゲートウェイと共に使用することもできます。ゲートウェイは、コネクタの設定を提供する必要があります。詳細については、「IoT SiteWise ユーザーガイド」の「Configure an Ethernet/IP (EIP) source」(イーサネット/IP (EIP) ソースを設定する) を参照してください。

> **Note**
>
> このコネクタはコンテナ分離なしモードで実行されるため、Docker コンテナで実行されている AWS IoT Greengrass グループにデプロイできます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
2 (推奨)	`arn:aws:greengrass:  region::/connectors/IoTEIPProtocolAdaptor/versions/2`
1	`arn:aws:greengrass:  region::/connectors/IoTEIPProtocolAdaptor/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 1 and 2

- AWS IoT Greengrass Core ソフトウェア v1.10.2 以降。
- AWS IoT Greengrass グループで有効になっているストリームマネージャー。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- 最低 256 MB の追加 RAM。この要件は、 AWS IoT Greengrass Core メモリ要件に追加されます。

> **Note**
>
> このコネクタは、次のリージョンでのみ利用できます。
>
> - cn-north-1
> - ap-southeast-1
> - ap-southeast-2
> - eu-central-1
> - eu-west-1
> - us-east-1
> - us-west-2

コネクタパラメータ

このコネクタでは、以下のパラメータがサポートされています。

`LocalStoragePath`

IoT SiteWise コネクタが永続データを書き込むことができる AWS IoT Greengrass ホスト上のディレクトリ。デフォルトのディレクトリは `/var/sitewise` です。

AWS IoT コンソールでの名前の表示: ローカルストレージパス

必須: `false`

型: `string`

有効なパターン: `^\s*$|\/.`

`ProtocolAdapterConfiguration`

コネクタのデータ収集先または接続先となるイーサネット/IP コレクタの一連の設定。空のリストを指定できます。

AWS IoT コンソールの表示名: Protocol Adapter の設定

必須: `true`

タイプ: サポートされているフィードバック設定のセットを定義する正しい形式の JSON 文字列。

次は、`ProtocolAdapterConfiguration` の例です。

```
{
    "sources": [
        {
            "type": "EIPSource",
            "name": "TestSource",
            "endpoint": {
                "ipAddress": "52.89.2.42",
                "port": 44818
            },
            "destination": {
                "type": "StreamManager",
                "streamName": "MyOutput_Stream",
                "streamBufferSize": 10
            },
            "destinationPathPrefix": "EIPSource_Prefix",
            "propertyGroups": [
                {
                    "name": "DriveTemperatures",
                    "scanMode": {
                        "type": "POLL",
                        "rate": 10000
                    },
                    "tagPathDefinitions": [
                        {
                            "type": "EIPTagPath",
                            "path": "arrayREAL[0]",
                            "dstDataType": "double"
                        }
```

```
                    ]
                }
            ]
        }
    ]
}
```

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドを実行すると、IoT イーサネット IP プロトコルアダプタコネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version
'{
    "Connectors": [
        {
            "Id": "MyIoTEIPProtocolConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
IoTEIPProtocolAdaptor/versions/2",
            "Parameters": {
                "ProtocolAdaptorConfiguration": "{ \"sources\": [{ \"type
\": \"EIPSource\", \"name\": \"Source1\", \"endpoint\": { \"ipAddress\":
 \"54.245.77.218\", \"port\": 44818 }, \"destinationPathPrefix\": \"EIPConnector_Prefix
\", \"propertyGroups\": [{ \"name\": \"Values\", \"scanMode\": { \"type\": \"POLL\",
 \"rate\": 2000 }, \"tagPathDefinitions\": [{ \"type\": \"EIPTagPath\", \"path\":
 \"arrayREAL[0]\", \"dstDataType\": \"double\" }]}]}]}",
                "LocalStoragePath": "/var/MyIoTEIPProtocolConnectorState"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

入力データ

このコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

このコネクタは、StreamManager にデータを発行します。宛先メッセージストリームを設定する必要があります。出力メッセージは以下の構造になります。

```
{
    "alias": "string",
    "messages": [
        {
            "name": "string",
            "value": boolean|double|integer|string,
            "timestamp": number,
            "quality": "string"
        }
    ]
}
```

ライセンス

IoT イーサネット IP プロトコルアダプタコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- イーサネット/IP クライアント
- MapDB
- Elsa

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更	日付
2	このバージョンには、バグ修正が含まれています。	2021 年 12 月 23 日
1	初回リリース。	2020 年 12 月 15 日

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

IoT SiteWise コネクタ

IoT SiteWise コネクタは、ローカルデバイスと機器のデータを のアセットプロパティに送信します AWS IoT SiteWise。このコネクタを使用して、複数の OPC-UA サーバーからデータを収集し、IoT SiteWise に公開できます。コネクタは、現在の AWS アカウント およびリージョンのアセットプロパティにデータを送信します。

> **Note**
>
> IoT SiteWise は、産業機器や装置からデータを収集、処理、視覚化する、完全マネージド型サービスです。このコネクタからアセットの測定プロパティに送信される未加工データを処理するアセットプロパティを設定できます。たとえば、デバイスの摂氏温度データポイントを華氏 に変換する変換プロパティを定義したり、毎時の平均温度を計算するメトリクスプロパティを定義したりできます。詳細については、「AWS IoT SiteWise ユーザーガイド」の「AWS IoT SiteWiseとは」を参照してください。

コネクタは、OPC-UA サーバーから送信される OPC-UA データストリームパスを使用して IoT SiteWise にデータを送信します。たとえば、データストリームパス `/company/windfarm/3/turbine/7/temperature` は、風力発電所 #3 のタービン #7 の温度センサーを表すことができます。 AWS IoT Greengrass コアがインターネットへの接続を失った場合、コネクタは に正常に接続できるようになるまでデータをキャッシュします AWS クラウド。データのキャッシュに使用する最大ディスクバッファサイズを設定できます。キャッシュサイズが最大ディスクバッファサイズを超えると、コネクタはキューから最も古いデータを破棄します。

IoT SiteWise コネクタを設定およびデプロイしたら、IoT SiteWise コンソールでゲートウェイと OPC-UA ソースを追加できます。コンソールでソースを設定するときに、IoT SiteWise コネクタによって送信される OPC-UA データストリームパスをフィルタリングまたはプレフィックスできま

す。ゲートウェイおよびソースのセットアップを完了する手順については、「AWS IoT SiteWise ユーザーガイド」の「ゲートウェイの追加」を参照してください。

IoT SiteWise は、IoT SiteWise アセットの測定プロパティにマップしたデータストリームからのみデータを受け取ります。データストリームをアセットプロパティにマッピングするために、プロパティのエイリアスを OPC-UA データストリームパスと同等に設定できます。アセットモデルの定義とアセットの作成については、「AWS IoT SiteWise ユーザーガイド」の「産業用アセットのモデリング」を参照してください。

> **メモ**
>
> ストリームマネージャーを使用して、OPC-UA サーバー以外のソースから IoT SiteWise にデータをアップロードできます。ストリームマネージャーは、永続性と帯域幅管理のカスタマイズ可能なサポートも提供します。詳細については、「データストリームの管理」を参照してください。
>
> このコネクタは、コンテナなし分離モードで実行されるため、Docker コンテナで実行される Greengrass グループにデプロイできます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
12 (推奨)	`arn:aws:greengrass:region::/connectors/IoTSiteWise/versions/12`
11	`arn:aws:greengrass:region::/connectors/IoTSiteWise/versions/11`
10	`arn:aws:greengrass:region::/connectors/IoTSiteWise/versions/10`
9	`arn:aws:greengrass:region::/connectors/IoTSiteWise/versions/9`

バージョン	ARN
8	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 8
7	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 7
6	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 6
5	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 5
4	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 4
3	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 3
2	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 2
1	arn:aws:greengrass: *region*::/connectors/IoTSiteWise/versions/ 1

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 9, 10, 11, and 12

> **Important**
>
> このバージョンでは、 AWS IoT Greengrass Core ソフトウェア v1.10.2 とストリームマネージャーという新しい要件が導入されています。

- AWS IoT Greengrass Core ソフトウェア v1.10.2。
- Greengrass グループで有効なストリームマネージャー。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- Greengrass グループロールに追加された IAM ポリシー。このロールは、次の例に示すように、ターゲットルートアセットとその子の `iotsitewise:BatchPutAssetPropertyValue`アクションへの AWS IoT Greengrass グループアクセスを許可します。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
             }
        }
    ]
```

```
}
```

詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Versions 6, 7, and 8

> **Important**
>
> このバージョンでは、 AWS IoT Greengrass Core ソフトウェア v1.10.0 とストリームマネージャーという新しい要件が導入されています。

- AWS IoT Greengrass Core ソフトウェア v1.10.0。
- Greengrass グループで有効なストリームマネージャー。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- Greengrass グループロールに追加された IAM ポリシー。このロールは、次の例に示すように、ターゲットルートアセットとその子の `iotsitewise:BatchPutAssetPropertyValue`アクションへの AWS IoT Greengrass グループアクセスを許可します。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
```

```
                }
            }
        ]
}
```

詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Version 5

- AWS IoT Greengrass Core ソフトウェア v1.9.4。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- Greengrass グループロールに追加された IAM ポリシー。このロールは、次の例に示すように、ターゲットルートアセットとその子の `iotsitewise:BatchPutAssetPropertyValue`アクションへの AWS IoT Greengrass グループアクセスを許可します。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "iotsitewise:assetHierarchyPath": [
                        "/root node asset ID",
                        "/root node asset ID/*"
                    ]
                }
            }
        }
    ]
}
```

詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Version 4

- AWS IoT Greengrass Core ソフトウェア v1.10.0。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- Greengrass グループロールに追加された IAM ポリシー。このロールは、次の例に示すように、ターゲットルートアセットとその子の `iotsitewise:BatchPutAssetPropertyValue`アクションへの AWS IoT Greengrass グループアクセスを許可します。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
             }
        }
    ]
}
```

詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Version 3

- AWS IoT Greengrass Core ソフトウェア v1.9.4。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- Greengrass グループロールに追加された IAM ポリシー。このロールは、次の例に示すように、ターゲットルートアセットとその子の `iotsitewise:BatchPutAssetPropertyValue`アクションへの AWS IoT Greengrass グループアクセスを許可します。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
             }
        }
    ]
}
```

詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

Versions 1 and 2

- AWS IoT Greengrass Core ソフトウェア v1.9.4。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

- このコネクタは、AWS IoT Greengrass と IoT SiteWise の両方がサポートされているアマゾンウェブ サービスリージョンでのみ使用できます。
- 次の例に示すように、ターゲットルートアセット AWS IoT Core とその子に対する および `iotsitewise:BatchPutAssetPropertyValue`アクションへのアクセスを許可する Greengrass グループロールに追加された IAM ポリシー。ポリシーから `Condition` を削除して、コネクタがすべての IoT SiteWise アセットにアクセスできるようにすることができます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
             "Effect": "Allow",
             "Action": "iotsitewise:BatchPutAssetPropertyValue",
             "Resource": "*",
             "Condition": {
                 "StringLike": {
                     "iotsitewise:assetHierarchyPath": [
                         "/root node asset ID",
                         "/root node asset ID/*"
                     ]
                 }
             }
        },
        {
            "Effect": "Allow",
            "Action": [
                 "iot:Connect",
                 "iot:DescribeEndpoint",
                 "iot:Publish",
                 "iot:Receive",
                 "iot:Subscribe"
            ],
            "Resource": "*"
        }
    ]
}
```

詳細については、「 IAM ユーザーガイド」の「IAM ID アクセス許可の追加および削除」を参照してください。

パラメータ

Versions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 12

`SiteWiseLocalStoragePath`

IoT SiteWise コネクタが永続データを書き込むことができる AWS IoT Greengrass ホスト上のディレクトリ。デフォルトは `/var/sitewise` です。

AWS IoT コンソールでの名前の表示: ローカルストレージパス

必須: `false`

型: `string`

有効なパターン: `^\s*$|\/.`

`AWSSecretsArnList`

OPC-UA ユーザー名とパスワードのキーと値のペアがそれぞれ含まれ AWS Secrets Manager る のシークレットのリスト。各シークレットは、キーと値のペアタイプのシークレットである必要があります。

AWS IoT コンソールの表示名: OPC-UA ユーザー名/パスワードシークレットの ARNs のリスト

必須: `false`

型: `JsonArrayOfStrings`

有効なパターン: `\[( ?,? ?\"(arn:(aws(-[a-z]+)*):secretsmanager:[a-z0-9\\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\\\]+\/)*[a-zA-Z0-9\/_+=,.@\\-]+-[a-zA-Z0-9]+)*\")*\]`

`MaximumBufferSize`

IoT SiteWise ディスク使用量の最大サイズ (GB)。デフォルトは 10 GB です。

AWS IoT コンソールの表示名: 最大ディスクバッファサイズ

必須: `false`

型: `string`

有効なパターン: `^\s*$|[0-9]+`

Version 1

`SiteWiseLocalStoragePath`

IoT SiteWise コネクタが永続データを書き込むことができる AWS IoT Greengrass ホスト上のディレクトリ。デフォルトは `/var/sitewise` です。

AWS IoT コンソールでの名前の表示: ローカルストレージパス

必須: `false`

型: `string`

有効なパターン: `^\s*$|\/.`

`SiteWiseOpcuaUserIdentityTokenSecretArn`

OPC-UA ユーザー名とパスワードのキーと値のペア AWS Secrets Manager を含む のシークレット。このシークレットは、キーと値のペアタイプのシークレットである必要があります。

AWS IoT コンソールの表示名: OPC-UA ユーザー名/パスワードシークレットの ARN

必須: `false`

型: `string`

有効なパターン: `^$|arn:(aws(-[a-z]+)*):secretsmanager:[a-z0-9\\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\\\]+/)*[a-zA-Z0-9/_+=,.@\\-]+-[a-zA-Z0-9]+`

`SiteWiseOpcuaUserIdentityTokenSecretArn-ResourceId`

OPC-UA ユーザー名とパスワードシークレットを参照する AWS IoT Greengrass グループのシークレットリソース。

AWS IoT コンソールでの名前の表示: OPC-UA ユーザー名/パスワードシークレットリソース

必須: `false`

型: `string`

有効なパターン: `^$|.+`

`MaximumBufferSize`

IoT SiteWise ディスク使用量の最大サイズ (GB)。デフォルトは 10 GB です。

AWS IoT コンソールの表示名: 最大ディスクバッファサイズ

必須: `false`

型: `string`

有効なパターン: `^\s*$|[0-9]+`

サンプルコネクタを作成する (AWS CLI)

次の AWS CLI コマンドは、IoT SiteWise コネクタを含む初期バージョン`ConnectorDefinition`でを作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyIoTSiteWiseConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/IoTSiteWise/
versions/11"
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

このコネクタは MQTT メッセージを出力データとして公開しません。

制限

このコネクタには、IoT SiteWise によって課される、以下を含むすべての制限が適用されます。詳細については、「AWS 全般のリファレンス」の「AWS IoT SiteWise のエンドポイントとクォータ」を参照してください。

- AWS アカウントあたりのゲートウェイの最大数。
- ゲートウェイごとの OPC-UA ソースの最大数。
- ごとに保存されるtimestamp-quality-value (TQV) データポイントの最大レート AWS アカウント。
- アセットプロパティごとに保存される TQV データポイントの最大レート。

ライセンス

Version 9, 10, 11, and 12

IoT SiteWise コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- MapDB
- Elsa
- Eclipse Milo

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

Versions 6, 7, and 8

IoT SiteWise コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- Milo / EDL 1.0

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

Versions 1, 2, 3, 4, and 5

IoT SiteWise コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- Milo / EDL 1.0

- Chronicle-Queue / Apache License 2.0

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更	日付
12	• このバージョンには、バグ修正が含まれています。	2021 年 12 月 22 日
11	• 隠し文字や印刷できない文字を含む文字列のサポート。隠し文字と印刷不可能な文字は、文字列が AWS クラウドに送信される前に自動的に削除されます。 • IoT SiteWise ゲートウェイが無効なリクエストを無限にリトライする問題を修正しました。 • IoT SiteWise ゲートウェイが高頻度データソースに接続されている場合にチェックポイントが破損する問題を修正しました。 • ゲートウェイ設定のトラブルシューティングに役立つエラーメッセージを改善します。	2021 年 3 月 24 日
10	送信元接続が失われ、再確立された場合の処理を改善するよう `StreamManager` を	2021 年 1 月 22 日

バージョン	変更	日付
	設定しました。本バージョンでは、`SourceTimestamp` がない場合に `ServerTimestamp` で OPC-UA 値を受け付けることも可能です。	
9	カスタム Greengrass `StreamManager` ストリームの送信先、OPC-UA のデッドバンド、カスタムスキャンモード、カスタムスキャンレートのサポートが開始されました。また、IoT SiteWise ゲートウェイから行われた設定更新時のパフォーマンスの向上も含まれます。	2020 年 12 月 15 日
8	コネクターのネットワーク接続が断続的に発生する場合の安定性を改善しました。	2020 年 11 月 19 日
7	ゲートウェイメトリックスに関する問題を修正しました。	2020 年 8 月 14 日
6	CloudWatch メトリクスのサポートと、新しい OPC-UA タグの自動検出が追加されました。このバージョンには、ストリームマネージャーと AWS IoT Greengrass Core ソフトウェア v1.10.0 以降が必要です。	2020 年 4 月 29 日
5	AWS IoT Greengrass Core ソフトウェア v1.9.4 との互換性の問題を修正しました。	2020 年 2 月 12 日

バージョン	変更	日付
4	OPC-UA サーバーの再接続の問題を修正しました。	2020 年 2 月 7 日
3	`iot:*` アクセス許可要件を削除しました。	2019 年 12 月 17 日
2	複数の OPC-UA シークレットリソースのサポートを追加しました。	2019 年 12 月 10 日
1	初回リリース。	2019 年 12 月 2 日

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- AWS IoT SiteWise ユーザーガイドの次のトピックを参照してください。
  - とは AWS IoT SiteWise
  - ゲートウェイの使用
  - ゲートウェイの CloudWatch メトリクス
  - IoT SiteWise ゲートウェイのトラブルシューティング

Kinesis Firehose

Kinesis Firehose コネクタは、Amazon Data Firehose 配信ストリームを通じて、Amazon S3、Amazon Redshift、Amazon OpenSearch Service などの送信先にデータを発行します。

このコネクタは Kinesis 配信ストリームのデータプロデューサーです。MQTT トピックの入力データを受信し、指定された配信ストリームにデータを送信します。その後、配信ストリームは、設定されたターゲット (S3 バケットなど) にデータレコードを送信します。

このコネクタには、次のバージョンがあります。

バージョン	ARN
5	`arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/5`
4	`arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/4`
3	`arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/3`
2	`arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/2`
1	`arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 4 - 5

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- 設定された Kinesis 配信ストリーム。詳細については、「Amazon Kinesis Firehose デベロッパーガイド」の「Amazon Data Firehose 配信ストリームの作成」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲット配信ストリームに対する `firehose:PutRecord` アクションと `firehose:PutRecordBatch` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"Stmt1528133056761",
            "Action":[
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Effect":"Allow",
            "Resource":[
                "arn:aws:firehose:region:account-id:deliverystream/stream-name"
            ]
        }
    ]
 }
```

このコネクタにより、入力メッセージペイロードのデフォルトの配信ストリームを動的に上書きできるようになります。実装でこの機能を使用する場合、IAM ポリシーには、すべてのターゲットストリームがリソースとして含まれている必要があります。リソースにきめ細かいアク

セス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

Versions 2 - 3

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- 設定された Kinesis 配信ストリーム。詳細については、「Amazon Kinesis Firehose デベロッパーガイド」の「Amazon Data Firehose 配信ストリームの作成」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲット配信ストリームに対する `firehose:PutRecord` アクションと `firehose:PutRecordBatch` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"Stmt1528133056761",
            "Action":[
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Effect":"Allow",
            "Resource":[
                "arn:aws:firehose:region:account-id:deliverystream/stream-name"
            ]
        }
    ]
 }
```

このコネクタにより、入力メッセージペイロードのデフォルトの配信ストリームを動的に上書きできるようになります。実装でこの機能を使用する場合、IAM ポリシーには、すべてのターゲットストリームがリソースとして含まれている必要があります。リソースにきめ細かいアク

セス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called "グループロールの管理 (コンソール)" または the section called "グループロールの管理 (CLI)" を参照してください。

Version 1

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- 設定された Kinesis 配信ストリーム。詳細については、「Amazon Kinesis Firehose デベロッパーガイド」の「Amazon Data Firehose 配信ストリームの作成」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲット配信ストリームに対する `firehose:PutRecord` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"Stmt1528133056761",
            "Action":[
                "firehose:PutRecord"
            ],
            "Effect":"Allow",
            "Resource":[
                "arn:aws:firehose:region:account-id:deliverystream/stream-name"
            ]
        }
    ]
 }
```

このコネクタにより、入力メッセージペイロードのデフォルトの配信ストリームを動的に上書きできるようになります。実装でこの機能を使用する場合、IAM ポリシーには、すべてのターゲットストリームがリソースとして含まれている必要があります。リソースにきめ細かいアク

セス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Versions 5

`DefaultDeliveryStreamArn`

データの送信先となるデフォルトの Firehose 配信ストリームの ARN。ターゲットストリームは、入力メッセージペイロードの `delivery_stream_arn` プロパティによって上書きできます。

> **Note**
> グループロールを使用して、すべてのターゲット配信ストリームで適切なアクションを許可する必要があります。詳細については、「the section called “要件”」を参照してください。

AWS IoT コンソールの表示名: デフォルトの配信ストリーム ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9_\-.]+)$`

`DeliveryStreamQueueSize`

同じ配信ストリームの新しいレコードを拒否する前に、メモリに保持するレコードの最大数。最小値は 2000 です。

AWS IoT コンソールの表示名: バッファするレコードの最大数 (ストリームあたり)

必須: `true`

型: `string`

有効なパターン: `^([2-9]\\d{3}|[1-9]\\d{4,})$`

`MemorySize`

このコネクタに割り当てるメモリ量 (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`PublishInterval`

Firehose にレコードを発行する間隔 (秒単位)。バッチ適用を無効にするには、この値を 0 に設定します。

AWS IoT コンソールの表示名: 発行間隔

必須: `true`

型: `string`

有効な値: `0 - 900`

有効なパターン: `[0-9]|[1-9]\\d|[1-9]\\d\\d|900`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Versions 2 - 4

`DefaultDeliveryStreamArn`

データの送信先となるデフォルトの Firehose 配信ストリームの ARN。ターゲットストリームは、入力メッセージペイロードの `delivery_stream_arn` プロパティによって上書きできます。

> **Note**
> グループロールを使用して、すべてのターゲット配信ストリームで適切なアクションを許可する必要があります。詳細については、「the section called “要件”」を参照してください。

AWS IoT コンソールの表示名: デフォルトの配信ストリーム ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9_\-.]+)$`

`DeliveryStreamQueueSize`

同じ配信ストリームの新しいレコードを拒否する前に、メモリに保持するレコードの最大数。最小値は 2000 です。

AWS IoT コンソールの表示名: バッファするレコードの最大数 (ストリームあたり)

必須: `true`

型: `string`

有効なパターン: `^([2-9]\\d{3}|[1-9]\\d{4,})$`

`MemorySize`

このコネクタに割り当てるメモリ量 (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`PublishInterval`

Firehose にレコードを発行する間隔 (秒単位)。バッチ適用を無効にするには、この値を 0 に設定します。

AWS IoT コンソールの表示名: 発行間隔

必須: `true`

型: `string`

有効な値: `0 - 900`

有効なパターン: `[0-9]|[1-9]\\d|[1-9]\\d\\d|900`

Version 1

`DefaultDeliveryStreamArn`

データの送信先となるデフォルトの Firehose 配信ストリームの ARN。ターゲットストリームは、入力メッセージペイロードの `delivery_stream_arn` プロパティによって上書きできます。

> **Note**
>
> グループロールを使用して、すべてのターゲット配信ストリームで適切なアクションを許可する必要があります。詳細については、「the section called “要件”」を参照してください。

AWS IoT コンソールの表示名: デフォルトの配信ストリーム ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9_\-.]+)$`

Example

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、コネクタを含む初期バージョンで ConnectorDefinition を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyKinesisFirehoseConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/KinesisFirehose/
versions/5",
            "Parameters": {
                "DefaultDeliveryStreamArn": "arn:aws:firehose:region:account-
id:deliverystream/stream-name",
                "DeliveryStreamQueueSize": "5000",
                "MemorySize": "65535",
                "PublishInterval": "10",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、MQTT トピックのストリームコンテンツを受け取り、そのコンテンツをターゲット配信ストリームに送信します。受け取る入力データは以下の 2 種類です。

- `kinesisfirehose/message` トピックの JSON データ。
- `kinesisfirehose/message/binary/#` トピックのバイナリデータ。

Versions 2 - 5

トピックのフィルター: `kinesisfirehose/message`

このトピックを使用して、JSON データを含むメッセージを送信します。

メッセージのプロパティ

`request`

配信ストリームに送信するデータ、およびターゲット配信ストリーム (デフォルトストリームと異なる場合)。

必須: `true`

型: 以下のプロパティを含む `object`。

`data`

配信ストリームに送信するデータ。

必須: `true`

型: `string`

`delivery_stream_arn`

ターゲット Kinesis 配信ストリームの ARN。デフォルトの配信ストリームを上書きするには、このプロパティを含めます。

必須: `false`

型: `string`

有効なパターン: `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9_\-.]+)$`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。指定すると、レスポンスオブジェクトの `id` プロパティがこの値に設定されます。この機能を使用しない場合は、このプロパティを省略するか空の文字列を指定できます。

必須: `false`

型: `string`

有効なパターン: `.*`

入力例

```
{
     "request": {
        "delivery_stream_arn": "arn:aws:firehose:region:account-
id:deliverystream/stream2-name",
        "data": "Data to send to the delivery stream."
     },
     "id": "request123"
}
```

トピックのフィルター: `kinesisfirehose/message/binary/#`

このトピックを使用して、バイナリデータを含むメッセージを送信します。コネクタではバイナリデータは解析されません。データはそのままストリーミングされます。

入力リクエストを出力レスポンスにマッピングするには、メッセージトピックの # ワイルドカードを任意のリクエスト ID に置き換えます。例えば、メッセージを `kinesisfirehose/message/binary/request123` に発行する場合、レスポンスオブジェクトの `id` プロパティを `request123` に設定します。

リクエストをレスポンスにマッピングしない場合は、メッセージを `kinesisfirehose/message/binary/` に発行できます。末尾にスラッシュを含めてください。

Version 1

トピックのフィルター: `kinesisfirehose/message`

このトピックを使用して、JSON データを含むメッセージを送信します。

メッセージのプロパティ

`request`

配信ストリームに送信するデータ、およびターゲット配信ストリーム (デフォルトストリームと異なる場合)。

必須: `true`

型: 以下のプロパティを含む `object`。

`data`

配信ストリームに送信するデータ。

必須: `true`

型: `string`

`delivery_stream_arn`

ターゲット Kinesis 配信ストリームの ARN。デフォルトの配信ストリームを上書きするには、このプロパティを含めます。

必須: `false`

型: `string`

有効なパターン: `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9_\-.]+)$`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。指定すると、レスポンスオブジェクトの `id` プロパティがこの値に設定されます。この機能を使用しない場合は、このプロパティを省略するか空の文字列を指定できます。

必須: `false`

型: `string`

有効なパターン: `.*`

入力例

```
{
     "request": {
        "delivery_stream_arn": "arn:aws:firehose:region:account-
id:deliverystream/stream2-name",
        "data": "Data to send to the delivery stream."
     },
```

```
    "id": "request123"
}
```

トピックのフィルター: `kinesisfirehose/message/binary/#`

このトピックを使用して、バイナリデータを含むメッセージを送信します。コネクタではバイナリデータは解析されません。データはそのままストリーミングされます。

入力リクエストを出力レスポンスにマッピングするには、メッセージトピックの # ワイルドカードを任意のリクエスト ID に置き換えます。例えば、メッセージを `kinesisfirehose/message/binary/request123` に発行する場合、レスポンスオブジェクトの `id` プロパティを `request123` に設定します。

リクエストをレスポンスにマッピングしない場合は、メッセージを `kinesisfirehose/message/binary/` に発行できます。末尾にスラッシュを含めてください。

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。

Versions 2 - 5

サブスクリプションのトピックフィルター

`kinesisfirehose/message/status`

出力例

応答には、バッチで送信される各データレコードのステータスが含まれます。

```
{
    "response": [
        {
            "ErrorCode": "error",
            "ErrorMessage": "test error",
            "id": "request123",
            "status": "fail"
        },
        {
            "firehose_record_id": "xyz2",
            "id": "request456",
```

```
            "status": "success"
        },
        {
            "firehose_record_id": "xyz3",
            "id": "request890",
            "status": "success"
        }
    ]
}
```

> **Note**
>
> コネクタが再試行可能なエラー (接続エラーなど) を検出した場合は、次のバッチ処理で再発行を試します。エクスポネンシャルバックオフは AWS SDK によって処理されます。再試行可能なエラーで失敗したリクエストは、今後の発行するためにキューの最後に追加されます。

Version 1

サブスクリプションのトピックフィルター

```
kinesisfirehose/message/status
```

出力例: 成功

```
{
   "response": {
       "firehose_record_id": "1lxfuuuFomkpJYzt/34ZU/r8JYPf8Wyf7AXqlXm",
       "status": "success"
    },
    "id": "request123"
}
```

出力例: 失敗

```
{
   "response" : {
       "error": "ResourceNotFoundException",
       "error_message": "An error occurred (ResourceNotFoundException) when
 calling the PutRecord operation: Firehose test1 not found under account
 123456789012.",
```

```
        "status": "fail"
    },
    "id": "request123"
}
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。

   グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

b. コネクタを追加し、そのパラメータを設定します。

c. コネクタが JSON 入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

- Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
- コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。このメッセージには JSON データが含まれています。

```
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'kinesisfirehose/message'

def create_request_with_all_fields():
    return  {
        "request": {
            "data": "Message from Firehose Connector Test"
        },
        "id" : "req_123"
    }

def publish_basic_message():
```

```
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=send_topic,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

Kinesis Firehose コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
5	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
4	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。

バージョン	変更
3	過度のログ記録を減らすための修正、およびその他の軽微なバグの修正。
2	バッチ処理されたデータレコードを指定された間隔で Firehose に送信するためのサポートが追加されました。 • また、グループロールの `firehose:PutRecordBatch` アクションが必要になります。 • 新しい `MemorySize` 、`DeliveryStreamQueueSize` 、および `PublishInterval` パラメータ。 • 出力メッセージには、公開されたデータレコードのステータス応答の配列が含まれます。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- 「Amazon Kinesis デベロッパーガイド」の「Amazon Kinesis Data Firehose とは」

ML フィードバックコネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

この ML フィードバックコネクタにより、モデルの再トレーニングと分析のための機械学習 (ML) モデルデータへのアクセスが容易になります。コネクタ:

- ML モデルで使用される入力データ (サンプル) を Amazon S3 にアップロードします。モデル入力は、イメージ、JSON、オーディオなど、任意の形式にすることができます。サンプルをクラウドにアップロードした後、そのサンプルを使用してモデルを再トレーニングし、予測の正確性と精度を向上させることができます。たとえば、SageMaker AI Ground Truth を使用してサンプルにラベルを付け、SageMaker AI を使用してモデルを再トレーニングできます。
- モデルからの予測結果を MQTT メッセージとして発行します。これにより、モデルの推論品質をリアルタイムでモニタリングおよび分析できます。また、予測結果を保存し、それを使用して経時的な傾向を分析することもできます。
- サンプルアップロードとサンプルデータに関するメトリクスを Amazon CloudWatch に発行します。

このコネクタを設定するには、サポートされているフィードバック設定を JSON 形式で記述します。フィードバック設定では、送信先 Amazon S3 バケット、コンテンツタイプ、サンプリング戦略などのプロパティを定義します。(サンプリング戦略は、アップロードするサンプルを決定するために使用されます)。

ML フィードバックコネクタは、次のシナリオで使用できます。

- ユーザー定義関数と共に使用する。ローカル推論 Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK を使用してこのコネクタを呼び出し、ターゲットフィードバック設定、モデル入力、モデル出力 (予測結果) を渡します。例については、the section called “使用例”を参照してください。

- ML イメージ分類コネクタ (v2) と共に使用する。このコネクタを ML イメージ分類コネクタで使用するには、ML イメージ分類コネクタの `MLFeedbackConnectorConfigId` パラメータを設定します。
- ML オブジェクト検出コネクタと共に使用する。このコネクタを ML オブジェクト検出コネクタで使用するには、ML オブジェクト検出コネクタの `MLFeedbackConnectorConfigId` パラメータを設定します。

ARN: `arn:aws:greengrass:region::/connectors/MLFeedback/versions/1`

要件

このコネクタには以下の要件があります。

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- 1 つ以上の Amazon S3 バケット。使用するバケットの数は、サンプリング戦略によって異なります。
- 以下の IAM ポリシーの例に示すように、送信先 Amazon S3 バケットのオブジェクトに対して `s3:PutObject` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": "s3:PutObject",
            "Resource": [
                "arn:aws:s3:::bucket-name/*"
            ]
        }
    ]
}
```

ポリシーには、すべての送信先バケットをリソースとして含める必要があります。リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)”またはthe section called “グループロールの管理 (CLI)”を参照してください。

- CloudWatch Metrics コネクタが Greengrass グループに追加され、設定されている。これは、メトリクスレポート機能を使用する場合にのみ必要です。
- このコネクタを操作するには、AWS IoT Greengrass Machine Learning SDK v1.1.0 が必要です。

パラメータ

`FeedbackConfigurationMap`

コネクタが Amazon S3 にサンプルをアップロードするために使用できる 1 つ以上のフィードバック設定のセット。フィードバック設定は、送信先バケット、コンテンツタイプ、 サンプリング戦略などのパラメータを定義します。このコネクタが呼び出されると、呼び出し元の Lambda 関数またはコネクタはターゲットフィードバック設定を指定します。

AWS IoT コンソールの表示名: フィードバック設定マップ

必須: `true`

タイプ: サポートされているフィードバック設定のセットを定義する正しい形式の JSON 文字列。例については、the section called “FeedbackConfigurationMap の例”を参照してください。

フィードバック設定オブジェクトの ID には、次の要件があります。

ID:

- 設定オブジェクト間で一意である必要があります。
- 文字または数字で始まる必要があります。小文字と大文字の英文字、数字、およびハイフン (-) を含めることができます。
- 2 ~ 63 文字の長さにする必要があります。

必須: `true`

型: `string`

有効なパターン: `^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$`

例: `MyConfig0`、`config-a`、`12id`

フィードバック設定オブジェクトの本文には、次のプロパティが含まれます。

`s3-bucket-name`

宛先 Amazon S3 バケットの名前。

> **Note**
>
> グループロールは、すべての送信先バケットで `s3:PutObject` アクションを許可する必要があります。詳細については、「the section called “要件”」を参照してください。

必須: `true`

型: `string`

有効なパターン: `^[a-z0-9\.\-]{3,63}$`

`content-type`

アップロードするサンプルのコンテンツタイプ。個々のフィードバック設定のすべてのコンテンツは、同じタイプである必要があります。

必須: `true`

型: `string`

例: `image/jpeg`、`application/json`、`audio/ogg`

`s3-prefix`

アップロードされたサンプルに使用するキープレフィックス。プレフィックスはディレクトリ名に似ています。これにより、バケット内の同じディレクトリに類似したデータを保存できます。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「オブジェクトメタデータの使用」を参照してください。

必須: `false`

型: `string`

`file-ext`

アップロードされたサンプルに使用するファイル拡張子。コンテンツタイプの有効なファイル拡張子である必要があります。

必須: `false`

型: `string`

例: `jpg`、`json`、`ogg`

`sampling-strategy`

アップロードするサンプルをフィルタリングするために使用するサンプリング戦略。省略した場合、コネクタは受け取ったすべてのサンプルのアップロードを試みます。

必須: `false`

タイプ: 次のプロパティを含む正しい形式の JSON 文字列。

`strategy-name`

サンプリング戦略の名前。

必須: `true`

型: `string`

有効な値: `RANDOM_SAMPLING`、`LEAST_CONFIDENCE`、`MARGIN`、または `ENTROPY`

`rate`

Random サンプリング戦略のレート。

必須: `strategy-name` が `RANDOM_SAMPLING` の場合、`true`。

型: `number`

有効な値: `0.0 - 1.0`

`threshold`

Least Confidence、Margin、または Entropy サンプリング戦略のしきい値。

必須: `strategy-name` が `LEAST_CONFIDENCE`、`MARGIN`、または `ENTROPY` の場合、`true`。

型: `number`

有効な値:

- `LEAST_CONFIDENCE` または `MARGIN` 戦略の場合は `0.0 - 1.0`。
- `ENTROPY` 戦略の場合は `0.0 - no limit`。

`RequestLimit`

コネクタが一度に処理できるリクエストの最大数。

このパラメータを使用すると、コネクタが同時に処理するリクエストの合計数を限定して、メモリ消費量を制限することができます。この制限を超えるリクエストは無視されます。

AWS IoT コンソールの表示名: リクエストの制限

必須: `false`

型: `string`

有効な値: `0 - 999`

有効なパターン: `^$|^[0-9]{1,3}$`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、ML フィードバックコネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
```

```
    "Connectors": [
        {
            "Id": "MyMLFeedbackConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/MLFeedback/
versions/1",
            "Parameters": {
                "FeedbackConfigurationMap": "{  \"RandomSamplingConfiguration\":
 {  \"s3-bucket-name\": \"my-aws-bucket-random-sampling\",  \"content-type\":
 \"image/png\",  \"file-ext\": \"png\",  \"sampling-strategy\": {  \"strategy-name
\": \"RANDOM_SAMPLING\",  \"rate\": 0.5  } },  \"LeastConfidenceConfiguration\": {
  \"s3-bucket-name\": \"my-aws-bucket-least-confidence-sampling\",  \"content-type\":
 \"image/png\",  \"file-ext\": \"png\",  \"sampling-strategy\": {  \"strategy-name\":
 \"LEAST_CONFIDENCE\",  \"threshold\": 0.4  } } }",
                "RequestLimit": "10"
            }
        }
    ]
}'
```

FeedbackConfigurationMap の例

以下に示しているのは、FeedbackConfigurationMap パラメータ値の拡張例です。この例には、さまざまなサンプリング戦略を使用するいくつかのフィードバック設定が含まれています。

```
{
    "ConfigID1": {
        "s3-bucket-name": "my-aws-bucket-random-sampling",
        "content-type": "image/png",
        "file-ext": "png",
        "sampling-strategy": {
            "strategy-name": "RANDOM_SAMPLING",
            "rate": 0.5
        }
    },
    "ConfigID2": {
        "s3-bucket-name": "my-aws-bucket-margin-sampling",
        "content-type": "image/png",
        "file-ext": "png",
        "sampling-strategy": {
            "strategy-name": "MARGIN",
            "threshold": 0.4
        }
    },
```

```
    "ConfigID3": {
        "s3-bucket-name": "my-aws-bucket-least-confidence-sampling",
        "content-type": "image/png",
        "file-ext": "png",
        "sampling-strategy": {
            "strategy-name": "LEAST_CONFIDENCE",
            "threshold": 0.4
        }
    },
    "ConfigID4": {
        "s3-bucket-name": "my-aws-bucket-entropy-sampling",
        "content-type": "image/png",
        "file-ext": "png",
        "sampling-strategy": {
            "strategy-name": "ENTROPY",
            "threshold": 2
        }
    },
    "ConfigID5": {
        "s3-bucket-name": "my-aws-bucket-no-sampling",
        "s3-prefix": "DeviceA",
        "content-type": "application/json"
    }
}
```

サンプリング戦略

コネクタは、コネクタに渡されたサンプルをアップロードするかどうかを決定する 4 つのサンプリング戦略をサポートします。サンプルは、モデルが予測に使用するデータの個別のインスタンスです。サンプリング戦略を使用して、モデルの精度を向上させる可能性が最も高いサンプルをフィルタリングできます。

RANDOM_SAMPLING

指定されたレートに基づいてサンプルをランダムにアップロードします。ランダムに生成された値がレートより小さい場合、サンプルをアップロードします。レートが高いほど、アップロードされるサンプルが増えます。

> **Note**
>
> この戦略では、提供されたモデル予測は無視されます。

LEAST_CONFIDENCE

最大信頼確率が指定されたしきい値を下回るサンプルをアップロードします。

シナリオの例:

しきい値: .6

モデル予測: [.2, .2, .4, .2]

最大信頼確率: .4

結果:

最大信頼確率 (.4) <= しきい値 (.6) であるため、サンプルを使用します。

MARGIN

上位 2 つの信頼確率間のマージンが指定されたしきい値内である場合、サンプルをアップロードします。マージンは、上位 2 つの確率の差です。

シナリオの例:

しきい値: .02

モデル予測: [.3, .35, .34, .01]

トップ 2 つの信頼確率: [.35, .34]

マージン: .01 .35 - .34

結果:

マージン (.01) <= しきい値 (.02) であるため、サンプルを使用します。

ENTROPY

指定されたしきい値よりエントロピーが大きいサンプルをアップロードします。モデル予測の正規化されたエントロピーを使用します。

シナリオの例:

しきい値: 0.75

モデル予測: [.5, .25, .25]

予測のエントロピー: 1.03972

結果:

エントロピー (`1.03972`) > しきい値 (`0.75`) のため、サンプルを使用します。

入力データ

ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK の `feedback`クライアントの `publish`関数を使用してコネクタを呼び出します。例については、the section called “使用例”を参照してください。

> **Note**
>
> このコネクタは MQTT メッセージを入力データとして受け入れません。

`publish` 関数は次の引数を取ります。

ConfigId

ターゲットフィードバック設定の ID。これは、ML フィードバックコネクタの FeedbackConfigurationMap パラメータに定義されたフィードバック設定の ID と一致する必要があります。

必須: true

タイプ: 文字列

ModelInput

推論のためにモデルに渡された入力データ。この入力データは、サンプリング戦略に基づいて除外されない限り、ターゲット設定を使用してアップロードされます。

必須: true

タイプ: バイト

ModelPrediction

モデルからの予測結果。結果タイプは、ディクショナリまたはリストです。例えば、ML イメージ分類コネクタコネクタからの予測結果は、確率のリスト (`[0.25, 0.60, 0.15]` など) です。このデータは `/feedback/message/prediction` トピックに発行されます。

必須: true

タイプ: ディクショナリまたは float 値のリスト

メタデータ

アップロードされたサンプルにアタッチされ、/feedback/message/prediction トピックに発行される、お客様が定義したアプリケーション固有のメタデータ。また、コネクタはタイムスタンプ値を持つ publish-ts キーをメタデータに挿入します。

必須: false

タイプ: ディクショナリ

例: {"some-key": "some value"}

出力データ

このコネクタは、3 つの MQTT トピックにデータを発行します。

- feedback/message/status トピックに関するコネクタからのステータス情報。
- feedback/message/prediction トピックに関する予測結果。
- cloudwatch/metric/put トピックの CloudWatch を対象とするメトリクス。

コネクタが MQTT トピックで通信できるようにするには、サブスクリプションを設定する必要があります。詳細については、「the section called “入力と出力”」を参照してください。

トピックのフィルター: feedback/message/status

このトピックを使用して、サンプルのアップロードと削除されたサンプルのステータスをモニタリングします。コネクタは、リクエストを受け取るたびに、このトピックに発行します。

出力例: サンプルアップロードが成功しました

```
{
  "response": {
    "status": "success",
    "s3_response": {
      "ResponseMetadata": {
        "HostId": "IOWQ4fDEXAMPLEQM+ey7N9WgVhSnQ6JEXAMPLEZb7hSQDASK
+Jd1vEXAMPLEa3Km",
        "RetryAttempts": 1,
        "HTTPStatusCode": 200,
        "RequestId": "79104EXAMPLEB723",
```

```
        "HTTPHeaders": {
          "content-length": "0",
          "x-amz-id-2":
 "lbbqaDVFOhMlyU3gRvAX1ZIdg8P0WkGkCSSFsYFvSwLZk3j7QZhG5EXAMPLEdd4/pEXAMPLEUqU=",
          "server": "AmazonS3",
          "x-amz-expiration": "expiry-date=\"Wed, 17 Jul 2019 00:00:00 GMT\",
 rule-id=\"OGZjYWY3OTgtYWI2Zi00ZDllLWE4YmQtNzMyYzEXAMPLEoUw\"",
          "x-amz-request-id": "79104EXAMPLEB723",
          "etag": "\"b9c4f172e64458a5fd674EXAMPLE5628\"",
          "date": "Thu, 11 Jul 2019 00:12:50 GMT",
          "x-amz-server-side-encryption": "AES256"
        }
      },
      "bucket": "greengrass-feedback-connector-data-us-west-2",
      "ETag": "\"b9c4f172e64458a5fd674EXAMPLE5628\"",
      "Expiration": "expiry-date=\"Wed, 17 Jul 2019 00:00:00 GMT\", rule-id=
\"OGZjYWY3OTgtYWI2Zi00ZDllLWE4YmQtNzMyYzEXAMPLEoUw\"",
      "key": "s3-key-prefix/UUID.file_ext",
      "ServerSideEncryption": "AES256"
    }
  },
  "id": "5aaa913f-97a3-48ac-5907-18cd96b89eeb"
}
```

コネクタは、`bucket` フィールドと `key` フィールドを Amazon S3 からのレスポンスに追加します。Amazon S3 のレスポンスについては、「Amazon Simple Storage Service API リファレンス」の「PUT オブジェクト」を参照してください。

出力例: サンプリング戦略のために削除されたサンプル

```
{
  "response": {
    "status": "sample_dropped_by_strategy"
  },
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

出力例: サンプルアップロードに失敗しました

失敗ステータスには、エラーメッセージが `error_message` 値、例外クラスが `error` 値として含まれます。

```
{
```

```
  "response": {
    "status": "fail",
    "error_message": "[RequestId: 4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3] Failed
 to upload model input data due to exception. Model prediction will not be
 published. Exception type: NoSuchBucket, error: An error occurred (NoSuchBucket)
 when calling the PutObject operation: The specified bucket does not exist",
    "error": "NoSuchBucket"
  },
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

出力例: リクエスト制限のために調整されたリクエスト

```
{
  "response": {
    "status": "fail",
    "error_message": "Request limit has been reached (max request: 10 ). Dropping
 request.",
    "error": "Queue.Full"
  },
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

トピックのフィルター: `feedback/message/prediction`

このトピックを使用して、アップロードされたサンプルデータに基づく予測をリッスンします。これにより、モデルのパフォーマンスをリアルタイムで分析できます。モデル予測は、データが Amazon S3 に正常にアップロードされた場合にのみ、このトピックに発行されます。このトピックで発行されるメッセージは JSON 形式です。これには、アップロードされたデータオブジェクトへのリンク、モデルの予測、およびリクエストに含まれるメタデータが含まれます。

また、予測結果を保存し、それを使用して経時的な傾向を報告および分析することもできます。傾向は、価値あるインサイトを提供できます。たとえば、時間の傾向に伴って精度が低下した場合に、モデルを再トレーニングする必要があるかどうかを判断するのに役立ちます。

出力例

```
{
  "source-ref": "s3://greengrass-feedback-connector-data-us-west-2/s3-key-prefix/
UUID.file_ext",
  "model-prediction": [
    0.5,
```

```
    0.2,
    0.2,
    0.1
  ],
  "config-id": "ConfigID2",
  "metadata": {
    "publish-ts": "2019-07-11 00:12:48.816752"
  }
}
```

> **Tip**
>
> IoT Analytics コネクタを設定して、このトピックをサブスクライブし、情報を AWS IoT Analytics に送信して詳細分析または履歴分析を行うことができます。

トピックのフィルター: `cloudwatch/metric/put`

これは、CloudWatch にメトリクスを発行するために使用される出力トピックです。この機能を使用するには、CloudWatch Metrics コネクタをインストールして設定する必要があります。

メトリクスには次のものがあります。

- アップロードされたサンプルの数。
- アップロードされたサンプルのサイズ。
- Amazon S3 へのアップロードからのエラーの数。
- サンプリング戦略に基づいて削除されたサンプルの数。
- スロットルされたリクエストの数。

出力例: データサンプルのサイズ (実際のアップロード前に発行)

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 47592,
      "unit": "Bytes",
      "metricName": "SampleSize"
    }
  }
}
```

出力例: サンプルアップロードが成功しました

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleUploadSuccess"
    }
  }
}
```

出力例: サンプルアップロードが成功し、予測結果が発行されました

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleAndPredictionPublished"
    }
  }
}
```

出力例: サンプルアップロードに失敗しました

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleUploadFailure"
    }
  }
}
```

出力例: サンプリング戦略のために削除されたサンプル

```
{
```

```
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleNotUsed"
    }
  }
}
```

出力例: リクエスト制限のために調整されたリクエスト

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "ErrorRequestThrottled"
    }
  }
}
```

使用例

次の例は、AWS IoT Greengrass Machine Learning SDK を使用して ML フィードバックコネクタにデータを送信するユーザー定義 Lambda 関数です。

> **Note**
>
> AWS IoT Greengrass Machine Learning SDK は、 AWS IoT Greengrass ダウンロードページからダウンロードできます。

```
import json
import logging
import os
import sys
import greengrass_machine_learning_sdk as ml

client = ml.client('feedback')
```

```
try:
    feedback_config_id = os.environ["FEEDBACK_CONFIG_ID"]
    model_input_data_dir = os.environ["MODEL_INPUT_DIR"]
    model_prediction_str = os.environ["MODEL_PREDICTIONS"]
    model_prediction = json.loads(model_prediction_str)
except Exception as e:
    logging.info("Failed to open environment variables. Failed with exception:
{}".format(e))
    sys.exit(1)

try:
    with open(os.path.join(model_input_data_dir, os.listdir(model_input_data_dir)[0]),
 'rb') as f:
        content = f.read()
except Exception as e:
    logging.info("Failed to open model input directory. Failed with exception:
{}".format(e))
    sys.exit(1)

def invoke_feedback_connector():
    logging.info("Invoking feedback connector.")
    try:
        client.publish(
            ConfigId=feedback_config_id,
            ModelInput=content,
            ModelPrediction=model_prediction
        )
    except Exception as e:
        logging.info("Exception raised when invoking feedback connector:{}".format(e))
        sys.exit(1)

invoke_feedback_connector()

def function_handler(event, context):
    return
```

ライセンス

ML フィードバックコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0

- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

- six/MIT

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

ML イメージ分類コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

ML イメージ分類コネクタは、 AWS IoT Greengrass コアで実行される機械学習 (ML) 推論サービスを提供します。このローカル推論サービスは、SageMaker AI イメージ分類アルゴリズムによってトレーニングされたモデルを使用してイメージ分類を実行します。

ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK を使用して、ローカル推論サービスに推論リクエストを送信します。このサービスは、推論をローカルで実行し、入力イメージが特定のカテゴリに属する確率を返します。

AWS IoT Greengrass では、このコネクタの次のバージョンが提供されます。このバージョンは、複数のプラットフォームで使用できます。

Version 2

コネクタ	説明と ARN
ML イメージ分類 Aarch64 JTX2	NVIDIA Jetson TX2 のイメージ分類推論サービス。GPU アクセラレーションをサポートします。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationAarch64JTX2/versions/2`
ML イメージ分類 x86_64	x86_64 プラットフォーム用のイメージ分類推論サービス。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationx86-64/versions/2`
ML イメージ分類 ARMv7	ARMv7 プラットフォーム用のイメージ分類推論サービス。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationARMv7/versions/2`

Version 1

コネクタ	説明と ARN
ML イメージ分類 Aarch64 JTX2	NVIDIA Jetson TX2 のイメージ分類推論サービス。GPU アクセラレーションをサポートします。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationAarch64JTX2/versions/1`
ML イメージ分類 x86_64	x86_64 プラットフォーム用のイメージ分類推論サービス。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationx86-64/versions/1`
ML イメージ分類 Armv7	Armv7 プラットフォーム用のイメージ分類推論サービス。 ARN: `arn:aws:greengrass:region::/connectors/ImageClassificationARMv7/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

これらのコネクタには以下の要件があります。

Version 2

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- Core デバイスにインストールされた Apache MXNet フレームワークの依存関係。詳細については、「the section called “MXNet 依存関係のインストール”」を参照してください。
- SageMaker AI モデルソースを参照する Greengrass グループの ML リソース。このモデルは、SageMaker AI イメージ分類アルゴリズムによってトレーニングされる必要があります。詳細については、Amazon SageMakerデベロッパーガイド」の「イメージ分類アルゴリズム」を参照してください。
- ML フィードバックコネクタが Greengrass グループに追加され、設定されている。これは、コネクタを使用してモデル入力データをアップロードし、予測を MQTT トピックに発行する場合にのみ必要です。
- 以下の IAM ポリシーの例に示すように、ターゲットトレーニングジョブで `sagemaker:DescribeTrainingJob` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:DescribeTrainingJob"
            ],
            "Resource": "arn:aws:sagemaker:region:account-id:training-
job:training-job-name"
        }
```

```
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。ターゲットトレーニングジョブを今後変更する場合は、グループロールを更新してください。

- このコネクタを操作するには、AWS IoT Greengrass Machine Learning SDK v1.1.0 が必要です。

Version 1

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- Core デバイスにインストールされた Apache MXNet フレームワークの依存関係。詳細については、「the section called “MXNet 依存関係のインストール”」を参照してください。
- SageMaker AI モデルソースを参照する Greengrass グループの ML リソース。このモデルは、SageMaker AI イメージ分類アルゴリズムによってトレーニングされる必要があります。詳細については、Amazon SageMakerデベロッパーガイド」の「イメージ分類アルゴリズム」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲットトレーニングジョブで `sagemaker:DescribeTrainingJob` アクションを許可するために Greengrass グループロールが設定されている。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:DescribeTrainingJob"
            ],
```

```
            "Resource": "arn:aws:sagemaker:region:account-id:training-
job:training-job-name"
        }
    ]
}
```

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called "グループロールの管理 (コンソール)" または the section called "グループロールの管理 (CLI)" を参照してください。

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。ターゲットトレーニングジョブを今後変更する場合は、グループロールを更新してください。

- このコネクタを操作するには、AWS IoT Greengrass Machine Learning SDK v1.0.0 以降が必要です。

コネクタパラメータ

これらのコネクタでは、以下のパラメータを使用できます。

Version 2

`MLModelDestinationPath`

Lambda 環境内の ML リソースの絶対ローカルパス。これは、ML リソースに指定されたターゲットパスです。

> **Note**
> コンソールに ML リソースを作成した場合、これはローカルパスです。

AWS IoT コンソールの表示名: モデル送信先パス

必須: `true`

型: `string`

有効なパターン: `.+`

`MLModelResourceId`

ソースモデルを参照する ML リソースの ID。

AWS IoT コンソールの表示名: SageMaker ジョブ ARN リソース

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9:_-]+`

`MLModelSageMakerJobArn`

SageMaker AI モデルソースを表す SageMaker AI トレーニングジョブの ARN。モデルは SageMaker AI イメージ分類アルゴリズムによってトレーニングされる必要があります。

AWS IoT コンソールの表示名: SageMaker ジョブ ARN

必須: `true`

型: `string`

有効なパターン: `^arn:aws:sagemaker:[a-zA-Z0-9-]+:[0-9]+:training-job/[a-zA-Z0-9][a-zA-Z0-9-]+$`

`LocalInferenceServiceName`

ローカル推論サービスの名前。ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK の `invoke_inference_service`関数に名前を渡すことでサービスを呼び出します。例については、the section called “使用例”を参照してください。

AWS IoT コンソールでの名前の表示: ローカル推論サービス名

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9][a-zA-Z0-9-]{1,62}`

`LocalInferenceServiceTimeoutSeconds`

推論リクエストが終了するまでの時間 (秒単位)。最小値は 1 です。

AWS IoT コンソールの表示名: タイムアウト (秒)

必須: `true`

型: `string`

有効なパターン: `[1-9][0-9]*`

`LocalInferenceServiceMemoryLimitKB`

サービスがアクセスできるメモリの量 (KB 単位)。最小値は 1 です。

AWS IoT コンソールの表示名: メモリ制限 (KB)

必須: `true`

型: `string`

有効なパターン: `[1-9][0-9]*`

`GPUAcceleration`

CPU または GPU (アクセラレーション) コンピューティングの場合。このプロパティは ML イメージ分類 Aarch64 JTX2 コネクタにのみ適用されます。

AWS IoT コンソールの表示名: GPU アクセラレーション

必須: `true`

型: `string`

有効な値: `CPU` または `GPU`

`MLFeedbackConnectorConfigId`

モデル入力データのアップロードに使用するフィードバック設定の ID。これは、ML フィードバックコネクタに定義されたフィードバック設定の ID と一致する必要があります。

この パラメータは、ML フィードバックコネクタを使用してモデル入力データをアップロードし、予測を MQTT トピックに発行する場合にのみ必要です。

AWS IoT コンソールの表示名: ML フィードバックコネクタ設定 ID

必須: `false`

型: `string`

有効なパターン: `^$|^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$`

Version 1

`MLModelDestinationPath`

Lambda 環境内の ML リソースの絶対ローカルパス。これは、ML リソースに指定されたターゲットパスです。

> **Note**
> コンソールに ML リソースを作成した場合、これはローカルパスです。

AWS IoT コンソールの表示名: モデル送信先パス

必須: `true`

型: `string`

有効なパターン: `.+`

`MLModelResourceId`

ソースモデルを参照する ML リソースの ID。

AWS IoT コンソールの表示名: SageMaker ジョブ ARN リソース

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9:_-]+`

`MLModelSageMakerJobArn`

SageMaker AI モデルソースを表す SageMaker AI トレーニングジョブの ARN。モデルは SageMaker AI イメージ分類アルゴリズムによってトレーニングされる必要があります。

AWS IoT コンソールでの名前の表示: SageMaker ジョブ ARN

必須: `true`

型: `string`

有効なパターン: `^arn:aws:sagemaker:[a-zA-Z0-9-]+:[0-9]+:training-job/[a-zA-Z0-9][a-zA-Z0-9-]+$`

`LocalInferenceServiceName`

ローカル推論サービスの名前。ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK の `invoke_inference_service`関数に名前を渡すことでサービスを呼び出します。例については、the section called “使用例”を参照してください。

AWS IoT コンソールでの名前の表示: ローカル推論サービス名

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9][a-zA-Z0-9-]{1,62}`

`LocalInferenceServiceTimeoutSeconds`

推論リクエストが終了するまでの時間 (秒単位)。最小値は 1 です。

AWS IoT コンソールの表示名: タイムアウト (秒)

必須: `true`

型: `string`

有効なパターン: `[1-9][0-9]*`

`LocalInferenceServiceMemoryLimitKB`

サービスがアクセスできるメモリの量 (KB 単位)。最小値は 1 です。

AWS IoT コンソールの表示名: メモリ制限 (KB)

必須: `true`

型: `string`

有効なパターン: `[1-9][0-9]*`

`GPUAcceleration`

CPU または GPU (アクセラレーション) コンピューティングの場合。このプロパティは ML イメージ分類 Aarch64 JTX2 コネクタにのみ適用されます。

AWS IoT コンソールの表示名: GPU アクセラレーション

必須: `true`

型: `string`

有効な値: CPU または GPU

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、ML イメージ分類コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

例: CPU インスタンス

この例では、ML イメージ分類 Armv7l コネクタのインスタンスを作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyImageClassificationConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
ImageClassificationARMv7/versions/2",
            "Parameters": {
                "MLModelDestinationPath": "/path-to-model",
                "MLModelResourceId": "my-ml-resource",
                "MLModelSageMakerJobArn": "arn:aws:sagemaker:us-
west-2:123456789012:training-job:MyImageClassifier",
                "LocalInferenceServiceName": "imageClassification",
                "LocalInferenceServiceTimeoutSeconds": "10",
                "LocalInferenceServiceMemoryLimitKB": "500000",
                "MLFeedbackConnectorConfigId": "MyConfig0"
            }
        }
    ]
}'
```

例: GPU インスタンス

この例では、NVIDIA Jetson TX2 ボードで GPU アクセラレーションをサポートする ML イメージ分類 Aarch64 JTX2 コネクタのインスタンスを作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
```

```
        {
            "Id": "MyImageClassificationConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
ImageClassificationAarch64JTX2/versions/2",
            "Parameters": {
                "MLModelDestinationPath": "/path-to-model",
                "MLModelResourceId": "my-ml-resource",
                "MLModelSageMakerJobArn": "arn:aws:sagemaker:us-
west-2:123456789012:training-job:MyImageClassifier",
                "LocalInferenceServiceName": "imageClassification",
                "LocalInferenceServiceTimeoutSeconds": "10",
                "LocalInferenceServiceMemoryLimitKB": "500000",
                "GPUAcceleration": "GPU",
                "MLFeedbackConnectorConfigId": "MyConfig0"
            }
        }
    ]
}'
```

> **Note**
>
> これらのコネクタの Lambda 関数には、存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

これらのコネクタは、イメージファイルを入力として受け入れます。入力イメージファイルは `jpeg` または `png` 形式である必要があります。詳細については、「the section called “使用例”」を参照してください。

これらのコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

これらのコネクタは、入力イメージで識別されたオブジェクトのフォーマットされた予測を返します。

```
[0.3,0.1,0.04,...]
```

予測には、モデルトレーニング中にトレーニングデータセットで使用されたカテゴリに対応する値のリストが含まれます。各値は、イメージが対応するカテゴリに分類される確率を表します。確率が最も高いカテゴリが主要な予測です。

これらのコネクタは MQTT メッセージを出力データとして公開しません。

使用例

次の Lambda 関数の例は、AWS IoT Greengrass Machine Learning SDK を使用して、ML イメージ分類コネクタと対話します。

> **Note**
>
> この SDK は、AWS IoT Greengrass Machine Learning SDK のダウンロードページからダウンロードできます。

この例では、SDK クライアントを初期化し、SDK の `invoke_inference_service` 関数の同期呼び出しにより、ローカル推論サービスを呼び出します。次に、アルゴリズムタイプ、サービス名、イメージタイプ、イメージコンテンツを渡します。その後、サービスのレスポンスを解析して、確率の結果 (予測) を取得します。

Python 3.7

```
import logging
from threading import Timer

import numpy as np

import greengrass_machine_learning_sdk as ml

# We assume the inference input image is provided as a local file
# to this inference client Lambda function.
with open('/test_img/test.jpg', 'rb') as f:
    content = bytearray(f.read())

client = ml.client('inference')
```

```
def infer():
    logging.info('invoking Greengrass ML Inference service')

    try:
        resp = client.invoke_inference_service(
            AlgoType='image-classification',
            ServiceName='imageClassification',
            ContentType='image/jpeg',
            Body=content
        )
    except ml.GreengrassInferenceException as e:
        logging.info('inference exception {}("{}")'.format(e.__class__.__name__, e))
        return
    except ml.GreengrassDependencyException as e:
        logging.info('dependency exception {}("{}")'.format(e.__class__.__name__,
 e))
        return

    logging.info('resp: {}'.format(resp))
    predictions = resp['Body'].read().decode("utf-8")
    logging.info('predictions: {}'.format(predictions))

    # The connector output is in the format: [0.3,0.1,0.04,...]
    # Remove the '[' and ']' at the beginning and end.
    predictions = predictions[1:-1]
    count = len(predictions.split(','))
    predictions_arr = np.fromstring(predictions, count=count, sep=',')

    # Perform business logic that relies on the predictions_arr, which is an array
    # of probabilities.

    # Schedule the infer() function to run again in one second.
    Timer(1, infer).start()
    return

infer()

def function_handler(event, context):
    return
```

Python 2.7

```
import logging
```

```
from threading import Timer

import numpy

import greengrass_machine_learning_sdk as gg_ml

# The inference input image.
with open("/test_img/test.jpg", "rb") as f:
    content = f.read()

client = gg_ml.client("inference")


def infer():
    logging.info("Invoking Greengrass ML Inference service")

    try:
        resp = client.invoke_inference_service(
            AlgoType="image-classification",
            ServiceName="imageClassification",
            ContentType="image/jpeg",
            Body=content,
        )
    except gg_ml.GreengrassInferenceException as e:
        logging.info('Inference exception %s("%s")', e.__class__.__name__, e)
        return
    except gg_ml.GreengrassDependencyException as e:
        logging.info('Dependency exception %s("%s")', e.__class__.__name__, e)
        return

    logging.info("Response: %s", resp)
    predictions = resp["Body"].read()
    logging.info("Predictions: %s", predictions)

    # The connector output is in the format: [0.3,0.1,0.04,...]
    # Remove the '[' and ']' at the beginning and end.
    predictions = predictions[1:-1]
    predictions_arr = numpy.fromstring(predictions, sep=",")
    logging.info("Split into %s predictions.", len(predictions_arr))

    # Perform business logic that relies on predictions_arr, which is an array
    # of probabilities.

    # Schedule the infer() function to run again in one second.
```

```
    Timer(1, infer).start()


infer()


# In this example, the required AWS Lambda handler is never called.
def function_handler(event, context):
    return
```

Machine AWS IoT Greengrass Machine Learning SDK の `invoke_inference_service`関数は、次の引数を受け入れます。

引数	説明
`AlgoType`	推論に使用するアルゴリズムタイプの名前。現在は、`image-classification` のみがサポートされます。 必須: `true` 型: `string` 有効な値: `image-classification`
`ServiceName`	ローカル推論サービスの名前。コネクタを設定したときに `LocalInferenceServiceName` パラメータに指定した名前を使用します。 必須: `true` 型: `string`
`ContentType`	入力イメージの MIME タイプ。 必須: `true` 型: `string`

引数	説明
	有効な値: `image/jpeg, image/png`
Body	入力イメージファイルの内容。 必須: `true` 型: `binary`

AWS IoT Greengrass コアへの MXNet 依存関係のインストール

ML イメージ分類コネクタを使用するには、Apache MXNet フレームワークの依存関係をコアデバイスにインストールする必要があります。コネクタは、このフレームワークを使用して ML モデルを提供します。

> **Note**
> これらのコネクタは、プリコンパイルされた MXNet ライブラリにバンドルされているため、MXNet フレームワークをコアデバイスにインストールする必要はありません。

AWS IoT Greengrass には、次の一般的なプラットフォームとデバイスの依存関係をインストールするためのスクリプトが用意されています (または、インストールのリファレンスとして使用) 。別のプラットフォームやデバイスを使用している場合は、設定に応じた MXNet ドキュメントを参照してください。

MXNet の依存関係をインストールする前に、必要なシステムライブラリ (指定された最小バージョン) がデバイスに存在することを確認してください。

NVIDIA Jetson TX2

1. CUDA Toolkit 9.0 と cuDNN 7.0 をインストールします。開始方法チュートリアルの「the section called “他のデバイスの設定”」の手順に従うことができます。
2. コネクタでコミュニティ管理のオープンなソフトウェアをインストールできるように、ユニバースリポジトリを有効にします。詳細については、Ubuntu ドキュメントの Repositories/Ubuntu を参照してください。

   a. `/etc/apt/sources.list` ファイルを開きます。

b. 以下の行のコメントが解除されていることを確認してください。

```
deb http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
```

3. 以下のインストールスクリプトのコピーを Core デバイス上の `nvidiajtx2.sh` というファイルに保存します。

Python 3.7

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libatlas-base-
dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install
 OpenCV with pip on this platform. Try building the latest OpenCV from source
 (https://github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

Python 2.7

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libatlas-base-dev
 python-dev

echo 'Install latest pip...'
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
rm get-pip.py

pip install numpy==1.15.0 scipy

echo 'Dependency installation/upgrade complete.'
```

4. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo nvidiajtx2.sh
```

x86_64 (Ubuntu or Amazon Linux)

1. 以下のインストールスクリプトのコピーを Core デバイス上の `x86_64.sh` というファイルに保存します。

   Python 3.7

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)

if [ "$release" == '"Ubuntu"' ]; then
```

```
  # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies
 installed, so
  # this is mostly to prepare dependencies on Ubuntu EC2 instance.
  apt-get -y update
  apt-get -y dist-upgrade

  apt-get install -y libgfortran3 libsm6 libxext6 libxrender1
  apt-get install -y python3.7 python3.7-dev
elif [ "$release" == '"Amazon Linux"' ]; then
  # Amazon Linux. Expect python to be installed already
  yum -y update
  yum -y upgrade

  yum install -y compat-gcc-48-libgfortran libSM libXrender libXext
else
  echo "OS Release not supported: $release"
  exit 1
fi

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install
 OpenCV with pip on this platform. Try building the latest OpenCV from source
 (https://github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

Python 2.7

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
```

```
release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)

if [ "$release" == '"Ubuntu"' ]; then
  # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies
 installed, so
  # this is mostly to prepare dependencies on Ubuntu EC2 instance.
  apt-get -y update
  apt-get -y dist-upgrade

  apt-get install -y libgfortran3 libsm6 libxext6 libxrender1 python-dev
 python-pip
elif [ "$release" == '"Amazon Linux"' ]; then
  # Amazon Linux. Expect python to be installed already
  yum -y update
  yum -y upgrade

  yum install -y compat-gcc-48-libgfortran libSM libXrender libXext python-
pip
else
  echo "OS Release not supported: $release"
  exit 1
fi

pip install numpy==1.15.0 scipy opencv-python

echo 'Dependency installation/upgrade complete.'
```

2. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo x86_64.sh
```

Armv7 (Raspberry Pi)

1. 以下のインストールスクリプトのコピーを Core デバイス上の `armv7l.sh` というファイルに保存します。

   Python 3.7

```
#!/bin/bash
set -e
```

```
echo "Installing dependencies on the system..."

apt-get update
apt-get -y upgrade

apt-get install -y liblapack3 libopenblas-dev liblapack-dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install
 OpenCV with pip on this platform. Try building the latest OpenCV from source
 (https://github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

Python 2.7

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

apt-get update
apt-get -y upgrade

apt-get install -y liblapack3 libopenblas-dev liblapack-dev python-dev

# python-opencv depends on python-numpy. The latest version in the APT
 repository is python-numpy-1.8.2
# This script installs python-numpy first so that python-opencv can be
 installed, and then install the latest
# numpy-1.15.x with pip
```

```
apt-get install -y python-numpy python-opencv
dpkg --remove --force-depends python-numpy

echo 'Install latest pip...'
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
rm get-pip.py

pip install --upgrade numpy==1.15.0 picamera scipy

echo 'Dependency installation/upgrade complete.'
```

2. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo bash armv7l.sh
```

> **Note**
>
> Raspberry Pi では、`pip` を使用して機械学習の依存関係をインストールすると、メモリを大量に消費し、デバイスがメモリ不足になって応答しなくなる可能性があります。回避策として、スワップサイズを一時的に増やすことができます。`/etc/dphys-swapfile` で、`CONF_SWAPSIZE` 変数の値を増やし、次のコマンドを実行して `dphys-swapfile` を再起動します。
>
> ```
> /etc/init.d/dphys-swapfile restart
> ```

ログ記録とトラブルシューティング

グループ設定に応じて、イベントログとエラーログは、CloudWatch ログ、ローカルファイルシステム、またはその両方に書き込まれます。このコネクタのログにはプレフィックス `LocalInferenceServiceName` が使用されます。コネクタが予期しない動作を示した場合は、コネクタのログを確認します。このログには、通常、ML ライブラリの依存関係の不足やコネクタの起動失敗の原因など、デバッグに役立つ情報が含まれています。

AWS IoT Greengrass グループがローカルログを書き込むように設定されている場合、コネクタはログファイルを に書き込みます*`greengrass-root`*`/ggc/var/log/user/`*`region`*`/aws/`。Greengrass のログ記録の詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

ML イメージ分類コネクタの問題のトラブルシューティングには、以下の情報が役立ちます。

必須のシステムライブラリ

以下の各タブは、ML イメージ分類コネクタごとに必要なシステムライブラリを一覧表示します。

ML Image Classification Aarch64 JTX2

ライブラリ	最小バージョン
ld-linux-aarch64.so.1	GLIBC_2.17
libc.so.6	GLIBC_2.17
libcublas.so.9.0	該当なし
libcudart.so.9.0	該当なし
libcudnn.so.7	該当なし
libcufft.so.9.0	該当なし
libcurand.so.9.0	該当なし
libcusolver.so.9.0	該当なし
libgcc_s.so.1	GCC_4.2.0
libgomp.so.1	GOMP_4.0、OMP_1.0
libm.so.6	GLIBC_2.23
libpthread.so.0	GLIBC_2.17
librt.so.1	GLIBC_2.17
libstdc++.so.6	GLIBCXX_3.4.21、CXXABI_1.3.8

ML Image Classification x86_64

ライブラリ	最小バージョン
ld-linux-x86-64.so.2	GCC_4.0.0
libc.so.6	GLIBC_2.4
libgfortran.so.3	GFORTRAN_1.0
libm.so.6	GLIBC_2.23
libpthread.so.0	GLIBC_2.2.5
librt.so.1	GLIBC_2.2.5
libstdc++.so.6	CXXABI_1.3.8、GLIBCXX_3.4.21

ML Image Classification Armv7

ライブラリ	最小バージョン
ld-linux-armhf.so.3	GLIBC_2.4
libc.so.6	GLIBC_2.7
libgcc_s.so.1	GCC_4.0.0
libgfortran.so.3	GFORTRAN_1.0
libm.so.6	GLIBC_2.4
libpthread.so.0	GLIBC_2.4
librt.so.1	GLIBC_2.4
libstdc++.so.6	CXXABI_1.3.8、CXXABI_ARM_1.3.3、GLIBCXX_3.4.20

問題点

症状	ソリューション
Raspberry Pi で、次のエラーメッセージがログに記録される。カメラは使用していない。`Failed to initialize libdc1394`	次のコマンドを実行してドライバーを無効にします。 `sudo ln /dev/null /dev/raw1394` このオペレーションは一時的なものであり、シンボリックリンクは再起動後に消えます。再起動時にリンクを自動的に作成する方法については、OS ディストリビューションのマニュアルを参照してください。

ライセンス

ML イメージ分類コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

- Deep Neural Network Library (DNNL)/Apache License 2.0
- OpenMP* ランタイムライブラリ/「Intel OpenMP ランタイムライブラリのライセンス」を参照してください。
- mxnet/Apache License 2.0
- six/MIT

Intel OpenMP ランタイムライブラリのライセンス。Intel® OpenMP* ランタイムはデュアルライセンスで、Intel® Parallel Studio XE Suite 製品の一部としての商用 (COM) ライセンスと、BSD オープンソース (OSS) ライセンスがあります。

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
2	モデル入力データのアップロード、MQTT トピックへの予測の発行、Amazon CloudWatch へのメトリクスの発行のために ML フィードバックコネクタの使用をサポートする `MLFeedbackConnectorConfigId` パラメータが追加されました。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- 機械学習の推論を実行する
- Amazon SageMaker AI デベロッパーガイド」の「イメージ分類アルゴリズム」

ML オブジェクト検出コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

ML オブジェクト検出コネクタは、 AWS IoT Greengrass コアで実行される機械学習 (ML) 推論サービスを提供します。このローカル推論サービスは、SageMaker AI Neo 深層学習コンパイラによってコンパイルされたオブジェクト検出モデルを使用してオブジェクト検出を実行します。Single Shot Multibox Detector (SSD) と You Only Look Once (YOLO) v3 の 2 種類のオブジェクト検出モデルがサポートされています。詳細については、「オブジェクト検出モデルの要件」を参照してください。

ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK を使用して、ローカル推論サービスに推論リクエストを送信します。このサービスは、入力イメージでローカル推論を実行し、イメージで検出された各オブジェクトの予測のリストを返します。各予測には、オブジェクトカテゴリ、予測の信頼スコア、および予測されたオブジェクトの周囲の境界ボックスを指定するピクセル座標が含まれます。

AWS IoT Greengrass は、複数のプラットフォーム用の ML オブジェクト検出コネクタを提供します。

コネクタ	説明と ARN
ML オブジェクト検出 Aarch64 JTX2	NVIDIA Jetson TX2 のオブジェクト検出推論サービス。GPU アクセラレーションをサポートします。 ARN: `arn:aws:greengrass:region::/connectors/ObjectDetectionAarch64JTX2/versions/1`
ML オブジェクト検出 x86_64	x86_64 プラットフォームのオブジェクト検出推論サービス。

コネクタ	説明と ARN
	ARN: `arn:aws:greengrass:region::/connectors/ObjectDetectionx86-64/versions/1`
ML オブジェクト検出 ARMv7	ARMv7 プラットフォームのオブジェクト検出推論サービス。 ARN: `arn:aws:greengrass:region::/connectors/ObjectDetectionARMv7/versions/1`

要件

これらのコネクタには以下の要件があります。

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- コアデバイスにインストールされた SageMaker AI Neo 深層学習ランタイムの依存関係。詳細については、「the section called “Neo 深層学習ランタイム依存関係のインストール”」を参照してください。
- Greengrass グループの ML リソース。ML リソースは、オブジェクト検出モデルを含む Amazon S3 バケットを参照する必要があります。詳細については、「Amazon S3 のモデルソース」を参照してください。

> **Note**
>
> このモデルは、Single Shot Multibox Detector または You Only Look Once v3 オブジェクト検出モデルタイプである必要があります。SageMaker AI Neo 深層学習コンパイラを使用してコンパイルする必要があります。詳細については、「オブジェクト検出モデルの要件」を参照してください。

- ML フィードバックコネクタが Greengrass グループに追加され、設定されている。これは、コネクタを使用してモデル入力データをアップロードし、予測を MQTT トピックに発行する場合にのみ必要です。
- このコネクタを操作するには、AWS IoT Greengrass Machine Learning SDK v1.1.0 が必要です。

オブジェクト検出モデルの要件

ML オブジェクト検出コネクタは、Single Shot multibox Detector (SSD) および You Only Look Once (YOLO) v3 オブジェクト検出モデルタイプをサポートします。GluonCV が提供するオブジェクト検出コンポーネントを使用して、独自のデータセットでモデルをトレーニングできます。または、GluonCV Model Zoo から事前にトレーニングされたモデルを使用できます。

- トレーニング済みの SSD モデル
- トレーニング済みの YOLO v3 モデル

オブジェクト検出モデルは、512 x 512 の入力イメージを使用してトレーニングする必要があります。GluonCV Model Zoo の事前トレーニング済みモデルは、すでにこの要件を満たしています。

トレーニングされたオブジェクト検出モデルは、SageMaker AI Neo 深層学習コンパイラでコンパイルする必要があります。コンパイルするときは、ターゲットハードウェアが Greengrass コアデバイスのハードウェアと一致していることを確認します。詳細については、「Amazon SageMaker AI デベロッパーガイド」の「SageMaker AI Neo」を参照してください。 Amazon SageMaker

コンパイルされたモデルは、コネクタと同じ Greengrass グループに ML リソース (Amazon S3 モデルソース) として追加する必要があります。

コネクタパラメータ

これらのコネクタでは、以下のパラメータを使用できます。

`MLModelDestinationPath`

Neo 互換 ML モデルを含む Amazon S3 バケットへの絶対パス。これは、ML モデルリソースに指定されたターゲットパスです。

AWS IoT コンソールの表示名: モデル送信先パス

必須: `true`

型: `string`

有効なパターン: `.+`

`MLModelResourceId`

ソースモデルを参照する ML リソースの ID。

AWS IoT コンソールの表示名: Greengrass グループ ML リソース

必須: `true`

型: `S3MachineLearningModelResource`

有効なパターン: `^[a-zA-Z0-9:_-]+$`

`LocalInferenceServiceName`

ローカル推論サービスの名前。ユーザー定義の Lambda 関数は、 AWS IoT Greengrass Machine Learning SDK の `invoke_inference_service`関数に名前を渡すことでサービスを呼び出します。例については、the section called “使用例”を参照してください。

AWS IoT コンソールでの名前の表示: ローカル推論サービス名

必須: `true`

型: `string`

有効なパターン: `^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$`

`LocalInferenceServiceTimeoutSeconds`

推論リクエストが終了するまでの時間 (秒単位)。最小値は 1 です。デフォルト値は 10 です。

AWS IoT コンソールの表示名: タイムアウト (秒)

必須: `true`

型: `string`

有効なパターン: `^[1-9][0-9]*$`

`LocalInferenceServiceMemoryLimitKB`

サービスがアクセスできるメモリの量 (KB 単位)。最小値は 1 です。

AWS IoT コンソールの表示名: メモリ制限

必須: `true`

型: `string`

有効なパターン: `^[1-9][0-9]*$`

`GPUAcceleration`

CPU または GPU (アクセラレーション) コンピューティングの場合。このプロパティは ML イメージ分類 Aarch64 JTX2 コネクタにのみ適用されます。

AWS IoT コンソールの表示名: GPU アクセラレーション

必須: `true`

型: `string`

有効な値: `CPU` または `GPU`

`MLFeedbackConnectorConfigId`

モデル入力データのアップロードに使用するフィードバック設定の ID。これは、ML フィードバックコネクタに定義されたフィードバック設定の ID と一致する必要があります。

この パラメータは、ML フィードバックコネクタを使用してモデル入力データをアップロードし、予測を MQTT トピックに発行する場合にのみ必要です。

AWS IoT コンソールの表示名: ML フィードバックコネクタ設定 ID

必須: `false`

型: `string`

有効なパターン: `^$|^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、ML オブジェクト検出コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。この例では、ML オブジェクト検出 ARMv7l コネクタのインスタンスを作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyObjectDetectionConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
ObjectDetectionARMv7/versions/1",
            "Parameters": {
                "MLModelDestinationPath": "/path-to-model",
                "MLModelResourceId": "my-ml-resource",
                "LocalInferenceServiceName": "objectDetection",
                "LocalInferenceServiceTimeoutSeconds": "10",
                "LocalInferenceServiceMemoryLimitKB": "500000",
                "MLFeedbackConnectorConfigId" : "object-detector-random-sampling"
            }
        }
    ]
}'
```

> **Note**
>
> これらのコネクタの Lambda 関数には、存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

これらのコネクタは、イメージファイルを入力として受け入れます。入力イメージファイルは `jpeg` または `png` 形式である必要があります。詳細については、「the section called “使用例”」を参照してください。

これらのコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

これらのコネクタは、入力イメージで識別されたオブジェクトのフォーマットされた予測結果のリストを返します。

```
{
    "prediction": [
        [
            14,
            0.9384938478469849,
            0.37763649225234985,
            0.5110225081443787,
            0.6697432398796082,
            0.8544386029243469
        ],
        [
            14,
            0.8859519958496094,
            0,
            0.43536216020584106,
            0.3314110040664673,
            0.9538808465003967
        ],
        [
            12,
            0.04128098487854004,
            0.5976729989051819,
            0.5747185945510864,
            0.704264223575592,
            0.857937216758728
        ],
        ...
    ]
}
```

リストの各予測は角括弧で囲まれ、6 つの値が含まれます。

- 最初の値は、識別されたオブジェクトの予測されるオブジェクトカテゴリを表します。オブジェクトカテゴリと対応する値は、Neo 深層学習コンパイラでオブジェクト検出機械学習モデルをトレーニングするときに決定されます。
- 2 番目の値は、オブジェクトカテゴリ予測の信頼スコアです。これは、予測が正しい確率を表します。

- 最後の 4 つの値は、イメージの予測オブジェクト周囲の境界ボックスを表すピクセルディメンションに対応しています。

これらのコネクタは MQTT メッセージを出力データとして公開しません。

使用例

次の Lambda 関数の例は、AWS IoT Greengrass Machine Learning SDK を使用して、ML オブジェクト検出コネクタと対話します。

> **Note**
>
> この SDK は、AWS IoT Greengrass Machine Learning SDK のダウンロードページからダウンロードできます。

この例では、SDK クライアントを初期化し、SDK の `invoke_inference_service` 関数の同期呼び出しにより、ローカル推論サービスを呼び出します。次に、アルゴリズムタイプ、サービス名、イメージタイプ、イメージコンテンツを渡します。その後、サービスのレスポンスを解析して、確率の結果 (予測) を取得します。

```
import logging
from threading import Timer

import numpy as np

import greengrass_machine_learning_sdk as ml

# We assume the inference input image is provided as a local file
# to this inference client Lambda function.
with open('/test_img/test.jpg', 'rb') as f:
    content = bytearray(f.read())

client = ml.client('inference')

def infer():
    logging.info('invoking Greengrass ML Inference service')

    try:
        resp = client.invoke_inference_service(
            AlgoType='object-detection',
```

```
            ServiceName='objectDetection',
            ContentType='image/jpeg',
            Body=content
        )
    except ml.GreengrassInferenceException as e:
        logging.info('inference exception {}("{}")'.format(e.__class__.__name__, e))
        return
    except ml.GreengrassDependencyException as e:
        logging.info('dependency exception {}("{}")'.format(e.__class__.__name__, e))
        return

    logging.info('resp: {}'.format(resp))
    predictions = resp['Body'].read().decode("utf-8")
    logging.info('predictions: {}'.format(predictions))
    predictions = eval(predictions)

    # Perform business logic that relies on the predictions.

    # Schedule the infer() function to run again in ten second.
    Timer(10, infer).start()
    return

infer()

def function_handler(event, context):
    return
```

Machine AWS IoT Greengrass Machine Learning SDK の `invoke_inference_service`関数は、次の引数を受け入れます。

引数	説明
`AlgoType`	推論に使用するアルゴリズムタイプの名前。現在は、`object-detection` のみがサポートされます。 必須: `true` 型: `string` 有効な値: `object-detection`

引数	説明
`ServiceName`	ローカル推論サービスの名前。コネクタを設定したときに `LocalInferenceServiceName` パラメータに指定した名前を使用します。 必須: `true` 型: `string`
`ContentType`	入力イメージの MIME タイプ。 必須: `true` 型: `string` 有効な値: `image/jpeg, image/png`
`Body`	入力イメージファイルの内容。 必須: `true` 型: `binary`

AWS IoT Greengrass コアへの Neo 深層学習ランタイム依存関係のインストール

ML オブジェクト検出コネクタは、SageMaker AI Neo 深層学習ランタイム (DLR) にバンドルされています。コネクタは、ランタイムを使用して ML モデルを提供します。これらのコネクタを使用するには、コアデバイスに DLR の依存関係をインストールする必要があります。

DLR の依存関係をインストールする前に、必要なシステムライブラリ (指定された最小バージョン) がデバイスに存在することを確認してください。

NVIDIA Jetson TX2

1. CUDA Toolkit 9.0 と cuDNN 7.0 をインストールします。開始方法チュートリアルの「the section called “他のデバイスの設定”」の手順に従うことができます。

2. コネクタでコミュニティ管理のオープンなソフトウェアをインストールできるように、ユニバースリポジトリを有効にします。詳細については、Ubuntu ドキュメントの Repositories/Ubuntu を参照してください。

   a. `/etc/apt/sources.list` ファイルを開きます。

   b. 以下の行のコメントが解除されていることを確認してください。

```
deb http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
```

3. 以下のインストールスクリプトのコピーを Core デバイス上の `nvidiajtx2.sh` というファイルに保存します。

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libatlas-base-dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV
 with pip on this platform. Try building the latest OpenCV from source (https://
github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

4. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo nvidiajtx2.sh
```

x86_64 (Ubuntu or Amazon Linux)

1. 以下のインストールスクリプトのコピーを Core デバイス上の `x86_64.sh` というファイルに保存します。

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)

if [ "$release" == '"Ubuntu"' ]; then
  # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies
 installed, so
  # this is mostly to prepare dependencies on Ubuntu EC2 instance.
  apt-get -y update
  apt-get -y dist-upgrade

  apt-get install -y libgfortran3 libsm6 libxext6 libxrender1
  apt-get install -y python3.7 python3.7-dev
elif [ "$release" == '"Amazon Linux"' ]; then
  # Amazon Linux. Expect python to be installed already
  yum -y update
  yum -y upgrade

  yum install -y compat-gcc-48-libgfortran libSM libXrender libXext
else
  echo "OS Release not supported: $release"
  exit 1
fi

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV
 with pip on this platform. Try building the latest OpenCV from source (https://
github.com/opencv/opencv).'
```

```
echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

2. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo x86_64.sh
```

ARMv7 (Raspberry Pi)

1. 以下のインストールスクリプトのコピーを Core デバイス上の `armv7l.sh` というファイルに保存します。

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

apt-get update
apt-get -y upgrade

apt-get install -y liblapack3 libopenblas-dev liblapack-dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV
 with pip on this platform. Try building the latest OpenCV from source (https://
github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

> **Note**
>
> OpenCV がこのスクリプトを使用して正常にインストールしない場合、ソースからビルドしてみることができます。詳細については、OpenCV ドキュメントの「Linux でのインストール」、またはお使いのプラットフォーム用の他のオンラインリソースを参照してください。

2. ファイルを保存したディレクトリから、次のコマンドを実行します。

```
sudo bash armv7l.sh
```

> **Note**
>
> Raspberry Pi では、`pip` を使用して機械学習の依存関係をインストールすると、メモリを大量に消費し、デバイスがメモリ不足になって応答しなくなる可能性があります。回避策として、スワップサイズを一時的に増やすことができます。`/etc/dphys-swapfile` で、`CONF_SWAPSIZE` 変数の値を増やし、次のコマンドを実行して `dphys-swapfile` を再起動します。
>
> ```
> /etc/init.d/dphys-swapfile restart
> ```

ログ記録とトラブルシューティング

グループ設定に応じて、イベントログとエラーログは、CloudWatch ログ、ローカルファイルシステム、またはその両方に書き込まれます。このコネクタのログにはプレフィックス `LocalInferenceServiceName` が使用されます。コネクタが予期しない動作を示した場合は、コネクタのログを確認します。このログには、通常、ML ライブラリの依存関係の不足やコネクタの起動失敗の原因など、デバッグに役立つ情報が含まれています。

AWS IoT Greengrass グループがローカルログを書き込むように設定されている場合、コネクタはログファイルを に書き込みます*`greengrass-root`*`/ggc/var/log/user/`*`region`*`/aws/`。Greengrass の ログ記録の詳細については、「the section called “ AWS IoT Greengrass ログでのモニタリング”」を参照してください。

ML オブジェクト検出コネクタの問題のトラブルシューティングには、以下の情報が役立ちます。

必須のシステムライブラリ

以下の各タブは、ML オブジェクト検出コネクタごとに必要なシステムライブラリを一覧表示します。

ML Object Detection Aarch64 JTX2

ライブラリ	最小バージョン
ld-linux-aarch64.so.1	GLIBC_2.17
libc.so.6	GLIBC_2.17
libcublas.so.9.0	該当なし
libcudart.so.9.0	該当なし
libcudnn.so.7	該当なし
libcufft.so.9.0	該当なし
libcurand.so.9.0	該当なし
libcusolver.so.9.0	該当なし
libgcc_s.so.1	GCC_4.2.0
libgomp.so.1	GOMP_4.0、OMP_1.0
libm.so.6	GLIBC_2.23
libnvinfer.so.4	該当なし
libnvrm_gpu.so	該当なし
libnvrm.so	該当なし
libnvidia-fatbinaryloader.so.28.2.1	該当なし
libnvos.so	該当なし
libpthread.so.0	GLIBC_2.17

ライブラリ	最小バージョン
librt.so.1	GLIBC_2.17
libstdc++.so.6	GLIBCXX_3.4.21、CXXABI_1.3.8

ML Object Detection x86_64

ライブラリ	最小バージョン
ld-linux-x86-64.so.2	GCC_4.0.0
libc.so.6	GLIBC_2.4
libgfortran.so.3	GFORTRAN_1.0
libm.so.6	GLIBC_2.23
libpthread.so.0	GLIBC_2.2.5
librt.so.1	GLIBC_2.2.5
libstdc++.so.6	CXXABI_1.3.8、GLIBCXX_3.4.21

ML Object Detection ARMv7

ライブラリ	最小バージョン
ld-linux-armhf.so.3	GLIBC_2.4
libc.so.6	GLIBC_2.7
libgcc_s.so.1	GCC_4.0.0
libgfortran.so.3	GFORTRAN_1.0
libm.so.6	GLIBC_2.4
libpthread.so.0	GLIBC_2.4

ライブラリ	最小バージョン
librt.so.1	GLIBC_2.4
libstdc++.so.6	CXXABI_1.3.8、CXXABI_ARM_1.3.3、GLIBCXX_3.4.20

問題点

症状	ソリューション
Raspberry Pi で、次のエラーメッセージがログに記録される。カメラは使用していない。`Failed to initialize libdc1394`	次のコマンドを実行してドライバーを無効にします。 `sudo ln /dev/null /dev/raw1394` このオペレーションはエフェメラルです。再起動すると、シンボリックリンクが消えます。再起動時にリンクを自動的に作成する方法については、OS ディストリビューションのマニュアルを参照してください。

ライセンス

ML オブジェクト検出コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

- 深層学習ランタイム/Apache License 2.0
- six/MIT

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- 機械学習の推論を実行する
- Amazon SageMaker AI デベロッパーガイドのオブジェクト検出アルゴリズム

Modbus-RTU プロトコルアダプタコネクタ

Modbus-RTU プロトコルアダプタコネクタは、 AWS IoT Greengrass グループ内の Modbus RTU デバイスから情報をポーリングします。

このコネクタは、ユーザー定義の Lambda 関数からの Modbus RTU リクエストのパラメータを受け取ります。その後、対応するリクエストを送信し、MQTT メッセージとしてターゲットデバイスからのレスポンスを発行します。

このコネクタには、次のバージョンがあります。

バージョン	ARN
3	`arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/3`
2	`arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/2`
1	`arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- AWS IoT Greengrass コアと Modbus デバイス間の物理的な接続。Core はシリアルポート (USB ポートなど) を介して Modbus RTU ネットワークに物理的に接続する必要があります。
- 物理 Modbus シリアルポートを参照する、Greengrass グループのローカルデバイスリソース。
- このコネクタに Modbus RTU リクエストパラメータを送信するユーザー定義の Lambda 関数。リクエストパラメータは、想定されるパターンに従い、Modbus RTU ネットワーク上のターゲットデバイスの ID とアドレスを含む必要があります。詳細については、「the section called “入力データ”」を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

- AWS IoT Greengrass コアと Modbus デバイス間の物理的な接続。Core はシリアルポート (USB ポートなど) を介して Modbus RTU ネットワークに物理的に接続する必要があります。
- 物理 Modbus シリアルポートを参照する、Greengrass グループのローカルデバイスリソース。
- このコネクタに Modbus RTU リクエストパラメータを送信するユーザー定義の Lambda 関数。リクエストパラメータは、想定されるパターンに従い、Modbus RTU ネットワーク上のターゲットデバイスの ID とアドレスを含む必要があります。詳細については、「the section called “入力データ”」を参照してください。

コネクタパラメータ

このコネクタでは、以下のパラメータがサポートされています。

`ModbusSerialPort-ResourceId`

物理 Modbus シリアルポートを表すローカルデバイスリソースの ID。

> **Note**
> このコネクタには、リソースへの読み取りと書き込みアクセスが付与されています。

AWS IoT コンソールの表示名: Modbus シリアルポートリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`ModbusSerialPort`

デバイス上の物理 Modbus シリアルポートへの絶対パス。これは、Modbus ローカルデバイスリソース用に指定されたソースパスです。

AWS IoT コンソールの表示名: Modbus シリアルポートリソースのソースパス

必須: `true`

型: `string`

有効なパターン: `.+`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Modbus-RTU プロトコルアダプタコネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyModbusRTUProtocolAdapterConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
ModbusRTUProtocolAdapter/versions/3",
            "Parameters": {
                "ModbusSerialPort-ResourceId": "MyLocalModbusSerialPort",
                "ModbusSerialPort": "/path-to-port"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

> **Note**
>
> Modbus-RTU Protocol Adapter コネクタをデプロイした後、 AWS IoT Things Graph を使用してグループ内のデバイス間のインタラクションをオーケストレーションできます。詳細については、「AWS IoT Things Graph ユーザーガイド」の「Modbus」を参照してください。

入力データ

このコネクタは、ユーザー定義の Lambda 関数から MQTT トピックに対する Modbus RTU リクエストのパラメータを受け取ります。入力メッセージは JSON 形式である必要があります。

サブスクリプションのトピックフィルター

```
modbus/adapter/request
```

メッセージのプロパティ

リクエストメッセージは、Modbus RTU リクエストのタイプによって異なります。以下のプロパティはすべてのリクエストに必須です。

- `request` オブジェクト:
  - `operation`。実行するオペレーションの名前。例えば、コイルを読み取るには `"operation": "ReadCoilsRequest"` を指定します。この値は Unicode 文字列であることが必要です。サポートされているオペレーションについては、「the section called “Modbus RTU のリクエストとレスポンス”」を参照してください。
  - `device`。リクエストのターゲットデバイス。これは `0 - 247` のいずれかの値であることが必要です。
- `id` プロパティ。リクエストの ID。この値は、データ重複除外に使用され、エラーレスポンスを含むすべてのレスポンスの `id` プロパティでそのまま返されます。この値は Unicode 文字列であることが必要です。

> **Note**
>
> リクエストにアドレスフィールドが含まれている場合は、値を整数で指定する必要があります。例えば、`"address": 1` と指定します。

リクエストに含まれるその他のパラメータはオペレーションによって異なります。個別に処理される CRC を除くすべてのリクエストパラメータは必須です。例については「the section called “リクエストとレスポンスの例”」を参照してください。

入力例: コイルの読み取りリクエスト

```
{
    "request": {
        "operation": "ReadCoilsRequest",
     "device": 1,
     "address": 1,
     "count": 1
    },
    "id": "TestRequest"
}
```

出力データ

このコネクタは、受信 Modbus RTU リクエストへのレスポンスを発行します。

サブスクリプションのトピックフィルター

```
modbus/adapter/response
```

メッセージのプロパティ

レスポンスメッセージの形式は、対応するリクエストとレスポンスステータスによって異なります。例については「the section called “リクエストとレスポンスの例”」を参照してください。

> **Note**
>
> 書き込みオペレーションのレスポンスはリクエストのエコーのみです。書き込みレスポンスで意味のある情報は返されませんが、レスポンスのステータスを確認することをお勧めします。

レスポンスごとに、以下のプロパティが含まれます。

- `response` オブジェクト:
  - `status`。リクエストのステータス。ステータスは、次のいずれかの値になります。
    - `Success`。リクエストは有効で、Modbus RTU ネットワークに送信され、レスポンスが返されました。
    - `Exception`。リクエストは有効で、Modbus RTU ネットワークに送信され、例外レスポンスが返されました。詳細については、「the section called “レスポンスステータス: 例外”」を参照してください。
    - `No Response`。リクエストは無効で、リクエストが Modbus RTU ネットワークを介して送信される前にコネクタがエラーを検出しました。詳細については、「the section called “レスポンスステータス: No Response”」を参照してください。
  - `device`。リクエストが送信されたデバイス。
  - `operation`。送信されたリクエストのタイプ。
  - `payload`。返されたレスポンスの内容。`status` が `No Response` の場合、このオブジェクトには `error` プロパティがエラーの説明 (`"error": "[Input/Output] No Response received from the remote unit"` など) と共に含まれています。
- `id` プロパティ。データ重複削除に使用されるリクエストの ID。

出力例: 成功

```
{
    "response" : {
        "status" : "success",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "function_code": 1,
         "bits": [1]
     }
     },
     "id" : "TestRequest"
}
```

出力例: 失敗

```
{
    "response" : {
        "status" : "fail",
        "error_message": "Internal Error",
        "error": "Exception",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "function_code": 129,
         "exception_code": 2
     }
     },
     "id" : "TestRequest"
}
```

その他の例については、「the section called “リクエストとレスポンスの例”」を参照してください。

Modbus RTU のリクエストとレスポンス

このコネクタは、Modbus RTU のリクエストパラメータを入力データとして受け取り、レスポンスを出力データとして発行します。

以下の一般的なオペレーションがサポートされています。

リクエストのオペレーション名	レスポンスの関数コード
ReadCoilsRequest	01
ReadDiscreteInputsRequest	02
ReadHoldingRegistersRequest	03
ReadInputRegistersRequest	04
WriteSingleCoilRequest	05
WriteSingleRegisterRequest	06
WriteMultipleCoilsRequest	15
WriteMultipleRegistersRequest	16
MaskWriteRegisterRequest	22
ReadWriteMultipleRegistersRequest	23

リクエストとレスポンスの例

以下に示しているのは、サポートされているオペレーションのリクエストとレスポンスの例です。

コイルの読み取り

リクエストの例:

```
{
    "request": {
        "operation": "ReadCoilsRequest",
     "device": 1,
     "address": 1,
     "count": 1
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "function_code": 1,
         "bits": [1]
     }
     },
     "id" : "TestRequest"
}
```

個別入力の読み取り

リクエストの例:

```
{
    "request": {
        "operation": "ReadDiscreteInputsRequest",
        "device": 1,
        "address": 1,
        "count": 1
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "ReadDiscreteInputsRequest",
        "payload": {
            "function_code": 2,
            "bits": [1]
        }
     },
     "id" : "TestRequest"
}
```

保持レジスタの読み取り

リクエストの例:

```
{
    "request": {
        "operation": "ReadHoldingRegistersRequest",
     "device": 1,
     "address": 1,
     "count": 1
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "ReadHoldingRegistersRequest",
     "payload": {
         "function_code": 3,
            "registers": [20,30]
     }
     },
     "id" : "TestRequest"
}
```

入力レジスタの読み取り

リクエストの例:

```
{
    "request": {
        "operation": "ReadInputRegistersRequest",
     "device": 1,
     "address": 1,
     "value": 1
    },
    "id": "TestRequest"
}
```

単一コイルの書き込み

リクエストの例:

```
{
    "request": {
        "operation": "WriteSingleCoilRequest",
     "device": 1,
     "address": 1,
     "value": 1
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "WriteSingleCoilRequest",
     "payload": {
         "function_code": 5,
         "address": 1,
         "value": true
     }
     },
     "id" : "TestRequest"
```

単一レジスタの書き込み

リクエストの例:

```
{
    "request": {
        "operation": "WriteSingleRegisterRequest",
     "device": 1,
     "address": 1,
     "value": 1
    },
    "id": "TestRequest"
}
```

複数コイルの書き込み

リクエストの例:

```
{
    "request": {
        "operation": "WriteMultipleCoilsRequest",
     "device": 1,
     "address": 1,
     "values": [1,0,0,1]
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "WriteMultipleCoilsRequest",
     "payload": {
         "function_code": 15,
         "address": 1,
         "count": 4
     }
     },
     "id" : "TestRequest"
}
```

複数レジスタの書き込み

リクエストの例:

```
{
    "request": {
        "operation": "WriteMultipleRegistersRequest",
     "device": 1,
     "address": 1,
     "values": [20,30,10]
    },
    "id": "TestRequest"
```

```
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "WriteMultipleRegistersRequest",
     "payload": {
         "function_code": 23,
         "address": 1,
         "count": 3
     }
     },
     "id" : "TestRequest"
}
```

書き込みレジスタのマスク

リクエストの例:

```
{
    "request": {
        "operation": "MaskWriteRegisterRequest",
     "device": 1,
     "address": 1,
        "and_mask": 175,
        "or_mask": 1
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "MaskWriteRegisterRequest",
     "payload": {
         "function_code": 22,
```

```
            "and_mask": 0,
            "or_mask": 8
     }
     },
     "id" : "TestRequest"
}
```

複数レジスタの読み書き

リクエストの例:

```
{
    "request": {
        "operation": "ReadWriteMultipleRegistersRequest",
     "device": 1,
     "read_address": 1,
        "read_count": 2,
        "write_address": 3,
        "write_registers": [20,30,40]
    },
    "id": "TestRequest"
}
```

レスポンスの例:

```
{
    "response": {
        "status": "success",
        "device": 1,
     "operation": "ReadWriteMultipleRegistersRequest",
     "payload": {
         "function_code": 23,
         "registers": [10,20,10,20]
     }
     },
     "id" : "TestRequest"
}
```

> **Note**
>
> このレスポンスで返されるレジスタは読み取り元のレジスタです。

レスポンスステータス: 例外

リクエスト書式が有効で、リクエストが正常に完了していない場合、例外が発生している可能性があります。この場合、レスポンスには以下の情報が含まれています。

- `status` は `Exception` に設定されています。
- `function_code` がリクエストの関数コード + 128 に等しい。
- `exception_code` に例外コードが含まれている。詳細については、Modbus 例外コードを参照してください。

例:

```
{
    "response" : {
        "status" : "fail",
        "error_message": "Internal Error",
        "error": "Exception",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "function_code": 129,
         "exception_code": 2
     }
     },
     "id" : "TestRequest"
}
```

レスポンスステータス: No Response

このコネクタは Modbus リクエストの検証チェックを実行します。例えば、無効な書式や不足しているフィールドがないかどうかを確認します。検証に失敗すると、コネクタはリクエストを送信しません。代わりに、以下の情報を含むレスポンスを返します。

- `status` は `No Response` に設定されています。
- `error` にはエラーの理由が含まれています。
- `error_message` にはエラーメッセージが含まれています。

例:

```
{
    "response" : {
        "status" : "fail",
        "error_message": "Invalid address field. Expected <type 'int'>, got <type
 'str'>",
        "error": "No Response",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "error": "Invalid address field. Expected <type 'int'>, got <type 'str'>"
     }
     },
     "id" : "TestRequest"
}
```

リクエストのターゲットデバイスが存在しない場合、または Modbus RTU ネットワークが機能していない場合、No Response 形式を使用する `ModbusIOException` が返されます。

```
{
    "response" : {
        "status" : "fail",
        "error_message": "[Input/Output] No Response received from the remote unit",
        "error": "No Response",
        "device": 1,
     "operation": "ReadCoilsRequest",
     "payload": {
         "error": "[Input/Output] No Response received from the remote unit"
     }
     },
     "id" : "TestRequest"
}
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。

- 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。
3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

   b. 必要なローカルデバイスリソースを追加し、Lambda 関数への読み取り/書き込みアクセスを許可します。

   c. コネクタを追加し、そのパラメータを設定します。

   d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。
4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import json

TOPIC_REQUEST = 'modbus/adapter/request'

# Creating a greengrass core sdk client
iot_client = greengrasssdk.client('iot-data')

def create_read_coils_request():
 request = {
  "request": {
   "operation": "ReadCoilsRequest",
   "device": 1,
   "address": 1,
   "count": 1
  },
  "id": "TestRequest"
 }
 return request

def publish_basic_request():
 iot_client.publish(payload=json.dumps(create_read_coils_request()),
 topic=TOPIC_REQUEST)

publish_basic_request()

def lambda_handler(event, context):
 return
```

ライセンス

Modbus-RTU プロトコルアダプタコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- pymodbus/BSD
- pyserial/BSD

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	AWS リージョン サポート用にコネクタ ARN を更新しました。 エラーログ記録を改良しました。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

Modbus-TCP プロトコルアダプターコネクタ

Modbus-TCP プロトコルアダプタコネクタは、ModbusTCP プロトコルを介してローカルデバイスからデータを収集し、選択した `StreamManager` ストリームに発行します。

このコネクタは、IoT SiteWise コネクタや IoT SiteWise ゲートウェイと共に使用することもできます。ゲートウェイは、コネクタの設定を提供する必要があります。詳細については、「IoT SiteWise ユーザーガイド」の「Modbus TCP ソースを設定する」を参照してください。

> **Note**
>
> このコネクタはコンテナ分離なしモードで実行されるため、Docker コンテナで実行されている AWS IoT Greengrass グループにデプロイできます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
3	`arn:aws:greengrass:region::/connectors/ModbusTCPConnector/versions/3`
2	`arn:aws:greengrass:region::/connectors/ModbusTCPConnector/versions/2`
1	`arn:aws:greengrass:region::/connectors/ModbusTCPConnector/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 1 - 3

- AWS IoT Greengrass Core ソフトウェア v1.10.2 以降。
- AWS IoT Greengrass グループで有効になっているストリームマネージャー。
- Java 8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> このコネクタは、次のリージョンでのみ利用できます。
>
> - ap-southeast-1
> - ap-southeast-2
> - eu-central-1
> - eu-west-1
> - us-east-1
> - us-west-2
> - cn-north-1

コネクタパラメータ

このコネクタでは、以下のパラメータがサポートされています。

`LocalStoragePath`

IoT SiteWise コネクタが永続データを書き込むことができる AWS IoT Greengrass ホスト上のディレクトリ。デフォルトのディレクトリは `/var/sitewise` です。

AWS IoT コンソールでの名前の表示: ローカルストレージパス

必須: `false`

型: `string`

有効なパターン: `^\s*$|\/.`

`MaximumBufferSize`

IoT SiteWise ディスク使用量の最大サイズ (GB)。デフォルトサイズは 10 GB です。

AWS IoT コンソールの表示名: 最大ディスクバッファサイズ

必須: `false`

型: `string`

有効なパターン: `^\s*$|[0-9]+`

`CapabilityConfiguration`

コネクタがデータを収集し、接続する一連の Modbus TCP コレクタ構成。

AWS IoT コンソールの表示名: CapabilityConfiguration

必須: `false`

タイプ: サポートされているフィードバック設定のセットを定義する正しい形式の JSON 文字列。

次は、`CapabilityConfiguration` の例です。

```
{
    "sources": [
        {
            "type": "ModBusTCPSource",
            "name": "SourceName1",
            "measurementDataStreamPrefix": "SourceName1_Prefix",
            "destination": {
                "type": "StreamManager",
                "streamName": "SiteWise_Stream_1",
                "streamBufferSize": 8
            },
            "endpoint": {
                "ipAddress": "127.0.0.1",
                "port": 8081,
                "unitId": 1
            },
            "propertyGroups": [
                {
                    "name": "GroupName",
                    "tagPathDefinitions": [
                        {
                            "type": "ModBusTCPAddress",
                            "tag": "TT-001",
                            "address": "30001",
                            "size": 2,
                            "srcDataType": "float",
                            "transformation": "byteWordSwap",
                            "dstDataType": "double"
```

```
                        }
                    ],
                    "scanMode": {
                        "type": "POLL",
                        "rate": 100
                    }
                }
            ]
        }
    ]
}
```

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Modbus-TCP プロトコルアダプタコネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '
{
    "Connectors": [
        {
            "Id": "MyModbusTCPConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/ModbusTCP/
versions/3",
            "Parameters": {
                "capability_configuration": "{\"version\":1,\"namespace\":
\"iotsitewise:modbuscollector:1\",\"configuration\":\"{\"sources\":[{\"type
\":\"ModBusTCPSource\",\"name\":\"SourceName1\",\"measurementDataStreamPrefix
\":\"\",\"endpoint\":{\"ipAddress\":\"127.0.0.1\",\"port\":8081,\"unitId\":1},
\"propertyGroups\":[{\"name\":\"PropertyGroupName\",\"tagPathDefinitions\":[{\"type
\":\"ModBusTCPAddress\",\"tag\":\"TT-001\",\"address\":\"30001\",\"size\":2,
\"srcDataType\":\"hexdump\",\"transformation\":\"noSwap\",\"dstDataType\":\"string
\"}],\"scanMode\":{\"rate\":200,\"type\":\"POLL\"}}],\"destination\":{\"type\":
\"StreamManager\",\"streamName\":\"SiteWise_Stream\",\"streamBufferSize\":10},
\"minimumInterRequestDuration\":200}]}\"}"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

入力データ

このコネクタは MQTT メッセージを入力データとして受け入れません。

出力データ

このコネクタは、`StreamManager` にデータを発行します。宛先メッセージストリームを設定する必要があります。出力メッセージは以下の構造になります。

```
{
    "alias": "string",
    "messages": [
        {
            "name": "string",
            "value": boolean|double|integer|string,
            "timestamp": number,
            "quality": "string"
        }
    ]
}
```

ライセンス

Modbus-TCP プロトコルアダプタコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- デジタルペトリ Modbus

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更	日付
3 (推奨)	このバージョンには、バグ修正が含まれています。	2021 年 12 月 22 日
2	ASCII、UTF8、および ISO8859 エンコードされたソース文字列のサポートが追加されました。	2021 年 5 月 24 日
1	初回リリース。	2020 年 12 月 15 日

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

Raspberry Pi GPIO コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

Raspberry Pi GPIO コネクタは、Raspberry Pi コアデバイスの汎用入力/出力 (GPIO) ピンをコントロールします。

このコネクタは、指定された間隔で入力ピンをポーリングし、状態の変化を MQTT トピックに発行します。また、ユーザー定義の Lambda 関数から MQTT メッセージとして読み取りおよび書き込み

リクエストを受け入れます。書き込みリクエストは、ピンを高電圧または低電圧に設定するために使用します。

コネクタには、入力ピンと出力ピンを指定するためのパラメータが用意されています。この動作は、グループデプロイの前に設定します。実行時に変更することはできません。

- 入力ピンを使用して、周辺デバイスからデータを受信できます。
- 出力ピンを使用して周辺機器をコントロールしたり、周辺機器にデータを送信したりできます。

このコネクタは、以下のような多くのシナリオで使用できます。

- 信号灯の緑色、黄色、赤色の LED ライトをコントロールする。
- 湿度センサーからのデータに基づいてファン (電気リレーに取り付けられている) をコントロールする。
- 顧客がボタンを押したときに小売店の従業員に通知する。
- スマートライトスイッチを使用して他の IoT デバイスをコントロールする。

> **Note**
>
> このコネクタは、リアルタイム要件がある用途には適していません。期間が短いイベントは見逃される可能性があります。

このコネクタには、次のバージョンがあります。

バージョン	ARN
3	`arn:aws:greengrass:  region::/connectors/RaspberryPiGPIO/versions/3`
2	`arn:aws:greengrass:  region::/connectors/RaspberryPiGPIO/versions/2`

バージョン	ARN
1	`arn:aws:greengrass:  region::/connectors/RaspberryPiGPIO/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- Raspberry Pi 4 モデル B、または Raspberry Pi 3 モデル B/B+。Raspberry Pi のピン配列を知る必要があります。詳細については、「the section called “GPIO ピンシーケンス”」を参照してください。
- Raspberry Pi の `/dev/gpiomem` を参照する Greengrass グループのローカルデバイスリソース。コンソールでリソースを作成する場合、[Automatically add OS group permissions of the Linux group that owns the resource] (リソースを所有する Linux グループの OS グループアクセス権限を自動的に追加する) オプションを選択する必要があります。API で、`GroupOwnerSetting.AutoAddGroupOwner` プロパティを `true` に設定します。
- Raspberry Pi にインストールされた RPi.GPIO モジュール。Raspbian では、このモジュールがデフォルトでインストールされています。以下のコマンドを使用して再インストールできます。

  ```
  sudo pip install RPi.GPIO
  ```

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

- Raspberry Pi 4 モデル B、または Raspberry Pi 3 モデル B/B+。Raspberry Pi のピン配列を知る必要があります。詳細については、「the section called “GPIO ピンシーケンス”」を参照してください。
- Raspberry Pi の `/dev/gpiomem` を参照する Greengrass グループのローカルデバイスリソース。コンソールでリソースを作成する場合、[Automatically add OS group permissions of the Linux group that owns the resource] (リソースを所有する Linux グループの OS グループアクセス権限を自動的に追加する) オプションを選択する必要があります。API で、`GroupOwnerSetting.AutoAddGroupOwner` プロパティを `true` に設定します。
- Raspberry Pi にインストールされた RPi.GPIO モジュール。Raspbian では、このモジュールがデフォルトでインストールされています。以下のコマンドを使用して再インストールできます。

  ```
  sudo pip install RPi.GPIO
  ```

GPIO ピンシーケンス

Raspberry Pi GPIO コネクタは、GPIO ピンの物理的なレイアウトではなく、基礎となるシステムオンチップ (SoC) の番号付け方法によって GPIO ピンを参照します。Raspberry Pi のバージョンでは、ピンの物理的な順序が異なる場合があります。詳細については、Raspberry Pi ドキュメントの「GPIO」を参照してください。

コネクタは、設定されている入力ピンと出力ピンが、Raspberry Pi の基本ハードウェアに正しくマッピングされていることを検証できません。ピン設定が無効な場合、コネクタは、デバイスでの起動試行時に、ランタイムエラーを返します。この問題を解決するには、コネクタを再設定してから再デプロイします。

> **Note**
> コンポーネントの損傷を防ぐために、GPIO ピンの周辺機器が適切に配線されていることを確認してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

`InputGpios`

入力として設定する、カンマで区切られた GPIO ピン番号のリスト。オプションで、ピンのプルアップ抵抗を設定するには `U` を付加し、プルダウン抵抗を設定するには `D` を付加します。例えば、`"5,6U,7D"` などです。

AWS IoT コンソールでの表示名: GPIO ピンの入力

必須: `false`。入力 ピン、出力 ピン、またはその両方を指定する必要があります。

型: `string`

有効なパターン: `^$|^[0-9]+[UD]?(,[0-9]+[UD]?)*$`

`InputPollPeriod`

入力 GPIO ピンの状態の変化を確認する各ポーリングオペレーションの間隔 (ミリ秒単位)。最小値は 1 です。

この値は、シナリオとポーリングされるデバイスのタイプによって異なります。例えば、値が `50` であれば、ボタンの押下を検出するのに十分な速さになります。

AWS IoT コンソールの表示名: 入力 GPIO ポーリング期間

必須: `false`

型: `string`

有効なパターン: `^$|^[1-9][0-9]*$`

`OutputGpios`

出力として設定する、カンマで区切られた GPIO ピン番号のリスト。オプションで、高電圧状態 (1) を設定するには `H` を付加し、低電圧状態 (0) を設定するには `L` を付加します。例えば、`"8H,9,27L"` などです。

AWS IoT コンソールの表示名: GPIO ピンを出力する

必須: `false`。入力 ピン、出力 ピン、またはその両方を指定する必要があります。

型: `string`

有効なパターン: `^$|^[0-9]+[HL]?(,[0-9]+[HL]?)*$`

`GpioMem-ResourceId`

`/dev/gpiomem` を表すローカルデバイスリソースの ID。

> **Note**
> このコネクタには、リソースへの読み取りと書き込みアクセスが付与されています。

AWS IoT コンソールの表示名: /dev/gpiomem デバイスのリソース

必須: `true`

型: `string`

有効なパターン: `.+`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Raspberry Pi GPIO コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyRaspberryPiGPIOConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/RaspberryPiGPIO/
versions/3",
            "Parameters": {
                "GpioMem-ResourceId": "my-gpio-resource",
                "InputGpios": "5,6U,7D",
                "InputPollPeriod": 50,
                "OutputGpios": "8H,9,27L"
            }
        }
    ]
}'
```

> **Note**
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、2 つの MQTT トピックの GPIO ピンに対する読み取りリクエストまたは書き込みリクエストを受け入れます。

- `gpio/+/+/read` トピックの読み取りリクエスト。
- `gpio/+/+/write` トピックの書き込みリクエスト。

これらのトピックに発行するには、+ ワイルドカードをそれぞれ Core モノ名とターゲットピン番号に置き換えます。例:

```
gpio/core-thing-name/gpio-number/read
```

> **Note**
>
> 現時点では、Raspberry Pi GPIO コネクタを使用するサブスクリプションを作成するときに、トピックの少なくとも 1 つの + ワイルドカードに値を指定する必要があります。

トピックのフィルター: `gpio/+/+/read`

このトピックを使用して、トピックで指定されている GPIO ピンの状態を読み取るようトピックに指示します。

コネクタは、対応する出力トピック (`gpio/core-thing-name/gpio-number/state` など) にレスポンスを発行します。

メッセージのプロパティ

なし。このトピックに送信されるメッセージは無視されます。

トピックのフィルター: `gpio/+/+/write`

このトピックを使用して、GPIO ピンに書き込みリクエストを送信します。これは、トピックで指定されている GPIO ピンを低電圧または高電圧に設定するようコネクタに指示します。

- `0` はピンを低電圧に設定します。

- 1 はピンを高電圧に設定します。

コネクタは、対応する出力 /state トピック (gpio/*core-thing-name*/*gpio-number*/state など) にレスポンスを発行します。

メッセージのプロパティ

整数または文字列の値 0 または 1。

入力例

```
0
```

出力データ

このコネクタは、以下の 2 つのトピックにデータを発行します。

- gpio/+/+/state トピックでの高電圧または低電圧への状態の変化。
- gpio/+/error トピックでのエラー。

トピックのフィルター: gpio/+/+/state

このトピックを使用して、入力ピンの状態の変化と読み取りリクエストへのレスポンスをリッスンします。コネクタは、ピンが低電圧状態の場合は文字列 "0" を返し、高電圧状態の場合は文字列 "1" を返します。

このトピックに公開するときに、コネクタはワイルドカード + をそれぞれ Core モノ名とターゲット PIN に置き換えます。例:

```
gpio/core-thing-name/gpio-number/state
```

> **Note**
> 現時点では、Raspberry Pi GPIO コネクタを使用するサブスクリプションを作成するときに、トピックの少なくとも 1 つの + ワイルドカードに値を指定する必要があります。

出力例

```
0
```

トピックのフィルター: `gpio/+/error`

このトピックを使用して、エラーをリッスンします。コネクタは、無効なリクエスト (入力ピンでの状態の変更のリクエストなど) の結果としてこのトピックに発行します。

このトピックに発行するとき、コネクタは + ワイルドカードをコアのモノ名に置き換えます。

出力例

```
{
   "topic": "gpio/my-core-thing/22/write",
   "error": "Invalid GPIO operation",
   "long_description": "GPIO 22 is configured as an INPUT GPIO. Write operations
 are not permitted."
 }
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

   b. 必要なローカルデバイスリソースを追加し、Lambda 関数への読み取り/書き込みアクセスを許可します。

   c. コネクタを追加し、そのパラメータを設定します。

   d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。

5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。この例では、入力 GPIO ピンのセットに対する読み取りリクエストを送信します。コアの名前とピン番号を使用してトピックを構成する方法を示します。

```
import greengrasssdk
import json
import os

iot_client = greengrasssdk.client('iot-data')
INPUT_GPIOS = [6, 17, 22]

thingName = os.environ['AWS_IOT_THING_NAME']
```

```
def get_read_topic(gpio_num):
    return '/'.join(['gpio', thingName, str(gpio_num), 'read'])

def get_write_topic(gpio_num):
    return '/'.join(['gpio', thingName, str(gpio_num), 'write'])

def send_message_to_connector(topic, message=''):
    iot_client.publish(topic=topic, payload=str(message))

def set_gpio_state(gpio, state):
    send_message_to_connector(get_write_topic(gpio), str(state))

def read_gpio_state(gpio):
    send_message_to_connector(get_read_topic(gpio))

def publish_basic_message():
    for i in INPUT_GPIOS:
     read_gpio_state(i)

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

Raspberry Pi GPIO コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- RPi.GPIO/MIT

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	AWS リージョン サポート用にコネクタ ARN を更新しました。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- Raspberry Pi ドキュメントの「GPIO」

シリアルストリーミングコネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

シリアルストリームコネクタは、 AWS IoT Greengrass コアデバイスのシリアルポートに読み書きします。

このコネクタは 2 つのオペレーションモードをサポートしています。

- オンデマンド読み取り MQTT トピックに対する読み取りおよび書き込みリクエストを受け取り、読み取りオペレーションのレスポンスまたは書き込みオペレーションのステータスを発行します。
- ポーリング読み取り 定期的にシリアルポートから読み取ります。このモードではオンデマンド読み取りリクエストもサポートされます。

> **Note**
>
> 読み取りリクエストの最大読み取り長は 63,994 バイトに制限されています。書き込みリクエストの最大データ長は 128,000 バイトに制限されています。

このコネクタには、次のバージョンがあります。

バージョン	ARN
3	`arn:aws:greengrass:  region::/connectors/SerialStream/versions/3`
2	`arn:aws:greengrass:  region::/connectors/SerialStream/versions/2`
1	`arn:aws:greengrass:  region::/connectors/SerialStream/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- ターゲットシリアルポートを参照する Greengrass グループのローカルデバイスリソース。

> **Note**
>
> このコネクタをデプロイする前に、シリアルポートを設定し、このシリアルポートに対して読み書きできることを確認してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- ターゲットシリアルポートを参照する Greengrass グループのローカルデバイスリソース。

> **Note**
>
> このコネクタをデプロイする前に、シリアルポートを設定し、このシリアルポートに対して読み書きできることを確認してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

`BaudRate`

シリアル接続のボーレート。

AWS IoT コンソールの表示名: ボーレート

必須: `true`

型: `string`

有効な値: `110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200, 230400`

有効なパターン: `^110$|^300$|^600$|^1200$|^2400$|^4800$|^9600$|^14400$|^19200$|^28800$|^38400$|^56000$|^57600$|^115200$|^230400$`

`Timeout`

読み取りオペレーションのタイムアウト (秒単位)。

AWS IoT コンソールの表示名: タイムアウト

必須: `true`

型: `string`

有効な値: `1 - 59`

有効なパターン: `^([1-9]|[1-5][0-9])$`

`SerialPort`

デバイス上の物理シリアルポートへの絶対パス。これは、ローカルデバイスリソースに指定されているソースパスです。

AWS IoT コンソールの表示名: シリアルポート

必須: `true`

型: `string`

有効なパターン: `[/a-zA-Z0-9_-]+`

`SerialPort-ResourceId`

物理シリアルポートを表すローカルデバイスリソースの ID。

> **Note**
> このコネクタには、リソースへの読み取りと書き込みアクセスが付与されています。

AWS IoT コンソールの表示名: シリアルポートリソース

必須: `true`

型: `string`

有効なパターン: `[a-zA-Z0-9_-]+`

`PollingRead`

読み取りモードとしてポーリング読み取りまたはオンデマンド読み取りを設定します。

- ポーリング読み取りモードの場合は、`true` を指定します。このモードでは、`PollingInterval`、`PollingReadType`、および `PollingReadLength` プロパティは必須です。
- オンデマンド読み取りモードの場合は、`false` を指定します。このモードでは、タイプと長さの値が読み取りリクエストで指定されます。

AWS IoT コンソールの表示名: 読み取りモード

必須: `true`

型: `string`

有効な値: `true, false`

有効なパターン: `^([Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])$`

`PollingReadLength`

各ポーリング読み取りオペレーションで読み取るデータの長さ (バイト単位)。これは、ポーリング読み取りモードを使用している場合にのみ適用されます。

AWS IoT コンソールの表示名: ポーリングの読み取り長

必須: `false`。このプロパティは、`PollingRead` が `true` の場合に必須です。

型: `string`

有効なパターン: ^(|[1-9][0-9]{0,3}|[1-5][0-9]{4}|6[0-2][0-9]{3}|63[0-8][0-9]{2}|639[0-8][0-9]|6399[0-4])$

PollingReadInterval

ポーリング読み取りが行われる間隔 (秒単位)。これは、ポーリング読み取りモードを使用している場合にのみ適用されます。

AWS IoT コンソールの表示名: ポーリング読み取り間隔

必須: false。このプロパティは、PollingRead が true の場合に必須です。

型: string

有効な値: 1 ~ 999

有効なパターン: ^(|[1-9]|[1-9][0-9]|[1-9][0-9][0-9])$

PollingReadType

ポーリングスレッドが読み取るデータのタイプ。これは、ポーリング読み取りモードを使用している場合にのみ適用されます。

AWS IoT コンソールの表示名: ポーリング読み取りタイプ

必須: false。このプロパティは、PollingRead が true の場合に必須です。

型: string

有効な値: ascii, hex

有効なパターン: ^(|[Aa][Ss][Cc][Ii][Ii]|[Hh][Ee][Xx])$

RtsCts

RTS/CTS フロー制御を有効にするかどうかを示します。デフォルト値は false です。詳細については、「RTS、CTS、RTR」を参照してください。

AWS IoT コンソールの表示名: RTS/CTS フローコントロール

必須: false

型: string

有効な値: true, false

有効なパターン: ^(|[Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])$

XonXoff

ソフトウェアフロー制御を有効にするかどうかを示します。デフォルト値は false です。詳細については、「ソフトウェアフロー制御」を参照してください。

AWS IoT コンソールの表示名: ソフトウェアフローコントロール

必須: false

型: string

有効な値: true, false

有効なパターン: ^(|[Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])$

Parity

シリアルポートのパリティ。デフォルト値は N です。詳細については、「パリティ」を参照してください。

AWS IoT コンソールの表示名: シリアルポートパリティ

必須: false

型: string

有効な値: N, E, O, S, M

有効なパターン: ^(|[NEOSMneosm])$

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、シリアルストリーミングコネクタを含む初期バージョンで ConnectorDefinition を作成します。ポーリング読み取りモード用にコネクタを設定します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MySerialStreamConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/SerialStream/
versions/3",
```

```
            "Parameters": {
                "BaudRate" : "9600",
                "Timeout" : "25",
                "SerialPort" : "/dev/serial1",
                "SerialPort-ResourceId" : "my-serial-port-resource",
                "PollingRead" : "true",
                "PollingReadLength" : "30",
                "PollingReadInterval" : "30",
                "PollingReadType" : "hex"
            }
        }
    ]
}'
```

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、以下の 2 つの MQTT トピックのシリアルポートに対する読み取りリクエストまたは書き込みリクエストを受け入れます。入力メッセージは JSON 形式である必要があります。

- `serial/+/read/#` トピックの読み取りリクエスト。
- `serial/+/write/#` トピックの書き込みリクエスト。

これらのトピックに発行するには、+ ワイルドカードを Core モノ名に置き換え、# ワイルドカードをシリアルポートへのパスに置き換えます。例:

```
serial/core-thing-name/read/dev/serial-port
```

トピックのフィルター: `serial/+/read/#`

このトピックを使用して、オンデマンド読み取りリクエストをシリアルピンに送信します。読み取りリクエストの最大読み取り長は 63,994 バイトに制限されています。

メッセージのプロパティ

`readLength`

シリアルポートから読み取るデータの長さ。

必須: `true`

型: `string`

有効なパターン: `^[1-9][0-9]*$`

`type`

読み取るデータのタイプ。

必須: `true`

型: `string`

有効な値: `ascii, hex`

有効なパターン: `(?i)^(ascii|hex)$`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。

必須: `false`

型: `string`

有効なパターン: `.+`

入力例

```
{
    "readLength": "30",
    "type": "ascii",
    "id": "abc123"
}
```

トピックのフィルター: `serial/+/write/#`

このトピックを使用して書き込みリクエストをシリアルピンに送信します。書き込みリクエストの最大データ長は 128,000 バイトに制限されています。

メッセージのプロパティ

`data`

シリアルポートに書き込む文字列。

必須: `true`

型: `string`

有効なパターン: `^[1-9][0-9]*$`

`type`

読み取るデータのタイプ。

必須: `true`

型: `string`

有効な値: `ascii, hex`

有効なパターン: `^(ascii|hex|ASCII|HEX)$`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。

必須: `false`

型: `string`

有効なパターン: `.+`

入力例: ASCII リクエスト

```
{
    "data": "random serial data",
    "type": "ascii",
    "id": "abc123"
}
```

入力例: 16 進リクエスト

```
{
    "data": "base64 encoded data",
    "type": "hex",
    "id": "abc123"
```

```
}
```

出力データ

コネクタは 2 つのトピックに関する出力データを発行します。

- `serial/+/status/#` トピックに関するコネクタからのステータス情報。
- `serial/+/read_response/#` トピックに関する読み取りリクエストからのレスポンス。

このトピックに発行すると、コネクタは + ワイルドカードを Core モノ名に、# ワイルドカードをシリアルポートへのパスに置き換えます。例:

```
serial/core-thing-name/status/dev/serial-port
```

トピックのフィルター: `serial/+/status/#`

このトピックを使用して、読み取りおよび書き込みリクエストのステータスをリッスンします。`id` プロパティがリクエストに含まれている場合は、レスポンスで返されます。

出力例: 成功

```
{
    "response": {
        "status": "success"
    },
    "id": "abc123"
}
```

出力例: 失敗

失敗レスポンスには、読み取りまたは書き込みオペレーションの実行中に発生したエラーまたはタイムアウトを説明する `error_message` プロパティが含まれています。

```
{
    "response": {
        "status": "fail",
        "error_message": "Could not write to port"
    },
    "id": "abc123"
}
```

トピックのフィルター: `serial/+/read_response/#`

このトピックを使用して、読み取りオペレーションからレスポンスデータを受信します。タイプが hex の場合、レスポンスデータは Base64 でエンコードされます。

出力例

```
{
    "data": "output of serial read operation"
    "id": "abc123"
}
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。
3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

b. 必要なローカルデバイスリソースを追加し、Lambda 関数への読み取り/書き込みアクセスを許可します。

c. コネクタをグループに追加し、そのパラメータを設定します。

d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションをグループに追加します。

- Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
- コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import json

TOPIC_REQUEST = 'serial/CORE_THING_NAME/write/dev/serial1'

# Creating a greengrass core sdk client
iot_client = greengrasssdk.client('iot-data')

def create_serial_stream_request():
 request = {
  "data": "TEST",
  "type": "ascii",
  "id": "abc123"
 }
 return request
```

```
def publish_basic_request():
 iot_client.publish(payload=json.dumps(create_serial_stream_request()),
 topic=TOPIC_REQUEST)

publish_basic_request()

def lambda_handler(event, context):
 return
```

ライセンス

シリアルストリーミングコネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- pyserial/BSD

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	AWS リージョン サポート用にコネクタ ARN を更新しました。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する

- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

ServiceNow MetricBase 統合コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

ServiceNow MetricBase 統合コネクタは、Greengrass デバイスから ServiceNow MetricBase に時系列メトリクスを発行します。これにより、Greengrass Core 環境からの時系列データを保存、分析、可視化して、ローカルイベントに対処できます。

このコネクタは、MQTT トピックに関する時系列データを受け取り、ServiceNow API に定期的に発行します。

このコネクタを使用して、以下のようなシナリオをサポートできます。

- Greengrass デバイスから収集した時系列データに基づいて、しきい値ベースのアラートとアラームを作成する。
- ServiceNow プラットフォームに構築されたカスタムアプリケーションで、Greengrass デバイスのタイムサービスデータを使用する。

このコネクタには、次のバージョンがあります。

バージョン	ARN
4	`arn:aws:greengrass:  region::/connectors/ServiceNowMetricBaseIntegration/versions/4`

バージョン	ARN
3	`arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/3`
2	`arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/2`
1	`arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3 - 4

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

  > **Note**
  > この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- MetricBase へのサブスクリプションを有効化した ServiceNow アカウント。さらに、メトリクスとメトリクステーブルをアカウントに作成する必要があります。詳細については、ServiceNow ドキュメントの「MetricBase」を参照してください。
- 基本認証を使用して ServiceNow インスタンスにログインするためのユーザー名とパスワード AWS Secrets Manager を保存する のテキストタイプのシークレット。シークレットには、対応する値が設定された「user」と「password」キーが含まれている必要があります。詳細については、「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。
- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

- MetricBase へのサブスクリプションを有効化した ServiceNow アカウント。さらに、メトリクスとメトリクステーブルをアカウントに作成する必要があります。詳細については、ServiceNow ドキュメントの「MetricBase」を参照してください。
- 基本認証を使用して ServiceNow インスタンスにログインするためのユーザー名とパスワード AWS Secrets Manager を保存する のテキストタイプのシークレット。シークレットには、対応する値が設定された「user」と「password」キーが含まれている必要があります。詳細については、「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。
- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Version 4

`PublishInterval`

ServiceNow への発行イベント間の最大待機秒数。最大値は 900 です。

`PublishBatchSize` に達するか、`PublishInterval` が有効期限切れになると、コネクタは ServiceNow への発行を行います。

AWS IoT コンソールの表示名: 秒単位の発行間隔

必須: `true`

型: `string`

有効な値: `1 - 900`

有効なパターン: `[1-9]|[1-9]\d|[1-9]\d\d|900`

`PublishBatchSize`

ServiceNow への発行前にバッチにまとめることのできるメトリクス値の最大数。

`PublishBatchSize` に達するか、`PublishInterval` が有効期限切れになると、コネクタは ServiceNow への発行を行います。

AWS IoT コンソールの表示名: バッチサイズを発行する

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`InstanceName`

ServiceNow への接続に使用されるインスタンスの名前。

AWS IoT コンソールの表示名: ServiceNow インスタンスの名前

必須: `true`

型: `string`

有効なパターン: `.+`

`DefaultTableName`

時系列 MetricBase データベースに関連付けられた `GlideRecord` を含むテーブルの名前。入力メッセージペイロードの `table` プロパティは、この値を上書きするために使用できます。

AWS IoT コンソールの表示名: メトリクスを含むテーブルの名前

必須: `true`

型: `string`

有効なパターン: `.+`

`MaxMetricsToRetain`

新しいメトリクスに置き換えられるまでメモリに保存するメトリクスの最大数。

この制限は、インターネットへの接続がなく、コネクタが後で発行するメトリクスをバッファし始めるときに適用されます。バッファが満杯になると、最も古いメトリクスが新しいメトリクスに置き換えられます。

> **Note**
>
> コネクタのホストプロセスが中断された場合、メトリクスは保存されません。例えば、この状況はグループデプロイ中またはデバイスの再起動時に発生する可能性があります。

この値は、バッチサイズより大きく、MQTT メッセージの受信率に基づいてメッセージを保持するのに十分な大きさであることが必要です。

AWS IoT コンソールの表示名: メモリに保持する最大メトリクス

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`AuthSecretArn`

ServiceNow ユーザー名とパスワード AWS Secrets Manager を保存する のシークレット。これはテキスト形式のシークレットであることが必要です。シークレットには、対応する値が設定された「user」と「password」キーが含まれている必要があります。

AWS IoT コンソールの表示名: 認証シークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z0-9\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\]+/)*[a-zA-Z0-9/_+=,.@\-]+-[a-zA-Z0-9]+`

`AuthSecretArn-ResourceId`

ServiceNow 認証情報の Secrets Manager シークレットを参照するグループのシークレットリソース。

AWS IoT コンソールの表示名: 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Version 1 - 3

`PublishInterval`

ServiceNow への発行イベント間の最大待機秒数。最大値は 900 です。

`PublishBatchSize` に達するか、`PublishInterval` が有効期限切れになると、コネクタは ServiceNow への発行を行います。

AWS IoT コンソールの表示名: 秒単位の発行間隔

必須: `true`

型: `string`

有効な値: `1 - 900`

有効なパターン: `[1-9]|[1-9]\d|[1-9]\d\d|900`

`PublishBatchSize`

ServiceNow への発行前にバッチにまとめることのできるメトリクス値の最大数。

`PublishBatchSize` に達するか、`PublishInterval` が有効期限切れになると、コネクタは ServiceNow への発行を行います。

AWS IoT コンソールの表示名: バッチサイズを発行する

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`InstanceName`

ServiceNow への接続に使用されるインスタンスの名前。

AWS IoT コンソールの表示名: ServiceNow インスタンスの名前

必須: `true`

型: `string`

有効なパターン: `.+`

`DefaultTableName`

時系列 MetricBase データベースに関連付けられた `GlideRecord` を含むテーブルの名前。入力メッセージペイロードの `table` プロパティは、この値を上書きするために使用できます。

AWS IoT コンソールの表示名: メトリクスを含むテーブルの名前

必須: `true`

型: `string`

有効なパターン: `.+`

`MaxMetricsToRetain`

新しいメトリクスに置き換えられるまでメモリに保存するメトリクスの最大数。

この制限は、インターネットへの接続がなく、コネクタが後で発行するメトリクスをバッファし始めるときに適用されます。バッファが満杯になると、最も古いメトリクスが新しいメトリクスに置き換えられます。

> **Note**
>
> コネクタのホストプロセスが中断された場合、メトリクスは保存されません。例えば、この状況はグループデプロイ中またはデバイスの再起動時に発生する可能性があります。

この値は、バッチサイズより大きく、MQTT メッセージの受信率に基づいてメッセージを保持するのに十分な大きさであることが必要です。

AWS IoT コンソールの表示名: メモリに保持する最大メトリクス

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`AuthSecretArn`

ServiceNow ユーザー名とパスワード AWS Secrets Manager を保存する のシークレット。これはテキスト形式のシークレットであることが必要です。シークレットには、対応する値が設定された「user」と「password」キーが含まれている必要があります。

AWS IoT コンソールの表示名: 認証シークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z0-9\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\]+/)*[a-zA-Z0-9/_+=,.@\-]+-[a-zA-Z0-9]+`

`AuthSecretArn-ResourceId`

ServiceNow 認証情報の Secrets Manager シークレットを参照するグループのシークレットリソース。

AWS IoT コンソールの表示名: 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、ServiceNow MetricBase 統合コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyServiceNowMetricBaseIntegrationConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
ServiceNowMetricBaseIntegration/versions/4",
            "Parameters": {
                "PublishInterval" : "10",
                "PublishBatchSize" : "50",
                "InstanceName" : "myinstance",
                "DefaultTableName" : "u_greengrass_app",
                "MaxMetricsToRetain" : "20000",
                "AuthSecretArn" : "arn:aws:secretsmanager:region:account-
id:secret:greengrass-secret-hash",
                "AuthSecretArn-ResourceId" : "MySecretResource",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、MQTT トピックに関する時系列メトリクスを受け取り、それを ServiceNow に発行します。入力メッセージは JSON 形式である必要があります。

サブスクリプションのトピックフィルター

```
servicenow/metricbase/metric
```

メッセージのプロパティ

`request`

テーブル、レコード、メトリクスに関する情報。このリクエストは時系列 POST リクエストの `seriesRef` オブジェクトを表します。詳細については、「Clotho Time Series API - POST」を参照してください。

必須: `true`

型: 以下のプロパティを含む `object`。

`subject`

テーブル内の特定のレコードの `sys_id`。

必須: `true`

型: `string`

`metric_name`

メトリクスフィールドの名前。

必須: `true`

型: `string`

`table`

レコードを保存するテーブルの名前。`DefaultTableName` パラメータを無効にするには、この値を指定します。

必須: `false`

型: `string`

`value`

個々のデータポイントの値。

必須: `true`

型: `float`

`timestamp`

個々のデータポイントのタイムスタンプ。デフォルト値は現在の時刻です。

必須: `false`

型: `string`

入力例

```
{
    "request": {
        "subject":"ef43c6d40a0a0b5700c77f9bf387afe3",
        "metric_name":"u_count",
        "table": "u_greengrass_app"
        "value": 1.0,
        "timestamp": "2018-10-14T10:30:00"
    }
}
```

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。

サブスクリプションのトピックフィルター

```
servicenow/metricbase/metric/status
```

出力例: 成功

```
{
    "response": {
        "metric_name": "Errors",
        "table_name": "GliderProd",
        "processed_on": "2018-10-14T10:35:00",
        "response_id": "khjKSkj132qwr23fcba",
        "status": "success",
        "values": [
            {
                "timestamp": "2016-10-14T10:30:00",
                "value": 1.0
            },
            {
                "timestamp": "2016-10-14T10:31:00",
                "value": 1.1
            }
        ]
```

```
    }
}
```

出力例: 失敗

```
{
    "response": {
        "error": "InvalidInputException",
        "error_message": "metric value is invalid",
        "status": "fail"
    }
}
```

> **Note**
>
> コネクタが再試行可能なエラー (接続エラーなど) を検出した場合は、次のバッチ処理で再発行を試します。

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォル

ダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。
   b. 必要なシークレットリソースを追加し、Lambda 関数への読み取りアクセスを許可します。
   c. コネクタを追加し、そのパラメータを設定します。
   d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
SEND_TOPIC = 'servicenow/metricbase/metric'

def create_request_with_all_fields():
```

```
    return {
        "request": {
             "subject": '2efdf6badbd523803acfae441b961961',
             "metric_name": 'u_count',
             "value": 1234,
             "timestamp": '2018-10-20T20:22:20',
             "table": 'u_greengrass_metricbase_test'
        }
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=SEND_TOPIC,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

ServiceNow MetricBase 統合コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- pysnow/MIT

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
4	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。

バージョン	変更
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

SNS コネクタ

SNS コネクタは、Amazon SNS トピックにメッセージを発行します。これにより、ウェブサーバー、E メールアドレス、および他のメッセージサブスクライバーは Greengrass グループのイベントに応答できます。

このコネクタは、MQTT トピックに関する SNS メッセージ情報を受信し、指定された SNS トピックに送信します。オプションでカスタム Lambda 関数を使用して、メッセージをこのコネクタに発行する前にフィルタリング処理および書式設定するロジックを実装できます。

このコネクタには、次のバージョンがあります。

バージョン	ARN
4	`arn:aws:greengrass:  region::/connectors/SNS/versions/4`

バージョン	ARN
3	`arn:aws:greengrass:  region::/connectors/SNS/versions/3`
2	`arn:aws:greengrass:  region::/connectors/SNS/versions/2`
1	`arn:aws:greengrass:  region::/connectors/SNS/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3 - 4

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。
- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- 設定済みの SNS トピック。詳細については、Amazon Simple 通知サービス デベロッパーガイドの「Amazon SNS トピックの作成」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲットの Amazon SNS トピックでの `sns:Publish` アクションを許可するように設定された Greengrass グループロール。

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"Stmt1528133056761",
            "Action":[
                "sns:Publish"
            ],
            "Effect":"Allow",
            "Resource":[
                "arn:aws:sns:region:account-id:topic-name"
            ]
        }
    ]
 }
```

このコネクタでは、入力メッセージペイロードのデフォルトトピックを動的に上書きできます。実装でこの機能を使用している場合、IAM ポリシーはすべてのターゲットトピックに対する `sns:Publish` アクセス許可を付与する必要があります。リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。
- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- 設定済みの SNS トピック。詳細については、Amazon Simple 通知サービス デベロッパーガイドの「Amazon SNS トピックの作成」を参照してください。
- 以下の IAM ポリシーの例に示すように、ターゲットの Amazon SNS トピックでの `sns:Publish` アクションを許可するように設定された Greengrass グループロール。

```
{
```

```
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"Stmt1528133056761",
            "Action":[
                "sns:Publish"
            ],
            "Effect":"Allow",
            "Resource":[
                "arn:aws:sns:region:account-id:topic-name"
            ]
        }
    ]
}
```

このコネクタでは、入力メッセージペイロードのデフォルトトピックを動的に上書きできます。実装でこの機能を使用している場合、IAM ポリシーはすべてのターゲットトピックに対する `sns:Publish` アクセス許可を付与する必要があります。リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。

グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Version 4

`DefaultSNSArn`

メッセージを発行するデフォルト SNS トピックの ARN。ターゲットトピックは入力メッセージペイロードの `sns_topic_arn` プロパティによって上書きできます。

> **Note**
>
> グループロールは、すべてのターゲットトピックに対する `sns:Publish` アクセス許可を付与する必要があります。詳細については、「the section called “要件”」を参照してください。

AWS IoT コンソールの表示名: デフォルトの SNS トピック ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:sns:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):([a-zA-Z0-9-_]+)$`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
>
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Versions 1 - 3

`DefaultSNSArn`

メッセージを発行するデフォルト SNS トピックの ARN。ターゲットトピックは入力メッセージペイロードの `sns_topic_arn` プロパティによって上書きできます。

> **Note**
>
> グループロールは、すべてのターゲットトピックに対する `sns:Publish` アクセス許可を付与する必要があります。詳細については、「the section called “要件”」を参照してください。

AWS IoT コンソールの表示名: デフォルトの SNS トピック ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:sns:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):([a-zA-Z0-9-_]+)$`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、SNS コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MySNSConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/SNS/versions/4",
            "Parameters": {
                "DefaultSNSArn": "arn:aws:sns:region:account-id:topic-name",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、MQTT トピックに関する SNS メッセージ情報を受け取り、その情報をそのままターゲット SNS トピックに発行します。入力メッセージは JSON 形式である必要があります。

サブスクリプションのトピックフィルター

`sns/message`

メッセージのプロパティ

`request`

SNS トピックに送信するメッセージに関する情報。

必須: `true`

型: 以下のプロパティを含む `object`。

`message`

文字列または JSON 形式でのメッセージの内容。例については、「入力例」を参照してください。

JSON を送信するには、`message_structure` プロパティを `json` に設定する必要があります。メッセージは、文字列エンコードされた JSON オブジェクト (`default` キーを含む) であることが必要です。

必須: `true`

型: `string`

有効なパターン: `.*`

`subject`

メッセージの件名。

必須: `false`

: ASCII テキスト (最大 100 文字)。これは、文字、数字、または句読点で始まる必要があります。改行や制御文字は使用しないでください。

有効なパターン: `.*`

`sns_topic_arn`

メッセージを発行する SNS トピックの ARN。指定した場合、コネクタは、デフォルトのトピックではなくこのトピックに発行します。

> **Note**
>
> グループロールは、すべてのターゲットトピックに対する `sns:Publish` アクセス許可を付与する必要があります。詳細については、「the section called “要件”」を参照してください。

必須: `false`

型: `string`

有効なパターン: `arn:aws:sns:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):([a-zA-Z0-9-_]+)$`

`message_structure`

メッセージの構造。

必須: `false`。これは、JSON メッセージを送信するために指定する必要があります。

型: `string`

有効な値: `json`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。指定すると、レスポンスオブジェクトの `id` プロパティがこの値に設定されます。この機能を使用しない場合は、このプロパティを省略するか空の文字列を指定できます。

必須: `false`

型: `string`

有効なパターン: `.*`

制限

メッセージサイズは、256 KB の最大 SNS メッセージサイズによって制限されます。

入力例: 文字列メッセージ

この例では、文字列メッセージを送信します。オプションの `sns_topic_arn` プロパティを指定します。このプロパティは、デフォルトの送信先トピックよりも優先されます。

```
{
    "request": {
        "subject": "Message subject",
        "message": "Message data",
        "sns_topic_arn": "arn:aws:sns:region:account-id:topic2-name"
    },
    "id": "request123"
}
```

入力例: JSON メッセージ

この例では、メッセージを文字列エンコードされた JSON オブジェクト (`default` キーを含む) として送信します。

```
{
    "request": {
        "subject": "Message subject",
        "message": "{ \"default\": \"Message data\" }",
        "message_structure": "json"
    },
    "id": "request123"
}
```

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。

サブスクリプションのトピックフィルター

`sns/message/status`

出力例: 成功

```
{
```

```
    "response": {
        "sns_message_id": "f80a81bc-f44c-56f2-a0f0-d5af6a727c8a",
        "status": "success"
    },
    "id": "request123"
}
```

出力例: 失敗

```
{
   "response" : {
        "error": "InvalidInputException",
        "error_message": "SNS Topic Arn is invalid",
        "status": "fail"
   },
   "id": "request123"
}
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。
> - 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。

   グループロール要件では、必要なアクセス許可を付与するようにロールを設定し、ロールがグループに追加されていることを確認する必要があります。詳細については、the section called “グループロールの管理 (コンソール)” または the section called “グループロールの管理 (CLI)” を参照してください。

2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。

3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

   b. コネクタを追加し、そのパラメータを設定します。

   c. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。

5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
```

```
send_topic = 'sns/message'

def create_request_with_all_fields():
    return  {
        "request": {
            "message": "Message from SNS Connector Test"
        },
        "id" : "req_123"
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=send_topic,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

SNS コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF ライセンス
- docutils/BSD ライセンス、GNU 一般パブリックライセンス (GPL)、Python Software Foundation ライセンス、パブリックドメイン
- jmespath/MIT ライセンス
- s3transfer/Apache License 2.0
- urllib3/MIT ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
4	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- Boto 3 ドキュメントの「発行アクション」
- Amazon Simple Notification Service デベロッパーガイドの Amazon Simple Notification Service とは

Splunk 統合コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

Splunk 統合コネクタは Greengrass デバイスから Splunk にデータを発行します。これにより、Splunk を使用して Greengrass Core 環境をモニタリングおよび分析し、ローカルイベントに対処できます。コネクタは HTTP Event Collector (HEC) と統合されています。詳細については、「Splunk HTTP Event Collector の概要」を参照してください。

このコネクタは、MQTT トピックのログおよびイベントデータを受信し、そのデータをそのまま Splunk API に発行します。

このコネクタを使用して、以下のような産業シナリオをサポートできます。

- オペレーターは、アクチュエーターやセンサーからの定期的なデータ (温度、圧力、水の読み取り値など) を使用して、値が特定のしきい値を超えたときにアラームが開始されるようにする。
- 開発者は、産業機械から収集されたデータを使用して、問題の兆候をデバイスでモニタリングできる ML モデルを構築する。

このコネクタには、次のバージョンがあります。

バージョン	ARN
4	`arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/4`
3	`arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/3`
2	`arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/2`
1	`arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 3 - 4

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- HTTP Event Collector 機能を Splunk で有効にする必要があります。詳細については、Splunk ドキュメントの「Set up and use HTTP Event Collector in Splunk Web」を参照してください。
- Splunk HTTP Event Collector トークン AWS Secrets Manager を保存する のテキストタイプのシークレット。詳細については、Splunk ドキュメントの「About event collector tokens」(イベントコレクタトークンについて) と「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。

> **Note**
>
> Secrets Manager コンソールでシークレットを作成するには、[Plaintext] (プレーンテキスト) タブにトークンを入力します。引用符やその他の書式は含めないでください。API で、`SecretString` プロパティの値としてトークンを指定します。

- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

Versions 1 - 2

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- HTTP Event Collector 機能を Splunk で有効にする必要があります。詳細については、Splunk ドキュメントの「Set up and use HTTP Event Collector in Splunk Web」を参照してください。
- Splunk HTTP Event Collector トークン AWS Secrets Manager を保存する のテキストタイプのシークレット。詳細については、Splunk ドキュメントの「About event collector tokens」(イベントコレクタトークンについて) と「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。

> **Note**
>
> Secrets Manager コンソールでシークレットを作成するには、[Plaintext] (プレーンテキスト) タブにトークンを入力します。引用符やその他の書式は含めないでください。API で、`SecretString` プロパティの値としてトークンを指定します。

- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Version 4

`SplunkEndpoint`

Splunk インスタンスのエンドポイント。この値には、プロトコル、ホスト名、ポートが含まれている必要があります。

AWS IoT コンソールの表示名: Splunk エンドポイント

必須: `true`

型: `string`

有効なパターン: `^(http:\/\/|https:\/\/)?[a-z0-9]+([-.]{1}[a-z0-9]+)*.[a-z]{2,5}(:[0-9]{1,5})?(\/.*)?$`

`MemorySize`

コネクタに割り当てるメモリ量 (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`SplunkQueueSize`

項目が送信または破棄されるまでにメモリに保存する項目の最大数。この制限に達すると、キュー内の最も古い項目が新しい項目に置き換えられます。この制限は通常、インターネットに接続していない場合に適用されます。

AWS IoT コンソールの表示名: 保持する最大項目数

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`SplunkFlushIntervalSeconds`

受信したデータを Splunk HEC に発行する間隔 (秒単位)。最大値は 900 です。アイテムを受信時にバッチにまとめないで発行するようにコネクタを設定するには、0 を指定します。

AWS IoT コンソールでの名前の表示: Splunk 発行間隔

必須: `true`

型: `string`

有効なパターン: `[0-9]|[1-9]\d|[1-9]\d\d|900`

`SplunkTokenSecretArn`

Splunk トークン AWS Secrets Manager を保存する のシークレット。これはテキスト形式のシークレットであることが必要です。

AWS IoT コンソールの表示名: Splunk 認証トークンシークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z]{2}-[a-z]+-\d{1}:\d{12}?:secret:[a-zA-Z0-9-_]+-[a-zA-Z0-9-_]+`

`SplunkTokenSecretArn-ResourceId`

Splunk シークレットを参照する Greengrass グループのシークレットリソース。

AWS IoT コンソールでの名前の表示: Splunk 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`SplunkCustomCALocation`

Splunk のカスタム認証機関 (CA) のファイルパス (`/etc/ssl/certs/splunk.crt` など)。

AWS IoT コンソールでの名前の表示: Splunk カスタム認証機関の場所

必須: `false`

型: `string`

有効なパターン: `^$|/.*`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Version 1 - 3

`SplunkEndpoint`

Splunk インスタンスのエンドポイント。この値には、プロトコル、ホスト名、ポートが含まれている必要があります。

AWS IoT コンソールでの名前の表示: Splunk エンドポイント

必須: `true`

型: `string`

有効なパターン: `^(http:\/\/|https:\/\/)?[a-z0-9]+([-.]{1}[a-z0-9]+)*.[a-z]{2,5}(:[0-9]{1,5})?(\/.*)?$`

`MemorySize`

コネクタに割り当てるメモリ量 (KB 単位)。

AWS IoT コンソールの表示名: メモリサイズ

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`SplunkQueueSize`

項目が送信または破棄されるまでにメモリに保存する項目の最大数。この制限に達すると、キュー内の最も古い項目が新しい項目に置き換えられます。この制限は通常、インターネットに接続していない場合に適用されます。

AWS IoT コンソールの表示名: 保持する最大項目数

必須: `true`

型: `string`

有効なパターン: `^[0-9]+$`

`SplunkFlushIntervalSeconds`

受信したデータを Splunk HEC に発行する間隔 (秒単位)。最大値は 900 です。アイテムを受信時にバッチにまとめないで発行するようにコネクタを設定するには、0 を指定します。

AWS IoT コンソールの表示名: Splunk 発行間隔

必須: `true`

型: `string`

有効なパターン: `[0-9]|[1-9]\d|[1-9]\d\d|900`

`SplunkTokenSecretArn`

Splunk トークン AWS Secrets Manager を保存する のシークレット。これはテキスト形式のシークレットであることが必要です。

AWS IoT コンソールでの名前の表示: Splunk 認証トークンシークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z]{2}-[a-z]+-\d{1}:\d{12}?:secret:[a-zA-Z0-9-_]+-[a-zA-Z0-9-_]+`

`SplunkTokenSecretArn-ResourceId`

Splunk シークレットを参照する Greengrass グループのシークレットリソース。

AWS IoT コンソールの表示名: Splunk 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`SplunkCustomCALocation`

Splunk のカスタム認証機関 (CA) のファイルパス (`/etc/ssl/certs/splunk.crt` など)。

AWS IoT コンソールでの名前の表示: Splunk カスタム認証機関の場所

必須: `false`

型: `string`

有効なパターン: `^$|/.*`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドは、Splunk 統合コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MySplunkIntegrationConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/SplunkIntegration/
versions/4",
            "Parameters": {
                "SplunkEndpoint": "https://myinstance.cloud.splunk.com:8088",
                "MemorySize": 200000,
```

```
                "SplunkQueueSize": 10000,
                "SplunkFlushIntervalSeconds": 5,
                "SplunkTokenSecretArn":"arn:aws:secretsmanager:region:account-
id:secret:greengrass-secret-hash",
                "SplunkTokenSecretArn-ResourceId": "MySplunkResource",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

> **Note**
>
> このコネクタの Lambda 関数には存続期間の長いライフサイクルがあります。

AWS IoT Greengrass コンソールでは、グループのコネクタページからコネクタを追加できます。詳細については、「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、MQTT トピックのログおよびイベントデータを受け取り、そのまま Splunk API に発行します。入力メッセージは JSON 形式である必要があります。

サブスクリプションのトピックフィルター

`splunk/logs/put`

メッセージのプロパティ

`request`

Splunk API に送信するイベントデータ。イベントは services/collector API の仕様を満たしている必要があります。

必須: `true`

型: `object`。`event` プロパティのみが必須です。

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力ステータスにマッピングするために使用します。

必須: `false`

型: `string`

制限

このコネクタを使用する場合、Splunk API によって課せられるすべての制限が適用されます。詳細については、「services/collector」を参照してください。

入力例

```
{
    "request": {
        "event": "some event",
        "fields": {
            "severity": "INFO",
            "category": [
                "value1",
                "value2"
            ]
        }
    },
    "id": "request123"
}
```

出力データ

このコネクタは 2 つのトピックに関する出力データを発行します。

- `splunk/logs/put/status` トピックのステータス情報。
- `splunk/logs/put/error` トピックでのエラー。

トピックのフィルター: `splunk/logs/put/status`

このトピックを使用して、リクエストの状態をリッスンします。コネクタは受信したデータのバッチを Splunk API に送信するたびに、成功したリクエストの ID と失敗したリクエストの ID のリストを発行します。

出力例

```
{
    "response": {
```

```
        "succeeded": [
            "request123",
            ...
        ],
        "failed": [
            "request789",
            ...
        ]
    }
}
```

トピックのフィルター: `splunk/logs/put/error`

このトピックを使用して、コネクタからのエラーをリッスンします。リクエストの処理中に発生したエラーまたはタイムアウトを表す `error_message` プロパティ。

出力例

```
{
    "response": {
        "error": "UnauthorizedException",
        "error_message": "invalid splunk token",
        "status": "fail"
    }
}
```

> **Note**
>
> コネクタが再試行可能なエラー (接続エラーなど) を検出した場合は、次のバッチ処理で再発行を試します。

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> - 他の Python ランタイムを使用する場合は、Python 3.x から Python 3.7 へのシンボリックリンクを作成します。

- 「コネクタの使用を開始する (コンソール)」および「コネクタの使用を開始する (CLI)」トピックには、Twilio 通知コネクタの例を設定およびデプロイする方法を示す詳細なステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。
3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。

   b. 必要なシークレットリソースを追加し、Lambda 関数への読み取りアクセスを許可します。

   c. コネクタを追加し、そのパラメータを設定します。

   d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。
4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。

```
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'splunk/logs/put'

def create_request_with_all_fields():
    return {
        "request": {
            "event": "Access log test message."
        },
        "id" : "req_123"
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print("Message To Publish: ", messageToPublish)
    iot_client.publish(topic=send_topic,
        payload=json.dumps(messageToPublish))

publish_basic_message()

def lambda_handler(event, context):
    return
```

ライセンス

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
4	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
3	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
2	過剰なログ記録を減らすための修正。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”

Twilio 通知コネクタ

> **Warning**
>
> このコネクタは延長ライフサイクルフェーズに移行しており、 AWS IoT Greengrass では、機能、既存機能の拡張、セキュリティパッチ、バグ修正を提供するアップデートはリリースされません。詳細については、「AWS IoT Greengrass Version 1 メンテナンスポリシー」を参照してください。

Twilio 通知コネクタは、電話の通話を自動化したり、Twilio を介してテキストメッセージを送信したりします。このコネクタを使用して、Greengrass グループのイベントに応じて通知を送信できます。通話の場合、コネクタは音声メッセージを受取人に転送できます。

このコネクタは、MQTT トピックに関する Twilio メッセージ情報を受信し、Twilio 通知をトリガーします。

> **Note**
>
> Twilio 通知コネクタを使用する方法のチュートリアルについては、「the section called “コネクタの使用を開始する (コンソール)”」または「the section called “コネクタの使用を開始する (CLI)”」を参照してください。

このコネクタには、次のバージョンがあります。

バージョン	ARN
5	`arn:aws:greengrass:  region::/connectors/TwilioNotifications/versions/5`
4	`arn:aws:greengrass:  region::/connectors/TwilioNotifications/versions/4`
3	`arn:aws:greengrass:  region::/connectors/TwilioNotifications/versions/3`
2	`arn:aws:greengrass:  region::/connectors/TwilioNotifications/versions/2`
1	`arn:aws:greengrass:  region::/connectors/TwilioNotifications/versions/1`

バージョンの変更については、「Changelog」を参照してください。

要件

このコネクタには以下の要件があります。

Version 4 - 5

- AWS IoT Greengrass Core ソフトウェア v1.9.3 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 3.7 または 3.8 が Core デバイスにインストールされ、PATH 環境変数に追加されている。

> **Note**
>
> Python 3.8 を使用するには、次のコマンドを実行して、Python 3.7 のデフォルトのインストールフォルダからインストール済みの Python 3.8 バイナリへのシンボリックリンクを作成します。
>
> ```
> sudo ln -s path-to-python-3.8/python3.8 /usr/bin/python3.7
> ```
>
> これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。

- Twilio アカウントの SID、認証トークン、Twilio 対応の電話番号。Twilio プロジェクトを作成した後、これらの値をプロジェクトダッシュボードで使用できます。

> **Note**
>
> Twilio トライアルアカウントを使用できます。トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要が

あります。詳細については、「無料の Twilio トライアルアカウントを使用する方法」を参照してください。

- Twilio 認証トークン AWS Secrets Manager を保存する のテキストタイプのシークレット。詳細については、「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。

> **Note**
>
> Secrets Manager コンソールでシークレットを作成するには、[Plaintext] (プレーンテキスト) タブにトークンを入力します。引用符やその他の書式は含めないでください。API で、`SecretString` プロパティの値としてトークンを指定します。

- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

Versions 1 - 3

- AWS IoT Greengrass Core ソフトウェア v1.7 以降。「シークレット要件」で説明されているように、ローカルシークレットをサポートするように設定 AWS IoT Greengrass する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Python バージョン 2.7 が Core デバイスにインストールされ、PATH 環境変数に追加されている。
- Twilio アカウントの SID、認証トークン、Twilio 対応の電話番号。Twilio プロジェクトを作成した後、これらの値をプロジェクトダッシュボードで使用できます。

> **Note**
>
> Twilio トライアルアカウントを使用できます。トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要が

あります。詳細については、「無料の Twilio トライアルアカウントを使用する方法」を参照してください。

- Twilio 認証トークン AWS Secrets Manager を保存する のテキストタイプのシークレット。詳細については、「AWS Secrets Manager ユーザーガイド」の「Creating a basic secret」(基本的なシークレットの作成) を参照してください。

> **Note**
>
> Secrets Manager コンソールでシークレットを作成するには、[Plaintext] (プレーンテキスト) タブにトークンを入力します。引用符やその他の書式は含めないでください。API で、`SecretString` プロパティの値としてトークンを指定します。

- Secrets Manager シークレットを参照する Greengrass グループのシークレットリソース。詳細については、「Core にシークレットをデプロイする」を参照してください。

コネクタパラメータ

このコネクタには、以下のパラメータが用意されています。

Version 5

`TWILIO_ACCOUNT_SID`

Twilio API を呼び出すために使用される Twilio アカウント SID。

AWS IoT コンソールでの名前の表示: Twilio アカウント SID

必須: `true`

型: `string`

有効なパターン: `.+`

`TwilioAuthTokenSecretArn`

Twilio 認証トークンを保存する Secrets Manager シークレットの ARN。

> **Note**
>
> この ARN は、Core 上のローカルシークレットの値にアクセスするために使用されます。

AWS IoT コンソールの表示名: Twilio 認証トークンシークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z0-9\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\]+/)*[a-zA-Z0-9/_+=,.@\-]+-[a-zA-Z0-9]+`

`TwilioAuthTokenSecretArn-ResourceId`

Twilio 認証トークンのシークレットを参照する Greengrass グループのシークレットリソースの ID。

AWS IoT コンソールの表示名: Twilio 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`DefaultFromPhoneNumber`

Twilio がメッセージの送信に使用するデフォルトの Twilio 対応電話番号。Twilio はこの番号を使用してテキストまたは通話を開始します。

- デフォルトの電話番号を設定しない場合は、入力メッセージ本文の `from_number` プロパティに電話番号を指定する必要があります。
- デフォルトの電話番号を設定する場合は、オプションで、入力メッセージ本文の `from_number` プロパティを指定してデフォルトを上書きすることができます。

AWS IoT コンソールの表示名: 電話番号のデフォルト

必須: `false`

型: `string`

有効なパターン: `^$|\+[0-9]+`

`IsolationMode`

このコネクタのコンテナ化モード。デフォルトは です。つまり`GreengrassContainer`、コネクタは AWS IoT Greengrass コンテナ内の分離されたランタイム環境で実行されます。

> **Note**
> グループの既定のコンテナ化設定は、コネクタには適用されません。

AWS IoT コンソールの表示名: コンテナ分離モード

必須: `false`

型: `string`

有効な値: `GreengrassContainer` または `NoContainer`

有効なパターン: `^NoContainer$|^GreengrassContainer$`

Version 1 - 4

`TWILIO_ACCOUNT_SID`

Twilio API を呼び出すために使用される Twilio アカウント SID。

AWS IoT コンソールでの名前の表示: Twilio アカウント SID

必須: `true`

型: `string`

有効なパターン: `.+`

`TwilioAuthTokenSecretArn`

Twilio 認証トークンを保存する Secrets Manager シークレットの ARN。

> **Note**
>
> この ARN は、Core 上のローカルシークレットの値にアクセスするために使用されます。

AWS IoT コンソールの表示名: Twilio 認証トークンシークレットの ARN

必須: `true`

型: `string`

有効なパターン: `arn:aws:secretsmanager:[a-z0-9\-]+:[0-9]{12}:secret:([a-zA-Z0-9\\]+/)*[a-zA-Z0-9/_+=,.@\-]+-[a-zA-Z0-9]+`

`TwilioAuthTokenSecretArn-ResourceId`

Twilio 認証トークンのシークレットを参照する Greengrass グループのシークレットリソースの ID。

AWS IoT コンソールの表示名: Twilio 認証トークンリソース

必須: `true`

型: `string`

有効なパターン: `.+`

`DefaultFromPhoneNumber`

Twilio がメッセージの送信に使用するデフォルトの Twilio 対応電話番号。Twilio はこの番号を使用してテキストまたは通話を開始します。

- デフォルトの電話番号を設定しない場合は、入力メッセージ本文の `from_number` プロパティに電話番号を指定する必要があります。
- デフォルトの電話番号を設定する場合は、オプションで、入力メッセージ本文の `from_number` プロパティを指定してデフォルトを上書きすることができます。

AWS IoT コンソールの表示名: 電話番号のデフォルト

必須: `false`

型: `string`

有効なパターン: `^$|\+[0-9]+`

サンプルコネクタを作成する (AWS CLI)

以下の CLI コマンドの例では、Twilio 通知コネクタを含む初期バージョンで `ConnectorDefinition` を作成します。

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-
version '{
    "Connectors": [
        {
            "Id": "MyTwilioNotificationsConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
TwilioNotifications/versions/5",
            "Parameters": {
                "TWILIO_ACCOUNT_SID": "abcd12345xyz",
                "TwilioAuthTokenSecretArn": "arn:aws:secretsmanager:region:account-
id:secret:greengrass-secret-hash",
                "TwilioAuthTokenSecretArn-ResourceId": "MyTwilioSecret",
                "DefaultFromPhoneNumber": "+19999999999",
                "IsolationMode" : "GreengrassContainer"
            }
        }
    ]
}'
```

グループに Twilio 通知コネクタを追加する方法を示すチュートリアルについては、「the section called “コネクタの使用を開始する (CLI)”」および「the section called “コネクタの使用を開始する (コンソール)”」を参照してください。

入力データ

このコネクタは、2 つの MQTT トピックに関する Twilio メッセージ情報を受け取ります。入力メッセージは JSON 形式である必要があります。

- `twilio/txt` トピックに関するテキストメッセージ情報。
- `twilio/call` トピックに関する電話メッセージ情報。

> **Note**
>
> 入力メッセージのペイロードには、テキストメッセージ (`message`) またはボイスメッセージ (`voice_message_location`) を含めることができますが、両方を含めることはできません。

トピックのフィルター: **`twilio/txt`**

- メッセージのプロパティ
  - `request`

    Twilio 通知に関する情報。

    必須: `true`

    型: 以下のプロパティを含む `object`。
    - `recipient`

      メッセージ受取人。サポートされる受取人は 1 人のみです。

      必須: `true`

      タイプ: `object`。以下のプロパティを含みます。
      - `name`

        受取人の名前。

        必須: `true`

        型: `string`

        有効なパターン: `.*`
      - `phone_number`

        受取人の電話番号。

        必須: `true`

        型: `string`

有効なパターン: \+[1-9]+

message

テキストメッセージのテキスト内容。このトピックでは、テキストメッセージのみがサポートされています。音声メッセージの場合は、twilio/call を使用します。

必須: true

型: string

有効なパターン: .+

from_number

送信者の電話番号。Twilio はこの電話番号を使用してメッセージを開始します。このプロパティは、DefaultFromPhoneNumber パラメータが設定されていない場合に必須です。DefaultFromPhoneNumber が設定されている場合は、このプロパティを使用してデフォルトを上書きできます。

必須: false

型: string

有効なパターン: \+[1-9]+

retries

再試行の回数。デフォルトは 0 です。

必須: false

型: integer

id

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。

必須: true

型: string

有効なパターン: `.+`

入力例

```
{
    "request": {
        "recipient": {
            "name": "Darla",
            "phone_number": "+12345000000",
            "message": "Hello from the edge"
        },
        "from_number": "+19999999999",
        "retries": 3
    },
    "id": "request123"
}
```

トピックのフィルター: **twilio/call**

メッセージのプロパティ

`request`

Twilio 通知に関する情報。

必須: `true`

型: 以下のプロパティを含む `object`。

`recipient`

メッセージ受取人。サポートされる受取人は 1 人のみです。

必須: `true`

タイプ: `object`。以下のプロパティを含みます。

`name`

受取人の名前。

必須: `true`

型: `string`

有効なパターン: `.+`

`phone_number`

受取人の電話番号。

必須: `true`

型: `string`

有効なパターン: `\+[1-9]+`

`voice_message_location`

音声メッセージのオーディオコンテンツの URL。これは TwiML 形式であることが必要です。このトピックでは、音声メッセージのみがサポートされています。テキストメッセージの場合は、`twilio/txt` を使用します。

必須: `true`

型: `string`

有効なパターン: `.+`

`from_number`

送信者の電話番号。Twilio はこの電話番号を使用してメッセージを開始します。このプロパティは、`DefaultFromPhoneNumber` パラメータが設定されていない場合に必須です。`DefaultFromPhoneNumber` が設定されている場合は、このプロパティを使用してデフォルトを上書きできます。

必須: `false`

型: `string`

有効なパターン: `\+[1-9]+`

`retries`

再試行の回数。デフォルトは 0 です。

必須: `false`

型: `integer`

`id`

リクエストの任意の ID。このプロパティは、入力リクエストを出力レスポンスにマッピングするために使用します。

必須: `true`

型: `string`

有効なパターン: `.+`

入力例

```
{
    "request": {
        "recipient": {
            "name": "Darla",
            "phone_number": "+12345000000",
            "voice_message_location": "https://some-public-TwiML"
        },
        "from_number": "+19999999999",
        "retries": 3
    },
    "id": "request123"
}
```

出力データ

このコネクタは、MQTT トピックの出力データとしてステータス情報を発行します。

サブスクリプションのトピックフィルター

`twilio/message/status`

出力例: 成功

```
{
    "response": {
        "status": "success",
        "payload": {
            "from_number": "+19999999999",
            "messages": {
                "message_status": "queued",
```

```
                "to_number": "+12345000000",
                "name": "Darla"
            }
        }
    },
    "id": "request123"
}
```

出力例: 失敗

```
{
    "response": {
        "status": "fail",
        "error_message": "Recipient name cannot be None",
        "error": "InvalidParameter",
        "payload": None
        }
    },
    "id": "request123"
}
```

出力の payload プロパティは、メッセージが送信されたときの Twilio API からのレスポンスです。コネクタは、入力データが無効であること (必須の入力フィールドが指定されていないなど) を検出した場合、エラーを返し、値を None に設定します。以下に示しているのは、ペイロードの例です。

```
{
    'from_number':'+19999999999',
    'messages': {
        'name':'Darla',
        'to_number':'+12345000000',
        'message_status':'undelivered'
    }
}
```

```
{
    'from_number':'+19999999999',
    'messages': {
        'name':'Darla',
        'to_number':'+12345000000',
        'message_status':'queued'
    }
```

```
}
```

使用例

コネクタの試用に利用できる Python 3.7 Lambda 関数の例を設定するには、次のステップ (概要) を使用します。

> **Note**
>
> the section called “コネクタの使用を開始する (コンソール)” および the section called “コネクタの使用を開始する (CLI)” のトピックには、Twilio 通知コネクタのセットアップ、デプロイ、およびテスト方法を示すエンドツーエンドのステップが含まれています。

1. コネクタの要件を満たしていることを確認します。
2. 入力データをコネクタに送信する Lambda 関数を作成して発行します。

   サンプルコードを PY ファイルとして保存します。AWS IoT Greengrass Core SDK for Python をダウンロードして解凍します。次に、PY ファイルとルートレベルの `greengrasssdk` フォルダを含む zip パッケージを作成します。この zip パッケージは、 AWS Lambdaにアップロードするデプロイパッケージです。

   Python 3.7 Lambda 関数を作成したら、関数バージョンを公開し、エイリアスを作成します。
3. Greengrass グループを設定します。

   a. エイリアスで Lambda 関数を追加します (推奨)。Lambda ライフサイクルを長期間有効に (または CLI で `"Pinned": true` に) 設定します。
   b. 必要なシークレットリソースを追加し、Lambda 関数への読み取りアクセスを許可します。
   c. コネクタを追加し、そのパラメータを設定します。
   d. コネクタが入力データを受信し、サポートされているトピックフィルターで出力データを送信できるようにするサブスクリプションを追加します。

      - Lambda 関数をソースに、コネクタをターゲットに設定し、サポートされている入力トピックフィルターを使用します。
      - コネクタをソースとして、 AWS IoT Core をターゲットとして設定し、サポートされている出力トピックフィルターを使用します。このサブスクリプションを使用して、 AWS IoT コンソールでステータスメッセージを表示します。

4. グループをデプロイします。
5. AWS IoT コンソールのテストページで、出力データトピックをサブスクライブして、コネクタからのステータスメッセージを表示します。この例の Lambda 関数は長期間有効であり、グループがデプロイされた直後にメッセージの送信を開始します。

   テストが終了したら、Lambda ライフサイクルをオンデマンドに (または CLI で `"Pinned": false` に) 設定して、グループをデプロイできます。これにより、関数がメッセージの送信を停止します。

例

次の例では、Lambda 関数で入力メッセージをコネクタに送信します。この例では、テキストメッセージがトリガーされます。

```
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
TXT_INPUT_TOPIC = 'twilio/txt'
CALL_INPUT_TOPIC = 'twilio/call'

def publish_basic_message():

    txt = {
        "request": {
            "recipient" : {
                "name": "Darla",
                "phone_number": "+12345000000",
                "message": 'Hello from the edge'
            },
            "from_number" : "+19999999999"
        },
        "id" : "request123"
    }

    print("Message To Publish: ", txt)

    client.publish(topic=TXT_INPUT_TOPIC,
                   payload=json.dumps(txt))

publish_basic_message()
```

```
def lambda_handler(event, context):
    return
```

ライセンス

Twilio 通知コネクタには、以下のサードパーティーのソフトウェアおよびライセンスが含まれています。

- twilio-python/MIT

このコネクタは、Greengrass Core ソフトウェアライセンス契約に従ってリリースされます。

変更ログ

次の表に、コネクタの各バージョンにおける変更点を示します。

バージョン	変更
5	コネクタのコンテナ化モードを設定するための `IsolationMode` パラメータが追加されました。
4	Lambda ランタイムを Python 3.7 にアップグレードしたことで、ランタイム要件が変更。
3	過剰なログ記録を減らすための修正。
2	小さなバグ修正と機能向上。
1	初回リリース。

Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する

- the section called “コネクタの使用を開始する (コンソール)”
- the section called “コネクタの使用を開始する (CLI)”
- Twilio API リファレンス

Greengrass コネクタの開始方法 (コンソール)

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

このチュートリアルでは、 AWS Management Console を使用してコネクタを操作する方法を示します。

コネクタを使用して、開発ライフサイクルを短縮します。コネクタは、サービス、プロトコル、リソースとのやり取りを簡略する、あらかじめ組み込まれた再利用可能なモジュールです。それらのモジュールは、Greengrass デバイスにビジネスロジックをよりすばやくデプロイするのに役立ちます。詳細については、「コネクタを使用してサービスおよびプロトコルと統合する」を参照してください。

このチュートリアルでは、Twilio 通知コネクタを設定してデプロイします。コネクタは Twilio メッセージ情報を入力データとして受け取り、Twilio テキストメッセージをトリガーします。データフローを以下の図に示します。

コネクタを設定したら、Lambda 関数とサブスクリプションを作成します。

- この関数は、温度センサーからシミュレートされたデータを評価します。その後、Twilio メッセージ情報を条件付きで MQTT トピックに発行します。このトピックは、コネクタによってサブスクライブされます。

- このサブスクリプションでは、関数はトピックに発行でき、コネクタはトピックからデータを受信できます。

Twilio 通知コネクタには、Twilio API とやり取りするための Twilio 認証トークンが必要です。トークンは、 で作成 AWS Secrets Manager され、グループリソースから参照されるテキストタイプのシークレットです。これにより、 AWS IoT Greengrass は Greengrass コアにシークレットのローカルコピーを作成し、暗号化されてコネクタで使用できるようになります。詳細については、「Core にシークレットをデプロイする」を参照してください。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Secrets Manager シークレットを作成する
2. グループにシークレットリソースを追加する
3. グループにコネクタを追加する
4. Lambda 関数デプロイパッケージを作成する
5. 「Lambda 関数の作成」
6. グループに関数を追加する
7. サブスクリプションをグループに追加する
8. グループをデプロイする
9. the section called “ソリューションをテストする”

このチュートリアルは完了までに約 20 分かかります。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.9.3 以降)。Greengrass のグループまたは Core を作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。開始方法チュートリアルには、 AWS IoT Greengrass Core ソフトウェアのインストール手順も含まれています。
- Python 3.7 が AWS IoT Greengrass コアデバイスにインストールされている。
- AWS IoT Greengrass は、「シークレットの要件」で説明されているように、ローカルシークレットをサポートするように設定する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Twilio アカウントの SID、認証トークン、Twilio 対応の電話番号。Twilio プロジェクトを作成した後、これらの値をプロジェクトダッシュボードで使用できます。

> **Note**
>
> Twilio トライアルアカウントを使用できます。トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要があります。詳細については、「無料の Twilio トライアルアカウントを使用する方法」を参照してください。

ステップ 1: Secrets Manager シークレットを作成する

このステップでは、 AWS Secrets Manager コンソールを使用して Twilio 認証トークンのテキストタイプのシークレットを作成します。

1. AWS Secrets Manager コンソール にサインインします。

   > **Note**
   >
   > このプロセスの詳細については、「AWS Secrets Manager ユーザーガイド」の「ステップ 1: AWS Secrets Managerでシークレットを作成および保存する」を参照してください。

2. [新しいシークレットを保存] を選択します。
3. [Choose secret type] (シークレットの種類を選択) で、[Other type of secret] (他の種類のシークレット) を選択します。
4. [このシークレットに保存するキーと値のペアを指定します] の [プレーンテキスト] タブで、Twilio 認証トークンを入力します。すべての JSON 書式を削除し、トークンの値のみ入力します。

5. 暗号化には aws/secretsmanager を暗号キーとして選択した状態で、[Next] (次へ) を選択します。

> **Note**
>
> Secrets Manager がアカウントで作成するデフォルトの AWS マネージドキー AWS KMS を使用する場合、 は課金されません。

6. [シークレット名] に「**`greengrass-TwilioAuthToken`**」と入力し、[次へ] を選択します。

> **Note**
>
> デフォルトでは、Greengrass サービスロールは greengrass- で始まる名前のシークレットの値を取得 AWS IoT Greengrass できます。詳細については、「シークレットの要件」を参照してください。

7. このチュートリアルではローテーションは不要であるため、[disable automatic rotation] (自動ローテーションを無効化)、Next (次へ) の順に選択します。
8. [確認] ページで、設定を確認し、[保存] を選択します。

   次に、シークレットを参照する Greengrass グループにシークレットリソースを作成します。

ステップ 2: Greengrass グループにローカルシークレットリソースを追加する

このステップでは、シークレットリソースを Greengrass グループに追加します。このリソースは、前のステップで作成したシークレットへの参照です。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. シークレットリソースを追加するグループを選択します。
3. グループ設定ページの [Resources] (リソース) タブから、[Secrets] (シークレット) セクションを選択します。[Secrets] (シークレット) セクションには、グループに属するシークレットリソースが表示されます。このセクションからシークレットリソースを追加、編集、削除できます。

> **Note**
>
> 別の方法として、コネクタや Lambda 関数を設定するときに、コンソールでシークレットとシークレットリソースを作成することもできます。これはコネクタの [Configure parameters] (パラメータの設定) ページか、Lambda 関数の [Resources] (リソース) ページから実行できます。

4. [Secrets] (シークレット) セクションから [Add] (追加) を選択します。
5. [Add a secret resource] (シークレットリソースを追加) ページで、[Resource name] (リソース名)に **`MyTwilioAuthToken`** を入力します。
6. [Secret] (シークレット) には、[greengrass-TwilioAuthToken] を選択します。
7. [Select labels (Optional)] (ラベルを選択 (オプション)) セクションの、AWSCURRENT ステージングラベルはシークレットの最新バージョンを表します。このラベルはシークレットリソースに常に含まれています。

> **Note**
>
> このチュートリアルでは、AWSCURRENT ラベルのみが必要です。オプションで、Lambda 関数またはコネクタで必要になるラベルを含めることができます。

8. [Add resource] (リソースを追加) を選択します。

ステップ 3: Greengrass グループにコネクタを追加する

このステップでは、Twilio 通知コネクタのパラメータを設定し、グループに追加します。

1. グループ設定ページで、[Connectors] (コネクタ)、[Add a connector] (コネクタの追加) の順に選択します。
2. [Add connector] (コネクタの追加) ページで、[Twilio Notifications] (Twilio 通知) を選択します。
3. バージョンを選択します。
4. [Configuration] (設定) セクションで次を行います。
   - [Twilio auth token resource] (Twilio 認証トークンリソース) に、前のステップで作成したリソースを入力します。

> **Note**
>
> リソースを入力すると、[ARN of Twilio auth token secret] (Twilio 認証トークンシークレットの ARN) プロパティが設定されます。

- [Default from phone number] (電話番号のデフォルト) に Twilio 対応の電話番号を入力します。
- [Twilio account SID] (Twilio アカウント SID) に Twilio アカウント SID を入力します。

5. [リソースを追加] を選択します。

ステップ 4: Lambda 関数デプロイパッケージを作成する

Lambda 関数を作成するには、関数のコードと依存関係を含む Lambda 関数デプロイパッケージを最初に作成する必要があります。Greengrass Lambda 関数には、コア環境での MQTT メッセージとの通信やローカルシークレットへのアクセスなどのタスクに使用する、AWS IoT Greengrass Core SDK が必要です。このチュートリアルでは Python 関数を作成するため、デプロイパッケージには Python 版の SDK を使用します。

1. AWS IoT Greengrass Core SDK ダウンロードページから、 AWS IoT Greengrass Core SDK for Python をコンピュータにダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
3. 以下の Python コード関数を `temp_monitor.py` というローカルファイルに保存します。

```
import greengrasssdk
import json
import random

client = greengrasssdk.client('iot-data')

# publish to the Twilio Notifications connector through the twilio/txt topic
def function_handler(event, context):
    temp = event['temperature']

    # check the temperature
    # if greater than 30C, send a notification
    if temp > 30:
        data = build_request(event)
```

```
        client.publish(topic='twilio/txt', payload=json.dumps(data))
        print('published:' + str(data))

    print('temperature:' + str(temp))
    return

# build the Twilio request from the input data
def build_request(event):
    to_name = event['to_name']
    to_number = event['to_number']
    temp_report = 'temperature:' + str(event['temperature'])

    return {
        "request": {
            "recipient": {
                "name": to_name,
                "phone_number": to_number,
                "message": temp_report
            }
        },
        "id": "request_" + str(random.randint(1,101))
    }
```

4. 以下の項目を `temp_monitor_python.zip` という名前のファイルに圧縮します。ZIP ファイルを作成するときは、コードおよび依存関係のみを含め、フォルダは含めません。

   - temp_monitor.py。アプリケーションロジック。
   - greengrasssdk。MQTT メッセージを発行する Python Greengrass Lambda 関数で必須のライブラリ。

   これが Lambda 関数デプロイパッケージです。

ここでは、デプロイパッケージを使用する Lambda 関数を作成します。

ステップ 5: AWS Lambda コンソールで Lambda 関数を作成する

このステップでは、 AWS Lambda コンソールを使用して Lambda 関数を作成し、デプロイパッケージを使用するように設定します。次に、関数のバージョンを公開し、エイリアスを作成します。

1. 最初に Lambda 関数を作成します。

a. で AWS Management Console、サービスを選択し、 AWS Lambda コンソールを開きます。

b. [Create function] (関数の作成) を選択し、[Author from scratch] (一から作成) を選択します。

c. [基本的な情報] セクションで、以下の値を使用します。

- [関数名] に「**`TempMonitor`**」と入力します。
- [ランタイム] で [Python 3.7] を選択します。
- [アクセス許可] はデフォルト設定のままにしておきます。これで Lambda への基本的なアクセス許可を付与する実行ロールが作成されます。このロールは では使用されません AWS IoT Greengrass。

d. ページの下部で、[関数の作成] を選択します。

2. 今度は、ハンドラを登録し、Lambda 関数デプロイパッケージをアップロードします。

a. [Code] (コード) タブの [Code source] (コードソース) で、[Upload from] (アップロード元) を選択します。ドロップダウンから [.zip ファイル] を選択します。

b. [アップロード] を選択し、`temp_monitor_python.zip` デプロイパッケージを選択します。次に、[保存] を選択します。

c. 関数の [Code] (コード) タブにある [Runtime settings] (ランタイム設定) で [Edit] (編集) を選択し、次の値を入力します。

- [ランタイム] で [Python 3.7] を選択します。
- [ハンドラ] に **`temp_monitor.function_handler`** と入力します。

d. [Save] を選択します。

> **Note**
>
> AWS Lambda コンソールのテストボタンは、この関数では機能しません。 AWS IoT Greengrass Core SDK には、Greengrass Lambda 関数を AWS Lambda コン

ソールで個別に実行するために必要なモジュールは含まれていません。これらのモジュール (例えば `greengrass_common`) が関数に提供されるのは、Greengrass Core にデプロイされた後になります。

3. ここで、Lambda 関数の最初のバージョンを公開し、バージョンのエイリアスを作成します。

> **Note**
>
> Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

   a. [アクション] メニューから、[新しいバージョンを発行] を選択します。
   b. [バージョンの説明] に **`First version`** と入力し、[発行] を選択します。
   c. [TempMonitor: 1] 設定ページで、[Actions (アクション)] メニューの [エイリアスの作成] を選択します。
   d. [新しいエイリアスの作成] ページで、次の値を使用します。
      - [名前] に **`GG_TempMonitor`** と入力します。
      - [バージョン] で、[1] を選択します。

      > **Note**
      >
      > AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

   e. [Create] (作成) を選択します。

これで、Greengrass グループに Lambda 関数を追加する準備ができました。

ステップ 6: Greengrass グループに Lambda 関数を追加する

このステップでは、Lambda 関数をグループに追加し、そのライフサイクルと環境変数を設定します。詳細については、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。

1. グループ設定ページで、[Lambda functions] (Lambda 関数) タブを選択します。
2. [My Lambda functions] (自分の Lambda 関数) で、[Add] を選択します。
3. [Add Lambda function] (Lambda 関数を追加) ページで、Lambda 関数として [TempMonitor] を選択します。
4. Lambda 関数のバージョンで、Alias: GG_TempMonitor を選択します。
5. [Add Lambda function] (Lambda 関数の追加) を選択します。

ステップ 7: サブスクリプションを Greengrass グループに追加する

このステップでは、Lambda 関数にコネクタへの入力データの送信を許可するサブスクリプションを追加します。コネクタは、サブスクライブする MQTT トピックを定義しているため、このサブスクリプションはそのいずれかのトピックを使用します。これは、サンプル関数が発行するのと同じトピックです。

このチュートリアルでは、 関数が から AWS IoT シミュレートされた温度測定値を受信し、 AWS IoT がコネクタからステータス情報を受信できるようにするサブスクリプションも作成します。

1. グループ設定ページの [Subscriptions] (サブスクリプション) タブで、[Add Subscription] (サブスクリプションの追加) を選択します。
2. [Create a subscription] (サブスクリプションの作成) ページで、ソースおよびターゲットを次のように設定します。

   a. [Source type] (ソースタイプ) は、[Lambda function] (Lambda 関数)、[TempMonitor] の順に選択します。

   b. [Target type] (ターゲットタイプ) は、[Connector] (コネクタ)、[Twilio Notifications] (Twilio 通知) の順に選択します。
3. [Topic filter] (トピックのフィルター) に **`twilio/txt`** を選択します。
4. [Create subscription] を選択します。
5. ステップ 1～4 を繰り返して、 が 関数 AWS IoT にメッセージを発行できるようにするサブスクリプションを作成します。

a. [Source type] (ソースタイプ) は、Service (サービス)、IoT Cloud (IoT Cloud (IoT クラウド)) の順に選択します。

b. [Select a target] (ターゲットの選択) は、[Lambda function] (Lambda 関数)、[TempMonitor] の順に選択します。

c. [トピックのフィルター] に「**`temperature/input`**」と入力します。

6. ステップ 1〜4 を繰り返して、 AWS IoTへのメッセージの発行をコネクタに許可するサブスクリプションを作成します。

   a. [Source type] (ソースタイプ) は、[Connectors] (コネクタ)、[Twilio Notifications] (Twilio 通知) の順に選択します。

   b. [Target type] (ターゲットタイプ) は、[サービス]、[IoT Cloud] (IoT クラウド) の順に選択します。

   c. [トピックフィルター] には自動的に「**`twilio/message/status`**」と入力されます。これは、コネクタが発行する先の事前定義済みトピックです。

ステップ 8: Greengrass グループをデプロイする

Core デバイスにグループをデプロイします。

1. AWS IoT Greengrass コアが実行されていることを確認します。必要に応じて、Raspberry Pi のターミナルで以下のコマンドを実行します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   出力に `root` で実行中の `/greengrass/ggc/packages/`*`ggc-version`*`/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

   > **Note**
   > パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

   b. デーモンを開始するには、以下を実行します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. グループ設定ページで、[Deploy] (デプロイ) を選択します。
3. a. [Lambda functions] (Lambda 関数) タブの [System Lambda functions] (システム Lambda 関数) セクションで、[IP detector] (IP ディテクター)、[Edit] (編集) の順に選択します。

   b. [IP ディテクターの設定を編集] ダイアログボックスで、[MQTT ブローカーエンドポイントを自動的に検出して上書きする] を選択します。

   c. [Save] を選択します。

   これにより、デバイスは、IP アドレス、DNS、ポート番号など、コアの接続情報を自動的に取得できます。自動検出が推奨されますが、手動で指定されたエンドポイント AWS IoT Greengrass もサポートされます。グループが初めてデプロイされたときにのみ、検出方法の確認が求められます。

   > **Note**
   >
   > プロンプトが表示されたら、Greengrass サービスロールを作成し、それを現在の AWS アカウント の に関連付けるアクセス許可を付与します AWS リージョン。このロールにより AWS IoT Greengrass 、 は AWS サービスのリソースにアクセスできます。

   [デプロイ] ページには、デプロイのタイムスタンプ、バージョン ID、ステータスが表示されます。完了すると、デプロイのステータスが [完了] と表示されます。

   トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

> **Note**
>
> Greengrass グループには、一度に 1 つのバージョンのコネクタしか含めることができません。コネクタのバージョンのアップグレードについては、「the section called “コネクタのバージョンのアップグレード”」を参照してください。

ソリューションをテストする

1. AWS IoT コンソールのホームページで、テストを選択します。
2. [Subscribe to topic] (トピックへのサブスクライブ) で、以下の値を使用し、[Subscribe] (サブスクリプション) を選択します。Twilio 通知コネクタはこのトピックにステータス情報を発行します。

プロパティ	値
トピックのサブスクリプション	twilio/message/status
MQTT ペイロード表示	文字列としてペイロードを表示

3. [Publish to topic] (トピックに発行) で、以下の値を使用し、[Publish] (発行) を選択して関数を呼び出します。

プロパティ	値
トピック	temperature/input

メッセージ	*recipient-name* は名前に、*recipient-phone-number* はテキストメッセージの受取人の電話番号に置き換えます。例: +12345000000
	
	{
	"to_name": " *recipient-name* ",
	"to_number": " *recipient-phone-number* ",
	"temperature": 31
	}
	
	トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要があります。詳細については、「個人の電話番号を認証する」を参照してください。

正しく実行できていれば、受信者にテキストメッセージが届き、コンソールの出力データに `success` のステータスが表示されます。

ここでは、入力メッセージの `temperature` を **29** に変更して発行します。これは 30 未満であるため、TempMonitor 関数は Twilio メッセージをトリガーしません。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “AWSが提供する Greengrass コネクタ”

Greengrass コネクタの開始方法 (CLI)

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

このチュートリアルでは、 AWS CLI を使用してコネクタを操作する方法を示します。

コネクタを使用して、開発ライフサイクルを短縮します。コネクタは、サービス、プロトコル、リソースとのやり取りを簡略する、あらかじめ組み込まれた再利用可能なモジュールです。それらのモジュールは、Greengrass デバイスにビジネスロジックをよりすばやくデプロイするのに役立ちます。詳細については、「コネクタを使用してサービスおよびプロトコルと統合する」を参照してください。

このチュートリアルでは、Twilio 通知コネクタを設定してデプロイします。コネクタは Twilio メッセージ情報を入力データとして受け取り、Twilio テキストメッセージをトリガーします。データフローを以下の図に示します。

コネクタを設定したら、Lambda 関数とサブスクリプションを作成します。

- この関数は、温度センサーからシミュレートされたデータを評価します。その後、Twilio メッセージ情報を条件付きで MQTT トピックに発行します。このトピックは、コネクタによってサブスクライブされます。
- このサブスクリプションでは、関数はトピックに発行でき、コネクタはトピックからデータを受信できます。

Twilio 通知コネクタには、Twilio API とやり取りするための Twilio 認証トークンが必要です。トークンは、 で作成 AWS Secrets Manager され、グループリソースから参照されるテキストタイプのシークレットです。これにより、 AWS IoT Greengrass は Greengrass コアにシークレットのローカルコピーを作成し、暗号化されてコネクタで使用できるようになります。詳細については、「Core にシークレットをデプロイする」を参照してください。

このチュートリアルのおおまかな手順は以下のとおりです。

1. Secrets Manager シークレットを作成する
2. リソースの定義とバージョンを作成する
3. コネクタの定義とバージョンを作成する
4. Lambda 関数デプロイパッケージを作成する
5. 「Lambda 関数の作成」
6. 関数の定義とバージョンを作成する
7. サブスクリプションの定義とバージョンを作成する

8. グループバージョンを作成する
9. デプロイを作成する
10. the section called “ソリューションをテストする”

このチュートリアルは完了までに約 30 分かかります。

AWS IoT Greengrass API の使用

以下のパターンを理解しておくと、Greengrass のグループとグループコンポーネント (グループのコネクタ、関数、リソースなど) を操作するときに役立ちます。

- 階層の最上部で、コンポーネントには definition オブジェクトがあり、definition は version オブジェクトのコンテナです。同様に、version は、コネクタ、関数、などのコンポーネントタイプのコンテナです。
- Greengrass Core にデプロイするときは、特定のグループバージョンをデプロイします。グループバージョンには、各タイプのコンポーネントの 1 つのバージョンを含めることができます。Core は必須ですが、その他はオプションです。
- バージョンは不変なので、変更するときは新しいバージョンを作成する必要があります。

> **Tip**
> AWS CLI コマンドの実行時にエラーが発生した場合は、 `--debug`パラメータを追加し、コマンドを再実行してエラーに関する詳細情報を取得します。

AWS IoT Greengrass API を使用すると、コンポーネントタイプに対して複数の定義を作成できます。例えば、`FunctionDefinitionVersion` を作成するたびに `FunctionDefinition` オブジェクトを作成することも、既存の定義に新しいバージョンを追加することもできます。この柔軟性により、バージョニングシステムをカスタマイズできます。

前提条件

このチュートリアルを完了するには、以下が必要です。

- Greengrass グループと Greengrass コア (v1.9.3 以降)。Greengrass のグループまたは Core を作成する方法については、「の開始方法 AWS IoT Greengrass」を参照してください。開始方法

チュートリアルには、 AWS IoT Greengrass Core ソフトウェアのインストール手順も含まれています。

- Python 3.7 が AWS IoT Greengrass コアデバイスにインストールされている。
- AWS IoT Greengrass は、「シークレットの要件」で説明されているように、ローカルシークレットをサポートするように設定する必要があります。

> **Note**
>
> この要件には、Secrets Manager シークレットへのアクセス許可が含まれます。デフォルトの Greengrass サービスロールを使用している場合、Greengrass は greengrass- で始まる名前の付いたシークレット値にアクセスできます。

- Twilio アカウントの SID、認証トークン、Twilio 対応の電話番号。Twilio プロジェクトを作成した後、これらの値をプロジェクトダッシュボードで使用できます。

> **Note**
>
> Twilio トライアルアカウントを使用できます。トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要があります。詳細については、「無料の Twilio トライアルアカウントを使用する方法」を参照してください。

- AWS CLI がコンピュータにインストールおよび設定されている。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS Command Line Interfaceのインストール」および「AWS CLIの設定」を参照してください。

このチュートリアルの例は、Linux やその他の Unix ベースのシステム向けに書かれています。Windows を使用している場合、構文の違いについては、「AWS Command Line Interfaceのパラメータ値の指定」を参照してください。

コマンドに JSON 文字列が含まれている場合、このチュートリアルでは、1 行形式の JSON の例を示しています。システムによっては、この形式を使用したほうが、コマンドの編集と実行が容易になる場合があります。

ステップ 1: Secrets Manager シークレットを作成する

このステップでは、 AWS Secrets Manager API を使用して Twilio 認証トークンのシークレットを作成します。

1. まず、シークレットを作成します。

   - *twilio-auth-token* を Twilio 認証トークンに置き換えます。

```
aws secretsmanager create-secret --name greengrass-TwilioAuthToken --secret-
string twilio-auth-token
```

> **Note**
>
> デフォルトでは、Greengrass サービスロールにより AWS IoT Greengrass 、 は greengrass- で始まる名前のシークレットの値を取得できます。詳細については、「シークレットの要件」を参照してください。

2. 出力からシークレットの ARN をコピーします。この情報は、シークレットリソースの作成時と Twilio 通知コネクタの設定時に使用します。

ステップ 2: リソースの定義とバージョンを作成する

このステップでは、 AWS IoT Greengrass API を使用して Secrets Manager シークレットのシークレットリソースを作成します。

1. 初期バージョンを含むリソース定義を作成します。

   - *secret-arn* を、前のステップでコピーしたシークレットの ARN に置き換えます。

JSON Expanded

```
aws greengrass create-resource-definition --name MyGreengrassResources --
initial-version '{
    "Resources": [
        {
```

```
            "Id": "TwilioAuthToken",
            "Name": "MyTwilioAuthToken",
            "ResourceDataContainer": {
                "SecretsManagerSecretResourceData": {
                    "ARN": "secret-arn"
                }
            }
        }
    ]
}'
```

JSON Single-line

```
aws greengrass create-resource-definition \
--name MyGreengrassResources \
--initial-version '{"Resources": [{"Id": "TwilioAuthToken",
 "Name": "MyTwilioAuthToken", "ResourceDataContainer":
 {"SecretsManagerSecretResourceData": {"ARN": "secret-arn"}}}]}'
```

2. 出力からリソース定義の `LatestVersionArn` をコピーします。この値を使用して、Core にデプロイするグループバージョンに、リソース定義バージョンを追加します。

ステップ 3: コネクタの定義とバージョンを作成する

このステップでは、Twilio 通知コネクタのパラメータを設定します。

1. コネクタの定義と初期バージョンを作成します。

   - *account-sid* を Twilio アカウントの SID に置き換えます。
   - *secret-arn* を Secrets Manager シークレットの ARN に置き換えます。コネクタではこの ARN を使用してローカルシークレットの値を取得します。
   - *phone-number* を Twilio 対応の電話番号に置き換えます。Twilio はこの番号を使用してテキストメッセージを開始します。このメッセージは入力メッセージのペイロードで上書きできます。形式は以下のようになります: +19999999999

JSON Expanded

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --
initial-version '{
    "Connectors": [
        {
            "Id": "MyTwilioNotificationsConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/
TwilioNotifications/versions/4",
            "Parameters": {
                "TWILIO_ACCOUNT_SID": "account-sid",
                "TwilioAuthTokenSecretArn": "secret-arn",
                "TwilioAuthTokenSecretArn-ResourceId": "TwilioAuthToken",
                "DefaultFromPhoneNumber": "phone-number"
            }
        }
    ]
}'
```

JSON Single-line

```
aws greengrass create-connector-definition \
--name MyGreengrassConnectors \
--initial-version '{"Connectors": [{"Id": "MyTwilioNotificationsConnector",
 "ConnectorArn": "arn:aws:greengrass:region::/connectors/TwilioNotifications/
versions/4", "Parameters": {"TWILIO_ACCOUNT_SID": "account-sid",
 "TwilioAuthTokenSecretArn": "secret-arn", "TwilioAuthTokenSecretArn-
ResourceId": "TwilioAuthToken", "DefaultFromPhoneNumber": "phone-number"}}]}'
```

> **Note**
>
> `TwilioAuthToken` は、前のステップでシークレットリソースの作成に使用した ID です。

2. 出力からコネクタ定義の `LatestVersionArn` をコピーします。この値を使用して、Core にデプロイするグループバージョンに、コネクタ定義バージョンを追加します。

ステップ 4: Lambda 関数デプロイパッケージを作成する

Lambda 関数を作成するには、関数のコードと依存関係を含む Lambda 関数デプロイパッケージを最初に作成する必要があります。Greengrass Lambda 関数には、コア環境での MQTT メッセージとの通信やローカルシークレットへのアクセスなどのタスクに使用する、AWS IoT Greengrass Core SDK が必要です。このチュートリアルでは Python 関数を作成するため、デプロイパッケージには Python 版の SDK を使用します。

1. AWS IoT Greengrass Core SDK ダウンロードページから、 AWS IoT Greengrass Core SDK for Python をコンピュータにダウンロードします。
2. ダウンロードしたパッケージを解凍し、SDK を取得します。SDK は `greengrasssdk` フォルダです。
3. 以下の Python コード関数を `temp_monitor.py` というローカルファイルに保存します。

```
import greengrasssdk
import json
import random

client = greengrasssdk.client('iot-data')

# publish to the Twilio Notifications connector through the twilio/txt topic
def function_handler(event, context):
    temp = event['temperature']

    # check the temperature
    # if greater than 30C, send a notification
    if temp > 30:
        data = build_request(event)
        client.publish(topic='twilio/txt', payload=json.dumps(data))
        print('published:' + str(data))

    print('temperature:' + str(temp))
    return

# build the Twilio request from the input data
def build_request(event):
    to_name = event['to_name']
    to_number = event['to_number']
    temp_report = 'temperature:' + str(event['temperature'])

    return {
```

```
        "request": {
            "recipient": {
                "name": to_name,
                "phone_number": to_number,
                "message": temp_report
            }
        },
        "id": "request_" + str(random.randint(1,101))
    }
```

4. 以下の項目を `temp_monitor_python.zip` という名前のファイルに圧縮します。ZIP ファイルを作成するときは、コードおよび依存関係のみを含め、フォルダは含めません。

   - temp_monitor.py。アプリケーションロジック。
   - greengrasssdk。MQTT メッセージを発行する Python Greengrass Lambda 関数で必須のライブラリ。

   これが Lambda 関数デプロイパッケージです。

ステップ 5: Lambda 関数を作成する

ここでは、デプロイパッケージを使用する Lambda 関数を作成します。

1. 関数の作成時に ARN を渡せるように、IAM ロールを作成します。

   JSON Expanded

```
aws iam create-role --role-name Lambda_empty --assume-role-policy '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
           "Action": "sts:AssumeRole"
        }
    ]
}'
```

JSON Single-line

```
aws iam create-role --role-name Lambda_empty --assume-role-policy '{"Version":
 "2012-10-17", "Statement": [{"Effect": "Allow", "Principal": {"Service":
 "lambda.amazonaws.com"},"Action": "sts:AssumeRole"}]}'
```

> **Note**
>
> AWS IoT Greengrass Greengrass Lambda 関数のアクセス許可は Greengrass グループロールで指定されているため、 はこのロールを使用しません。このチュートリアルでは、空のロールを作成します。

2. 出力から `Arn` をコピーします。
3. AWS Lambda API を使用して TempMonitor 関数を作成します。以下のコマンドでは、zip ファイルが現在のディレクトリにあるとします。

   - *`role-arn`* を、コピーした `Arn` に置き換えます。

```
aws lambda create-function \
--function-name TempMonitor \
--zip-file fileb://temp_monitor_python.zip \
--role role-arn \
--handler temp_monitor.function_handler \
--runtime python3.7
```

4. 関数のバージョンを発行します。

```
aws lambda publish-version --function-name TempMonitor --description 'First
 version'
```

5. 発行されるバージョンのエイリアスを作成します。

   Greengrass グループは、Lambda 関数をエイリアス別 (推奨) またはバージョン別に参照できます。エイリアスを使用すると、関数コードを更新する時にサブスクリプションテーブルやグループ定義を変更する必要がないため、コード更新を簡単に管理できます。その代わりに、新しい関数バージョンにエイリアスを指定するだけで済みます。

> **Note**
>
> AWS IoT Greengrass は、$LATEST バージョンの Lambda エイリアスをサポートしていません。

```
aws lambda create-alias --function-name TempMonitor --name GG_TempMonitor --
function-version 1
```

6. 出力から `AliasArn` をコピーします。この値は、 の 関数を設定するとき AWS IoT Greengrass と、サブスクリプションを作成するときに使用します。

これで、 関数を設定する準備が整いました AWS IoT Greengrass。

ステップ 6: 関数の定義とバージョンを作成する

AWS IoT Greengrass コアで Lambda 関数を使用するには、エイリアスで Lambda 関数を参照し、グループレベルの設定を定義する関数定義バージョンを作成します。詳細については、「the section called “Greengrass Lambda 関数の実行の制御”」を参照してください。

1. 初期バージョンを含む関数定義を作成します。

   - *alias-arn* を、エイリアスの作成時にコピーした `AliasArn` に置き換えます。

   JSON Expanded

```
aws greengrass create-function-definition --name MyGreengrassFunctions --
initial-version '{
    "Functions": [
        {
            "Id": "TempMonitorFunction",
            "FunctionArn": "alias-arn",
            "FunctionConfiguration": {
                "Executable": "temp_monitor.function_handler",
                "MemorySize": 16000,
                "Timeout": 5
            }
```

```
        }
    ]
}'
```

JSON Single-line

```
aws greengrass create-function-definition \
--name MyGreengrassFunctions \
--initial-version '{"Functions": [{"Id": "TempMonitorFunction",
 "FunctionArn": "alias-arn", "FunctionConfiguration": {"Executable":
 "temp_monitor.function_handler", "MemorySize": 16000,"Timeout": 5}}]}'
```

2. 出力から `LatestVersionArn` をコピーします。この値を使用して、Core にデプロイするグループバージョンに、関数定義バージョンを追加します。
3. 出力から `Id` をコピーします。この値は、後でその関数を更新するときに使用します。

ステップ 7: サブスクリプションの定義とバージョンを作成する

このステップでは、Lambda 関数にコネクタへの入力データの送信を許可するサブスクリプションを追加します。コネクタは、サブスクライブする MQTT トピックを定義しているため、このサブスクリプションはそのいずれかのトピックを使用します。これは、サンプル関数が発行するのと同じトピックです。

このチュートリアルでは、 関数が から AWS IoT シミュレートされた温度測定値を受信し、 コネクタからステータス情報 AWS IoT を受信できるようにするサブスクリプションも作成します。

1. サブスクリプションを含む初期バージョンでサブスクリプション定義を作成します。

   - *alias-arn* を、関数のエイリアスの作成時にコピーした `AliasArn` に置き換えます。この ARN を両方のサブスクリプションに使用します。

JSON Expanded

```
aws greengrass create-subscription-definition --initial-version '{
    "Subscriptions": [
        {
            "Id": "TriggerNotification",
            "Source": "alias-arn",
```

```
            "Subject": "twilio/txt",
            "Target": "arn:aws:greengrass:region::/connectors/
TwilioNotifications/versions/4"
        },
        {
            "Id": "TemperatureInput",
            "Source": "cloud",
            "Subject": "temperature/input",
            "Target": "alias-arn"
        },
        {
            "Id": "OutputStatus",
            "Source": "arn:aws:greengrass:region::/connectors/
TwilioNotifications/versions/4",
            "Subject": "twilio/message/status",
            "Target": "cloud"
        }
    ]
}'
```

JSON Single-line

```
aws greengrass create-subscription-definition \
--initial-version '{"Subscriptions": [{"Id": "TriggerNotification", "Source":
 "alias-arn", "Subject": "twilio/txt", "Target": "arn:aws:greengrass:region::/
connectors/TwilioNotifications/versions/4"},{"Id": "TemperatureInput",
 "Source": "cloud", "Subject": "temperature/input", "Target": "alias-arn"},
{"Id": "OutputStatus", "Source": "arn:aws:greengrass:region::/connectors/
TwilioNotifications/versions/4", "Subject": "twilio/message/status", "Target":
 "cloud"}]}'
```

2. 出力から `LatestVersionArn` をコピーします。この値を使用して、Core にデプロイするグループバージョンに、サブスクリプション定義バージョンを追加します。

ステップ 8: グループバージョンを作成する

デプロイするすべての項目を含むグループバージョンを作成する準備が整いました。ここでは、各コンポーネントタイプのターゲットバージョンを参照するグループバージョンを作成します。

まず、Core 定義バージョンのグループ ID と ARN を取得します。これらの値は、グループバージョンを作成するために必要です。

1. グループ ID と最新のグループバージョンを取得します。

   a. ターゲットの Greengrass グループとグループバージョンの ID を取得します。この手順では、これが最新のグループおよびグループのバージョンであると仮定します。次のクエリは、最後に作成されたグループを返します。

   ```
   aws greengrass list-groups --query "reverse(sort_by(Groups,
    &CreationTimestamp))[0]"
   ```

   または、名前でクエリを実行することもできます。グループ名は一意である必要はないため、複数のグループが返されることがあります。

   ```
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ```

   > **Note**
   > これらの値は、 AWS IoT コンソールでも確認できます。グループ ID は、グループの [設定] ページに表示されます。グループバージョン ID は、グループの [Deployments] (デプロイ) タブに表示されます。

   b. 出力からターゲットグループの `Id` をコピーします。この情報は、コア定義バージョンを取得するときとグループをデプロイするときに使用します。

   c. 出力から `LatestVersion` をコピーします。これは、グループに追加された最後のバージョンの ID です。この情報は、コア定義バージョンを取得するときに使用します。

2. コア定義バージョンの ARN を取得します。

   a. グループバージョンを取得します。このステップでは、最新のグループバージョンに Core 定義バージョンが含まれているものとします。

   - *group-id* は、コピーしたグループの `Id` に置き換えます。
   - *group-version-id* は、コピーしたグループの `LatestVersion` に置き換えます。

   ```
   aws greengrass get-group-version \
   --group-id group-id \
   --group-version-id group-version-id
   ```

   b. 出力から `CoreDefinitionVersionArn` をコピーします。

3. グループバージョンを作成します。

   - *group-id* は、コピーしたグループの Id に置き換えます。
   - *core-definition-version-arn* は、コピーしたコア定義バージョンの CoreDefinitionVersionArn に置き換えます。
   - *resource-definition-version-arn* を、リソース定義のコピー済み LatestVersionArn に置き換えます。
   - *connector-definition-version-arn* を、コネクタ定義のコピー済み LatestVersionArn に置き換えます。
   - *function-definition-version-arn* を、関数定義のコピー済み LatestVersionArn に置き換えます。
   - *subscription-definition-version-arn* を、サブスクリプション定義のコピー済み LatestVersionArn に置き換えます。

```
aws greengrass create-group-version \
--group-id group-id \
--core-definition-version-arn core-definition-version-arn \
--resource-definition-version-arn resource-definition-version-arn \
--connector-definition-version-arn connector-definition-version-arn \
--function-definition-version-arn function-definition-version-arn \
--subscription-definition-version-arn subscription-definition-version-arn
```

4. 出力から Version の値をコピーします。これはグループバージョンの ID です。この値を使用して、グループバージョンをデプロイします。

ステップ 9: デプロイを作成する

Core デバイスにグループをデプロイします。

1. コアデバイスターミナルで、 AWS IoT Greengrass デーモンが実行されていることを確認します。

   a. デーモンが実行中かどうかを確認するには、以下を実行します。

```
ps aux | grep -E 'greengrass.*daemon'
```

出力に root で実行中の /greengrass/ggc/packages/1.11.6/bin/daemon のエントリが含まれていれば、デーモンは実行されています。

b. 次のようにしてデーモンを開始します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

2. デプロイを作成します。

- *group-id* は、コピーしたグループの Id に置き換えます。
- *group-version-id* は、コピーした新しいグループバージョンの Version に置き換えます。

```
aws greengrass create-deployment \
--deployment-type NewDeployment \
--group-id group-id \
--group-version-id group-version-id
```

3. 出力から DeploymentId をコピーします。
4. デプロイのステータスを取得します。

- *group-id* は、コピーしたグループの Id に置き換えます。
- *deployment-id* は、コピーしたデプロイの DeploymentId に置き換えます。

```
aws greengrass get-deployment-status \
--group-id group-id \
--deployment-id deployment-id
```

ステータスが Success の場合、デプロイは成功しています。トラブルシューティングのヘルプについては、「トラブルシューティング」を参照してください。

ソリューションをテストする

1. AWS IoT コンソールのホームページで、テストを選択します。

2. [Subscribe to topic] (トピックへのサブスクライブ) で、以下の値を使用し、[Subscribe] (サブスクリプション) を選択します。Twilio 通知コネクタはこのトピックにステータス情報を発行します。

プロパティ	値
トピックのサブスクリプション	twilio/message/status
MQTT ペイロード表示	文字列としてペイロードを表示

3. [Publish to topic] (トピックに発行) で、以下の値を使用し、[Publish] (発行) を選択して関数を呼び出します。

プロパティ	値
トピック	temperature/input

メッセージ	*recipient-name* は名前に、*recipient-phone-number* はテキストメッセージの受取人の電話番号に置き換えます。例: `+12345000000`
	
	`{`
	`"to_name": " recipient-name ",`
	`"to_number": " recipient-phone-number ",`
	`"temperature": 31`
	`}`
	
	トライアルアカウントを使用している場合は、Twilio 以外の受信者の電話番号を、確認済みの電話番号リストに追加する必要があります。詳細については、「個人の電話番号を認証する」を参照してください。

正しく実行できていれば、受信者にテキストメッセージが届き、コンソールの出力データに success のステータスが表示されます。

ここでは、入力メッセージの `temperature` を **29** に変更して発行します。これは 30 未満であるため、TempMonitor 関数は Twilio メッセージをトリガーしません。

関連情報

- コネクタを使用してサービスおよびプロトコルと統合する
- the section called “AWSが提供する Greengrass コネクタ”
- the section called “コネクタの使用を開始する (コンソール)”
- コマンドリファレンスAWS Secrets Managerの コマンド AWS CLI
- コマンドリファレンスの AWS Identity and Access Management (IAM) コマンド AWS CLI
- コマンドリファレンスAWS Lambdaの コマンド AWS CLI
- コマンドリファレンスAWS IoT Greengrassの コマンド AWS CLI

Greengrass Discovery RESTful API

AWS IoT Greengrass コアと通信するすべてのクライアントデバイスは、Greengrass グループのメンバーである必要があります。各グループには Greengrass コアが必要です。Discovery API により、デバイスは、クライアントデバイスと同じ Greengrass グループ内の Greengrass コアに接続するために必要な情報を取得します。クライアントデバイスが最初にオンラインになると、 AWS IoT Greengrass サービスに接続し、Discovery API を使用して以下を検索できます。

- デバイスが属しているグループ。クライアントデバイスは、最大 10 個のグループのメンバーにすることができます。
- グループ内の Greengrass コアの IP アドレスとポート。
- Greengrass コアデバイスを認証するために使用できる、グループ CA 証明書。

> **Note**
>
> クライアントデバイスは、 AWS IoT Device SDKsを使用して Greengrass コアの接続情報を検出することもできます。詳細については、「AWS IoT デバイス SDK」を参照してください。

この API を使用するには、検出 API エンドポイントに HTTP リクエストを送信します。以下に例を示します。

```
https://greengrass-ats.iot.region.amazonaws.com:port/greengrass/discover/thing/thing-name
```

AWS IoT Greengrass Discovery API でサポートされている Amazon Web Services リージョンとエンドポイントのリストについては、 の「 AWS IoT Greengrass エンドポイントとクォータ」を参照してくださいAWS 全般のリファレンス。これはデータプレーンのみの API です。グループ管理と AWS IoT Core オペレーションのエンドポイントは、Discovery API エンドポイントとは異なります。

リクエスト

次の例で示すように、リクエストにはスタンダードな HTTP ヘッダーが含まれ、Greengrass 検出エンドポイントに送信されます。

ポート番号は、コアがポート 8443 またはポート 443 のどちらで HTTPS トラフィックを送信するように設定されているかによって異なります。詳細については、「the section called “ポート 443 での接続またはネットワークプロキシを通じた接続”」を参照してください。

ポート 8443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:8443/greengrass/discover/
thing/thing-name
```

ポート 443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:443/greengrass/discover/
thing/thing-name
```

ポート 443 に接続するクライアントは、Application Layer Protocol Negotiation (ALPN) の TLS 拡張機能を実装するとともに、`x-amzn-http-ca` を `ProtocolName` として `ProtocolNameList` に渡す必要があります。詳細については、「AWS IoT デベロッパーガイド」の「プロトコル」を参照してください。

> **Note**
>
> この例では、ATS ルート CA 証明書 (推奨) で使用される Amazon Trust Services (ATS) エンドポイントを使用します。エンドポイントはルート CA 証明書タイプと一致する必要があります。詳細については、「the section called “サービスエンドポイントは証明書タイプと一致する必要がある”」を参照してください。

レスポンス

成功した場合、レスポンスには標準の HTTP ヘッダーに加えて、以下のコードと本体が含まれます。

```
HTTP 200
BODY: response document
```

詳細については、「検出レスポンスドキュメントの例」を参照してください。

検出の認可

接続情報を取得するには、呼び出し元に `greengrass:Discover` アクションの実行を許可するポリシーが必要です。クライアント証明書による TLS 相互認証が、許可される唯一の認証形式です。呼び出し元がこのアクションを実行できるポリシー例を次に示します。

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "greengrass:Discover",
        "Resource": ["arn:aws:iot:us-west-2:123456789012:thing/MyThingName"]
     }]
}
```

検出レスポンスドキュメントの例

次のドキュメントは、1 つの Greengrass コア、1 つのエンドポイント、および 1 つのグループ CA 証明書を持つグループのメンバーであるクライアントデバイス用のレスポンスを示しています。

```
{
  "GGGroups": [
    {
      "GGGroupId": "gg-group-01-id",
      "Cores": [
        {
          "thingArn": "core-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-01-connection-id",
              "hostAddress": "core-01-address",
              "portNumber": core-01-port,
              "metadata": "core-01-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    }
```

```
  ]
}
```

次のドキュメントは、1 つの Greengrass コア、複数のエンドポイント、および複数のグループ CA 証明書を持つ 2 つのグループのメンバーであるクライアントデバイス用のレスポンスを示しています。

```
{
  "GGGroups": [
    {
      "GGGroupId": "gg-group-01-id",
      "Cores": [
        {
          "thingArn": "core-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-01-connection-id",
              "hostAddress": "core-01-address",
              "portNumber": core-01-port,
              "metadata": "core-01-connection-1-description"
            },
            {
              "id": "core-01-connection-id-2",
              "hostAddress": "core-01-address-2",
              "portNumber": core-01-port-2,
              "metadata": "core-01-connection-2-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    },
    {
      "GGGroupId": "gg-group-02-id",
      "Cores": [
         {
            "thingArn":"core-02-thing-arn",
            "Connectivity" : [
            {
```

```
                "id": "core-02-connection-id",
                "hostAddress": "core-02-address",
                "portNumber": core-02-port,
                "metadata": "core-02-connection-1-description"
            }
            ],
            "CAs": [
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
            ]
        }
    ]
    }
}
```

> **Note**
>
> Greengrass グループには、厳密に 1 つの Greengrass コアを定義する必要があります。Greengrass コアのリストを含む AWS IoT Greengrass サービスからのレスポンスには、Greengrass コアが 1 つだけ含まれます。

cURL をインストールしている場合は、検出リクエストをテストできます。例:

```
$ curl --cert 1a23bc4d56.cert.pem --key 1a23bc4d56.private.key https://greengrass-
ats.iot.us-west-2.amazonaws.com:8443/greengrass/discover/thing/MyDevice
{"GGGroups":[{"GGGroupId":"1234a5b6-78cd-901e-2fgh-3i45j6k1789","Cores":
[{"thingArn":"arn:aws:iot:us-west-2:1234567
89012:thing/MyFirstGroup_Core","Connectivity":
[{"Id":"AUTOIP_192.168.1.4_1","HostAddress":"192.168.1.5","PortNumber
":8883,"Metadata":""}]}],"CAs":["-----BEGIN CERTIFICATE-----\ncert-contents\n-----END
 CERTIFICATE-----\n"]}]}
```

のセキュリティ AWS IoT Greengrass

のクラウドセキュリティが最優先事項 AWS です。 AWS カスタマーは、最もセキュリティの影響を受けやすい組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャを活用できます。

セキュリティは、 AWS とユーザーの間で共有される責任です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティと説明しています。

- クラウドのセキュリティ – AWS は、 で AWS サービスを実行するインフラストラクチャを保護する責任を担います AWS クラウド。 AWS また、 は、お客様が安全に使用できるサービスも提供します。サードパーティーの監査者は、AWS コンプライアンスプログラムコンプライアンスプログラムの一環として、当社のセキュリティの有効性を定期的にテストおよび検証。が適用されるコンプライアンスプログラムの詳細については AWS IoT Greengrass、「コンプライアンスプログラムAWS による対象範囲内のサービスコンプライアンスプログラム」を参照してください。
- クラウド内のセキュリティ - ユーザーの責任は、使用する 「 AWS 」 のサービスに応じて異なります。また、お客様は、お客様のデータの機密性、企業の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

を使用する場合は AWS IoT Greengrass、デバイス、ローカルネットワーク接続、プライベートキーを保護する責任も負います。

このドキュメントは、 を使用する際の責任共有モデルの適用方法を理解するのに役立ちます AWS IoT Greengrass。以下のトピックでは、セキュリティおよびコンプライアンスの目的 AWS IoT Greengrass を達成するために を設定する方法を示します。また、 AWS IoT Greengrass リソースのモニタリングや保護に役立つ他の AWS サービスの使用方法についても説明します。

トピック

- AWS IoT Greengrass セキュリティの概要
- でのデータ保護 AWS IoT Greengrass
- AWS IoT Greengrassのデバイス認証と認可
- の ID とアクセスの管理 AWS IoT Greengrass
- AWS IoT Greengrassのコンプライアンス検証
- の耐障害性 AWS IoT Greengrass
- のインフラストラクチャセキュリティ AWS IoT Greengrass

- の設定と脆弱性の分析 AWS IoT Greengrass
- AWS IoT Greengrass およびインターフェイス VPC エンドポイント (AWS PrivateLink)
- のセキュリティのベストプラクティス AWS IoT Greengrass

AWS IoT Greengrass セキュリティの概要

AWS IoT Greengrass は X.509 証明書、 AWS IoT ポリシー、IAM ポリシーとロールを使用して、ローカル Greengrass 環境のデバイスで実行されるアプリケーションを保護します。

次の図は、 AWS IoT Greengrass セキュリティモデルのコンポーネントを示しています。

A - Greengrass サービスロール

、 およびその他の AWS サービスから AWS リソースにアクセスする AWS IoT Greengrass ときに AWS IoT Core AWS Lambdaが引き受ける、お客様が作成した IAM ロール。詳細については、「the section called “Greengrass サービスロール”」を参照してください。

B - コアデバイス証明書

AWS IoT Core および で Greengrass コアを認証するために使用される X.509 証明書 AWS IoT Greengrass。詳細については、「the section called “デバイス認証と認可”」を参照してください。

C - デバイス証明書

クライアントデバイスを認証するために使用される X.509 証明書。接続デバイスとも呼ばれ、AWS IoT Core および を使用します AWS IoT Greengrass。詳細については、「the section called "デバイス認証と認可"」を参照してください。

D - グループロール

Greengrass コアから AWS サービスを呼び出す AWS IoT Greengrass ときに が引き受ける、お客様が作成した IAM ロール。

このロールを使用して、ユーザー定義の Lambda 関数とコネクタが DynamoDB などの AWS サービスにアクセスするために必要なアクセス許可を指定します。また、ストリームマネージャーストリーム AWS IoT Greengrass を サービスに AWS エクスポートし、CloudWatch Logs に書き込むことを に許可するためにも使用します。詳細については、「the section called "Greengrass グループのロール"」を参照してください。

> **Note**
>
> AWS IoT Greengrass は、Lambda 関数のクラウドバージョンに で AWS Lambda 指定された Lambda 実行ロールを使用しません。

E - MQTT サーバー証明書

Greengrass コアデバイスと Greengrass グループ内のクライアントデバイスとの間の Transport Layer Security (TLS) 相互認証に使用される証明書。証明書は、 AWS クラウドに保存されているグループ CA 証明書によって署名されます。

デバイス接続のワークフロー

このセクションでは、クライアントデバイスが AWS IoT Greengrass サービスと Greengrass コアデバイスに接続する方法について説明します。クライアントデバイスは、コア AWS IoT Core デバイスと同じ Greengrass グループにある登録済みデバイスです。

- Greengrass コアデバイスは、デバイス証明書、プライベートキー、ルート AWS IoT Core CA 証明書を使用して AWS IoT Greengrass サービスに接続します。コアデバイスでは、設定ファイル内の `crypto` オブジェクトがこれらの項目のファイルパスを指定します。
- Greengrass コアデバイスは、 AWS IoT Greengrass サービスからグループメンバーシップ情報をダウンロードします。

- Greengrass コアデバイスに対してデプロイが行われた場合に、Device Certificate Manager (DCM) は Greengrass コアデバイスに対してローカルサーバー証明書管理を行います。
- クライアントデバイスは、デバイス証明書、プライベートキー、および AWS IoT Core ルート CA 証明書を使用して AWS IoT Greengrass サービスに接続します。クライアントデバイスは、接続後に Greengrass Discovery Service を使用して Greengrass コアデバイスの IP アドレスを見つけます。また、クライアントデバイスはグループ CA 証明書をダウンロードします。この証明書は、Greengrass コアデバイスとの TLS 相互認証に使用されます。
- クライアントデバイスは Greengrass コアデバイスへの接続を試み、そのデバイス証明書とクライアント ID を渡します。クライアント ID がクライアントデバイスのモノ名と一致し、証明書が有効である (その Greengrass グループに所属する) 場合、接続が実行されます。それ以外の場合は、接続は終了します。

クライアントデバイスの AWS IoT ポリシーは、クライアントデバイスがコアの接続情報を検出できるようにするアクセス`greengrass:Discover`許可を付与する必要があります。このポリシーステートメントの詳細については、「the section called “検出の認可”」を参照してください。

AWS IoT Greengrass セキュリティの設定

Greengrass アプリケーションのセキュリティを設定するには

1. Greengrass コアデバイスの AWS IoT Core モノを作成します。
2. Greengrass コアデバイスのキーペアとデバイス証明書を生成します。
3. AWS IoT ポリシーを作成してデバイス証明書にアタッチします。証明書とポリシーにより、Greengrass コアデバイスが AWS IoT Core および AWS IoT Greengrass サービスにアクセスできるようになります。詳細については、「コアデバイスの最小 AWS IoT ポリシー」を参照してください。

   > **Note**
   >
   > AWS IoT コアデバイスのポリシーでのモノのポリシー変数 (`iot:Connection.Thing.*`) の使用はサポートされていません。コアは同じデバイス証明書を使用して に複数の接続を行います AWS IoT Core が、接続内のクライアント ID がコアモノの名前と完全に一致しない可能性があります。

4. Greengrass サービスロールを作成します。この IAM ロールは、ユーザーに代わって が他の AWS のサービスからリソースにアクセス AWS IoT Greengrass することを許可します。これに

より、 AWS IoT Greengrass は関数の取得 AWS Lambda やデバイスシャドウの管理など、重要なタスクを実行できます。

間で同じサービスロールを使用できますが、 AWS リージョン 使用するすべての AWS リージョン AWS アカウント で に関連付ける必要があります AWS IoT Greengrass。

5. (オプション) Greengrass グループロールを作成します。この IAM ロールは、Greengrass コアで実行されている Lambda 関数とコネクタに、 AWS サービスを呼び出すアクセス許可を付与します。例えば、Kinesis Firehose コネクタには、Amazon Data Firehose 配信システムにレコードを書き込むアクセス許可が必要です。

   Greengrass グループにアタッチできるロールは 1 つだけです。

6. Greengrass コアに接続するデバイスごとに AWS IoT Core モノを作成します。

   > **Note**
   >
   > 既存の AWS IoT Core モノと証明書を使用することもできます。

7. Greengrass コアに接続するデバイスごとに、デバイス証明書、キーペア、 AWS IoT およびポリシーを作成します。

AWS IoT Greengrass コアセキュリティプリンシパル

Greengrass コアは、 AWS IoT クライアント、ローカル MQTT サーバー、ローカルシークレットマネージャーのセキュリティプリンシパルを使用します。これらのプリンシパルの設定は、`config.json` 設定ファイルの `crypto` オブジェクトに格納されます。詳細については、「the section called “AWS IoT Greengrass コア設定ファイル”」を参照してください。

この設定には、認証および暗号化の主要なコンポーネントが使用するプライベートキーへのパスが含まれています。 AWS IoT Greengrass では、ハードウェアベースあるいはファイルシステムベース (デフォルト) の 2 種類のプライベートキーストレージがサポートされています。ハードウェアセキュリティモジュールにキーを保存する方法の詳細については、「the section called “ハードウェアセキュリティ統合”」を参照してください。

AWS IoT クライアント

AWS IoT クライアント (IoT クライアント) は、Greengrass コアと 間のインターネット経由の通信を管理します AWS IoT Core。 AWS IoT Greengrass は、この通信の TLS 接続を確立するときに、パブリックキーとプライベートキーを持つ X.509 証明書を相互認証に使用します。詳細につ

いては、「AWS IoT Core デベロッパーガイド」の「X.509 証明書と AWS IoT Core」を参照してください。

IoT クライアントでは RSA および EC 証明書とキーがサポートされています。証明書とプライベートキーのパスは、`config.json` の `IoTCertificate` プリンシパルで指定されています。

MQTT サーバー

ローカル MQTT サーバーは、 グループ内の Greengrass コアデバイスとクライアントデバイス間のローカルネットワークを介した通信を管理します。 は、この通信の TLS 接続を確立するときに、相互認証にパブリックキーとプライベートキーを持つ X.509 証明書 AWS IoT Greengrass を使用します。

デフォルトでは、 は RSA プライベートキー AWS IoT Greengrass を生成します。別のプライベートキーを使用するコアを設定するには、`config.json` で `MQTTServerCertificate` プリンシパルへのキーパスを指定する必要があります。お客様が用意したキーのローテーションは、お客様が行います。

プライベートキーサポート

	RSA キー	EC キー
キーのタイプ	Supported	Supported
キーのパラメータ	Minimum 2048-bit length	NIST P-256 or NIST P-384 curve
ディスク形式	PKCS#1, PKCS#8	SECG1, PKCS#8
最小 GGC バージョン	• デフォルトの RSA キー使用: 1.0 • RSA キーの指定: 1.7	• EC キーの指定: 1.9

プライベートキーの設定は、関連するプロセスを決定します。Greengrass コアでサーバーとしてサポートされる暗号化スイートの一覧については、「the section called “TLS 暗号スイートのサポート”」を参照してください。

プライベートキーが指定されていない場合 (デフォルト)

- AWS IoT Greengrass は、ローテーション設定に基づいてキーをローテーションします。
- コアは、証明書の生成に使用される RSA キーを生成します。

- MQTT サーバー証明書には、RSA パブリックキーと SHA-256 RSA 署名が含まれています。

RSA プライベートキーが指定されている場合 (GGC v1.7 以降が必要)

- キーのローテーションはお客様が行います。
- コアは指定されたキーを使用して証明書を生成します。
- RSA キーは、少なくとも 2048 ビットの長さにする必要があります。
- MQTT サーバー証明書には、RSA パブリックキーと SHA-256 RSA 署名が含まれています。

EC プライベートキーが指定されている場合 (GGC v1.9 以降が必要)

- キーのローテーションはお客様が行います。
- コアは指定されたキーを使用して証明書を生成します。
- EC プライベートキーは、NIST P-256 または NIST P-384 curve を使用する必要があります。
- MQTT サーバー証明書には、EC パブリックキーと SHA-256 RSA 署名があります。

  コアが提供する MQTT サーバー証明書には、キーのタイプに関係なく、SHA-256 RSA 署名があります。このため、クライアントはコアと安全な接続を確立するために、SHA-256 RSA 証明書の検証をサポートしている必要があります。

シークレットマネージャー

ローカルシークレットマネージャーは、作成したシークレットのローカルコピーを安全に管理します AWS Secrets Manager。ここでは、プライベートキーを使用して、シークレットを暗号化するために使用されるデータキーを保護します。詳細については、「Core にシークレットをデプロイする」を参照してください。

デフォルトでは IoT クライアントプライベートキーが使用されますが、`config.json` の `SecretsManager` プリンシパルに別のプライベートキーを指定することもできます。RSA キータイプのみがサポートされています。詳細については、「the section called “シークレット暗号化用のプライベートキーを指定する”」を参照してください。

> **Note**
>
> 現在、 は、ハードウェアベースのプライベートキーを使用する場合のローカルシークレットの暗号化と復号のための PKCS#1 v1.5 パディングメカニズムのみ AWS IoT Greengrass をサポートしています。ベンダーが提供する指示に従ってハードウェアベースのプライベートキーを手動で生成する場合は、PKCS#1 v1.5. AWS IoT Greengrass を

選択し、最適な非対称暗号化パディング (OAEP) をサポートしていないことを確認してください。

プライベートキーサポート

	RSA キー	EC キー
キーのタイプ	Supported	Not supported
キーのパラメータ	Minimum 2048-bit length	Not applicable
ディスク形式	PKCS#1, PKCS#8	Not applicable
最小 GGC バージョン	1.7	Not applicable

MQTT メッセージングワークフローにおけるマネージドサブスクリプション

AWS IoT Greengrass は、サブスクリプションテーブルを使用して、Greengrass グループのクライアントデバイス、関数、コネクタ間、および AWS IoT Core またはローカルシャドウサービスと MQTT メッセージを交換する方法を定義します。各サブスクリプションは、メッセージが送受信される送信元、ターゲット、および MQTT トピック (またはサブジェクト) を指定します。 は、対応するサブスクリプションが定義されている場合にのみ、送信元からターゲットへのメッセージの送信 AWS IoT Greengrass を許可します。

サブスクリプションは一方向のメッセージフローのみを定義します。双方向のメッセージ交換をサポートするには、各方向に 1 つずつ、2 つのサブスクリプションを作成する必要があります。

TLS 暗号スイートのサポート

AWS IoT Greengrass は、 AWS IoT Core トランスポートセキュリティモデルを使用して、TLS 暗号スイートを使用してクラウドとの通信を暗号化します。さらに、 AWS IoT Greengrass データは保管時 (クラウド内) に暗号化されます。 AWS IoT Core トランスポートセキュリティとサポートされている暗号スイートの詳細については、「 AWS IoT Core デベロッパーガイド」の「トランスポートセキュリティ」を参照してください。

ローカルネットワーク通信向けにサポートされる暗号化スイート

これとは対照的に AWS IoT Core、 AWS IoT Greengrass コアは証明書署名アルゴリズム用に以下のローカルネットワーク TLS 暗号スイートをサポートしています。プライベートキーがファイルシステムに保存されている場合、これらの暗号化スイートはすべてサポートされます。サブセットは、コアがハードウェアセキュリティモジュール (HSM) を使用するように設定されている場合にサポートされます。詳細については、the section called “セキュリティプリンシパル”およびthe section called “ハードウェアセキュリティ統合”を参照してください。この表には、サポートに必要な AWS IoT Greengrass Core ソフトウェアの最小バージョンも含まれています。

	暗号	HSM のサポート	最小 GGC バージョン
TLSv1.2	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	Supported	1.0
	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Supported	1.0
	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	Supported	1.0
	TLS_RSA_WITH_AES_128_CBC_SHA	Not supported	1.0
	TLS_RSA_WITH_AES_128_GCM_SHA256	Not supported	1.0
	TLS_RSA_WITH_AES_256_CBC_SHA	Not supported	1.0
	TLS_RSA_WITH_AES_256_GCM_SHA384	Not supported	1.0

	暗号	HSM のサポート	最小 GGC バージョン
	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	Supported	1.9
	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	Supported	1.9
TLSv1.1	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	Supported	1.0
	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Supported	1.0
	TLS_RSA_WITH_AES_128_CBC_SHA	Not supported	1.0
	TLS_RSA_WITH_AES_256_CBC_SHA	Not supported	1.0
TLSv1.0	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	Supported	1.0
	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Supported	1.0
	TLS_RSA_WITH_AES_128_CBC_SHA	Not supported	1.0

暗号	HSM のサポート	最小 GGC バージョン
TLS_RSA_WITH_AES_256_CBC_SHA	Not supported	1.0

でのデータ保護 AWS IoT Greengrass

責任 AWS 共有モデル、 でのデータ保護に適用されます AWS IoT Greengrass。このモデルで説明されているように、 AWS はすべての を実行するグローバルインフラストラクチャを保護する責任があります AWS クラウド。ユーザーは、このインフラストラクチャでホストされるコンテンツに対する管理を維持する責任があります。また、使用する「 AWS のサービス 」のセキュリティ設定と管理タスクもユーザーの責任となります。データプライバシーの詳細については、データプライバシーに関するよくある質問を参照してください。欧州でのデータ保護の詳細については、AWS セキュリティブログに投稿された AWS 責任共有モデルおよび GDPR のブログ記事を参照してください。

データ保護の目的で、認証情報を保護し AWS アカウント 、 AWS IAM Identity Center または AWS Identity and Access Management (IAM) を使用して個々のユーザーを設定することをお勧めします。この方法により、それぞれのジョブを遂行するために必要な権限のみが各ユーザーに付与されます。また、次の方法でデータを保護することもお勧めします:

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- で API とユーザーアクティビティのログ記録を設定します AWS CloudTrail。CloudTrail 証跡を使用して AWS アクティビティをキャプチャする方法については、「 AWS CloudTrail ユーザーガイド」のCloudTrail 証跡の使用」を参照してください。
- AWS 暗号化ソリューションと、 内のすべてのデフォルトのセキュリティコントロールを使用します AWS のサービス。
- Amazon Macie などの高度な管理されたセキュリティサービスを使用します。これらは、Amazon S3 に保存されている機密データの検出と保護を支援します。
- コマンドラインインターフェイスまたは API AWS を介して にアクセスするときに FIPS 140-3 検証済み暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「連邦情報処理規格 (FIPS) 140-3」を参照してください。

お客様の E メールアドレスなどの極秘または機密情報を、タグ、または [名前] フィールドなどの自由形式のテキストフィールドに含めないことを強くお勧めします。これは、コンソール AWS IoT Greengrass 、API、または SDK を使用して AWS CLIまたは他の AWS のサービス を操作する場合も同様です。 AWS SDKs タグ、または名前に使用される自由記述のテキストフィールドに入力したデータは、請求または診断ログに使用される場合があります。外部サーバーに URL を提供する場合、そのサーバーへのリクエストを検証できるように、認証情報を URL に含めないことを強くお勧めします。

での機密情報の保護の詳細については AWS IoT Greengrass、「」を参照してくださいthe section called “機密情報を記録しない”。

データ保護の詳細については、AWS セキュリティブログ のブログ投稿「AWS の責任共有モデルと GDPR」を参照してください。

トピック

- データ暗号化
- ハードウェアセキュリティ統合

データ暗号化

AWS IoT Greengrass は暗号化を使用して、転送中 (インターネットまたはローカルネットワーク経由) および保管中 (に保存) のデータを保護します AWS クラウド。

AWS IoT Greengrass 環境内のデバイスは、多くの場合、さらなる処理のために AWS サービスに送信されるデータを収集します。他の AWS サービスのデータ暗号化の詳細については、そのサービスのセキュリティドキュメントを参照してください。

トピック

- 転送中の暗号化
- 保管中の暗号化
- Greengrass Core Device のキー管理

転送中の暗号化

AWS IoT Greengrass には、データが転送される 3 つの通信モードがあります。

- the section called “インターネット経由で転送中のデータ”。 Greengrass コアとインターネット AWS IoT Greengrass 経由の通信は暗号化されます。

- the section called “ローカルネットワーク経由で転送中のデータ”。ローカルネットワークを介した Greengrass Core とクライアントデバイス間の通信は暗号化されます。
- the section called “コアデバイス上のデータ”: Greengrass コアデバイス上のコンポーネント間の通信は暗号化されません。

インターネット経由で転送中のデータ

AWS IoT Greengrass は Transport Layer Security (TLS) を使用して、インターネット経由のすべての通信を暗号化します。に送信されるすべてのデータは、MQTT または HTTPS プロトコルを使用して TLS 接続を介して送信されるため、デフォルトでは安全です。 AWS クラウド は AWS IoT トランスポートセキュリティモデル AWS IoT Greengrass を使用します。詳細については、[AWS IoT Core Developer Guide] (デベロッパーガイド) の[Transport security] (トランスポートセキュリティ) を参照してください。

ローカルネットワーク経由で転送中のデータ

AWS IoT Greengrass は TLS を使用して、Greengrass コアデバイスとクライアントデバイス間のローカルネットワーク経由のすべての通信を暗号化します。詳細については、「ローカルネットワーク通信でサポートされる暗号スイート」を参照してください。

ローカルネットワークとプライベートキーを保護するのはお客様の責任となります。

Greengrass コアデバイスに関するお客様の責任は次のとおりです。

- 最新のセキュリティパッチでカーネルを最新の状態に保ちます。
- 最新のセキュリティパッチでシステムライブラリを最新の状態に保ちます。
- プライベートキーを保護します。詳細については、「the section called “キー管理”」を参照してください。

クライアントデバイスに関するお客様の責任は次のとおりです。

- TLS スタックを最新の状態に保ちます。
- プライベートキーを保護します。

コアデバイス上のデータ

AWS IoT Greengrass は、Greengrass コアデバイスでローカルに交換されたデータを暗号化しません。これは、データがデバイスから離れないためです。これには、ユーザー定義の Lambda 関数、コネクタ、 AWS IoT Greengrass Core SDK、ストリームマネージャーなどのシステムコンポーネント間の通信が含まれます。

保管中の暗号化

AWS IoT Greengrass はデータを保存します。

- the section called “の保管中のデータ AWS クラウド”. このデータは暗号化されます。
- the section called “Greengrass コアに保管されているデータ”. このデータは暗号化されません (シークレットのローカルコピーを除きます)。

の保管中のデータ AWS クラウド

AWS IoT Greengrass は、 に保存されている顧客データを暗号化します AWS クラウド。このデータは、 によって管理される AWS KMS キーを使用して保護されます AWS IoT Greengrass。

Greengrass コアに保管されているデータ

AWS IoT Greengrass は、Unix ファイルのアクセス許可とフルディスク暗号化 (有効になっている場合) を使用して、コア上の保管中のデータを保護します。ファイルシステムとデバイスを保護するのはお客様の責任となります。

ただし、 AWS IoT Greengrass は取得元のシークレットのローカルコピーを暗号化します AWS Secrets Manager。詳細については、「the section called “シークレットの暗号化”」を参照してください。

Greengrass Core Device のキー管理

Greengrass Core Device の暗号化 (パブリックおよびプライベート) キーを安全に保管するのは、お客様の責任です。 AWS IoT Greengrass は次のシナリオでパブリックキーおよびプライベートキーを使用します。

- IoT クライアントキーは IoT 証明書とともに使用され、Greengrass Core が AWS IoT Coreに接続すると Transport Layer Security (TLS) ハンドシェイクを認証します。詳細については、「the section called “デバイス認証と認可”」を参照してください。

  > **Note**
  > キーおよび証明書は、コアプライベートキーおよびコアデバイス証明書とも呼ばれます。

- MQTT サーバーキーは MQTT サーバー証明書を使用して、コアデバイスとクライアントデバイス間の TLS 接続を認証します。詳細については、「the section called “デバイス認証と認可”」を参照してください。

- ローカルの Secrets Manager は、IoT クライアントキーを使用してローカルのシークレットの暗号化に使用されるデータキーを保護しますが、独自のプライベートキーを提供することもできます。詳細については、「the section called “シークレットの暗号化”」を参照してください。

Greengrass Core は、ファイルシステムのアクセス許可、ハードウェアセキュリティモジュール、またはその両方を使用して、プライベートキーストレージをサポートします。ファイルシステムベースのプライベートキーを使用する場合は、お客様がコアデバイス上の安全な保管の責任を負います。

Greengrass Core では、プライベートキーの場所は `config.json` ファイルの `crypto` セクションに指定されています。MQTT サーバー証明書にお客様が提供するキーを使用するようにコアを設定する場合、キーを回転するのはお客様の責任です。詳細については、「the section called “セキュリティプリンシパル”」を参照してください。

クライアントデバイスの場合、TLS スタックを最新の状態に保ち、プライベートキーを保護するのはお客様の責任です。プライベートキーは、 AWS IoT Greengrass サービスとの TLS 接続を認証するためにデバイス証明書とともに使用されます。

ハードウェアセキュリティ統合

この機能は AWS IoT Greengrass Core v1.7 以降で使用できます。

AWS IoT Greengrass は、PKCS#11 インターフェイスを介したハードウェアセキュリティモジュール (HSM) の使用をサポートし、プライベートキーの安全なストレージとオフロードをサポートします。これにより、ソフトウェアでキーが漏洩したり複製されたりするのを防ぎます。プライベートキーは、HSM、Trusted Platform Modules (TPM)、その他の暗号化要素などのハードウェアモジュールに安全に保存できます。

AWS Partner Device Catalog で、この機能に適合するデバイスを検索します。

次の図は、 AWS IoT Greengrass コアのハードウェアセキュリティアーキテクチャを示しています。

標準インストールでは、 は 2 つのプライベートキー AWS IoT Greengrass を使用します。1 つのキーは、Greengrass コアが接続するときに Transport Layer Security (TLS) ハンドシェイク中に AWS IoT クライアント (IoT クライアント) コンポーネントによって使用されます AWS IoT Core。(このキーは Core プライベートキーとも呼ばれます)。もう 1 つのキーは、Greengrass デバイスと Greengrass Core との通信を可能にするために、ローカル MQTT サーバーが使用します。両方のコンポーネントにハードウェアセキュリティを使用する場合は、共有のプライベートキーを使用しても、個別のプライベートキーを使用してもかまいません。詳細については、「the section called “ プロビジョニング慣行”」を参照してください。

> **Note**
>
> 標準インストールでは、ローカルシームレットマネージャは暗号化目的で IoT クライアントキーも使用しますが、独自のプライベートキーを使用することもできます。この場合、最小長 の 2048 ビットの RSA キーを使用する必要があります。詳細については、「the section called “シークレット暗号化用のプライベートキーを指定する”」を参照してください。

要件

Greengrass Core のハードウェアセキュリティを設定する前に、以下のものが必要です。

- IoT クライアント、ローカル MQTT サーバー、およびローカルサービスマネージャコンポーネント用のターゲットプライベートキー設定をサポートするハードウェアセキュリティモジュール (HSM)。この設定には、コンポーネントでキーを共有するかどうかの設定に応じて、1 つ、2 つ、

あるいは3 つのハードウェアベースのプライベートキーを含めることができます。プライベートキーのサポートの詳細については、「the section called “セキュリティプリンシパル”」を参照してください。

- RSA キーでは: RSA-2048 キーサイズ (またはそれ以上) および PKCS#1 v1.5 署名スキーム。
- EC キーでは: NIST P-256 または NIST P-384 curve。

> **Note**
>
> AWS Partner Device Catalog で、この機能に適合するデバイスを検索します。

- 実行時に (libdl を使用して) ロード可能であり、PKCS#11 関数を提供する PKCS#11 プロバイダーライブラリ。
- ハードウェアモジュールは、「PKCS#11 仕様」で定義されているスロットラベルで解決できる必要があります。
- ベンダー提供のプロビジョニングツールを使用して、プライベートキーを生成して HSM にロードする必要があります。
- プライベートキーはオブジェクトラベルで解決できる必要があります。
- Core デバイス証明書。これは、プライベートキーに対応する IoT クライアント証明書です。
- Greengrass OTA Update Agent を使用している場合は、OpenSSL libp11 PKCS#11 ラッパーライブラリをインストールする必要があります。詳細については、「the section called “OTA 更新を設定する”」を参照してください。

また、以下の条件が満たされていることを確認してください。

- プライベートキーに関連付けられている IoT クライアント証明書は、 に登録 AWS IoT され、アクティブ化されます。これは、 AWS IoT コンソールの「すべてのデバイスの管理」、「展開」、「モノ」、「コアモノの証明書」タブで確認できます。
- AWS IoT Greengrass Core ソフトウェア v1.7 以降は、入門チュートリアルのモジュール 2 で説明されているように、コアデバイスにインストールされます。バージョン 1.9 以降は、MQTT サーバー用に EC キーを使用するために必要です。
- 証明書は Greengrass Core にアタッチされています。これは、 AWS IoT コンソールのコアモノの管理ページから確認できます。

> **Note**
>
> 現在、 AWS IoT Greengrass は、HSM からの CA 証明書または IoT クライアント証明書の直接ロードをサポートしていません。証明書は、ファイルシステムの Greengrass で読み取り可能な場所に、プレーンテキストファイルとしてロードする必要があります。

AWS IoT Greengrass コアのハードウェアセキュリティ設定

ハードウェアセキュリティは、Greengrass 設定ファイルで設定します。これは、`/greengrass-root/config` ディレクトリにある `config.json` ファイルです。

> **Note**
>
> 純粋なソフトウェア実装を使用して HSM 設定をセットアップするプロセスについては、「the section called “モジュール 7: ハードウェアセキュリティ統合のシミュレーション”」を参照してください。
>
> > **Important**
> >
> > この例のシミュレートされた設定では、セキュリティ上の利点は得られません。例の目的は、ハードウェアベースの HSM を今後使用する場合に備えて、PKCS#11 の仕様について学習し、ソフトウェアの初期テストを行うことです。

でハードウェアセキュリティを設定するには AWS IoT Greengrass、 で `crypto` オブジェクトを編集します`config.json`。

ハードウェアセキュリティを使用する場合は、以下の例に示すように、`crypto` オブジェクトを使用して、Core 上の PKCS#11 プロバイダーライブラリの証明書、プライベートキー、アセットへのパスを指定します。

```
"crypto": {
  "PKCS11" : {
    "OpenSSLEngine" : "/path-to-p11-openssl-engine",
    "P11Provider" : "/path-to-pkcs11-provider-so",
    "slotLabel" : "crypto-token-name",
    "slotUserPin" : "crypto-token-user-pin"
  },
```

```
  "principals" : {
    "IoTCertificate" : {
      "privateKeyPath" : "pkcs11:object=core-private-key-label;type=private",
      "certificatePath" : "file:///path-to-core-device-certificate"
    },
    "MQTTServerCertificate" : {
      "privateKeyPath" : "pkcs11:object=server-private-key-label;type=private"
    },
    "SecretsManager" : {
      "privateKeyPath": "pkcs11:object=core-private-key-label;type=private"
    }
  },
  "caPath" : "file:///path-to-root-ca"
```

`crypto` オブジェクトには、以下のプロパティが含まれています。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`PKCS11`		
`OpenSSLEngine`	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン `.so` ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合

フィールド	説明	メモ
		、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
`P11Provider`	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
`slotLabel`	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。

フィールド	説明	メモ
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`PKCS11`		

フィールド	説明	メモ
`OpenSSLEngine`	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン `.so` ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
`P11Provider`	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
`slotLabel`	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	

フィールド	説明	メモ
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate .privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

フィールド	説明	メモ
`caPath`	AWS IoT ルート CA への絶対パス。	`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 **Note** エンドポイントが証明書タイプに対応していることを確認してください。
`PKCS11`		

フィールド	説明	メモ
`OpenSSLEngine`	オプション。OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジン `.so` ファイルへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。 ハードウェアセキュリティで Greengrass OTA Update Agent を使用している場合、このプロパティは必須です。詳細については、「the section called “OTA 更新を設定する”」を参照してください。
`P11Provider`	PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。	ファイルシステム上のファイルへのパスであることが必要です。
`slotLabel`	ハードウェアモジュールを識別するために使用されるスロットラベル。	PKCS#11 ラベル仕様に準拠していることが必要です。
`slotUserPin`	Greengrass Core をモジュールに対して認証するために使用されるユーザー PIN。	設定されたプライベートキーで C_Sign を実行するのに十分なアクセス許可があることが必要です。
`principals`		
`IoTCertificate`	The certificate and private key that the core uses to make requests to AWS IoT.	

フィールド	説明	メモ
`IoTCertificate.privateKeyPath`	Core プライベートキーへのパス。	ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。
`IoTCertificate.certificatePath`	コアデバイス証明書への絶対パス。	`file:///`*`absolute/path/to/file` という形式のファイル URI である必要があります。
`MQTTServerCertificate`	オプション。Core が MQTT サーバーまたはゲートウェイとして機能するために証明書と組み合わせて使用するプライベートキー。	

フィールド	説明	メモ
`MQTTServerCertificate.privateKeyPath`	ローカル MQTT サーバーのプライベートキーへのパス。	この値を使用して、ローカル MQTT サーバーの独自のプライベートキーを指定します。 ファイルシステムストレージの場合、`file:///`*`absolute/path/to/file`* という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。 このプロパティを省略すると、 はローテーション設定に基づいてキーを AWS IoT Greengrass ローテーションします。指定した場合は、お客様がキーを更新する必要があります。
`SecretsManager`	The private key that secures the data key used for encryption. For more information, see Core にシークレットをデプロイする.	

フィールド	説明	メモ
`SecretsManager.privateKeyPath`	ローカル Secrets Manager のプライベートキーへのパス。	RSA キーのみがサポートされています。 ファイルシステムストレージの場合、`file:///absolute/path/to/file` という形式のファイル URI である必要があります。 HSM ストレージの場合、オブジェクトラベルを指定する RFC 7512 PKCS#11 パスである必要があります。PKCS#1 v1.5 パディング方式を使用してプライベートキーを生成する必要があります。

AWS IoT Greengrass ハードウェアセキュリティのプロビジョニングプラクティス

以下に示しているのは、セキュリティおよびパフォーマンス関連のプロビジョニング慣行です。

セキュリティ

- 内部ハードウェア乱数ジェネレーターを使用して、HSM に直接プライベートキーを生成します。

> **Note**
>
> この機能で使用するようにプライベートキーを設定する場合 (ハードウェアベンダーから提供された指示に従う)、 AWS IoT Greengrass は現在、ローカルシークレットの暗号化と復号のための PKCS1 v1.5 パディングメカニズムのみをサポートしていることに注意してください。 AWS IoT Greengrass は、Optimal Asymmetric Encryption Padding (OAEP) をサポートしていません。

- エクスポートを禁止するようにプライベートキーを設定します。

- ハードウェアベンダーが提供するプロビジョニングツールを使用して、ハードウェアで保護されたプライベートキーを使用して証明書署名リクエスト (CSR) を生成し、 AWS IoT コンソールを使用してクライアント証明書を生成します。

> **Note**
>
> HSM でプライベートキーが生成されている場合、キーの更新は適用されません。

パフォーマンス

次の図は、 AWS IoT Greengrass コア上の IoT クライアントコンポーネントとローカル MQTT サーバーを示しています。両方のコンポーネントに HSM 設定を使用する場合は、同じプライベートキーを使用しても、個別のプライベートキーを使用してもかまいません。別のキーを使用する場合は、同じスロットに保存されている必要があります。

> **Note**
>
> AWS IoT Greengrass は、HSM に保存するキーの数に制限を課さないため、IoT クライアント、MQTT サーバー、およびシークレットマネージャーコンポーネントのプライベートキーを保存できます。ただし、一部の HSM のベンダーでは 1 つのスロットに保存できるキーの数に制限を設けていることがあります。

一般的に、 AWS IoT Greengrass Core ソフトウェアはクラウドへの存続期間の長い接続を維持するため、IoT クライアントキーはあまり使用されません。ただし、MQTT サーバーキーは、Greengrass デバイスが Core に接続するたびに使用されます。これらのやり取りはパフォーマンスに直接影響します。

MQTT サーバーキーが HSM に保存されていると、デバイスが接続できる速度は、HSM が実行できる 1 秒あたりの RSA 署名オペレーションの数によって決まります。例えば、HSM が RSA-2048 プライベートキーで RSASSA-PKCS1-v1.5 署名を実行するのに 300 ミリ秒かかる場合、1 秒あたり 3 台のデバイスしか Greengrass Core に接続できません。接続後、HSM は使用されなくなり、標準の AWS IoT Greengrassのクォータが適用されます。

パフォーマンスのボトルネックを軽減するために、MQTT サーバーのプライベートキーを HSM ではなくファイルシステムに保存できます。この設定では、MQTT サーバーは、ハードウェアセキュリティが有効でないかのように動作します。

AWS IoT Greengrass は IoT クライアントおよび MQTT サーバーコンポーネントに対して複数のキーストレージ設定をサポートしているため、セキュリティとパフォーマンスの要件に合わせて最適化できます。次の表では設定例を示しています。

設定	IoT キー	MQTT キー	パフォーマンス
HSM 共有キー	HSM: キー A	HSM: キー A	HSM または CPU によって制限

設定	IoT キー	MQTT キー	パフォーマンス
HSM 分離キー	HSM: キー A	HSM: キー B	HSM または CPU によって制限
IoT 専用 HSM	HSM: キー A	ファイルシステム: キー B	CPU によって制限
レガシー	ファイルシステム: キー A	ファイルシステム: キー B	CPU によって制限

MQTT サーバーのファイルシステムベースのキーを使用するように Greengrass コアを設定するには、 から `principals.MQTTServerCertificate`セクションを省略します `config.json` (または、 によって生成されたデフォルトキーを使用しない場合は、キーへのファイルベースのパスを指定します AWS IoT Greengrass) 。生成される `crypto` オブジェクトは以下のようになります。

```
"crypto": {
  "PKCS11": {
    "OpenSSLEngine": "...",
    "P11Provider": "...",
    "slotLabel": "...",
    "slotUserPin": "..."
  },
  "principals": {
    "IoTCertificate": {
      "privateKeyPath": "...",
      "certificatePath": "..."
    },
    "SecretsManager": {
      "privateKeyPath": "..."
    }
  },
  "caPath" : "..."
}
```

ハードウェアセキュリティ統合用にサポートされている暗号スイート

AWS IoT Greengrass コアがハードウェアセキュリティ用に設定されている場合、 は一連の暗号スイートをサポートします。これは、Core がファイルベースのセキュリティを使用するように設定されている場合にサポートされる暗号スイートのサブセットです。詳細については、「the section called “TLS 暗号スイートのサポート”」を参照してください。

> **Note**
>
> ローカルネットワークを介して Greengrass デバイスから Greengrass コアに接続する場合、TLS 接続を行うためにサポートされた暗号化スイートのいずれかを使用していることを確認してください。

無線通信経由更新のサポートを設定する

ハードウェアセキュリティを使用するときに AWS IoT Greengrass Core over-the-air (OTA) 更新を有効にするには、OpenSC libp11 PKCS#11 ラッパーライブラリをインストールし、Greengrass 設定ファイルを編集する必要があります。OTA 更新の詳細については、「 AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。

1. Greengrass デーモンを停止します。

   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd stop
   ```

   > **Note**
   >
   > *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

2. OpenSSL エンジンをインストールします。OpenSSL 1.0 または 1.1 がサポートされています。

   ```
   sudo apt-get install libengine-pkcs11-openssl
   ```

3. システム上の OpenSSL エンジン (`libpkcs11.so`) へのパスを見つけます。

   a. インストールされているライブラリのパッケージのリストを取得します。

```
sudo dpkg -L libengine-pkcs11-openssl
```

libpkcs11.so ファイルは engines ディレクトリにあります。

b. このファイルへの完全パス (/usr/lib/ssl/engines/libpkcs11.so など) をコピーします。

4. Greengrass 設定ファイルを開きます。これは、*/greengrass-root*/config ディレクトリにある config.json ファイルです。
5. OpenSSLEngine プロパティには、libpkcs11.so ファイルへのパスを入力します。

```
{
 "crypto": {
   "caPath" : "file:///path-to-root-ca",
   "PKCS11" : {
     "OpenSSLEngine" : "/path-to-p11-openssl-engine",
     "P11Provider" : "/path-to-pkcs11-provider-so",
     "slotLabel" : "crypto-token-name",
     "slotUserPin" : "crypto-token-user-pin"
    },
    ...
  }
  ...
}
```

> **Note**
>
> OpenSSLEngine プロパティが PKCS11 オブジェクトに存在しない場合は、追加します。

6. Greengrass デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

以前のバージョンの AWS IoT Greengrass コアソフトウェアとの下位互換性

ハードウェアセキュリティをサポートする AWS IoT Greengrass Core ソフトウェアは、v1.6 以前で生成されたconfig.jsonファイルと完全に下位互換性があります。crypto

オブジェクトがconfig.json設定ファイルに存在しない場合、 はファイルベースの coreThing.certPath、coreThing.keyPath、および coreThing.caPathプロパティ AWS IoT Greengrass を使用します。この下位互換性は、config.json で指定されたファイルベースの設定を上書きしない Greengrass OTA 更新に適用されます。

PKCS#11 をサポートしないハードウェア

PKCS#11 ライブラリは通常、ハードウェアベンダーによって提供されるか、オープンソースです。例えば、標準準拠のハードウェア (TPM1.2 など) では、既存のオープンソースソフトウェアを使用できます。ただし、ハードウェアに対応する PKCS#11 ライブラリの実装がない場合、またはカスタム PKCS#11 プロバイダーを作成する場合は、統合関連の質問について AWS エンタープライズサポート担当者にお問い合わせください。

関連情報

- PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40 John Leiseboer および Robert Griffin 編集。2014 年 11 月 16 日。「OASIS Committee Note 02」http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/cn02/pkcs11-ug-v2.40-cn02.html 最新バージョン: http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html
- RFC 7512
- PKCS #1: RSA Encryption Version 1.5

AWS IoT Greengrassのデバイス認証と認可

AWS IoT Greengrass 環境にあるデバイスは、認証に X.509 証明書を使用し、認可に AWS IoT ポリシーを使用します。証明書とポリシーにより、デバイスは相互に安全に接続し AWS IoT Core、 および を行うことができます AWS IoT Greengrass。

X.509 証明書は、X.509 パブリックキーインフラストラクチャ規格を使用して、パブリックキーと証明書内の ID を関連付けるための、デジタル証明書です。X.509 証明書は、証明機関 (CA) と呼ばれる信頼された団体によって発行されます。CA は、CA 証明書と呼ばれる 1 つ以上の特別な証明書を管理しており、この証明書は X.509 証明書を発行するために使用されます。証明機関にのみ CA 証明書に対するアクセス権限があります。

AWS IoT ポリシーは、 AWS IoT デバイスに対して許可される一連のオペレーションを定義します。具体的には、MQTT メッセージの発行やデバイスシャドウの取得など、 および AWS IoT Greengrass データプレーンオペレーションへのアクセス AWS IoT Core を許可および拒否します。

すべてのデバイスには、 AWS IoT Core レジストリのエントリと、 AWS IoT ポリシーがアタッチされたアクティブ化された X.509 証明書が必要です。デバイスは、次の 2 つのカテゴリに分類されます。

- Greengrass コア。Greengrass コアデバイスは、証明書と AWS IoT ポリシーを使用して接続します AWS IoT Core。証明書とポリシーにより、 AWS IoT Greengrass は設定情報、Lambda 関数、コネクタ、マネージドサブスクリプションをコアデバイスにデプロイすることもできます。
- クライアントデバイス。クライアントデバイス (接続されたデバイス、Greengrass デバイス、またはデバイスとも呼ばれる) は、MQTT を介して Greengrass コアに接続するデバイスのことです。証明書とポリシーを使用して、 AWS IoT Core および AWS IoT Greengrass サービスに接続します。これにより、クライアントデバイスは AWS IoT Greengrass Discovery Service を使用してコアデバイスを検索して接続できます。クライアントデバイスは、同じ証明書を使用して AWS IoT Core デバイスゲートウェイとコアデバイスに接続します。また、クライアントデバイスは、コアデバイスとの相互認証に検出情報を使用します。詳細については、the section called “デバイス接続のワークフロー”およびthe section called “Greengrass コアを使用したデバイス認証の管理”を参照してください。

X.509 証明書

コアデバイスとクライアントデバイス間、およびデバイスと AWS IoT Core または AWS IoT Greengrass 間の通信は認証する必要があります。この相互認証は、登録された X.509 デバイス証明書と暗号化キーに基づいています。

AWS IoT Greengrass 環境では、デバイスは次の Transport Layer Security (TLS) 接続にパブリックキーとプライベートキーを持つ証明書を使用します。

- インターネットに接続 AWS IoT Core し、インターネット AWS IoT Greengrass 経由で接続する Greengrass コア上の AWS IoT クライアントコンポーネント。
- インターネット経由でコア検出情報を取得 AWS IoT Greengrass するために に接続するクライアントデバイス。
- ローカルネットワークを介してグループ内のクライアントデバイスに接続する Greengrass コアにある MQTT サーバーコンポーネント。

AWS IoT Greengrass コアデバイスは証明書を 2 つの場所に保存します。

- `/greengrass-root/certs` のコアデバイス証明書。通常、コアデバイス証明書の名前は `hash.cert.pem` です (例えば、`86c84488a5.cert.pem`)。この証明書は、コアが および AWS

IoT Greengrass サービスに接続する AWS IoT Core ときに、相互認証のために AWS IoT クライアントによって使用されます。

- `/greengrass-root/ggc/var/state/server` の MQTT サーバー証明書。MQTT サーバー証明書の名前は `server.crt` です。この証明書は、ローカル MQTT サーバー (Greengrass コア上) と Greengrass デバイスとの間で相互認証に使用されます。

> **Note**
>
> *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは `/greengrass` ディレクトリです。

詳細については、「the section called “セキュリティプリンシパル”」を参照してください。

認証機関 (CA) 証明書

コアデバイスとクライアントデバイスは、 AWS IoT Core および AWS IoT Greengrass サービスによる認証に使用されるルート CA 証明書をダウンロードします。Amazon ルート CA 1 など、Amazon Trust Services (ATS) のルート CA 証明書を使用することをお勧めします。詳細については、「AWS IoT Core デベロッパーガイド」の「サーバー認証用の CA 証明書」を参照してください。

> **Note**
>
> ルート CA 証明書タイプはエンドポイントと一致する必要があります。ATS ルート CA 証明書を ATS エンドポイントで使用するか (推奨)、VeriSign ルート CA 証明書をレガシーエンドポイントで使用します。レガシーエンドポイントがサポートされるのは、アマゾン ウェブサービスの一部のリージョンに限られます。詳細については、「the section called “サービスエンドポイントは証明書タイプと一致する必要がある”」を参照してください。

また、クライアントデバイスでは、Greengrass グループ CA 証明書もダウンロードされます。これは、相互認証中に Greengrass コアにある MQTT サーバー証明書を検証するために使用されます。詳細については、「the section called “デバイス接続のワークフロー”」を参照してください。MQTT サーバー証明書のデフォルトの有効期限は 7 日間です。

ローカル MQTT サーバーの証明書ローテーション

クライアントデバイスは、ローカルの MQTT サーバー証明書を使用して、Greengrass コアデバイスとの相互認証を行います。デフォルトでは、この証明書の有効期間は 7 日です。この制限期間は、セキュリティのベストプラクティスに基づいています。MQTT サーバー証明書は、クラウド内に保存されているグループ CA 証明書によって署名されます。

証明書のローテーションを行うには、Greengrass コアデバイスがオンラインであり、 AWS IoT Greengrass サービスに直接定期的にアクセスできる必要があります。証明書の有効期限が切れると、コアデバイスは AWS IoT Greengrass サービスに接続して新しい証明書を取得しようとします。接続に成功すると、コアデバイスは新しい MQTT サーバー証明書をダウンロードし、ローカル MQTT サービスを再起動します。この時点で、コアに接続されたすべてのクライアントデバイスは切断されます。有効期限が切れた時点で Core デバイスがオフラインになっていると、代替証明書は送信されません。コアデバイスに接続しようとする新しい試みはすべて拒否されます。既存の接続は影響を受けません。クライアントデバイスは、 AWS IoT Greengrass サービスへの接続が復元され、新しい MQTT サーバー証明書をダウンロードできるようになるまで、コアデバイスに接続できません。

必要に応じて、有効期限は、7 ~ 30 日間の任意の値に設定できます。より頻繁にローテーションを行うには、より頻繁なクラウド接続が必要になります。頻繁にローテーションを行わないと、セキュリティ上の問題が発生する可能性があります。証明書の有効期限を 30 日を超える値に設定する場合は、 サポートにお問い合わせください。

AWS IoT コンソールでは、グループの設定ページで証明書を管理できます。 AWS IoT Greengrass API では、UpdateGroupCertificateConfiguration アクションを使用できます。

MQTT サーバー証明書の有効期限が切れた場合、証明書の検証はすべて失敗します。クライアントデバイスは失敗を検出し、接続を終了できる必要があります。

AWS IoT データプレーンオペレーションの ポリシー

AWS IoT ポリシーを使用して、 AWS IoT Core および AWS IoT Greengrass データプレーンへのアクセスを許可します。 AWS IoT Core データプレーンは、 への接続やトピックへのサブスクライブなど、デバイス、ユーザー AWS IoT Core 、アプリケーションのオペレーションで構成されます。 AWS IoT Greengrass データプレーンは、デプロイの取得や接続情報の更新など、Greengrass デバイスのオペレーションで構成されます。

AWS IoT ポリシーは、IAM ポリシーに似た JSON ドキュメントです。これには、次のプロパティを指定する 1 つ以上のポリシーステートメントが含まれます。

- `Effect`。アクセスモードを指定するプロパティで、`Allow` か `Deny` のどちらかになります。
- `Action`。ポリシーによって許可または拒否されるアクションのリストです。
- `Resource`。アクションが許可または拒否されるリソースのリストです。

AWS IoT ポリシーは、 をワイルドカード文字*としてサポートし、MQTT ワイルドカード文字 (+ および #) をリテラル文字列として扱います。「*」ワイルドカードの詳細については、「AWS Identity and Access Management ユーザーガイド」の「リソース ARN でのワイルドカードの使用」を参照してください。

詳細については、「AWS IoT Core デベロッパーガイド」の「AWS IoT ポリシー」と「AWS IoT ポリシーアクション」を参照してください。

> **Note**
>
> AWS IoT Core では、モノのグループに AWS IoT ポリシーをアタッチして、デバイスのグループのアクセス許可を定義できます。モノのグループポリシーは、 AWS IoT Greengrass データプレーンオペレーションへのアクセスを許可しません。 AWS IoT Greengrass データプレーンオペレーションへのモノのアクセスを許可するには、モノの証明書にアタッチする AWS IoT ポリシーに アクセス許可を追加します。

AWS IoT Greengrass ポリシーアクション

Greengrass コアアクション

AWS IoT Greengrass は、Greengrass コアデバイスがポリシーで使用できる以下の AWS IoT ポリシーアクションを定義します。

`greengrass:AssumeRoleForGroup`

Greengrass Core デバイスが、Token Exchange Service (TES) システム Lambda 関数を使用して認証情報を取得するためのアクセス許可。取得した認証情報に関連付けられているアクセス許可は、設定されたグループロールにアタッチされたポリシーに基づきます。

このアクセス許可は、Greengrass コアデバイスが認証情報を取得しようとしたときにチェックされます (認証情報がローカルにキャッシュされていないと仮定します)。

`greengrass:CreateCertificate`

Greengrass コアデバイスが独自のサーバー証明書を作成するためのアクセス許可。

このアクセス許可は、Greengrass コアデバイスが証明書を作成するときにチェックされます。Greengrass コアデバイスは、初回実行時、コアの接続情報が変更されたとき、および指定されたローテーション期間にサーバー証明書を作成しようとします。

`greengrass:GetConnectivityInfo`

Greengrass コアデバイスが、独自の接続情報を取得するためのアクセス許可。

このアクセス許可は、Greengrass コアデバイスが接続情報の取得を試みるときにチェックされます AWS IoT Core。

`greengrass:GetDeployment`

Greengrass コアデバイスがデプロイを取得するためのアクセス許可。

このアクセス許可は、Greengrass コアデバイスがクラウドからデプロイとデプロイステータスを取得しようとしたときにチェックされます。

`greengrass:GetDeploymentArtifacts`

グループ情報や Lambda 関数などのデプロイアーティファクトを取得するための Greengrass Core デバイスのアクセス許可。

このアクセス許可は、Greengrass コアデバイスがデプロイを受け取り、デプロイアーティファクトを取得しようとしたときにチェックされます。

`greengrass:UpdateConnectivityInfo`

Greengrass コアデバイスが IP またはホスト名情報を使用して自身の接続情報を更新するためのアクセス許可。

このアクセス許可は、Greengrass コアデバイスがクラウド内の接続情報を更新しようとしたときにチェックされます。

`greengrass:UpdateCoreDeploymentStatus`

Greengrass コアデバイスがデプロイの状態を更新するためのアクセス許可。

このアクセス許可は、Greengrass コアデバイスがデプロイを受け取り、デプロイメントステータスを更新しようとしたときにチェックされます。

Greengrass デバイスのアクション

AWS IoT Greengrass では、クライアントデバイスがポリシーで使用できる AWS IoT 次のポリシーアクションを定義します。

`greengrass:Discover`

クライアントデバイスが Discovery API を使用してグループのコア接続情報とグループ認証機関を取得するためのアクセス許可。

このアクセス許可は、クライアントデバイスが TLS 相互認証を使用して Discovery API を呼び出すときにチェックされます。

AWS IoT Greengrass コアデバイスの最小 AWS IoT ポリシー

次のポリシーの例には、コアデバイスの基本的な Greengrass 機能をサポートするのに必要な最小限のアクションが含まれています。

- ポリシーには、シャドウステータスに使用されるトピックを含む、コアデバイスがメッセージを発行、サブスクライブし、メッセージを受信できる、MQTT トピックとトピックのフィルターが一覧表示されます。Greengrass グループ内の AWS IoT Core Lambda 関数、コネクタ、クライアントデバイス間のメッセージ交換をサポートするには、許可するトピックとトピックフィルターを指定します。詳細については、「AWS IoT Core デベロッパーガイド」の「パブリッシュ/サブスクライブポリシーの例」を参照してください。
- このポリシーには、 がコアデバイスのシャドウを取得、更新、削除 AWS IoT Core できるようにするセクションが含まれています。Greengrass グループ内のクライアントデバイスのシャドウ同期を可能にするには、`Resource` リスト内のターゲット Amazon リソースネーム (ARN) を指定します (例: `arn:aws:iot:region:account-id:thing/device-name`)。
- AWS IoT コアデバイスのポリシーでのモノのポリシー変数 (`iot:Connection.Thing.*`) の使用はサポートされていません。コアは同じデバイス証明書を使用して に複数の接続を行います AWS IoT Core が、接続内のクライアント ID がコアモノの名前と完全に一致しない可能性があります。
- `greengrass:UpdateCoreDeploymentStatus` アクセス許可の場合、`Resource` ARN の最終セグメントは、コアデバイスの URL エンコード ARN です。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:client/core-name-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topic/$aws/things/core-name-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-name-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DeleteThingShadow"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/core-name-*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:AssumeRoleForGroup",
                "greengrass:CreateCertificate"
            ],
```

```
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetDeployment"
            ],
            "Resource": [
                "arn:aws:greengrass:region:account-id:/greengrass/groups/group-id/
deployments/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetDeploymentArtifacts"
            ],
            "Resource": [
                "arn:aws:greengrass:region:account-id:/greengrass/groups/group-id/
deployments/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:UpdateCoreDeploymentStatus"
            ],
            "Resource": [
                "arn:aws:greengrass:region:account-id:/greengrass/groups/group-id/
deployments/*/cores/arn%3Aaws%3Aiot%3Aregion%3Aaccount-id%3Athing%2Fcore-name"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetConnectivityInfo",
                "greengrass:UpdateConnectivityInfo"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/core-name-*"
            ]
        }
```

```
    ]
}
```

> **Note**
>
> AWS IoT クライアントデバイスの ポリシーには、通常、`iot:Connect`、、`iot:Publishiot:Receive`および `iot:Subscribe`アクションに同様のアクセス許可が必要です。
> クライアントデバイスが、デバイスが属する Greengrass グループのコアの接続情報を自動的に検出できるようにするには、クライアントデバイスの AWS IoT ポリシーに `greengrass:Discover`アクションを含める必要があります。`Resource` セクションには、Greengrass コアデバイスの ARN ではなく、クライアントデバイスの ARN を指定します。以下に例を示します。
>
> ```
> {
>  "Effect": "Allow",
>  "Action": [
>  "greengrass:Discover"
>  ],
>  "Resource": [
>  "arn:aws:iot:region:account-id:thing/device-name"
>  ]
> }
> ```
>
> Greengrass コアはクライアントデバイスのシャドウ同期オペレーションを処理するため、クライアントデバイスの AWS IoT ポリシーには通常`iot:GetThingShadow`、`iot:UpdateThingShadow`、、または `iot:DeleteThingShadow`アクションのアクセス許可は必要ありません。この場合、コアの AWS IoT ポリシーのシャドウアクション`Resource`のセクションに、クライアントデバイスの ARNs が含まれていることを確認してください。

AWS IoT コンソールでは、コアの証明書にアタッチされているポリシーを表示および編集できます。

1. ナビゲーションペインの[Manage] (管理)で、[All devices] (すべてのデバイス) を展開してから、[Things] (モノ) を選択します。

2. コアを選択します。
3. コアの設定ページで、[Certificates] (証明書) タブを選択します。
4. [Certificates] (証明書) タブで、証明書を選択します。
5. 証明書の設定ページで、[ポリシー] を選択し、ポリシーを選択します。

   ポリシーを編集するには、[Edit active version] (アクティブなバージョンの編集) を選択します。
6. ポリシーを確認し、必要に応じてアクセス許可を追加、削除、または編集します。
7. 新しいポリシーバージョンをアクティブなバージョンとして設定するには、[Policy version status] (ポリシーバージョンのステータス) で、[Set the edited version as the active version for this policy] (編集したバージョンをこのポリシーのアクティブバージョンとして設定) を選択します。
8. [Save as new version] (新しいバージョンとして保存) を選択します。

の ID とアクセスの管理 AWS IoT Greengrass

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つ です。IAM 管理者は、誰を認証 (サインイン) し、誰に AWS IoT Greengrass リソースの使用を許可する (アクセス許可を付与する) かを制御します。IAM は、追加料金なしで AWS のサービス 使用できる です。

> **Note**
>
> このトピックでは、IAM の概念と機能について説明します。でサポートされている IAM 機能の詳細については AWS IoT Greengrass、「」を参照してくださいthe section called “が IAM と AWS IoT Greengrass 連携する方法”。

対象者

AWS Identity and Access Management (IAM) の使用方法は、作業内容によって異なります AWS IoT Greengrass。

サービスユーザー – AWS IoT Greengrass サービスを使用してジョブを実行する場合、管理者から必要な認証情報とアクセス許可が与えられます。さらに多くの AWS IoT Greengrass 機能を使用して作業を行う場合は、追加のアクセス許可が必要になる場合があります。アクセスの管理方法を理解すると、管理者から適切な権限をリクエストするのに役に立ちます。 AWS IoT Greengrass機能に

アクセスできない場合は、「AWS IoT Greengrassのアイデンティティとアクセスの問題のトラブルシューティング」を参照してください。

サービス管理者 – 社内の AWS IoT Greengrass リソースを担当している場合は、通常、 へのフルアクセスがあります AWS IoT Greengrass。サービスユーザーがどの AWS IoT Greengrass 機能やリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。会社で IAM を で使用する方法の詳細については AWS IoT Greengrass、「」を参照してくださいが IAM と AWS IoT Greengrass 連携する方法。

IAM 管理者 - 管理者は、 AWS IoT Greengrassへのアクセスを管理するポリシーの書き込み方法の詳細について確認する場合があります。IAM で使用できる AWS IoT Greengrass アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS IoT Greengrassのアイデンティティベースのポリシーの例。

アイデンティティを使用した認証

認証は、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして、または IAM ロールを引き受けることによって認証 (にサインイン AWS) される必要があります。AWS アカウントのルートユーザー

ID ソースを介して提供された認証情報を使用して、フェデレーティッド ID AWS として にサインインできます。 AWS IAM Identity Center （ IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook 認証情報は、フェデレーティッド ID の例です。フェデレーティッド ID としてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーション AWS を使用して にアクセスすると、間接的にロールを引き受けることになります。

ユーザーのタイプに応じて、 AWS Management Console または AWS アクセスポータルにサインインできます。へのサインインの詳細については AWS、「 AWS サインイン ユーザーガイド」の「 へのサインイン方法 AWS アカウント」を参照してください。

AWS プログラムで にアクセスする場合、 は、ソフトウェア開発キット (SDK) とコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用してリクエストを暗号化して署名します。AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。リクエストに自分で署名する推奨方法の使用については、「IAM ユーザーガイド」の「API リクエストに対するAWS Signature Version 4」を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。たとえば、 では、多要素認証 (MFA) を使用してアカウントのセキュリティを向上させる AWS ことをお

勧めします。詳細については、「AWS IAM Identity Center ユーザーガイド」の「多要素認証」および「IAM ユーザーガイド」の「IAM のAWS 多要素認証」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての およびリソースへの AWS のサービス 完全なアクセス権を持つ 1 つのサインイン ID から始めます。この ID は AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることでアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、IAM ユーザーガイドの「ルートユーザー認証情報が必要なタスク」を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、1 人のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 内の ID です。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する許可を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時認証情報が提供されます。詳細については、「IAM ユーザーガイド」の「IAM ユーザーに関するユースケース」を参照してください。

IAM ロール

IAM ロールは、特定のアクセス許可 AWS アカウント を持つ 内のアイデンティティです。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。で IAM ロールを一時的に引き受けるには AWS Management Console、ユーザーから IAM ロール (コンソール) に切り替えることができます。ロールを引き受けるには、 または AWS API オペレーションを AWS CLI 呼び

出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「ロールを引き受けるための各種方法」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッド ID に許可を割り当てるには、ロールを作成してそのロールの許可を定義します。フェデレーティッド ID が認証されると、その ID はロールに関連付けられ、ロールで定義されている許可が付与されます。フェデレーションのロールについては、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。IAM Identity Center を使用する場合は、許可セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。アクセス許可セットの詳細については、「AWS IAM Identity Center User Guide」の「Permission sets」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の では AWS のサービス、 (ロールをプロキシとして使用する代わりに) ポリシーをリソースに直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。
- クロスサービスアクセス — 一部の は他の の機能 AWS のサービス を使用します AWS のサービス。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの許可、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。
  - 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用して でアクションを実行すると AWS、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、 を呼び出すプリンシパルのアクセス許可と AWS のサービス、ダウンストリームサービス AWS のサービス へのリクエストのリクエストをリクエストする を使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

- サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除することができます。詳細については、「IAM ユーザーガイド」の「AWS のサービスに許可を委任するロールを作成する」を参照してください。
- サービスにリンクされたロール – サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、 サービスによって所有されます。IAM 管理者は、サービスリンクロールのアクセス許可を表示できますが、編集することはできません。

- Amazon EC2 で実行されているアプリケーション – IAM ロールを使用して、EC2 インスタンスで実行され、 AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。EC2 インスタンスに AWS ロールを割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスにアタッチされたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して許可を付与する」を参照してください。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、ID AWS またはリソースにアタッチします。ポリシーは AWS 、アイデンティティまたはリソースに関連付けられているときにアクセス許可を定義する のオブジェクトです。 は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うときに、これらのポリシー AWS を評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。ほとんどのポリシーは JSON ドキュメント AWS として に保存されます。JSON ポリシードキュメントの構造と内容の詳細については、IAM ユーザーガイドの JSON ポリシー概要を参照してください。

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き受けることができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの許可を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。そのポリシーを持つユーザーは、 AWS Management Console、、 AWS CLIまたは AWS API からロール情報を取得できます。

アイデンティティベースのポリシー

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

アイデンティティベースのポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれています。管理ポリシーは、 内の複数のユーザー、グループ、ロールにアタッチできるスタンドアロンポリシーです AWS アカウント。管理ポリシーには、 AWS 管理ポリシーとカスタマー管理ポリシーが含まれます。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、 AWS WAF、および Amazon VPC は、ACLs。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS は、追加のあまり一般的ではないポリシータイプをサポートしています。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- アクセス許可の境界 - アクセス許可の境界は、アイデンティティベースポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとそのアクセス許可の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、アクセス許可の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。アクセス許可の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティのアクセス許可の境界」を参照してください。
- サービスコントロールポリシー (SCPs) – SCPsは、 の組織または組織単位 (OU) の最大アクセス許可を指定する JSON ポリシーです AWS Organizations。 AWS Organizations は、ビジネスが所有する複数の AWS アカウント をグループ化して一元管理するためのサービスです。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP は、各 を含むメンバーアカウントのエンティティのアクセス許可を制限します AWS アカウントのルートユーザー。Organizations と SCP の詳細については、「AWS Organizations ユーザーガイド」の「サービスコントロールポリシー (SCP)」を参照してください。
- リソースコントロールポリシー (RCP) – RCP は、所有する各リソースにアタッチされた IAM ポリシーを更新することなく、アカウント内のリソースに利用可能な最大数のアクセス許可を設定するために使用できる JSON ポリシーです。RCP は、メンバーアカウントのリソースに対するアクセス許可を制限し、組織に属するかどうかにかかわらず AWS アカウントのルートユーザー、 を含む ID に対する有効なアクセス許可に影響を与える可能性があります。RCP AWS のサービス をサポートする のリストを含む Organizations と RCPs「リソースコントロールポリシー (RCPs」を参照してください。 AWS Organizations
- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もありま

す。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。複数のポリシータイプが関係する場合にリクエストを許可するかどうか AWS を決定する方法については、IAM ユーザーガイドの「ポリシー評価ロジック」を参照してください。

関連情報

- the section called “が IAM と AWS IoT Greengrass 連携する方法”
- the section called “アイデンティティベースのポリシーの例”
- the section called “アイデンティティとアクセスの問題のトラブルシューティング”

が IAM と AWS IoT Greengrass 連携する方法

IAM を使用して へのアクセスを管理する前に AWS IoT Greengrass、 で使用できる IAM 機能を理解しておく必要があります AWS IoT Greengrass。

IAM 機能	Greengrass によってサポートされていますか。
リソースレベルのアクセス許可を持つアイデンティティベースポリシー	はい
リソースベースのポリシー	いいえ
アクセスコントロールリスト (ACL)	いいえ
タグベースの承認	いいえ
一時的な認証情報	はい
サービスリンクロール	いいえ
サービスロール	はい

他の AWS のサービスが IAM とどのように連携するかの概要については、AWS IAM ユーザーガイドの「IAM と連携する のサービス」を参照してください。

のアイデンティティベースのポリシー AWS IoT Greengrass

IAM アイデンティティベースのポリシーでは、許可または拒否されたアクションとリソース、およびアクションを許可または拒否する条件を指定できます。 は、特定のアクション、リソース、および条件キー AWS IoT Greengrass をサポートします。ポリシーで使用するすべての要素については、「IAM ユーザーガイド」の「[IAM JSON policy elements reference] (IAM JSON ポリシーエレメントのリファレンス)」を参照してください。

アクション

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連付けられた AWS API オペレーションと同じです。一致する API オペレーションのない許可のみのアクションなど、いくつかの例外があります。また、ポリシーに複数のアクションが必要なオペレーションもあります。これらの追加アクションは依存アクションと呼ばれます。

このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

のポリシーアクションは、アクションの前に `greengrass:` プレフィックス AWS IoT Greengrass を使用します。例えば、API `ListGroups`オペレーションを使用してグループを一覧表示できるようにするには AWS アカウント、ポリシーに `greengrass:ListGroups`アクションを含めます。ポリシーステートメントには、`Action` 要素または `NotAction` 要素のいずれかを含める必要があります。 AWS IoT Greengrass は、このサービスで実行できるタスクを説明する独自の一連のアクションを定義します。

1 つのステートメントで複数のアクションを指定するには、次のようにアクションをカンマで区切って全体を括弧 ([ ]) で囲って表示します。

```
"Action": [
    "greengrass:action1",
    "greengrass:action2",
    "greengrass:action3"
```

```
]
```

ワイルドカード (*) を使用して、複数のアクションを指定できます。例えば、List という単語で始まるすべてのアクションを指定するには次のアクションを含めます。

```
"Action": "greengrass:List*"
```

> **Note**
>
> サービスに対して使用可能なすべてのアクションを指定するには、ワイルドカードを使用しないことをお勧めします。ベストプラクティスとして、ポリシー内で最小限の特権と狭い範囲のアクセス許可を付与する必要があります。詳細については、「the section called “最小限のアクセス許可を付与する”」を参照してください。

AWS IoT Greengrass アクションの完全なリストについては、IAM ユーザーガイドの「 で定義されるアクション AWS IoT Greengrass」を参照してください。

リソース

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルが、どのリソースに対してどのような条件下でアクションを実行できるかということです。

Resource JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ステートメントにはResource または NotResource 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの許可と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの権限をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

次の表に、ポリシーステートメントの Resource要素で使用できる AWS IoT Greengrass リソース ARNs を示します。 AWS IoT Greengrass アクションでサポートされているリソースレベルのアクセス許可のマッピングについては、IAM ユーザーガイドの「 で定義されるアクション AWS IoT Greengrass」を参照してください。

リソース	ARN
Group	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/groups/${GroupId}`
GroupVersion	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/groups/${GroupId}/versions/${VersionId}`
CertificateAuthority	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/groups/${GroupId}/certificateauthorities/${CertificateAuthorityId}`
Deployment	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/groups/${GroupId}/deployments/${DeploymentId}`
BulkDeployment	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/bulk/deployments/${BulkDeploymentId}`
ConnectorDefinition	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/connectors/${ConnectorDefinitionId}`
ConnectorDefinitionVersion	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/connectors/${ConnectorDefinitionId}/versions/${VersionId}`
CoreDefinition	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/cores/${CoreDefinitionId}`
CoreDefinitionVersion	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/cores/${CoreDefinitionId}/versions/${VersionId}`
DeviceDefinition	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/devices/${DeviceDefinitionId}`

リソース	ARN
DeviceDefinitionVersion	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/devices/${DeviceDefinitionId}/versions/${VersionId}
FunctionDefinition	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/functions/${FunctionDefinitionId}
FunctionDefinitionVersion	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/functions/${FunctionDefinitionId}/versions/${VersionId}
LoggerDefinition	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/loggers/${LoggerDefinitionId}
LoggerDefinitionVersion	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/loggers/${LoggerDefinitionId}/versions/${VersionId}
ResourceDefinition	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/resources/${ResourceDefinitionId}
ResourceDefinitionVersion	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/resources/${ResourceDefinitionId}/versions/${VersionId}
SubscriptionDefinition	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/subscriptions/${SubscriptionDefinitionId}
SubscriptionDefinitionVersion	arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/definition/subscriptions/${SubscriptionDefinitionId}/versions/${VersionId}

リソース	ARN
ConnectivityInfo	`arn:${Partition}:greengrass:${Region}:${Account}:/greengrass/things/${ThingName}/connectivityInfo`

次の例のResource要素は、 の米国西部 (オレゴン) リージョンにあるグループの ARN を指定します AWS アカウント 123456789012。

```
"Resource": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
```

または、特定の AWS アカウント の に属するすべてのグループを指定するには AWS リージョン、グループ ID の代わりにワイルドカードを使用します。

```
"Resource": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/*"
```

一部の AWS IoT Greengrass アクション (一部のリストオペレーションなど) は、特定のリソースで実行できません。このような場合は、ワイルドカードのみを使用する必要があります。

```
"Resource": "*"
```

ステートメントで複数のリソース ARN を指定するには、次のようにアクションをカンマで区切って全体を括弧 ([ ]) で囲って表示します。

```
"Resource": [
    "resource-arn1",
    "resource-arn2",
    "resource-arn3"
]
```

ARN 形式の詳細については、の「Amazon リソースネーム (ARNs) と AWS サービス名前空間」を参照してくださいAmazon Web Services 全般のリファレンス。

条件キー

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルが、どのリソースに対してどのような条件下でアクションを実行できるかということです。

Condition 要素 (または Condition ブロック) を使用すると、ステートメントが有効な条件を指定できます。Condition 要素はオプションです。イコールや未満などの 条件演算子 を使用して条件式を作成して、ポリシーの条件とリクエスト内の値を一致させることができます。

1 つのステートメントに複数の Condition 要素を指定する場合、または 1 つの Condition 要素に複数のキーを指定する場合、 AWS では AND 論理演算子を使用してそれらを評価します。1 つの条件キーに複数の値を指定すると、 は論理ORオペレーションを使用して条件 AWS を評価します。ステートメントの権限が付与される前にすべての条件が満たされる必要があります。

条件を指定する際にプレースホルダー変数も使用できます。例えば IAM ユーザーに、IAM ユーザー名がタグ付けされている場合のみリソースにアクセスできる権限を付与することができます。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの要素: 変数およびタグ」を参照してください。

AWS は、グローバル条件キーとサービス固有の条件キーをサポートしています。すべての AWS グローバル条件キーを確認するには、「IAM ユーザーガイド」のAWS 「グローバル条件コンテキストキー」を参照してください。

AWS IoT Greengrass では、次のグローバル条件キーがサポートされています。

キー	説明
aws:CurrentTime	現在の日時の日付/時刻条件を確認してアクセスをフィルタリングします。
aws:EpochTime	エポックまたは Unix 時間の現在の日時の日付/時刻条件を確認してアクセスをフィルタリングします。
aws:MultiFactorAuthAge	リクエスト内の Multi-Factor Authentication (MFA) によって検証されたセキュリティ認証情報が MFA を使用して何秒前に発行されたかを確認して、アクセスをフィルタリングします。
aws:MultiFactorAuthPresent	現在のリクエストで一時的なセキュリティ認証情報を検証するために Multi-Factor Authentication (MFA) が使用されたかどうかを確認して、アクセスをフィルタリングします。
aws:RequestTag/${TagKey}	各必須タグで許可されている値のセットに基づいて作成リクエストをフィルタリングします。

キー	説明
`aws:ResourceTag/${TagKey}`	リソースに関連付けられているタグ値に基づいてアクションをフィルタリングします。
`aws:SecureTransport`	リクエストが SSL を使用して送信されたかどうかを確認して、アクセスをフィルタリングします。
`aws:TagKeys`	リクエスト内の必須のタグの存在に基づいて作成リクエストをフィルタリングします。
`aws:UserAgent`	リクエスタのクライアントアプリケーションによってアクセスをフィルタリングします。

条件キーの詳細については、IAM ユーザーガイド の「AWS グローバル条件コンテキストキー」 を参照してください。

例

AWS IoT Greengrass アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいthe section called “アイデンティティベースのポリシーの例”。

のリソースベースのポリシー AWS IoT Greengrass

AWS IoT Greengrass はリソースベースのポリシーをサポートしていません。

アクセスコントロールリスト (ACL)

AWS IoT Greengrass は ACLs をサポートしていません。

AWS IoT Greengrass タグに基づく認可

AWS IoT Greengrass はタグベースの認可をサポートしていません。

の IAM ロール AWS IoT Greengrass

IAM ロールは、特定のアクセス許可 AWS アカウント を持つ 内のエンティティです。

での一時的な認証情報の使用 AWS IoT Greengrass

一時的な認証情報は、フェデレーションでサインイン、IAM ロールを引き受ける、またはクロスアカウントロールを引き受けるために使用されます。一時的なセキュリティ認証情報を取得するには、AssumeRole や GetFederationToken などの AWS STS API オペレーションを呼び出します。

Greengrass コアでは、グループロールの一時的な認証情報がユーザー定義の Lambda 関数とコネクタで使用できるようになります。Lambda 関数が AWS SDK を使用する場合、 AWS SDK がこれを行うため、認証情報を取得するためのロジックを追加する必要はありません。

サービスにリンクされた役割

AWS IoT Greengrass は、サービスにリンクされたロールをサポートしていません。

サービス役割

この機能により、ユーザーに代わってサービスがサービス役割を引き受けることが許可されます。この役割により、サービスがお客様に代わって他のサービスのリソースにアクセスし、アクションを完了することが許可されます。サービス役割はIAM アカウントに表示され、アカウントによって所有されます。つまり、IAM 管理者はこの役割の権限を変更できます。ただし、それにより、サービスの機能が損なわれる場合があります。

AWS IoT Greengrass はサービスロールを使用して、ユーザーに代わって一部の AWS リソースにアクセスします。詳細については、「the section called “Greengrass サービスロール”」を参照してください。

AWS IoT Greengrass コンソールでの IAM ロールの選択

AWS IoT Greengrass コンソールでは、アカウントの IAM ロールのリストから Greengrass サービスロールまたは Greengrass グループロールを選択する必要があります。

- Greengrass サービスロールにより AWS IoT Greengrass 、 はユーザーに代わって他の サービスの AWS リソースにアクセスできます。通常、サービスロールはコンソールで作成および設定できるため、サービスロールを選択する必要はありません。詳細については、「the section called “Greengrass サービスロール”」を参照してください。
- Greengrass グループロールは、グループ内の Greengrass Lambda 関数とコネクタが AWS リソースにアクセスできるようにするために使用されます。また、ストリームを AWS サービスにエクスポートし、CloudWatch ログを書き込む AWS IoT Greengrass アクセス許可を付与することもできます。詳細については、「the section called “Greengrass グループのロール”」を参照してください。

Greengrass サービスロール

Greengrass サービスロールは、 がユーザーに代わって のサービスからリソースにアクセス AWS IoT Greengrass することを許可する AWS Identity and Access Management (IAM) AWS サービスロールです。これにより、 は AWS Lambda 関数の取得や AWS IoT シャドウの管理など、重要なタスク AWS IoT Greengrass を実行できます。

AWS IoT Greengrass が リソースにアクセスできるようにするには、Greengrass サービスロールをに関連付け AWS アカウント 、信頼されたエンティティ AWS IoT Greengrass として を指定する必要があります。ロールには、AWSGreengrassResourceAccessRolePolicy 管理ポリシー、または使用する AWS IoT Greengrass 機能に対する同等のアクセス許可を定義するカスタムポリシーを含める必要があります。このポリシーは によって AWS 維持され、 AWS IoT Greengrass が AWS リソースにアクセスするために使用する一連のアクセス許可を定義します。

間で同じ Greengrass AWS リージョンサービスロールを再利用できますが、使用するすべての AWS リージョン でアカウントに関連付ける必要があります AWS IoT Greengrass。サービスロールが現在の AWS アカウント およびリージョンに存在しない場合、グループのデプロイは失敗します。

以下のセクションでは、 AWS Management Console または で Greengrass サービスロールを作成および管理する方法を説明します AWS CLI。

- サービスロールの管理 (コンソール)
- サービスロールの管理 (CLI)

> **Note**
>
> サービスレベルのアクセスを許可するサービスロールに加えて、 グループにグループロールを割り当てることができます。 AWS IoT Greengrass グループロールは、グループ内の Greengrass Lambda 関数とコネクタが AWS サービスにアクセスする方法を制御する別の IAM ロールです。

Greengrass サービスロールの管理 (コンソール)

AWS IoT コンソールを使用すると、Greengrass サービスロールを簡単に管理できます。例えば、Greengrass グループを作成またはデプロイすると、コンソール AWS アカウント は、 がコンソールで現在選択されている の Greengrass サービスロールにアタッチ AWS リージョン されてい

るかどうかを確認します。アタッチされていない場合、コンソールによるサービスロールの作成および設定が可能です。詳細については、「the section called “Greengrass サービスロールを作成する”」を参照してください。

AWS IoT コンソールは、次のロール管理タスクに使用できます。

- Greengrass サービスロールを見つける
- Greengrass サービスロールを作成する
- Greengrass サービスロールを変更する
- Greengrass サービスロールをデタッチする

> **Note**
> コンソールにサインインするユーザーには、サービスロールを表示、作成、または変更するためのアクセス許可が必要です。

Greengrass サービスロールを見つける (コンソール)

以下のステップを使用して、現在の で AWS IoT Greengrass が使用しているサービスロールを見つけます AWS リージョン。

1. AWS IoT コンソール のナビゲーションペインから、[Settings] (設定) を選択します。
2. [Greengrass service role (Greengrass サービスロール)] セクションまでスクロールして、サービスロールとそのポリシーを表示します。

   サービスロールが表示されない場合は、コンソールによるサービスロールの作成また設定が可能です。詳細については、「Greengrass サービスロールを作成する」を参照してください。

Greengrass サービスロールを作成する (コンソール)

コンソールによるデフォルトの Greengrass サービスロールの作成と設定が可能です。このロールには以下のプロパティがあります。

プロパティ	値
名前	`Greengrass_ServiceRole`
信頼されたエンティティ	`AWS service: greengrass`
ポリシー	AWSGreengrassResourceAccessRolePolicy

> **Note**
>
> Greengrass デバイスセットアップがサービスロールを作成する場合、ロール名は `GreengrassServiceRole_random-string` です。

AWS IoT コンソールから Greengrass グループを作成またはデプロイすると、コンソールは、コンソールで現在選択されている AWS アカウント AWS リージョン で Greengrass サービスロールが に関連付けられているかどうかを確認します。そうでない場合、コンソールは がユーザーに代わって AWS のサービス AWS IoT Greengrass を読み書きすることを許可するように求めます。

許可を付与すると、コンソールでは、 AWS アカウントに `Greengrass_ServiceRole` という名前のロールがあるかどうかの確認が行われます。

- ロールが存在する場合、コンソールは現在の AWS アカウント の にサービスロールをアタッチします AWS リージョン。
- ロールが存在しない場合、コンソールはデフォルトの Greengrass サービスロールを作成し、それを現在の AWS アカウント の にアタッチします AWS リージョン。

> **Note**
>
> カスタムロールポリシーを使用してサービスロールを作成する場合は、IAM コンソールを使用してロールを作成または変更します。詳細については、「IAM ユーザーガイド」の「 AWS サービスにアクセス許可を委任するロールの作成」または「ロールの変更」を参照してください。使用する機能およびリソースの `AWSGreengrassResourceAccessRolePolicy` マネージドポリシーと同等のアクセス許可が、ロールによって付与されることを確認します。また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含

めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。
サービスロールを作成する場合は、 AWS IoT コンソールに戻り、ロールをグループにアタッチします。これは、グループの [Settings] (設定) ページにある [Greengrass service role] (Greengrass サービスロール) で行えます。

Greengrass サービスロールを変更する (コンソール)

コンソール AWS アカウント で AWS リージョン 現在選択されている で、 にアタッチする別の Greengrass サービスロールを選択するには、次の手順に従います。

1. AWS IoT コンソール のナビゲーションペインから、[Settings] (設定) を選択します。
2. [Greengrass service role] (Greengrass サービスロール) で、[Change role] (ロールの変更) を選択します。

   Greengrass サービスロールの更新ダイアログボックスが開き、信頼 AWS アカウント されたエンティティ AWS IoT Greengrass として を定義する の IAM ロールが表示されます。
3. アタッチする Greengrass サービスロールを選択します。
4. [Attach role] (ロールをアタッチする) を選択します。

> **Note**
>
> コンソールによるデフォルトの Greengrass サービスロールの作成を許可するには、リストからロールを選択する代わりに [Create role for me (ロールの作成を許可)] を選択します。`Greengrass_ServiceRole` という名前のロールが AWS アカウントにある場合、[Create role for me] (自分のロールを作成) リンクは表示されません。

Greengrass サービスロールをデタッチする (コンソール)

コンソール AWS アカウント で AWS リージョン 現在選択されている で から Greengrass サービスロールをデタッチするには、次の手順に従います。これにより、 が現在の AWS のサービスにアクセス AWS IoT Greengrass するためのアクセス許可が取り消されます AWS リージョン。

> **Important**
>
> サービスロールをデタッチすると、アクティブなオペレーションが中断される場合があります。

1. AWS IoT コンソール のナビゲーションペインから、[Settings] (設定) を選択します。
2. [Greengrass service role] (Greengrass サービスロール) で、[Detach role] (ロールのデタッチ) を選択します。
3. 確認ダイアログボックスで、[Detach] (デタッチ) を選択します。

> **Note**
>
> ロールが不要になった場合は、IAM コンソールで削除できます。詳細については、「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」を参照してください。
> 他のロールでは、 AWS IoT Greengrass が リソースにアクセスできる場合があります。ユーザーに代わってアクセス権限を引き受けることを AWS IoT Greengrass に許可するロールをすべて見つけるには、IAM コンソールの [Roles] (ロール) ページにある [Trusted entities] (信頼済みエンティティ) 列で、[AWS service: greengrass] (サービス: greengrass) を含むロールを探します。

Greengrass サービスロールの管理 (CLI)

次の手順では、 AWS CLI がインストールされ、 AWS アカウント ID を使用するように設定されていることを前提としています。詳細については、「 AWS Command Line Interface ユーザーガイド」の AWS 「コマンドラインインターフェイスのインストール」および「 の設定 AWS CLI」を参照してください。

は、次のロール管理タスク AWS CLI に使用できます。

- Greengrass サービスロールを取得する
- Greengrass サービスロールを作成する
- Greengrass サービスロールを削除する

Greengrass サービスロールを取得する (CLI)

Greengrass サービスロールが AWS アカウント の に関連付けられているかどうかを確認するには、次の手順に従います AWS リージョン。

- サービスロールを取得します。*region* を your AWS リージョン （ 例:) に置き換えますus-west-2。

  ```
  aws Greengrass get-service-role-for-account --region region
  ```

  Greengrass サービスロールが既にアカウントに関連付けられている場合、以下のロールメタデータが返されます。

  ```
  {
    "AssociatedAt": "timestamp",
    "RoleArn": "arn:aws:iam::account-id:role/path/role-name"
  }
  ```

  ロールメタデータが返されない場合は、サービスロールを作成し (存在しない場合)、 AWS リージョン内でアカウントに関連付ける必要があります。

Greengrass サービスロールを作成する (CLI)

次のステップを使用してロールを作成し、 AWS アカウントに関連付けます。

IAM を使用して、サービスロールを作成するには

1. がロールを引き受けること AWS IoT Greengrass を許可する信頼ポリシーを使用してロールを作成します。この例では、`Greengrass_ServiceRole` という名前のロールを作成しますが、別の名前を使用できます。また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含めて、混乱した代理によるセキュ

リティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

Linux, macOS, or Unix

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        }
      }
    }
  ]
}'
```

Windows command prompt

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-
policy-document "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Effect
\":\"Allow\",\"Principal\":{\"Service\":\"greengrass.amazonaws.com\"},
\"Action\":\"sts:AssumeRole\",\"Condition\":{\"ArnLike\":{\"aws:SourceArn
\":\"arn:aws:greengrass:region:account-id:*\"},\"StringEquals\":
{\"aws:SourceAccount\":\"account-id\"}}}]}"
```

2. 出力のロールメタデータからロールの ARN をコピーします。ARN を使用して、ロールをアカウントに関連付けます。
3. `AWSGreengrassResourceAccessRolePolicy` ポリシーをロールにアタッチします。

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

サービスロールを に関連付けるには AWS アカウント

- ロールとアカウントを関連付けます。*role-arn* をサービスロール ARN に置き換え、*region* を に置き換えます AWS リージョン（例: `us-west-2`）。

  ```
  aws greengrass associate-service-role-to-account --role-arn role-arn --
  region region
  ```

  成功すると、以下のレスポンスが返されます。

  ```
  {
    "AssociatedAt": "timestamp"
  }
  ```

Greengrass サービスロールを削除する (CLI)

次のステップを使用して、Greengrass サービスロールの関連付けを AWS アカウントから解除します。

- アカウントからサービスロールの関連付けを解除します。*region* を your AWS リージョン（例: ）に置き換えます`us-west-2`。

  ```
  aws greengrass disassociate-service-role-from-account --region region
  ```

  成功すると、以下のレスポンスが返されます。

  ```
  {
    "DisassociatedAt": "timestamp"
  }
  ```

> **Note**
>
> サービスロールを使用していない場合は、削除する必要があります AWS リージョン。最初に、delete-role-policy を使用して `AWSGreengrassResourceAccessRolePolicy` 管理ポリシーをロールからデタッチし、次に delete-role を使用してロールを削除します。詳細については、「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」を参照してください。

関連情報

- IAM ユーザーガイドの AWS サービスにアクセス許可を委任するロールの作成
- 「IAM ユーザーガイド」の「ロールの修正」
- 「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」
- AWS IoT Greengrass コマンドAWS CLI リファレンスの コマンド
  - associate-service-role-to-account
  - disassociate-service-role-from-account
  - get-service-role-for-account
- AWS CLI コマンドリファレンスで使用可能な IAM コマンド
  - attach-role-policy
  - create-role
  - delete-role
  - delete-role-policy

Greengrass グループのロール

Greengrass グループロールは、Greengrass コアで実行されているコードが AWS リソースにアクセスすることを許可する IAM ロールです。 AWS Identity and Access Management (IAM) でロールを作成し、アクセス許可を管理し、そのロールを Greengrass グループにアタッチします。Greengrass グループには 1 つのグループロールがあります。アクセス許可を追加または変更するには、異なるロールをアタッチするか、ロールにアタッチされている IAM ポリシーを変更します。

ロールは、 を信頼されたエンティティ AWS IoT Greengrass として定義する必要があります。ビジネス事例によっては、グループロールに次の項目を定義する IAM ポリシーが含まれている場合があります。

- ユーザー定義の Lambda 関数が AWS サービスにアクセスするためのアクセス許可。
- コネクタが AWS サービスにアクセスするためのアクセス許可。
- ストリームマネージャーがストリームを AWS IoT Analytics および Kinesis Data Streams にエクスポートするためのアクセス許可。
- CloudWatch ログ記録を許可するためのアクセス許可。

以下のセクションでは、 AWS Management Console または で Greengrass グループロールをアタッチまたはデタッチする方法について説明します AWS CLI。

- グループロールの管理 (コンソール)
- グループロールの管理 (CLI)

> **Note**
>
> Greengrass コアからのアクセスを許可するグループロールに加えて、 がユーザーに代わって AWS リソースにアクセス AWS IoT Greengrass できるようにする Greengrass サービスロールを割り当てることができます。

Greengrass グループロールの管理 (コンソール)

AWS IoT コンソールは、次のロール管理タスクに使用できます。

- Greengrass グループロールを見つける
- Greengrass グループロールの追加または変更
- Greengrass グループロールの削除

> **Note**
>
> コンソールにサインインするユーザーには、ロールを管理するアクセス許可が必要です。

Greengrass グループロールを見つける (コンソール)

Greengrass グループにアタッチされるロールを見つけるには、以下の手順に従います。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. グループの設定ページで、[View settings] (設定を表示) を選択します。

ロールがグループにアタッチされている場合は、[Group role] (グループのロール) の下に表示されます。

Greengrass グループロールの追加または変更 (コンソール)

から Greengrass グループ AWS アカウント に追加する IAM ロールを選択するには、次の手順に従います。

グループロールには、次の要件があります。

- AWS IoT Greengrass 信頼されたエンティティとして定義されます。
- ロールにアタッチされたアクセス許可ポリシーは、グループ内の Lambda 関数とコネクタ、および Greengrass システムコンポーネントに必要なアクセス許可を AWS リソースに付与する必要があります。

> **Note**
>
> また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。「混乱した代理」問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

IAM コンソールを使用して、ロールとそのアクセス許可を作成および設定します。Amazon DynamoDB テーブルへのアクセスを許可するロールの例を作成する手順については、「the section

called “グループロールの設定”」を参照してください。一般的な手順については、IAM ユーザーガイドの「 AWS サービス (コンソール) のロールの作成」を参照してください。

ロールを設定したら、 AWS IoT コンソールを使用してロールをグループに追加します。

> **Note**
>
> この手順は、グループのロールを選択する場合にのみ必要です。現在選択されているグループロールのアクセス許可を変更した後は不要です。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. グループの設定ページで、[View settings] (設定を表示) を選択します。
4. [Group role] (グループロール) で、次のロールの追加または変更を選択します。
   - ロールを追加するには、[Associate role] (ロールに関連付ける) を選択し、次に、役割のリストから自分の役割を選択します。これらは、信頼 AWS アカウント されたエンティティ AWS IoT Greengrass として を定義する のロールです。
   - 別の役割を選択するには、[Edit role] (ロールを編集) を選択し、次に、役割のリストから自分の役割を選択します。
5. [Save] を選択します。

Greengrass グループロールの削除 (コンソール)

Greengrass グループからロールをデタッチするには、以下の手順に従います。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. グループの設定ページで、[View settings] (設定を表示) を選択します。
4. [Group role] から、[Disassociate role] (ロールの関連付けを解除する) を選択します。

5. 確認ダイアログボックスで、[Disassociate role] (ロールの関連付けを解除) を選択します。この手順では、グループからロールが削除されますが、ロールは削除されません。ロールを削除する場合は、IAM コンソールを使用します。

Greengrass グループロールの管理 (CLI)

は、次のロール管理タスク AWS CLI に使用できます。

- Greengrass グループロールの取得
- Greengrass グループロールの作成
- Greengrass グループロールの削除

Greengrass グループロールの取得 (CLI)

Greengrass グループに関連付けられたロールがあるかどうかを確認するには、以下の手順に従います。

1. グループのリストからターゲットグループの ID を取得します。

```
aws greengrass list-groups
```

以下に、`list-groups` 応答の例を示します。応答の各グループには、グループ ID を含む `Id` プロパティが含まれます。

```
{
    "Groups": [
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/00dedaaa-ac16-484d-ad77-c3eedEXAMPLE/versions/4cbc3f07-fc5e-48c4-
a50e-7d356EXAMPLE",
            "Name": "MyFirstGroup",
            "LastUpdatedTimestamp": "2019-11-11T05:47:31.435Z",
            "LatestVersion": "4cbc3f07-fc5e-48c4-a50e-7d356EXAMPLE",
            "CreationTimestamp": "2019-11-11T05:47:31.435Z",
            "Id": "00dedaaa-ac16-484d-ad77-c3eedEXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/groups/00dedaaa-
ac16-484d-ad77-c3eedEXAMPLE"
```

```
        },
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE/versions/8fe9e8ec-64d1-4647-
b0b0-01dc8EXAMPLE",
            "Name": "GreenhouseSensors",
            "LastUpdatedTimestamp": "2020-01-07T19:58:36.774Z",
            "LatestVersion": "8fe9e8ec-64d1-4647-b0b0-01dc8EXAMPLE",
            "CreationTimestamp": "2020-01-07T19:58:36.774Z",
            "Id": "036ceaf9-9319-4716-ba2a-237f9EXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE"
        },
        ...
    ]
}
```

query オプションを使用して結果をフィルタリングする例など、詳細については、「the section called “グループ ID の取得”」を参照してください。

2. 出力からターゲットグループの Id をコピーします。
3. グループロールを取得します。*group-id* をターゲットグループの ID に置き換えます。

```
aws greengrass get-associated-role --group-id group-id
```

ロールが Greengrass グループに関連付けられている場合、次のロールメタデータが返されます。

```
{
  "AssociatedAt": "timestamp",
  "RoleArn": "arn:aws:iam::account-id:role/path/role-name"
}
```

グループにロールが関連付けられていない場合は、次のエラーが返されます。

```
An error occurred (404) when calling the GetAssociatedRole operation: You need to
 attach an IAM role to this deployment group.
```

Greengrass グループロールの作成 (CLI)

以下の手順に従ってロールを作成し、それを Greengrass グループに関連付けます。

IAM を使用してグループロールを作成するには

1. がロールを引き受けること AWS IoT Greengrass を許可する信頼ポリシーを使用してロールを作成します。この例では、`MyGreengrassGroupRole` という名前のロールを作成しますが、別の名前を使用できます。また、信頼ポリシーには、`aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーも含めて、混乱した代理によるセキュリティ問題を防止することをお勧めします。条件コンテキストキーを使用すると、指定したアカウントと Greengrass ワークスペースからのリクエストのみを許可するようにアクセスを制限できます。混乱した代理に関する問題の詳細については、「サービス間の混乱した代理の防止」を参照してください。

   Linux, macOS, or Unix

```
aws iam create-role --role-name MyGreengrassGroupRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:/greengrass/
groups/group-id"
        }
      }
    }
  ]
}'
```

Windows command prompt

```
aws iam create-role --role-name MyGreengrassGroupRole --assume-role-
policy-document "{\"Version\":\"2012-10-17\",\"Statement\":[{\"Effect
\":\"Allow\",\"Principal\":{\"Service\":\"greengrass.amazonaws.com\"},
\"Action\":\"sts:AssumeRole\",\"Condition\":{\"ArnLike\":{\"aws:SourceArn
\":\"arn:aws:greengrass:region:account-id:/greengrass/groups/group-id\"},
\"StringEquals\":{\"aws:SourceAccount\":\"account-id\"}}}]}"
```

2. 出力のロールメタデータからロールの ARN をコピーします。ARN を使用して、ロールをグループに関連付けます。
3. 管理ポリシーまたはインラインポリシーをビジネスケースをサポートするロールにアタッチします。例えば、ユーザー定義 Lambda 関数が Amazon S3 から読み取る場合、`AmazonS3ReadOnlyAccess` 管理ポリシーをロールにアタッチできます。

```
aws iam attach-role-policy --role-name MyGreengrassGroupRole --policy-arn
 arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
```

成功した場合、応答は返されません。

ロールを Greengrass グループに関連付けるには

1. グループのリストからターゲットグループの ID を取得します。

```
aws greengrass list-groups
```

以下に、`list-groups` 応答の例を示します。応答の各グループには、グループ ID を含む `Id` プロパティが含まれます。

```
{
    "Groups": [
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/00dedaaa-ac16-484d-ad77-c3eedEXAMPLE/versions/4cbc3f07-fc5e-48c4-
a50e-7d356EXAMPLE",
            "Name": "MyFirstGroup",
            "LastUpdatedTimestamp": "2019-11-11T05:47:31.435Z",
```

```
            "LatestVersion": "4cbc3f07-fc5e-48c4-a50e-7d356EXAMPLE",
            "CreationTimestamp": "2019-11-11T05:47:31.435Z",
            "Id": "00dedaaa-ac16-484d-ad77-c3eedEXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/groups/00dedaaa-
ac16-484d-ad77-c3eedEXAMPLE"
        },
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE/versions/8fe9e8ec-64d1-4647-
b0b0-01dc8EXAMPLE",
            "Name": "GreenhouseSensors",
            "LastUpdatedTimestamp": "2020-01-07T19:58:36.774Z",
            "LatestVersion": "8fe9e8ec-64d1-4647-b0b0-01dc8EXAMPLE",
            "CreationTimestamp": "2020-01-07T19:58:36.774Z",
            "Id": "036ceaf9-9319-4716-ba2a-237f9EXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE"
        },
        ...
    ]
}
```

query オプションを使用して結果をフィルタリングする例など、詳細については、「the section called “グループ ID の取得”」を参照してください。

2. 出力からターゲットグループの Id をコピーします。
3. ロールとグループを関連付けます。*group-id* をターゲットグループの ID に置き換え、*role-arn* をグループロールの ARN に置き換えます。

```
aws greengrass associate-role-to-group --group-id group-id --role-arn role-arn
```

成功すると、以下のレスポンスが返されます。

```
{
  "AssociatedAt": "timestamp"
}
```

Greengrass グループロールの削除 (CLI)

以下の手順に従って、グループロールから Greengrass グループの関連付けを解除します。

1. グループのリストからターゲットグループの ID を取得します。

```
aws greengrass list-groups
```

以下に、`list-groups` 応答の例を示します。応答の各グループには、グループ ID を含む `Id` プロパティが含まれます。

```
{
    "Groups": [
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/00dedaaa-ac16-484d-ad77-c3eedEXAMPLE/versions/4cbc3f07-fc5e-48c4-
a50e-7d356EXAMPLE",
            "Name": "MyFirstGroup",
            "LastUpdatedTimestamp": "2019-11-11T05:47:31.435Z",
            "LatestVersion": "4cbc3f07-fc5e-48c4-a50e-7d356EXAMPLE",
            "CreationTimestamp": "2019-11-11T05:47:31.435Z",
            "Id": "00dedaaa-ac16-484d-ad77-c3eedEXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/groups/00dedaaa-
ac16-484d-ad77-c3eedEXAMPLE"
        },
        {
            "LatestVersionArn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE/versions/8fe9e8ec-64d1-4647-
b0b0-01dc8EXAMPLE",
            "Name": "GreenhouseSensors",
            "LastUpdatedTimestamp": "2020-01-07T19:58:36.774Z",
            "LatestVersion": "8fe9e8ec-64d1-4647-b0b0-01dc8EXAMPLE",
            "CreationTimestamp": "2020-01-07T19:58:36.774Z",
            "Id": "036ceaf9-9319-4716-ba2a-237f9EXAMPLE",
            "Arn": "arn:aws:us-west-2:123456789012:/greengrass/
groups/036ceaf9-9319-4716-ba2a-237f9EXAMPLE"
        },
        ...
    ]
}
```

query オプションを使用して結果をフィルタリングする例など、詳細については、「the section called “グループ ID の取得”」を参照してください。

2. 出力からターゲットグループの Id をコピーします。
3. グループからロールの関連付けを解除します。*group-id* をターゲットグループの ID に置き換えます。

```
aws greengrass disassociate-role-from-group --group-id group-id
```

成功すると、以下のレスポンスが返されます。

```
{
  "DisassociatedAt": "timestamp"
}
```

> **Note**
> グループロールを使用していない場合は、そのグループロールを削除できます。最初に、delete-role-policy を使用して各管理ポリシーをロールからデタッチし、次に delete-role を使用してロールを削除します。詳細については、「IAM ユーザーガイド」の「ロールまたはインスタンスプロファイルを削除する」を参照してください。

関連情報

- 「IAM ユーザーガイド」の関連トピック
  - AWS サービスにアクセス許可を委任するロールの作成
  - ロールの修正
  - IAM ID のアクセス許可の追加および削除
  - ロールまたはインスタンスプロファイルの削除
- AWS IoT Greengrass コマンドAWS CLI リファレンスの コマンド
  - list-groups
  - associate-role-to-group
  - disassociate-role-from-group
  - get-associated-role

- 「AWS CLI コマンドリファレンス」の IAM コマンド
  - attach-role-policy
  - create-role
  - delete-role
  - delete-role-policy

サービス間の混乱した代理の防止

混乱した代理問題は、アクションを実行する許可を持たないエンティティが、より特権のあるエンティティにアクションを実行するように強制できるセキュリティの問題です。では AWS、サービス間のなりすましにより、混乱した代理問題が発生する可能性があります。サービス間でのなりすましは、1 つのサービス (呼び出し元サービス) が、別のサービス (呼び出し対象サービス) を呼び出すときに発生する可能性があります。呼び出し元サービスは、本来ならアクセスすることが許可されるべきではない方法でその許可を使用して、別のお客様のリソースに対する処理を実行するように操作される場合があります。これを防ぐため、 AWS では、アカウントのリソースへのアクセス権が付与されたサービスプリンシパルで、すべてのサービスのデータを保護するために役立つツールを提供しています。

リソースポリシーで `aws:SourceArn`および `aws:SourceAccount` グローバル条件コンテキストキーを使用して、 がリソースに別のサービス AWS IoT Greengrass に付与するアクセス許可を制限することをお勧めします。両方のグローバル条件コンテキストキーを同じポリシーステートメントで使用する場合は、`aws:SourceAccount` 値と、`aws:SourceArn` 値に含まれるアカウントが、同じアカウント ID を示している必要があります。

`aws:SourceArn` の値は、`sts:AssumeRole` リクエストに関連付けられている Greengrass のカスタマーリソースである必要があります。

混乱した代理問題から保護するための最も効果的な方法は、リソースの完全な ARN を指定して `aws:SourceArn` グローバル条件コンテキストキーを使用することです。リソースの完全な ARN が不明な場合や、複数のリソースを指定する場合は、`aws:SourceArn` グローバルコンテキスト条件キーを使用して、ARN の未知部分をワイルドカード (*) で表します。例えば、`arn:aws:greengrass:`*`region`*`:`*`account-id`*`:*` と指定します。

グローバル条件コンテキストキーの `aws:SourceArn` と `aws:SourceAccount` を使用するポリシーの例については、以下の各項目を参照してください。

- Greengrass サービスロールを作成する

- Greengrass グループロールの作成
- 一括デプロイ用の IAM 実行ロールを作成および設定する

AWS IoT Greengrassのアイデンティティベースのポリシーの例

デフォルトでは、IAM ユーザーおよびロールには、 AWS IoT Greengrass リソースを作成または変更するアクセス許可はありません。また、 AWS Management Console、 AWS CLI、または AWS API を使用してタスクを実行することはできません。IAM 管理者は、ユーザーとロールに必要な、指定されたリソースで特定の API オペレーションを実行する権限をユーザーとロールに付与する IAM ポリシーを作成する必要があります。続いて、管理者はそれらの権限が必要な IAM ユーザーまたはグループにそのポリシーをアタッチする必要があります。

ポリシーに関するベストプラクティス

アイデンティティベースのポリシーは、アカウント内の AWS IoT Greengrass リソースを誰かが作成、アクセス、または削除できるかどうかを決定します。これらのアクションを実行すると、 AWS アカウントに料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS 管理ポリシーを開始し、最小特権のアクセス許可に移行 – ユーザーとワークロードにアクセス許可の付与を開始するには、多くの一般的なユースケースにアクセス許可を付与するAWS 管理ポリシーを使用します。これらは で使用できます AWS アカウント。ユースケースに固有の AWS カスタマー管理ポリシーを定義することで、アクセス許可をさらに減らすことをお勧めします。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「ジョブ機能の AWS マネージドポリシー」を参照してください。
- 最小特権を適用する – IAM ポリシーで許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーとアクセス許可」を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する - ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。条件を使用して、 などの特定の を通じてサービスアクションが使用される場合に AWS のサービス、サービスアクションへのアクセスを許可することもできます AWS CloudFormation。詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素:条件」を参照してください。

- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer でポリシーを検証する」を参照してください。
- 多要素認証 (MFA) を要求する – で IAM ユーザーまたはルートユーザーを必要とするシナリオがある場合は AWS アカウント、セキュリティを強化するために MFA を有効にします。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、「IAM ユーザーガイド」の「MFA を使用した安全な API アクセス」を参照してください。

IAM でのベストプラクティスの詳細については、IAM ユーザーガイドの IAM でのセキュリティのベストプラクティスを参照してください。

AWS の マネージドポリシー AWS IoT Greengrass

AWS IoT Greengrass は、IAM ユーザーとロールにアクセス許可を付与するために使用できる以下の AWS 管理ポリシーを保持します。

ポリシー	説明
AWSGreengrassFullAccess	すべての AWS リソースに対するすべての AWS IoT Greengrass アクションを許可します。このポリシーは、 AWS IoT Greengrass サービス管理者またはテスト目的で推奨されます。
AWSGreengrassReadOnlyAccess	すべての AWS リソースに対して `List`および `Get` AWS IoT Greengrass アクションを許可します。
AWSGreengrassResourceAccessRolePolicy	や AWS IoT Device Shadow などの AWS AWS Lambda サービスからのリソースへのアクセスを許可します。これは、Greengrass サービスロールに使用されるデフォルトのポリシーです。このポリシーは、一般的なアクセスしやすさ

ポリシー	説明
	を実現するように設計されています。より制限の厳しいカスタムポリシーを定義できます。
GreengrassOTAUpdateArtifactAccess	すべての で AWS IoT Greengrass Core ソフトウェアのover-the-air (OTA) 更新アーティファクトへの読み取り専用アクセスを許可します AWS リージョン。

ポリシーの例

以下のお客様定義のポリシーは、一般的なシナリオのアクセス許可を付与します。

例

- 自分の権限の表示をユーザーに許可する

これらの JSON ポリシードキュメント例を使用して IAM のアイデンティティベースのポリシーを作成する方法については、『IAM ユーザーガイド』の「JSON タブでのポリシーの作成」を参照してください。

自分の権限の表示をユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI または AWS API を使用してプログラムでこのアクションを実行するアクセス許可が含まれています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
```

```
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS IoT Greengrassのアイデンティティとアクセスの問題のトラブルシューティング

次の情報は、 および IAM の使用時に発生する可能性がある一般的な問題の診断 AWS IoT Greengrass と修正に役立ちます。

問題

- でアクションを実行する権限がありません AWS IoT Greengrass
- エラー: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account.
- 次のエラーが発生する。Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/core/<architecture>/greengrass-core-<distribution-version>.tar.gz
- デバイスシャドウがクラウドと同期していない。
- iam: PassRole を実行する権限がない
- 管理者として、他のユーザーにアクセスを許可したい AWS IoT Greengrass

- 自分の 以外のユーザーに自分の AWS IoT Greengrass リソース AWS アカウント へのアクセスを許可したい

一般的なトラブルシューティングヘルプについては、「トラブルシューティング」を参照してください。

でアクションを実行する権限がありません AWS IoT Greengrass

アクションを実行する権限がないというエラーが表示された場合、管理者に問い合わせてサポートを依頼する必要があります。お客様のユーザー名とパスワードを発行したのが、担当の管理者です。

以下の例のエラーは、`mateojackson` IAM ユーザーがコア定義バージョンの詳細を表示しようとしているが、`greengrass:GetCoreDefinitionVersion` アクセス許可がない場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 greengrass:GetCoreDefinitionVersion on resource: resource: arn:aws:greengrass:us-
west-2:123456789012:/greengrass/definition/cores/78cd17f3-bc68-ee18-47bd-5bda5EXAMPLE/
versions/368e9ffa-4939-6c75-859c-0bd4cEXAMPLE
```

この場合、Mateo は、`greengrass:GetCoreDefinitionVersion` アクションを使用して `arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/cores/78cd17f3-bc68-ee18-47bd-5bda5EXAMPLE/versions/368e9ffa-4939-6c75-859c-0bd4cEXAMPLE` リソースへのアクセスが許可されるように、管理者にポリシーの更新を依頼します。

エラー: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account.

解決策: デプロイに失敗した場合に、このエラーが表示されることがあります。Greengrass サービスロールが、現在の AWS リージョンの AWS アカウント に関連付けられていることを確認します。詳細については、the section called “サービスロールの管理 (CLI) ” または the section called “サービスロールの管理 (コンソール) ” を参照してください。

次のエラーが発生する。Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/core/<architecture>/greengrass-core-<distribution-version>.tar.gz

解決策: 無線通信 (OTA) での更新が失敗した場合に、このエラーが表示されることがあります。署名者ロールポリシーで、ターゲット AWS リージョン を `Resource` として追加します。この署名者ロールは、 AWS IoT Greengrass ソフトウェア更新の S3 URL に事前署名するために使用されます。詳細については、「S3 URL の署名者ロール」を参照してください。

デバイスシャドウがクラウドと同期していない。

解決策： に Greengrass サービスロールの `iot:UpdateThingShadow` および `iot:GetThingShadow`アクションに対するアクセス許可 AWS IoT Greengrass があることを確認します。サービスロールで `AWSGreengrassResourceAccessRolePolicy` 管理ポリシーを使用している場合は、これらのアクセス許可はデフォルトで含まれています。

「シャドウ同期タイムアウト問題のトラブルシューティング」を参照してください。

以下は、 AWS IoT Greengrassを操作するときに発生する可能性がある一般的な IAM の問題です。

iam: PassRole を実行する権限がない

`iam:PassRole` アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して AWS IoT Greengrassにロールを渡すことができるようにする必要があります。

一部の AWS のサービス では、新しいサービスロールまたはサービスにリンクされたロールを作成する代わりに、そのサービスに既存のロールを渡すことができます。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、`marymajor` という IAM ユーザーがコンソールを使用して AWS IoT Greengrassでアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。メアリーには、ロールをサービスに渡す許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに `iam:PassRole` アクションの実行を許可する必要があります。

サポートが必要な場合は、 AWS 管理者にお問い合わせください。サインイン認証情報を提供した担当者が管理者です。

管理者として、他のユーザーにアクセスを許可したい AWS IoT Greengrass

他のユーザーにアクセスを許可するには AWS IoT Greengrass、アクセスを必要とするユーザーまたはアプリケーションにアクセス許可を付与する必要があります。 AWS IAM Identity Center を使用してユーザーとアプリケーションを管理する場合は、アクセスレベルを定義するアクセス許可セットをユーザーまたはグループに割り当てます。アクセス許可セットは、ユーザーまたはアプリケーションに関連付けられている IAM ロールに自動的に IAM ポリシーを作成して割り当てます。詳細については、「AWS IAM Identity Center ユーザーガイド」の「アクセス許可セット」を参照してください。

IAM アイデンティティセンターを使用していない場合は、アクセスを必要としているユーザーまたはアプリケーションの IAM エンティティ (ユーザーまたはロール) を作成する必要があります。次に、 AWS IoT Greengrassの適切なアクセス許可を付与するポリシーを、そのエンティティにアタッチする必要があります。アクセス許可が付与されたら、ユーザーまたはアプリケーション開発者に認証情報を提供します。これらの認証情報を使用して AWSにアクセスします。IAM ユーザー、グループ、ポリシー、アクセス許可の作成の詳細については、「IAM ユーザーガイド」の「IAM アイデンティティ」と「IAM のポリシーとアクセス許可」を参照してください。

自分の 以外のユーザーに自分の AWS IoT Greengrass リソース AWS アカウント へのアクセスを許可したい

他のアカウントのユーザーや組織外のユーザーが AWS リソースにアクセスするために使用できる IAM ロールを作成できます。ロールを引き受けるように信頼されたユーザーを指定することができます。詳細については、「IAM ユーザーガイド」の「所有 AWS アカウント している別の の IAM ユーザーへのアクセスを許可する」および「サードパーティーが所有する Amazon Web Services アカウントへのアクセスを許可する」を参照してください。

AWS IoT Greengrass は、リソースベースのポリシーまたはアクセスコントロールリスト (ACLs) に基づくクロスアカウントアクセスをサポートしていません。

AWS IoT Greengrassのコンプライアンス検証

AWS のサービス が特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、AWS のサービス 「コンプライアンスプログラムによる対象範囲内」を参照して、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、AWS 「コンプライアンスプログラム」を参照してください。

を使用して、サードパーティーの監査レポートをダウンロードできます AWS Artifact。詳細については、「Downloading Reports in AWS Artifact」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービス は、お客様のデータの機密性、貴社のコンプライアンス目的、適用される法律および規制によって決まります。 は、コンプライアンスに役立つ以下のリソース AWS を提供します。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスの機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべて AWS のサービス HIPAA の対象となるわけではありません。
- AWS コンプライアンスリソース – このワークブックとガイドのコレクションは、お客様の業界と地域に適用される場合があります。
- AWS カスタマーコンプライアンスガイド – コンプライアンスの観点から責任共有モデルを理解します。このガイドでは、複数のフレームワーク (米国国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティコントロールを保護し、そのガイダンスに AWS のサービス マッピングするためのベストプラクティスをまとめています。
- 「 デベロッパーガイド」の「ルールによるリソースの評価」 – この AWS Config サービスは、リソース設定が内部プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。 AWS Config
- AWS Security Hub – これにより AWS のサービス 、 内のセキュリティ状態を包括的に把握できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。
- Amazon GuardDuty – 環境をモニタリングして AWS アカウント、疑わしいアクティビティや悪意のあるアクティビティがないか調べることで、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager – これにより AWS のサービス 、 AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

の耐障害性 AWS IoT Greengrass

AWS グローバルインフラストラクチャは、Amazon Web Services リージョンとアベイラビリティーゾーンを中心に構築されています。各 AWS リージョン は、低レイテンシー、高スループット、および冗長性の高いネットワークで接続された、物理的に分離された複数のアベイラビリティーゾーンを提供します。アベイラビリティーゾーンでは、ゾーン間で中断することなく自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性が高く、フォールトトレラントで、スケーラブルです。

アマゾン ウェブ サービスリージョンとアベイラビリティーゾーンの詳細については、「AWS グローバルインフラストラクチャ」を参照してください。

グローバル AWS インフラストラクチャに加えて、 AWS IoT Greengrass には、データの耐障害性とバックアップのニーズをサポートするためにいくつかの機能が用意されています。

- コアがインターネット接続から切断された場合、クライアントデバイスはローカルネットワークを介して通信を続けることができます。
- インメモリストレージではなくローカルストレージキャッシュに AWS クラウド ターゲット宛ての未処理のメッセージを保存するようにコアを設定できます。ローカルストレージキャッシュはコアの再起動中 (グループのデプロイ後やデバイスの再起動後など) に保持できるため、 AWS IoT Greengrass は引き続き送信先メッセージを処理します AWS IoT Core。詳細については、「the section called “MQTT メッセージキュー”」を参照してください。
- AWS IoT Core メッセージブローカーとの永続セッションを確立するようにコアを設定できます。これにより、コアは、オフラインのときに送信されたメッセージを受信できます。詳細については、「the section called “AWS IoT Coreを使用した MQTT 永続セッション”」を参照してください。
- ログをローカルファイルシステムおよび CloudWatch Logs に書き込むよう Greengrass グループを設定できます。コアが接続から切断された場合、ローカルロギングは続行できますが、CloudWatch のログ送信の再試行回数は制限されます。再試行回数の上限に達すると、イベントは削除されます。また、ログ記録の制限にも注意する必要があります。
- ストリームマネージャーのストリームを読み込み、ローカルストレージの送信先にデータを送る Lambda 関数を作成できます。

のインフラストラクチャセキュリティ AWS IoT Greengrass

マネージドサービスである AWS IoT Greengrass は、 AWS グローバルネットワークセキュリティで保護されています。 AWS セキュリティサービスと がインフラストラクチャ AWS を保護する方法については、AWS 「 クラウドセキュリティ」を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して AWS 環境を設計するには、「Security Pillar AWS Well-Architected Framework」の「Infrastructure Protection」を参照してください。

AWS 公開された API コールを使用して、ネットワーク AWS IoT Greengrass 経由で にアクセスします。クライアントは以下をサポートする必要があります。

- Transport Layer Security (TLS)。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- DHE (楕円ディフィー・ ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードはJava 7 以降など、ほとんどの最新システムでサポートされています。

また、リクエストにはアクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。またはAWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

AWS IoT Greengrass 環境では、デバイスは X.509 証明書と暗号化キーを使用して に接続し、認証します AWS クラウド。詳細については、「the section called “デバイス認証と認可”」を参照してください。

の設定と脆弱性の分析 AWS IoT Greengrass

IoT 環境は、多様な機能を持ち、存続期間が長く、地理的に分散される多数のデバイスで設定されることがあります。このような特性によってデバイスのセットアップが複雑になり、エラーを起こしやすくなります。また、デバイスの計算能力、メモリ、ストレージの機能には制約があるため、デバイス自体での暗号化や他の形式のセキュリティの使用が制限されます。さらに、デバイスは多く場合、既知の脆弱性を持つソフトウェアを使用しています。このような要素が原因で、IoT デバイスはハッカーの魅力的なターゲットとなり、継続的に保護することが困難になっています。

AWS IoT Device Defender セキュリティの問題やベストプラクティスからの逸脱を特定するためのツールを提供することで、これらの課題に対処します。 AWS IoT Device Defender を使用して、接続されたデバイスの分析、監査、モニタリングを行い、異常な動作を検出し、セキュリティリスクを軽減できます。 AWS IoT Device Defender は、デバイスを監査して、セキュリティのベストプラク

ティスに準拠していることを確認して、デバイスの異常な動作を検出できます。これにより、デバイス間でセキュリティポリシーを維持し、デバイスが侵害された場合にはすばやく応答することができます。との接続で AWS IoT Core、 は AWS IoT Device Defender 機能で使用できる予測可能なクライアント IDs AWS IoT Greengrass を生成します。詳細については、「AWS IoT Core デベロッパーガイド」の 「AWS IoT Device Defender」を参照してください。

AWS IoT Greengrass 環境では、次の考慮事項に注意してください。

- 物理デバイス、デバイスのファイルシステム、ローカルネットワークの保護はお客様の責任です。
- AWS IoT Greengrass は、Greengrass コンテナで実行されるかどうかにかかわらず、ユーザー定義の Lambda 関数にネットワーク分離を適用しません。したがって、Lambda 関数は、システム内またはネットワークを介して外部で実行されている他のプロセスと通信することができます。

Greengrass コアデバイスの制御が失われた場合に、クライアントデバイスがコアにデータを送信しないようにするには、以下を実行します。

1. Greengrass グループから Greengrass コアを削除します。
2. グループ CA 証明書をローテーションします。 AWS IoT コンソールでは、グループの設定ページで CA 証明書をローテーションできます。 AWS IoT Greengrass API では、CreateGroupCertificateAuthority アクションを使用できます。

   また、コアデバイスのハードドライブが盗難にあう危険がある場合は、フルディスク暗号化の使用をお勧めします。

AWS IoT Greengrass およびインターフェイス VPC エンドポイント (AWS PrivateLink)

インターフェイス VPC エンドポイントを作成することで、VPC と AWS IoT Greengrass コントロールプレーン間のプライベート接続を確立できます。このエンドポイントを使用して、 AWS IoT Greengrass サービス内のグループ、Lambda 関数、デプロイ、およびその他のリソースを管理できます。インターフェイスエンドポイントは、インターネットゲートウェイAWS PrivateLink、NAT デバイス、VPN 接続、または AWS Direct Connect 接続なしで AWS IoT Greengrass APIs にプライベートにアクセスできるテクノロジーである を利用しています。VPC 内のインスタンスは、 AWS IoT Greengrass APIs と通信するためにパブリック IP アドレスを必要としません。VPC と の間のトラフィック AWS IoT Greengrass は、Amazon ネットワークを離れません。

> **Note**
>
> 現在、VPC 内で完全に動作するように Greengrass コアデバイスを設定することはできません。

各インターフェイスエンドポイントは、サブネット内の 1 つ、または複数の Elastic Network Interface によって表されます。

詳細については、「Amazon VPC ユーザーガイド」の「インターフェイス VPC エンドポイント (AWS PrivateLink)」を参照してください。

トピック

- AWS IoT Greengrass VPC エンドポイントに関する考慮事項
- AWS IoT Greengrass コントロールプレーン操作用インターフェイス VPC エンドポイントの作成
- の VPC エンドポイントポリシーの作成 AWS IoT Greengrass

AWS IoT Greengrass VPC エンドポイントに関する考慮事項

のインターフェイス VPC エンドポイントを設定する前に AWS IoT Greengrass、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントのプロパティと制限」を参照してください。さらに、次の点について注意してください:

- AWS IoT Greengrass は、VPC からのすべてのコントロールプレーン API アクションの呼び出しをサポートしています。コントロールプレーンには、CreateDeployment や StartBulkDeployment などの操作が含まれます。コントロールプレーンには、データプレーンの操作である GetDeployment や Discover などの操作は含まれていません。
- の VPC エンドポイント AWS IoT Greengrass は現在、中国リージョンでは AWS サポートされていません。

AWS IoT Greengrass コントロールプレーン操作用インターフェイス VPC エンドポイントの作成

Amazon VPC コンソールまたは AWS Command Line Interface () を使用して AWS IoT Greengrass 、コントロールプレーンの VPC エンドポイントを作成できますAWS CLI。詳細につい

ては、 Amazon VPC ユーザーガイド のインターフェイスエンドポイントの作成を参照してください。

次のサービス名 AWS IoT Greengrass を使用して、 用の VPC エンドポイントを作成します。

- com.amazonaws.*region*.greengrass

エンドポイントのプライベート DNS を有効にすると、 などのリージョンのデフォルトの DNS 名 AWS IoT Greengrass を使用して に API リクエストを行うことができます`greengrass.us-east-1.amazonaws.com`。プライベート DNS はデフォルトで有効になっています。

詳細については、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントを介したサービスへのアクセス」を参照してください。

の VPC エンドポイントポリシーの作成 AWS IoT Greengrass

VPC エンドポイントに、 AWS IoT Greengrass コントロールプレーン操作へのアクセスを制御するエンドポイントポリシーをアタッチできます。このポリシーでは、以下の情報を指定します。

- アクションを実行できるプリンシパル。
- プリンシパルが実行できるアクション。
- プリンシパルがアクションを実行できるリソース。

詳細については、「Amazon VPC ユーザーガイド」の「VPC エンドポイントでサービスへのアクセスを制御する」を参照してください。

Example 例: AWS IoT Greengrass アクションの VPC エンドポイントポリシー

以下は、 のエンドポイントポリシーの例です AWS IoT Greengrass。エンドポイントにアタッチすると、このポリシーは、すべてのリソースのすべてのプリンシパルに対して、リストされた AWS IoT Greengrass アクションへのアクセスを許可します。

```
{
    "Statement": [
        {
            "Principal": "*",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "greengrass:StartBulkDeployment"
```

```
            ],
            "Resource": "*"
        }
    ]
}
```

のセキュリティのベストプラクティス AWS IoT Greengrass

このトピックには、 のセキュリティのベストプラクティスが含まれています AWS IoT Greengrass。

最小限のアクセス許可を付与する

IAM ロールの最小限のアクセス許可セットを使用して、最小特権の原則に従います。IAM ポリシーの `Action` プロパティおよび `Resource` プロパティに対する `*` ワイルドカードの使用を制限します。代わりに、可能な場合はアクションとリソースの有限セットを宣言します。最小特権およびその他のポリシーのベストプラクティスの詳細については、「the section called “ポリシーに関するベストプラクティス”」を参照してください。

最小特権のベストプラクティスは、Greengrass コアデバイスとクライアントデバイスにアタッチする AWS IoT ポリシーにも適用されます。

Lambda 関数で認証情報をハードコードしない

ユーザー定義の Lambda 関数で認証情報をハードコードしないでください。認証情報をより適切に保護するには:

- AWS サービスとやり取りするには、Greengrass グループロールで特定のアクションとリソースのアクセス許可を定義します。
- ローカルシークレット を使用して認証情報を保存します。または、関数が AWS SDK を使用している場合は、デフォルトの認証情報プロバイダーチェーンの認証情報を使用します。

機密情報を記録しない

認証情報やその他の個人を特定できる情報 (PII) のログを記録しないようにしてください。コアデバイスのローカルログへのアクセスにはルートアクセス許可が必要であり、CloudWatch Logs へのアクセスには IAM のアクセス許可が必要ですが、次の防止策を実施することをお勧めします。

- MQTT トピックパスに機密情報を使用しないでください。

- AWS IoT Core レジストリ内のデバイス (モノ) の名前、タイプ、属性に機密情報を使用しないでください。
- ユーザー定義 Lambda 関数に機密情報を記録しないでください。
- Greengrass リソースの名前と ID に機密情報を使用しないでください。
  - Connector
  - コア
  - デバイス
  - 関数
  - グループ
  - Loggers
  - リソース (ローカル、Machine Learning、シークレット)
  - サブスクリプション

ターゲットを絞ったサブスクリプションの作成

サブスクリプションは、サービス、デバイス、および Lambda 関数間でメッセージを交換する方法を定義することによって、Greengrass グループの情報フローを制御します。アプリケーションが意図したことだけを実行できるようにするには、サブスクリプションでパブリッシャーが特定のトピックにのみメッセージを送信できるようにし、サブスクライバーがその機能に必要なトピックからのみメッセージを受信するように制限する必要があります。

デバイスのクロックを同期させる

デバイスの時刻を正確に保つことが重要です。X.509 証明書には有効期限の日時があります。デバイスのクロックは、サーバー証明書が現在も有効であることを確認するために使用されます。時間の経過とともにデバイスのクロックがドリフトしたり、バッテリが放電したりする可能性があります。

詳細については、「AWS IoT Core デベロッパーガイド」の「デバイスのクロックを同期化させる」を参照してください。

Greengrass コアを使用したデバイス認証の管理

クライアントデバイスは、FreeRTOS を実行、もしくは AWS IoT Device SDK または AWS IoT Greengrass Discovery API を使用して、同じ Greengrass グループ内のコアとの接続と認証に使用される検出情報を取得できます。検出情報には、以下が含まれます。

- クライアントデバイスと同じ Greengrass グループに属する Greengrass コアの接続情報。この情報には、コアデバイスの各エンドポイントのホストアドレスとポート番号が含まれます。
- ローカル MQTT サーバー証明書の署名に使用されるグループ CA 証明書。クライアントデバイスは、グループ CA 証明書を使用して、コアによって提示される MQTT サーバー証明書を検証します。

次に、Greengrass コアとの相互認証を管理するためのクライアントデバイスのベストプラクティスを示します。これらのプラクティスは、コアデバイスが侵害された場合のリスクを軽減するのに役立ちます。

各接続のローカル MQTT サーバー証明書を検証します。

クライアントデバイスは、コアとの接続を確立するたびに、コアによって提示される MQTT サーバー証明書を検証する必要があります。この検証は、コアデバイスとクライアントデバイス間の相互認証のクライアントデバイス側です。クライアントデバイスは失敗を検出し、接続を終了できる必要があります。

検出情報をハードコードしないでください。

コアが静的 IP アドレスを使用している場合でも、クライアントデバイスは検出オペレーションに依存してコア接続情報とグループ CA 証明書を取得する必要があります。クライアントデバイスがこの検出情報をハードコードすることはできません。

検出情報を定期的に更新します。

クライアントデバイスは定期的に検出を実行して、コア接続情報とグループ CA 証明書を更新する必要があります。コアとの接続を確立する前に、クライアントデバイスでこの情報を更新することをお勧めします。検出オペレーションの時間を短くすると、潜在的なエクスポージャー時間が最小限に抑えられるため、クライアントデバイスを定期的に切断して再接続して、更新をトリガーすることをお勧めします。

Greengrass コアデバイスの制御が失われた場合に、クライアントデバイスがコアにデータを送信しないようにするには、以下を実行します。

1. Greengrass グループから Greengrass コアを削除します。
2. グループ CA 証明書をローテーションします。 AWS IoT コンソールでは、グループの設定ページで CA 証明書をローテーションできます。 AWS IoT Greengrass API では、CreateGroupCertificateAuthority アクションを使用できます。

また、コアデバイスのハードドライブが盗難にあう危険がある場合は、フルディスク暗号化の使用をお勧めします。

詳細については、「the section called “デバイス認証と認可”」を参照してください。

関連情報

- 「AWS IoT デベロッパーガイド」の「AWS IoT Coreでのセキュリティのベストプラクティス」
- AWS 公式ブログのモノのインターネットに関する産業用 IoT ソリューションの 10 のセキュリティゴールデンルール

でのログ記録とモニタリング AWS IoT Greengrass

モニタリングは、 および AWS IoT Greengrass AWS ソリューションの信頼性、可用性、パフォーマンスを維持する上で重要な部分です。マルチポイント障害が発生した場合は、その障害をより簡単にデバッグできるように、 AWS ソリューションのすべての部分からモニタリングデータを収集する必要があります。モニタリングを開始する前に AWS IoT Greengrass、以下の質問に対する回答を含むモニタリング計画を作成する必要があります。

- どのような目的でモニタリングしますか?
- どのリソースをモニタリングしますか?
- どのくらいの頻度でこれらのリソースをモニタリングしますか?
- どのモニタリングツールを使用しますか?
- 誰がモニタリングタスクを実行しますか?
- 問題が発生したときに誰が通知を受け取りますか?

モニタリングツール

AWS には、モニタリングに使用できるツールが用意されています AWS IoT Greengrass。自動的にモニタリングが行われるように、これらのツールを設定できます。手動操作を必要とするツールもあります。モニタリングタスクをできるだけ自動化することをお勧めします。

次の自動モニタリングツールを使用して、問題をモニタリング AWS IoT Greengrass および報告できます。

- Amazon CloudWatch Logs - AWS CloudTrail またはその他の出典のログファイルのモニタリング、保存、アクセスを行います。詳細については、Amazon CloudWatch ユーザーガイドのログファイルのモニタリングを参照してください。
- AWS CloudTrail ログモニタリング – アカウント間でログファイルを共有し、CloudWatch Logs に送信CloudWatch CloudTrail ログファイルをリアルタイムでモニタリングし、Java でログ処理アプリケーションを書き込み、CloudTrail による配信後にログファイルが変更されていないことを確認します。詳細については、AWS CloudTrail ユーザーガイドのCloudTrail ログファイルの操作を参照してください。
- Amazon EventBridge – EventBridge イベントルールを使用して、Greengrass グループのデプロイの状態変更または CloudTrail で記録された API コールに関する通知を受け取ることができます。

詳細については、「Amazon EventBridge ユーザーガイド」の「the section called “デプロイ通知の取得”」または「Amazon EventBridge とは」を参照してください。

- Greengrass システムヘルステレメトリー — Greengrass コアから送信されたテレメトリデータを受信するためにサブスクライブします。詳細については、「the section called “システムヘルステレメトリデータの収集”」を参照してください。
- ローカルヘルスチェック – ヘルス APIs を使用して、コアデバイス上のローカル AWS IoT Greengrass プロセスの状態のスナップショットを取得します。詳細については、「the section called “ローカルヘルスチェック API を呼び出す”」を参照してください。

関連情報

- the section called “ AWS IoT Greengrass ログでのモニタリング”
- the section called “を使用した AWS IoT Greengrass API コールのログ記録 AWS CloudTrail”
- the section called “デプロイ通知の取得”

AWS IoT Greengrass ログによるモニタリング

AWS IoT Greengrass は、クラウドサービスと AWS IoT Greengrass Core ソフトウェアで構成されます。 AWS IoT Greengrass Core ソフトウェアは、Amazon CloudWatch とコアデバイスのローカルファイルシステムにログを書き込むことができます。コア上で実行されている Lambda 関数とコネクタは CloudWatch Logs とローカルファイルシステムにログを書き込むこともできます。問題をトラブルシューティングするには、ログを使用してイベントをモニタリングします。すべての AWS IoT Greengrass ログエントリには、タイムスタンプ、ログレベル、およびイベントに関する情報が含まれます。ログ設定の変更は、グループをデプロイした後に有効になります。

ログ記録は、グループレベルで設定されます。Greengrass グループのログ記録を設定する方法を示すステップについては、the section called “のログ記録を設定する AWS IoT Greengrass” を参照してください。

CloudWatch Logs にアクセスする

CloudWatch ログ記録を設定すると、Amazon CloudWatch コンソールの [Logs] (ログ) ページにログを表示できます。ログの AWS IoT Greengrass ロググループは、次の命名規則を使用します。

```
/aws/greengrass/GreengrassSystem/greengrass-system-component-name
/aws/greengrass/Lambda/aws-region/account-id/lambda-function-name
```

各ロググループには、次の命名規則を使用するログストリームが含まれています。

```
date/account-id/greengrass-group-id/name-of-core-that-generated-log
```

以下の考慮事項は、CloudWatch Logs を使用する場合に適用されます。

- ログは、インターネットに接続できない場合、制限された再試行回数で CloudWatch Logs に送信されます。再試行回数の上限に達すると、イベントは削除されます。
- トランザクション、メモリ、その他の制限が適用されます。詳細については、「the section called "ログ記録の制限"」を参照してください。
- Greengrass グループロールは、 AWS IoT Greengrass が CloudWatch Logs に書き込むことを許可する必要があります。アクセス許可を付与するには、グループロールに次のインラインポリシーを組み込みます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws:logs:*:*:*"
            ]
        }
    ]
}
```

> **Note**
>
> 詳細なアクセス権をログリソースに付与することができます。詳細については、「Amazon CloudWatch ユーザーガイド」の「CloudWatch Logs でアイデンティティベースのポリシー (IAM ポリシー) を使用する」を参照してください。

グループロールは、ユーザーが作成して Greengrass グループにアタッチした IAM ロールです。コンソールまたは AWS IoT Greengrass API を使用して、グループロールを管理できます。

コンソールの使用

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ターゲットグループを選択します。
3. [View Settings] (設定を表示) をクリックします。[Group role] (グループロール) では、グループロールを表示、関連付けまたは関連付けの解除ができます。

   グループロールをアタッチする手順については、「グループロール」を参照してください。

CLI の使用

- グループロールを検索するには、get-associated-role コマンドを使用します。
- グループロールをアタッチするには、associate-role-to-group コマンドを使用します。
- グループロールを削除するには、disassociate-role-from-group コマンドを使用します。

これらのコマンドで使用するグループ ID を取得する方法については、「the section called “グループ ID の取得”」を参照してください。

ファイルシステムログへのアクセス

ファイルシステムのログ記録を設定すると、ログファイルは、コアデバイスの *greengrass-root*/ggc/var/log に保存されます。高レベルのディレクトリ構造を以下に示します。

```
greengrass-root/ggc/var/log
    - crash.log
    - system
        - log files for each Greengrass system component
    - user
```

```
        - region
            - account-id
                - log files generated by each user-defined Lambda function
            - aws
                - log files generated by each connector
```

> **Note**
>
> デフォルトでは、*greengrass-root* は /greengrass ディレクトリです。書き込みディレクトリが設定されている場合には、ログはこのディレクトリにあります。

以下の考慮事項は、ファイルシステムログを使用する場合に適用されます。

- ファイルシステム上の AWS IoT Greengrass ログの読み取りには、ルートアクセス許可が必要です。
- AWS IoT Greengrass は、ログデータの量が設定された制限に近い場合、サイズベースのローテーションと自動クリーンアップをサポートします。
- `crash.log` ファイルは、ファイルシステムログでのみ使用できます。このログは CloudWatch Logs に書き込まれません。
- ディスク使用量の制限が適用されます。詳細については、「the section called “ログ記録の制限”」を参照してください。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェア v1.0 のログは *greengrass-root*/var/log ディレクトリに保存されます。

デフォルトのログ記録設定

ログ記録設定が明示的に設定されていない場合、 は最初のグループのデプロイ後に次のデフォルトのログ記録設定 AWS IoT Greengrass を使用します。

AWS IoT Greengrass システムコンポーネント

- タイプ - `FileSystem`
- コンポーネント - `GreengrassSystem`

- レベル - `INFO`
- スペース - `128 KB`

ユーザー定義 Lambda 関数

- タイプ - `FileSystem`
- コンポーネント - `Lambda`
- レベル - `INFO`
- スペース - `128 KB`

> **Note**
> 最初のデプロイ前に、ユーザー定義の Lambda 関数がデプロイされていないため、システムコンポーネントだけがファイルシステムにログを書き込みます。

のログ記録を設定する AWS IoT Greengrass

AWS IoT コンソールまたは AWS IoT Greengrass APIsを使用して AWS IoT Greengrass ログ記録を設定できます。

> **Note**
> AWS IoT Greengrass が CloudWatch Logs にログを書き込むには、グループロールで必要な CloudWatch Logs アクションを許可する必要があります。

ログ記録の設定 (コンソール)

グループの [設定] ページでログ記録を設定できます。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. ログ記録を設定するグループを選択します。
3. グループの設定ページで、[Logs] (ログ) タブを選択します。
4. 次のようにログ記録の場所を選択します。

- CloudWatch ログ記録を設定するには、[CloudWatch logs configuration] (CloudWatch ログ設定) の、[Edit] (編集) を選択します。
- ファイルシステムログ記録を設定するには、[Local logs configuration] (ローカルログ設定) の、[Edit] (編集) を選択します。

1 つの場所または両方の場所のログ記録を設定できます。

5. ログの編集設定モーダルで、[Greengrass system log level] (Greengrass システムのログレベル) または [User Lambda functions log level] (ユーザーの Lambda 関数のログレベル) を選択します。1 つのコンポーネントまたは両方のコンポーネントを選択することができます。
6. ログに記録するイベントの最低レベルを選択します。このしきい値に達しないイベントは除外され、保存されません。
7. [Save] を選択します。変更は、グループをデプロイした後に有効になります。

ログ記録の設定 (API)

AWS IoT Greengrass ロガー APIs を使用して、プログラムでログ記録を設定できます。例えば、`CreateLoggerDefinition` アクションを使用して、`LoggerDefinitionVersion` ペイロードに基づくロガー定義を作成します。これは、次の構文を使用します。

```
{
  "Loggers": [
    {
      "Id": "string",
      "Type": "FileSystem|AWSCloudWatch",
      "Component": "GreengrassSystem|Lambda",
      "Level": "DEBUG|INFO|WARN|ERROR|FATAL",
      "Space": "integer"
    },
    {
      "Id": "string",
      ...
    }
  ]
}
```

`LoggerDefinitionVersion` は、次のプロパティを持つ 1 つ以上の `Logger` オブジェクトの配列です。

`Id`

ロガーの識別子。

`Type`

ログイベントのストレージメカニズム。`AWSCloudWatch` を使用する場合、ログイベントは CloudWatch Logs に送信されます。`FileSystem` を使用すると、ログイベントはローカルファイルシステムに保存されます。

有効な値: `AWSCloudWatch`、`FileSystem`

`Component`

ログイベントのソース。`GreengrassSystem` を使用すると、Greengrass システムコンポーネントのイベントがログに記録されます。`Lambda` を使用すると、ユーザー定義の Lambda 関数のイベントがログに記録されます。

有効な値: `GreengrassSystem`、`Lambda`

`Level`

ログレベルのしきい値。このしきい値に達しないログイベントは除外され、保存されません。

有効な値: `DEBUG`、`INFO` (推奨)、`WARN`、`ERROR`、`FATAL`

`Space`

ログを保存するローカルストレージの最大容量 (KB 単位)。このフィールドは、`Type` を `FileSystem` に設定した場合にのみ適用されます。

設定例

次の `LoggerDefinitionVersion` の例では、次のようなログ記録設定を指定しています。

- AWS IoT Greengrass システムコンポーネントのファイルシステム`ERROR`以上のログ記録を有効にします。
- ユーザー定義の Lambda 関数のファイルシステム `INFO` (以上) のログ記録を有効にします。
- ユーザー定義の Lambda 関数の CloudWatch `INFO` (以上) のログ記録を有効にします。

```
{
```

```
  "Name": "LoggingExample",
  "InitialVersion": {
    "Loggers": [
      {
        "Id": "1",
        "Component": "GreengrassSystem",
        "Level": "ERROR",
        "Space": 10240,
        "Type": "FileSystem"
      },
      {
        "Id": "2",
        "Component": "Lambda",
        "Level": "INFO",
        "Space": 10240,
        "Type": "FileSystem"
      },
      {
        "Id": "3",
        "Component": "Lambda",
        "Level": "INFO",
        "Type": "AWSCloudWatch"
      }
    ]
  }
}
```

ロガー定義バージョンを作成した後、そのバージョン ARN を使用して、グループをデプロイする前にグループバージョンを作成できます。

ログ記録の制限

AWS IoT Greengrass には、次のログ記録の制限があります。

1 秒あたりのトランザクション

CloudWatch へのログ記録を有効にすると、ログイベントはログコンポーネントでローカルにバッチ処理された後で CloudWatch に送信されるため、ログストリームごとに 5 リクエスト/秒よりも高いレートでログを記録できます。

「メモリ」

AWS IoT Greengrass が CloudWatch にログを送信するように設定されていて、Lambda 関数が長期間 5 MB/秒を超えると、内部処理パイプラインは最終的にいっぱいになります。理論的には Lambda 関数あたり 6 MB で限界に達します。

クロックスキュー

CloudWatch へのログ記録を有効にすると、ログ記録コンポーネントは標準の署名バージョン 4 の署名プロセスを使用して CloudWatch へのリクエストに署名します。 AWS IoT Greengrass コアデバイスのシステム時刻が同期していない時間が 15 分を超えていると、リクエストは拒否されます。

ディスク使用量

次の式を使用して、ログ記録用の最大の合計ディスク使用量を計算します。

```
greengrass-system-component-space * 8    // 7 if automatic IP detection is disabled
  + 128KB                                // the internal log for the local logging
 component
  + lambda-space * lambda-count          // different versions of a Lambda function are
 treated as one
```

コードの説明は以下のとおりです。

`greengrass-system-component-space`

AWS IoT Greengrass システムコンポーネントログのローカルストレージの最大量。

`lambda-space`

Lambda 関数のログ用の最大のローカルストレージ容量。

`lambda-count`

Lambda 関数のデプロイ数。

ログの損失

AWS IoT Greengrass コアデバイスが CloudWatch にのみログ記録するように設定されていて、インターネット接続がない場合、現在メモリにあるログを取得する方法はありません。

Lambda 関数が (デプロイ時などに) 終了すると、数秒分のログは CloudWatch に書き込まれません。

CloudTrail ログ

AWS IoT Greengrass は、ユーザー AWS CloudTrail、ロール、または のサービスによって実行されたアクションを記録する AWS サービスで実行されます AWS IoT Greengrass。詳細については、「the section called “を使用した AWS IoT Greengrass API コールのログ記録 AWS CloudTrail”」を参照してください。

を使用した AWS IoT Greengrass API コールのログ記録 AWS CloudTrail

AWS IoT Greengrass は、ユーザー AWS CloudTrail、ロール、または のサービスによって実行されたアクションを記録する AWS サービスである と統合されています AWS IoT Greengrass。CloudTrail は、 AWS IoT Greengrass のすべての API コールをイベントとしてキャプチャします。キャプチャされた呼び出しには、 AWS IoT Greengrass コンソールからの呼び出しと AWS IoT Greengrass API オペレーションへのコード呼び出しが含まれます。証跡を作成する場合は、 イベントなど、Amazon S3 バケットへの CloudTrail イベントの継続的な配信を有効にすることができます AWS IoT Greengrass。証跡を設定しない場合でも、CloudTrail コンソールの [イベント履歴] で最新のイベントを表示できます。CloudTrail によって収集された情報を使用して、リクエストの実行元の IP アドレス AWS IoT Greengrass、リクエストの実行者、リクエストの実行日時などの詳細を確認できます。

CloudTrail の詳細については、「AWS CloudTrail ユーザーガイド」を参照してください。

AWS IoT Greengrass CloudTrail の情報

CloudTrail は、アカウントの作成 AWS アカウント 時に で有効になります。でアクティビティが発生すると AWS IoT Greengrass、そのアクティビティは CloudTrail イベントとイベント履歴の他の AWS サービスイベントに記録されます。で最近のイベントを表示、検索、ダウンロードできます AWS アカウント。詳細については、「Viewing events with CloudTrail event history」(CloudTrail イベント履歴でのイベントの表示) を参照してください。

のイベントなど AWS アカウント、 のイベントの継続的な記録については AWS IoT Greengrass、証跡を作成します。追跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成すると、証跡はすべての AWS リージョンに適用されます。証跡は、 AWS パーティション内のすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集されたイベントデータをさらに分析して処理するように、他の AWS サービスを設定できます。詳細については、次を参照してください:

- 追跡を作成するための概要
- 「CloudTrail がサポートされているサービスと統合」
- 「CloudTrail の Amazon SNS 通知の設定」
- 「複数のリージョンから CloudTrail ログファイルを受け取る」および「複数のアカウントから CloudTrail ログファイルを受け取る」

すべての AWS IoT Greengrass アクションは CloudTrail によってログに記録され、 AWS IoT Greengrass API リファレンスに記載されています。例えば、`AssociateServiceRoleToAccount`、`GetGroupVersion`、`GetConnectivityInfo`、`CreateFun` の各アクションを呼び出すと、CloudTrail ログファイルにエントリが生成されます。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます。

- リクエストが root または AWS Identity and Access Management (IAM) ユーザー認証情報を使用して行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、「CloudTrail userIdentity エレメント」を参照してください。

AWS IoT Greengrass ログファイルエントリについて

「トレイル」は、指定した Amazon S3 バケットにイベントをログファイルとして配信するように設定できます。CloudTrail のログファイルは、単一か複数のログエントリを含みます。イベントは任意ソースからの単一リクエストを表し、リクエストされたアクション、アクションの日時、リクエストパラメータなどの情報を含みます。CloudTrail ログファイルは、パブリック API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

以下の例は、`AssociateServiceRoleToAccount` アクションを示す CloudTrail ログエントリです。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
```

```
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major"
    },
    "eventTime": "2018-10-17T17:04:02Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "AssociateServiceRoleToAccount",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "apimanager.amazonaws.com",
    "errorCode": "BadRequestException",
    "requestParameters": null,
    "responseElements": {
        "Message": "That role ARN is invalid."
    },
    "requestID": "a5990ec6-d22e-11e8-8ae5-c7d2eEXAMPLE",
    "eventID": "b9070ce2-0238-451a-a9db-2dbf1EXAMPLE",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

以下の例は、`GetGroupVersion` アクションを示す CloudTrail ログエントリです。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-10-17T18:14:57Z"
            }
        },
        "invokedBy": "apimanager.amazonaws.com"
    },
```

```
    "eventTime": "2018-10-17T18:15:11Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "GetGroupVersion",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "apimanager.amazonaws.com",
    "requestParameters": {
        "GroupVersionId": "6c477753-dbf2-4cb8-acc3-5ba4eEXAMPLE",
        "GroupId": "90fcf6df-413c-4515-93a8-00056EXAMPLE"
    },
    "responseElements": null,
    "requestID": "95dcffce-d238-11e8-9240-a3993EXAMPLE",
    "eventID": "8a608034-82ed-431b-b5e0-87fbdEXAMPLE",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

以下の例は、GetConnectivityInfo アクションを示す CloudTrail ログエントリです。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major"
    },
    "eventTime": "2018-10-17T17:02:12Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "GetConnectivityInfo",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "apimanager.amazonaws.com",
    "requestParameters": {
        "ThingName": "us-east-1_CIS_1539795000000_"
    },
    "responseElements": null,
    "requestID": "63e3ebe3-d22e-11e8-9ddd-5baf3EXAMPLE",
    "eventID": "db2260d1-a8cc-4a65-b92a-13f65EXAMPLE",
    "readOnly": true,
```

```
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

以下の例は、CreateFunctionDefinition アクションを示す CloudTrail ログエントリです。

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major"
    },
    "eventTime": "2018-10-17T18:01:11Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "CreateFunctionDefinition",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "apimanager.amazonaws.com",
    "requestParameters": {
        "InitialVersion": "***"
    },
    "responseElements": {
        "CreationTimestamp": "2018-10-17T18:01:11.449Z",
        "LatestVersion": "dae06a61-c32c-41e9-b983-ee5cfEXAMPLE",
        "LatestVersionArn": "arn:aws:greengrass:us-east-1:123456789012:/greengrass/
definition/functions/7a94847d-d4d2-406c-9796-a3529EXAMPLE/versions/dae06a61-c32c-41e9-
b983-ee5cfEXAMPLE",
        "LastUpdatedTimestamp": "2018-10-17T18:01:11.449Z",
        "Id": "7a94847d-d4d2-406c-9796-a3529EXAMPLE",
        "Arn": "arn:aws:greengrass:us-east-1:123456789012:/greengrass/definition/
functions/7a94847d-d4d2-406c-9796-a3529EXAMPLE"
    },
    "requestID": "a17d4b96-d236-11e8-a74e-3db27EXAMPLE",
    "eventID": "bdbf6677-a47a-4c78-b227-c5f64EXAMPLE",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

関連情報

- AWS CloudTrail ユーザーガイドの「 とは AWS CloudTrail」
- Amazon EventBridge ユーザーガイドのCloudTrail を使用して AWS API コールでトリガーする EventBridge ルールの作成EventBridge」
- AWS IoT Greengrass API リファレンス

AWS IoT Greengrass コアデバイスからのシステムヘルステレメトリデータの収集

システムヘルステレメトリデータは、Greengrass コアデバイスにおける重要な操作のパフォーマンス監視に役立つ診断データです。Greengrass コアのテレメトリエージェントは、お客様とのやり取りを必要とせずに、ローカルテレメトリデータを収集して、Amazon EventBridge に発行します。コアデバイスは、ベストエフォートベースで EventBridge にテレメトリデータを発行します。例えば、コアデバイスはオフライン中にテレメトリデータの配信に失敗することがあります。

> **Note**
>
> Amazon EventBridge は、アプリケーションを Greengrass Core デバイスやデプロイ通知などのさまざまなソースのデータに接続するために使用できるイベントバスサービスです。詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge とは」を参照してください。

プロジェクトとアプリケーションを作成して、エッジデバイスからのテレメトリデータを取得、分析、変換、レポートできます。プロセスエンジニアといった特定分野のエキスパートは、これらのアプリケーションを使用して、フリートのヘルスに関する洞察を得られます。

Greengrass エッジコンポーネントが適切に機能するように、 は開発および品質向上の目的でデータ AWS IoT Greengrass を使用します。また、この機能は、新しいエッジ機能や拡張エッジ機能を通知するのに役立ちます。 はテレメトリデータを最大 7 日間 AWS IoT Greengrass のみ保持します。

この機能は AWS IoT Greengrass Core ソフトウェア v1.11.0 で使用でき、既存のコアを含むすべての Greengrass コアでデフォルトで有効になっています。 AWS IoT Greengrass Core ソフトウェア v1.11.0 以降にアップグレードするとすぐに、自動的にデータの受信が開始されます。

発行されたテレメトリデータへのアクセスやそうしたデータの管理を行う方法については、「the section called “テレメトリデータを受信するためのサブスクライブ”」を参照してください。

テレメトリエージェントは、次のシステムメトリックを収集して発行します。

テレメトリメトリック

名前	説明	ソース
`SystemMemUsage`	オペレーティングシステムを含む、Greengrass コアデバイスのすべてのアプリケーションで現在使用されているメモリの量。	システム
`CpuUsage`	オペレーティングシステムを含む Greengrass コアデバイスのすべてのアプリケーションで現在使用されている CPU の量。	システム
`TotalNumberOfFDs`	Greengrass コアデバイスのオペレーティングシステムによって保存されているファイルディスクリプタの数。1 つのファイルディスクリプタは、1 つのオープンファイルを一意に識別します。	システム
`LambdaOutOfMemory`	Lambda 関数がメモリを使い切った実行の回数。	システム
`DroppedMessageCount`	AWS IoT Coreに送信されドロップされたメッセージの数。	`GGCloudSpooler` システムコンポーネント
`LambdaTimeout`	ユーザー定義 Lambda 関数の実行がタイムアウトした回数。	ユーザー定義の Lambda 関数 AWS クラウド、およびシステム

名前	説明	ソース
LambdaUngracefullyKilled	ユーザー定義 Lambda 関数の実行が完了しなかった回数。	ユーザー定義の Lambda 関数 AWS クラウド、およびシステム
LambdaError	ユーザー定義 Lambda 関数の実行によってエラーログが書き込まれた回数。	ユーザー定義の Lambda 関数 AWS クラウド、およびシステム
BytesAppended	ストリームマネージャーに追加されたデータのバイト数。	GGStreamManager システムコンポーネント
BytesUploadedToIoTAnalytics	ストリームマネージャーがチャネルにエクスポートするデータのバイト数 AWS IoT Analytics。	GGStreamManager システムコンポーネント
BytesUploadedToKinesis	ストリームマネージャーが Amazon Kinesis Data Streams のストリームにエクスポートするデータのバイト数。	GGStreamManager システムコンポーネント
BytesUploadedToIoTSiteWise	ストリームマネージャーがアセットプロパティにエクスポートするデータのバイト数 AWS IoT SiteWise。	GGStreamManager システムコンポーネント
BytesUploadedToS3ExportTaskExecutor	ストリームマネージャーが Amazon S3 のオブジェクトにエクスポートするデータのバイト数。	GGStreamManager システムコンポーネント
BytesUploadedToHTTP	ストリームマネージャーが HTTP にエクスポートするデータのバイト数。	GGStreamManager システムコンポーネント

テレメトリ設定の構成

Greengrass テレメトリでは、次の設定が使用されます。

- テレメトリエージェントは、1 時間ごとにテレメトリデータを集約します。
- テレメトリエージェントは 24 時間ごとにテレメトリメッセージを発行します。

> **Note**
> こうした設定は変更できません。

Greengrass コアデバイスのテレメトリ機能は、有効または無効にできます。 AWS IoT Greengrass は、シャドウを使用してテレメトリ設定を管理します。変更は、コアが に接続されるとすぐに有効になります AWS IoT Core。

テレメトリエージェントは、サービス品質 (QoS) レベルが 0 の MQTT プロトコルを使用してデータを発行します。つまり、配信の確認や発行の再試行は行いません。テレメトリメッセージは、MQTT 接続を、 AWS IoT Coreを送信先とする他のサブスクリプションメッセージと共有します。

データリンクのコストを除き、コアから へのデータ転送 AWS IoT Core は無料です。これは、エージェントが AWS 予約済みトピックに発行するためです。ただし、ユースケースによっては、データを受信または処理するときにコストが発生する場合もあります。

要件

テレメトリ設定を構成するときには、次の要件が適用されます。

- AWS IoT Greengrass Core ソフトウェア v1.11.0 以降を使用する必要があります。

  > **Note**
  > 以前のバージョンを実行していて、テレメトリを使用しない場合は、何もする必要はありません。

- IAM アクセス権限を付与する必要があります。これにより、コア (モノ) シャドウを更新して、テレメトリ設定の更新前に設定 API を呼び出します。

  次の IAM ポリシー例を使用すると、特定のコアのシャドウとランタイム設定を管理できます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowManageShadow",
            "Effect": "Allow",
            "Action": [
                "iot:GetThingShadow",
                "iot:UpdateThingShadow",
                "iot:DeleteThingShadow",
                "iot:DescribeThing"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/core-name-*"
            ]
        },
        {
            "Sid": "AllowManageRuntimeConfig",
            "Effect": "Allow",
            "Action": [
                "greengrass:GetCoreRuntimeConfiguration",
                "greengrass:UpdateCoreRuntimeConfiguration"
            ],
            "Resource": [
                "arn:aws:iot:region:account-id:thing/core-name"
            ]
        }
    ]
}
```

リソースにきめ細かいアクセス権限または条件付きアクセス権限を付与できます (例えば、ワイルドカード * 命名スキームを使用)。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの追加と削除」を参照してください。

テレメトリ設定の構成 (コンソール)

以下は、 AWS IoT Greengrass コンソールで Greengrass コアのテレメトリ設定を更新する方法を示しています。

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. [Greengrass groups] (グリーングラスのグループ) で、対象グループを選択します。
3. グループ設定ページの [Overview] (概要) セクションで、[Greengrass core] (Greengrass コア) を選択します。
4. コアの設定ページで、 [Telemetry] (テレメトリ) タブを選択します。
5. [System health telemetry] (システム ヘルス テレメトリ) セクションで、[Configure] (設定) を選択します。
6. [Configure telemetry] (テレメトリの設定) で、[Telemetry] (テレメトリ) を選択して、テレメトリのステータスを有効または無効にします。

> **Important**
>
> デフォルトでは、テレメトリ機能は AWS IoT Greengrass Core ソフトウェア v1.11.0 以降で有効になっています。

変更は実行時に有効になります。グループをデプロイする必要はありません。

テレメトリ設定の構成 (CLI)

AWS IoT Greengrass API では、 `TelemetryConfiguration` オブジェクトは Greengrass コアのテレメトリ設定を表します。このオブジェクトは、コアに関連する `RuntimeConfiguration` オブジェクトの一部です。 AWS IoT Greengrass API、 AWS CLI、または AWS SDK を使用して Greengrass テレメトリを管理できます。このセクションの例では AWS CLIを使用します。

テレメトリ設定を確認するには

次のコマンドでは、Greengrass コアのテレメトリ設定を取得します。

- *`core-thing-name`* をターゲットコアの名前に置き換えます。

  モノの名前を取得するには、get-core-definition-version コマンドを使用します。このコマンドは、モノの名前を含むモノの ARN を返します。

```
aws greengrass get-thing-runtime-configuration --thing-name core-thing-name
```

このコマンドでは、`GetCoreRuntimeConfigurationResponse` オブジェクトが JSON 応答で返ります。例:

```
{
    "RuntimeConfiguration": {
        "TelemetryConfiguration": {
            "ConfigurationSyncStatus": "OutOfSync",
            "Telemetry": "On"
        }
    }
}
```

テレメトリ設定を構成するには

次のコマンドでは、Greengrass コアのテレメトリ設定を更新します。

- *core-thing-name* をターゲットコアの名前に置き換えます。

  モノの名前を取得するには、get-core-definition-version コマンドを使用します。このコマンドは、モノの名前を含むモノの ARN を返します。

JSON expanded

```
aws greengrass update-thing-runtime-configuration --thing-name core-thing-name --
telemetry-configuration  '{
    "RuntimeConfiguration": {
        "TelemetryConfiguration": {
            "ConfigurationSyncStatus": "InSync",
            "Telemetry": "Off"
        }
    }
}
```

JSON single-line

```
aws greengrass update-thing-runtime-configuration --thing-name core-thing-name --
telemetry-configuration "{\"TelemetryConfiguration\":{\"ConfigurationSyncStatus
\":\"InSync\",\"Telemetry\":\"Off\"}}"
```

`ConfigurationSyncStatus` が `InSync` の場合、テレメトリ設定の変更が適用されています。変更は実行時に有効になります。グループをデプロイする必要はありません。

TelemetryConfiguration オブジェクト

`TelemetryConfiguration` オブジェクトには以下のプロパティがあります。

`ConfigurationSyncStatus`

テレメトリ設定が同期しているかどうかを確認します。このプロパティは変更できません。

タイプ: 文字列

有効な値: `InSync` または `OutOfSync`

`Telemetry`

テレメトリをオンまたはオフにします。デフォルト: `On`。

タイプ: 文字列

有効な値: `On` または `Off`

テレメトリデータを受信するためのサブスクライブ

Amazon EventBridge では、ルールを作成して、Greengrass コアデバイスから発行されたテレメトリデータの処理方法を定義できます。データを受信した EventBridge によって、ルールで定義したターゲットアクションが呼び出されます。例えば、通知の送信、イベント情報の保存、是正措置の実践、他のイベントの呼び出しなどを行うイベントルールを作成できます。

テレメトリイベント

テレメトリデータを含むデプロイ状態変更のイベントでは、次の形式を使用します。

```
{
    "version": "0",
    "id": "f70f943b-9ae2-e7a5-fec4-4c22178a3e6a",
    "detail-type": "Greengrass Telemetry Data",
    "source": "aws.greengrass",
    "account": "123456789012",
    "time": "2020-07-28T20:45:53Z",
    "region": "us-west-1",
    "resources": [],
    "detail": {
        "ThingName": "CoolThing",
        "Schema": "2020-06-30",
        "ADP": [
            {
                "TS": 123231546,
```

```
                "NS": "StreamManager",
                "M": [
                    {
                        "N": "BytesAppended|BytesUploadedToKinesis",
                        "Sum": 11,
                        "U": "Bytes"
                    }
                ]
            },
            {
                "TS": 123231546,
                "NS": "StreamManager",
                "M": [
                    {
                        "N": "BytesAppended|BytesUploadedToS3ExportTaskExecutor",
                        "Sum": 11,
                        "U": "Bytes"
                    }
                ]
            },
            {
                "TS": 123231546,
                "NS": "StreamManager",
                "M": [
                    {
                        "N": "BytesAppended|BytesUploadedToHTTP",
                        "Sum": 11,
                        "U": "Bytes"
                    }
                ]
            },
            {
                "TS": 123231546,
                "NS": "StreamManager",
                "M": [
                    {
                        "N": "BytesAppended|BytesUploadedToIoTAnalytics",
                        "Sum": 11,
                        "U": "Bytes"
                    }
                ]
            },
            {
                "TS": 123231546,
```

```
                "NS": "StreamManager",
                "M": [
                    {
                        "N": "BytesAppended|BytesUploadedToIoTSiteWise",
                        "Sum": 11,
                        "U": "Bytes"
                    }
                ]
            },
            {
                "TS": 123231546,
                "NS": "arn:aws:lambda:us-west-1:123456789012:function:my-function",
                "M": [
                    {
                        "N": "LambdaTimeout",
                        "Sum": 15,
                        "U": "Count"
                    }
                ]
            },
            {
                "TS": 123231546,
                "NS": "CloudSpooler",
                "M": [
                    {
                        "N": "DroppedMessageCount",
                        "Sum": 15,
                        "U": "Count"
                    }
                ]
            },
            {
                "TS": 1593727692,
                "NS": "SystemMetrics",
                "M": [
                    {
                        "N": "SystemMemUsage",
                        "Sum": 11.23,
                        "U": "Megabytes"
                    },
                    {
                        "N": "CpuUsage",
                        "Sum": 35.63,
                        "U": "Percent"
```

```
                    },
                    {
                        "N": "TotalNumberOfFDs",
                        "Sum": 416,
                        "U": "Count"
                    }
                ]
            },
            {
                "TS": 1593727692,
                "NS": "arn:aws:lambda:us-west-1:123456789012:function:my-function",
                "M": [
                    {
                        "N": "LambdaOutOfMemory",
                        "Sum": 12,
                        "U": "Count"
                    },
                    {
                        "N": "LambdaUngracefullyKilled",
                        "Sum": 100,
                        "U": "Count"
                    },
                    {
                        "N": "LambdaError",
                        "Sum": 7,
                        "U": "Count"
                    }
                ]
            }
        ]
    }
}
```

ADP 配列には、次のプロパティを持つ集約データポイントのリストが含まれています。

TS

必須。データが集約された時刻のタイムスタンプ。

NS

必須。システムの名前空間。

M

必須。メトリクスのリスト。メトリクスには次のプロパティが含まれています。

N

メトリクスの名前。

Sum

集計されたメトリック値。テレメトリエージェントは前の合計に新しい値を加算するため、合計の値は絶えず増加します。タイムスタンプを使用すると、特定の集計値を求められます。例えば、最近集計された値を求めるには、最新のタイムスタンプ付き値から前のタイムスタンプ値を減算します。

U

メトリクス値の単位。

ThingName

必須。ターゲットとするモノデバイスの名前。

EventBridge ルールを作成するための前提条件

の EventBridge ルールを作成する前に AWS IoT Greengrass、以下を実行する必要があります。

- EventBridge のイベント、ルール、ターゲットに精通しておいてください。
- EventBridge ルールによって呼び出される [targets] (ターゲット) を作成して設定します。ルールは、Amazon Kinesis ストリーム、 AWS Lambda 関数、Amazon SNS トピック、Amazon SQS キューなど、さまざまなタイプのターゲットを呼び出すことができます。

  EventBridge ルールと関連ターゲットは、Greengrass リソースを作成した AWS リージョン 内になければなりません。詳細については、「AWS 全般のリファレンス」の「サービスのエンドポイントとクォータ」を参照してください。

詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge とは」および「Amazon EventBridge の開始方法」を参照してください。

テレメトリデータを取得するイベントルールを作成する (コンソール)

次の手順を使用して、Greengrass コアによって発行されたテレメトリデータを受信する EventBridge ルール AWS Management Console を作成します。これにより、ウェブサーバー、E

メールアドレス、その他のトピック受信者がイベントに応答できるようになります。詳細については、「Amazon EventBridge ユーザーガイド」の「 AWS リソースからイベントでトリガーする EventBridge ルールの作成」を参照してください。 EventBridge

1. Amazon EventBridge コンソールを開き、[Create rule] (ルールの作成) を選択します。
2. [Name and description (名前と説明)] に、ルールの名前と説明を入力します。
3. [Event bus] (イベントバス) を選択し、選択したイベントバスでルールを有効にします。
4. [Rule type] (ルールタイプ)、[Rule with an event pattern] (イベントパターンを持つルール) の順に選択します。
5. [Next (次へ)] を選択します。
6. [Event source] (イベントソース) で、[AWS events or EventBridge partner events] (イベントまたは EventBridge パートナーイベント) を選択してください。
7. [サンプルイベント] では、[AWS イベント]、[Greengrass テレメトリデータ] の順に選択します。
8. [Event pattern] (イベントパターン) では、次のように選択します。
   a. [イベントソース] で、[AWS のサービス] を選択してください。
   b. [AWS Service] (サービス) では、 [Greengrass] を選択します。
   c. [Event type] (イベントタイプ) では、[Greengrass Telemetry Data] (Greengrass テレメトリデータ) を選択します。
9. [Next (次へ)] を選択します。
10. [Target 1] (ターゲット 1) で、[AWS services] (サービス) を選択します。
11. [Select a target] (ターゲットの選択) で、[SQS queue] (SQS キュー) を選択します。
12. [Queue] (キュー) で、関数を選択します。

テレメトリデータを取得するイベントルールを作成する (CLI)

次の手順を使用して、Greengrass コアによって発行されたテレメトリデータを受信する EventBridge ルール AWS CLI を作成します。これにより、ウェブサーバー、E メールアドレス、その他のトピック受信者がイベントに応答できるようになります。

1. ルールを作成します。
   - *thing-name* をコアのモノ名に置き換えます。

モノの名前を取得するには、get-core-definition-version コマンドを使用します。このコマンドは、モノの名前を含むモノの ARN を返します。

```
aws events put-rule \
  --name TestRule \
  --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName\":
 [\"thing-name\"]}}"
```

パターンで省略されたプロパティは無視されます。

2. トピックをルールターゲットとして追加します。次の例では、Amazon SQS を使用していますが、他のターゲットタイプも設定できます。

   - *queue-arn* を Amazon SQS キューの ARN に置き換えます。

```
aws events put-targets \
  --rule TestRule \
  --targets "Id"="1","Arn"="queue-arn"
```

> **Note**
>
> Amazon EventBridge にターゲットキューの呼び出しを許可するには、トピックにリソースベースのポリシーを追加する必要があります。詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon SQS のアクセス許可」を参照してください。

詳細については、「Amazon EventBridge ユーザーガイド」の「EventBridge のイベントとイベントパターン」を参照してください。

AWS IoT Greengrass テレメトリのトラブルシューティング

AWS IoT Greengrass テレメトリの設定に関する問題のトラブルシューティングには、次の情報を使用します。

次のエラーが発生する。The response contains "ConfigurationStatus": "OutOfSync" after you run the get-thing-runtime-configuration command

解決方法

- AWS IoT Device Shadow サービスは、ランタイム設定の更新を処理し、Greengrass コアデバイスに更新を配信するまでに時間がかかります。しばらく待った後に、テレメトリ設定が同期しているかどうかを確認してください。
- コアデバイスがオンラインになっていることを確認します。
- AWS IoT Coreの Amazon CloudWatch Logs を有効にして、シャドウを監視します。
- AWS IoT メトリクスを使用して、モノを監視します。

ローカルヘルスチェック API を呼び出す

AWS IoT Greengrass には、 によって開始されたローカルワーカープロセスの現在の状態のスナップショットを提供するローカル HTTP API が含まれています AWS IoT Greengrass。このスナップショットには、ユーザー定義の Lambda 関数とシステム Lambda 関数が含まれます。System Lambda 関数は AWS IoT Greengrass Core ソフトウェアの一部です。これらは、コアデバイス上でローカルワーカープロセスとして実行され、メッセージルーティング、ローカルシャドウ同期、および自動 IP アドレス検出などのオペレーションを管理します。

ヘルスチェック API は、次のリクエストをサポートしています。

- すべてのワーカーのヘルス情報を取得するために `GET` リクエストを送信する。
- 指定されたワーカーのヘルス情報を取得するために `POST` リクエストを送信する。

リクエストはデバイス上でローカルに送信され、インターネット接続は必要ありません。

すべてのワーカーのヘルス情報を取得する

実行中のすべてのワーカーに関するヘルス情報を取得するために `GET` リクエストを送信します。

- *`port`* を IPC のポート番号に置き換えます。

```
GET http://localhost:port/2016-11-01/health/workers
```

`port`

IPC のポート番号。

値は 1024 ~ 65535 の間で変更できます。デフォルト値は 8000 です。

このポート番号を変更するには、`config.json` ファイル内の `ggDaemonPort` プロパティを更新します。詳細については、「AWS IoT Greengrass コア設定ファイル」を参照してください。

リクエストの例

以下の例の `curl` リクエストは、すべてのワーカーのヘルス情報を取得します。

```
curl http://localhost:8000/2016-11-01/health/workers
```

JSON レスポンス

このリクエストは、ワーカーのヘルス情報オブジェクトの配列を返します。

レスポンスの例

次のレスポンス例では、 によって開始されたすべてのワーカープロセスのヘルス情報オブジェクトを一覧表示します AWS IoT Greengrass。

```
[
    {
      "FuncArn": "arn:aws:lambda:::function:GGShadowService:1",
      "WorkerId" : "65515053-2f70-43dc-7cc0-1712bEXAMPLE",
      "ProcessId": "1234",
      "WorkerState": "Waiting"
    },
    {
      "FuncArn": "arn:aws:lambda:::function:GGSecretManager:1",
      "WorkerId": "a9916cc2-1b4d-4f0e-4b12-b1872EXAMPLE",
      "ProcessId": "9798",
      "WorkerState": "Waiting"
    },
    {
      "FuncArn": "arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function:3",
      "WorkerId": "2e6f785e-66a5-42c9-67df-42073EXAMPLE",
      "ProcessId": "11837",
```

```
      "WorkerState": "Waiting"
    },
    ...
]
```

指定されたワーカーに関するヘルス情報を取得する

指定されたワーカーに関するヘルス情報を取得するために、POST リクエストを送信します。*port* を IPC のポート番号に置き換えます。デフォルトは 8,000 です。

```
POST http://localhost:port/2016-11-01/health/workers
```

リクエストの例

以下の例の curl リクエストは、指定されたワーカーのヘルス情報を取得します。

```
curl --data "@body.json" http://localhost:8000/2016-11-01/health/workers
```

body.json リクエストボディの例を以下に示します。

```
{
    "FuncArns": [
        "arn:aws:lambda:::function:GGShadowService:1",
        "arn:aws:lambda:us-west-2:123456789012:function:my-lambda-function:3"
    ]
}
```

リクエストボディには、FuncArns 配列が含まれます。

FuncArns

ターゲットワーカーを表す Lambda 関数の Amazon リソースネーム (ARN) のリスト。

- ユーザー定義 Lambda 関数の場合は、現在デプロイされているバージョンの ARN を指定します。エイリアス ARN を使用して Lambda 関数をグループに追加した場合、GET リクエストを使用してすべてのワーカーを取得してから、クエリする ARN を選択できます。
- システム Lambda 関数の場合は、対応する Lambda 関数の ARN を指定します。詳細については、「the section called “システム Lambda 関数”」を参照してください。

型: 文字列の配列

最小長: 1

最大長: コアデバイスで AWS IoT Greengrass によって開始されたワーカーの総数。

JSON レスポンス

このリクエストは `Workers` 配列と `InvalidArns` 配列を返します。

`Workers`

指定されたワーカーのヘルス情報オブジェクトのリスト。

タイプ : ヘルス情報オブジェクトの配列

`InvalidArns`

ワーカーが関連付けられていない関数 ARN を含む、無効な関数 ARN のリスト。

型: 文字列の配列

レスポンスの例

次のレスポンスの例は、指定されたワーカーのヘルス情報オブジェクトの一覧を示しています。

```
{
    "Workers": [
        {
            "FuncArn": "arn:aws:lambda:::function:GGShadowService:1",
            "WorkerId" : "65515053-2f70-43dc-7cc0-1712bEXAMPLE",
            "ProcessId": "1234",
            "WorkerState": "Waiting"
        },
        {
            "FuncArn": "arn:aws:lambda:us-west-2:123456789012:function:my-lambda-
function:3",
            "WorkerId": "2e6f785e-66a5-42c9-67df-42073ESAMPLE",
            "ProcessId": "11837",
            "WorkerState": "Waiting"
        }
    ],
    "InvalidArns" : [
        "some-malformed-arn",
        "arn:aws:lambda:us-west-2:123456789012:function:some-unknown-function:1"
```

```
    ]
}
```

このリクエストは次のエラーを返します。

400 Invalid request

リクエストボディの形式が正しくありません。この問題を解決するには、以下の形式を使用してリクエストを再送信します。

```
{"FuncArns":["function-1-arn","function-2-arn"]}
```

400 Request exceeds max number of workers

`FuncArns` 配列で指定された ARN の数がワーカーの数を超えています。

ワーカーのヘルス情報

ヘルス情報オブジェクトには、以下のプロパティが含まれています。

`FuncArn`

ワーカーを表すシステム Lambda 関数の ARN。

型: `string`

`WorkerId`

ワーカーの ID。このプロパティはデバッグに役立ちます。`runtime.log` ファイルと Lambda 関数のログにはワーカー ID が含まれているため、このプロパティは複数のインスタンスをスピンアップするオンデマンド Lambda 関数をデバッグする場合に特に便利です。

型: `string`

`ProcessId`

ワーカープロセスのプロセス ID (PID)。

型: `int`

`WorkerState`

ワーカーの状態。

型: `string`

有効なワーカーの状態には以下のものがあります。

`Working`

メッセージの処理

`Waiting`

メッセージの待機 デーモンまたはスタンドアロンプロセスとして実行される 存続期間の長い Lambda 関数に適用されます。

`Starting`

スピンアップして、開始します。

`FailedInitialization`

初期化に失敗しました。

`Terminated`

Greengrass デーモンによって停止されました。

`NotStarted`

開始に失敗し、別の開始が試行されました。

`Initialized`

正常に初期化されました。

システム Lambda 関数

次のシステム Lambda 関数のヘルス情報をリクエストできます。

`GGCloudSpooler`

をソースまたはターゲット AWS IoT Core とする MQTT メッセージのキューを管理します。

ARN: `arn:aws:lambda:::function:GGCloudSpooler:1`

`GGConnManager`

Greengrass コアとクライアントデバイス間で MQTT メッセージをルーティングします。

ARN: `arn:aws:lambda:::function:GGConnManager`

`GGDeviceCertificateManager`

コアの IP AWS IoT エンドポイントへの変更をシャドウにリッスンし、相互認証のために GGConnManager で使用されるサーバー側の証明書を生成します。

ARN: `arn:aws:lambda:::function:GGDeviceCertificateManager`

`GGIPDetector`

Greengrass グループのデバイスが Greengrass コアデバイスを検出できるようにするための IP アドレス自動検出を管理します。IP アドレスを手動で指定する場合、このサービスは適用されません。

ARN: `arn:aws:lambda:::function:GGIPDetector:1`

`GGSecretManager`

ローカルシークレットの安全な保管と、ユーザー定義 Lambda およびコネクタによるアクセスを管理します。

ARN: `arn:aws:lambda:::function:GGSecretManager:1`

`GGShadowService`

クライアントデバイスのローカルシャドウを管理します。

ARN: `arn:aws:lambda:::function:GGShadowService`

`GGShadowSyncManager`

デバイスの `syncShadow`プロパティが に設定されている場合、ローカルシャドウを AWS クラウド コアデバイスおよびクライアントデバイスの と同期します`true`。

ARN: `arn:aws:lambda:::function:GGShadowSyncManager`

`GGStreamManager`

データストリームをローカルで処理し、 への自動エクスポートを実行します AWS クラウド。

ARN: `arn:aws:lambda:::function:GGStreamManager:1`

`GGTES`

ローカルコードがサービスへのアクセスに使用する Greengrass グループロールで定義された IAM 認証情報を取得するローカルトークン交換 AWS サービス。

ARN: `arn:aws:lambda:::function:GGTES`

AWS IoT Greengrass リソースのタグ付け

タグは、 AWS IoT Greengrass グループの整理と管理に役立ちます。タグを使用して、グループ、一括デプロイ、およびグループに追加されたコア、デバイス、その他のリソースにメタデータを割り当てることができます。タグを IAM ポリシーで使用して、Greengrass リソースへの条件付きアクセスを定義することもできます。

> **Note**
>
> 現在、Greengrass リソースタグは、 AWS IoT 請求グループまたはコスト配分レポートではサポートされていません。

タグの基本

タグを使用すると、リソースを目的 AWS IoT Greengrass 、所有者、環境などで分類できます。同じタイプのリソースが多い場合に、アタッチしたタグに基づいて特定のリソースをすばやく識別できます。タグはキーとオプションの値で構成されており、どちらもお客様側が定義します。各リソースタイプに対して一連のタグキーを設定することをお勧めします。一貫性のある一連のタグキーを使用することで、リソースの管理が容易になります。例えば、グループに対して一連のタグを定義でき、それによりコアデバイスの施設の場所の追跡に役立ちます。詳細については、「AWS タグ付け戦略」を参照してください。

AWS IoT コンソールでのタグ付けのサポート

AWS IoT コンソールで Greengrass `Group`リソースのタグを作成、表示、管理できます。タグを作成する前に、タグの制限に注意してください。詳細については、「Amazon Web Services 全般のリファレンス」の「タグの命名規則と使用規則」を参照してください。

グループの作成時にタグを割り当てるには

グループを作成するときに、グループにタグを割り当てることができます。タグ付け入力フィールドを表示するには、[Tags] (タグ) セクションで[Add new tag] (新しいタグの追加) を選択します。

グループ設定ページからタグを表示および管理するには

[View settings] (設定の表示) を選択すると、グループ設定ページからタグを表示および管理することができます。グループの [Tags] (タグ) セクションで、 [Manage tags] (タグの管理) を選択して、グループタグを追加、編集、または削除します。

AWS IoT Greengrass API でのタグ付けのサポート

AWS IoT Greengrass API を使用して、タグ付けをサポートする AWS IoT Greengrass リソースのタグを作成、一覧表示、管理できます。タグを作成する前に、タグの制限に注意してください。詳細については、「Amazon Web Services 全般のリファレンス」の「タグの命名規則と使用規則」を参照してください。

- リソースの作成中にタグを追加するには、リソースの `tags` プロパティで定義します。
- リソースの作成後にタグを追加する、またはタグ値を更新するには、`TagResource` アクションを使用します。
- リソースからタグを削除するには、`UntagResource` アクションを使用します。
- リソースに関連付けられたタグを取得するには、`ListTagsForResource` アクションを使用するか、リソースを取得してその `tags` プロパティを調べます。

次の表に、 AWS IoT Greengrass API でタグ付けできるリソースと、対応する `Create` および `Get`アクションを示します。

リソース	作成	Get
`Group`	`CreateGroup`	`GetGroup`
`ConnectorDefinition`	`CreateConnectorDefinition`	`GetConnectorDefinition`
`CoreDefinition`	`CreateCoreDefinition`	`GetCoreDefinition`
`DeviceDefinition`	`CreateDeviceDefinition`	`GetDeviceDefinition`
`FunctionDefinition`	`CreateFunctionDefinition`	`GetFunctionDefinition`

リソース	作成	Get
LoggerDefinition	CreateLoggerDefinition	GetLoggerDefinition
ResourceDefinition	CreateResourceDefinition	GetResourceDefinition
SubscriptionDefinition	CreateSubscriptionDefinition	GetSubscriptionDefinition
BulkDeployment	StartBulkDeployment	GetBulkDeploymentStatus

リストで次のアクションを使用して、タグ付けをサポートしているリソースのタグを管理します。

- TagResource。リソースにタグを追加します。タグのキーと値のペアの値を変更するためにも使用されます。
- ListTagsForResource。リソースのタグをリスト表示します。
- UntagResource。リソースからタグを削除します。

リソースでのタグの追加または削除は随時行うことができます。タグキーの値を変更するには、同じキーと新しい値を定義するリソースにタグを追加します。古い値は新しい値を上書きします。値を空の文字列に設定することはできますが、値を null に設定することはできません。

リソースを削除すると、リソースに関連付けられているすべてのタグも削除されます。

> **Note**
>
> リソースタグを、 AWS IoT モノに割り当てることができる属性と混同しないでください。Greengrass コアは AWS IoT モノですが、このトピックで説明されているリソースタグは、コアモノではなく CoreDefinitionにアタッチされます。

IAM ポリシーでのタグの使用

IAM ポリシーでは、リソースタグを使用して、ユーザーのアクセスとアクセス許可を制御できます。例えば、ポリシーにより、ユーザーは特定のタグがあるリソースのみを作成できます。ポリシーにより、特定のタグを持つリソースをユーザーが作成または変更できないよう制限することもできます。作成時にリソースにタグを付けることができるため (作成時のタグ付けと呼ばれます)、後でカスタムタグ付けスクリプトを実行する必要はありません。タグを使用して新しい環境が起動されるときは、対応する IAM アクセス権限が自動的に適用されます。

次の条件コンテキストキーと値は、ポリシーの `Condition` 要素 (`Condition` ブロックとも呼ばれます) で使用できます。

`greengrass:ResourceTag/`*`tag-key`*`: `*`tag-value`*

特定のタグを持つリソースに対してユーザーアクションを許可または拒否します。

`aws:RequestTag/`*`tag-key`*`: `*`tag-value`*

タグ付け可能なリソースでタグを作成または変更する API リクエストを作成する場合に、特定のタグが使用されている (または、使用されていない) ことを要求します。

`aws:TagKeys: [`*`tag-key`*`, ...]`

タグ付け可能なリソースを作成または変更する API リクエストを作成する場合に、特定のタグキーのセットが使用されている (または、使用されていない) ことを要求します。

条件コンテキストキーと値は、タグ付け可能なリソースで動作する AWS IoT Greengrass アクションでのみ使用できます。これらのアクションは、必須パラメータとしてリソースを受け取ります。例えば、`GetGroupVersion` で条件付きアクセスを設定できます。`AssociateServiceRoleToAccount` で条件付きアクセスを設定することはできません。これは、タグ付け可能なリソース (グループ、コア定義、デバイス定義など) が、リクエストで参照されるためです。

詳細については、「IAM ユーザーガイド」の「タグを使用したアクセスの制御」および「IAM JSON ポリシーリファレンス」を参照してください。JSON ポリシーリファレンスには、IAM の JSON ポリシーの要素、変数、および評価ロジックの詳細な構文、説明、および例が含まれています。

IAM ポリシーの例

次のポリシー例では、ベータユーザーをベータリソースでのアクションのみに制限する、タグベースのアクセス権限を適用します。

- 最初のステートメントによって、IAM ユーザーは env=beta タグのみが付いたリソースを操作できます。
- 2 番目のステートメントにより、IAM ユーザーは env=beta タグをリソースから削除できなくなります。これにより、ユーザーは自分のアクセスを削除することがありません。

> **Note**
>
> タグを使用してリソースへのアクセスを制御する場合は、同じリソースに対するタグの追加または削除をユーザーに許可するアクセス権限も管理する必要があります。そうしないと、場合によっては、ユーザーはそのタグを変更することで、制限を回避してリソースにアクセスできる可能性があります。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "greengrass:*",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "greengrass:ResourceTag/env": "beta"
                }
            }
        },
        {
            "Effect": "Deny",
            "Action": "greengrass:UntagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/env": "beta"
                }
            }
        }
    ]
}
```

作成時にユーザーにタグ付けを許可するには、最初に適切なアクセス権限を付与する必要があります。次のポリシー例では、`greengrass:TagResource` および`greengrass:CreateGroup` アクションに `"aws:RequestTag/env": "beta"` 条件が含まれています。これにより、ユーザーは、env=beta が指定されたグループにタグを付ける場合のみ、グループを作成できます。その結果、実質的にユーザーに新しいグループへのタグ付けを強制します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "greengrass:TagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": "beta"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "greengrass:CreateGroup",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": "beta"
                }
            }
        }
    ]
}
```

次のスニペットは、タグ値を 1 つのリストとして指定して、1 つのタグキーに対して複数のタグ値を指定することもできます。

```
"StringEquals" : {
    "greengrass:ResourceTag/env" : ["dev", "test"]
}
```

関連情報

- でのAWS リソースのタグ付け Amazon Web Services 全般のリファレンス

AWS CloudFormation のサポート AWS IoT Greengrass

AWS CloudFormation は、 AWS リソースの作成、管理、レプリケートに役立つサービスです。AWS CloudFormation テンプレートを使用して、デプロイする AWS IoT Greengrass グループとクライアントデバイス、サブスクリプション、およびその他のコンポーネントを定義できます。例については、the section called “サンプルテンプレート”を参照してください。

テンプレートから生成するリソースとインフラストラクチャリソースは、スタックと呼ばれます。すべてのリソースを 1 つのテンプレートに定義するか、他のスタックのリソースを参照できます。AWS CloudFormation テンプレートと機能の詳細については、 AWS CloudFormation ユーザーガイドの「 とは AWS CloudFormation」を参照してください。

リソースの作成

AWS CloudFormation テンプレートは、 AWS リソースのプロパティと関係を記述する JSON または YAML ドキュメントです。次の AWS IoT Greengrass リソースがサポートされています。

- グループ
- コア
- クライアントデバイス (デバイス)
- Lambda 関数
- Connector
- リソース (ローカル、機械学習、シークレット)
- サブスクリプション
- ロガー (ログ記録の設定)

AWS CloudFormation テンプレートでは、Greengrass リソースの構造と構文は AWS IoT Greengrass API に基づいています。例えば、サンプルテンプレートは、最上位の `DeviceDefinition` を、個々のクライアントデバイスを含む `DeviceDefinitionVersion` に関連付けます。詳細については、「the section called “グループオブジェクトモデルの概要”」を参照してください。

AWS CloudFormation ユーザーガイドのAWS IoT Greengrass リソースタイプのリファレンスでは、 で管理できる Greengrass リソースについて説明します AWS CloudFormation。 AWS CloudFormation テンプレートを使用して Greengrass リソースを作成する場合は、 からのみ管理す

ることをお勧めします AWS CloudFormation。例えば、デバイスを追加、変更、または削除する場合は (AWS IoT Greengrass API または AWS IoT コンソールを使用する代わりに)、テンプレートを更新する必要があります。これにより、ロールバックや AWS CloudFormation のその他の変更管理機能を使用できます。を使用してリソースとスタックを作成および管理 AWS CloudFormation する方法の詳細については、 AWS CloudFormation ユーザーガイドの「スタックの使用」を参照してください。

AWS CloudFormation テンプレートで AWS IoT Greengrass リソースを作成してデプロイする方法を示すチュートリアルについては、 AWS 公式ブログの「モノのインターネット」の「 による AWS IoT Greengrass セットアップの自動化 AWS CloudFormation」を参照してください。

リソースのデプロイ

グループバージョンを含む AWS CloudFormation スタックを作成したら、 AWS CLI または AWS IoT コンソールを使用してデプロイできます。

> **Note**
>
> グループをデプロイするには、 に関連付けられた Greengrass サービスロールが必要です AWS アカウント。サービスロールは AWS IoT Greengrass 、 が AWS Lambda およびその他の AWS サービスのリソースにアクセスできるようにします。このロールは、現在の AWS リージョンに Greengrass グループをデプロイ済みである場合に存在します。詳細については、「the section called “Greengrass サービスロール”」を参照してください。

グループをデプロイするには (AWS CLI)

- create-deployment コマンドを実行します。

```
aws greengrass create-deployment --group-id GroupId --group-version-
id GroupVersionId --deployment-type NewDeployment
```

> **Note**
>
> サンプルテンプレート内の `CommandToDeployGroup` ステートメントは、スタックを作成する際の、グループおよびグループバージョン ID を使用したコマンドの出力例を示しています。

グループをデプロイするには (コンソール)

1. AWS IoT コンソールナビゲーションペインの「管理」で Greengrass デバイスを展開し、「グループ (V1)」を選択します。
2. グループを選択します。
3. グループ設定ページで、[Deploy] (デプロイ) を選択します。

サンプルテンプレート

次のサンプルテンプレートでは、コア、クライアントデバイス、関数、ロガー、サブスクリプション、および 2 つのリソースを含む Greengrass グループを作成します。これを行うには、テンプレートは AWS IoT Greengrass API のオブジェクトモデルに従います。例えば、グループに追加するクライアントデバイスは `DeviceDefinitionVersion` リソースに含まれます。これは `DeviceDefinition` リソースに関連付けられています。グループにデバイスを追加するため、グループバージョンは `DeviceDefinitionVersion` の ARN を参照します。

テンプレートには、コアとデバイスの証明書 ARNs とソース Lambda 関数 (AWS Lambda リソース) のバージョン ARN を指定できるパラメータが含まれています。ID、ARN、および Greengrass リソースの作成に必要なその他の属性を参照するため、`Ref` および `GetAtt` 組み込み関数を使用します。

テンプレートでは、Greengrass グループに追加されるコアデバイスとクライアントデバイスを表す 2 つの AWS IoT デバイス (モノ) も定義されています。

Greengrass リソースを使用してスタックを作成したら、 AWS CLI または AWS IoT コンソールを使用してグループをデプロイできます。

> **Note**
>
> この例の `CommandToDeployGroup` ステートメントは、グループをデプロイするために使用できる完全な create-deployment CLI コマンドを出力する方法を示しています。

JSON

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "AWS IoT Greengrass example template that creates a group version
 with a core, device, function, logger, subscription, and resources.",
```

```
    "Parameters": {
        "CoreCertificateArn": {
            "Type": "String"
        },
        "DeviceCertificateArn": {
            "Type": "String"
        },
        "LambdaVersionArn": {
            "Type": "String"
        }
    },
    "Resources": {
        "TestCore1": {
            "Type": "AWS::IoT::Thing",
            "Properties": {
                "ThingName": "TestCore1"
            }
        },
        "TestCoreDefinition": {
            "Type": "AWS::Greengrass::CoreDefinition",
            "Properties": {
                "Name": "DemoTestCoreDefinition"
            }
        },
        "TestCoreDefinitionVersion": {
            "Type": "AWS::Greengrass::CoreDefinitionVersion",
            "Properties": {
                "CoreDefinitionId": {
                    "Ref": "TestCoreDefinition"
                },
                "Cores": [
                    {
                        "Id": "TestCore1",
                        "CertificateArn": {
                            "Ref": "CoreCertificateArn"
                        },
                        "SyncShadow": "false",
                        "ThingArn": {
                            "Fn::Join": [
                                ":",
                                [
                                    "arn:aws:iot",
                                    {
                                        "Ref": "AWS::Region"
```

```
                                    },
                                    {
                                        "Ref": "AWS::AccountId"
                                    },
                                    "thing/TestCore1"
                                ]
                            ]
                        }
                    }
                ]
            }
        },
        "TestClientDevice1": {
            "Type": "AWS::IoT::Thing",
            "Properties": {
                "ThingName": "TestClientDevice1"
            }
        },
        "TestDeviceDefinition": {
            "Type": "AWS::Greengrass::DeviceDefinition",
            "Properties": {
                "Name": "DemoTestDeviceDefinition"
            }
        },
        "TestDeviceDefinitionVersion": {
            "Type": "AWS::Greengrass::DeviceDefinitionVersion",
            "Properties": {
                "DeviceDefinitionId": {
                    "Fn::GetAtt": [
                        "TestDeviceDefinition",
                        "Id"
                    ]
                },
                "Devices": [
                    {
                        "Id": "TestClientDevice1",
                        "CertificateArn": {
                            "Ref": "DeviceCertificateArn"
                        },
                        "SyncShadow": "true",
                        "ThingArn": {
                            "Fn::Join": [
                                ":",
                                [
```

```
                                    "arn:aws:iot",
                                    {
                                        "Ref": "AWS::Region"
                                    },
                                    {
                                        "Ref": "AWS::AccountId"
                                    },
                                    "thing/TestClientDevice1"
                                ]
                            ]
                        }
                    }
                ]
            }
        },
        "TestFunctionDefinition": {
            "Type": "AWS::Greengrass::FunctionDefinition",
            "Properties": {
                "Name": "DemoTestFunctionDefinition"
            }
        },
        "TestFunctionDefinitionVersion": {
            "Type": "AWS::Greengrass::FunctionDefinitionVersion",
            "Properties": {
                "FunctionDefinitionId": {
                    "Fn::GetAtt": [
                        "TestFunctionDefinition",
                        "Id"
                    ]
                },
                "DefaultConfig": {
                    "Execution": {
                        "IsolationMode": "GreengrassContainer"
                    }
                },
                "Functions": [
                    {
                        "Id": "TestLambda1",
                        "FunctionArn": {
                            "Ref": "LambdaVersionArn"
                        },
                        "FunctionConfiguration": {
                            "Pinned": "true",
                            "Executable": "run.exe",
```

```
                            "ExecArgs": "argument1",
                            "MemorySize": "512",
                            "Timeout": "2000",
                            "EncodingType": "binary",
                            "Environment": {
                                "Variables": {
                                    "variable1": "value1"
                                },
                                "ResourceAccessPolicies": [
                                    {
                                        "ResourceId": "ResourceId1",
                                        "Permission": "ro"
                                    },
                                    {
                                        "ResourceId": "ResourceId2",
                                        "Permission": "rw"
                                    }
                                ],
                                "AccessSysfs": "false",
                                "Execution": {
                                    "IsolationMode": "GreengrassContainer",
                                    "RunAs": {
                                        "Uid": "1",
                                        "Gid": "10"
                                    }
                                }
                            }
                        }
                    }
                ]
            }
        },
        "TestLoggerDefinition": {
            "Type": "AWS::Greengrass::LoggerDefinition",
            "Properties": {
                "Name": "DemoTestLoggerDefinition"
            }
        },
        "TestLoggerDefinitionVersion": {
            "Type": "AWS::Greengrass::LoggerDefinitionVersion",
            "Properties": {
                "LoggerDefinitionId": {
                    "Ref": "TestLoggerDefinition"
                },
```

```
                "Loggers": [
                    {
                        "Id": "TestLogger1",
                        "Type": "AWSCloudWatch",
                        "Component": "GreengrassSystem",
                        "Level": "INFO"
                    }
                ]
            }
        },
        "TestResourceDefinition": {
            "Type": "AWS::Greengrass::ResourceDefinition",
            "Properties": {
                "Name": "DemoTestResourceDefinition"
            }
        },
        "TestResourceDefinitionVersion": {
            "Type": "AWS::Greengrass::ResourceDefinitionVersion",
            "Properties": {
                "ResourceDefinitionId": {
                    "Ref": "TestResourceDefinition"
                },
                "Resources": [
                    {
                        "Id": "ResourceId1",
                        "Name": "LocalDeviceResource",
                        "ResourceDataContainer": {
                            "LocalDeviceResourceData": {
                                "SourcePath": "/dev/TestSourcePath1",
                                "GroupOwnerSetting": {
                                    "AutoAddGroupOwner": "false",
                                    "GroupOwner": "TestOwner"
                                }
                            }
                        }
                    },
                    {
                        "Id": "ResourceId2",
                        "Name": "LocalVolumeResourceData",
                        "ResourceDataContainer": {
                            "LocalVolumeResourceData": {
                                "SourcePath": "/dev/TestSourcePath2",
                                "DestinationPath": "/volumes/TestDestinationPath2",
                                "GroupOwnerSetting": {
```

```
                                    "AutoAddGroupOwner": "false",
                                    "GroupOwner": "TestOwner"
                                }
                            }
                        }
                    }
                ]
            }
        },
        "TestSubscriptionDefinition": {
            "Type": "AWS::Greengrass::SubscriptionDefinition",
            "Properties": {
                "Name": "DemoTestSubscriptionDefinition"
            }
        },
        "TestSubscriptionDefinitionVersion": {
            "Type": "AWS::Greengrass::SubscriptionDefinitionVersion",
            "Properties": {
                "SubscriptionDefinitionId": {
                    "Ref": "TestSubscriptionDefinition"
                },
                "Subscriptions": [
                    {
                        "Id": "TestSubscription1",
                        "Source": {
                            "Fn::Join": [
                                ":",
                                [
                                    "arn:aws:iot",
                                    {
                                        "Ref": "AWS::Region"
                                    },
                                    {
                                        "Ref": "AWS::AccountId"
                                    },
                                    "thing/TestClientDevice1"
                                ]
                            ]
                        },
                        "Subject": "TestSubjectUpdated",
                        "Target": {
                            "Ref": "LambdaVersionArn"
                        }
                    }
```

```
                ]
            }
        },
        "TestGroup": {
            "Type": "AWS::Greengrass::Group",
            "Properties": {
                "Name": "DemoTestGroupNewName",
                "RoleArn": {
                    "Fn::Join": [
                        ":",
                        [
                            "arn:aws:iam:",
                            {
                                "Ref": "AWS::AccountId"
                            },
                            "role/TestUser"
                        ]
                    ]
                },
                "InitialVersion": {
                    "CoreDefinitionVersionArn": {
                        "Ref": "TestCoreDefinitionVersion"
                    },
                    "DeviceDefinitionVersionArn": {
                        "Ref": "TestDeviceDefinitionVersion"
                    },
                    "FunctionDefinitionVersionArn": {
                        "Ref": "TestFunctionDefinitionVersion"
                    },
                    "SubscriptionDefinitionVersionArn": {
                        "Ref": "TestSubscriptionDefinitionVersion"
                    },
                    "LoggerDefinitionVersionArn": {
                        "Ref": "TestLoggerDefinitionVersion"
                    },
                    "ResourceDefinitionVersionArn": {
                        "Ref": "TestResourceDefinitionVersion"
                    }
                },
                "Tags": {
                    "KeyName0": "value",
                    "KeyName1": "value",
                    "KeyName2": "value"
                }
```

```
            }
        }
    },
    "Outputs": {
        "CommandToDeployGroup": {
            "Value": {
                "Fn::Join": [
                    " ",
                    [
                        "groupVersion=$(cut -d'/' -f6 <<<",
                        {
                            "Fn::GetAtt": [
                                "TestGroup",
                                "LatestVersionArn"
                            ]
                        },
                        ");",
                        "aws --region",
                        {
                            "Ref": "AWS::Region"
                        },
                        "greengrass create-deployment --group-id",
                        {
                            "Ref": "TestGroup"
                        },
                        "--deployment-type NewDeployment --group-version-id",
                        "$groupVersion"
                    ]
                ]
            }
        }
    }
}
```

YAML

```
AWSTemplateFormatVersion: 2010-09-09
Description: >-
  AWS IoT Greengrass example template that creates a group version with a core,
  device, function, logger, subscription, and resources.
Parameters:
  CoreCertificateArn:
    Type: String
```

```
  DeviceCertificateArn:
    Type: String
  LambdaVersionArn:
    Type: String
Resources:
  TestCore1:
    Type: 'AWS::IoT::Thing'
    Properties:
      ThingName: TestCore1
  TestCoreDefinition:
    Type: 'AWS::Greengrass::CoreDefinition'
    Properties:
      Name: DemoTestCoreDefinition
  TestCoreDefinitionVersion:
    Type: 'AWS::Greengrass::CoreDefinitionVersion'
    Properties:
      CoreDefinitionId: !Ref TestCoreDefinition
      Cores:
        - Id: TestCore1
          CertificateArn: !Ref CoreCertificateArn
          SyncShadow: 'false'
          ThingArn: !Join
            - ':'
            - - 'arn:aws:iot'
              - !Ref 'AWS::Region'
              - !Ref 'AWS::AccountId'
              - thing/TestCore1
  TestClientDevice1:
    Type: 'AWS::IoT::Thing'
    Properties:
      ThingName: TestClientDevice1
  TestDeviceDefinition:
    Type: 'AWS::Greengrass::DeviceDefinition'
    Properties:
      Name: DemoTestDeviceDefinition
  TestDeviceDefinitionVersion:
    Type: 'AWS::Greengrass::DeviceDefinitionVersion'
    Properties:
      DeviceDefinitionId: !GetAtt
        - TestDeviceDefinition
        - Id
      Devices:
        - Id: TestClientDevice1
          CertificateArn: !Ref DeviceCertificateArn
```

```
          SyncShadow: 'true'
          ThingArn: !Join
            - ':'
            - - 'arn:aws:iot'
              - !Ref 'AWS::Region'
              - !Ref 'AWS::AccountId'
              - thing/TestClientDevice1
  TestFunctionDefinition:
    Type: 'AWS::Greengrass::FunctionDefinition'
    Properties:
      Name: DemoTestFunctionDefinition
  TestFunctionDefinitionVersion:
    Type: 'AWS::Greengrass::FunctionDefinitionVersion'
    Properties:
      FunctionDefinitionId: !GetAtt
        - TestFunctionDefinition
        - Id
      DefaultConfig:
        Execution:
          IsolationMode: GreengrassContainer
      Functions:
        - Id: TestLambda1
          FunctionArn: !Ref LambdaVersionArn
          FunctionConfiguration:
            Pinned: 'true'
            Executable: run.exe
            ExecArgs: argument1
            MemorySize: '512'
            Timeout: '2000'
            EncodingType: binary
            Environment:
              Variables:
                variable1: value1
              ResourceAccessPolicies:
                - ResourceId: ResourceId1
                  Permission: ro
                - ResourceId: ResourceId2
                  Permission: rw
              AccessSysfs: 'false'
              Execution:
                IsolationMode: GreengrassContainer
                RunAs:
                  Uid: '1'
                  Gid: '10'
```

```
  TestLoggerDefinition:
    Type: 'AWS::Greengrass::LoggerDefinition'
    Properties:
      Name: DemoTestLoggerDefinition
  TestLoggerDefinitionVersion:
    Type: 'AWS::Greengrass::LoggerDefinitionVersion'
    Properties:
      LoggerDefinitionId: !Ref TestLoggerDefinition
      Loggers:
        - Id: TestLogger1
          Type: AWSCloudWatch
          Component: GreengrassSystem
          Level: INFO
  TestResourceDefinition:
    Type: 'AWS::Greengrass::ResourceDefinition'
    Properties:
      Name: DemoTestResourceDefinition
  TestResourceDefinitionVersion:
    Type: 'AWS::Greengrass::ResourceDefinitionVersion'
    Properties:
      ResourceDefinitionId: !Ref TestResourceDefinition
      Resources:
        - Id: ResourceId1
          Name: LocalDeviceResource
          ResourceDataContainer:
            LocalDeviceResourceData:
              SourcePath: /dev/TestSourcePath1
              GroupOwnerSetting:
                AutoAddGroupOwner: 'false'
                GroupOwner: TestOwner
        - Id: ResourceId2
          Name: LocalVolumeResourceData
          ResourceDataContainer:
            LocalVolumeResourceData:
              SourcePath: /dev/TestSourcePath2
              DestinationPath: /volumes/TestDestinationPath2
              GroupOwnerSetting:
                AutoAddGroupOwner: 'false'
                GroupOwner: TestOwner
  TestSubscriptionDefinition:
    Type: 'AWS::Greengrass::SubscriptionDefinition'
    Properties:
      Name: DemoTestSubscriptionDefinition
  TestSubscriptionDefinitionVersion:
```

```
    Type: 'AWS::Greengrass::SubscriptionDefinitionVersion'
    Properties:
      SubscriptionDefinitionId: !Ref TestSubscriptionDefinition
      Subscriptions:
        - Id: TestSubscription1
          Source: !Join
            - ':'
            - - 'arn:aws:iot'
              - !Ref 'AWS::Region'
              - !Ref 'AWS::AccountId'
              - thing/TestClientDevice1
          Subject: TestSubjectUpdated
          Target: !Ref LambdaVersionArn
  TestGroup:
    Type: 'AWS::Greengrass::Group'
    Properties:
      Name: DemoTestGroupNewName
      RoleArn: !Join
        - ':'
        - - 'arn:aws:iam:'
          - !Ref 'AWS::AccountId'
          - role/TestUser
      InitialVersion:
        CoreDefinitionVersionArn: !Ref TestCoreDefinitionVersion
        DeviceDefinitionVersionArn: !Ref TestDeviceDefinitionVersion
        FunctionDefinitionVersionArn: !Ref TestFunctionDefinitionVersion
        SubscriptionDefinitionVersionArn: !Ref TestSubscriptionDefinitionVersion
        LoggerDefinitionVersionArn: !Ref TestLoggerDefinitionVersion
        ResourceDefinitionVersionArn: !Ref TestResourceDefinitionVersion
      Tags:
        KeyName0: value
        KeyName1: value
        KeyName2: value
Outputs:
  CommandToDeployGroup:
    Value: !Join
      - ' '
      - - groupVersion=$(cut -d'/' -f6 <<<
        - !GetAtt
          - TestGroup
          - LatestVersionArn
        - );
        - aws --region
        - !Ref 'AWS::Region'
```

```
        - greengrass create-deployment --group-id
        - !Ref TestGroup
        - '--deployment-type NewDeployment --group-version-id'
        - $groupVersion
```

サポートされている AWS リージョン

現在、 AWS IoT Greengrass リソースを作成および管理できるのは、次の AWS リージョンのみです。

- 米国東部(オハイオ)
- 米国東部 (バージニア北部)
- 米国西部 (オレゴン)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (ソウル)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- アジアパシフィック (東京)
- 中国 (北京)
- 欧州 (フランクフルト)
- 欧州 (アイルランド)
- 欧州 (ロンドン)
- AWS GovCloud (米国西部)

AWS IoT Device Tester for AWS IoT Greengrass V1 の使用

AWS IoT Device Tester (IDT) は、IoT デバイスを検証できるダウンロード可能なテストフレームワークです。 AWS IoT Greengrass Version 1 がメンテナンスモードに移行されたため、IDT for は署名付き認定レポートを生成 AWS IoT Greengrass V1 しなくなりました。Device Qualification Program を通じて AWS Partner Device Catalog にリスト AWS IoT Greengrass V1 する新しいデバイスを認定できなくなります。 AWSただし、引き続き IDT for AWS IoT Greengrass V1 を使用して Greengrass V1 デバイスをテストできます。IDT for AWS IoT Greengrass V2 を使用して、Greengrass デバイスを認定し、AWS Partner Device Catalog に含めることをお勧めします。

IDT for は、テスト対象のデバイスに接続されたホストコンピュータ (Windows、macOS、または Linux) で AWS IoT Greengrass 実行されます。また、テストを実行して結果を集計します。また、テストプロセスを管理するためのコマンドラインインターフェイスも用意されています。

AWS IoT Greengrass 認定スイート

IDT for AWS IoT Greengrass を使用して、 AWS IoT Greengrass Core ソフトウェアがハードウェア上で実行され、 と通信できることを確認します AWS クラウド。また、 を使用してend-to-endのテストも実行します AWS IoT Core。例えば、デバイスで MQTT メッセージを送受信して正しく処理できることを確認します。

AWS IoT Device Tester for は、テストスイートとテストグループの概念を使用してテストを AWS IoT Greengrass 整理します。

- テストスイートは、デバイスが AWS IoT Greengrassの特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、Greengrass グループデプロイや MQTT メッセージングなど、特定の機能に関連する個々のテストのセットです。

詳細については、「IDT を使用して AWS IoT Greengrass 認定スイートを実行する」を参照してください。

カスタムテストスイート

IDT v4.0.0 以降、IDT for は標準化された設定設定と結果形式をテストスイート環境と AWS IoT Greengrass 組み合わせ、デバイスとデバイスソフトウェア用のカスタムテストスイートを開発できます。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

テスト作成者がカスタムテストスイートをどのように設定するかによって、カスタムテストスイートの実行に必要な設定が変わってきます。詳細については、「IDT を使用して独自のテストスイートを開発および実行する」を参照してください。

AWS IoT Device Tester for AWS IoT Greengrass V1 でサポートされているバージョン

AWS IoT Greengrass Version 1 がメンテナンスモードに移行されたため、IDT for は署名付き認定レポートを生成 AWS IoT Greengrass V1 しなくなりました。IDT for AWS IoT Greengrass V2 を使用することをお勧めします。

IDT for AWS IoT Greengrass V2 の詳細については、 AWS IoT Greengrass V2 デベロッパーガイドの「 での AWS IoT Device Tester の使用 AWS IoT Greengrass V2」を参照してください。

> **Note**
>
> IDT for が AWS IoT Greengrass 使用している のバージョンと互換性がない場合 AWS IoT Greengrass 、テストランを開始すると通知を受け取ります。

ソフトウェアをダウンロードすると、AWS IoT Device Tester ライセンス契約に同意したと見なされます。

for でサポートされていない IDT バージョン AWS IoT Greengrass

このトピックでは、IDT for のサポートされていないバージョンを一覧表示します AWS IoT Greengrass。サポートされていないバージョンのバグ修正や更新プログラムは受けられません。詳細については、「the section called “の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1”」を参照してください。

AWS IoT Greengrass バージョン v1.11.6、v1.10.5 の IDT v4.4.1

リリースノート:

- AWS IoT Greengrass コアソフトウェア v1.11.6 および v1.10.5 を実行しているデバイスを検証および認定できます。
- 軽微なバグ修正が含まれています。

テストスイートのバージョン:

`GGQ_1.3.1`

- リリース日: 2021 年 12 月 20 日

AWS IoT Greengrass バージョン v1.11.4、v1.10.4 の IDT v4.1.0

リリースノート:

- AWS IoT Greengrass コアソフトウェア v1.11.4 および v1.10.4 を実行しているデバイスを検証して認定できます。
- テスト実行時に表示されるログで冗長タグが使用される問題を修正しました。

テストスイートのバージョン:

`GGQ_1.3.0`

- リリース日: 2021 年 6 月 23 日
- Lambda、IAM、および への API コールの再試行を追加 AWS STS し、スロットリングまたはサーバーの問題の処理を改善しました。
- Python 3.8 に ML や Docker のテストケースへのサポートを追加します。

AWS IoT Greengrass バージョン v1.11.1、v1.11.0、v1.10.3 の IDT v4.0.2

リリースノート:

- IDT でハードウェアセキュリティ統合 (HSI) のエラーを認識できない原因となった問題を修正しました。

- AWS IoT Device Tester for を使用してカスタムテストスイートを開発および実行できます AWS IoT Greengrass。詳細については、「IDT を使用して独自のテストスイートを開発および実行する」を参照してください。
- macOS および Windows 用のコード署名付き IDT アプリケーションを提供します。macOS でセキュリティ警告メッセージが表示される場合は、IDT のセキュリティ例外を許可する必要が出てくる場合があります。詳細については、「macOS でのセキュリティ例外」を参照してください。

> **Note**
>
> AWS IoT Greengrass は、コアソフトウェアのバージョン 1.11.1 AWS IoT Greengrass の Dockerfile または Docker イメージを提供しません。デバイスの Docker 認定をテストするには、以前のバージョンの AWS IoT Greengrass Core ソフトウェアを使用します。

AWS IoT Greengrass バージョン v1.11.0、v1.10.1、v1.10.0 の IDT v3.2.0

リリースノート:

- IDT がデフォルトで実行するテストは認定に必要なものだけです。その他の機能を認定するには、`device.json` ファイルの内容を変更します。
- `device.json` にポート番号が追加され、SSH 接続用の設定に対応しました。
- Docker はストリームマネージャーとコンテナ化を伴わない機械学習 (ML) のみをサポートしています。コンテナ、Docker、ハードウェアセキュリティ統合 (HSI) は Docker デバイスでは使用できません。
- `device-ml.json` と `device-hsm.json` を `device.json` にまとめました。

IDT v3.1.3 AWS IoT Greengrass バージョン: v1.10.x、v1.9.x、v1.8.x

リリースノート:

- v1.10.x および AWS IoT Greengrass v1.9.x の ML 機能認定のサポートが追加されました。クラウド内に保存されているトレーニング済みモデルを使用し、デバイスが ML 推論をローカルで実行できることを IDT で検証できるようになりました。
- `run-suite` コマンドに `--stop-on-first-failure` オプションが追加されました。このオプションを使用すると、最初に失敗が発生した時点で実行を停止するように IDT を設定でき

ます。このオプションは、テストグループレベルのデバッグ段階で使用することをお勧めします。

- テスト対象のデバイスが正しいシステム時間を使用していることを確認するために、MQTT テストにクロックドリフトチェックが追加されました。使用するシステム時間は、許容時間範囲内に収まっている必要があります。
- `run-suite` コマンドに `--update-idt` オプションが追加されました。このオプションを使用すると、IDT を更新するプロンプトの応答を設定できます。
- `run-suite` コマンドに `--update-managed-policy` オプションが追加されました。このオプションを使用すると、マネージドポリシーを更新するプロンプトの応答を設定できます。
- IDT テストスイートの各種バージョンの自動更新に関するバグ修正が追加されました。この修正によって IDT は、お使いの AWS IoT Greengrass バージョンで利用できる最新のテストスイートを実行できるようになります。

の IDT v3.0.1 AWS IoT Greengrass

リリースノート:

- v1.10.1 AWS IoT Greengrass のサポートを追加しました。
- IDT テストスイートのバージョンの自動更新。IDT は、お使いの AWS IoT Greengrass バージョンで利用可能な最新のテストスイートをダウンロードできます。この機能を使用すると、次の操作を実行できます。
  - テストスイートは、*`major.minor.patch`* 形式を使用してバージョン管理されます。最初のテストスイートのバージョンは `GGQ_1.0.0` です。
  - コマンドラインインターフェイスで新しいテストスイートをインタラクティブにダウンロードしたり、IDT の起動時に `upgrade-test-suite` フラグを設定したりできます。

  詳細については、「the section called “テストスイートのバージョン”」を参照してください。
- `list-supported-products` が追加されました。このコマンドを使用して、インストールされている IDT のバージョンでサポートされている AWS IoT Greengrass およびテストスイートのバージョンを一覧表示できます。
- `list-test-cases` が追加されました。このコマンドを使用して、テストグループで使用できるテストケースを一覧表示できます。
- `run-suite` コマンドに `test-id` オプションが追加されました。このオプションを使用して、テストグループ内の個々のテストケースを実行できます。

IDT v2.3.0 for AWS IoT Greengrass v1.10、v1.9.x、および v1.8.x

物理デバイスでテストする場合、 AWS IoT Greengrass v1.10、v1.9.x、および v1.8.x がサポートされています。

Docker コンテナでテストする場合、 AWS IoT Greengrass v1.10 と v1.9.x がサポートされています。

リリースノート:

- the section called “Docker コンテナ AWS IoT Greengrass で を実行する” のサポートが追加されました。IDT を使用して、デバイスが Docker コンテナ AWS IoT Greengrass で実行できることを認定および検証できるようになりました。
- AWS IoT Device Tester の実行に必要なアクセス許可を定義する AWS マネージドポリシー (`AWSIoTDeviceTesterForGreengrassFullAccess`) を追加しました。新しいリリースで追加のアクセス許可が必要な場合は、IAM アクセス許可を更新する必要がないように、 がこの管理ポリシー AWS に追加します。
- テストケースを実行する前に、環境 (デバイス接続やインターネット接続など) が正しく設定されていることを検証するためのチェックを導入しました。
- IDT の Greengrass 依存関係チェッカーを改善し、デバイスで libc を確認する際の柔軟性を高めました。

IDT v2.2.0 for AWS IoT Greengrass v1.10、v1.9.x、および v1.8.x

リリースノート:

- v1.10 AWS IoT Greengrass のサポートが追加されました。
- Greengrass Docker アプリケーションデプロイコネクタのサポートが追加されました。
- AWS IoT Greengrass ストリームマネージャーのサポートを追加しました。
- 中国 (北京) リージョン AWS IoT Greengrass で のサポートが追加されました。

IDT v2.1.0 for AWS IoT Greengrass v1.9.x、v1.8.x、および v1.7.x

リリースノート:

- v1.9.4 AWS IoT Greengrass のサポートを追加しました。
- Linux-ARMv6l デバイスのサポートが追加されました。

IDT v2.0.0 for AWS IoT Greengrass v1.9.3、v1.9.2、v.1.9.1、v1.9.0、v1.8.4、v1.8.3、v1.8.2

リリースノート:

- テスト対象デバイスの Python への依存関係を削除しました。
- テストスイートの実行時間が 50% 以上短縮され、適格性確認プロセスが高速化されます。
- 実行可能なサイズが 50% 以上短縮され、ダウンロードとインストールが高速になります。
- すべてのテストケースでタイムアウト乗数のサポートを改善しました。
- エラーのトラブルシューティングを高速化するため、診断後のメッセージが強化されました。
- IDT の実行に必要なアクセス許可ポリシーテンプレートを更新しました。
- v1.9.3 AWS IoT Greengrass のサポートを追加しました。

IDT v1.9.2、v1. AWS IoT Greengrass 9.1、v1.9.0、v1.8.3、v1.8.2 用 v1.3.3

リリースノート:

- Greengrass v1.9.2 および v1.8.3 のサポートが追加されました。
- Greengrass OpenWrt のサポートが追加されました。
- デバイスにサインインする SSH ユーザー名とパスワードを追加しました。
- OpenWrt-ARMv7l プラットフォーム用のネイティブテストバグ修正が追加されました。

IDT v1.2 for AWS IoT Greengrass v1.8.1

リリースノート:

- タイムアウトの問題 (たとえば、低帯域幅接続) に対処し、トラブルシューティングする設定可能なタイムアウト乗数が追加されました。

IDT v1.1 for AWS IoT Greengrass v1.8.0

リリースノート:

- AWS IoT Greengrass ハードウェアセキュリティ統合 (HSI) のサポートが追加されました。
- AWS IoT Greengrass コンテナのサポートが追加されました。コンテナはありません。
- 自動 AWS IoT Greengrass サービスロールの作成を追加しました。

- テストリソースのクリーンアップが改善されました。
- テスト実行の要約レポートが追加されました。

IDT v1.1 for AWS IoT Greengrass v1.7.1

リリースノート:

- AWS IoT Greengrass ハードウェアセキュリティ統合 (HSI) のサポートが追加されました。
- AWS IoT Greengrass コンテナのサポートが追加されました。コンテナはありません。
- 自動 AWS IoT Greengrass サービスロールの作成を追加しました。
- テストリソースのクリーンアップが改善されました。
- テスト実行の要約レポートが追加されました。

IDT v1.0 for AWS IoT Greengrass v1.6.1

リリースノート:

- 将来の AWS IoT Greengrass バージョンの互換性のために OTA テストバグ修正を追加しました。

> **Note**
>
> IDT v1.0 for v1.6.1 AWS IoT Greengrass を使用している場合は、Greengrass サービスロールを作成する必要があります。これより後のバージョンでは、IDT によってサービスロールが自動的に作成されます。

IDT を使用して AWS IoT Greengrass 認定スイートを実行する

の AWS IoT Device Tester (IDT) を使用して AWS IoT Greengrass 、 AWS IoT Greengrass Core ソフトウェアがハードウェア上で実行され、 と通信できることを確認できます AWS クラウド。また、エンドツーエンドのテストが AWS IoT Coreで実行されます。例えば、デバイスで MQTT メッセージを送受信して正しく処理できることを確認します。

AWS IoT Greengrass Version 1 がメンテナンスモードに移行されたため、IDT for は署名付き認定レポートを生成 AWS IoT Greengrass V1 しなくなりました。Device AWS Partner Catalog にハードウェアを追加する場合は、 AWS IoT Greengrass V2 認定スイートを実行して、送信できるテストレ

ポートを生成します AWS IoT。詳細については、「AWS デバイス認定プログラム」および「IDT for AWS IoT Greengrass V2のサポートされているバージョン」を参照してください。

デバイスのテストに加えて、IDT for は にリソース (AWS IoT モノ、 AWS IoT Greengrass グループ、Lambda 関数など) AWS IoT Greengrass を作成し、認定プロセス AWS アカウント を容易にします。

これらのリソースを作成するために、IDT for AWS IoT Greengrass は `config.json` ファイルで設定された AWS 認証情報を使用してユーザーに代わって API コールを行います。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

IDT for を使用して認定スイート AWS IoT Greengrass を実行すると、IDT AWS IoT Greengrass は次のステップを実行します。

1. デバイスおよび認証情報の設定をロードして検証します。
2. 必要なローカルリソースとクラウドリソースを使用して選択したテストを実行します。
3. ローカルリソースとクラウドリソースをクリーンアップします。
4. デバイスが資格に必要なテストに合格したかどうかを示すテストレポートを生成します。

テストスイートのバージョン

IDT for は、テストをテストスイートとテストグループに AWS IoT Greengrass 整理します。

- テストスイートは、デバイスが AWS IoT Greengrassの特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、Greengrass グループデプロイや MQTT メッセージングなど、特定の機能に関連する個々のテストのセットです。

IDT v3.0.0 以降、テストスイートは *`major.minor.patch`* 形式を使用してバージョン管理されます (例: `GGQ_1.0.0`)。IDT をダウンロードすると、パッケージに最新のテストスイートのバージョンが含まれます。

> **Important**
>
> IDT では、デバイスの認定のために 3 つの最新のテストスイートバージョンをサポートしています。詳細については、「the section called “の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1”」を参照してください。

`list-supported-products` を実行して、現在のバージョンの IDT でサポートされている AWS IoT Greengrass およびテストスイートのバージョンを一覧表示できます。サポートされていないテストスイートのバージョンからのテストは、デバイスの認定には有効ではありません。IDT では、サポートされていないバージョンの認定レポートは印刷されません。

IDT 構成設定の更新

新しいテストでは、新しい IDT 構成設定が導入される可能性があります。

- その設定がオプションの場合は、IDT は引き続きテストを実行します。
- その設定が必要な場合は、IDT から通知され、実行が停止します。設定を構成したら、テストの実行を再開します。

  構成設定は、*<device-tester-extract-location>*/configs フォルダにあります。詳細については、「the section called “IDT 設定を設定する”」を参照してください。

更新されたテストスイートのバージョンによって構成設定が追加されると、IDT は *<device-tester-extract-location>*/configs に元の設定ファイルのコピーを作成します。

テストグループの説明

IDT v2.0.0 and later

コア資格に必要なテストグループ

これらのテストグループは、 AWS IoT Greengrass デバイスを AWS Partner Device Catalog に認定するために必要です。

AWS IoT Greengrass コア依存関係

デバイスが AWS IoT Greengrass Core ソフトウェアのすべてのソフトウェアおよびハードウェア要件を満たしていることを検証します。

このテストグループの `Software Packages Dependencies` テストケースは、Docker コンテナでテストする場合は適用されません。

デプロイ

Lambda 関数をデバイスにデプロイできることを検証します。

MQTT

Greengrass コアとローカル IoT デバイスであるクライアントデバイス間のローカル通信をチェックして、 AWS IoT Greengrass メッセージルーターの機能を検証します。

無線 (OTA)

デバイスが AWS IoT Greengrass Core ソフトウェアの OTA 更新を正常に実行できることを検証します。

このテストグループは、Docker コンテナでテストするときには適用されません。

バージョン

AWS IoT Greengrass 提供された のバージョンが、使用している AWS IoT Device Tester のバージョンと互換性があることを確認します。

オプションのテストグループ

これらのテストグループはオプションです。オプションテストの資格を選択した場合、デバイスは AWS Partner Device Catalog に追加機能とともに一覧表示されます。

コンテナの依存関係

デバイスが Greengrass コアのコンテナモードで Lambda 関数を実行するためのすべてのソフトウェア要件とハードウェア要件を満たしているかどうかを検証します。

このテストグループは、Docker コンテナでテストするときには適用されません。

デプロイコンテナ

Lambda 関数をデバイスにデプロイし、Greengrass コアのコンテナモードで実行できることを検証します。

このテストグループは、Docker コンテナでテストするときには適用されません。

Docker 依存関係 (IDT v2.2.0 以降でサポートされています)

Greengrass Docker アプリケーションデプロイコネクタを使用してコンテナを実行するために必要なすべての技術的依存関係をデバイスが満たしていることを検証します。

このテストグループは、Docker コンテナでテストするときには適用されません。

ハードウェアセキュリティ統合 (HSI)

提供された HSI 共有ライブラリがハードウェアセキュリティモジュール (HSM) とやり取りでき、必要な PKCS#11 API を正しく実装することを検証します。HSM および共有ライ

ブラリは、CSR に署名し、TLS オペレーションを実行して、正しいキー長と公開キーアルゴリズムを提供できる必要があります。

ストリームマネージャーの依存関係 (IDT v2.2.0 以降でサポート)

デバイスが AWS IoT Greengrass ストリームマネージャーを実行するために必要な技術的な依存関係をすべて満たしていることを検証します。

機械学習の依存関係 (IDT v3.1.0 以降でサポート)

ML 推論をローカルで実行するために必要なすべての技術的依存関係をデバイスが満たしていることを検証します。

機械学習の推論テスト (IDT v3.1.0 以降でサポート)

テスト中の特定のデバイスで ML 推論を実行できることを検証します。詳細については、「the section called “オプション: ML 認定のためのデバイスの設定”」を参照してください。

機械学習の推論コンテナテスト (IDT v3.1.0 以降でサポート)

テスト中の特定のデバイスで ML 推論を実行でき、Greengrass コアのコンテナモードで実行できることを検証します。詳細については、「the section called “オプション: ML 認定のためのデバイスの設定”」を参照してください。

IDT v1.3.3 and earlier

コア資格に必要なテストグループ

これらのテストは、 AWS IoT Greengrass デバイスを AWS Partner Device Catalog に認定するために必要です。

AWS IoT Greengrass コア依存関係

デバイスが AWS IoT Greengrass Core ソフトウェアのすべてのソフトウェアおよびハードウェア要件を満たしていることを検証します。

コンビネーション (デバイスのセキュリティ操作)

クラウド内の Greengrass グループの接続情報を変更することで、Greengrass コアの Device Certificate Manager と IP 検出の動作を検証します。テストグループは AWS IoT Greengrass サーバー証明書をローテーションし、 が接続 AWS IoT Greengrass を許可することを確認します。

デプロイ（IDT v1.2 以前のバージョンで必要）

Lambda 関数をデバイスにデプロイできることを検証します。

Device Certificate Manager (DCM)

AWS IoT Greengrass デバイス証明書マネージャーが起動時にサーバー証明書を生成し、有効期限が近い場合は証明書をローテーションできることを確認します。

IP 検出 (IPD)

Greengrass コアデバイスでの IP アドレスの変更に応じてコア接続情報が更新されることを検証します。詳細については、「自動 IP 検出のアクティブ化」を参照してください。

ロギング

AWS IoT Greengrass ログ記録サービスが Python で記述されたユーザー Lambda 関数を使用してログファイルに書き込めることを確認します。

MQTT

2 つの Lambda 関数にルーティングされるトピックでメッセージを送信することで、AWS IoT Greengrass メッセージルーターの機能を検証します。

ネイティブ

がネイティブ (コンパイル済み) Lambda 関数 AWS IoT Greengrass を実行できることを確認します。

無線 (OTA)

デバイスが AWS IoT Greengrass Core ソフトウェアの OTA 更新を正常に実行できることを検証します。

侵入

ハードリンク/ソフトリンクの保護と seccomp が有効になっていない場合、 AWS IoT Greengrass Core ソフトウェアが起動しないことを確認します。また、他のセキュリティ関連の機能も検証するために使用されます。

シャドウ

ローカルのシャドウとシャドウのクラウド同期機能を検証します。

スプーラー

スプーラーのデフォルト設定を使用して MQTT メッセージをキューに挿入できることを検証します。

Token Exchange Service (TES)

がコア証明書を有効な AWS 認証情報と交換 AWS IoT Greengrass できることを確認します。

バージョン

AWS IoT Greengrass 提供された のバージョンが、使用している AWS IoT Device Tester のバージョンと互換性があることを確認します。

オプションのテストグループ

これらのテストはオプションです。オプションテストの資格を選択した場合、デバイスは AWS Partner Device Catalog に追加機能とともに一覧表示されます。

コンテナの依存関係

コンテナモードで Lambda 関数を実行するために必要なすべての依存関係をデバイスが満たしていることを確認します。

ハードウェアセキュリティ統合 (HSI)

提供された HSI 共有ライブラリがハードウェアセキュリティモジュール (HSM) とやり取りでき、必要な PKCS#11 API を正しく実装することを検証します。HSM および共有ライブラリは、CSR に署名し、TLS オペレーションを実行して、正しいキー長と公開キーアルゴリズムを提供できる必要があります。

ローカルリソースアクセス

さまざまな Linux ユーザーやグループが所有するローカルファイルやディレクトリへのアクセスを LRA API を介してコンテナ化された Lambda 関数に提供 AWS IoT Greengrass することで、 のローカルリソースアクセス (LRA) AWS IoT Greengrass 機能を検証します。 APIs ローカルリソースアクセスの設定に基づいて、ローカルリソースへのアクセスを Lambda 関数に許可または拒否する必要があります。

ネットワーク

Lambda 関数からソケット接続を確立できることを検証します。Greengrass コア設定に基づいて、これらのソケット接続を許可または拒否する必要があります。

AWS IoT Greengrass 認定スイートを実行するための前提条件

このセクションでは、 AWS IoT Greengrass の AWS IoT Device Tester (IDT) AWS IoT Greengrass を使用して認定スイートを実行するための前提条件について説明します。

の最新バージョンの AWS IoT Device Tester をダウンロードする AWS IoT Greengrass

IDT の最新バージョンをダウンロードし、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。
> Windows では、パスの長さは 260 文字に制限されています。Windows を使用している場合は、パスが 260 文字以内になるようにして、IDT をルートディレクトリ (`C:\` または `D:\` など) に展開します。

の作成と設定 AWS アカウント

IDT for を使用する前に AWS IoT Greengrass、次の手順を実行する必要があります。

1. を作成します AWS アカウント。が既にある場合は AWS アカウント、ステップ 2 に進みます。
2. IDT 用のアクセス許可を設定する。

これらのアカウントのアクセス許可により、IDT はユーザーに代わって AWS サービスにアクセスし、 AWS IoT モノ、Greengrass グループ、Lambda 関数などの AWS リソースを作成できます。

これらのリソースを作成するために、IDT for AWS IoT Greengrass は `config.json` ファイルで設定された AWS 認証情報を使用してユーザーに代わって API コールを行います。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

> **Note**
>
> ほとんどのテストはアマゾン ウェブ サービス無料利用枠の対象となりますが、 AWS アカウントにサインアップするときにクレジットカード情報を提供する必要があります。詳細については、「アカウントが無料利用枠の対象であるのに、支払い方法が必要なのはなぜですか?」を参照してください。

ステップ 1: を作成する AWS アカウント

このステップでは、 AWS アカウントを作成して設定します。が既にある場合は AWS アカウント、「」に進みますthe section called “ステップ 2: IDT 用のアクセス許可を設定する”。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。

2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

ステップ 2: IDT 用のアクセス許可を設定する

このステップでは、IDT for がテストを実行し、IDT AWS IoT Greengrass 使用状況データを収集するために使用するアクセス許可を設定します。 AWS Management Console または AWS Command Line Interface （ AWS CLI) を使用して IDT の IAM ポリシーとテストユーザーを作成し、ユーザーにポリシーをアタッチできます。IDT 用のテストユーザーをすでに作成している場合は、「the section called “IDT テストを実行するようにデバイスを設定する”」または「the section called “オプション: Docker コンテナの設定”」に進みます。

- IDT 用のアクセス許可を設定するには (コンソール)
- IDT 用のアクセス許可を設定するには (AWS CLI)

IDT 用のアクセス許可を設定するには (コンソール)

コンソールを使用して IDT for AWS IoT Greengrass用のアクセス許可を設定するには、次のステップに従ってください。

1. IAM コンソールにサインインします。
2. 特定のアクセス許可を持つロールを作成するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。

   a. ナビゲーションペインで ポリシーを選択してから ポリシーの作成を選択します。

   b. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageRolePoliciesForIDTGreengrass",
            "Effect": "Allow",
            "Action": [
                "iam:DetachRolePolicy",
                "iam:AttachRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iam::*:role/GreengrassServiceRole"
            ],
            "Condition": {
                "ArnEquals": {
```

```
                    "iam:PolicyARN": [
                        "arn:aws:iam::aws:policy/service-role/
AWSGreengrassResourceAccessRolePolicy",
                        "arn:aws:iam::aws:policy/service-role/
GreengrassOTAUpdateArtifactAccess",
                        "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                    ]
                }
            }
        },
        {
            "Sid": "ManageRolesForIDTGreengrass",
            "Effect": "Allow",
            "Action": [
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:PassRole",
                "iam:GetRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iam::*:role/GreengrassServiceRole"
            ]
        }
    ]
}
```

> **Important**
>
> 次のポリシーは、IDT for AWS IoT Greengrassに必要なロールを作成および管理するためのアクセス許可を付与します。これには、次の AWS 管理ポリシーをアタッチするアクセス許可が含まれます。
>
> - AWSGreengrassResourceAccessRolePolicy
> - GreengrassOTAUpdateArtifactAccess
> - AWSLambdaBasicExecutionRole

c. [Next: Tags (次へ: タグ)] を選択します。

d. [次へ: レビュー] を選択します。

e. [名前] に**IDTGreengrassIAMPermissions**と入力してください。[概要] で、ポリシーによって付与されたアクセス許可を確認します。

f. [Create policy] (ポリシーの作成) を選択します。

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrassに必要なアクセス許可をアタッチします。

   a. IAM ユーザーを作成します。IAM ユーザーガイドの IAM ユーザーの作成 (コンソール) のステップ 1 ~ 5 に従います。

   b. アクセス許可を IAM ユーザーにアタッチします。

      i. [Set permissions] (許可を設定) ページで、[Attach existing policies directly] (既存のポリシーを直接アタッチする) を選択します。

      ii. 前のステップで作成した IDTGreengrassIAMPermissions ポリシーを検索します。チェックボックスをオンにします。

      iii. AWSIoTDeviceTesterForGreengrassFullAccess ポリシーを検索します。チェックボックスをオンにします。

      > **Note**
      >
      > AWSIoTDeviceTesterForGreengrassFullAccess は、IDT がテストに使用される AWS リソースを作成およびアクセスするために必要なアクセス許可を定義する AWS マネージドポリシーです。詳細については、「the section called “AWS IDT の マネージドポリシー”」を参照してください。

   c. [Next: Tags] (次へ: タグ) を選択します。

   d. [Next: Review] (次へ: レビュー) を選択して、選択内容の概要を表示します。

   e. [ユーザーの作成] を選択します。

   f. ユーザーのアクセスキー (アクセスキー ID とシークレットアクセスキー) を表示するには、パスワードとアクセスキーの横にある [Show (表示)] を選択します。アクセスキーを保存するには、[Download .csv] を選択し、安全な場所にファイルを保存します。この情報を後で使用して、 AWS 認証情報ファイルを設定します。

4. 次のステップ: 物理デバイスを設定します。

IDT 用のアクセス許可を設定するには (AWS CLI)

を使用して IDT for のアクセス許可を設定するには AWS CLI 、次の手順に従います AWS IoT Greengrass。コンソールでアクセス許可をすでに設定している場合は、「the section called “IDT テストを実行するようにデバイスを設定する”」または「the section called “オプション: Docker コンテナの設定”」に進みます。

1. コンピュータで、まだインストールされていない場合 AWS CLI は、 をインストールして設定します。AWS Command Line Interface ユーザーガイドの AWS CLIのインストールのステップに従います。

   > **Note**
   >
   > AWS CLI は、コマンドラインシェルから AWS サービスとやり取りするために使用できるオープンソースツールです。

2. IDT と AWS IoT Greengrass ロールを管理するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。

   Linux, macOS, or Unix

```
aws iam create-policy --policy-name IDTGreengrassIAMPermissions --policy-
document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageRolePoliciesForIDTGreengrass",
            "Effect": "Allow",
            "Action": [
                "iam:DetachRolePolicy",
                "iam:AttachRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iam::*:role/GreengrassServiceRole"
            ],
            "Condition": {
                "ArnEquals": {
                    "iam:PolicyARN": [
                        "arn:aws:iam::aws:policy/service-role/
AWSGreengrassResourceAccessRolePolicy",
```

```
                        "arn:aws:iam::aws:policy/service-role/
GreengrassOTAUpdateArtifactAccess",
                        "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                    ]
                }
            }
        },
        {
            "Sid": "ManageRolesForIDTGreengrass",
            "Effect": "Allow",
            "Action": [
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:PassRole",
                "iam:GetRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iam::*:role/GreengrassServiceRole"
            ]
        }
    ]
}'
```

Windows command prompt

```
aws iam create-policy --policy-name IDTGreengrassIAMPermissions --
policy-document '{\"Version\": \"2012-10-17\", \"Statement\": [{\"Sid
\": \"ManageRolePoliciesForIDTGreengrass\",\"Effect\": \"Allow\",
\"Action\": [\"iam:DetachRolePolicy\", \"iam:AttachRolePolicy\"],
 \"Resource\": [\"arn:aws:iam::*:role/idt-*\",\"arn:aws:iam::*:role/
GreengrassServiceRole\"],\"Condition\": {\"ArnEquals\": {\"iam:PolicyARN\":
 [\"arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
\",\"arn:aws:iam::aws:policy/service-role/GreengrassOTAUpdateArtifactAccess
\",\"arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole\"]}}},
{\"Sid\": \"ManageRolesForIDTGreengrass\",\"Effect\": \"Allow\",\"Action\":
 [\"iam:CreateRole\",\"iam:DeleteRole\", \"iam:PassRole\", \"iam:GetRole
\"],\"Resource\": [\"arn:aws:iam::*:role/idt-*\",\"arn:aws:iam::*:role/
GreengrassServiceRole\"]}]}'
```

> **Note**
>
> このステップには、Linux、macOS、または Unix のターミナルコマンドとは異なる JSON 構文を使用するため、Windows コマンドプロンプトの例が含まれています。

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrassに必要なアクセス許可をアタッチします。

   a. IAM ユーザーを作成します。このセットアップ例では、ユーザーは `IDTGreengrassUser` という名前になります。

   ```
   aws iam create-user --user-name IDTGreengrassUser
   ```

   b. ステップ 2 で作成した `IDTGreengrassIAMPermissions` ポリシーを IAM ユーザーにアタッチします。コマンドの *`<account-id>`* を の ID に置き換えます AWS アカウント。

   ```
   aws iam attach-user-policy --user-name IDTGreengrassUser --policy-arn
    arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
   ```

   c. `AWSIoTDeviceTesterForGreengrassFullAccess` ポリシーを IAM ユーザーにアタッチします。

   ```
   aws iam attach-user-policy --user-name IDTGreengrassUser --policy-arn
    arn:aws:iam::aws:policy/AWSIoTDeviceTesterForGreengrassFullAccess
   ```

   > **Note**
   >
   > AWSIoTDeviceTesterForGreengrassFullAccess は、IDT がテストに使用される AWS リソースを作成およびアクセスするために必要なアクセス許可を定義する AWS マネージドポリシーです。詳細については、「the section called “AWS IDT のマネージドポリシー”」を参照してください。

4. ユーザーのシークレットアクセスキーを作成します。

   ```
   aws iam create-access-key --user-name IDTGreengrassUser
   ```

   この出力は安全な場所に保存してください。後でこの情報を使用して認証情報 AWS ファイルを設定します。

5. 次のステップ: 物理デバイスを設定します。

AWSAWS IoT Device Tester の マネージドポリシー

AWSIoTDeviceTesterForGreengrassFullAccess 管理ポリシーにより、IDT はオペレーションを実行し、使用状況メトリクスを収集できます。このポリシーは以下の IDT アクセス許可を与えます。

- `iot-device-tester:CheckVersion`。 AWS IoT Greengrass、テストスイート、および IDT バージョンのセットに互換性があるかどうかを確認します。
- `iot-device-tester:DownloadTestSuite`。テストスイートをダウンロードします。
- `iot-device-tester:LatestIdt`。ダウンロード可能な最新の IDT バージョンに関する情報を取得します。
- `iot-device-tester:SendMetrics`。IDT がテストに関して収集する使用状況データを公開します。
- `iot-device-tester:SupportedVersion`。 IDT でサポートされている AWS IoT Greengrass およびテストスイートのバージョンのリストを取得します。この情報は、コマンドラインウィンドウに表示されます。

IDT テストを実行するようにデバイスを設定する

デバイスを設定するには、 AWS IoT Greengrass 依存関係をインストールし、 AWS IoT Greengrass Core ソフトウェアを設定し、デバイスにアクセスするようにホストコンピュータを設定し、デバイスに対するユーザーアクセス許可を設定する必要があります。

テスト対象のデバイスへの AWS IoT Greengrass 依存関係を検証する

IDT for AWS IoT Greengrass がデバイスをテストする前に、「 の開始方法 AWS IoT Greengrass」の説明に従ってデバイスをセットアップしていることを確認してください。サポートされているプラットフォームについては、「サポートされているプラットフォーム」を参照してください。

AWS IoT Greengrass ソフトウェアを設定する

IDT for は、特定のバージョンの との互換性についてデバイスを AWS IoT Greengrass テストします AWS IoT Greengrass。IDT には、デバイスでテストするための 2 AWS IoT Greengrass つのオプションがあります。

- AWS IoT Greengrass Core ソフトウェアのバージョンをダウンロードして使用します。IDT によってソフトウェアがインストールされます。
- デバイスに既にインストールされている AWS IoT Greengrass Core ソフトウェアのバージョンを使用します。

> **Note**
>
> の各バージョンには、対応する IDT バージョン AWS IoT Greengrass があります。 AWS IoT Greengrass 使用している のバージョンに対応する IDT のバージョンをダウンロードする必要があります。

以下のセクションでは、これらについて説明します。1 度のみ実行してください。

オプション 1: AWS IoT Greengrass Core ソフトウェアをダウンロードし、それを使用するように Device Tester を設定する AWS IoT

AWS IoT Greengrass Core ソフトウェアのダウンロードページから AWS IoT Greengrass Core ソフトウェアをダウンロードできます。

1. 適切なアーキテクチャと Linux ディストリビューションを見つけ、[ダウンロード] を選択します。
2. tar.gz ファイルを `<device-tester-extract-location>/products/greengrass/ggc` にコピーします。

> **Note**
>
> AWS IoT Greengrass tar.gz ファイルの名前を変更しないでください。同じオペレーティングシステムとアーキテクチャのこのディレクトリに複数のファイルを配置しないでください。たとえば、`greengrass-linux-armv7l-1.7.1.tar.gz` および `greengrass-linux-armv7l-1.8.1.tar.gz` のファイルをいずれもそのディレクトリに置くと、テストが失敗する原因になります。

オプション 2: Device Tester AWS IoT Greengrass で AWS IoT の既存のインストールを使用する

`<device-tester-extract-location>/configs` フォルダの `device.json` ファイルに `greengrassLocation` 属性を追加して、デバイスにインストールされている AWS IoT Greengrass Core ソフトウェアをテストするように IDT を設定します。以下に例を示します。

```
"greengrassLocation" : "<path-to-greengrass-on-device>"
```

`device.json` ファイルの詳細については、「device.json の設定」を参照してください。

Linux デバイスの場合、 AWS IoT Greengrass Core ソフトウェアのデフォルトの場所は です/`greengrass`。

> **Note**
>
> デバイスには、起動されていない AWS IoT Greengrass Core ソフトウェアがインストールされている必要があります。
>
> `ggc_user` ユーザーおよび `ggc_group` がデバイスに追加されていることを確認します。詳細については、「AWS IoT Greengrassの環境設定」を参照してください。

テスト対象デバイスにアクセスするようにホストコンピュータを設定する

IDT はホストコンピュータで動作し、SSH を使用してデバイスに接続できる必要があります。IDT がテスト対象のデバイスへの SSH アクセスを許可するには、2 つのオプションがあります。

1. こちらの手順に従って SSH キーペアを作成し、パスワードを指定せずにテスト対象のデバイスにサインインすることをキーに承認します。
2. `device.json` ファイルに各デバイスのユーザー名とパスワードを入力します。詳細については、「device.json の設定」を参照してください。

任意の SSL 実装を使用して SSH キーを作成できます。次の手順は、SSH-KEYGEN または PuTTYgen (Windows の場合) を使用する方法を示しています。別の SSL 実装を使用する場合は、その実装に関するドキュメントを参照してください。

テスト対象デバイスで認証するには、IDT で SSH キーを使用します。

SSH-KEYGEN を使用して SSH キーを作成するには

1. SSH キーを作成します。

   OpenSSH ssh-keygen コマンドを使用して SSH キーペアを作成できます。ホストコンピュータに SSH キーペアがすでにある場合は、IDT 専用の SSH キーペアを作成することをお勧めします。こうすることで、テストを完了した後、ホストコンピュータはパスワードを入力しないとデバイスに接続できなくなります。また、リモートデバイスへのアクセスを必要なユーザーのみに制限することもできます。

> **Note**
>
> Windows に SSH クライアントがインストールされていません。Windows での SSH クライアントのインストールについては、「SSH クライアントソフトウェアをダウンロードする」を参照してください。

ssh-keygen コマンドは、キーペアを保存する名前とパスの入力を求めます。デフォルトでは、キーペアファイルの名前は `id_rsa` (プライベートキー) と `id_rsa.pub` (パブリックキー) です。macOS および Linux の場合、これらのファイルのデフォルトの場所は `~/.ssh/` です。Windows の場合、デフォルトの場所は `C:\Users\<user-name>\.ssh` です。

プロンプトが表示されたら、SSH キーを保護するキーフレーズを入力します。詳細については、「新しい SSH キーを生成する」を参照してください。

2. テスト対象デバイスに承認済み SSH キーを追加します。

   IDT で SSH プライベートキーを使用して、テスト対象デバイスにサインインする必要があります。SSH プライベートキーがテスト対象デバイスにサインインすることを承認するには、ホストコンピュータから ssh-copy-id コマンドを使用します。このコマンドは、テスト対象デバイスの `~/.ssh/authorized_keys` ファイルにパブリックキーを追加します。例:

   $ ssh-copy-id ***<remote-ssh-user>***@***<remote-device-ip>***

   `remote-ssh-user` は、テスト対象デバイスへのサインインに使用するユーザー名です。*`remote-device-ip`* は、テスト対象デバイスの IP アドレスです。例:

   ssh-copy-id pi@192.168.1.5

   プロンプトが表示されたら、ssh-copy-id コマンドで指定したユーザー名に対応するパスワードを入力します。

   ssh-copy-id では、パブリックキー名が `id_rsa.pub` で、デフォルトの保存先が `~/.ssh/` (macOS と Linux の場合) または `C:\Users\<user-name>\.ssh` (Windows の場合) であるとみなされます。パブリックキーに別の名前や別の保存先を指定した場合は、ssh-copy-id で -i オプションを使用し、SSH 公開鍵への完全修飾パス (ssh-copy-id -i ~/my/path/myKey.pub など) を指定する必要があります。SSH キーの作成とパブリックキーのコピーの詳細については、「SSH-COPY-ID」を参照してください。

PuTTYgen を使用して SSH キーを作成するには (Windows のみ)

1. テスト対象デバイスに OpenSSH サーバーとクライアントがインストールされていることを確認します。詳細については、「OpenSSH」を参照してください。
2. テスト対象のデバイスに PuTTYgen をインストールします。
3. PuTTYGen を開きます。
4. [Generate] を選択し、ボックス内にマウスカーソルを移動してプライベートキーを生成します。
5. [Conversions] メニューから [Export OpenSSH key] を選択し、プライベートキーに `.pem` ファイル拡張子を付けて保存します。
6. テスト対象デバイスの `/home/<user>/.ssh/authorized_keys` ファイルにパブリックキーを追加します。
   a. PuTTYgen ウィンドウからパブリックキーテキストをコピーします。
   b. PuTTY を使用して、テスト対象のデバイスでセッションを作成します。
      i. コマンドプロンプトまたは Windows Powershell ウィンドウから、次のコマンドを実行します。

         C:/***<path-to-putty>***/putty.exe -ssh ***<user>***@***<dut-ip-address>***

      ii. プロンプトが表示されたら、デバイスのパスワードを入力します。
      iii. vi などのテキストエディタを使用して、テスト対象のデバイスの `/home/<user>/.ssh/authorized_keys` ファイルにパブリックキーを追加します。
7. `device.json` ファイルを、ユーザー名、IP アドレス、およびテスト対象の各デバイスのホストコンピュータに保存したプライベートキーファイルへのパスで更新します。詳細については、「the section called “device.json の設定”」を参照してください。必ずプライベートキーの完全パスとファイル名を指定し、スラッシュ (「/」) を使用してください。たとえば、Windows パス `C:\DT\privatekey.pem` の場合は、`device.json` ファイルで `C:/DT/privatekey.pem` を使用します。

デバイスに対するユーザーのアクセス許可を設定する

IDT は、テスト対象デバイスのさまざまなディレクトリやファイルに対してオペレーションを実行します。このようなオペレーションの中には、高いアクセス許可が必要な場合があります (sudo を使用)。これらのオペレーションを自動化するには、IDT for がパスワードの入力を求められることなく sudo でコマンドを実行できる AWS IoT Greengrass 必要があります。

パスワードの入力を求めることなく、sudo にアクセスを許可するには、テスト対象デバイスで以下の手順を実行します。

> **Note**
>
> `username` は、テスト対象デバイスにアクセスするために IDT で使用する SSH ユーザーを指します。

ユーザーを sudo グループに追加するには

1. テスト対象のデバイスで、`sudo usermod -aG sudo <username>` を実行します。
2. サインアウトし、再度サインインして、変更を反映します。
3. ユーザー名が正常に追加されたことを確認するには、sudo echo test を実行します。パスワードの入力を要求されない場合、ユーザーは正しく設定されています。
4. `/etc/sudoers` ファイルを開き、ファイルの末尾に次の行を追加します:

   ```
   <ssh-username> ALL=(ALL) NOPASSWD: ALL
   ```

オプション機能をテストするためのデバイスの設定

以下のトピックでは、オプション機能の IDT テストを実行するようにデバイスを設定する方法について説明します。これらの機能をテストする場合のみ、以下の設定手順に従ってください。それ以外の場合は、「the section called “IDT 設定を設定する”」に進みます。

トピック

- オプション: IDT for の Docker コンテナの設定 AWS IoT Greengrass
- オプション: ML 認定のためのデバイスの設定

オプション: IDT for の Docker コンテナの設定 AWS IoT Greengrass

AWS IoT Greengrass は、Docker コンテナで AWS IoT Greengrass Core ソフトウェアを簡単に実行できるようにする Docker イメージと Dockerfile を提供します。 AWS IoT Greengrass コンテナを設定したら、IDT テストを実行できます。現在、IDT for AWS IoT Greengrassを実行するためにサポートされているアーキテクチャは x86_64 Docker のみです。

この機能を使用するには、IDT v2.3.0 以降が必要です。

IDT テストを実行するように Docker コンテナを設定するプロセスは、 が提供する Docker イメージと Dockerfile のどちらを使用するかによって異なります AWS IoT Greengrass。

- Docker イメージを使用する。Docker イメージには AWS IoT Greengrass Core ソフトウェアと依存関係がインストールされています。
- Dockerfile を使用する。Dockerfile には、カスタム AWS IoT Greengrass コンテナイメージの構築に使用できるソースコードが含まれています。イメージは、別のプラットフォームアーキテクチャで実行できるようにするため、またはイメージサイズを縮小するために変更できます。

> **Note**
>
> AWS IoT Greengrass は、 AWS IoT Greengrass コアソフトウェアバージョン 1.11.1 の Dockerfile または Docker イメージを提供しません。独自のカスタムコンテナイメージで IDT テストを実行するには、イメージに が提供する Dockerfile で定義されている依存関係が含まれている必要があります AWS IoT Greengrass。

Docker コンテナ AWS IoT Greengrass で を実行する場合、次の機能は利用できません。

- [Greengrass container] (Greengrass コンテナ) モードで実行されるコネクタ。Docker コンテナでコネクタを実行するには、コネクタをコンテナなしモードで実行する必要があります。コンテナなしモードをサポートするコネクタを検索するには、「the section called “AWSが提供する Greengrass コネクタ”」を参照してください。これらのコネクタの一部では、分離モードパラメータを使用されており、[コンテナなし] に設定する必要があります。
- ローカルデバイスおよびボリュームリソース。Docker コンテナで実行されるユーザー定義 Lambda 関数は、コア上のデバイスとボリュームに直接アクセスする必要があります。

が提供する Docker イメージを設定する AWS IoT Greengrass

IDT テストを実行するように AWS IoT Greengrass Docker イメージを設定するには、次の手順に従います。

前提条件

このチュートリアルを開始する前に、以下を実行する必要があります。

- 選択した AWS Command Line Interface（AWS CLI) バージョンに基づいて、ホストコンピュータに次のソフトウェアとバージョンをインストールする必要があります。

AWS CLI version 2

- Docker バージョン 18.09 以降。以前のバージョンでも動作する可能性がありますが、18.09 以降を推奨します。
- AWS CLI バージョン 2.0.0 以降。
  - AWS CLI バージョン 2 をインストールするには、「バージョン 2 AWS CLIのインストール」を参照してください。
  - を設定するには AWS CLI、「 の設定 AWS CLI」を参照してください。

> **Note**
>
> Windows コンピュータで新しい AWS CLI バージョン 2 にアップグレードするには、MSI のインストールプロセスを繰り返す必要があります。

AWS CLI version 1

- Docker バージョン 18.09 以降。以前のバージョンでも動作する可能性がありますが、18.09 以降を推奨します。
- Python バージョン 3.6 以降。
- pip バージョン 18.1 以降。
- AWS CLI バージョン 1.17.10 以降
  - AWS CLI バージョン 1 をインストールするには、「バージョン 1 AWS CLIのインストール」を参照してください。
  - を設定するには AWS CLI、「 の設定 AWS CLI」を参照してください。
  - バージョン 1 AWS CLI の最新バージョンにアップグレードするには、次のコマンドを実行します。

```
pip install awscli --upgrade --user
```

> **Note**
>
> Windows で AWS CLI バージョン 1 の MSI インストールを使用する場合は、次の点に注意してください。
>
> - AWS CLI バージョン 1 のインストールで botocore のインストールに失敗する場合は、Python と pip のインストールを試してください。

- 新しい AWS CLI バージョン 1 にアップグレードするには、MSI のインストールプロセスを繰り返す必要があります。

- ユーザーが Amazon Elastic Container Registry (Amazon ECR) のリソースにアクセスできるようにするには、次のアクセス権限を付与する必要があります。
  - Amazon ECR では、レジストリに対して認証し、Amazon ECR リポジトリからイメージをプッシュまたはプルする前に、 AWS Identity and Access Management (IAM) ポリシーを通じてアクセス`ecr:GetAuthorizationToken`許可を付与する必要があります。詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR Repository Policy Examples」(Amazon ECR リポジトリポリシーの例) および「1 つの Amazon ECR リポジトリにアクセスする」を参照してください。

1. Docker イメージをダウンロードし、コンテナを設定します。Docker Hub または Amazon Elastic Container Registry (Amazon ECR) から作成済みのイメージをダウンロードして、Windows、macOS、および Linux (x86_64) プラットフォームで実行できます。

   Amazon ECR から Docker イメージをダウンロードするには、「the section called “Amazon ECR から AWS IoT Greengrass コンテナイメージを取得する”」のすべてのステップを実行します。次に、このトピックに戻って設定を続行します。

2. Linux ユーザーのみ: IDT を実行するユーザーに Docker コマンドを実行するアクセス許可があることを確認してください。詳細については、Docker ドキュメントの「Docker を非ルートユーザーとして管理する」を参照してください。

3. AWS IoT Greengrass コンテナを実行するには、オペレーティングシステムの コマンドを使用します。

   Linux

   ```
   docker run --rm --init -it -d --name aws-iot-greengrass \
   -p 8883:8883 \
   -v <host-path-to-kernel-config-file>:<container-path> \
   <image-repository>:<tag>
   ```

   - *<host-path-to-kernel-config-file>* はホストのカーネル設定ファイルへのパスに置き換え、*<container-path>* はコンテナ内でボリュームがマウントされているパスに置き換えます。

ホストのカーネル設定ファイルは、通常、/proc/config.gz または /boot/config-*<kernel-release-date>* にあります。uname -r を実行して *<kernel-release-date>* 値を検索できます。

例: 設定ファイルを /boot/config-*<kernel-release-date>* からマウントするには

```
-v /boot/config-4.15.0-74-generic:/boot/config-4.15.0-74-generic \
```

例: 設定ファイルを proc/config.gz からマウントするには

```
-v /proc/config.gz:/proc/config.gz \
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

  Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

  ```
  aws ecr list-images --region us-west-2 --registry-id 216483018798 --
  repository-name aws-iot-greengrass
  ```

macOS

```
docker run --rm --init -it -d --name aws-iot-greengrass \
-p 8883:8883 \
<image-repository>:<tag>
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

```
aws ecr list-images --region us-west-2 --registry-id 216483018798 --
repository-name aws-iot-greengrass
```

Windows

```
docker run --rm --init -it -d --name aws-iot-greengrass \
-p 8883:8883 \
<image-repository>:<tag>
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

  Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

  ```
  aws ecr list-images --region us-west-2 --registry-id 216483018798 --
  repository-name aws-iot-greengrass
  ```

> **Important**
>
> IDT でテストする場合は、イメージの実行に使用される `--entrypoint /greengrass-entrypoint.sh \`引数を汎用に含めないでください AWS IoT Greengrass 。

4. 次のステップ: AWS 認証情報と `device.json` ファイルを設定します。

が提供する dockerfile を設定する AWS IoT Greengrass

IDT テストを実行するように Dockerfile から構築された Docker AWS IoT Greengrass イメージを設定するには、次の手順に従います。

1. the section called “AWS IoT Greengrass Docker ソフトウェア” からホストコンピュータに Dockerfile パッケージをダウンロードして解凍します。
2. `README.md` を開きます。次の 3 つのステップでは、このファイルのセクションを参照します。
3. 「前提条件」セクションの要件を満たしていることを確認します。
4. Linux ユーザーのみ: 「シンボリックリンク保護とハードリンク保護を有効にする」ステップと「IPv4 ネットワーク転送を有効にする」ステップを完了します。
5. Docker イメージを構築するには、ステップ 1 のすべての手順を完了させます。Docker AWS IoT Greengrass イメージを構築します。次に、このトピックに戻って設定を続行します。
6. AWS IoT Greengrass コンテナを実行するには、オペレーティングシステムの コマンドを使用します。

   Linux

   ```
   docker run --rm --init -it -d --name aws-iot-greengrass \
   -p 8883:8883 \
   -v <host-path-to-kernel-config-file>:<container-path> \
   <image-repository>:<tag>
   ```

   - *`<host-path-to-kernel-config-file>`* はホストのカーネル設定ファイルへのパスに置き換え、*`<container-path>`* はコンテナ内でボリュームがマウントされているパスに置き換えます。

     ホストのカーネル設定ファイルは、通常、`/proc/config.gz` または `/boot/config-`*`<kernel-release-date>`* にあります。`uname -r` を実行して *`<kernel-release-date>`* 値を検索できます。

     例: 設定ファイルを `/boot/config-`*`<kernel-release-date>`* からマウントするには

     ```
     -v /boot/config-4.15.0-74-generic:/boot/config-4.15.0-74-generic \
     ```

     例: 設定ファイルを `proc/config.gz` からマウントするには

```
-v /proc/config.gz:/proc/config.gz \
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

  Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

  ```
  aws ecr list-images --region us-west-2 --registry-id 216483018798 --
  repository-name aws-iot-greengrass
  ```

macOS

```
docker run --rm --init -it -d --name aws-iot-greengrass \
-p 8883:8883 \
<image-repository>:<tag>
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

  Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

  ```
  aws ecr list-images --region us-west-2 --registry-id 216483018798 --
  repository-name aws-iot-greengrass
  ```

Windows

```
docker run --rm --init -it -d --name aws-iot-greengrass \
-p 8883:8883 \
<image-repository>:<tag>
```

- コマンドの *<image-repository>*:*<tag>* は、リポジトリの名前とターゲットイメージのタグに置き換えます。

  例： AWS IoT Greengrass Core ソフトウェアの最新バージョンを参照するには

  ```
  216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
  ```

  Docker AWS IoT Greengrass イメージのリストを取得するには、次のコマンドを実行します。

  ```
  aws ecr list-images --region us-west-2 --registry-id 216483018798 --
  repository-name aws-iot-greengrass
  ```

> **Important**
>
> IDT でテストする場合は、イメージの実行に使用される `--entrypoint /greengrass-entrypoint.sh \`引数を汎用に含めないでください AWS IoT Greengrass 。

7. 次のステップ: AWS 認証情報と `device.json` ファイルを設定します。

IDT for の Docker コンテナ設定のトラブルシューティング AWS IoT Greengrass

以下の情報は、IDT 用の Docker コンテナを AWS IoT Greengrass テスト用に実行する際の問題のトラブルシューティングに役立ちます。

警告: 設定ファイルの読み込みエラー: /home/user/.docker/config.json - stat /home/<user>/.docker/config.json: アクセス許可が拒否されました

Linux で `docker` コマンドの実行中にこのエラーが発生した場合は、次のコマンドを実行します。次のコマンドの *<user>* は、IDT を実行するユーザーに置き換えます。

```
sudo chown <user>:<user> /home/<user>/.docker -R
sudo chmod g+rwx /home/<user>/.docker -R
```

オプション: ML 認定のためのデバイスの設定

IDT for AWS IoT Greengrass は、機械学習 (ML) 認定テストを提供し、デバイスがクラウドでトレーニングされたモデルを使用してローカルで ML 推論を実行できることを検証します。

ML 認定テストを実行するには、まず、「the section called “IDT テストを実行するようにデバイスを設定する”」の説明に従ってデバイスを設定する必要があります。次に、このトピックの手順に従って、実行する ML フレームワークの依存関係をインストールします。

ML 認定のためのテストを実行するには、IDT v3.1.0 以降が必要です。

ML フレームワークの依存関係のインストール

ML フレームワークの依存関係はすべて、`/usr/local/lib/python3.x/site-packages` ディレクトリ以下にインストールする必要があります。確実に正しいディレクトリにインストールするには、依存関係をインストールするときに sudo root アクセス許可を使用することをお勧めします。仮想環境は、認定テストではサポートされていません。

> **Note**
>
> コンテナ化を使用して実行される Lambda 関数を (Greengrass コンテナモードで) テストしている場合、`/usr/local/lib/python3.x` で Python ライブラリのシンボリックリンクを作成することはサポートされていません。エラーを回避するには、依存関係を正しいディレクトリにインストールする必要があります。

ターゲットフレームワークの依存関係をインストールするには、以下のステップに従います。

- MXNet 依存関係のインストール
- the section called “TensorFlow の依存関係のインストール”
- DLR 依存関係のインストール

Apache MXNet 依存関係のインストール

このフレームワークの IDT 認定テストには、次の依存関係があります。

- Python 3.6 or Python 3.7。

> **Note**
>
> Python 3.6を使用している場合は、Python 3.7 からPython 3.6 バイナリへのシンボリックリンクを作成する必要があります。これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。以下に例を示します。
>
> ```
> sudo ln -s path-to-python-3.6/python3.6 path-to-python-3.7/python3.7
> ```

- Apache MXNet v1.2.1 以降。
- NumPy。MXNet バージョンと互換性があるバージョンのものをしようする必要があります。

MXNet のインストール

MXNet のドキュメントの指示に従って、MXNet をインストールします。

> **Note**
>
> Python 2.x と Python 3.x の両方がデバイスにインストールされている場合は、実行するコマンドで Python 3.x を使用して依存関係をインストールします。

MXNet のインストールの検証

次のいずれかのオプションを選択して、MXNet のインストールを検証します。

オプション 1: デバイスに SSH 接続してスクリプトを実行する

1. デバイスに SSH 接続します。
2. 次のスクリプトを実行して、依存関係が正しくインストールされていることを確認します。

```
sudo python3.7 -c "import mxnet; print(mxnet.__version__)"
```

```
sudo python3.7 -c "import numpy; print(numpy.__version__)"
```

出力ではバージョン番号が表示されます。スクリプトがエラーなく終了する必要があります。

オプション 2: IDT 依存関係テストを実行する

1. `device.json` が ML 認定用に設定されていることを確認します。詳細については、「the section called “ML 認定のための device.json の設定”」を参照してください。
2. フレームワークの依存関係テストを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id mldependencies --test-id mxnet_dependency_check
```

テストサマリーに、`mldependencies` の結果として `PASSED` が表示されます。

TensorFlow の依存関係のインストール

このフレームワークの IDT 認定テストには、次の依存関係があります。

- Python 3.6 or Python 3.7。

> **Note**
>
> Python 3.6を使用している場合は、Python 3.7 からPython 3.6 バイナリへのシンボリックリンクを作成する必要があります。これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。以下に例を示します。
>
> ```
> sudo ln -s path-to-python-3.6/python3.6 path-to-python-3.7/python3.7
> ```

- TensorFlow 1.x。

TensorFlow のインストール

TensorFlow のドキュメントの指示に従い、pip またはソースを使用して、TensorFlow 1.x をインストールします。

> **Note**
>
> Python 2.x と Python 3.x の両方がデバイスにインストールされている場合は、実行するコマンドで Python 3.x を使用して依存関係をインストールします。

TensorFlow のインストールの検証

次のいずれかのオプションを選択して、TensorFlow のインストールを検証します。

オプション1：デバイスに SSH 接続してスクリプトを実行する

1. デバイスに SSH 接続します。
2. 次のスクリプトを実行して、依存関係が正しくインストールされていることを確認します。

```
sudo python3.7 -c "import tensorflow; print(tensorflow.__version__)"
```

出力ではバージョン番号が表示されます。スクリプトがエラーなく終了する必要があります。

オプション 2: IDT 依存関係テストを実行する

1. `device.json` が ML 認定用に設定されていることを確認します。詳細については、「the section called “ML 認定のための device.json の設定”」を参照してください。
2. フレームワークの依存関係テストを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id mldependencies --test-
id tensorflow_dependency_check
```

テストサマリーに、`mldependencies` の結果として `PASSED` が表示されます。

Amazon SageMaker AI Neo Deep Learning Runtime (DLR) の依存関係をインストールする

このフレームワークの IDT 認定テストには、次の依存関係があります。

- Python 3.6 or Python 3.7。

> **Note**
>
> Python 3.6を使用している場合は、Python 3.7 からPython 3.6 バイナリへのシンボリックリンクを作成する必要があります。これにより、 AWS IoT Greengrassの Python 要件を満たすようにデバイスが設定されます。以下に例を示します。

```
sudo ln -s path-to-python-3.6/python3.6 path-to-python-3.7/python3.7
```

- SageMaker AI Neo DLR。
- numpy。

DLR テストの依存関係をインストールした後、モデルをコンパイルする必要があります。

DLR のインストール

DLR のドキュメントの指示に従って、Neo DLR をインストールします。

> **Note**
> Python 2.x と Python 3.x の両方がデバイスにインストールされている場合は、実行するコマンドで Python 3.x を使用して依存関係をインストールします。

DLR のインストールの検証

次のいずれかのオプションを選択して、DLR のインストールを検証します。

オプション 1: デバイスに SSH 接続してスクリプトを実行する

1. デバイスに SSH 接続します。
2. 次のスクリプトを実行して、依存関係が正しくインストールされていることを確認します。

```
sudo python3.7 -c "import dlr; print(dlr.__version__)"
```

```
sudo python3.7 -c "import numpy; print(numpy.__version__)"
```

出力ではバージョン番号が表示されます。スクリプトがエラーなく終了する必要があります。

オプション 2: IDT 依存関係テストを実行する

1. `device.json` が ML 認定用に設定されていることを確認します。詳細については、「the section called “ML 認定のための device.json の設定”」を参照してください。
2. フレームワークの依存関係テストを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id mldependencies --test-
id dlr_dependency_check
```

テストサマリーに、`mldependencies` の結果として `PASSED` が表示されます。

DLR モデルのコンパイル

DLR モデルは、ML 認定テストに使用する前にコンパイルする必要があります。手順については、次のいずれかのオプションを選択します。

オプション 1: Amazon SageMaker AI を使用してモデルをコンパイルする

SageMaker AI を使用して IDT が提供する ML モデルをコンパイルするには、次の手順に従います。このモデルは Apache MXnet で事前にトレーニングされています。

1. デバイスタイプが SageMaker AI でサポートされていることを確認します。詳細については、Amazon SageMaker AI API リファレンスのターゲットデバイスオプションを参照してください。デバイスタイプが現在 SageMaker AI でサポートされていない場合は、「」の手順に従ってくださいthe section called “オプション 2: TVM を使用して DLR をコンパイルする”。

   > **Note**
   >
   > SageMaker AI でコンパイルされたモデルで DLR テストを実行するには、4 分または 5 分かかる場合があります。この間、IDT を停止しないでください。

2. 事前トレーニング済みでコンパイルされていない DLR 用 MxNet モデルを含む tarball ファイルをダウンロードします。

   - dlr-noncompiled-model-1.0.tar.gz

3. tarball を解凍します。このコマンドを実行すると、以下のディレクトリ構造が生成されます。

4. `synset.txt` を `resnet18` ディレクトリ以外の場所に移動します。新しい場所を書き留めておきます。このファイルは、コンパイルされたモデルのディレクトリに、後でコピーします。

5. `resnet18` ディレクトリの内容を圧縮します。

```
tar cvfz model.tar.gz resnet18v1-symbol.json resnet18v1-0000.params
```

6. 圧縮ファイルを の Amazon S3 バケットにアップロードし AWS アカウント、「モデルをコンパイルする (コンソール)」の手順に従ってコンパイルジョブを作成します。

   a. [入力設定] では、次の値を使用します。

      - [データ入力設定] に `{"data": [1, 3, 224, 224]}` と入力します。
      - [機械学習フレームワーク] で、`MXNet` を選択します。

   b. [出力設定] では、次の値を使用します。

      - [S3 Output location] (S3 出力場所) には、コンパイル済みモデルを保存する Amazon S3 バケットまたはフォルダへのパスを入力します。
      - [ターゲットデバイス] で、デバイスタイプを選択します。

7. 指定した出力場所からコンパイル済みモデルをダウンロードし、ファイルを解凍します。
8. コンパイル済みモデルのディレクトリに `synset.txt` をコピーします。
9. コンパイル済みモデルのディレクトリの名前を `resnet18` に変更します。

   コンパイル済みモデルのディレクトリでは、次のディレクトリ構造が必要です。

オプション 2: TVM を使用して DLR をコンパイルする

TVM を使用して、IDT によって提供される ML モデルをコンパイルするには、以下のステップに従います。このモデルは Apache MXNet で事前にトレーニングされているため、モデルをコンパイルするコンピューターまたはデバイスに MXNet をインストールする必要があります。MXNet をインストールするには、MXNet のドキュメントの指示に従います。

> **Note**
>
> ターゲットデバイス上でモデルをコンパイルすることをお勧めします。この方法はオプションですが、互換性を確保し、潜在的な問題を軽減するために役立ちます。

1. 事前トレーニング済みでコンパイルされていない DLR 用 MxNet モデルを含む tarball ファイルをダウンロードします。

   - dlr-noncompiled-model-1.0.tar.gz

2. tarball を解凍します。このコマンドを実行すると、以下のディレクトリ構造が生成されます。

3. TVM のドキュメントの指示に従って、お使いのプラットフォームのソースから TVM のビルドとインストールを行います。

4. TVM がビルドされたら、resnet18 モデルの TVM コンパイルを実行します。以下のステップは、TVM のドキュメントの「Quick Start Tutorial for Compiling Deep Learning Models」に基づいています。

   a. クローン作成された TVM リポジトリから `relay_quick_start.py` ファイルを開きます。

   b. リレーでニューラルネットワークを定義するコードを更新します。次のいずれかのオプションを使用できます。

      - オプション 1: `mxnet.gluon.model_zoo.vision.get_model` を使用してリレーモジュールとパラメータを取得します。

```
from mxnet.gluon.model_zoo.vision import get_model
block = get_model('resnet18_v1', pretrained=True)
mod, params = relay.frontend.from_mxnet(block, {"data": data_shape})
```

- オプション 2: ステップ 1 でダウンロードしたコンパイルされていないモデルから、以下のファイルをrelay_quick_start.py ファイルと同じディレクトリにコピーします。これらのファイルには、リレーモジュールとパラメータが含まれています。
  - resnet18v1-symbol.json
  - resnet18v1-0000.params

c. コンパイル済みモジュールを保存してロードするコードを更新し、次のコードを使用します。

```
from tvm.contrib import util
path_lib = "deploy_lib.so"
#  Export the model library based on your device architecture
lib.export_library("deploy_lib.so", cc="aarch64-linux-gnu-g++")
with open("deploy_graph.json", "w") as fo:
    fo.write(graph)
with open("deploy_param.params", "wb") as fo:
    fo.write(relay.save_param_dict(params))
```

d. モデルをビルドします。

```
python3 tutorials/relay_quick_start.py --build-dir ./model
```

このコマンドを実行すると、以下のファイルが生成されます。

- deploy_graph.json
- deploy_lib.so
- deploy_param.params

5. 生成されたモデルファイルを resnet18 という名前のディレクトリにコピーします。これはコンパイル済みモデルのディレクトリです。
6. コンパイル済みモデルのディレクトリをホストコンピュータにコピーします。次に、ステップ 1 でダウンロードしたコンパイルされていないモデルから、コンパイル済みモデルのディレクトリに synset.txt をコピーします。

   コンパイル済みモデルのディレクトリでは、次のディレクトリ構造が必要です。

次に、AWS 認証情報と `device.json` ファイルを設定します。

認定スイートを実行するように IDT AWS IoT Greengrass 設定を構成する

テストを実行する前に、ホストコンピュータの AWS 認証情報とデバイスの設定を構成する必要があります。

AWS 認証情報を設定する

IAM ユーザー認証情報を `<device-tester-extract-location> /configs/config.json` ファイルで設定する必要があります。で作成した AWS IoT Greengrass ユーザーの IDT の認証情報を使用しますthe section called “の作成と設定 AWS アカウント”。以下のいずれかの方法で認証情報を指定できます。

- 認証情報ファイル
- 環境変数

AWS 認証情報ファイルを使用して認証情報を設定する

IDT では、 AWS CLIと同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

- macOS、Linux: `~/.aws/credentials`
- Windows: `C:\Users\UserName\.aws\credentials`

次の形式で AWS 認証情報を `credentials` ファイルに追加します。

```
[default]
```

```
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

credentials ファイルの AWS 認証情報を使用する AWS IoT Greengrass ように IDT for を設定するには、次のようにconfig.jsonファイルを編集します。

```
{
 "awsRegion": "us-west-2",
 "auth": {
  "method": "file",
  "credentials": {
   "profile": "default"
  }
 }
}
```

> **Note**
>
> default AWS プロファイルを使用しない場合は、必ず config.json ファイルでプロファイル名を変更してください。詳細については、「名前付きプロファイル」を参照してください。

環境変数を使用して AWS 認証情報を設定する

環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッションを閉じると、これらは保存されません。IDT for AWS IoT Greengrass は、AWS_ACCESS_KEY_IDおよび AWS_SECRET_ACCESS_KEY環境変数を使用して AWS 認証情報を保存できます。

これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

Windows でこれらの変数を設定するには、set を使用します。

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

環境変数を使用するように IDT を設定するには、`config.json` ファイルの `auth` セクションを編集します。以下がその例です。

```
{
 "awsRegion": "us-west-2",
 "auth": {
  "method": "environment"
 }
}
```

device.json の設定

AWS 認証情報に加えて、IDT for には、テストが実行されるデバイスに関する情報 (IP アドレス、ログイン情報、オペレーティングシステム、CPU アーキテクチャなど) AWS IoT Greengrass が必要です。

これらの情報を指定するには、 *<device_tester_extract_location>*`/configs/device.json` にある `device.json` テンプレートを使用する必要があります。

Physical device

```
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "os",
        "value": "linux | ubuntu | openwrt"
      },
      {
        "name": "arch",
        "value": "x86_64 | armv6l | armv7l | aarch64"
      },
      {
        "name": "container",
        "value": "yes | no"
      },
      {
        "name": "docker",
        "value": "yes | no"
      },
```

```
      {
        "name": "streamManagement",
        "value": "yes | no"
      },
      {
        "name": "hsi",
        "value": "yes | no"
      },
      {
        "name": "ml",
        "value": "mxnet | tensorflow | dlr | mxnet,dlr,tensorflow | no"
      },
      *********** Remove the section below if the device is not qualifying for ML
**************,
      {
        "name": "mlLambdaContainerizationMode",
        "value": "container | process | both"
      },
      {
        "name": "processor",
        "value": "cpu | gpu"
      },

*****************************************************************************************
    ],
    *********** Remove the section below if the device is not qualifying for HSI
***************
    "hsm": {
      "p11Provider": "/path/to/pkcs11ProviderLibrary",
      "slotLabel": "<slot_label>",
      "slotUserPin": "<slot_pin>",
      "privateKeyLabel": "<key_label>",
      "openSSLEngine": "/path/to/openssl/engine"
    },

*****************************************************************************************
    *********** Remove the section below if the device is not qualifying for ML
****************
    "machineLearning": {
      "dlrModelPath": "/path/to/compiled/dlr/model",
      "environmentVariables": [
        {
          "key": "<environment-variable-name>",
          "value": "<Path:$PATH>"
```

```
        }
      ],
      "deviceResources": [
        {
          "name": "<resource-name>",
          "path": "<resource-path>",
          "type": "device | volume"
        }
      ]
    },

 ******************************************************************************************
    "kernelConfigLocation": "",
    "greengrassLocation": "",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": 22,
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

Docker container

```
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "os",
        "value": "linux | ubuntu | openwrt"
      },
      {
        "name": "arch",
        "value": "x86_64"
      },
      {
        "name": "container",
        "value": "no"
      },
      {
        "name": "docker",
        "value": "no"
      },
      {
        "name": "streamManagement",
        "value": "yes | no"
      },
      {
        "name": "hsi",
        "value": "no"
      },
      {
        "name": "ml",
        "value": "mxnet | tensorflow | dlr | mxnet,dlr,tensorflow | no"
      },
      *********** Remove the section below if the device is not qualifying for ML
 **************,
      {
        "name": "mlLambdaContainerizationMode",
        "value": "process"
      },
      {
        "name": "processor",
        "value": "cpu | gpu"
```

```
      },

*****************************************************************************************
    ],
    *********** Remove the section below if the device is not qualifying for ML
****************
    "machineLearning": {
      "dlrModelPath": "/path/to/compiled/dlr/model",
      "environmentVariables": [
        {
          "key": "<environment-variable-name>",
          "value": "<Path:$PATH>"
        }
      ],
      "deviceResources": [
        {
          "name": "<resource-name>",
          "path": "<resource-path>",
          "type": "device | volume"
        }
      ]
    },

*****************************************************************************************
    "kernelConfigLocation": "",
    "greengrassLocation": "",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "docker",
          "containerId": "<container-name | container-id>",
          "containerUser": "<user>"
        }
      }
    ]
  }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`sku`

テスト対象デバイスを一意に識別する英数字の値。SKU は、適格性が確認されたボードの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog にボードを一覧表示する場合、ここで指定する SKU は、一覧表示プロセスで使用する SKU と一致する必要があります。

`features`

デバイスでサポートされている機能を含む配列。すべての機能が必要です。

`os` および `arch`

サポート対象のオペレーティングシステム (OS) とアーキテクチャの組み合わせ。

- `linux`, `x86_64`
- `linux`, `armv6l`
- `linux`, `armv7l`
- `linux`, `aarch64`
- `ubuntu`, `x86_64`
- `openwrt`, `armv7l`
- `openwrt`, `aarch64`

> **Note**
>
> IDT を使用して Docker コンテナで AWS IoT Greengrass の実行をテストする場合、x86_64 Docker アーキテクチャのみがサポートされます。

`container`

デバイスが Greengrass コアのコンテナモードで Lambda 関数を実行するためのすべてのソフトウェア要件とハードウェア要件を満たしているかどうかを検証します。

有効な値は `yes` または `no` です。

`docker`

Greengrass Docker アプリケーションデプロイコネクタを使用してコンテナを実行するために必要なすべての技術的依存関係をデバイスが満たしていることを検証します。

有効な値は `yes` または `no` です。

`streamManagement`

デバイスが AWS IoT Greengrass ストリームマネージャーを実行するために必要な技術的な依存関係をすべて満たしていることを検証します。

有効な値は `yes` または `no` です。

`hsi`

提供された HSI 共有ライブラリがハードウェアセキュリティモジュール (HSM) とやり取りでき、必要な PKCS#11 API を正しく実装することを検証します。HSM および共有ライブラリは、CSR に署名し、TLS オペレーションを実行して、正しいキー長と公開キーアルゴリズムを提供できる必要があります。

有効な値は `yes` または `no` です。

`ml`

ML 推論をローカルで実行するために必要なすべての技術的依存関係をデバイスが満たしていることを検証します。

有効な値は、`mxnet`、`tensorflow`、`dlr`、および `no` の任意の組み合わせです (例: `mxnet`、`mxnet,tensorflow`、`mxnet,tensorflow,dlr`、または `no`)。

`mlLambdaContainerizationMode`

コンテナモードの Greengrass デバイスで ML 推論を実行するための技術的依存関係要件をデバイスがすべて満たしていることを検証します。

有効な値は `container`、`process`、または `both` です。

`processor`

指定したプロセッサタイプのハードウェア要件をデバイスがすべて満たしていることを検証します。

有効な値は `cpu` または `gpu` です。

> **Note**
>
> `container`、`docker`、`streamManager`、`hsi`、または `ml` 機能を使用しない場合は、対応する `value` を `no` に設定できます。
> Docker は、`streamManagement` と `ml` の機能認定のみをサポートしています。

`machineLearning`

オプション。ML 認定テストの設定情報。詳細については、「the section called “ML 認定のための device.json の設定”」を参照してください。

`hsm`

オプション。 AWS IoT Greengrass ハードウェアセキュリティモジュール (HSM) でテストするための設定情報。それ以外の場合は、`hsm` プロパティを省略する必要があります。詳細については、「ハードウェアセキュリティ統合」を参照してください。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`hsm.p11Provider`

PKCS#11 実装の libdl-loadable ライブラリへの絶対パス。

`hsm.slotLabel`

ハードウェアモジュールを識別するために使用されるスロットラベル。

`hsm.slotUserPin`

モジュールへの AWS IoT Greengrass コアの認証に使用されるユーザー PIN。

`hsm.privateKeyLabel`

ハードウェアモジュールでキーを識別するために使用されるラベル。

`hsm.openSSLEngine`

OpenSSL での PKCS#11 のサポートを有効にするための、OpenSSL エンジンの `.so` ファイルへの絶対パス。 AWS IoT Greengrass OTA 更新エージェントによって使用されます。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。現在、サポートされている値は、物理デバイス用の `ssh` と Docker コンテナ用の `docker` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.containerId`

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、`connectivity.protocol` が `docker` に設定されている場合にのみ適用されます。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.password`

テスト中のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.privKeyPath`

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.user`

テスト対象デバイスにサインインするためのユーザー名。

`connectivity.auth.credentials.privKeyPath`

テスト対象デバイスにサインインするためのプライベートキーへの完全パス。

`connectivity.port`

オプション。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`greengrassLocation`

デバイス上の AWS IoT Greengrass Core ソフトウェアの場所。

物理デバイスの場合、この値は の既存のインストールを使用する場合にのみ使用されます AWS IoT Greengrass。この属性を使用して、デバイスにインストール済みのバージョンの AWS IoT Greengrass Core ソフトウェアを使用するよう IDT に指示します。

が提供する Docker イメージまたは Dockerfile から Docker コンテナでテストを実行する場合は AWS IoT Greengrass、この値を に設定します`/greengrass`。

`kernelConfigLocation`

オプション。カーネル設定ファイルへのパス。 AWS IoT Device Tester はこのファイルを使用して、デバイスに必要なカーネル機能が有効になっているかどうかを確認します。指定しない場合、IDT は次のパスを使用してカーネル設定ファイル `/proc/config.gz`と を検索します`/`

`boot/config-<kernel-version>`。 AWS IoT Device Tester は、検出した最初のパスを使用します。

ML 認定のための device.json の設定

このセクションでは、ML 認定に適用されるデバイス設定ファイルのオプションプロパティについて説明します。ML 認定のテストを実行する場合は、ユースケースに適したプロパティを定義する必要があります。

`device-ml.json` テンプレートを使用して、デバイスの構成設定を定義できます。このテンプレートには、オプションの ML プロパティが含まれています。また、`device.json` を使用して ML 修飾プロパティを追加することもできます。これらのファイルは `<device-tester-extract-location>/configs` にあり、これらには ML 認定プロパティが含まれています。`device-ml.json` を使用する場合は、IDT テストを実行する前に、ファイルの名前を `device.json` に変更する必要があります。

ML 認定に適用されないデバイス設定プロパティについては、「the section called “device.json の設定”」を参照してください。

`features` 配列内の `ml`

ボードがサポートする ML フレームワーク。このプロパティには IDT v3.1.0 以降が必要です。

- 対象のボードでサポートされるフレームワークが 1 つだけの場合は、そのフレームワークを指定します。例:

```
{
    "name": "ml",
    "value": "mxnet"
}
```

- 対象のボードで複数のフレームワークがサポートされている場合は、フレームワークをカンマで区切って指定します。例:

```
{
    "name": "ml",
    "value": "mxnet,tensorflow"
}
```

`features` 配列内の `mlLambdaContainerizationMode`

テストに使用するコンテナ化モード。このプロパティには IDT v3.1.0 以降が必要です。

- コンテナ化されていない Lambda 関数を使用して ML 推論コードを実行するには、`process` を選択します。このオプションには AWS IoT Greengrass v1.10.x 以降が必要です。
- コンテナ化された Lambda 関数を使用して ML 推論コードを実行するには、`container` を選択します。
- 両方のモードで ML 推論コードを実行するには、`both` を選択します。このオプションには AWS IoT Greengrass v1.10.x 以降が必要です。

`features` 配列内の `processor`

対象のボードがサポートするハードウェアアクセラレーターを示します。このプロパティには IDT v3.1.0 以降が必要です。

- 対象のボードがプロセッサとして CPU を使用する場合は `cpu` を選択します。
- 対象のボードがプロセッサとして GPU を使用する場合は `gpu` を選択します。

`machineLearning`

オプション。ML 認定テストの設定情報。このプロパティには IDT v3.1.0 以降が必要です。

`dlrModelPath`

`dlr` フレームワークを使用するために必要です。DLR コンパイル済みモデルディレクトリへの絶対パス。`resnet18` を指定する必要があります。詳細については、「the section called “DLR モデルのコンパイル”」を参照してください。

> **Note**
>
> macOS でのパスの例は次のとおりです: `/Users/<user>/Downloads/resnet18`

`environmentVariables`

ML 推論テストに設定を動的に渡すことができるキーと値のペアの配列。CPU デバイスの場合はオプションです。このセクションを使用して、デバイスタイプに必要なフレームワーク固有の環境変数を追加できます。これらの要件の詳細については、フレームワークまたはデバイスの公式ウェブサイトを参照してください。たとえば、一部のデバイスで MXNet 推論テストを実行するには、次のような環境変数が必要になります。

```
"environmentVariables": [
    ...
    {
        "key": "PYTHONPATH",
        "value": "$MXNET_HOME/python:$PYTHONPATH"
    },
    {
        "key": "MXNET_HOME",
        "value": "$HOME/mxnet/"
    },
    ...
]
```

> **Note**
>
> `value` フィールドは、MXNet のインストールによって異なる場合があります。

GPU デバイスでコンテナ化によって実行される Lambda 関数をテストしている場合は、GPU ライブラリの環境変数を追加します。これにより、GPU が計算を実行できるようになります。異なる GPU ライブラリを使用するには、ライブラリまたはデバイスの公式ドキュメントを参照してください。

> **Note**
>
> `mlLambdaContainerizationMode` 機能が `container` または `both` に設定されている場合は、次のキーを設定します。

```
"environmentVariables": [
    {
        "key": "PATH",
        "value": "<path/to/software/bin>:$PATH"
    },
    {
        "key": "LD_LIBRARY_PATH",
        "value": "<path/to/ld/lib>"
    },
    ...
]
```

`deviceResources`

GPU デバイスの場合に必要です。Lambda 関数からアクセス可能なローカルリソースが含まれています。このセクションを使用して、ローカルデバイスとボリュームリソースを追加します。

- デバイスリソースの場合は、`"type": "device"` を指定します。GPU デバイスの場合、デバイスリソースは、`/dev` 以下にある GPU 関連デバイスファイルである必要があります。

> **Note**
>
> `/dev/shm` ディレクトリは例外です。これは、ボリュームリソースとしてのみ設定できます。

- ボリュームリソースの場合は、`"type": "volume"` を指定します。

AWS IoT Greengrass 認定スイートを実行する

必要な設定を定義したら、テストを開始できます。完全なテストスイートのランタイムはハードウェアによって異なります。参考までに、Raspberry Pi 3B で完全なテストスイートを完了するには約 30 分かかります。

以下の `run-suite` コマンドの例では、デバイスプールに対して適格性確認テストを実行する方法を示します。デバイスプールは、同一のデバイスのセットです。

IDT v3.0.0 and later

指定したテストスイート内のすべてのテストグループを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1.0.0 --pool-id <pool-id>
```

`list-suites` コマンドを使用して、`tests` フォルダ内にあるテストスイートを一覧表示します。

テストスイートで特定のテストグループを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1.0.0 --group-id <group-id> --pool-id <pool-id>
```

テストスイートのテストグループを一覧表示するには、`list-groups` コマンドを使用します。

テストグループ内の特定のテストケースを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id <group-id> --test-id <test-id>
```

テストグループ内の複数のテストケースを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id <group-id> --test-id <test-id1>,<test-id2>
```

テストグループ内のテストケースを一覧表示します。

```
devicetester_[linux | mac | win_x86-64] list-test-cases --group-id <group-id>
```

`run-suite` コマンドのオプションは省略可能です。たとえば、`device.json` ファイルに定義されているデバイスプールが 1 つしかない場合は、`pool-id` を省略できます。または、`tests` フォルダ内の最新のテストスイートのバージョンを実行する場合は、`suite-id` を省略できます。

> **Note**
>
> IDT は、新しいテストスイートのバージョンをオンラインで入手できるかどうかを尋ねるプロンプトを表示します。詳細については、「the section called “デフォルトの更新動作を設定する”」を参照してください。

`run-suite` およびその他の IDT コマンドの詳細については、「the section called “IDT コマンド”」を参照してください。

IDT v2.3.0 and earlier

指定したスイート内のすべてのテストグループを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1 --pool-id <pool-id>
```

特定のテストグループを実行します。

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1 --group-
id <group-id> --pool-id <pool-id>
```

単一のデバイスプールで単一のテストスイートを実行している場合は、suite-id および pool-id は省略可能です。つまり、device.json ファイルに定義されているデバイスプールは 1 つだけです。

Greengrass の依存関係を確認する

関連するテストグループを実行する前に、すべての Greengrass 依存関係がインストールされていることを確認するため、依存関係チェッカーテストグループを実行することをお勧めします。例:

- コア資格テストグループを実行する前に ggcdependencies を実行します。
- コンテナ固有のテストグループを実行する前に、containerdependencies を実行します。
- Docker 固有のテストグループを実行する前に dockerdependencies を実行します。
- ストリームマネージャ固有のテストグループを実行する前に ggcstreammanagementdependencies を実行します。

デフォルトの更新動作を設定する

テストランを開始すると、IDT はオンラインで新しいテストスイートのバージョンを確認します。使用可能な場合、IDT は利用可能な最新バージョンに更新するよう求めるプロンプトを表示します。upgrade-test-suite (または u) フラグを設定して、デフォルトの更新動作を制御できます。次の値を指定できます:

- y。IDT は、利用可能な最新バージョンをダウンロードして使用します。
- n (デフォルト)。IDT は、suite-id オプションで指定されたバージョンを使用します。suite-id が指定されていない場合、IDT は tests フォルダにある最新バージョンを使用します。

upgrade-test-suite フラグを含めない場合、IDT は更新が利用可能になったときにプロンプトを表示し、入力 (y または n) まで 30 秒待ちます。何も入力されていない場合、デフォルトは n に設定され、テストの実行が続行されます。

次の例は、この機能の一般的ユースケースを示しています。

テストグループで使用可能な最新のテストを自動的に使用します。

```
devicetester_linux run-suite -u y --group-id mqtt --pool-id DevicePool1
```

特定のテストスイートのバージョンでテストを実行します。

```
devicetester_linux run-suite -u n --suite-id GGQ_1.0.0 --group-id mqtt --pool-id
 DevicePool1
```

実行時に更新を要求します。

```
devicetester_linux run-suite --pool-id DevicePool1
```

AWS IoT Greengrass コマンドの IDT

IDT コマンドは、*<device-tester-extract-location>*/bin ディレクトリにあります。これらのコマンドは次のオペレーションに使用します。

IDT v3.0.0 and later

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のテストスイート内のグループを一覧表示します。

`list-suites`

使用可能なテストスイートを一覧表示します。

`list-supported-products`

サポートされている製品、この場合は AWS IoT Greengrass バージョン、現在の IDT バージョンのテストスイートバージョンを一覧表示します。

`list-test-cases`

指定したテストグループのテストケースを一覧表示します。次のオプションがサポートされています。

- `group-id`。検索するテストグループ。このオプションは必須で、1 つのグループを指定する必要があります。

`run-suite`

デバイスプールに対してテストスイートを実行します。サポートされているオプションの一部は以下のとおりです。

- `suite-id`。実行するテストスイートのバージョン。指定しない場合、IDT は `tests` フォルダにある最新バージョンを使用します。
- `group-id`。実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。
- `test-id`。実行するテストケース (カンマ区切りリストとして)。指定した場合は、`group-id` は 1 つのグループを指定する必要があります。
- `pool-id`。テストするデバイスプール。`device.json` ファイルに複数のデバイスプールが定義されている場合は、プールを指定する必要があります。
- `upgrade-test-suite`。テストスイートのバージョン更新の処理方法を制御します。IDT v3.0.0 以降、IDT は更新されたテストスイートのバージョンをオンラインでチェックします。詳細については、「the section called “テストスイートのバージョン”」を参照してください。
- `stop-on-first-failure`。最初に障害が発生した時点で実行を停止するように IDT を設定します。指定されたテストグループをデバッグするには、このオプションを `group-id` とともに使用する必要があります。完全テストスイートを実行して認定レポートを生成する場合は、このオプションを使用しないでください。
- `update-idt`。IDT を更新するプロンプトの応答を設定します。入力値が `Y` であれば、IDT が新しいバージョンを検出した場合にテストの実行を停止します。入力値が `N` であれば、テストの実行を続行します。
- `update-managed-policy`。入力値が `Y` であれば、IDT がユーザーのマネージドポリシーが更新されていないことを検出した場合にテストの実行を停止します。入力値が `N` であれば、テストの実行を続行します。

`run-suite` オプションの詳細については、次の `help` オプションを使用してください。

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

IDT v2.3.0 and earlier

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のテストスイート内のグループを一覧表示します。

`list-suites`

使用可能なテストスイートを一覧表示します。

`run-suite`

デバイスプールに対してテストスイートを実行します。

`run-suite` オプションの詳細については、次の `help` オプションを使用してください。

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

結果とログを理解する

このセクションでは、IDT の結果レポートとログを表示し、解釈する方法について説明します。

結果の表示

実行中、IDT はコンソール、ログファイル、テストレポートにエラーを書き込みます。IDT で適格性テストスイートを実行すると、2 つのレポートが生成されます。これらのレポートは *<device-tester-extract-location>*`/results/`*<execution-id>*`/` にあります。両レポートでは、適格性確認テストスイートの実行の結果をキャプチャします。

`awsiotdevicetester_report.xml` は、 AWS Partner Device Catalog にデバイスを一覧表示 AWS するために に送信する認定テストレポートです。レポートには、次の要素が含まれます。

- IDT バージョン。
- テストされた AWS IoT Greengrass バージョン。
- `device.json` ファイルで指定されている SKU とデバイスプール。
- `device.json` ファイルで指定されているデバイスプールの機能。
- テスト結果の概要の集計。
- デバイスの機能 (ローカルリソースアクセス、シャドウ、MQTT など) に基づいてテストしたライブラリごとのテスト結果の内訳。

`GGQ_Result.xml` レポートは JUnit XML 形式です。Jenkins、Bamboo などのように継続的な統合 (CI) と継続的なデプロイ (CD) のプラットフォームに統合することができます。レポートには、次の要素が含まれます。

- テスト結果の概要を集計したもの。
- テストされた AWS IoT Greengrass 機能別のテスト結果の内訳。

IDT レポートの解釈

`awsiotdevicetester_report.xml` または `awsiotdevicetester_report.xml` のレポートセクションには、実行されたテストとその結果が一覧表示されます。

最初の XML タグ `<testsuites>` には、テストの実行の概要が含まれています。例:

```
<testsuites name="GGQ results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

`<testsuites>` タグで使用される属性

`name`

テストスイートの名前。

`time`

適格性確認スイートの実行所要時間 (秒)。

`tests`

実行されたテストの数。

`failures`

実行されたテストのうち、合格しなかったものの数。

`errors`

IDT で実行できなかったテストの数。

`disabled`

この属性は使用されていないため無視できます。

awsiotdevicetester_report.xml ファイルには、テスト対象の製品および一連のテストの実行後に検証された製品機能に関する情報を含む <awsproduct> タグが含まれています。

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

features

検証された機能です。required としてマークされている機能については、資格を得るためにボードを提出するために必要です。次のスニペットは、この情報が awsiotdevicetester_report.xml ファイルで表示される方法を示します。

```
<feature name="aws-iot-greengrass-no-container" value="supported" type="required"></
feature>
```

optional としてマークされている機能は、資格を得るために必要ありません。次のスニペットは、オプションの機能を示しています。

```
<feature name="aws-iot-greengrass-container" value="supported" type="optional"></
feature>

<feature name="aws-iot-greengrass-hsi" value="not-supported" type="optional"></
feature>
```

必要な機能にテストの失敗やエラーがない場合、デバイスは を実行するための技術要件を満た AWS IoT Greengrass し、 AWS IoT サービスと相互運用できます。 AWS Partner Device Catalog にデバイスを一覧表示する場合は、このレポートを認定の証拠として使用できます。

テストに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテスト結果の要約を示します。例:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

形式は <testsuites> タグと似ていますが、使用されていないため無視できる skipped という属性があります。各 <testsuite> XML タグ内には、テストグループの実行されたテスト別の <testcase> タグがあります。例:

```
<testcase classname="Security Combination (IPD + DCM) Test Context" name="Security
 Combination IP Change Tests sec4_test_1: Should rotate server cert when IPD disabled
 and following changes are made:Add CIS conn info and Add another CIS conn info"
 attempts="1"></testcase>>
```

<testcase> タグで使用される属性

name

テストの名前。

attempts

IDT でテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

ログの表示

IDT は、*<devicetester-extract-location>*/results/*<execution-id>*/logs のテスト実行によってログを生成します。2 組のログが生成されます。

test_manager.log

AWS IoT Device Tester の Test Manager コンポーネントから生成されたログ (設定、テストシーケンス、レポート生成に関連するログなど) 。

<test_case_id>.log (for example, ota.log)

テストされているデバイスからのログを含む、テストグループのログ。テストが失敗すると、テストのテスト対象デバイスのログを含む tar.gz ファイルが作成されます (例: ota_prod_test_1_ggc_logs.tar.gz)。

詳細については、「AWS IoT Greengrass トラブルシューティング用の IDT」を参照してください。

IDT を使用して独自のテストスイートを開発および実行する

IDT v4.0.0 以降、IDT for は標準化された設定設定と結果形式をテストスイート環境と AWS IoT Greengrass 組み合わせ、デバイスとデバイスソフトウェア用のカスタムテストスイートを開発できます。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

IDT を使用してカスタムテストスイートを開発および実行するには、次の手順を実行します。

カスタムテストスイートを開発するには

- テストする Greengrass デバイス用のカスタムテストロジックを使用して、テストスイートを作成します。
- IDT と作成したカスタムテストスイートをテストの実行者に提供します。作成したテストスイートの構成設定に関する情報も提供します。

カスタムテストスイートを実行するには

- テストするデバイスをセットアップします。
- 使用するテストスイートに必要な構成設定を実装します。
- IDT を使用して、カスタムテストスイートを実行します。
- IDT によって実行されたテストのテスト結果と実行ログを表示します。

AWS IoT Device Tester for の最新バージョンをダウンロードする AWS IoT Greengrass

IDT の最新バージョンをダウンロードし、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

Windows では、パスの長さは 260 文字に制限されています。Windows を使用している場合は、パスが 260 文字以内になるようにして、IDT をルートディレクトリ (`C:\` または `D:\` など) に展開します。

テストスイート作成ワークフロー

テストスイートは 3 つのタイプのファイルで構成されます。

- IDT にテストスイートの実行方法に関する情報を提供する JSON 設定ファイル。
- IDT がテストケースの実行に使用するテスト実行可能ファイル。
- テストの実行に必要な追加のファイル。

カスタム IDT テストを作成するには、次の基本的な手順を実行します。

1. テストスイート用の JSON 設定ファイルを作成します。
2. テストスイート用のテストロジックが含まれるテストケース実行可能ファイルを作成します。
3. テストスイートを実行するためにテストの実行者に必要な設定情報を検証し、文書化します。
4. IDT が予想通りにテストスイートを実行し、テスト結果を生成できることを確認します。

サンプルカスタムスイートを迅速に構築して実行するには、チュートリアル: サンプル IDT テストスイートを構築して実行するの手順に従ってください。

Python でカスタムテストスイートの作成を開始するには、チュートリアル: シンプルな IDT テストスイートの開発を参照してください。

チュートリアル: サンプル IDT テストスイートを構築して実行する

AWS IoT Device Tester のダウンロードには、サンプルテストスイートのソースコードが含まれています。このチュートリアルを完了すると、サンプルテストスイートを構築して実行し、 AWS IoT Device Tester for を使用してカスタムテストスイート AWS IoT Greengrass を実行する方法を理解できます。

このチュートリアルでは、次の手順を実行します。

1. サンプルテストスイートを構築する
2. IDT を使用してサンプルテストスイートを実行する

前提条件

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - AWS IoT Device Tester の最新バージョン
  - Python 3.7 以降

    コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

    ```
    python3 --version
    ```

    Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

    ```
    Set-Alias -Name "python3" -Value "python"
    ```

    バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。
  - urllib3

    `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

    ```
    python3 -c 'import urllib3'
    ```

    `urllib3` がインストールされていない場合は、次のコマンドを実行してインストールします。

    ```
    python3 -m pip install urllib3
    ```
- デバイスの要件
  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

id

自分のデバイスのユーザー定義の一意の識別子。

`connectivity.ip`

自分のデバイスの IP アドレス。

`connectivity.port`

オプション。デバイスへの SSH 接続に使用するポート番号。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.user`

デバイスへのサインインに使用するユーザー名。

`connectivity.auth.credentials.privKeyPath`

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`devices.connectivity.auth.credentials.password`

自分のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

サンプルテストスイートを構築する

<device-tester-extract-location>/samples/python フォルダには、サンプル設定ファイル、ソースコード、および提供されたビルドスクリプトを使用してテストスイートに結合できる IDT クライアント SDK が含まれています。次のディレクトリツリーは、これらのサンプルファイルの場所を示しています。

```
<device-tester-extract-location>
### ...
### tests
### samples
#   ### ...
#   ### python
#       ### configuration
#       ### src
#       ### build-scripts
#           ### build.sh
#           ### build.ps1
### sdks
    ### ...
    ### python
        ### idt_client
```

テストスイートを構築するには、ホストコンピュータで次のコマンドを実行します。

Windows

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.ps1
```

Linux, macOS, or UNIX

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.sh
```

これにより、*<device-tester-extract-location>*/tests フォルダ内の IDTSampleSuitePython_1.0.0 フォルダにサンプルテストスイートが作成されます。IDTSampleSuitePython_1.0.0 フォルダのファイルを確認して、サンプルテストスイートの構造を理解し、テストケース実行可能ファイルとテスト設定 JSON ファイルのさまざまな例を参照してください。

次のステップ: IDT を使用して、作成したサンプルテストスイートを実行します。

IDT を使用してサンプルテストスイートを実行する

サンプルテストスイートを実行するには、ホストコンピュータで次のコマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT はサンプルテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
========== Test Summary ==========
Execution Time:         5s
Tests Completed:        4
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    sample_group:       PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/IDTSampleSuitePython_Report.xml
```

トラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。このチュートリアルのすべての前提条件を満たしていることを確認してください。

テスト対象のデバイスに接続できない

以下について確認します。

- `device.json` ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。
- ホストコンピュータから SSH 経由でデバイスに接続できる。

チュートリアル: シンプルな IDT テストスイートの開発

テストスイートは、以下を組み合わせたものです。

- テストロジックが含まれるテスト実行可能ファイル
- テストスイートが記述された JSON 設定ファイル

このチュートリアルでは、IDT for AWS IoT Greengrass を使用して、単一のテストケースを含む Python テストスイートを開発する方法を示します。このチュートリアルでは、次の手順を実行します。

1. テストスイートディレクトリを作成する
2. JSON 設定ファイルを作成する
3. テストケース実行可能ファイルを作成する
4. テストスイートを実行する

前提条件

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - AWS IoT Device Tester の最新バージョン
  - Python 3.7 以降

    コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

```
python3 --version
```

Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

```
Set-Alias -Name "python3" -Value "python"
```

バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。

- urllib3

  `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

  ```
  python3 -c 'import urllib3'
  ```

  `urllib3` がインストールされていない場合は、次のコマンドを実行してインストールします。

  ```
  python3 -m pip install urllib3
  ```

- デバイスの要件

  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

    Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

テストスイートディレクトリを作成する

IDT は、テストケースを、各テストスイート内のテストグループに論理的に分離します。各テストケースはテストグループ内に存在する必要があります。このチュートリアルでは、`MyTestSuite_1.0.0` という名前のフォルダを作成し、このフォルダ内に次のディレクトリツリーを作成します。

```
MyTestSuite_1.0.0
### suite
    ### myTestGroup
        ### myTestCase
```

JSON 設定ファイルを作成する

テストスイートには、次の必須の JSON 設定ファイルが含まれている必要があります。

必須の JSON ファイル

`suite.json`

テストスイートに関する情報が含まれています。「suite.json を設定する」を参照してください。

`group.json`

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに `group.json` ファイルを作成する必要があります。「group.json を設定する」を参照してください。

`test.json`

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに `test.json` ファイルを作成する必要があります。「test.json を設定する」を参照してください。

1. `MyTestSuite_1.0.0/suite` フォルダで、次の構造の `suite.json` ファイルを作成します。

```
{
    "id": "MyTestSuite_1.0.0",
    "title": "My Test Suite",
    "details": "This is my test suite.",
    "userDataRequired": false
}
```

2. `MyTestSuite_1.0.0/myTestGroup` フォルダで、次の構造の `group.json` ファイルを作成します。

```
{
    "id": "MyTestGroup",
    "title": "My Test Group",
```

```
    "details": "This is my test group.",
    "optional": false
}
```

3. MyTestSuite_1.0.0/myTestGroup/myTestCase フォルダで、次の構造の test.json ファイルを作成します。

```
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

MyTestSuite_1.0.0 フォルダのディレクトリツリーは次のようになります。

```
MyTestSuite_1.0.0
### suite
    ### suite.json
    ### myTestGroup
        ### group.json
        ### myTestCase
```

```
            ### test.json
```

IDT クライアント SDK を入手する

IDT がテスト対象のデバイスとやり取りし、テスト結果をレポートできるようにするには、IDT クライアント SDK を使用します。このチュートリアルでは、Python バージョンの SDK を使用します。

<device-tester-extract-location>/sdks/python/ フォルダから、idt_client フォルダを自分の MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダにコピーします。

SDK が正常にコピーされたことを確認するには、次のコマンドを実行します。

```
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

テストケース実行可能ファイルを作成する

テストケース実行可能ファイルには、実行するテストロジックが含まれています。テストスイートには、複数のテストケース実行可能ファイルを含めることができます。このチュートリアルでは、テストケース実行可能ファイルを 1 つだけ作成します。

1. テストスイートファイルを作成します。

   MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダで、次の内容の myTestCase.py ファイルを作成します。

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

   if __name__ == "__main__":
       main()
   ```

2. クライアント SDK 関数を使用して、自分の myTestCase.py ファイルに次のテストロジックを追加します。

   a. テスト対象のデバイスで SSH コマンドを実行します。

      ```
      from idt_client import *
      ```

```
def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

if __name__ == "__main__":
    main()
```

b. テスト結果を IDT に送信します。

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))

    # Send the result
    client.send_result(sr_req)

if __name__ == "__main__":
    main()
```

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

id

自分のデバイスのユーザー定義の一意の識別子。

connectivity.ip

自分のデバイスの IP アドレス。

connectivity.port

オプション。デバイスへの SSH 接続に使用するポート番号。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.user`

デバイスへのサインインに使用するユーザー名。

`connectivity.auth.credentials.privKeyPath`

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`devices.connectivity.auth.credentials.password`

自分のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

> **Note**
>
> `method` が `pki` に設定されている場合のみ `privKeyPath` を指定します。
> `method` が `password` に設定されている場合のみ `password` を指定します。

テストスイートを実行する

テストスイートを作成したら、テストスイートが期待どおりに機能することを確認します。そのために、次の手順に従って、既存のデバイスプールを使用してテストスイートを実行します。

1. 自分の MyTestSuite_1.0.0 フォルダを *<device-tester-extract-location>*/tests にコピーします。
2. 以下の コマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
```

IDT はテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
time="2020-10-19T09:24:47-07:00" level=info msg=Using pool: pool
time="2020-10-19T09:24:47-07:00" level=info msg=Using test suite "MyTestSuite_1.0.0"
 for execution
time="2020-10-19T09:24:47-07:00" level=info msg=b'hello world\n'
 suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:47-07:00" level=info msg=All tests finished.
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:48-07:00" level=info msg=

========== Test Summary ==========
Execution Time:         1s
Tests Completed:        1
Tests Passed:           1
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    myTestGroup:        PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/logs
```

```
Path to Aggregated JUnit Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
```

トラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。エラーログを確認する前に、次の点を確認してください。

- IDT クライアント SDK が、このステップで説明された通りの正しいフォルダにある。
- このチュートリアルのすべての前提条件を満たしている。

テスト対象のデバイスに接続できない

以下について確認します。

- `device.json` ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。
- ホストコンピュータから SSH 経由でデバイスに接続できる。

IDT テストスイート設定ファイルを作成する

このセクションでは、カスタムテストスイートの作成時に、スートに含める JSON 設定ファイルを作成する形式について説明します。

必須の JSON ファイル

`suite.json`

テストスイートに関する情報が含まれています。「suite.json を設定する」を参照してください。

`group.json`

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに `group.json` ファイルを作成する必要があります。「group.json を設定する」を参照してください。

`test.json`

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに `test.json` ファイルを作成する必要があります。「test.json を設定する」を参照してください。

オプションの JSON ファイル

`state_machine.json`

IDT がテストスイートを実行するときのテストの実行方法を定義します。「state_machine.json を構成する」を参照してください。

`userdata_schema.json`

テストの実行者が設定構成に含めることができる `userdata.json` ファイルのスキーマを定義します。`userdata.json` ファイルは、テストの実行に必要であるものの、`device.json` ファイルに含まれていない、追加の設定情報用に使用します。「userdata_schema.json を構成する」を参照してください。

JSON 設定ファイルは、以下に示すように *<custom-test-suite-folder>* に配置します。

```
<custom-test-suite-folder>
### suite
    ### suite.json
    ### state_machine.json
    ### userdata_schema.json
    ### <test-group-folder>
        ### group.json
        ### <test-case-folder>
            ### test.json
```

suite.json を設定する

`suite.json` ファイルは、環境変数を設定し、テストスイートの実行にユーザーデータが必要かどうかを決定します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/`suite.json` ファイルを設定します。

```
{
    "id": "<suite-name>_<suite-version>",
```

```
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストスイートの一意のユーザー定義 ID。`id` の値は、`suite.json` ファイルが配置されているテストスイートフォルダ名と一致する必要があります。スイート名とスイートのバージョンは、次の要件も満たしている必要があります。

- *<suite-name>* にアンダースコアを含めることはできません。
- *<suite-version>* が *x.x.x* として表されている (x は数字)。

ID は IDT によって生成されるテストレポートに表示されます。

`title`

このテストスイートでテストされる製品または機能のユーザー定義名。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストスイートの目的の簡単な説明。

`userDataRequired`

テストの実行者が `userdata.json` ファイルにカスタム情報を含める必要があるかどうかを定義します。この値を `true` に設定した場合は、テストスイートフォルダに `userdata_schema.json` ファイルも含める必要があります。

`environmentVariables`

オプション。このテストスイートに設定する環境変数の配列。

`environmentVariables.key`

環境変数の名前。

`environmentVariables.value`

環境変数の値。

group.json を設定する

`group.json` ファイルは、テストグループが必須かオプションかを定義します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/group.json ファイルを設定します。

```
{
    "id": "<group-id>",
    "title": "<group-title>",
    "details": "<group-details>",
    "optional": true | false,
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストグループの一意のユーザー定義 ID。`id` の値は、`group.json` ファイルが配置されているテストグループフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

`title`

テストグループのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストグループの目的の簡単な説明。

`optional`

オプション。`true` に設定すると、IDT が必須テストの実行を完了した後に、このテストグループがオプショングループとして表示されます。デフォルト値は `false` です。

test.json を設定する

test.json ファイルは、テストケースによって使用されるテストケース実行ファイルと環境変数を決定します。テストケース実行可能ファイルの作成の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/*<test-case>*/test.json ファイルを設定します。

```
{
    "id": "<test-id>",
    "title": "<test-title>",
    "details": "<test-details>",
    "requireDUT": true | false,
    "requiredResources": [
        {
            "name": "<resource-name>",
            "features": [
                {
                    "name": "<feature-name>",
                    "version": "<feature-version>",
                    "jobSlots": <job-slots>
                }
            ]
        }
    ],
    "execution": {
        "timeout": <timeout>,
        "mac": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "linux": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "win": {
            "cmd": "/path/to/executable",
            "args": [
```

```
                "<argument>"
            ]
        }
    },
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

テストケースの一意のユーザー定義 ID。`id` の値は、`test.json` ファイルが配置されているテストケースフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

`title`

テストケースのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

`details`

テストケースの目的の簡単な説明。

`requireDUT`

オプション。このテストの実行にデバイスが必要な場合は `true` に設定します。必要ない場合は `false` に設定します。デフォルト値は `true` です。テストの実行者は、テストの実行に使用するデバイスを `device.json` ファイルに設定します。

`requiredResources`

オプション。このテストの実行に必要なリソースデバイスに関する情報を指定する配列。

`requiredResources.name`

このテストの実行時にリソースデバイスに与える一意の名前。

`requiredResources.features`

ユーザー定義のリソースデバイス機能の配列。

`requiredResources.features.name`

機能の名前。このデバイスが使用するデバイス機能。この名前は、テストの実行者によって `resource.json` ファイルに指定される機能名に対してマッチングされます。

`requiredResources.features.version`

オプション。機能のバージョン。この値は、テストの実行者によって `resource.json` ファイルに指定される機能のバージョンに対してマッチングされます。バージョンが指定されていない場合、機能はチェックされません。機能にバージョン番号が必要ない場合は、このフィールドは空白のままにしてください。

`requiredResources.features.jobSlots`

オプション。この機能がサポートできる同時テストの数。デフォルト値は 1 です。IDT が機能ごとに異なるデバイスを使用する場合は、この値を 1 に設定することをお勧めします。

`execution.timeout`

IDT がテスト実行終了まで待機する時間 (ミリ秒単位)。この値の設定の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

`execution.os`

IDT を実行するホストコンピュータのオペレーティングシステムに基づいて実行されるテストケースの実行可能ファイル。サポートされている値は `linux`、`mac`、`win` です。

`execution.os.cmd`

指定されたオペレーティングシステムで実行するテストケース実行可能ファイルへのパス。この場所は、システムパス内に存在する必要があります。

`execution.os.args`

オプション。テストケースの実行可能ファイルを実行するために指定する引数。

`environmentVariables`

オプション。このテストケース用に設定された環境変数の配列。

`environmentVariables.key`

環境変数の名前。

`environmentVariables.value`

環境変数の値。

> **Note**
>
> test.json ファイルと suite.json ファイルに同じ環境変数を設定した場合は、test.json ファイルの値が優先されます。

state_machine.json を構成する

ステートマシンは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のステートマシンが含まれていない場合は、IDT によってステートマシンが生成されます。デフォルトのステートマシンには、次の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT ステートマシンの機能の詳細については、IDT ステートマシンを設定するを参照してください。

userdata_schema.json を構成する

userdata_schema.json ファイルは、テストの実行者がユーザーデータを指定するスキーマを決定します。ユーザーデータは、テストスイートが device.json ファイルに含まれていない情報を必要とする場合に必要になります。例えば、テストを実行するために、Wi-Fi ネットワークの認証情報、特定のオープンポート、またはユーザーが提供する証明書が必要になる場合があります。この情報は、userdata という入力パラメータとして IDT に提供できます。これは、ユーザーが *<device-tester-extract-location>*/config フォルダに作成するuserdata.json ファイルの値です。userdata.json の形式は、テストケースに含まれている userdata_schema.json ファイルに基づきます。

テストの実行者が userdata.json ファイルを提供しなければならないことを示す方法

1. suite.json ファイルで、userDataRequired を true に設定します。

2. *<custom-test-suite-folder>* で、`userdata_schema.json` ファイルを作成します。
3. `userdata_schema.json` ファイルを編集して、有効な IETF Draft v4 JSON Schema を作成します。

IDT は、テストスイートを実行するときに、このスキーマを自動的に読み込み、テストの実行者によって提供される `userdata.json` ファイルの検証に使用します。有効な場合、`userdata.json` ファイルのコンテンツは IDT コンテキストおよび、ステートマシンコンテキストの両方で使用可能になります。

IDT ステートマシンを設定する

ステートマシンは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のステートマシンが含まれていない場合は、IDT によってステートマシンが生成されます。デフォルトのステートマシンには、次の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストスイートのステートマシンは、次の基準を満たす必要があります。

- 各ステートが、IDT が実行する各アクション (テストグループの実行、レポートファイルの生成など) に対応する。
- ステートが移行すると、そのステートに関連付けられたアクションを実行する。
- 各ステートが、次のステートの移行ルールを定義する。
- 終了ステートが `Succeed` または `Fail` である。

ステートマシンの形式

次のテンプレートを使用して、独自の *<custom-test-suite-folder>*/suite/state_machine.json ファイルを設定できます。

```
{
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }

    // Required states
    "Succeed": {
      "Type": "Succeed"
    },
    "Fail": {
      "Type": "Fail"
    }
  }
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Comment`

ステートマシンの説明。

`StartAt`

IDT がテストスイートの実行を開始するステートの名前。`StartAt` の値は、`States` オブジェクトにリストされているいずれかのステートに設定する必要があります。

`States`

ユーザー定義のステート名を有効な IDT ステートにマッピングするオブジェクト。各 States.*state-name* オブジェクトには、*state-name* にマッピングされた有効なステートの定義が含まれています。

`States` オブジェクトには、`Succeed` ステートおよび `Fail` ステートを含める必要があります。有効なステートについては、有効なステートとステートの定義を参照してください。

有効なステートとステートの定義

このセクションでは、IDT ステートマシンで使用可能なすべての有効なステートのステート定義について説明します。以下に示すステートの一部は、テストケースレベルでの設定をサポートしています。ただし、絶対に必要な場合を除き、テストケースレベルではなく、テストグループレベルでステート移行ルールを設定することをお勧めします。

ステートの定義

- RunTask
- 選択
- 並行
- AddProductFeatures
- レポートを行う
- LogMessage
- SelectGroup
- 失敗
- 成功

RunTask

RunTask ステートは、テストスイートで定義されているテストグループからテストケースを実行します。

```
{
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
    "TestCases": [
        "<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`TestGroup`

オプション。実行するテストグループの ID。この値を指定しない場合、IDT はテストの実行者が選択するテストグループを実行します。

`TestCases`

オプション。`TestGroup` に指定されたグループのテストケース ID の配列。IDT は、`TestGroup` と `TestCases` の値に基づいて、次のようにテストの実行動作を決定します。

- `TestGroup` と `TestCases` 両方が指定されている場合、IDT はテストグループから指定されたテストケースを実行します。
- `TestCases` が指定され、`TestGroup` が指定されていない場合、IDT は指定されたテストケースを実行します。
- `TestGroup` が指定され、`TestCases` が指定されていない場合は、IDT は指定されたテストグループ内のすべてのテストケースを実行します。
- `TestGroup` も `TestCases` も指定されていない場合、IDT は、テストの実行者が IDT CLI から選択したテストグループからすべてのテストケースを実行します。テストの実行者がグループを選択できるようにするには、`statemachine.json` ファイルに `RunTask` ステートと `Choice`ステート両方を含める必要があります。これを行う方法の例については、ステートマシンの例: ユーザーが選択したテストグループを実行するを参照してください。

  テストの実行者向けの IDT CLI コマンドを有効にする方法については「the section called “IDT CLI コマンドを有効にする”」を参照してください。

`ResultVar`

テスト実行の結果によって設定するコンテキスト変数の名前。`TestGroup` の値を指定しなかった場合は、この値を指定しないでください。IDT は、以下に基づいて、`ResultVar` に定義された変数を `true` または `false` に設定します。

- 変数名の形式が *`text`*_*`text`*_passed の場合、この値は、最初のテストグループのすべてのテストが合格したか、スキップされたかに設定されます。
- それ以外の場合、この値は、すべてのテストグループのすべてのテストが合格したか、スキップされたかに設定されます。

通常、`RunTask` ステートは、個々のテストケース ID を指定せずにテストグループ ID を指定するために使用されます。この指定により、IDT は指定されたテストグループ内のすべてのテストケースを実行します。このステートで実行されるすべてのテストケースは、ランダムな順序で並行して実行さ

れます。ただし、すべてのテストケースが実行に 1 つのデバイスを必要とし、単一のデバイスしか使用できない場合は、テストケースは順次実行されます。

エラー処理

指定されたテストグループ ID またはテストケース ID のいずれかが有効でない場合、このステートは RunTaskError 実行エラーを発行します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionError 変数を true に設定します。

選択

Choice ステートでは、ユーザー定義の条件に基づいて、移行先の次のステートを動的に設定できます。

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Default

Choices に定義されているいずれの式も true に評価されない場合に移行先になるデフォルトのステート。

FallthroughOnError

オプション。このステートが式評価エラーに遭遇したときの動作を指定します。評価結果がエラーになったときに式をスキップしたい場合は true に設定します。一致する式がない場合、ステートマシンは Default ステートに移行します。FallthroughOnError 値は、指定されない場合、デフォルトで false になります。

Choices

現在のステートのアクションを実行した後に移行するステートを決定する式とステートの配列。

`Choices.Expression`

ブール値に評価される式文字列。式が `true` と評価された場合、ステートマシンは `Choices.Next` に定義されているステートに移行します。式文字列は、ステートマシンコンテキストから値を取得し、オペレーションを実行してブール値に到達します。ステートマシンコンテキストへのアクセスについては、「ステートマシンコンテキスト」を参照してください。

`Choices.Next`

`Choices.Expression` で定義されている式が `true` に評価された場合の移行先のステート名。

エラー処理

以下に示すケースでは、`Choice` ステートでエラー処理が必要になることがあります。

- choice 式の一部の変数が、ステートマシンのコンテキストに存在しない。
- 式の結果がブール値ではない。
- JSON 検索の結果が、文字列、数値、またはブール値ではない。

このステートのエラー処理に `Catch` ブロックを使用することはできません。ステートマシンがエラーに遭遇したときに、その実行を停止するには、`FallthroughOnError` を `false` に設定する必要があります。ただし、`FallthroughOnError` は `true` に設定し、ユースケースに応じて、次のいずれかの操作を実行することをお勧めします。

- アクセスしている変数が一部のケースに存在しないと考えられる場合は、`Default` の値と追加の `Choices` ブロックを使用して次のステートを指定します。
- 使用している変数が必ず存在するものである場合は、`Default` ステートを `Fail` に設定します。

並行

`Parallel` ステートでは、新しいステートマシンを互いに並列に定義して実行できます。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Branches": [
```

```
        <state-machine-definition>
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`Branches`

実行するステートマシン定義の配列。各ステートマシン定義には、それぞれの `StartAt`、`Succeed`、および `Fail` ステートを含める必要があります。この配列内のステートマシン定義は、各自の定義外のステートを参照することはできません。

> **Note**
>
> 各ブランチステートマシンは同じステートマシンコンテキストを共有するため、あるブランチに変数を設定し、別のブランチからそれらの変数を読み込むと、予期しない動作が発生する可能性があります。

`Parallel` ステートは、すべてのブランチステートマシンを実行してから次のステートに移行します。デバイスを必要とする各ステートは、デバイスが利用可能になるまで実行を待ちます。複数のデバイスが利用可能な場合、このステートは並行して複数のグループからテストケースを実行します。十分な数のデバイスが利用できない場合、テストケースは順次実行されます。テストケースは、並列して実行される場合、ランダムな順序で実行されるため、同じテストグループからのテストの実行に異なるデバイスが使用されることがあります。

エラー処理

実行エラーを処理するには、ブランチステートマシンと親ステートマシンの両方が、`Fail` ステートに移行していることを確認します。

ブランチステートマシンは親ステートマシンに実行エラーを送信しないため、ブランチステートマシンの実行エラーを処理するために `Catch` ブロックを使用することはできません。代わりに、共有ステートマシンコンテキストの `hasExecutionErrors` 値を使用します。これを行う方法の例については、ステートマシンの例: 2 つのテストグループを並行して実行するを参照してください。

AddProductFeatures

AddProductFeatures ステートでは、IDT によって生成される awsiotdevicetester_report.xml ファイルに製品機能を追加できます。

製品機能とは、デバイスが満たしている可能性のある特定の基準に関するユーザー定義の情報です。例えば、MQTT 製品機能には、デバイスが MQTT メッセージを適切に公開することを指定できます。レポートでは、製品機能は指定されたテストが合格したかどうかに応じて、supported、not-supported、カスタム値に設定されます。

> **Note**
>
> AddProductFeatures ステートだけではレポートは生成されません。レポートを生成するには、このステートが Report ステートに移行する必要があります。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Features": [
        {
            "Feature": "<feature-name>",
            "Groups": [
                "<group-id>"
            ],
            "OneOfGroups": [
                "<group-id>"
            ],
            "TestCases": [
                "<test-id>"
            ],
            "IsRequired": true | false,
            "ExecutionMethods": [
                "<execution-method>"
            ]
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`Features`

`awsiotdevicetester_report.xml` ファイルに表示される製品機能の配列。

`Feature`

機能の名前。

`FeatureValue`

オプション。`supported` の代わりにレポートで使用するカスタム値。この値を指定しない場合、テスト結果に基づいて、機能値は `supported` または `not-supported` に設定されます。

`FeatureValue` にカスタム値を使用する場合は、同じ機能を異なる条件でテストできます。IDT は、サポート条件の機能値を連結します。例えば、以下の抜粋は、`MyFeature` 機能と 2 つの異なる機能値を示しています。

```
...
{
    "Feature": "MyFeature",
    "FeatureValue": "first-feature-supported",
    "Groups": ["first-feature-group"]
},
{
    "Feature": "MyFeature",
    "FeatureValue": "second-feature-supported",
    "Groups": ["second-feature-group"]
},
...
```

両方のテストグループが合格した場合、機能値は `first-feature-supported, second-feature-supported` に設定されます。

`Groups`

オプション。テストグループ ID の配列。機能をサポートするには、指定された各テストグループ内のすべてのテストが合格である必要があります。

`OneOfGroups`

オプション。テストグループ ID の配列。機能をサポートするには、指定されたテストグループうち、少なくとも 1 つのグループに含まれるすべてのテストが合格である必要があります。

`TestCases`

オプション。テストケース ID の配列。この値を指定すると、次のことが適用されます。

- 機能をサポートするには、指定されたすべてのテストケースが合格である必要があります。
- `Groups` には、テストグループ ID を 1 つだけ含める必要があります。
- `OneOfGroups` は指定できません。

`IsRequired`

オプション。この機能をレポートでオプション機能としてマークするには、`false` に設定します。デフォルト値は `true` です。

`ExecutionMethods`

オプション。`device.json` ファイルに指定された `protocol` 値と一致する実行メソッドの配列。この値を指定した場合、この機能をレポートに含めるには、テストの実行者はこの配列の値の 1 つに一致する `protocol` 値を指定する必要があります。この値を指定しない場合、この機能は常にレポートに含まれます。

`AddProductFeatures` ステートを使用するには、`RunTask` ステートの `ResultVar` の値を以下のいずれかの値に指定する必要があります。

- 個々のテストケース ID を指定した場合は、`ResultVar` を*`group-id_test-id`*_passed に指定します。
- 個々のテストケース ID を指定しなかった場合は、`ResultVar` を*`group-id`*_passed に指定します。

`AddProductFeatures` ステートは、次の方法でテスト結果をチェックします。

- テストケース ID を指定しなかった場合は、各テストグループの結果は、ステートマシンコンテキスト内の *`group-id`*_passed 変数の値から決定されます。
- テストケース ID を指定した場合は、各テストの結果は、ステートマシンコンテキスト内の *`group-id_test-id`*_passed 変数の値から決定されます。

エラー処理

このステートで指定されたグループ ID が有効なグループ ID でない場合、このステートで AddProductFeaturesError 実行エラーが発生します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionErrors 変数を true に設定します。

レポートを行う

Report ステートでは、*suite-name*_Report.xml ファイルと awsiotdevicetester_report.xml ファイルが生成されます。またこのステートでは、レポートがコンソールにストリーミングされます。

```
{
    "Type": "Report",
    "Next": "<state-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

テストの実行者がテスト結果を確認できるように、テスト実行フローの終了ステートの前に Report ステートに移行する必要があります。通常、このステートの次のステートは Succeed です。

エラー処理

このステートは、レポート生成時に問題に遭遇した場合、ReportError 実行エラーを発行します。

LogMessage

LogMessage ステートでは、test_manager.log ファイルが生成され、ログメッセージがコンソールにストリーミングされます。

```
{
    "Type": "LogMessage",
    "Next": "<state-name>"
    "Level": "info | warn | error"
    "Message": "<message>"
```

```
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`Level`

ログメッセージを作成するエラーレベル。有効でないレベルを指定すると、エラーメッセージが生成され、そのレベルは破棄されます。

`Message`

ログに記録するメッセージ。

SelectGroup

`SelectGroup` ステートでは、ステートマシンコンテキストを更新して選択されたグループを示します。このステートで設定した値は、後続のすべての `Choice` ステートによって使用されます。

```
{
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [
        <group-id>"
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`TestGroups`

選択済みとしてマークされるテストグループの配列。この配列の各テストグループ ID について、*group-id*_selected 変数がコンテキストで `true` に設定されます。IDT は、指定されたグループが存在するかどうかを検証しないため、有効なテストグループ ID を指定するようにしてください。

失敗

Fail ステートは、ステートマシンが正しく実行されなかったことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
    "Type": "Fail"
}
```

成功

Succeed ステートは、ステートマシンが正しく実行されたことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
    "Type": "Succeed"
}
```

ステートマシンコンテキスト

ステートマシンコンテキストは、実行中のステートマシンに利用可能なデータが含まれている読み取り専用 JSON ドキュメントです。ステートマシンコンテキストは、ステートマシンからのみアクセス可能で、テストフローを決定する情報が含まれています。例えば、テストの実行者によって userdata.json ファイルに設定された情報を使用して、特定のテストを実行する必要があるかどうかを決定できます。

ステートマシンコンテキストでは、次の形式が使用されます。

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    },
    "suiteFailed": true | false,
    "specificTestGroups": [
```

```
        "<group-id>"
    ],
    "specificTestCases": [
        "<test-id>"
    ],
    "hasExecutionErrors": true
}
```

pool

テスト実行用に選択されたデバイスプールに関する情報。選択されたデバイスプールのこの情報は、device.json ファイルで定義された、対応する最上位レベルのデバイスプール配列要素から取得されます。

userData

userdata.json ファイル内の情報。

config

config.json ファイル内の情報。

suiteFailed

この値は、ステートマシンが起動すると false に設定されます。テストグループが RunTask ステートで失敗した場合、この値はステートマシン実行の残りの時間の間 true に設定されます。

specificTestGroups

テストの実行者がテストスイート全体ではなく特定のテストグループを選択して実行する場合に、このキーが作成され、特定のテストグループ ID のリストが格納されます。

specificTestCases

テストの実行者がテストスイート全体ではなく特定のテストケースを選択して実行する場合に、このキーが作成され、特定のテストケース ID のリストが格納されます。

hasExecutionErrors

ステートマシンの起動時には存在しません。いずれかのステートが実行エラーに遭遇した場合に、この変数が作成され、ステートマシンの実行の残りの時間の間 true に設定されます。

コンテキストは、JSONPath 表記法を使用してクエリできます。ステート定義における JSonPath クエリの構文は {{$.*query*}} です。JSONPath クエリは、一部のステートではプレースホル

ダー文字列として使用できます。IDT は、プレースホルダー文字列をコンテキストから評価された JSONPath クエリの値に置き換えます。プレースホルダーは、次の値に使用できます。

- `RunTask` ステートの `TestCases` 値。
- `Choice` ステートの `Expression` 値。

ステートマシンコンテキストからデータにアクセスする場合は、次の条件を満たしていることを確認します。

- JSON パスが `$.` で始まっている。
- 各値が、文字列、数値、またはブール値として評価される。

JSONPath 表記を使用してコンテキストのデータにアクセスする方法の詳細については、IDT コンテキストを使用するを参照してください。

実行エラー

実行エラーとは、ステートの実行時にステートマシンが遭遇する、ステートマシン定義内のエラーです。IDT は、各エラーに関する情報を `test_manager.log` ファイルに記録し、ログメッセージをコンソールにストリーミングします。

実行エラーは、次の方法を使用して処理できます。

- ステート定義内に `Catch`ブロック を追加する。
- ステートマシンコンテキストで `hasExecutionErrors` 値 の値を確認する。

Catch

`Catch` を使用するには、ステート定義に以下を追加します。

```
"Catch": [
    {
        "ErrorEquals": [
            "<error-type>"
        ]
        "Next": "<state-name>"
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Catch.ErrorEquals`

キャッチするエラータイプの配列。実行エラーが指定された値のいずれかと一致する場合、ステートマシンは、`Catch.Next` に指定されているステートに移行します。生成されるエラーのタイプの詳細については、各ステート定義を参照してください。

`Catch.Next`

現在のステートが、`Catch.ErrorEquals` に指定されている値のいずれかと一致する実行エラーに遭遇した場合に、移行する次のステート。

キャッチブロックは、いずれかが一致するまで順番に処理されます。どのエラーもキャッチブロックに指定されているエラーと一致しない場合、ステートマシンは実行を継続します。実行エラーは誤ったステート定義によって発生するため、ステートが実行エラーに遭遇したときは Fail ステートに移行することをお勧めします。

hasExecutionError

一部のステートは、実行エラーに遭遇した場合、エラーを発行するだけでなく、ステートマシンコンテキストの `hasExecutionError` 値も `true` に設定します。この値を使用して、エラーがいつ発生したかを特定してから、`Choice` ステートを使用してステートマシンを `Fail` ステートに移行することができます。

この方法には次の特徴があります。

- ステートマシンは、`hasExecutionError` に値が割り当てられていると開始しません。またこの値は、特定のステートによって設定されるまで得られません。つまり、明示的に `FallthroughOnError` の値を `false` に設定することによって、実行エラーが発生していない場合に、この値にアクセスする `Choice` ステートがステートマシンを停止しないようにする必要があります。
- `hasExecutionError` は、一度 `true` に設定されると、false に設定されることも、コンテキストから削除されることもありません。つまり、この値は `true` に設定された初回のみ有効であり、以降のすべてのステートに対して意味のある値を提供しないことを意味します。
- `hasExecutionError` 値は `Parallel` ステート内のすべてのブランチステートマシンで共有されるため、アクセスされる順序によっては、予期せぬ結果が発生する可能性があります。

これらの特性から、代わりに Catch ブロックを使用できる場合は、この方法を使用することはお勧めしません。

ステートマシンの例

このセクションでは、ステートマシンの設定の例を紹介します。

例

- ステートマシンの例: 1 つのテストグループを実行する
- ステートマシンの例: ユーザーが選択したテストグループを実行する
- ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する
- ステートマシンの例: 2 つのテストグループを並行して実行する

ステートマシンの例: 1 つのテストグループを実行する

このステートマシンの動作:

- ID GroupA のテストグループを実行します。このテストグループは、group.json ファイルのスイート内に存在している必要があります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
```

```
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: ユーザーが選択したテストグループを実行する

このステートマシンの動作:

- テストの実行者が特定のテストグループを選択したかどうかをチェックします。テストの実行者がテストケースを選択するには、テストグループも選択する必要があるため、ステートマシンは特定のテストケースはチェックしません。
- テストグループが選択されている場合:
  - 選択されたテストグループ内のテストケースを実行します。そのために、ステートマシンは、RunTask ステートでは、テストグループまたはテストケースを明示的に指定しません。
  - すべてのテストを実行した後にレポートを生成し、終了します。
- テストグループが選択されていない場合:
  - テストグループ GroupA のテストを実行します。
  - レポートを生成して終了します。

```
{
```

```
    "Comment": "Runs specific groups if the test runner chose to do that, otherwise
 runs GroupA.",
    "StartAt": "SpecificGroupsCheck",
    "States": {
        "SpecificGroupsCheck": {
            "Type": "Choice",
            "Default": "RunGroupA",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.specificTestGroups[0]}} != ''",
                    "Next": "RunSpecificGroups"
                }
            ]
        },
        "RunSpecificGroups": {
            "Type": "RunTask",
            "Next": "Report",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
```

```
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する

このステートマシンの動作:

- テストグループ GroupA を実行します。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- FeatureThatDependsOnGroupA 機能を awsiotdevicetester_report.xml ファイルに追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs GroupA and adds product features based on GroupA",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "AddProductFeatures",
            "TestGroup": "GroupA",
            "ResultVar": "GroupA_passed",
            "Catch": [
```

```
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 2 つのテストグループを並行して実行する

このステートマシンの動作:

- GroupA および GroupB テストグループを並行して実行します。ブランチステートマシンの RunTask ステートによってコンテキストに格納された ResultVar 変数が AddProductFeatures ステートに利用可能になります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。このステートマシンは、Catch ブロックを使用しません。この方法では、ブランチステートマシンの実行エラーが検出されないためです。
- 合格したグループに基づいて、awsiotdevicetester_report.xml ファイルに機能を追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

デバイスプールに 2 つのデバイスが設定されている場合、GroupA と GroupB 両方を同時に実行できます。ただし、GroupA または GroupB のどちらかに複数のテストが含まれている場合は、両方のデバイスがそれらのテストに割り当てられることがあります。デバイスが 1 つだけ設定されている場合、テストグループは順次実行されます。

```
{
    "Comment": "Runs GroupA and GroupB in parallel",
    "StartAt": "RunGroupAAndB",
    "States": {
        "RunGroupAAndB": {
            "Type": "Parallel",
            "Next": "CheckForErrors",
            "Branches": [
                {
                    "Comment": "Run GroupA state machine",
                    "StartAt": "RunGroupA",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupA",
                            "ResultVar": "GroupA_passed",
```

```
                            "Catch": [
                                {
                                    "ErrorEquals": [
                                        "RunTaskError"
                                    ],
                                    "Next": "Fail"
                                }
                            ]
                        },
                        "Succeed": {
                            "Type": "Succeed"
                        },
                        "Fail": {
                            "Type": "Fail"
                        }
                    }
                },
                {
                    "Comment": "Run GroupB state machine",
                    "StartAt": "RunGroupB",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupB",
                            "ResultVar": "GroupB_passed",
                            "Catch": [
                                {
                                    "ErrorEquals": [
                                        "RunTaskError"
                                    ],
                                    "Next": "Fail"
                                }
                            ]
                        },
                        "Succeed": {
                            "Type": "Succeed"
                        },
                        "Fail": {
                            "Type": "Fail"
                        }
                    }
                }
            ]
```

```
        },
        "CheckForErrors": {
            "Type": "Choice",
            "Default": "AddProductFeatures",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.hasExecutionErrors}} == true",
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                },
                {
                    "Feature": "FeatureThatDependsOnGroupB",
                    "Groups": [
                        "GroupB"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
```

```
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

IDT テストケース実行可能ファイルを作成する

テストケース実行可能ファイルは、次の方法でテストスイートフォルダ内に作成して配置できます。

- test.json ファイル内の引数または環境変数を使用して実行するテストを決定するテストスイートの場合は、テストスイート全体に対して 1 つのテストケース実行可能ファイルを作成することも、テストスイート内のテストグループごとに 1 つのテスト実行可能ファイルを作成することもできます。
- 指定したコマンドに基づいて特定のテストを実行するテストスイートの場合は、テストスイート内のテストケースごとに 1 つのテストケース実行可能ファイルを作成します。

テストを作成するユーザーは、ユースケースに適したアプローチを決定し、それに応じてテストケースの実行可能ファイルを構成できます。各 test.json ファイルに、正しいテストケース実行可能ファイルのパスを指定していること、および指定した実行可能ファイルが正常に実行されることを確認してください。

すべてのデバイスにテストケースを実行する準備が整うと、IDT は以下のファイルを読み取ります。

- test.json。選択されたテストケースが、開始するプロセスと設定する環境変数を決定するために使用します。
- suite.json。テストスイートが、設定する環境変数を決定するために使用します。

IDT は、test.json ファイルに指定されているコマンドと引数に基づいて必要なテスト実行可能ファイルプロセスを開始し、必要な環境変数をこのプロセスに渡します。

IDT クライアント SDK を使用する

IDT クライアント SDK を使用すると、IDT とテスト対象のデバイスとのやり取りに使用できる API コマンドを使用して、テスト実行可能ファイルにテストロジックを簡単に記述できます。現在、IDT では次の SDK が用意されています。

- IDT クライアント SDK for Python
- IDT クライアント SDK for Go

これらの SDK は、`<device-tester-extract-location>/sdks` フォルダにあります。新しいテストケース実行可能ファイルを作成するときは、使用する SDK をテストケース実行可能ファイルが含まれるフォルダにコピーし、コード内で SDK を参照する必要があります。このセクションでは、テストケースの実行可能ファイルで使用できる API コマンドについて簡単に説明します。

このセクションの内容

- デバイスとのやり取り
- IDT とのやり取り
- ホストとのやり取り

デバイスとのやり取り

次のコマンドを使用すると、デバイスとのやり取りや接続管理のための追加の関数を実装せずに、テスト対象デバイスと通信することができます。

`ExecuteOnDevice`

テストスイートが、SSH または Docker シェル接続をサポートするデバイス上で、シェルコマンドを実行できるようにします。

`CopyToDevice`

テストスイートが、IDT を実行するホストマシンから、SSH または Docker シェル接続をサポートするデバイス上の指定された場所にローカルファイルをコピーできるようにします。

`ReadFromDevice`

テストスイートが、UART 接続をサポートするデバイスのシリアルポートから読み取りできるようにします。

> **Note**
>
> IDT は、コンテキストからのデバイスアクセス情報を使用して確立されたデバイスへの直接接続を管理しないため、テストケース実行可能ファイルでは、デバイスとやり取り用のこれらの API コマンドを使用することをお勧めします。ただし、これらのコマンドがテストケー

スの要件を満たしていない場合は、IDT コンテキストからデバイスアクセス情報を取得し、この情報を使用してテストスイートからデバイスに直接接続できます。
直接接続するには、テスト対象デバイスとリソースデバイスそれぞれの `device.connectivity` フィールドと `resource.devices.connectivity` フィールドの情報を取得します。IDT コンテキスト使用の詳細については、IDT コンテキストを使用する を参照してください。

IDT とのやり取り

次のコマンドを使用すると、テストスイートが IDT と通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および `GetContextString`

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`SendResult`

テストスイートがテストケースの結果を IDT にレポートできるようにします。このコマンドは、テストスイートの各テストケースの最後に呼び出す必要があります。

ホストとのやり取り

次のコマンドを使用すると、テストスイートがホストマシンと通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および `GetContextString`

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`ExecuteOnHost`

テストスイートがローカルマシン上でコマンドを実行できるようにし、IDT がテストケース実行可能ファイルのライフサイクルを管理できるようにします。

IDT CLI コマンドを有効にする

`run-suite` コマンド IDT CLI には、テストの実行者がテスト実行をカスタマイズするためのいくつかのオプションがあります。テストの実行者がこれらのオプションを使用してカスタムテストスイートを実行できるようにするには、IDT CLI のサポートを実装します。サポートを実装しなくてもテストは実行できますが、一部の CLI オプションは正しく機能しません。理想的なカスタマーエクスペリエンスを提供するために、IDT CLI で `run-suite` コマンドの次の引数のサポートを実装することをお勧めします。

`timeout-multiplier`

テストの実行中にすべてのタイムアウトに適用される 1.0 より大きい値を指定します。

テストの実行者は、この引数を使用して、実行するテストケースのタイムアウトを増やすことができます。テストの実行者が `run-suite` コマンドにこの引数を指定すると、IDT はこの値を使用して IDT_TEST_TIMEOUT 環境変数の値を計算し、IDT コンテキストの `config.timeoutMultiplier` フィールドを設定します。この引数をサポートするには、以下の手順を実行する必要があります。

- `test.json` ファイルのタイムアウト値を直接使用する代わりに、IDT_TEST_TIMEOUT 環境変数を読み取り、正しく計算されたタイムアウト値を取得します。
- IDT コンテキストから `config.timeoutMultiplier` 値を取得し、長時間実行されるタイムアウトに適用します。

タイムアウトイベントによる早期終了の詳細については、終了動作を指定するを参照してください。

`stop-on-first-failure`

障害が発生した場合に、IDT がすべてのテスト実行を停止するように指定します。

テストの実行者がこの引数を `run-suite` コマンドを指定すると、IDT は障害が発生するとすぐにテストの実行を停止します。ただし、テストケースが並行して実行されている場合、この設定によって予期しない結果につながる可能性があります。このサポートを実装するには、テストロジックを使用して、IDT がこのイベントに遭遇した場合に、実行中のすべてのテストケースに対して、実行を停止し、一時リソースをクリーンアップし、テスト結果を IDT にレポートするように指示します。障害発生時の早期終了の詳細については、終了動作を指定するを参照してください。

`group-id` および `test-id`

IDT が選択されたテストグループまたはテストケースのみを実行するように指定します。

テストの実行者は、run-suite コマンドでこれらの引数を使用して、以下のテスト実行可能ファイルの動作を指定できます。

- 指定されたテストスイート内のすべてのテストグループを実行する。
- 指定されたテストグループ内から選択したテストを実行する。

これらの引数をサポートするには、テストスイート用のステートマシンに、自分のステートマシンの RunTask ステートおよび Choice ステートのセットが含まれている必要があります。カスタムステートマシンを使用しない場合は、デフォルトの IDT ステートマシンに必要なステートが含まれているため、追加のアクションを行う必要はありません。ただし、カスタムステートマシンを使用している場合は、サンプルとして ステートマシンの例: ユーザーが選択したテストグループを実行する を使用して、自分のステートマシンに必要なステートを追加してください。

IDT CLI コマンドの詳細については、カスタムテストスイートのデバッグと実行を参照してください。

イベントログの書き込み

テストの実行中は、イベントログとエラーメッセージをコンソールに書き込むために stdout と stderr にデータを送信します。コンソールメッセージの形式の詳細については、コンソールメッセージの形式を参照してください。

IDT がテストスイートの実行を終了すると、この情報は *<devicetester-extract-location>*/results/*<execution-id>*/logs フォルダにある test_manager.log ファイルでも利用可能になります。

各テストケースは、テスト実行のログ (テスト対象デバイスのログを含む) を *<device-tester-extract-location>*/results/*execution-id*/logs フォルダにある *<group-id>*_*<test-id>* ファイルに書き込むように設定できます。これを行うには、testData.logFilePath クエリを使用して IDT コンテキストからログファイルへのパスを取得し、そのパスにファイルを作成し、必要なコンテンツをそのファイルに書き込みます。IDT は、実行中のテストケースに基づいてこのパスを自動的に更新します。テストケースのログファイルを作成しないことを選択すると、そのテストケースのファイルは生成されません。

また、必要に応じて *<device-tester-extract-location>*/logs フォルダに追加のログファイルを作成するようにテキスト実行可能ファイルをセットアップすることもできます。ファイルが上書きされないように、ログファイル名に一意のプレフィックスを指定することをお勧めします。

IDT に結果をレポートする

IDT は、テスト結果を `awsiotdevicetester_report.xml` ファイルと *suite-name*`_report.xml` ファイルに書き込みます。これらのレポートファイルは、*<device-tester-extract-location>*`/results/`*<execution-id>*`/` にあります。両レポートとも、テストスイート実行の結果をキャプチャします。IDT がこれらのレポートに使用するスキーマの詳細については、IDT テストの結果とログを確認するを参照してください。

suite-name`_report.xml` ファイルのコンテンツを取得するには、`SendResult` コマンドを使用して、テスト実行が終了する前に、テスト結果を IDT にレポートする必要があります。IDT は、テスト結果を見つけられない場合、テストケースのエラーを発行します。次の Python の抜粋は、テスト結果を IDT に送信するコマンドを示しています。

```
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

API を使用して結果をレポートしない場合、IDT はテストアーティファクトフォルダでテスト結果を検索します。このフォルダのパスは、IDT コンテキストの `testData.testArtifactsPath` フィールドに格納されています。このフォルダで、IDT は、アルファベット順にソートされた最初の XML ファイルをテスト結果として使用します。

テストロジックが JUnit XML 結果を生成する場合は、結果を解析してから API を使用して IDT に送信する代わりに、アーティファクトフォルダ内の XML ファイルにテスト結果を書き込んで、直接 IDT に提供することができます。

この方法を使用する場合は、テストロジックによってテスト結果が正確に要約されていること、および *suite-name*`_report.xml` ファイルと同じ形式で結果ファイルがフォーマットされていることを確認してください。IDT は、次の例外を除き、提供されたデータの検証を実行しません。

- IDT は `testsuites` タグのすべてのプロパティを無視します。代わりに、レポートされた他のテストグループ結果からタグのプロパティを計算します。
- `testsuite` 内に少なくとも 1 つの `testsuites` タグが存在する必要があります。

IDT はすべてのテストケースで同じアーティファクトフォルダを使用し、テスト実行の終了後、次のテスト実行までに結果ファイルを削除しないため、この方法を使用すると、IDT が正しくないファイルを読み取った場合に、誤ったレポートが行われる可能性もあります。IDT が適切な結果を読み取れるように、すべてのテストケースで生成される XML 結果ファイルに同じ名前を使用して、各テストケースの結果を上書きすることをお勧めします。テストスイートのレポート作成に複合的なアプロー

チ (一部のテストケースには XML 結果ファイルを使用し、他のテストケースには API を使用して結果を送信する) を使用することもできますが、このアプローチはお勧めしません。

終了動作を指定する

テキスト実行可能ファイルは、テストケースが障害やエラー結果をレポートした場合でも、常に終了コード 0 で終了するように設定します。ゼロ以外の終了コードは、テストケースが実行されなかったこと、またはテストケース実行可能ファイルが結果を IDT に通知できなかったことを示す場合にのみ使用します。IDT は、0 以外の終了コードを受信すると、テスト実行を妨げるエラーが発生したとしてテストケースをマークします。

IDT は、以下に示すイベントが発生すると、終了前にテストケースに実行の停止を要求 (または想定) することがあります。以下の情報を使用して、テストケースから以下の各イベントを検出するようにテストケース実行可能ファイルを設定します。

タイムアウト

テストケースが、`test.json` ファイルで指定されたタイムアウト値よりも長く実行されたときに発生します。テストの実行者が `timeout-multiplier` 引数を使用してタイムアウト乗数を指定すると、IDT はこの乗数を使用してタイムアウト値を計算します。

このイベントを検出するには、IDT_TEST_TIMEOUT 環境変数を使用します。テストの実行者がテストを起動すると、IDT は IDT_TEST_TIMEOUT 環境変数の値を計算されたタイムアウト値 (秒単位) に設定し、その変数をテストケース実行可能ファイルに渡します。この変数の値を読み取って適切なタイマーを設定します。

割り込み

テストの実行者が IDT に割り込むと発生します。例えば、Ctrl+C を押した時です。

ターミナルはすべての子プロセスに通知を伝播するため、割り込み通知を検出する通知ハンドラをテストケースに簡単に設定できます。

または、API を定期的にポーリングして、`PollForNotifications` API 応答の `CancellationRequested` ブール値をチェックできます。IDT は割り込み通知を受信すると、`CancellationRequested` ブールの値を `true` に設定します。

最初の失敗時に停止する

現在のテストケースと並行して実行中のテストケースが失敗し、テストの実行者が `stop-on-first-failure` 引数を使用して、障害の発生時に IDT が実行を停止するように設定しているときに発生します。

このイベントを検出するには、`PollForNotifications` API を定期的にポーリングして、API レスポンスの `CancellationRequested` ブールの値をチェックします。IDT は、最初の障害発生時に停止するように設定されている場合、障害に遭遇すると、`CancellationRequested` ブールの値を `true` に設定します。

これらのいずれかのイベントが発生すると、IDT は現在実行中のテストケースの実行が終了するまで 5 分間待機します。実行中のすべてのテストケースが 5 分以内に終了しない場合、IDT は各プロセスを強制的に停止させます。IDT は、プロセスの終了前にテスト結果を受け取らなかった場合、テストケースをタイムアウトしたとしてマークします。ベストプラクティスとして、いずれかのイベントが発生したときは、テストケースが以下のアクションを実行するようにします。

1. 通常のテストロジックの実行を停止する。
2. テスト対象デバイスのテストアーティファクトなど、すべての一時的なリソースをクリーンアップする。
3. テスト結果 (テストの失敗やエラーなど) を IDT にレポートする。
4. 終了する。

IDT コンテキストを使用する

IDT がテストスイートを実行するとき、テストスイートは、各テストの実行方法の決定に使用できる一連のデータにアクセスできます。このデータは IDT コンテキストと呼ばれます。例えば、テストの実行者によって `userdata.json` ファイルに提供されるユーザーデータ設定は、IDT コンテキスト内でテストスイートに提供されます。

IDT コンテキストは、読み取り専用の JSON ドキュメントと考えることができます。テストスイートは、オブジェクト、配列、数値などの標準 JSON データ型を使用して、コンテキストからデータを取得することや、コンテキストにデータを書き込むことができます。

コンテキストスキーマ

IDT コンテキストは次の形式を使用します。

```
{
    "config": {
        <config-json-content>
        "timeoutMultiplier": timeout-multiplier
    },
```

```
    "device": {
        <device-json-device-element>
    },
    "devicePool": {
        <device-json-pool-element>
    },
    "resource": {
        "devices": [
            {
                <resource-json-device-element>
                "name": "<resource-name>"
            }
        ]
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>"
        },
        "logFilePath": "/path/to/log/file"
    },
    "userData": {
        <userdata-json-content>
    }
}
```

config

config.json ファイル からの情報。config フィールドには、次の追加フィールドも含まれます。

config.timeoutMultiplier

テストスイートによって使用される任意のタイムアウト値の乗数。この値は、IDT CLI からテストの実行者によって指定されます。デフォルト値は 1 です。

device

テスト実行用に選択されたデバイスに関する情報。この情報は、選択されたデバイスの device.json ファイル の devices 配列要素に相当します。

`devicePool`

テスト実行用に選択されたデバイスプールに関する情報。この情報は、選択されたデバイスプールの `device.json` ファイルに定義されている最上位レベルのデバイスプール配列要素に相当します。

`resource`

`resource.json` ファイルからのリソースデバイスに関する情報。

`resource.devices`

この情報は、`devices` ファイルに定義されている `resource.json` 配列に相当します。各 `devices` 要素には、以下の追加フィールドが含まれています。

`resource.device.name`

リソースデバイスの名前。この値は、`test.json` ファイルで `requiredResource.name` 値に設定されます。

`testData.awsCredentials`

AWS クラウドに接続するためにテストで使用される AWS 認証情報。この情報は、`config.json` ファイルから取得されます。

`testData.logFilePath`

テストケースがログメッセージを書き込むログファイルへのパス。このファイルは、存在しない場合、テストスイートによって作成されます。

`userData`

テストの実行者によって `userdata.json` ファイル に提供された情報。

コンテキスト内のデータにアクセスする

コンテキストは、JSONPath 表記を使用して JSON ファイルからクエリすることも、`GetContextValue` および `GetContextString` API を使用してテキスト実行可能ファイルからクエリすることもできます。IDT コンテキストにアクセスするための JSONPath 文字列の構文は、次のように異なります。

- `suite.json` および `test.json` では、`{{query}}` を使用します。つまり、式を開始するためにルート要素 `$.` を使用しません。
- `statemachine.json` では `{{$.query}}` を使用します。

- API コマンドでは、コマンドに応じて *query* または {{$.*query*}} を使用します。詳細については、SDK のインラインドキュメントを参照してください。

次の表に、一般的な JSONPath 式の演算子を示します。

Operator	Description
`$`	The root element. Because the top-level context value for IDT is an object, you will typically use `$.` to start your queries.
`.childName`	Accesses the child element with name `childName` from an object. If applied to an array, yields a new array with this operator applied to each element. The element name is case sensitive. For example, the query to access the `awsRegion` value in the `config` object is `$.config.awsRegion` .
`[start:end]`	Filters elements from an array, retrieving items beginning from the ## index and going up to the `end` index, both inclusive.
`[index1, index2, ... , indexN]`	Filters elements from an array, retrieving items from only the specified indices.
`[?(expr)]`	Filters elements from an array using the `expr` expression. This expression must evaluate to a boolean value.

フィルター式を作成するには、次の構文を使用します。

```
<jsonpath> | <value> operator <jsonpath> | <value>
```

この構文の説明は次のとおりです。

- `jsonpath` は、標準 JSON 構文を使用する JSONPath です。

- `value` は、標準 JSON 構文を使用するカスタム値です。
- `operator` は、以下のいずれかの演算子です。
  - < (未満)
  - <= (以下)
  - == (等しい)

    式内の JSONPath または値が配列、ブール値、またはオブジェクト値である場合は、これがユーザーに使用可能な唯一の二項演算子です。
  - >= (以上)
  - > (次より大きい)
  - =~ (正規表現の一致)。この演算子をフィルター式で使用するには、式の左側の JSONPath または値が文字列に評価される必要があり、右側が RE2 構文 に従ったパターン値である必要があります。

{{*query*}} 形式の JSONPath クエリは、プレースホルダ文字列として、`test.json` ファイルの `args` および `environmentVariables` フィールド内と、`suite.json` ファイルの `environmentVariables` フィールド内で使用できます。IDT はコンテキスト検索を実行し、クエリの評価値をフィールドに入力します。例えば、`suite.json` ファイルでは、プレースホルダー文字列を使用して、各テストケースとともに変化する環境変数の値を指定できます。IDT は、環境変数に各テストケースの正しい値を入力します。ただし、`test.json` ファイルおよび `suite.json` ファイルでプレースホルダー文字列を使用する場合は、クエリに次の考慮事項が適用されます。

- クエリに含まれる各 `devicePool` キーは、すべて小文字にする必要があります。つまり、代わりに `devicepool` を使用します。
- 配列には、文字列の配列のみを使用できます。さらに、配列は非標準の `item1, item2,...,itemN` の形式を使用します。配列は、要素が 1 つしか含まれていない場合、`item` としてシリアル化され、文字列フィールドと区別がつかなくなります。
- プレースホルダーを使用してコンテキストからオブジェクトを取得することはできません。

これらの事項を考慮して、テストロジックのコンテキストへのアクセスには、`test.json` ファイルおよび `suite.json` ファイルのプレースホルダー文字列ではなく、可能な限り API を使用することをお勧めします。ただし、環境変数として設定する単一の文字列を取得するときは、JSONPath プレースホルダーを使用する方が便利な場合があります。

テストの実行者向けの設定の設定

カスタムテストスイートを実行するには、テストの実行者は、実行するテストスイートに基づいて設定を構成する必要があります。設定は、*<device-tester-extract-location>*/configs/ フォルダにある JSON 設定ファイルテンプレートに基づいて指定します。必要に応じて、テストランナーは IDT が AWS クラウドへの接続に使用する AWS 認証情報も設定する必要があります。

テストを作成するユーザーは、テストスイートをデバッグするために、以下に示すファイルの設定が必要になります。また、テストスイートを実行するために必要な以下の設定を設定できるように、テストの実行者に指示を提供する必要があります。

device.json の設定

device.json ファイルには、テストが実行されるデバイスに関する情報 (IP アドレス、ログイン情報、オペレーティングシステム、CPU アーキテクチャなど) が含まれています。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート device.json ファイルを使用してこの情報を指定できます。

```
[
    {
        "id": "<pool-id>",
        "sku": "<pool-sku>",
        "features": [
            {
                "name": "<feature-name>",
                "value": "<feature-value>",
                "configs": [
                    {
                        "name": "<config-name>",
                        "value": "<config-value>"
                    }
                ],
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
```

```
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`sku`

テスト対象デバイスを一意に識別する英数字の値。SKU は、認定されたデバイスの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog にボードを一覧表示する場合、ここで指定する SKU は、一覧表示プロセスで使用する SKU と一致する必要があります。

`features`

オプション。デバイスでサポートされている機能を含む配列。デバイス機能は、テストスイートに設定するユーザー定義の値です。テストの実行者に、`device.json` ファイルに含める機能名および値に関する情報を提供する必要があります。例えば、他のデバイスの MQTT サーバーとして機能するデバイスをテストする場合は、`MQTT_QOS` という名前の機能に対する特定のサポートレベルを検証するようにテストロジックを設定します。テストの実行者は、この機能名を指定し、デバイスによってサポートされる QOS レベルにその機能値を設定します。指定された情報は、`devicePool.features` クエリを使用して IDT コンテキストから、または `pool.features` クエリを使用してステートマシンコンテキストから取得できます。

`features.name`

機能の名前。

`features.value`

サポートされている機能値。

`features.configs`

機能の構成設定 (必要な場合)。

`features.config.name`

構成設定の名前。

`features.config.value`

サポートされている設定値。

`devices`

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイス用の `ssh` および `uart` と、Docker コンテナ用の `docker` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.port`

オプション。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth`

接続の認証情報。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth.method`

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- `pki`
- `password`

`connectivity.auth.credentials`

認証に使用される認証情報。

`connectivity.auth.credentials.password`

テスト中のデバイスにサインインするためのパスワード。

この値は、`connectivity.auth.method` が `password` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.privKeyPath`

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、`connectivity.auth.method` が `pki` に設定されている場合にのみ適用されます。

`connectivity.auth.credentials.user`

テスト対象デバイスにサインインするためのユーザー名。

`connectivity.serialPort`

オプション。デバイスが接続されているシリアルポート。

このプロパティは、`connectivity.protocol` が `uart` に設定されている場合にのみ適用されます。

`connectivity.containerId`

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、`connectivity.protocol` が `docker` に設定されている場合にのみ適用されます。

`connectivity.containerUser`

オプション。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `docker` に設定されている場合にのみ適用されます。

> **Note**
>
> テストの実行者がテストに対して誤ったデバイス接続を構成しているかどうかを確認するには、ステートマシンコンテキストから

pool.Devices[0].Connectivity.Protocol を取得し、この値を Choice ステート内の予想値と比較します。正しくないプロトコルが使用されている場合は、LogMessage ステートを使用してメッセージを出力し、Fail ステートに移行します。
または、エラー処理コードを使用して、誤ったデバイスタイプによるテスト失敗をレポートすることもできます。

(オプション) userdata.json の設定

userdata.json ファイルには、device.json ファイルには指定されていない、テストスイートに必要とされる追加情報が含まれています。このファイルの形式は、テストスイートに定義されている userdata_scheme.json ファイルによって異なります。テストを作成するユーザーは、作成したテストスイートを実行するユーザーにこの情報を提供してください。

(オプション) resource.json の設定

resource.json ファイルには、リソースデバイスとして使用されるすべてのデバイスに関する情報が含まれています。リソースデバイスは、テスト対象のデバイスの特定の機能をテストするために必要なデバイスです。例えば、デバイスの Bluetooth 機能をテストするには、リソースデバイスを使用して、デバイスがリソースデバイスに正常に接続できるかどうかをテストできます。リソースデバイスはオプションで、必要な数だけリソースデバイスを要求できます。テストを作成するユーザーは、test.json ファイル を使用して、テストに必要なリソースデバイスの機能を定義します。テストの実行者は、resource.json ファイルを使用して、必要な機能を持つリソースデバイスのプールを指定します。作成したテストスイートを実行するユーザーに、以下の情報を提供してください。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート resource.json ファイルを使用してこの情報を指定できます。

```
[
    {
        "id": "<pool-id>",
        "features": [
            {
                "name": "<feature-name>",
                "version": "<feature-value>",
                "jobSlots": <job-slots>
            }
        ],
        "devices": [
            {
```

```
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`id`

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

`features`

オプション。デバイスでサポートされている機能を含む配列。このフィールドに必要な情報は、テストスイートの test.json ファイル に定義されています。この情報によって、実行するテスト

と、テストの実行方法が決まります。テストスイートに機能が必要ない場合は、このフィールドは必須ではありません。

`features.name`

機能の名前。

`features.version`

機能バージョン。

`features.jobSlots`

デバイスを同時に使用できるテストの数を示すための設定。デフォルト値は 1 です。

`devices`

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイス用の `ssh` および `uart` と、Docker コンテナ用の `docker` のみです。

`connectivity.ip`

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.port`

オプション。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

`connectivity.auth`

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.password

テスト中のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

connectivity.auth.credentials.privKeyPath

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

connectivity.auth.credentials.user

テスト対象デバイスにサインインするためのユーザー名。

connectivity.serialPort

オプション。デバイスが接続されているシリアルポート。

このプロパティは、connectivity.protocol が uart に設定されている場合にのみ適用されます。

connectivity.containerId

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、connectivity.protocol が docker に設定されている場合にのみ適用されます。

`connectivity.containerUser`

オプション。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、`connectivity.protocol` が `docker` に設定されている場合にのみ適用されます。

(オプション) config.json の設定

`config.json` ファイルには、IDT 向けの設定情報が含まれています。通常、テストランナーは、IDT、およびオプションで AWS リージョンの AWS ユーザー認証情報を提供する場合を除き、このファイルを変更する必要はありません。必要なアクセス許可を持つ認証情報が提供されている場合、 AWS IoT Device Tester AWS は使用状況メトリクスを収集して送信します AWS。これはオプトイン機能で、IDT 機能を改善するために使用されます。詳細については、「IDT 使用状況メトリクス」を参照してください。

テストランナーは、次のいずれかの方法で AWS 認証情報を設定できます。

- 認証情報ファイル

  IDT では、 AWS CLIと同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

  認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

  - macOS、Linux: `~/.aws/credentials`
  - Windows: `C:\Users\UserName\.aws\credentials`

- 環境変数

  環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッション中に定義された変数は、そのセッションの終了後は使用できません。IDT は `AWS_ACCESS_KEY_ID`および `AWS_SECRET_ACCESS_KEY`環境変数を使用して AWS 認証情報を保存できます

  これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
```

```
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

Windows でこれらの変数を設定するには、set を使用します。

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

IDT の AWS 認証情報を設定するには、テストランナーは *<device-tester-extract-location>*/configs/フォルダにある config.jsonファイルの authセクションを編集します。

```
{
    "log": {
        "location": "logs"
    },
    "configFiles": {
        "root": "configs",
        "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
        "method": "file | environment",
        "credentials": {
            "profile": "<profile-name>"
        }
    }
}
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

> **Note**
>
> このファイル内のすべてのパスは、*<device-tester-extract-location>* に関連して定義されています。

`log.location`

<device-tester-extract-location> のログフォルダへのパス。

`configFiles.root`

設定ファイルが含まれるフォルダへのパス。

`configFiles.device`

`device.json` ファイルへのパス。

`testPath`

テストスイートが含まれるフォルダへのパス。

`reportPath`

IDT がテストスイートを実行した後にテスト結果が含まれるフォルダへのパス。

`awsRegion`

オプション。テストスイートが使用する AWS リージョン。設定されない場合、テストスイートは各テストスイートに指定されているデフォルトのリージョンを使用します。

`auth.method`

IDT が AWS 認証情報を取得するために使用するメソッド。サポートされる値は、`file` (認証情報ファイルから認証情報を取得) と `environment` (環境変数を使用して認証情報を取得) です。

`auth.credentials.profile`

認証情報ファイルから使用する認証情報プロファイル。このプロパティは、`auth.method` が `file` に設定されている場合にのみ適用されます。

カスタムテストスイートのデバッグと実行

必要な設定 を終了すると、IDT はテストスイートを実行することができます。完全なテストスイートの実行時間は、ハードウェアとテストスイートの設定によって異なります。参考までに、Raspberry Pi 3B AWS IoT Greengrass で完全な認定テストスイートを完了するには約 30 分かかります。 3B

テストスイートの作成中に、IDT を使用してテストスイートをデバッグモードで実行すると、テストスイートを実行する前やテストの実行者に提供する前に、コードをチェックすることができます。

IDT をデバッグモードで実行する

テストスイートは、IDT に依存してデバイスとやり取りし、コンテキストを提供し、結果を受け取るため、IDT と通信しないと、IDE でテストスイートを簡単にデバッグすることはできません。そのために、IDT CLI は IDT をデバッグモードで実行できるようにする `debug-test-suite` コマンドを提供します。`debug-test-suite` で使用可能なオプションを表示するには、次のコマンドを実行します。

```
devicetester_[linux | mac | win_x86-64] debug-test-suite -h
```

IDT をデバッグモードで実行する場合、IDT は実際にテストスイートを起動したり、ステートマシンを実行したりしません。代わりに、IDE と通信して IDE で実行されているテストスイートからのリクエストに応答し、コンソールにログを出力します。IDT はタイムアウトせず、手動で中断されるまで待機してから終了します。デバッグモードでは、IDT はステートマシンを実行せず、レポートファイルも生成しません。テストスイートをデバッグするには、通常 IDT が設定 JSON ファイルから取得する情報を、IDE を使用して提供する必要があります。以下の情報を提供してください。

- 各テストの環境変数と引数。IDT はこの情報を `test.json` または `suite.json` から読み込みません。
- リソースデバイスを選択するための引数。IDT はこの情報を `test.json` から読み込みません。

テストスイートをデバッグするには、次の手順を実行します。

1. テストスイートの実行に必要な構成設定ファイルを作成します。例えば、テストスイートが `device.json`、`resource.json`、および `user data.json` を必要とする場合は、必要に応じてこれらすべてを設定してください。
2. 次のコマンドを実行して IDT をデバッグモードにし、テストの実行に必要なデバイスを選択します。

   ```
   devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
   ```

   このコマンドを実行すると、IDT はテストスイートからのリクエストを待機し、それらのリクエストに応答します。IDT は、IDT クライアント SDK がケースを処理するために必要な環境変数も生成します。
3. IDE で、`run` または `debug` 設定を使用して次の手順を実行します。

   a. IDT で生成された環境変数の値を設定します。

b. `test.json` ファイルと `suite.json` ファイルに指定したすべての環境変数または引数の値を設定します。

c. 必要に応じてブレークポイントを設定します。

4. IDE でテストスイートを実行します。

   テストスイートは、必要に応じて何度でもデバッグして再実行できます。IDT はデバッグモードではタイムアウトしません。

5. デバッグが完了したら、IDT を中断してデバッグモードを終了します。

テストを実行する IDT CLI コマンド

次のセクションでは、IDT CLI コマンドについて説明します。

IDT v4.0.0

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のテストスイート内のグループを一覧表示します。

`list-suites`

使用可能なテストスイートを一覧表示します。

`list-supported-products`

IDT のバージョンでサポートされている製品、この場合は AWS IoT Greengrass バージョン、および現在の IDT AWS IoT Greengrass バージョンで利用可能な認定テストスイートバージョンを一覧表示します。

`list-test-cases`

指定したテストグループのテストケースを一覧表示します。次のオプションがサポートされています。

- `group-id`。検索するテストグループ。このオプションは必須で、1 つのグループを指定する必要があります。

`run-suite`

デバイスプールに対してテストスイートを実行します。以下に、一般的に使用されるオプションの一部を示します。

- `suite-id`。実行するテストスイートのバージョン。指定しない場合、IDT は `tests` フォルダにある最新バージョンを使用します。
- `group-id`。実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。
- `test-id`。実行するテストケース (カンマ区切りリストとして)。指定した場合は、`group-id` は 1 つのグループを指定する必要があります。
- `pool-id`。テストするデバイスプール。`device.json` ファイルに複数のデバイスプールが定義されている場合、テストの実行者は 1 つのプールを指定する必要があります。
- `timeout-multiplier`。テスト用の `test.json` ファイルに指定されているテスト実行タイムアウトを、ユーザー定義乗数を使用して変更するように IDT を設定します。
- `stop-on-first-failure`。最初に障害が発生した時点で実行を停止するように IDT を設定します。指定されたテストグループをデバッグするには、このオプションを `group-id` とともに使用する必要があります。
- `userdata`。テストスイートの実行に必要なユーザーデータ情報を含むファイルを設定します。テストスイートの `suite.json` ファイルで、`userdataRequired` が true に設定されている場合にのみ必要です。

`run-suite` オプションの詳細については、次の `help` オプションを使用してください。

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

`debug-test-suite`

デバッグモードでテストスイートを実行します。詳細については、「IDT をデバッグモードで実行する」を参照してください。

IDT テストの結果とログを確認する

このセクションでは、IDT がコンソールログとテストレポートを生成する形式について説明します。

コンソールメッセージの形式

AWS IoT Device Tester は、テストスイートを開始するときに、コンソールにメッセージを印刷するための標準形式を使用します。以下の抜粋は、IDT によって生成されるコンソールメッセージの例を示しています。

```
time="2000-01-02T03:04:05-07:00" level=info msg=Using suite: MyTestSuite_1.0.0
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージの大半は、次のフィールドで設定されます。

`time`

ログに記録されたイベントの完全な ISO 8601 タイムスタンプ。

`level`

ログに記録されたイベントのメッセージレベル。通常、ログに記録されるメッセージレベルは、`info`、`warn`、または `error` のいずれかです。IDT は、早期終了の原因となる予期されるイベントが発生した場合は、`fatal` または `panic` メッセージを発行します。

`msg`

ログに記録されたメッセージ。

`executionId`

現在の IDT プロセスの一意の ID 文字列。この ID は、個々の IDT 実行を区別するために使用されます。

テストスイートから生成されたコンソールメッセージは、テスト対象のデバイスとテストスイート、テストグループ、IDT が実行するテストケースに関する追加情報を提供します。次の抜粋は、テストスイートから生成されたコンソールメッセージの例を示しています。

```
time="2000-01-02T03:04:05-07:00" level=info msg=Hello world! suiteId=MyTestSuite
groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージのテストスイート固有の部分には、次のフィールドが含まれています。

`suiteId`

現在実行中のテストスイートの名前。

`groupId`

現在実行中のテストグループの ID。

`testCaseId`

現在実行中のテストケースの ID。

`deviceId`

現在のテストケースが使用しているテスト対象デバイスの ID。

IDT のテスト実行完了時にテストサマリーをコンソールに出力するには、ステートマシンに Report ステートを含める必要があります。テストサマリーには、テストスイート、実行された各グループのテスト結果、生成されたログファイルとレポートファイルの場所に関する情報が含まれています。次の例は、テストサマリーメッセージを示しています。

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
----------------------------------
Test Groups:
    GroupA:         PASSED
    GroupB:         FAILED
----------------------------------
Failed Tests:
    Group Name: GroupB
        Test Name: TestB1
            Reason: Something bad happened
----------------------------------
Path to IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

AWS IoT Device Tester レポートスキーマ

`awsiotdevicetester_report.xml` は、次の情報が含まれる署名済みレポートです。

- IDT バージョン。

- テストスイートのバージョン。
- レポートの署名に使用されるレポートの署名とキー。
- device.json ファイルで指定されているデバイス SKU とデバイスプール。
- テストされた製品のバージョンとデバイスの機能。
- テスト結果の概要の集計。この情報は、*suite-name*_report.xml ファイルに含まれている情報と同じです。

```
<apnreport>
    <awsiotdevicetesterversion>idt-version</awsiotdevicetesterversion>
    <testsuiteversion>test-suite-version</testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession>execution-id</testsession>
        <starttime>start-time</starttime>
        <endtime>end-time</endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="<feature-name>" value="supported | not-supported | <feature-
value>" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
            <feature name="<feature-name>" value="<feature-value>"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="<os-name>"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
</apnreport>
```

awsiotdevicetester_report.xml ファイルには、テスト対象の製品および一連のテストの実行後に検証された製品機能に関する情報を含む <awsproduct> タグが含まれています。

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

features

検証された機能です。required としてマークされている機能は、テストスイートがデバイスを検証するために必要です。次のスニペットは、この情報が awsiotdevicetester_report.xml ファイルで表示される方法を示します。

```
<feature name="ssh" value="supported" type="required"></feature>
```

optional としてマークされている機能は、検証に必須ではありません。次のスニペットは、オプションの機能を示しています。

```
<feature name="hsi" value="supported" type="optional"></feature>

<feature name="mqtt" value="not-supported" type="optional"></feature>
```

テストスイートのレポートスキーマ

*suite-name*_Result.xml レポートは JUnit XML 形式です。Jenkins、Bamboo などのように継続的な統合 (CI) と継続的なデプロイ (CD) のプラットフォームに統合することができます。このレポートには、テスト結果の概要の集計が含まれています。

```
<testsuites name="<suite-name> results" time="<run-duration>" tests="<number-of-test>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
    <testsuite name="<test-group-id>" package="" tests="<number-of-tests>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
        <!--success-->
```

```
        <testcase classname="<classname>" name="<name>" time="<run-duration>"/>
        <!--failure-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <failure type="<failure-type>">
                reason
            </failure>
        </testcase>
        <!--skipped-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <skipped>
                reason
            </skipped>
        </testcase>
        <!--error-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <error>
                reason
            </error>
        </testcase>
    </testsuite>
</testsuites>
```

`awsiotdevicetester_report.xml` と *`suite-name`*`_report.xml` 両方のレポートセクションには、実行されたテストとその結果が一覧表示されます。

最初の XML タグ `<testsuites>` には、テストの実行の概要が含まれています。例:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

`<testsuites>` タグで使用される属性

`name`

テストスイートの名前。

`time`

スイートの実行所要時間 (秒)。

`tests`

実行されたテストの数。

`failures`

実行されたテストのうち、合格しなかったものの数。

`errors`

IDT で実行できなかったテストの数。

`disabled`

この属性は使用されていないため無視できます。

テストに障害やエラーが発生した場合は、`<testsuites>` XML タグを確認することで、障害の生じたテストを特定できます。`<testsuites>` タグ内の `<testsuite>` XML タグは、テストグループのテスト結果の要約を示します。例:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

形式は `<testsuites>` タグと似ていますが、使用されていないため無視できる `skipped` という属性があります。各 `<testsuite>` XML タグ内には、テストグループの実行されたテスト別の `<testcase>` タグがあります。例:

```
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>>
```

`<testcase>` タグで使用される属性

`name`

テストの名前。

`attempts`

IDT でテストケースを実行した回数。

テストに障害やエラーが発生した場合、`<failure>` タグまたは `<error>` タグがトラブルシューティングのための情報とともに `<testcase>` タグに追加されます。例:

```
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
```

```
</testcase>
```

IDT 使用状況メトリクス

必要なアクセス許可を持つ AWS 認証情報を提供すると、 AWS IoT Device Tester は使用状況メトリクスを収集して送信します AWS。これはオプトイン機能で、IDT 機能を改善するために使用されます。IDT は次のような情報を収集します。

- AWS アカウント IDT の実行に使用される ID
- テストの実行に使用される IDT CLI コマンド
- 実行されるテストスイート
- *<device-tester-extract-location>* フォルダにあるテストスイート
- デバイスプール内に設定されているデバイスの数
- テストケース名と実行時間
- テストに合格したか、失敗したか、エラーが発生したか、スキップされたかなどのテスト結果情報
- テストされた製品の機能
- 予期せぬ終了、早期終了などの IDT 終了動作

IDT が送信するすべての情報は、*<device-tester-extract-location>*/results/*<execution-id>*/ フォルダの metrics.log ファイルにもログが記録されます。ログファイルを表示すると、テスト実行中に収集された情報を確認できます。このファイルは、使用状況メトリックを収集することを選択した場合にのみ生成されます。

メトリクスの収集を無効にするために、追加のアクションを実行する必要はありません。 AWS 認証情報を保存せず、 AWS 認証情報を保存している場合は、それらにアクセスするように config.json ファイルを設定しないでください。

AWS 認証情報を設定する

をまだ作成していない場合は AWS アカウント、作成する必要があります。が既にある場合は AWS アカウント、IDT が AWS ユーザーに代わって に使用状況メトリクスを送信できるようにするアカウントに必要なアクセス許可を設定するだけです。

ステップ 1: を作成する AWS アカウント

このステップでは、 AWS アカウントを作成して設定します。が既にある場合は AWS アカウント、「」に進みますthe section called “ステップ 2: IDT 用のアクセス許可を設定する”。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 のセキュリティを確保し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

ステップ 2: IDT 用のアクセス許可を設定する

このステップでは、IDT がテストを実行して IDT 使用状況データを収集するために使用するアクセス許可を設定します。 AWS Management Console または AWS Command Line Interface（ AWS

CLI) を使用して IDT の IAM ポリシーとユーザーを作成し、ユーザーにポリシーをアタッチできます。

- IDT 用のアクセス許可を設定するには (コンソール)
- IDT 用のアクセス許可を設定するには (AWS CLI)

IDT 用のアクセス許可を設定するには (コンソール)

コンソールを使用して IDT for AWS IoT Greengrass用のアクセス許可を設定するには、次のステップに従ってください。

1. IAM コンソールにサインインします。
2. 特定のアクセス許可を持つロールを作成するためのアクセス許可を付与するカスタマー管理ポリシーを作成します。
   a. ナビゲーションペインで ポリシーを選択してから ポリシーの作成を選択します。
   b. JSON タブで、プレースホルダーコンテンツを以下のポリシーに置き換えます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}
```

   c. [Next: Tags (次へ: タグ)] を選択します。
   d. [次へ: レビュー] を選択します。
   e. [名前] に**IDTUsageMetricsIAMPermissions**と入力してください。[概要] で、ポリシーによって付与されたアクセス許可を確認します。
   f. [Create policy] (ポリシーの作成) を選択します。
3. IAM ユーザーを作成し、ユーザーにアクセス許可をアタッチします。

a. IAM ユーザーを作成します。IAM ユーザーガイドの IAM ユーザーの作成 (コンソール) のステップ 1 ~ 5 に従います。IAM ユーザーを作成済みの場合は、次のステップに進んでください。

b. アクセス許可を IAM ユーザーにアタッチします。

   i. [Set permissions] (許可を設定) ページで、[Attach existing policies directly] (既存のポリシーを直接アタッチする) を選択します。

   ii. 前のステップで作成した IDTGreengrassIAMPermissions ポリシーを検索します。チェックボックスをオンにします。

c. [Next: Tags] (次へ: タグ) を選択します。

d. [Next: Review] (次へ: レビュー) を選択して、選択内容の概要を表示します。

e. [ユーザーの作成] を選択します。

f. ユーザーのアクセスキー (アクセスキー ID とシークレットアクセスキー) を表示するには、パスワードとアクセスキーの横にある [Show (表示)] を選択します。アクセスキーを保存するには、[Download .csv] を選択し、安全な場所にファイルを保存します。後でこの情報を使用して、 AWS 認証情報ファイルを設定します。

IDT 用のアクセス許可を設定するには (AWS CLI)

を使用して IDT for のアクセス許可を設定するには AWS CLI 、次の手順に従います AWS IoT Greengrass。コンソールでアクセス許可をすでに設定している場合は、「the section called “IDT テストを実行するようにデバイスを設定する”」または「the section called “オプション: Docker コンテナの設定”」に進みます。

1. コンピュータに をインストールし、まだインストールされていない場合 AWS CLI は設定します。AWS Command Line Interface ユーザーガイドの AWS CLIのインストールのステップに従います。

> **Note**
>
> AWS CLI は、コマンドラインシェルから AWS サービスとやり取りするために使用できるオープンソースツールです。

2. IDT と AWS IoT Greengrass ロールを管理するアクセス許可を付与する次のカスタマー管理ポリシーを作成します。

   Linux, macOS, or Unix

```
aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}'
```

   Windows command prompt

```
aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
document
                                          '{\"Version\": \"2012-10-17\",
 \"Statement\": [{\"Effect\": \"Allow\", \"Action\": [\"iot-device-
tester:SendMetrics\"], \"Resource": \"*\"}]}'
```

> **Note**
>
> このステップには、Linux、macOS、または Unix のターミナルコマンドとは異なる JSON 構文を使用するため、Windows コマンドプロンプトの例が含まれています。

3. IAM ユーザーを作成し、IDT for AWS IoT Greengrassに必要なアクセス許可をアタッチします。

   a. IAM ユーザーを作成します。

```
aws iam create-user --user-name user-name
```

   b. 作成した `IDTUsageMetricsIAMPermissions` ポリシーを IAM ユーザーにアタッチします。*user-name* を IAM ユーザー名に置き換え、コマンドの *<account-id>* を AWS アカウントの ID に置き換えます。

```
aws iam attach-user-policy --user-name user-name --policy-arn
 arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
```

4. ユーザーのシークレットアクセスキーを作成します。

```
aws iam create-access-key --user-name user-name
```

この出力は安全な場所に保存してください。後でこの情報を使用して、 AWS 認証情報ファイルを設定します。

IDT に AWS 認証情報を提供する

IDT が AWS 認証情報にアクセスしてメトリクスを送信できるようにするには AWS、以下を実行します。

1. IAM ユーザーの AWS 認証情報を環境変数として、または認証情報ファイルに保存します。

   a. 環境変数を使用するには、次のコマンドを実行します。

   ```
   AWS_ACCESS_KEY_ID=access-key
   AWS_SECRET_ACCESS_KEY=secret-access-key
   ```

   b. 認証情報ファイルを使用するには、`.aws/credentials file:` に次の情報を追加します。

   ```
   [profile-name]
   aws_access_key_id=access-key
   aws_secret_access_key=secret-access-key
   ```

2. `config.json` ファイルの `auth` セクションを設定します。詳細については、「(オプション) config.json の設定」を参照してください。

AWS IoT Greengrass トラブルシューティング用の IDT

IDT for は、エラーのタイプに基づいてこれらのエラーをさまざまな場所に AWS IoT Greengrass 書き込みます。エラーは、コンソール、ログファイル、およびテストレポートに書き込まれます。

エラーコード

IDT for AWS IoT Greengrassによって生成されたエラーコードを次の表に示します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
101	InternalError	内部エラーが発生しました。	*<device-tester-extract-location>* `/results` ディレクトリの下のログを確認します。問題をデバッグできない場合は、AWS 開発者サポートにお問い合わせください。
102	TimeoutError	制限された時間範囲にテストを完了することができません。これは、次の場合に発生する可能性があります。 • テストマシンとデバイス間のネットワーク接続が低速です (VPN ネットワークを使用している場合など)。 • 遅いネットワークは、デバイスとクラウド間の通信を遅延させます。 •	• ネットワーク接続と速度をチェックしてください。 • `/test` ディレクトリの下のファイルを変更していないことを確認します。 • `"--group-id"` フラグを付けて失敗したテストグループを手動で実行してみてください。 • テストのタイムアウトを長くして、

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
		テスト設定ファイル (`test.json`) の `timeout` フィールドが誤って変更されています。	テストスイートを実行してみてください。詳細については、「タイムアウトエラー」を参照してください。
103	PlatformNotSupportError	`device.json` に誤った OS/アーキテクチャの組み合わせが指定されています。	サポートされている組み合わせのいずれかに設定を変更してください。 • Linux、x86_64 • Linux、ARMv6l • Linux、ARMv7l • Linux、AArch64 • Ubuntu、x86_64 • OpenWrt、ARMv7l • OpenWrt、AArch64 詳細については、「device.json の設定」を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
104	VersionNotSupportError	AWS IoT Greengrass Core ソフトウェアバージョンは、使用している IDT のバージョンではサポートされていません。	device_tester_bin version コマンドを使用して、サポートされているバージョンの AWS IoT Greengrass Core ソフトウェアを検索します。たとえば、macOS を使用している場合は、./devicetester_mac_x86_64 version を使用します。 使用している AWS IoT Greengrass Core ソフトウェアのバージョンを確認するには： • Core AWS IoT Greengrass ソフトウェアがプリインストールされたテストを実行している場合は、SSH を使用して AWS IoT Greengrass コアデバイスに接続し、 を実行します。 ***<path-to-preinstalled-green***

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			grass-location> /greengrass/ggc/core/greengrassd --version • 別のバージョンの AWS IoT Greengrass Core ソフトウェアでテストを実行している場合は、 `devicetester_greengrass_<os>/products/greengrass/gcc` ディレクトリに移動します。 AWS IoT Greengrass Core ソフトウェアバージョンは .zip ファイル名の一部です。 AWS IoT Greengrass Core ソフトウェアの別のバージョンをテストできます。詳細については、「の開始方法 AWS IoT Greengrass」を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
105	LanguageNotSupportError	IDT は、 AWS IoT Greengrass ライブラリと SDKsのみ Python をサポートしています。	次のことを確認してください。 • `devicetester_greengrass_<os>/products/greengrass/ggsdk` にある SDK パッケージは Python SDK。 • `devicetester_greengrass_<os>/tests/GGQ_1.0.0/suite/resources/run.runtimefarm/bin` にある bin フォルダの内容は変更されていない。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
106	ValidationError	`device.json` または `config.json` の一部のフィールドが無効です。	レポートのエラーコードの右側にあるエラーメッセージを確認します。 • デバイスの認証タイプが無効: デバイスに接続するための正しい方法を指定してください。詳細については、「the section called “device.json の設定”」を参照してください。 • Invalid private key path: プライベートキーへの正しいパスを指定します。詳細については、「device.json の設定」を参照してください。 • 無効 AWS リージョン: `config.json` ファイル AWS リージョン で有効な を指定します。詳細については、AWS サービスエンドポ

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			イントを参照してください。 • AWS 認証情報: テストマシンに有効な AWS 認証情報を設定します (環境変数または `credentials` ファイルを使用) 。`auth` フィールドが正しく設定されていることを確認します。詳細については、「the section called “の作成と設定 AWS アカウント”」を参照してください。 • HSM 入力が無効: `device.json` で `p11Provider` 、`privateKeyLabel` 、`slotLabel` 、`slotUserPin` 、および `openSSLEngine` フィールドを確認してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
107	SSHConnectionFailed	テストマシンが設定されたデバイスに接続できません。	`device.json` ファイルの以下のフィールドが正しいことを確認します。 • `ip` • `user` • `privKeyPath` • `password` 詳細については、「device.json の設定」を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
108	RunCommandError	テストでテスト対象デバイスでのコマンドを実行できませんでした。	root アクセスが `device.json` 内の設定済みユーザーに許可されていることを確認します。 root アクセスでコマンドを実行するとき、一部のデバイスではパスワードが必要です。ルートアクセスが、パスワードを使わずに許可されることを確認します。詳細については、デバイスのドキュメントを参照してください。 失敗したコマンドをデバイスで手動で実行して、エラーが発生するかどうかを確認します。
109	PermissionDeniedError	ルートアクセスがありません。	デバイス上で設定されたユーザーに root アクセスを設定します。
110	CreateFileError	ファイルを作成できません。	デバイスのディスク容量とディレクトリのアクセス許可を確認します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
111	CreateDirError	ディレクトリを作成できません。	デバイスのディスク容量とディレクトリのアクセス許可を確認します。
112	InvalidPathError	AWS IoT Greengrass Core ソフトウェアへのパスが正しくありません。	エラーメッセージのパスが有効であることを確認します。`devicetester_greengrass_<os>` ディレクトリのファイルを編集しないでください。
113	InvalidFileError	ファイル名が無効です。	エラーメッセージのファイルが有効であることを確認します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
114	ReadFileError	指定されたファイルを読み取ることができません。	以下について確認します。 • ファイルのアクセス許可が正しいことを確認します。 • `limits.config` では十分なファイルを開くことができます。 • エラーメッセージで指定されたファイルが存在し、有効です。 macOS でテストしている場合は、開くファイルの制限を増やします。デフォルトの制限は 256 で、テストには十分です。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
115	FileNotFoundError	必要なファイルが見つかりませんでした。	以下について確認します。 • 圧縮された Greengrass ファイルが `devicetester_greengrass_<os>/products/greengrass/ggc` にある。Core tar ファイルは、 AWS IoT Greengrass Core AWS IoT Greengrass ソフトウェアのダウンロードページからダウンロードできます。 • SDK パッケージは `devicetester_greengrass_<os>/products/greengrass/ggsdk` にあります。 • `devicetester_greengrass_<os>/`

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			testsのファイルは変更されていません。
116	OpenFileFailed	指定されたファイルを開くことができません。	以下について確認します。 • エラーメッセージで指定されたファイルが存在し、有効です。 • limits.configでは十分なファイルを開くことができます。 macOS でテストしている場合は、開くファイルの制限を増やします。デフォルトの制限は 256 で、テストには十分です。
117	WriteFileFailed	ファイルの書き込みに失敗しました (DUT またはテストマシンの可能性があります)。	エラーメッセージで示されたディレクトリが存在し、書き込みアクセス許可があることを確認します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
118	FileCleanUpError	テストで、指定されたファイルまたはディレクトリを削除すること、またはリモートデバイス上の指定されたファイルをマウント解除することに失敗しました。	バイナリファイルがまだ実行されている場合は、ファイルはロックされている可能性があります。プロセスを終了し、指定したファイルを削除します。
119	InvalidInputError	設定が無効です。	`suite.json` ファイルが有効であることを確認します。
120	InvalidCredentialError	AWS 認証情報が無効です。	• AWS 認証情報を確認します。詳細については、「the section called “AWS 認証情報を設定する”」を参照してください。 • ネットワーク接続を確認してテストグループを再実行します。ネットワークの問題も、このエラーの原因となります。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
121	AWSSessionError	AWS セッションの作成に失敗しました。	このエラーは、 AWS 認証情報が無効であるか、インターネット接続が不安定である場合に発生する可能性があります。を使用して AWS API オペレーションを呼び出し AWS CLI ます。
122	AWSApiCallError	AWS API エラーが発生しました。	このエラーは、ネットワークの問題が原因と考えられます。テストグループを再試行する前にネットワークを確認してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
123	IpNotExistError	IP アドレスは接続情報に含まれていません。	インターネット接続を確認してください。 AWS IoT Greengrass コンソールを使用して、テストで使用されている AWS IoT Greengrass コアモノの接続情報を確認できます。接続情報に 10 個のエンドポイントが含まれている場合は、その一部または全部を削除してテストを再実行できます。詳細については、「接続情報」を参照してください。
124	OTAJobNot CompleteError	OTA ジョブを完了できませんでした。	インターネット接続を確認して OTA テストグループを再実行します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
125	CreateGreengrassServiceRoleError	以下のいずれかが発生しました。 • ロールの作成中にエラーが発生しました。 • ポリシーを AWS IoT Greengrass サービスロールにアタッチ中にエラーが発生しました。 • サービスロールに関連付けられているポリシーが無効です。 • ロールを AWS アカウントに関連付けるときにエラーが発生しました。	AWS IoT Greengrass サービスロールを設定します。詳細については、「the section called “Greengrass サービスロール”」を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
126	DependenciesNotPresentError	特定のテストに必要な 1 つ以上の依存関係がデバイスに存在しません。	テストログ (`<device-tester-extract-location>` `/results/` `<execution-id>/logs/<test-case-name.log>`) を調べて、デバイスに欠落している依存関係を確認します。
127	InvalidHSMConfiguration	指定された HSM/PKCS 設定が正しくありません。	`device.json` ファイルで、PKCS#11 を使用して HSM とやり取りするために必要な正しい設定を指定します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング

128	OTAJobNot SuccededError	OTA ジョブは成功しませんでした。	• ota テストグループを個別に実行した場合、ggcdependencies テストグループを実行して、すべての依存関係 (wget など) が存在することを確認します。次に、ota テストグループを再試行します。
			• トラブルシューティングおよびエラー情報については、*<device-tester-extract-location>* /results/ *<execution-id>*/logs/ にある詳細なログを参照してください。具体的には、次のログを確認します。
			  • コンソールログ (test_manager.log)
			  •

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			OTA テストケースログ (`ota_test.log`) • GGC デーモンログ (`ota_test_ggc_logs.tar.gz`) • OTA Agent のログ (`ota_test_ota_logs.tar.gz`) • インターネット接続を確認して `ota` テストグループを再実行します。 • 問題が解決しない場合は、AWS 開発者サポートまでお問い合わせください。
129	NoConnectivityError	ホストエージェントがインターネットに接続できません。	ネットワーク接続とファイアウォールの設定を確認します。接続の問題が解決したら、テストグループを再試行します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
130	NoPermissionError	IDT for の実行に使用している IAM ユーザーには、IDT の実行に必要な AWS リソースを作成するアクセス許可 AWS IoT Greengrass がありません。	IDT for AWS IoT Greengrassの実行に必要な許可を付与するポリシーテンプレートについては、「アクセス許可ポリシーテンプレート」を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
131	LeftoverAgentExistError	IDT を開始しようとすると、デバイスは AWS IoT Greengrass プロセスを実行しています AWS IoT Greengrass。	デバイスで実行されている既存の Greengrass デーモンがないことを確認します。 • 次のコマンドを使用して、デーモンを停止できます: sudo ./*<absolute-path-to-greengrass-daemon>* / greengrassd stop。 • PID によって Greengrass デーモンを終了することもできます。 **Note** 再起動後に自動的に起動するように AWS IoT Greengrass 設定された の既存のインストールを使用している場合は、再起動後およびテス

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			トスイートを実行する前にデーモンを停止する必要があります。
132	DeviceTimeOffsetError	デバイスの時刻が正しくありません。	デバイスの時刻を正しく設定します。
133	InvalidMLConfiguration	指定された ML 設定が正しくありません。	`device.json` ファイルに、ML 推論テストの実行に必要な正しい設定を指定します。詳細については、「the section called “オプション: ML 認定のためのデバイスの設定”」を参照してください。

AWS IoT Greengrass エラーの IDT の解決

IDT を使用する場合は、IDT for を実行する前に正しい設定ファイルを用意する必要があります AWS IoT Greengrass。解析エラーや設定エラーが発生する場合は、まず環境に適した設定テンプレートを見つけて使用します。

それでも問題が解決されない場合は、次のデバッグプロセスを参照してください。

トピック

- エラーをどこで探せばよいか
- 解析エラー
- 必須パラメータが見つからないエラー
- テストを開始できなかったエラー

- リソースにアクセスする権限がないエラー
- アクセス拒否エラー
- SSH 接続エラー
- タイムアウトエラー
- コマンドが見つからないエラーがテスト中に発生する
- macOS でのセキュリティ例外

エラーをどこで探せばよいか

実行中にエラーの概要がコンソールに表示され、テストがすべて完了すると、失敗したテストの概要とエラーが表示されます。`awsiotdevicetester_report.xml` には、テストが失敗する原因となったすべてのエラーの概要が含まれます。テスト実行ごとのログファイルは、テスト実行中にコンソールに表示されたテスト実行用の UUID という名前のディレクトリに保存されます。

テストログのディレクトリは、*<device-tester-extract-location>*/results/*<execution-id>*/logs/ にあります。このディレクトリには、デバッグに役立つ次のファイルが含まれています。

ファイル	説明
`test_manager.log`	テスト実行中にコンソールに書き込まれたすべてのログ。結果の概要はこのファイルの最後にあり、失敗したテストのリストが含まれます。 失敗に関する情報は、このファイルの警告ログとエラーログで確認できます。
<test-group-id> __*<test-name>* .log	特定のテストの詳細なログ。
<test-name> _ggc_logs.tar.gz	テスト中に AWS IoT Greengrass コアデーモンが生成したすべてのログの圧縮されたコレクション。詳細については、「トラブルシューティング AWS IoT Greengrass」を参照してください。

ファイル	説明
`<test-name>_ota_logs.tar.gz`	テスト中に AWS IoT Greengrass OTA エージェントによって生成されたログの圧縮コレクション。OTA テストのみ。
`<test-name>_basic_assertion_publisher_ggad_logs.tar.gz`	テスト中に AWS IoT 発行者デバイスによって生成されたログの圧縮されたコレクションです。
`<test-name>_basic_assertion_subscriber_ggad_logs.tar.gz`	テスト中に AWS IoT サブスクライバーデバイスによって生成されたログの圧縮されたコレクションです。

解析エラー

場合によっては、JSON 設定のタイプミスが解析エラーにつながることがあります。ほとんどの場合、JSON ファイルで括弧やカンマ、引用符を忘れたことが原因です。IDT は、JSON 検証を行い、デバッグ情報を出力します。エラーが発生した行、構文エラーの行番号と列番号が出力されます。この情報だけでエラーの修正が可能なはずですが、それでもエラーを特定できない場合は、IDE、テキストエディタ (Atom、Sublime など)、またはオンラインツール (JSONLint など) を使って手動で検証できます。

必須パラメータが見つからないエラー

IDT には新機能が追加されているため、設定ファイルに変更が生じる可能性があります。古い設定ファイルを使用すると、設定が破損する可能性があります。このような場合は、`/results/<execution-id>/logs` にある `<test_case_id>.log` ファイルに、すべての不足しているパラメータが明確に示されています。また、IDT では、JSON 設定ファイルのスキーマを検証し、最新のサポートされているバージョンが使用されていることを確認します。

テストを開始できなかったエラー

テスト開始時の障害を示すエラーが発生する場合があります。考えられる原因にはさまざまなものがあるため、以下を実行します。

- 実行コマンドに含めたプール名が実際に存在することを確認します。プール名は、`device.json` ファイルから直接参照されます。

- プール内のデバイスの設定パラメータが正しいことを確認します。

リソースにアクセスする権限がないエラー

ターミナルの出力または `/results/<execution-id>/logs` の `test_manager.log` ファイルに `<user or role> is not authorized to access this resource` エラーメッセージが表示される場合があります。この問題を解決するには、`AWSIoTDeviceTesterForGreengrassFullAccess` 管理ポリシーをテストユーザーにアタッチします。詳細については、「the section called “の作成と設定 AWS アカウント”」を参照してください。

アクセス拒否エラー

IDT は、テスト対象デバイスのさまざまなディレクトリやファイルに対してオペレーションを実行します。一部のオペレーションにはルートアクセスが必要です。これらのオペレーションを自動化するには、パスワードを入力することなく、IDT で sudo を使用してコマンドを実行する必要があります。

パスワードを入力することなく、sudo にアクセスを許可するには、以下の手順を実行します。

> **Note**
>
> `user` および `username` は、テスト対象デバイスにアクセスするために IDT で使用する SSH ユーザーを指します。

1. SSH ユーザーを sudo グループに追加するには sudo usermod -aG sudo ***<ssh-username>*** を使用します。
2. サインアウトし、再度サインインして、変更を反映します。
3. `/etc/sudoers` ファイルを開き、ファイルの末尾に次の行を追加します: *<ssh-username>* `ALL=(ALL) NOPASSWD: ALL`

   > **Note**
   >
   > ベストプラクティスとして、`/etc/sudoers` を編集するときは sudo visudo を使用することをお勧めします。

SSH 接続エラー

IDT からテスト対象デバイスに接続できない場合は、接続エラーのログが `/results/<execution-id>/logs/<test-case-id>.log` に記録されます。SSH エラーに関するメッセージは、このログファイルの上部に表示されます。テスト対象デバイスへの接続は IDT が実行する最初のオペレーションの 1 つであるためです。

ほとんどの Windows セットアップでは、PuTTy ターミナルアプリケーションを使用して Linux ホストに接続します。このアプリケーションは、標準 PEM プライベートキーファイルを PPK と呼ばれる独自の Windows 形式に変換することを要求します。IDT を `device.json` ファイルで設定する場合は、PEM ファイルのみを使用します。PPK ファイルを使用する場合、IDT は AWS IoT Greengrass デバイスとの SSH 接続を作成できず、テストを実行できません。

タイムアウトエラー

各テストのタイムアウトを長くするには、タイムアウト乗数を指定します。この値は、各テストのタイムアウトのデフォルト値に適用されます。このフラグに設定された値はすべて、1.0 以上である必要があります。

タイムアウトの乗数を使用するには、テストの実行時に `--timeout-multiplier` フラグを使用します。例:

```
./devicetester_linux run-suite --suite-id GGQ_1.0.0 --pool-id DevicePool1 --timeout-
multiplier 2.5
```

詳細については、`run-suite --help` を実行してください。

コマンドが見つからないエラーがテスト中に発生する

AWS IoT Greengrass デバイスでテストを実行するには、古いバージョンの OpenSSL ライブラリ (libssl1.0.0) が必要です。通常、現在の Linux ディストリビューションでは、libssl バージョン 1.0.2 以降 (v1.1.0) を使用しています。

たとえば、Raspberry Pi で必要なバージョンの libssl をインストールするには、以下のコマンドを実行します。

1. ```
   wget http://ftp.us.debian.org/debian/pool/main/o/openssl/
   libssl1.0.0_1.0.2l-1~bpo8+1_armhf.deb
   ```
2. ```
   sudo dpkg -i libssl1.0.0_1.0.2l-1~bpo8+1_armhf.deb
   ```

macOS でのセキュリティ例外

macOS 10.15 を使用するホストマシンで IDT を実行すると、IDT の認証チケットが正しく検出されず、IDT の実行がブロックされます。IDT を実行するには、セキュリティ例外を `devicetester_mac_x86-64` 実行可能ファイルに付与する必要があります。

セキュリティ例外を IDT 実行可能ファイルに付与するには

1. [Apple] メニューから [System Preferences] (システム環境設定) を選択します。
2. [Security & Privacy] (セキュリティとプライバシー) を選択し、[General] (一般) タブでロックアイコンをクリックして、セキュリティ設定を変更します。
3. メッセージ `"devicetester_mac_x86-64" was blocked from use because it is not from an identified developer.` を選択して、[Allow Anyway] (すべてのアプリケーションを許可) を選択します。
4. セキュリティ警告を受け入れます。

IDT サポートポリシーについてご質問がある場合は、AWS カスタマーサポートまでお問い合わせください。

の AWS IoT Device Tester のサポートポリシー AWS IoT Greengrass V1

AWS IoT Device Tester (IDT) for AWS IoT Greengrass は、ダウンロード可能なテストフレームワークであり、 AWS IoT Greengrass Device AWS Partner Catalog に含めるデバイスを検証して認定できます。デバイスのテストまたは認定には、最新バージョンの AWS IoT Greengrass および IDT を使用することをお勧めします。詳細については、「AWS IoT Greengrass Version 2 デベロッパーガイド」の「IDT for AWS IoT Greengrass V2のサポートされているバージョン」を参照してください。

また、サポートされているバージョンの AWS IoT Greengrass および IDT を使用して、デバイスをテストまたは認定することもできます。サポートされていないバージョンの IDT は引き続き使用できますが、これらのバージョンはバグ修正や更新プログラムを受け取りません。

> **Important**
>
> 2022 年 4 月 4 日以降、 の AWS IoT デバイステスター (IDT) は署名付き認定レポートを生成 AWS IoT Greengrass V1 しなくなりました。Device Qualification Program を通じて AWS

Partner Device Catalog にリスト AWS IoT Greengrass V1 する新しいデバイスを認定することはできなくなりました。 AWSGreengrass V1 デバイスを認定することはできませんが、引き続き IDT for AWS IoT Greengrass V1 を使用して Greengrass V1 デバイスをテストできます。IDT for AWS IoT Greengrass V2 を使用して、Greengrass デバイスを認定し、AWS Partner Device Catalog に含めることをお勧めします。

サポートポリシーについてご質問がある場合は、AWS カスタマーサポートまでお問い合わせください。

トラブルシューティング AWS IoT Greengrass

このセクションでは、 の問題を解決するためのトラブルシューティング情報と可能な解決策を示します AWS IoT Greengrass。

AWS IoT Greengrass クォータ (制限) の詳細については、『』のService Quotas」を参照してくださいAmazon Web Services 全般のリファレンス。

AWS IoT Greengrass 主な問題

AWS IoT Greengrass Core ソフトウェアが起動しない場合は、次の一般的なトラブルシューティング手順を試してください。

- アーキテクチャに適したバイナリをインストールしていることを確認します。詳細については、「AWS IoT Greengrass Core ソフトウェア」を参照してください。
- Core デバイスに使用可能なローカルストレージがあることを確認します。詳細については、「the section called “ストレージ問題のトラブルシューティング”」を参照してください。
- `runtime.log` と `crash.log` でエラーメッセージを確認します。詳細については、「the section called “ログによるトラブルシューティング”」を参照してください。

以下の症状とエラーを検索して、 AWS IoT Greengrass コアに関する問題のトラブルシューティングに役立つ情報を見つけます。

問題

- エラー: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass daemon process with [pid = <pid>] died.
- エラー: Failed to parse /<greengrass-root>/config/config.json.
- エラー: Error occurred while generating TLS config: ErrUnknownURIScheme
- エラー: Runtime failed to start: unable to start workers: container test timed out.
- エラー: Failed to invoke PutLogEvents on local Cloudwatch, logGroup: /GreengrassSystem/connection_manager, error: RequestError: send request failed caused by: Post http://<path>/cloudwatch/logs/: dial tcp <address>: getsockopt: connection refused, response: { }.
- エラー: Unable to create server due to: failed to load group: chmod /<greengrass-root>/ggc/deployment/lambda/arn:aws:lambda:<region>:<account-id>:function:<function-name>:<version>/<file-name>: no such file or directory.

- AWS IoT Greengrass Core ソフトウェアは、コンテナ化なしでの実行から Greengrass コンテナでの実行に変更した後は起動しません。
- エラー: Spool size should be at least 262144 bytes.
- エラー: [ERROR]-Cloud messaging error: Error occurred while trying to publish a message. {"errorString": "operation timed out"}
- エラー: container_linux.go:344: starting container process caused "process_linux.go:424: container init caused \"rootfs_linux.go:64: mounting \\\"/greengrass/ggc/socket/greengrass_ipc.sock\\\" to rootfs \\\"/greengrass/ggc/packages/<version>/rootfs/merged\\\" at \\\"/greengrass_ipc.sock\\\" caused \\\"stat /greengrass/ggc/socket/greengrass_ipc.sock: permission denied\\\"\"".
- エラー: Greengrass daemon running with PID: <process-id>. Some system components failed to start. Check 'runtime.log' for errors.
- デバイスシャドウがクラウドと同期していない。
- エラー: unable to accept TCP connection. accept tcp [::]:8000: accept4: too many open files.
- エラー: Runtime execution error: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:50: preparing rootfs caused \\\"permission denied\\\"\"".
- 警告: [WARN]-[5]GK Remote: Error retrieving public key data: ErrPrincipalNotConfigured: private key for MqttCertificate is not set.
- エラー: Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/core/<architecture>/greengrass-core-<distribution-version>.tar.gz.
- AWS IoT Greengrass コアはネットワークプロキシを使用するように設定されており、Lambda 関数は送信接続を実行できません。
- コアで接続と切断の無限ループが発生し、runtime.log ファイルに連続する接続と切断のエントリが含まれている。
- エラー: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:62: mounting \\\"proc\\\" to rootfs \\\"
- [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"}
- [ERROR]-Deployment failed. {"deploymentId": "<deployment-id>", "errorString": "container test process with pid <pid> failed: container process state: exit status 1"}

- エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to create overlay fs for container: mounting overlay at /greengrass/ggc/packages/<ggc-version>/rootfs/merged failed: failed to mount with args source=\"no_source\" dest=\"/greengrass/ggc/packages/<ggc-version>/rootfs/merged\" fstype=\"overlay\" flags=\"0\" data=\"lowerdir=/greengrass/ggc/packages/<ggc-version>/dns:/,upperdir=/greengrass/ggc/packages/<ggc-version>/rootfs/upper,workdir=/greengrass/ggc/packages/<ggc-version>/rootfs/work\": too many levels of symbolic links"}
- エラー: [DEBUG] - ルートの取得に失敗しました。メッセージを破棄します。
- エラー: [Errno 24] 開いている <lambda-function> が多すぎます、[Errno 24] 開いているファイルが多すぎます
- エラー: ds server failed to start listening to socket: listen unix <ggc-path>/ggc/socket/greengrass_ipc.sock: bind: invalid argument
- [INFO] (Copier) aws.greengrass.StreamManager: stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException: Instant exceeds minimum or maximum instant
- GPG error: https://dnw9lb6lzp2d8.cloudfront.net stable InRelease: The following signatures were invalid: EXPKEYSIG 68D644ABD2327D47 AWS Greengrass Master Key

エラー: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass daemon process with [pid = <pid>] died.

解決策： AWS IoT Greengrass Core ソフトウェア`crash.log`が起動しない場合、 にこのエラーが表示されることがあります。このエラーは、v1.6 以前を実行している場合に発生する可能性があります。次のいずれかを行います：

- v1.7 以降にアップグレードします。常に最新バージョンの AWS IoT Greengrass Core ソフトウェアを実行することをお勧めします。ダウンロード情報については、「AWS IoT Greengrass Core ソフトウェア」を参照してください。
- AWS IoT Greengrass Core ソフトウェアバージョンに正しい`config.json`形式を使用します。詳細については、「the section called “AWS IoT Greengrass コア設定ファイル”」を参照してください。

> **Note**
>
> コアデバイスにインストールされている AWS IoT Greengrass Core ソフトウェアのバージョンを確認するには、デバイスターミナルで次のコマンドを実行します。
>
> ```
> cd /greengrass-root/ggc/core/
> sudo ./greengrassd --version
> ```

エラー: Failed to parse /<greengrass-root>/config/config.json.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。Greengrass 設定ファイルが有効な JSON 形式であることを確認します。

`config.json` (`/greengrass-root/config` に存在) を開き、JSON 形式を検証します。例えば、カンマが正しく使用されていることを確認してください。

エラー: Error occurred while generating TLS config: ErrUnknownURIScheme

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。Greengrass 設定ファイルの crypto セクションのプロパティが有効であることを確認します。エラーメッセージに詳細が説明されています。

`config.json` (`/greengrass-root/config` に存在) を開き、`crypto` セクションを確認します。例えば、証明書とキーのパスは正しい URI 形式を使用し、正しい場所を参照している必要があります。

エラー: Runtime failed to start: unable to start workers: container test timed out.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。Greengrass 設定ファイルの `postStartHealthCheckTimeout` プロパティを設定し

ます。このオプションプロパティは、起動後のヘルスチェックが終了するまで Greengrass デーモンが待機する時間 (ミリ秒単位) を設定します。デフォルト値は 30 秒 (30000 ミリ秒) です。

`config.json` (`/greengrass-root/config` に存在) を開きます。`runtime` オブジェクトに `postStartHealthCheckTimeout` プロパティを追加し、30,000 を超える値を設定します。有効な JSON ドキュメントを作成するために、必要に応じてカンマを追加してください。例:

```
...
"runtime" : {
  "cgroup" : {
    "useSystemd" : "yes"
  },
  "postStartHealthCheckTimeout" : 40000
},
...
```

エラー: Failed to invoke PutLogEvents on local Cloudwatch, logGroup: /GreengrassSystem/connection_manager, error: RequestError: send request failed caused by: Post http://<path>/cloudwatch/logs/: dial tcp <address>: getsockopt: connection refused, response: { }.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。これは、Raspberry Pi AWS IoT Greengrass で実行していて、必要なメモリ設定が完了していない場合に発生する可能性があります。詳細については、このステップを参照してください。

エラー: Unable to create server due to: failed to load group: chmod /<greengrass-root>/ggc/deployment/lambda/arn:aws:lambda:<region>:<account-id>:function:<function-name>:<version>/<file-name>: no such file or directory.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。コアに Lambda 実行可能ファイル をデプロイしたら、関数の `Handler` プロパティを `group.json` ファイル (`/greengrass-root`/ggc/deployment/group にある) で確認します。ハンド

ラがコンパイルした実行可能ファイルの正確な名前でない場合、`group.json` ファイルのコンテンツを空の JSON オブジェクト (`{}`) に置き換え、以下のコマンドを実行して AWS IoT Greengrassを開始します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

次に、AWS Lambda API を使用して関数設定の `handler` パラメータを更新し、新しい関数バージョンを発行してエイリアスを更新します。詳細については、「AWS Lambda 関数のバージョニングとエイリアス」を参照してください。

Greengrass グループにエイリアスで関数を追加したと想定すると (推奨)、これでグループを再デプロイできるようになります。(そうでない場合は、グループをデプロイする前に、グループ定義とサブスクリプションで新しい関数バージョンあるいはエイリアスを指定する必要があります。)

AWS IoT Greengrass Core ソフトウェアは、コンテナ化なしでの実行から Greengrass コンテナでの実行に変更した後は起動しません。

解決策: コンテナの依存関係が不足していないことを確認します。

エラー: Spool size should be at least 262144 bytes.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。`group.json` ファイル (`/greengrass-root/ggc/deployment/group` に存在) を開き、このファイルの内容を空の JSON オブジェクト (`{}`) に置き換え、以下のコマンドを実行して AWS IoT Greengrassを起動します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

次に、「the section called “メッセージをローカルストレージにキャッシュするには”」のステップに従います。`GGCloudSpooler` 関数で、`GG_CONFIG_MAX_SIZE_BYTES` 値が 262144 以上に指定されていることを確認します。

エラー: [ERROR]-Cloud messaging error: Error occurred while trying to publish a message. {"errorString": "operation timed out"}

解決策: Greengrass コアが MQTT メッセージを AWS IoT Coreに送信できない場合に、`GGCloudSpooler.log` にこのエラーが記録される場合があります。これは、コア環境の帯域幅が限られていて、レイテンシーが高い場合に発生する可能性があります。 AWS IoT Greengrass v1.10.2 以降を実行している場合は、config.json ファイル`mqttOperationTimeout`の値を増やしてみてください。このプロパティが存在しない場合は、`coreThing` オブジェクトに追加してください。例:

```
{
  "coreThing": {
    "mqttOperationTimeout": 10,
    "caPath": "root-ca.pem",
    "certPath": "hash.cert.pem",
    "keyPath": "hash.private.key",
    ...
  },
  ...
}
```

デフォルト値は 5 で、最小値は 5 です。

エラー: container_linux.go:344: starting container process caused "process_linux.go:424: container init caused \"rootfs_linux.go:64: mounting \\\"/greengrass/ggc/socket/greengrass_ipc.sock\\\" to rootfs \\\"/greengrass/ggc/packages/<version>/rootfs/merged\\\" at \\\"/greengrass_ipc.sock\\\" caused \\\"stat /greengrass/ggc/socket/greengrass_ipc.sock: permission denied\\\"\"".

解決策： AWS IoT Greengrass Core ソフトウェアが起動しない場合`runtime.log`、 にこのエラーが表示されることがあります。このエラーは、`umask` が `0022` よりも高い場合に発生します。この問題を解決するには、`umask` を `0022` 以下に設定する必要があります。値 `0022` では、デフォルトで新しいファイルに対する読み取りアクセス許可がすべてのユーザーに付与されます。

エラー: Greengrass daemon running with PID: <process-id>. Some system components failed to start. Check 'runtime.log' for errors.

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。固有のエラー情報については、`runtime.log` と `crash.log` を確認してください。詳細については、「the section called “ログによるトラブルシューティング”」を参照してください。

デバイスシャドウがクラウドと同期していない。

解決策： に Greengrass サービスロールの `iot:UpdateThingShadow` および `iot:GetThingShadow`アクションに対するアクセス許可 AWS IoT Greengrass があることを確認します。サービスロールで `AWSGreengrassResourceAccessRolePolicy` 管理ポリシーを使用している場合は、これらのアクセス許可はデフォルトで含まれています。

「シャドウ同期タイムアウト問題のトラブルシューティング」を参照してください。

エラー: unable to accept TCP connection. accept tcp [::]:8000: accept4: too many open files.

解決策: `greengrassd` スクリプトの出力にこのエラーが表示されることがあります。これは、 AWS IoT Greengrass Core ソフトウェアのファイル記述子の制限がしきい値に達し、引き上げる必要がある場合に発生する可能性があります。

次のコマンドを使用して Core AWS IoT Greengrass ソフトウェアを再起動します。

```
ulimit -n 2048
```

> **Note**
>
> この例では、制限を 2048 に引き上げています。ユースケースに適した値を選択してください。

エラー: Runtime execution error: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:50: preparing rootfs caused \\\"permission denied\\\"\"".

解決策： ルートディレクトリ AWS IoT Greengrass に直接 をインストールするか、 AWS IoT Greengrass Core ソフトウェアがインストールされているディレクトリとその親ディレクトリにすべてのユーザーに対する`execute`アクセス許可があることを確認します。

警告: [WARN]-[5]GK Remote: Error retrieving public key data: ErrPrincipalNotConfigured: private key for MqttCertificate is not set.

解決策： AWS IoT Greengrass 共通のハンドラーを使用して、すべてのセキュリティプリンシパルのプロパティを検証します。`runtime.log` 内のこの警告は、ローカル MQTT サーバー用のカスタムプライベートキーを指定しない限り予期されるものです。詳細については、「the section called “セキュリティプリンシパル”」を参照してください。

エラー: Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/core/<architecture>/greengrass-core-<distribution-version>.tar.gz.

解決策: 無線通信 (OTA) での更新が失敗した場合に、このエラーが表示されることがあります。署名者ロールポリシーで、ターゲット AWS リージョン を `Resource` として追加します。この署名者ロールは、 AWS IoT Greengrass ソフトウェア更新の S3 URL に事前署名するために使用されます。詳細については、「S3 URL の署名者ロール」を参照してください。

AWS IoT Greengrass コアはネットワークプロキシを使用するように設定されており、Lambda 関数は送信接続を実行できません。

解決策: 接続を行うために Lambda 関数で使用されるランタイムと実行可能ファイルによっては、接続タイムアウトのエラーも発生することがあります。Lambda 関数が適切なプロキシ設定を使用してネットワークプロキシ経由で接続していることを確認します。 はhttps_proxy、http_proxy、、および no_proxy環境変数を介して、プロキシ設定をユーザー定義の Lambda 関数 AWS IoT Greengrass に渡します。これらの変数には、以下の Python スニペットに示す方法でアクセスできます。

```
import os
print(os.environ['http_proxy'])
```

大文字と小文字の区別は、お使いの環境で定義されている変数に合わせてください。例えば、すべて小文字の http_proxy やすべて大文字の HTTP_PROXY にすることができます。これらの変数では、 は両方 AWS IoT Greengrass をサポートします。

> **Note**
>
> 接続を行うために使用されるほとんどの共通ライブラリ (boto3 や cURL など、および python requests パッケージ) は、デフォルトでこれらの環境変数を使用します。

コアで接続と切断の無限ループが発生し、runtime.log ファイルに連続する接続と切断のエントリが含まれている。

解決策: このエラーは、 AWS IoTへの MQTT 接続用に別のデバイスがクライアント ID をコアのモノ名を使用するようにハードコードされている場合に発生します。同じ AWS リージョン と の同時接続 AWS アカウント では、一意のクライアント IDsを使用する必要があります。デフォルトでは、コアは上記の接続にコアのモノ名をクライアント ID として使用します。

この問題を解結するには、他のデバイスが接続用に使用するクライアント ID を変更するか、またはコアのデフォルトの値を上書きできます。

コアデバイスのデフォルトのクライアント ID を上書きするには

1. 次のコマンドを実行して Greengrass デーモンを停止します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

2. su ユーザーとして *greengrass-root*/config/config.json を編集用に開きます。
3. coreThing オブジェクトに coreClientId プロパティを追加し、その値をカスタムのクライアント ID に設定します。値は 1 ~ 128 文字で指定します。この値は AWS アカウントの現在の AWS リージョン で一意であることが必要です。

```
"coreClientId": "MyCustomClientId"
```

4. デーモンを開始します。

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

エラー: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused \"rootfs_linux.go:62: mounting \\\"proc\\\" to rootfs \\\"

解決策： 一部のプラットフォームでは、 が/procファイルシステムをマウントして Lambda コンテナを作成 AWS IoT Greengrass しようとするruntime.logと、 にこのエラーが表示されることがあります。または、operation not permitted や EPERM などの類似するエラーが表示される場合があります。これらのエラーは、依存関係チェッカーのスクリプトパスによってプラットフォーム上でテストが実行された場合にも発生する可能性があります。

以下のいずれかの解決策を試してください。

- Linux カーネルで CONFIG_DEVPTS_MULTIPLE_INSTANCES オプションを有効にします。
- ホストの /proc マウントオプションを rw,relatim のみに設定します。
- Linux カーネルを 4.9 以降にアップグレードします。

> **Note**
>
> この問題は、ローカルリソースアクセスの `/proc` マウントには関連していません。

[ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"}

解決策: デプロイに失敗した場合に、runtime.log でこのエラーが表示されることがあります。このエラーは、 AWS IoT Greengrass グループの Lambda 関数がコアのファイルシステムの `/usr` ディレクトリにアクセスできない場合に発生します。

この問題を解決するには、ローカルボリュームリソースをグループに追加し、グループをデプロイします。このリソースは次の条件を満たす必要があります。

- `/usr` を [ソースパス] および [送信先パス] として指定します。
- リソースを所有する Linux グループの OS グループアクセス許可を自動的に追加します。
- Lambda 関数と関連付けて、読み取り専用アクセスを許可します。

[ERROR]-Deployment failed. {"deploymentId": "<deployment-id>", "errorString": "container test process with pid <pid> failed: container process state: exit status 1"}

解決策: デプロイに失敗した場合に、runtime.log でこのエラーが表示されることがあります。このエラーは、 AWS IoT Greengrass グループの Lambda 関数がコアのファイルシステムの `/usr` ディレクトリにアクセスできない場合に発生します。

`GGCanary.log` で追加のエラーを確認して、これが当てはまることを確認することができます。Lambda 関数が `/usr` ディレクトリにアクセスできない場合は、`GGCanary.log` に次のエラーが含まれます。

```
[ERROR]-standard_init_linux.go:207: exec user process caused "no such file or
 directory"
```

この問題を解決するには、ローカルボリュームリソースをグループに追加し、グループをデプロイします。このリソースは次の条件を満たす必要があります。

- `/usr` を [ソースパス] および [送信先パス] として指定します。
- リソースを所有する Linux グループの OS グループアクセス許可を自動的に追加します。
- Lambda 関数と関連付けて、読み取り専用アクセスを許可します。

エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to create overlay fs for container: mounting overlay at /greengrass/ggc/packages/<ggc-version>/rootfs/merged failed: failed to mount with args source=\"no_source\" dest=\"/greengrass/ggc/packages/<ggc-version>/rootfs/merged\" fstype=\"overlay\" flags=\"0\" data=\"lowerdir=/greengrass/ggc/packages/<ggc-version>/dns:/,upperdir=/greengrass/ggc/packages/<ggc-version>/rootfs/upper,workdir=/greengrass/ggc/packages/<ggc-version>/rootfs/work\": too many levels of symbolic links"}

解決策： AWS IoT Greengrass Core ソフトウェアが起動しない場合、このエラーが `runtime.log` ファイルに表示されることがあります。これは Debian オペレーティングシステムで生じやすい問題です。

この問題を解決するには、次の操作を行います。

1. AWS IoT Greengrass Core ソフトウェアを v1.9.3 以降にアップグレードします。これにより、この問題は自動的に解決されます。
2. AWS IoT Greengrass Core ソフトウェアのアップグレード後もこのエラーが続く場合は、config.json ファイル`true`で `system.useOverlayWithTmpfs`プロパティを に設定します。

Example 例

```
{
  "system": {
    "useOverlayWithTmpfs": true
  },
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud.pem.crt",
    "keyPath": "cloud.pem.key",
    ...
  },
  ...
}
```

> **Note**
>
> AWS IoT Greengrass Core ソフトウェアのバージョンがエラーメッセージに表示されます。Linux カーネルのバージョンを確認するには、`uname -r` を実行します。

エラー: [DEBUG] - ルートの取得に失敗しました。メッセージを破棄します。

解決策: グループのサブスクリプションを確認し、`[DEBUG]` メッセージにリストされているサブスクリプションが存在することを確認します。

エラー: [Errno 24] 開いている <lambda-function> が多すぎます、[Errno 24] 開いているファイルが多すぎます

解決策: Lamda 関数が関数ハンドラ内の `StreamManagerClient` をインスタンス化する場合、この関数のログファイルにこのエラーが表示されます。ハンドラの外部にクライアントを作成することをお勧めします。詳細については、「the section called “ストリームを操作するために StreamManagerClient を使用する”」を参照してください。

エラー: ds server failed to start listening to socket: listen unix <ggc-path>/ggc/socket/greengrass_ipc.sock: bind: invalid argument

解決策: AWS IoT Greengrass Core ソフトウェアが起動しない場合に、このエラーが表示されることがあります。このエラーは、長いファイルパスを持つフォルダに AWS IoT Greengrass Core ソフトウェアがインストールされている場合に発生します。 AWS IoT Greengrass Core ソフトウェアを、書き込みディレクトリを使用しない場合は 79 バイト未満のファイルパス、書き込みディレクトリを使用する場合は 83 バイトのフォルダに再インストールします。

[INFO] (Copier) aws.greengrass.StreamManager: stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException: Instant exceeds minimum or maximum instant

AWS IoT Greengrass コアソフトウェアを v1.11.3 にアップグレードすると、ストリームマネージャーの起動に失敗すると、ストリームマネージャーログに次のエラーが表示されることがあります。

```
2021-07-16T00:54:58.568Z [INFO] (Copier) aws.greengrass.StreamManager:
 stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException:
 Instant exceeds minimum or maximum instant (through reference chain:
 com.amazonaws.iot.greengrass.streammanager.export.PersistedSuccessExportStatesV1["lastExportTi
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
2021-07-16T00:54:58.579Z [INFO] (Copier) aws.greengrass.StreamManager: stdout.
 Caused by: java.time.DateTimeException: Instant exceeds minimum or maximum instant.
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
```

v1.11.3 より古いバージョンの AWS IoT Greengrass コアソフトウェアを使用していて、新しいバージョンにアップグレードする場合は、OTA 更新を使用して v1.11.4 にアップグレードします。

GPG error: https://dnw9lb6lzp2d8.cloudfront.net stable InRelease: The following signatures were invalid: EXPKEYSIG 68D644ABD2327D47 AWS Greengrass Master Key

APT リポジトリからコアソフトウェアをインストールしたapt updateデバイスで を実行すると、次のエラーが表示されることがあります。 AWS IoT Greengrass

```
Err:4 https://dnw9lb6lzp2d8.cloudfront.net stable InRelease
  The following signatures were invalid: EXPKEYSIG 68D644ABD2327D47 AWS Greengrass
 Master Key
Reading package lists... Done
W: GPG error: https://dnw9lb6lzp2d8.cloudfront.net stable InRelease: The following
 signatures were invalid: EXPKEYSIG 68D644ABD2327D47 AWS Greengrass Master Key
```

このエラーは、 が APT リポジトリから AWS IoT Greengrass コアソフトウェアをインストールまたは更新するオプションを提供し AWS IoT Greengrass なくなったために発生します。を正常に実行するにはapt update、デバイスのソースリストから AWS IoT Greengrass リポジトリを削除します。

```
sudo rm /etc/apt/sources.list.d/greengrass.list
sudo apt update
```

デプロイに関する問題

以下の情報は、デプロイの問題のトラブルシューティングに役立ちます。

問題

- 現在のデプロイが機能しないため、以前の正常なデプロイに戻す必要がある。
- デプロイに関する 403 Forbidden エラーがログに記録される。
- create-deployment コマンドを初めて実行すると、ConcurrentDeployment エラーが発生する。
- エラー: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account.
- エラー: デプロイのダウンロードステップを実行できません。ダウンロード中のエラー: グループ定義ファイルのダウンロード中のエラー: ... x509: 証明書の有効期限が切れているか、まだ有効ではありません

- デプロイが完了しない。
- エラー: Unable to find java or java8 executables, or the error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: worker with <worker-id> failed to initialize with reason Installed Java version must be greater than or equal to 8
- デプロイが完了せず、runtime.log に複数の「wait 1s for container to stop」エントリが含まれる。
- デプロイが完了せず、runtime.log に「[ERROR]-Greengrass deployment error: failed to report deployment status back to cloud {"deploymentId": "<deployment-id>", "errorString": "Failed to initiate PUT, endpoint: https://<deployment-status>, error: Put https://<deployment-status>: proxyconnect tcp: x509: certificate signed by unknown authority"}」が含まれる
- エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: <path>.
- エラー: <list-of-function-arns> are configured to run as root but Greengrass is not configured to run Lambda functions with root permissions.
- エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user: user: unknown user ggc_user.
- エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"}
- エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: process start failed: container_linux.go:259: starting container process caused "process_linux.go:250: running exec setns process for init caused \"wait: no child processes\"".
- エラー: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

現在のデプロイが機能しないため、以前の正常なデプロイに戻す必要がある。

解決策： AWS IoT コンソールまたは AWS IoT Greengrass API を使用して、以前の作業中のデプロイを再デプロイします。これにより、対応するグループバージョンが Core デバイスにデプロイされます。

デプロイを再デプロイするには (コンソール)

1. グループの設定ページで、[デプロイ] を選択します。このページには、日付と時刻、グループバージョン、各デプロイ試行のステータスなど、グループのデプロイ履歴が表示されます。
2. 再デプロイするデプロイが含まれている行を見つけます。再デプロイするデプロイを選択し、[Redeploy] (再デプロイ) を選択します。

デプロイを再デプロイするには (CLI)

1. ListDeployments を使用して、再デプロイするデプロイの ID を見つけます。例:

```
aws greengrass list-deployments --group-id 74d0b623-c2f2-4cad-9acc-ef92f61fcaf7
```

このコマンドは、グループのデプロイのリストを返します。

```
{
    "Deployments": [
        {
            "DeploymentId": "8d179428-f617-4a77-8a0c-3d61fb8446a6",
            "DeploymentType": "NewDeployment",
            "GroupArn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/
groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/8dd1d899-4ac9-4f5d-
afe4-22de086efc62",
```

```
            "CreatedAt": "2019-07-01T20:56:49.641Z"
        },
        {
            "DeploymentId": "f8e4c455-8ac4-453a-8252-512dc3e9c596",
            "DeploymentType": "NewDeployment",
            "GroupArn": "arn:aws:greengrass:us-west-2::123456789012:/greengrass/
groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/4ad66e5d-3808-446b-940a-
b1a788898382",
            "CreatedAt": "2019-07-01T20:41:47.048Z"
        },
        {
            "DeploymentId": "e4aca044-bbd8-41b4-b697-930ca7c40f3e",
            "DeploymentType": "NewDeployment",
            "GroupArn": "arn:aws:greengrass:us-west-2::123456789012:/greengrass/
groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/1f3870b6-850e-4c97-8018-
c872e17b235b",
            "CreatedAt": "2019-06-18T15:16:02.965Z"
        }
    ]
}
```

> **Note**
>
> これらの AWS CLI コマンドは、 グループとデプロイ ID のサンプル値を使用します。コマンドを実行するときは、サンプルの値を置き換えてください。

2. CreateDeployment を使用して、ターゲットデプロイを再デプロイします。デプロイタイプを `Redeployment` に設定します。例:

```
aws greengrass create-deployment --deployment-type Redeployment \
  --group-id 74d0b623-c2f2-4cad-9acc-ef92f61fcaf7 \
  --deployment-id f8e4c455-8ac4-453a-8252-512dc3e9c596
```

コマンドは、新しいデプロイの ARN と ID を返します。

```
{
    "DeploymentId": "f9ed02b7-c28e-4df6-83b1-e9553ddd0fc2",
    "DeploymentArn": "arn:aws:greengrass:us-west-2::123456789012:/greengrass/
groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/deployments/f9ed02b7-c28e-4df6-83b1-
e9553ddd0fc2"
}
```

3. GetDeploymentStatus を使用して、デプロイのステータスを取得します。

デプロイに関する 403 Forbidden エラーがログに記録される。

解決策： クラウド内の AWS IoT Greengrass コアのポリシーに、許可されたアクション`"greengrass:*"`として が含まれていることを確認します。

create-deployment コマンドを初めて実行すると、ConcurrentDeployment エラーが発生する。

解決策: デプロイが進行中である可能性があります。get-deployment-status を実行して、デプロイが作成されたかどうかを確認できます。作成済みでない場合は、デプロイを作成し直します。

エラー: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account.

解決策: デプロイに失敗した場合に、このエラーが表示されることがあります。Greengrass サービスロールが、現在の AWS リージョンの AWS アカウント に関連付けられていることを確認します。詳細については、the section called “サービスロールの管理 (CLI) ” または the section called “サービスロールの管理 (コンソール) ” を参照してください。

エラー: デプロイのダウンロードステップを実行できません。ダウンロード中のエラー: グループ定義ファイルのダウンロード中のエラー: ... x509: 証明書の有効期限が切れているか、まだ有効ではありません

解決策: デプロイに失敗した場合に、`runtime.log` にこのエラーが記録されることがあります。`x509: certificate has expired or is not yet valid` というメッセージを含む

`Deployment failed` エラーが発生した場合は、デバイスのクロックを確認してください。TLS 証明書と X.509 証明書は、IoT システムを構築するための安全な基盤を提供しますが、サーバーとクライアントでの正確な時間を必要とします。IoT デバイスは、サーバー証明書を使用する AWS IoT Greengrass 他の TLS サービスに接続しようとする前に、正しい時間 (15 分以内) が必要です。詳細については、公式ブログの「Using Device Time to Validate AWS IoT Server Certificates on the Internet of Things」を参照してください。 AWS

デプロイが完了しない。

解決策: 以下の操作を実行します。

- AWS IoT Greengrass デーモンがコアデバイスで実行されていることを確認します。コアデバイスのターミナルで以下のコマンドを実行し、デーモンが実行されているかどうかを確認します。必要ならばデーモンを起動します。

  1. デーモンが実行中かどうかを確認するには、以下を実行します。

     ```
     ps aux | grep -E 'greengrass.*daemon'
     ```

     出力に `root` で実行中の `/greengrass/ggc/packages/1.11.6/bin/daemon` のエントリが含まれていれば、デーモンは実行されています。

     パスのバージョンは、 AWS IoT Greengrass コアデバイスにインストールされている Core ソフトウェアのバージョンによって異なります。

  2. デーモンを開始するには、以下を実行します。

     ```
     cd /greengrass/ggc/core/
     sudo ./greengrassd start
     ```

- コアデバイスが接続されていて、コアの接続エンドポイントが適切に設定されていることを確認してください。

エラー: Unable to find java or java8 executables, or the error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: worker with <worker-id> failed to initialize with reason Installed Java version must be greater than or equal to 8

解決策: AWS IoT Greengrass コアでストリームマネージャーが有効になっている場合は、グループをデプロイする前に、コアデバイスに Java 8 ランタイムをインストールする必要があります。詳細については、ストリームマネージャーの要件を参照してください。 AWS IoT コンソールの [Default Group creation] (デフォルトのグループ作成) ワークフローを使用してグループを作成する場合、ストリームマネージャーはデフォルトで有効になります。

または、ストリームマネージャーを無効にしてから、グループをデプロイします。詳細については、「the section called “設定の構成 (コンソール)”」を参照してください。

デプロイが完了せず、runtime.log に複数の「wait 1s for container to stop」エントリが含まれる。

解決策： コアデバイスターミナルで次のコマンドを実行して、 AWS IoT Greengrass デーモンを再起動します。

```
cd /greengrass/ggc/core/
sudo ./greengrassd stop
sudo ./greengrassd start
```

デプロイが完了せず、**runtime.log** に「[ERROR]-Greengrass deployment error: failed to report deployment status back to cloud {"deploymentId": "<deployment-id>", "errorString": "Failed to initiate PUT, endpoint: https://<deployment-status>, error: Put https://<deployment-status>: proxyconnect tcp: x509: certificate signed by unknown authority"}」が含まれる

解決策: Greengrass コアが HTTPS プロキシ接続を使用するように設定されていて、プロキシサーバーの証明書チェーンがシステムで信頼されていない場合に、`runtime.log` にこのエラーが記録されることがあります。この問題を解決するには、ルート CA 証明書に証明書チェーンを追加します。Greengrass コアは、 AWS IoT Greengrassとの HTTPS および MQTT 接続で TLS 認証に使用される証明書プールに、このファイルの証明書を追加します。

次の例は、ルート CA 証明書ファイルに追加されたプロキシサーバー CA 証明書を示しています。

```
# My proxy CA
-----BEGIN CERTIFICATE-----
MIIEFTCCAv2gAwIQWgIVAMHSAzWG/5YVRYtRQOxXUTEpHuEmApzGCSqGSIb3DQEK
\nCwUAhuL9MQswCQwJVUzEPMAVUzEYMBYGA1UECgwP1hem9uLmNvbSBJbmMuMRww
... content of proxy CA certificate ...
+vHIRlt0e5JAm5\noTIZGoFbK82A0/nO7f/t5PSIDAim9V3Gc3pSXxCCAQoFYnui
GaPUlGk1gCE84a0X\n7Rp/lND/PuMZ/s8YjlkY2NmYmNjMCAXDTE5MTEyN2cM216
gJMIADggEPADf2/m45hzEXAMPLE=
-----END CERTIFICATE-----

# Amazon Root CA 1
-----BEGIN CERTIFICATE-----
MIIDQTCCAimgF6AwIBAgITBmyfz/5mjAo54vB4ikPmljZKyjANJmApzyMZFo6qBg
ADA5MQswCQYDVQQGEwJVUzEPMA0tMVT8QtPHRh8jrdkGA1UEChMGDV3QQDExBBKW
... content of root CA certificate ...
o/ufQJQWUCyziar1hem9uMRkwFwYVPSHCb2XV4cdFyQzR1KldZwgJcIQ6XUDgHaa
5MsI+yMRQ+hDaXJiobldXgjUka642M4UwtBV8oK2xJNDd2ZhwLnoQdeXeGADKkpy
rqXRfKoQnoZsG4q5WTP46EXAMPLE
-----END CERTIFICATE-----
```

デフォルトでは、ルート CA 証明書ファイルは `/greengrass-root/certs/root.ca.pem` にあります。コアデバイス上の場所を見つけるには、config.json の `crypto.caPath` のプロパティを確認します。

> **Note**
>
> *greengrass-root* は、デバイスに AWS IoT Greengrass Core ソフトウェアがインストールされているパスを表します。通常、これは /greengrass ディレクトリです。

エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: <path>.

解決策: *<path>* ディレクトリの所有者グループに、そのディレクトリに対する読み書きアクセス許可が付与されていることを確認します。

エラー: <list-of-function-arns> are configured to run as root but Greengrass is not configured to run Lambda functions with root permissions.

解決策: デプロイに失敗した場合に、runtime.log にこのエラーが記録されることがあります。Lambda 関数 AWS IoT Greengrass をルートアクセス許可で実行できるようにが設定されていることを確認します。greengrass_root/config/config.json で allowFunctionsToRunAsRoot の値を yes に変更するか、別のユーザー/グループとして実行するように Lambda 関数を変更します。詳細については、「the section called “root としての Lambda 関数の実行”」を参照してください。

エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user: user: unknown user ggc_user.

解決策： AWS IoT Greengrass グループのデフォルトのアクセスアイデンティティが標準システムアカウントを使用している場合、ggc_userユーザーとggc_groupグループがデバイスに存在する

必要があります。ユーザーとグループを追加する方法については、このステップを参照してください。表示されているとおりに正確に名前を入力してください。

エラー: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to mask greengrass root in overlay upper dir: failed to create mask device at directory <ggc-path>: file exists"}

解決策: デプロイに失敗した場合に、`runtime.log` にこのエラーが記録されることがあります。このエラーは、Greengrass グループの Lambda 関数がコアのファイルシステム内の `/usr` ディレクトリにアクセスできない場合に発生します。この問題を解決するには、ローカルボリュームリソースをグループに追加し、グループをデプロイします。リソースは次の条件を満たす必要があります。

- `/usr` を [ソースパス] および [送信先パス] として指定します。
- リソースを所有する Linux グループの OS グループアクセス許可を自動的に追加します。
- Lambda 関数と関連付けて、読み取り専用アクセスを許可します。

エラー: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: process start failed: container_linux.go:259: starting container process caused "process_linux.go:250: running exec setns process for init caused \"wait: no child processes\"".

解決策: デプロイに失敗した場合に、このエラーが表示されることがあります。デプロイを再試行します。

エラー: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

解決策: `systemd-resolved` を使用している場合にこのエラーが表示されることがあります。これにより、デフォルトで DNSSEC 設定が有効になります。その結果、多くのパブリックドメインは認識されません。 AWS IoT Greengrass エンドポイントに到達しようとするとホストが見つからないため、デプロイは `In Progress`状態のままになります。

この問題をテストするには、次のコマンドと出力を使用できます。エンドポイントの *`region`* プレースホルダーを AWS リージョンに置き換えます。

```
$ ping greengrass-ats.iot.region.amazonaws.com
ping: greengrass-ats.iot.region.amazonaws.com: Name or service not known
```

```
$ systemd-resolve greengrass-ats.iot.region.amazonaws.com
greengrass-ats.iot.region.amazonaws.com: resolve call failed: DNSSEC validation failed:
 failed-auxiliary
```

考えられる解決策の 1 つは、DNSSEC を無効にすることです。DNSSEC が `false` の場合、DNS ルックアップは DNSSEC で検証されません。詳細については、`systemd` の既知の問題を参照してください。

1. `/etc/systemd/resolved.conf` に `DNSSEC=false` を追加します。
2. `systemd-resolved` を再起動します。

`resolved.conf` および DNSSEC の詳細については、ターミナルで `man resolved.conf` を実行してください。

グループの作成と関数の作成に関する問題

AWS IoT Greengrass グループまたは Greengrass Lambda 関数の作成に関する問題のトラブルシューティングには、次の情報を使用します。

問題

- エラー: Your 'IsolationMode' configuration for the group is invalid.
- エラー: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid.
- エラー: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.
- エラー: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.
- エラー: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer.
- エラー: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer.
- エラー: Execution configuration for function with arn <function-arn> is not allowed.

エラー: Your 'IsolationMode' configuration for the group is invalid.

解決策: このエラーは、`function-definition-version` の `DefaultConfig` で `IsolationMode` 値がサポートされていない場合に発生します。サポートされている値は`GreengrassContainer` および `NoContainer` です。

エラー: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid.

解決策: このエラーは、`function-definition-version` の <function-arn> で `IsolationMode` 値がサポートされていない場合に発生します。サポートされている値は`GreengrassContainer` および `NoContainer` です。

エラー: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.

解決策: このエラーは、`MemorySize` 値を設定したときに、コンテナ化を使用せずに実行することを選択した場合に発生します。コンテナ化を使用せずに実行される Lambda 関数には、メモリ制限を

設定することができません。制限を削除するか、 AWS IoT Greengrass コンテナで実行するように Lambda 関数を変更できます。

エラー: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.

解決策: このエラーは、`AccessSysfs` を `true` に指定しているときに、コンテナ化なしで実行しようとした場合に発生します。コンテナ化を使用せずに実行する Lambda 関数は、ファイルシステムに直接アクセスできるようにコードを更新する必要があり、`AccessSysfs` を使用することはできません。`false` に の値を指定する`AccessSysfs`か、 AWS IoT Greengrass コンテナで実行する Lambda 関数を変更できます。

エラー: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer.

解決策： このエラーは、 AWS IoT Greengrass コンテナで実行している Lambda 関数`MemorySize`の制限を指定しなかったために発生します。エラーを解決するには、`MemorySize` の値を指定します。

エラー: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer.

解決策: コンテナ化を使用せずに Lambda 関数を実行する場合は、`Local.Device`、`Local.Volume`、`ML_Model.SageMaker.Job`、`ML_Model.S3_Object` または `S3_Object.Generic_Archive` リソースタイプにアクセスできません。これらのリソースタイプが必要な場合は、 AWS IoT Greengrass コンテナで を実行する必要があります。コンテナ化を使用しないで実行する場合は、Lambda 関数のコードを変更して、ローカルデバイスに直接アクセスすることもできます。

エラー: Execution configuration for function with arn <function-arn> is not allowed.

解決策: このエラーが発生するのは、`GGIPDetector` または `GGCloudSpooler` を使用してシステムの Lambda 関数を作成し、`IsolationMode` または `RunAs` 設定を指定した場合です。このシステムの Lambda 関数から `Execution` パラメータを削除する必要があります。

検出の問題

AWS IoT Greengrass Discovery サービスに関する問題のトラブルシューティングには、次の情報を使用します。

問題

- エラー: デバイスがメンバーになっているグループの数が多すぎます。デバイスを 10 個を超えるグループに含めることはできません。

エラー: デバイスがメンバーになっているグループの数が多すぎます。デバイスを 10 個を超えるグループに含めることはできません。

解決策: これは既知の制限です。クライアントデバイスは、最大 10 個のグループのメンバーにすることができます。

機械学習リソースの問題

機械学習リソースの問題のトラブルシューティングには、以下の情報が役立ちます。

問題

- InvalidMLModelOwner - GroupOwnerSetting は ML モデルリソースに提供されていますが、GroupOwner または GroupPermission がありません
- NoContainer 関数は、機械学習リソースをアタッチするときにアクセス権限を設定できません。<function-arn> は、リソースアクセスポリシーでアクセス権限 <ro/rw> を持つ機械学習リソース <resource-id> を参照します。

- 関数 <function-arn> は機械学習リソース <resource-id> を参照しますが、ResourceAccessPolicy とリソースの OwnerSetting のどちらにもアクセス権限がありません。
- 関数 <function-arn> は \"rw\" アクセス権限で機械学習リソース <resource-id> を参照しますが、リソース所有者設定の GroupPermission で許可されているのは \"ro\" のみです。
- NoContainer 関数 <function-arn> は、ネストされた送信先パスのリソースを参照します。
- Lambda <function-arn> は、同じグループ所有者 ID を共有することでリソース <resource-id> にアクセスします。

InvalidMLModelOwner - GroupOwnerSetting は ML モデルリソースに提供されていますが、GroupOwner または GroupPermission がありません

解決策: このエラーは、機械学習リソースに ResourceDownloadOwnerSetting オブジェクトが含まれていても、必須の `GroupOwner` または `GroupPermission` プロパティが定義されていない場合に表示されます。この問題を解決するには、不足しているプロパティを定義します。

NoContainer 関数は、機械学習リソースをアタッチするときにアクセス権限を設定できません。<function-arn> は、リソースアクセスポリシーでアクセス権限 <ro/rw> を持つ機械学習リソース <resource-id> を参照します。

解決策: コンテナ化されていない Lambda 関数が機械学習リソースに対する関数レベルのアクセス権限を指定した場合、このエラーが表示されます。コンテナ化されていない関数は、機械学習リソースに定義されているリソース所有者のアクセス権限からアクセス権限を継承する必要があります。この問題を解決するには、コンソールから [inherit resource owner permissions] (リソース所有者のアクセス許可を継承する) を選択するか、API を使用して Lambda 関数のリソースアクセスポリシーからアクセス権限を削除するかを選択します。

関数 <function-arn> は機械学習リソース <resource-id> を参照しますが、ResourceAccessPolicy とリソースの OwnerSetting のどちらにもアクセス権限がありません。

解決策: このエラーは、機械学習リソースへのアクセス権限が、アタッチされた Lambda 関数またはリソースに対して設定されていない場合に表示されます。この問題を解決するには、Lambda 関数の ResourceAccessPolicy プロパティか、またはリソースの OwnerSetting プロパティにアクセス権限を設定します。

関数 <function-arn> は \"rw\" アクセス権限で機械学習リソース <resource-id> を参照しますが、リソース所有者設定の GroupPermission で許可されているのは \"ro\" のみです。

解決策: このエラーは、アタッチされた Lambda 関数に定義されたアクセス権限が、機械学習リソースに対して定義されたリソース所有者のアクセス権限を超えた場合に表示されます。この問題を解決するには、Lambda 関数に対して制限のより厳しいアクセス権限を設定するか、リソース所有者の制限がより低いアクセス権限を設定します。

NoContainer 関数 <function-arn> は、ネストされた送信先パスのリソースを参照します。

解決策: コンテナ化されていない Lambda 関数にアタッチされた複数の機械学習リソースが同じ送信先パスまたはネストされた送信先パスを使用している場合に、このエラーが表示されます。この問題を解決するには、リソースに別の送信先パスを指定します。

Lambda <function-arn> は、同じグループ所有者 ID を共有することでリソース <resource-id> にアクセスします。

解決策: このエラーは、Lambda 関数の実行者 ID と、機械学習リソースのリソース所有者に同じ OS グループを指定しながら、リソースが Lambda 関数にアタッチされていない場合に `runtime.log`

に記録されます。この設定では、Lambda 関数に暗黙のアクセス権限が付与されます。このアクセス権限は、 AWS IoT Greengrass の認可なしでリソースにアクセスするために使用できます。

この問題を解決するには、プロパティの 1 つに別の OS グループを使用するか、機械学習リソースを Lambda 関数にアタッチします。

AWS IoT Greengrass Docker のコアの問題

以下の情報は、Docker コンテナで AWS IoT Greengrass コアを実行する際の問題のトラブルシューティングに役立ちます。

問題

- エラー: Unknown options: -no-include-email.
- 警告: IPv4 is disabled. Networking will not work.
- エラー: A firewall is blocking file Sharing between windows and the containers.
- エラー: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::<account-id>:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: *
- エラー: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use.
- エラー: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". 整合性を維持するには、GGC がクラッシュするため手動で再起動する必要があります。

エラー: Unknown options: -no-include-email.

解決策: `aws ecr get-login` コマンドを実行すると、このエラーが発生することがあります。AWS CLI 最新バージョンがインストールされていることを確認します (例: run: `pip install awscli --upgrade --user`) 。Windows を使用していて、MSI インストーラを使用して CLI をインストールした場合は、インストールプロセスを繰り返す必要があります。詳細については、「AWS Command Line Interface ユーザーガイド」の「Microsoft Windows に AWS Command Line Interface をインストールする」を参照してください。

警告: IPv4 is disabled. Networking will not work.

解決策： Linux コンピュータ AWS IoT Greengrass で を実行すると、この警告または同様のメッセージが表示されることがあります。このステップで説明しているように、IPv4 ネットワーク転送を有効にします。IPv4 転送が有効になっていない場合、 AWS IoT Greengrass クラウドデプロイと MQTT 通信は機能しません。詳細については、Docker ドキュメントの「Configure namespaced kernel parameters (sysctls) at runtime」を参照してください。

エラー: A firewall is blocking file Sharing between windows and the containers.

解決策: Windows コンピュータで Docker を実行すると、このエラーまたは `Firewall Detected` メッセージが表示されることがあります。このエラーは、仮想プライベートネットワーク (VPN) にサインインしていて、ネットワーク設定が原因で共有ドライブをマウントできない場合にも発生することがあります。このような場合は、VPN をオフにし、Docker コンテナを再実行します。

エラー: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::<account-id>:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: *

このエラーは、Amazon ECR リポジトリにアクセスするための十分な権限がない場合、`aws ecr get-login-password` コマンドの実行時に表示されることがあります。詳細については、「Amazon ECR ユーザーガイド」の「Amazon ECR Repository Policy Examples」(Amazon ECR リポジトリポリシーの例) および「1 つの Amazon ECR リポジトリにアクセスする」を参照してください。

エラー: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use.

解決策: このエラーは、コンテナ名が古いコンテナで使用されている場合に発生する可能性があります。この問題を解決するには、以下のコマンドを実行して古い Docker コンテナを削除します。

```
docker rm -f $(docker ps -a -q -f "name=aws-iot-greengrass")
```

エラー: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". 整合性を維持するには、GGC がクラッシュするため手動で再起動する必要があります。

解決策： このエラーは、Docker コンテナで実行されている AWS IoT Greengrass コアに `GreengrassContainer` Lambda 関数をデプロイしようとすると発生する`runtime.log`可能性があります。現在、Greengrass Docker コンテナにデプロイできるのは、`NoContainer` Lambda 関数のみです。

この問題を解決するには、Lambda 関数がすべて、`NoContainer` モードであることを確認し、新しいデプロイを開始します。次に、コンテナを起動するときに、既存の`deployment`ディレクトリを AWS IoT Greengrass コア Docker コンテナにバインドマウントしないでください。代わりに、空の `deployment` ディレクトリを所定の場所に作成して、Docker コンテナでそのディレクトリをバインドマウントします。これにより、新しい Docker コンテナは、`NoContainer` モードで実行されている Lambda 関数を備えた最新のデプロイを受け取ることができます。

詳細については、「the section called “Docker コンテナ AWS IoT Greengrass で を実行する”」を参照してください。

ログによるトラブルシューティング

Greengrass グループのログ設定を構成できます。ログを CloudWatch Logs に送信する、ログをローカルファイルシステムに保存する、またはその両方などです。問題のトラブルシューティングを行う場合に詳細情報を取得するには、ログレベルを一時的に `DEBUG` に変更します。ログ設定の変更は、グループをデプロイするときに有効になります。詳細については、「the section called “のログ記録を設定する AWS IoT Greengrass”」を参照してください。

ローカルファイルシステムでは、 は次の場所にログ AWS IoT Greengrass を保存します。ファイルシステム上のログを読み取るには、ルート権限が必要です。

greengrass-root`/ggc/var/log/crash.log`

AWS IoT Greengrass コアがクラッシュしたときに生成されたメッセージを表示します。

greengrass-root/ggc/var/log/system/runtime.log

失敗したコンポーネントに関するメッセージを示します。

greengrass-root/ggc/var/log/system/

証明書マネージャーや接続マネージャーなど、 AWS IoT Greengrass システムコンポーネントからのすべてのログが含まれます。ggc/var/log/system/ および のメッセージを使用することでggc/var/log/system/runtime.log、 AWS IoT Greengrass システムコンポーネントで発生したエラーを確認できます。

greengrass-root/ggc/var/log/system/localwatch/

Greengrass ログの CloudWatch Logs へのアップロードを処理する AWS IoT Greengrass コンポーネントのログが含まれます。CloudWatch で Greengrass ログを表示できない場合は、これらのログをトラブルシューティングに使用できます。

greengrass-root/ggc/var/log/user/

ユーザー定義 Lambda 関数のすべてのログが含まれています。このフォルダを確認して、ローカル Lambda 関数のエラーメッセージを検索します。

> **Note**
>
> デフォルトでは、*greengrass-root* は /greengrass ディレクトリです。書き込みディレクトリが設定されている場合には、ログはこのディレクトリにあります。

ログがクラウドに保存されるように設定されている場合は、CloudWatch Logs を使用してログメッセージを表示します。 crash.logは、 AWS IoT Greengrass コアデバイスのファイルシステムログにのみ存在します。

AWS IoT が CloudWatch にログを書き込むように設定されている場合は、システムコンポーネントが接続しようとしたときに接続エラーが発生するかどうか、それらのログを確認します AWS IoT。

AWS IoT Greengrass ログ記録の詳細については、「」を参照してくださいthe section called “ AWS IoT Greengrass ログでのモニタリング”。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェア v1.0 のログは *greengrass-root*/var/log ディレクトリに保存されます。

ストレージ問題のトラブルシューティング

ローカルファイルストレージがいっぱいになると、一部のコンポーネントが正常に動作しなくなる場合があります。

- ローカルシャドウの更新が実行されません。
- 新しい AWS IoT Greengrass コア MQTT サーバー証明書をローカルでダウンロードすることはできません。
- デプロイが失敗します。

ローカルで使用可能な空き領域のサイズを常に把握しておく必要があります。空き領域は、デプロイした Lambda 関数のサイズ、ログ記録の設定 (「the section called “ログによるトラブルシューティング”」を参照)、およびローカルに保存されているシャドウ数に基づいて計算できます。

メッセージのトラブルシューティング

でローカルに送信されるすべてのメッセージ AWS IoT Greengrass は、QoS 0 で送信されます。デフォルトでは、 はメッセージをインメモリキューに AWS IoT Greengrass 保存します。したがって、グループのデプロイ後やデバイスの再起動後などに Greengrass コアを再起動すると、未処理のメッセージは失われます。ただし、 AWS IoT Greengrass (v1.6 以降) を設定して、コアの再起動後もメッセージが保持されるようにファイルシステムにキャッシュできます。また、キューサイズを設定することもできます。キューサイズを設定する場合、262144 バイト (256 KB) 以上の値になるように注意してください。そうしないと、正しく起動しない AWS IoT Greengrass 可能性があります。詳細については、「the section called “MQTT メッセージキュー”」を参照してください。

> **Note**
>
> デフォルトのインメモリキューを使用する場合、サービスの中断が低いときにグループのデプロイあるいはデバイスの再起動を行うことが推奨されます。

また、 AWS IoTとの永続セッションを確立するようにコアを設定することもできます。これにより、コアはオフライン AWS クラウド 中に から送信されたメッセージを受信できます。詳細については、「the section called “AWS IoT Coreを使用した MQTT 永続セッション”」を参照してください。

シャドウ同期タイムアウト問題のトラブルシューティング

Greengrass コアデバイスとクラウドとの間で通信が大幅に遅延すると、タイムアウトによりシャドウの同期が失敗する場合があります。この場合は、次のようなログエントリが表示されます。

```
[2017-07-20T10:01:58.006Z][ERROR]-cloud_shadow_client.go:57,Cloud shadow
 client error: unable to get cloud shadow what_the_thing_is_named for
 synchronization. Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/
what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded
 while awaiting headers)
[2017-07-20T10:01:58.006Z][WARN]-sync_manager.go:263,Failed to get cloud
 copy: Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/
what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded
 while awaiting headers)
[2017-07-20T10:01:58.006Z][ERROR]-sync_manager.go:375,Failed to execute sync operation
 {what_the_thing_is_named VersionDiscontinued []}"
```

解決方法としては、コアデバイスでホストのレスポンスを待機する時間を設定します。*greengrass-root*/config の config.json ファイルを開き、タイムアウトの値を秒単位で指定する system.shadowSyncTimeout フィールドを追加します。例:

```
{
  "system": {
    "shadowSyncTimeout": 10
  },
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud.pem.crt",
    "keyPath": "cloud.pem.key",
    ...
  },
  ...
}
```

config.json に shadowSyncTimeout 値が指定されていない場合は、デフォルト値の 5 秒が使用されます。

> **Note**
>
> AWS IoT Greengrass Core ソフトウェア v1.6 以前では、デフォルトshadowSyncTimeoutは 1 秒です。

AWS re:Post を確認する

このトピックのトラブルシューティング情報を使用して問題を解決できない場合は、 を検索するトラブルシューティングか、AWS IoT GreengrassAWS re:Post のタグで関連する問題を確認したり、新しい質問を投稿したりできます。 AWS IoT Greengrass チームのメンバーは AWS re:Post を積極的にモニタリングします。

のドキュメント履歴 AWS IoT Greengrass

次の表に、2018 年 6 月以降の AWS IoT Greengrass デベロッパーガイドの重要な変更点を示します。このドキュメントの更新に関する通知を受け取るには、RSS フィードにサブスクライブできます。

変更	説明	日付
v1.11.x スナップのサポート終了に関する更新	snapcraft.io の AWS IoT Greengrass core v 1.11.x スナップのサポート終了情報を更新しました。	2023 年 9 月 22 日
v1.11.x スナップのサポート終了	AWS IoT Greengrass core v 1.11.x スナップのサポート終了情報を snapcraft.io に追加しました。	2023 年 9 月 19 日
v1.11.6 AWS IoT Greengrass の Docker イメージ	AWS IoT Greengrass Core ソフトウェア v1.11.6 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) と Docker Hub で利用できます。常に最新バージョンの使用をお勧めします。	2022 年 4 月 12 日
AWS IoTAWS IoT Greengrass V1 非推奨の Device Tester (IDT)	IDT for AWS IoT Greengrass V1 は、署名付き認定レポートを生成しなくなります。	2022 年 4 月 4 日
の AWS IoT Device Tester のサポート更新 AWS IoT Greengrass	IDT for AWS IoT Greengrass バージョン 4.4.1 では、デバイスの認定に AWS IoT Greengrass コアソフトウェアバージョン v1.11.6 の使用がサポートされるようになりました。	2022 年 3 月 24 日

AWS IoT Greengrass バージョン 1.11.6 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.6 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2022 年 3 月 24 日
IoT SiteWise コネクタバージョン 12 のリリース	IoT SiteWise コネクタのバージョン 12 が利用できるようになりました。このリリースには、バグ修正が含まれています。	2021 年 12 月 23 日
v1.11.5 および AWS IoT Greengrass v1.10.5 用の Docker イメージ	AWS IoT Greengrass Core ソフトウェア v1.11.5 および v1.10.5 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) および Docker Hub で利用できます。常に最新バージョンの使用をお勧めします。	2021 年 12 月 22 日
AWS IoT Greengrass V1 メンテナンスポリシー	AWS IoT Greengrass V1 メンテナンスポリシーは、 AWS IoT Greengrass V1 サービスと AWS IoT Greengrass コアソフトウェア v1.x のさまざまなレベルのメンテナンスと更新を定義します。	2021 年 12 月 20 日

AWS IoT Device Tester バージョン 4.4.1 をリリース	IDT for AWS IoT Greengrass バージョン 4.4.1 が利用可能になりました。このリリースには AWS IoT Greengrass 、認定スイート (GGQ) v1.3.1 が含まれており、デバイスの認定に AWS IoT Greengrass コアソフトウェアバージョン v1.11.5 および v1.10.5 の使用をサポートしています。	2021 年 12 月 20 日
AWS IoT Greengrass バージョン 1.11.5 および 1.10.5 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.5 および 1.10.5 を利用できます。これらのバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 12 月 12 日
v1.11.4 AWS IoT Greengrass および v1.10.4 Docker イメージを再公開しました	AWS IoT Greengrass Core ソフトウェアバージョン 1.11.4 および 1.10.4 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) と Docker Hub に再公開され、BusyBox によるバグ修正に対処しました。最新の Docker イメージを使用するには、`1.11.4-1` または `1.10.4-1` タグを使用してください。使用できるタグの詳細については、Docker Hub の amazon/aws-iot-Greengras を参照してください。	2021 年 12 月 8 日

CloudWatch Metrics コネクタが入力データの重複タイムスタンプに対応	タイムスタンプが重複している入力データを CloudWatch Metrics コネクタに送信できるようになりました。	2021 年 11 月 19 日
サービス間の混乱した代理防止の更新	AWS IoT Greengrass は、混乱した代理問題を防ぐために、IAM リソースポリシーで `aws:SourceArn` および `aws:SourceAccount` グローバル条件コンテキストキーの使用をサポートしています。	2021 年 11 月 1 日
v1.11.4 AWS IoT Greengrass の Docker イメージ	AWS IoT Greengrass Core ソフトウェア v1.11.4 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) と Docker Hub で利用できます。常に最新バージョンの使用をお勧めします。	2021 年 8 月 24 日
v1 AWS IoT Greengrass .11.4 スナップを公開	AWS IoT Greengrass スナップのバージョン 1.11.4 が利用可能になりました。常に最新バージョンの使用をお勧めします。	2021 年 8 月 20 日
の AWS IoT Device Tester のサポート更新 AWS IoT Greengrass	IDT for AWS IoT Greengrass バージョン 4.1.0 では、デバイスの認定に AWS IoT Greengrass コアソフトウェアバージョン v1.11.4 の使用がサポートされるようになりました。	2021 年 8 月 18 日

AWS IoT Greengrass バージョン 1.11.4 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.4 が利用可能です。このリリースでは、以前のバージョンの AWS IoT Greengrass Core ソフトウェアから v1.11.3 へのアップグレードが妨げられたストリームマネージャーの問題が修正されました。常に最新バージョンの使用をお勧めします。	2021 年 8 月 17 日
VPC エンドポイント (AWS PrivateLink)	AWS IoT Greengrass が AWS IoT Greengrass コントロールプレーンのインターフェイス VPC エンドポイント (AWS PrivateLink) をサポートするようになりました。これで、VPC と AWS IoT Greengrass コントロールプレーンとの間で、プライベート接続を確立できるようになります。	2021 年 8 月 16 日
AWS IoT Device Tester バージョン 4.1.0 をリリース	AWS IoT Device Tester for AWS IoT Greengrass のバージョン 4.1.0 が利用可能になりました。このバージョンでは、デバイスの認定に AWS IoT Greengrass コアソフトウェアバージョン 1.11.3 および 1.10.4 の使用がサポートされています。	2021 年 6 月 23 日

v1 AWS IoT Greengrass .11.3 スナップを公開	AWS IoT Greengrass スナップのバージョン 1.11.3 には、パフォーマンスの向上とバグ修正が含まれています。ベストプラクティスとして、常に最新のバージョンを実行することをお勧めします。	2021 年 6 月 15 日
v1.11.3 AWS IoT Greengrass および v1.10.4 用の Docker イメージをリリース	AWS IoT Greengrass Core ソフトウェア v1.11.3 および v1.10.4 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) および Docker Hub で利用できます。これらのバージョンの AWS IoT Greengrass Core には、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 6 月 15 日
AWS IoT Greengrass バージョン 1.11.3 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.3 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 6 月 14 日

AWS IoT Greengrass バージョン 1.10.4 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.10.4 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 6 月 14 日
Modbus-TCP プロトコルアダプタバージョン 2 のリリース	Modbus-TCP プロトコルアダプタコネクタのバージョン 2 が利用できるようになりました。このリリースでは、ASCII、UTF8、および ISO8859 エンコードされたソース文字列のサポートのサポートが追加されました。	2021 年 5 月 24 日
Docker アプリケーションデプロイコネクタバージョン 7 のリリース	Greengrass Docker アプリケーションデプロイコネクタのバージョン 7 が利用できるようになりました。	2021 年 4 月 5 日
AWS IoT Greengrass バージョン 1.11.1 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.1 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 3 月 29 日

AWS IoT Device Tester バージョン 4.0.2 がリリースされました	AWS IoT Device Tester for のバージョン 4.0.2 が利用可能 AWS IoT Greengrass になりました。このバージョンは IDT v4.0.0 を置き換え、 AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.1 のサポートを追加します。このバージョンを導入すると、IDT でハードウェアセキュリティ統合 (HSI) のエラーを認識できない原因となった問題も修正されます。	2021 年 3 月 29 日
IoT SiteWise コネクタバージョン 11 のリリース	IoT SiteWise コネクタのバージョン 11 が利用できるようになりました。これにより、隠し文字や印刷できない文字を含む文字列がサポートされるようになります。また、このリリースには一般的なパフォーマンスの向上とバグ修正も含まれています。	2021 年 3 月 24 日
v1 AWS IoT Greengrass .11.0 スナップを再公開	AWS IoT Greengrass スナップバージョン 1.11.0 は、Python インタープリタを使用する際のバグ修正とアプリケーションのクラッシュの可能性に対処するために Snapcraft に再公開されました。 AWS IoT Greengrass は、ソフトウェアバージョン 1.10 および 1.9 のスナップを提供しません。	2021 年 3 月 19 日

の AWS IoT Device Tester のサポート更新 AWS IoT Greengrass	IDT for AWS IoT Greengrass バージョン 4.0.0 では、デバイスの認定に AWS IoT Greengrass コアソフトウェアバージョン v1.10.3 の使用がサポートされるようになりました。	2021 年 3 月 18 日
v1 AWS IoT Greengrass .8.4 スナップを再公開	AWS IoT Greengrass スナップバージョン 1.8.4 が Snapcraft に再公開され、Python インタープリタの使用時にバグ修正とアプリケーションのクラッシュの可能性に対処しました。	2021 年 3 月 15 日
Armv7l AWS IoT Greengrass 用の v1.11.0 Docker イメージを再公開しました	Armv7l プラットフォーム用の AWS IoT Greengrass Core ソフトウェアバージョン 1.11.0 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) と Docker Hub に再公開され、Python インタープリタを使用する際のバグ修正とアプリケーションのクラッシュの可能性に対処しました。	2021 年 3 月 8 日
AWS IoT Greengrass v1.10.3 Docker イメージのリリース	AWS IoT Greengrass Core ソフトウェアバージョン 1.10.3 の Docker イメージが Amazon Elastic Container Registry (Amazon ECR) と Docker Hub で利用可能になりました。	2021 年 3 月 8 日

v1 AWS IoT Greengrass .11.0 および v1.9.4 Docker イメージを再公開しました	AWS IoT Greengrass Core ソフトウェアバージョン 1.11.0 および 1.9.4 の Docker イメージは、Amazon Elastic Container Registry (Amazon ECR) と Docker Hub に再公開され、Python インタープリタを使用する際のバグ修正とアプリケーションのクラッシュの可能性に対処しました。Armv7l 対応の Docker イメージは、現時点では再公開されていません。	2021 年 2 月 26 日
AWS IoT Greengrass バージョン 1.10.3 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.10.3 が利用可能です。このバージョンでは、`systemComponentAuthTimeout` コア設定プロパティが追加され、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2021 年 2 月 24 日

IoT SiteWise コネクタバージョン 10 のリリース	IoT SiteWise コネクタのバージョン 10 が利用できるようになりました。このリリースでは、接続が失われたときの StreamManager クライアントの安定性の問題を解決し、`SourceTimestamp` がない場合の OPC-UA 値の処理を改善しました。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2021 年 1 月 22 日
IoT SiteWise コネクタバージョン 9 のリリース	IoT SiteWise コネクタのバージョン 9 が利用できるようになりました。これにより、カスタム Greengrass `StreamManager` ストリームの送信先、OPC-UA のデッドバンド、カスタムスキャンモード、カスタムスキャンレートのサポートが開始されます。また、IoT SiteWise ゲートウェイから行われた設定更新時のパフォーマンスの向上も含まれます。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2020 年 12 月 15 日

AWS IoT Device Tester バージョン 4.0.0 をリリース	AWS IoT Device Tester for のバージョン 4.0.0 が利用可能 AWS IoT Greengrass になりました。このバージョンでは、IDT を使用して、デバイス検証用のカスタムテストスイートを開発および実行できます。これには、macOS および Windows 用のコード署名付き IDT アプリケーションも含まれています。	2020 年 12 月 15 日
AWS IoT Greengrass スナップ v1.11	AWS IoT Greengrass スナップのバージョン 1.11.0 では、コンテナ化されていない Lambda 関数がサポートされています。ベストプラクティスとして、常に最新のバージョンを実行することをお勧めします。	2020 年 12 月 6 日
IoT SiteWise コネクタバージョン 8 のリリース	IoT SiteWise コネクタのバージョン 8 が使用可能です。このリリースでは、コネクターのネットワーク接続が断続的に発生する場合の安定性を改善しました。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2020 年 11 月 19 日
Kinesis Firehose コネクタによる [No container] (コンテナなし) モードのサポート	コネクタのコンテナ化モードを設定するための `Isolation Mode` パラメータを使用できます。	2020 年 10 月 19 日

Docker アプリケーションデプロイコネクタバージョン 6 のリリース	Greengrass Docker アプリケーションデプロイコネクタのバージョン 6 が利用できるようになりました。	2020 年 9 月 18 日
AWS IoT Greengrass バージョン 1.11.0 をリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.11.0 が利用可能です。このバージョンでは、システムヘルスのテレメトリ機能とローカルヘルスチェック API が追加されています。ストリームマネージャーは、Amazon Simple Storage Service (Amazon S3) および IoT SiteWise へのデータのエクスポートに対応しました。このバージョンには、パフォーマンスの向上とバグ修正も含まれています。常に最新バージョンの使用をお勧めします。	2020 年 9 月 16 日
IoT SiteWise コネクタバージョン 7 のリリース	IoT SiteWise コネクタのバージョン 7 が使用可能です。このリリースでは、ゲートウェイメトリックスに関する問題が修正されています。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2020 年 8 月 14 日
ServiceNow MetricBase 統合、Splunk 統合、および Twilio 通知コネクタによるコンテナなしモードのサポート	コネクタのコンテナ化モードを設定するための `Isolation Mode` パラメータを使用できます。	2020 年 7 月 30 日

SNS コネクタによる [No container] (コンテナなし) モードのサポート	コネクタのコンテナ化モードを設定するための `Isolation Mode` パラメータを使用できます。	2020 年 7 月 6 日
CloudWatch Metrics コネクタが [No container] (コンテナなし) モードに対応	コネクタのコンテナ化モードを設定するための `Isolation Mode` パラメータを使用できます。	2020 年 6 月 17 日
AWS IoT Greengrass バージョン 1.10.2 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.10.2 が利用可能です。このバージョンでは、`mqttOperationTimeout` コア設定プロパティが追加され、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2020 年 6 月 8 日
Tensorflow 機械学習インストーラの廃止	AWS IoT Greengrass Tensorflow のパッケージ化された機械学習インストーラは廃止されました。機械学習のサンプルが Python 3.7 にアップグレードされました。	2020 年 5 月 29 日
Chainer フレームワークのサポートと Greengrass 機械学習インストーラの廃止	AWS IoT Greengrass MXNet と DLR の事前パッケージ化された機械学習インストーラとダウンロードは廃止されました。Chainer フレームワークのサポートとそれに関連するダウンロードは廃止されました。	2020 年 5 月 4 日

IoT SiteWise コネクタバージョン 6 のリリース	IoT SiteWise コネクタのバージョン 6 が使用可能です。このリリースでは、CloudWatch メトリクスのサポートと、新しい OPC-UA タグの自動検出が追加されました。これにより、OPC-UA ソースのタグが変更されたときに、ゲートウェイを再起動する必要がなくなります。このバージョンのコネクタには、ストリームマネージャーと AWS IoT Greengrass Core ソフトウェア v1.10.0 以降が必要です。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2020 年 4 月 29 日
Python 3.7 にアップグレードされたコネクタ	Python ランタイムをサポートするコネクタが Python 3.7 にアップグレードされました。コネクタのバージョンをアップグレードし、Python 2.7 から Python 3.7 にすることをお勧めします。	2020 年 4 月 29 日
サイレントモードによる Greengrass デバイスのセットアップに対応	Greengrass デバイスのセットアップをサイレントモードで実行して、スクリプトが値の入力を要求しないようにできます。	2020 年 4 月 27 日

新しい Docker ベースイメージ	Alpine Linux (x86_64、Armv7l、または AArch64) ベースイメージ上に構築された AWS IoT Greengrass Docker イメージをダウンロードできます。	2020 年 4 月 23 日
AWS IoT Greengrass バージョン 1.10.1 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.10.1 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。常に最新バージョンの使用をお勧めします。	2020 年 4 月 16 日
セキュリティに関する新しい章	AWS IoT Greengrass セキュリティコンテンツが再編成され、新しい情報が追加されました。	2020 年 3 月 30 日
APT パッケージマネージャーを使用して をインストールする AWS IoT Greengrass	サポートされている Debian ベースの Linux ディストリビューションでは、 `apt`を使用して AWS IoT Greengrass Core ソフトウェアをデバイスにインストールできます。	2020 年 2 月 26 日

IoT SiteWise コネクタバージョン 5 のリリース	IoT SiteWise コネクタのバージョン 5 が使用可能です。このリリースでは、 AWS IoT Greengrass Core ソフトウェア v1.9.4 との互換性の問題が修正されました。IoT SiteWise コネクタは、IoT SiteWise でローカルデバイスおよび機器データをアセットプロパティに送信します。	2020 年 2 月 12 日
コアデバイスをすばやくセットアップするための新しいスクリプト	Greengrass デバイスのセットアップを使用して、コアデバイスを数分で構成できます。また、 は Node.js 12.x Lambda 関数をサポートする AWS IoT Greengrass ようになりました。	2019 年 12 月 20 日
AWS IoT Greengrass バージョン 1.10.0 をリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.10.0 が利用可能です。このバージョンの新機能には、ストリームマネージャー、Docker アプリケーションデプロイコネクタによるコンテナのサポート、機械学習リソースにアクセス可能なコンテナ化されていない Lambda 関数、AWS IoTによる MQTT 永続セッションのサポート、指定ポートでのローカルの MQTT トラフィックのサポートなどが含まれます。	2019 年 11 月 25 日

コンソールでのデプロイ通知のサポート	Amazon EventBridge コンソールを使用して、Greengrass グループのデプロイの状態が変更されたときにトリガーするイベントルールを作成します。	2019 年 11 月 14 日
AWS IoT Greengrass バージョン 1.9.4 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.9.4 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。ベストプラクティスとして、常に最新のバージョンを実行することをお勧めします。	2019 年 10 月 17 日
Greengrass サービスロールを管理するためのコンソールのサポート	AWS IoT コンソールの新機能と改善された機能を使用して、Greengrass サービスロールを管理します。	2019 年 10 月 4 日
グループレベルのタグを管理するためのコンソールのサポート	AWS IoT コンソールで、Greengrass グループのタグを作成、表示、管理できます。	2019 年 9 月 23 日
新しい機械学習コネクタ	ML フィードバックコネクタを使用してモデルの入力と予測を発行し、ML オブジェクト検出コネクタを使用してローカルオブジェクト検出推論サービスを実行します。	2019 年 9 月 19 日

AWS IoT Greengrass バージョン 1.9.3 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.9.3 が利用可能です。このバージョンでは、Armv6l アーキテクチャの Raspbian ディストリビューションに AWS IoT Greengrass Core ソフトウェアをインストールし、ALPN でポート 443 の OTA 更新をサポートし、Python 2.7 Lambda 関数から他の Lambda 関数に送信されるバイナリペイロードのバグ修正が含まれています。	2019 年 9 月 12 日
AWS IoT Greengrass バージョン 1.8.4 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.8.4 が利用可能です。このバージョンには、パフォーマンスの向上とバグ修正が含まれています。v1.8.x を実行している場合は、v1.8.4 または v1.9.3 にアップグレードすることをお勧めします。それより前のバージョンでは、v1.9.3 にアップグレードすることをお勧めします。	2019 年 8 月 30 日

AWS IoT Greengrass OpenWrt をサポートする バージョン 1.9.2 をリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.9.2 が利用可能です。このバージョンでは、Armv8 (AArch64) および Armv7l アーキテクチャの OpenWrt ディストリビューションに AWS IoT Greengrass Core ソフトウェアをインストールできます。	2019 年 6 月 20 日
AWS IoT Greengrass バージョン 1.8.3 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.8.3 が利用可能です。このバージョンには、全般的なパフォーマンスの向上とバグ修正が含まれています。v1.8.x を実行している場合は、v1.8.3 または v1.9.2 にアップグレードすることをお勧めします。それより前のバージョンでは、v1.9.2 にアップグレードすることをお勧めします。	2019 年 6 月 20 日
AWS IoT Greengrass バージョン 1.9.1 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.9.1 が利用可能です。このバージョンには、トピックにワイルドカード文字 AWS IoT を含む からのメッセージのバグ修正が含まれています。	2019 年 5 月 10 日

AWS IoT Greengrass バージョン 1.8.2 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.8.2 が利用可能です。このバージョンには、全般的なパフォーマンスの向上とバグ修正が含まれています。v1.8.x を実行している場合は、v1.8.2 または v1.9.0 にアップグレードすることをお勧めします。それより前のバージョンでは、v1.9.0 にアップグレードすることをお勧めします。	2019 年 5 月 2 日
AWS IoT Greengrass バージョン 1.9.0 をリリース	新機能: Python 3.7 と Node.js 8.10 Lambda ランタイムのサポート、MQTT 接続の最適化、およびローカルの MQTT サーバー向けの Elliptic Curve (EC) キーのサポート。	2019 年 5 月 1 日
AWS IoT Greengrass バージョン 1.8.1 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.8.1 が利用可能です。このバージョンには、全般的なパフォーマンスの向上とバグ修正が含まれています。ベストプラクティスとして、常に最新のバージョンを実行することをお勧めします。	2019 年 4 月 18 日
AWS IoT Greengrass snapcraft で利用可能な snap	AWS IoT Greengrass Snap Store アプリを使用すると、を使用して Linux デバイスにソフトウェアをすばやく設計、テスト、デプロイできます AWS IoT Greengrass。	2019 年 4 月 1 日

タグベースのアクセス許可を使用したより多くのアクセス制御のサポート	AWS Identity and Access Management (IAM) ポリシーのタグを使用して、 リソースへのアクセスを AWS IoT Greengrass 制御できます。	2019 年 3 月 29 日
IoT Analytics コネクタのリリース	IoT Analytics コネクタを使用して、ローカルデバイスデータを AWS IoT Analytics チャネルに送信します。	2019 年 3 月 15 日
Kinesis Firehose コネクタでのバッチ処理のサポート	Kinesis Firehose コネクタは、バッチ処理されたデータレコードを指定の間隔で Amazon Data Firehose に送信する処理をサポートします。	2019 年 3 月 15 日
AWS CloudFormationAWS IoT Greengrass リソースのサポート	AWS CloudFormation テンプレートを使用して、 AWS IoT Greengrass リソースを作成および管理します。	2019 年 3 月 15 日
AWS IoT Greengrass バージョン 1.8.0 のリリース	新機能: Lambda 関数用の設定可能なデフォルトアクセスアイデンティティ、ポート 443 経由の HTTPS トラフィックのサポート、および との MQTT 接続用の予測可能な名前付きクライアント IDs AWS IoT。	2019 年 3 月 7 日

AWS IoT Greengrass バージョン 1.7.1 および 1.6.1 のリリース	AWS IoT Greengrass Core ソフトウェアのバージョン 1.7.1 および 1.6.1 が利用可能です。これらのバージョンでは、Linux カーネルバージョン 3.17 以降が必要になります。Greengrass コアソフトウェアの任意のバージョンを実行しているお客様は、すぐにバージョン 1.7.1 にアップグレードすることをお勧めします。	2019 年 2 月 11 日
SageMaker AI Neo 深層学習ランタイム	SageMaker AI Neo 深層学習ランタイムは、SageMaker AI Neo 深層学習コンパイラによって最適化された機械学習モデルをサポートします。	2018 年 11 月 28 日
Docker コンテナ AWS IoT Greengrass で を実行する	Docker コンテナ AWS IoT Greengrass で を実行するには、コンテナ化なしで実行するように Greengrass グループを設定します。	2018 年 11 月 26 日
AWS IoT Greengrass バージョン 1.7.0 のリリース	新機能: Greengrass コネクタ、ローカルシークレットマネージャー、Lambda 関数の分離およびアクセス許可の設定、信頼セキュリティのハードウェアルート、ALPN またはネットワークプロキシを使用した接続、および Raspbian Stretch のサポート。	2018 年 11 月 26 日

AWS IoT Greengrass ソフトウェアのダウンロード	AWS IoT Greengrass Core ソフトウェア、 AWS IoT Greengrass Core SDK、 AWS IoT Greengrass Machine Learning SDK パッケージは、Amazon CloudFront 経由でドロードできます。	2018 年 11 月 26 日
AWS IoT の Device Tester AWS IoT Greengrass	AWS IoT Device Tester for AWS IoT Greengrass を使用して、CPU アーキテクチャ、カーネル設定、ドライバーがと連携することを確認します AWS IoT Greengrass。	2018 年 11 月 26 日
AWS CloudTrailAWS IoT Greengrass API コールのログ記録	AWS IoT Greengrass は、ユーザー AWS CloudTrail、ロール、または のサービスによって実行されたアクションを記録する AWS サービスである と統合されています AWS IoT Greengrass。	2018 年 10 月 29 日
NVIDIA Jetson TX2 の TensorFlow v1.10.1 のサポート	AWS IoT Greengrass が提供する NVIDIA Jetson TX2 用の TensorFlow プリコンパイル済みライブラリは、TensorFlow v1.10.1 を使用するようになりました。ここでは、Jetpack 3.3 および CUDA Toolkit 9.0 がサポートされます。	2018 年 10 月 18 日
MXNet v1.2.1 機械学習リソースのサポート	AWS IoT Greengrass は、MXNet v1.2.1 を使用してトレーニングされた機械学習モデルをサポートしています。	2018 年 8 月 29 日

AWS IoT Greengrass バージョン 1.6.0 のリリース	新機能: Lambda 実行可能ファイル、設定可能なメッセージキュー、設定可能な再接続の再試行間隔、/proc 下のボリュームリソース、設定可能な書き込みディレクトリ。	2018 年 7 月 26 日

以前の更新

次の表は、2018 年 7 月以前の AWS IoT Greengrass デベロッパーガイドの重要な変更点を示しています。

変更	説明	日付
AWS IoT Greengrass バージョン 1.5.0 がリリースされました	新機能: • クラウドでトレーニングされたモデルを使用したローカルな Machine Learning (ML) Inference。詳細については、「機械学習の推論を実行する」を参照してください。 • Greengrass Lambda 関数は、JSON に加えてバイナリ入力データをサポートするようになりました。 詳細については、「AWS IoT Greengrass Core のバージョン」を参照してください。	2018 年 3 月 29 日
AWS IoT Greengrass バージョン 1.3.0 がリリースされました	新機能: • クラウドにデプロイした Greengrass 更新ジョブを処理できる OTA (無線通信経由) 更新エージェント。詳細については、「AWS IoT Greengrass Core ソフトウェアの OTA 更新」を参照してください。 • Greengrass Lambda 関数からローカル周辺機器とリソースにアクセスします。詳細については、「Lambda 関数とコネクタを使用してローカルリソースにアクセスする」を参照してください。	2017 年 11 月 27 日

変更	説明	日付
AWS IoT Greengrass バージョン 1.1.0 がリリースされました	新機能: • デプロイされた AWS IoT Greengrass グループをリセットします。詳細については、「デプロイのリセット」を参照してください。 • Python 2.7 に加えて、Node.js 6.10 と Java 8 の Lambda ランタイムをサポートしました。	2017 年 9 月 20 日
AWS IoT Greengrass バージョン 1.0.0 がリリースされました	AWS IoT Greengrass は一般利用可能です。	2017 年 6 月 7 日