开发人员指南

AWS Cloud Development Kit (AWS CDK) v2

版本 2

AWS Cloud Development Kit (AWS CDK) v2: 开发人员指南

Table of Contents

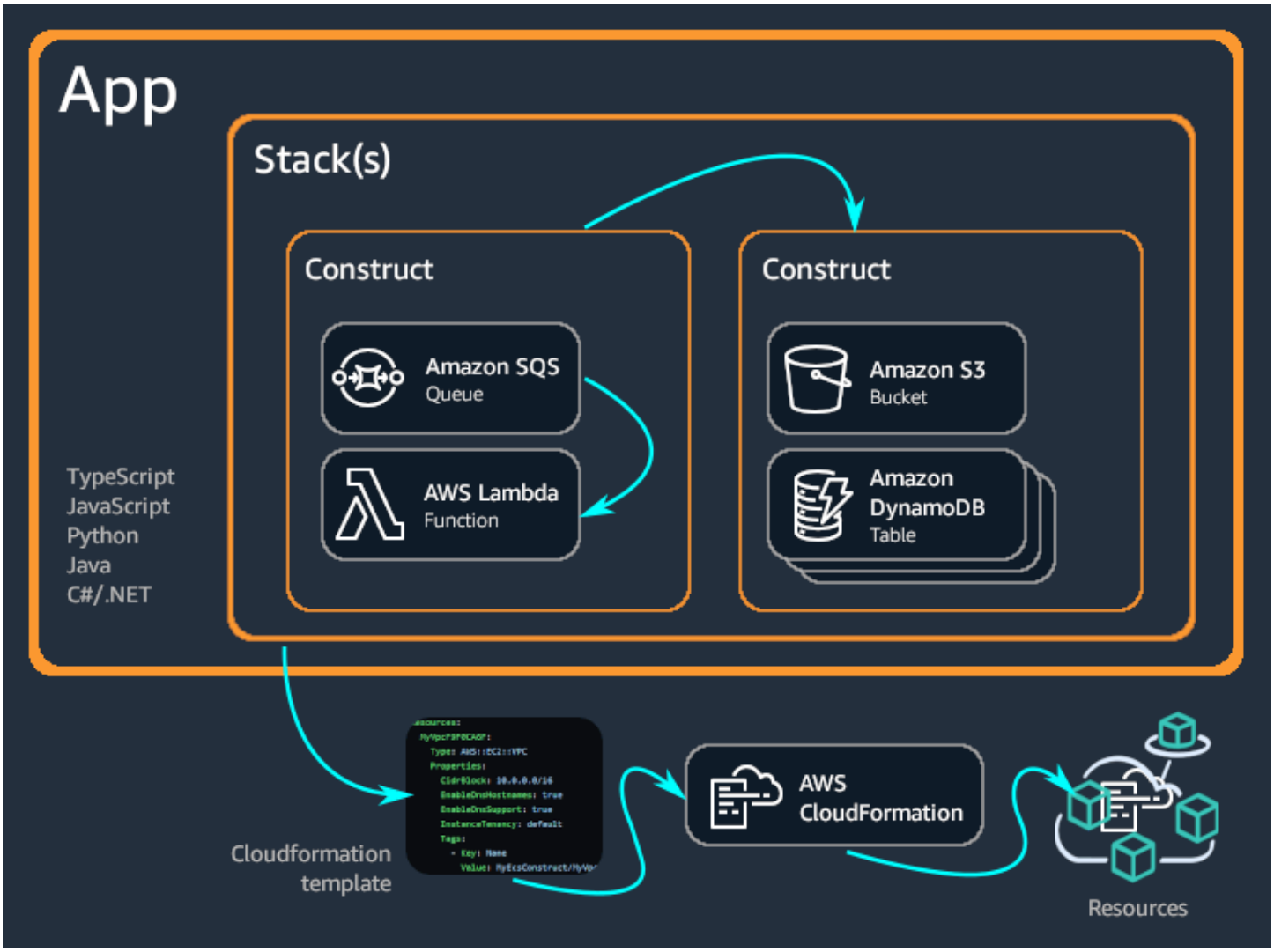
那是什么 AWS CDK？

AWS Cloud Development Kit (AWS CDK) 是一个开源软件开发框架，用于在代码中定义云基础架构并通过它进行配置 AWS CloudFormation。

由两个主要部分 AWS CDK 组成：

- AWS CDK 构造库 — 一组预先编写的模块化和可重复使用的代码，称为构造，您可以使用、修改和集成这些代码来快速开发基础架构。Constru AWS CDK ct Library 的目标是降低在构建应用程序时定义和集成 AWS 服务所需的复杂性 AWS。
- AWS CDK 工具包 — 用于与 CDK 应用程序交互的命令行工具。使用该 AWS CDK 工具包创建、管理和部署您的 AWS CDK 项目。

AWS CDK 支持TypeScript、JavaScript、Python、JavaC#/.Net、和Go。您可以使用这些支持的编程语言中的任何一种来定义称为构造的可重复使用的云组件。你可以将它们组合成堆栈和应用程序。然后，将 CDK 应用程序部署到 AWS CloudFormation 以配置或更新资源。



主题

的好处 AWS CDK

利用编程语言的 AWS CDK 强大表现力，使用在云端开发可靠、可扩展、经济实惠的应用程序。这种方法有许多好处，包括：

开发和管理您的基础设施即代码 (IaC)

实践基础架构即代码，以编程、描述性和声明性的方式创建、部署和维护基础架构。使用 IaC，您可以像开发人员对待代码一样对待基础架构。这就形成了一种可扩展的结构化方法来管理基础架构。要了解有关 IaC 的更多信息，请参阅 AWS 白皮书简介中的基础设施即 DevOps 代码。

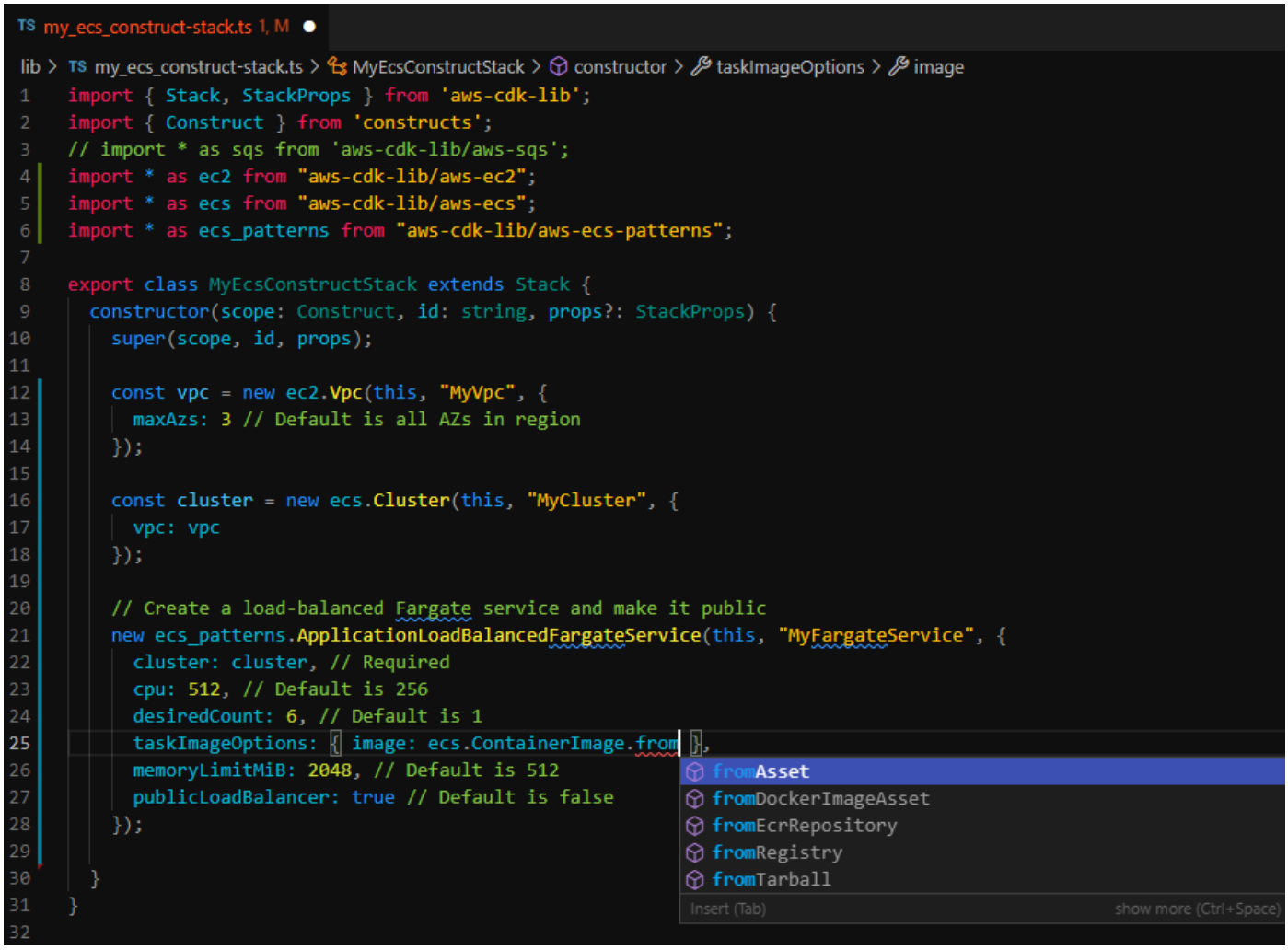
借助 AWS CDK，您可以将基础架构、应用程序代码和配置全部放在一个地方，确保在每个里程碑上都有一个完整的、可在云端部署的系统。采用软件工程最佳实践，例如代码审查、单元测试和源代码控制，使您的基础架构更加强大。

使用通用编程语言定义您的云基础架构

借助 AWS CDK，您可以使用以下任何一种编程语言来定义您的云基础架构：TypeScriptJavaScript、Python、Java、C#/.Net、和Go。选择您的首选语言，并使用参数、条件、循环、组合和继承等编程元素来定义基础设施的预期结果。

使用相同的编程语言来定义您的基础架构和应用程序逻辑。

享受在首选 IDE（集成开发环境）中开发基础架构的好处，例如语法高亮显示和智能代码完成。



通过以下方式部署基础架构 AWS CloudFormation

AWS CDK 与集成 AWS CloudFormation ，以便在上部署和配置您的基础架构 AWS。 AWS CloudFormation 是一种托管 AWS 服务 ，它为资源和属性配置提供广泛的支持，以便在上配置服务 AWS。使用 AWS CloudFormation，您可以以可预测的方式重复执行基础架构部署，并在出错时进行回滚。如果您已经熟悉了 AWS CloudFormation，那么在开始使用时，您不必学习新的 IaC 管理服务。 AWS CDK

使用构造快速开始开发应用程序

通过使用和共享称为构造的可重复使用的组件，加快开发速度。使用低级构造来定义单个 AWS CloudFormation 资源及其属性。使用高级结构快速定义应用程序的较大组件，为您的 AWS 资源设置合理、安全的默认值，用更少的代码定义更多的基础架构。

根据您的独特用例创建您自己的构造，并在您的组织内甚至与公众共享。

的示例 AWS CDK

以下是使用 AWS CDK 构造库创建具有启动类型的亚马逊弹性容器服务 (Amazon ECS) 服务的 AWS Fargate (Fargate) 示例。有关此示例的更多详细信息，请参阅the section called “ECS”。

TypeScript

```
export class MyEcsConstructStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}
```

JavaScript

```
class MyEcsConstructStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });
```

```
    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}

module.exports = { MyEcsConstructStack }
```

Python

```
class MyEcsConstructStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is False
```

Java

```
public class MyEcsConstructStack extends Stack {
```

```
    public MyEcsConstructStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyEcsConstructStack(final Construct scope, final String id,
            StackProps props) {
        super(scope, id, props);

        Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                .vpc(vpc).build();

        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                .cluster(cluster)
                .cpu(512)
                .desiredCount(6)
                .taskImageOptions(
                       ApplicationLoadBalancedTaskImageOptions.builder()
                               .image(ContainerImage
                                       .fromRegistry("amazon/amazon-ecs-sample"))
                               .build()).memoryLimitMiB(2048)
                .publicLoadBalancer(true).build();
    }
}
```

C#

```
public class MyEcsConstructStack : Stack
{
    public MyEcsConstructStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        var vpc = new Vpc(this, "MyVpc", new VpcProps
        {
            MaxAzs = 3
        });

        var cluster = new Cluster(this, "MyCluster", new ClusterProps
        {
            Vpc = vpc
```

```
        });

        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
        {
            Cluster = cluster,
            Cpu = 512,
            DesiredCount = 6,
            TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
            {
                Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
            },
            MemoryLimitMiB = 2048,
            PublicLoadBalancer = true,
        });
    }
}
```

Go

```
func NewMyEcsConstructStack(scope constructs.Construct, id string, props
 *MyEcsConstructStackProps) awscdk.Stack {

 var sprops awscdk.StackProps

 if props != nil {
  sprops = props.StackProps
 }

 stack := awscdk.NewStack(scope, &id, &sprops)

 vpc := awsec2.NewVpc(stack, jsii.String("MyVpc"), &awsec2.VpcProps{
  MaxAzs: jsii.Number(3), // Default is all AZs in region
 })

 cluster := awsecs.NewCluster(stack, jsii.String("MyCluster"), &awsecs.ClusterProps{
  Vpc: vpc,
 })

 awsecspatterns.NewApplicationLoadBalancedFargateService(stack,
 jsii.String("MyFargateService"),
  &awsecspatterns.ApplicationLoadBalancedFargateServiceProps{
   Cluster:        cluster,           // required
```

```
    Cpu:            jsii.Number(512),  // default is 256
    DesiredCount:   jsii.Number(5),    // default is 1
    MemoryLimitMiB: jsii.Number(2048), // Default is 512
    TaskImageOptions: &awsecspatterns.ApplicationLoadBalancedTaskImageOptions{
     Image: awsecs.ContainerImage_FromRegistry(jsii.String("amazon/amazon-ecs-
sample"), nil),
    },
    PublicLoadBalancer: jsii.Bool(true), // Default is false
   })

 return stack

}
```

该类生成的 AWS CloudFormation 模板超过 500 行。部署该 AWS CDK 应用程序会生成 50 多种以下类型的资源。

- AWS::EC2::EIP
- AWS::EC2::InternetGateway
- AWS::EC2::NatGateway
- AWS::EC2::Route
- AWS::EC2::RouteTable
- AWS::EC2::SecurityGroup
- AWS::EC2::Subnet
- AWS::EC2::SubnetRouteTableAssociation
- AWS::EC2::VPCGatewayAttachment
- AWS::EC2::VPC
- AWS::ECS::Cluster
- AWS::ECS::Service
- AWS::ECS::TaskDefinition
- AWS::ElasticLoadBalancingV2::Listener
- AWS::ElasticLoadBalancingV2::LoadBalancer
- AWS::ElasticLoadBalancingV2::TargetGroup
- AWS::IAM::Policy
- AWS::IAM::Role

- AWS::Logs::LogGroup

AWS CDK features

存储 AWS CDKGitHub库

有关官方 AWS CDK GitHub存储库的信息，请参阅 aws- cdk。在这里，您可以提交问题、查看我们的许可证、跟踪版本等。

由于 AWS CDK 它是开源的，因此团队鼓励您做出贡献，使其成为更好的工具。有关详细信息，请参阅贡献 AWS Cloud Development Kit (AWS CDK)。

AWS CDK API 参考资料

AWS CDK 构造库提供了 API 来定义您的 CDK 应用程序并向应用程序添加 CDK 结构。有关更多信息，请参阅 AWS CDK API 参考。

构造编程模型

构造编程模型 (CPM) 将背后的概念扩展 AWS CDK 到其他领域。使用 CPM 的其他工具包括：

- 适用于 T@@ erraform 的 CDK (cdkT F)
- 适用于 K@@ ubernetes 的 CDK (CDK8s)
- Projen，用于构建项目配置

构造中心

C onstruc t Hub 是一个在线注册表，您可以在其中查找、发布和共享开源 AWS CDK 库。

后续步骤

要开始使用 AWS CDK，请参阅开始使用 AWS CDK。

了解更多信息

要继续了解 AWS CDK，请参阅以下内容：

- AWS CDK 概念— 的重要概念和术语 AWS CDK.
- AWS CDK 研讨会 —动手研讨会，用于学习和使用 AWS CDK.
- AWS CDK 模式 — AWS CDK 由 AWS 专家为用户构建的 AWS 无服务器架构模式的开源集合。
- AWS CDK 代码示例 — 示例 AWS CDK 项目的GitHub存储库。
- cdk.dev — 社区驱动的中心 AWS CDK，包括社区工作区。Slack
- Awesome CDK — 包含 AWS CDK 开源项目、指南、博客和其他资源的精选列表的GitHub存储库。
- AWS 解决方案构造 — 经过审查的配置基础设施即代码 (IaC) 模式，可以轻松组装到生产就绪的应用程序中。
- AWS 开发者工具博客 —针对以下内容筛选的博客文章 AWS CDK.
- AWS CDK on Stack Overflow — 用 aws-cd k 标记的问题已开启。Stack Overflow
- AWS CDK 的教程 AWS Cloud9— 关于在 AWS Cloud9 开发环境中 AWS CDK 使用的教程。

要了解有关相关主题的更多信息 AWS CDK，请参阅以下内容：

- AWS CloudFormation 概念 — 由于 AWS CDK 是为使用而构建的 AWS CloudFormation，因此我们建议您学习和理解关键 AWS CloudFormation 概念。
- AWS 词汇表-使用的关键术语的定义 AWS。

要详细了解可用于简化无服务器应用程序开发和部署的相关工具，请参阅以下内容： AWS CDK

- AWS Serverless Application Model— 一种开源开发者工具，可简化和改善在上 AWS构建和运行无服务器应用程序的体验。
- AWSChalice— 用于在中编写无服务器应用程序的Python框架。

AWS CDK 概念

了解背后的核心概念 AWS Cloud Development Kit (AWS CDK).

AWS CDK 和 IaC

AWS CDK 是一个开源框架，您可以使用它来使用代码管理您的 AWS 基础架构。这种方法被称为基础架构即代码 (IaC)。通过将基础架构作为代码进行管理和配置，您可以像对待代码一样对待基础架构。这提供了许多好处，例如版本控制和可扩展性。要了解有关 IaC 的更多信息，请参阅什么是基础设施即代码？

AWS CDK 和 AWS CloudFormation

与 AWS CDK 紧密集成 AWS CloudFormation。 AWS CloudFormation 是一项完全托管的服务，您可以使用它来管理和配置您的基础架构 AWS。使用 AWS CloudFormation，您可以在模板中定义基础架构并将其部署到 AWS CloudFormation。然后，该 AWS CloudFormation 服务会根据您的模板上定义的配置来配置您的基础架构。

AWS CloudFormation 模板是声明性的，这意味着它们声明了基础架构的所需状态或结果。使用 JSON 或 YAML，您可以通过定义 AWS 资源和属性来声明您的 AWS 基础架构。资源代表上的许多服务 AWS ，属性代表您所需的这些服务配置。当您将模板部署到时 AWS CloudFormation，您的资源及其配置的属性将按照模板中的说明进行配置。

借助 AWS CDK，您可以使用通用编程语言强制管理您的基础架构。您可以定义达到所需状态所需的逻辑或顺序，而不仅仅是以声明方式定义所需的状态。例如，您可以使用`if`语句或条件循环来确定如何使您的基础架构达到所需的最终状态。

使用创建的基础架构最终会 AWS CDK 被转换，或者合成到 AWS CloudFormation 模板中，然后使用该 AWS CloudFormation 服务进行部署。因此，尽管 AWS CDK 提供了一种不同的方法来创建基础架构，但您仍然可以从中受益 AWS CloudFormation，例如广泛的 AWS 资源配置支持和强大的部署流程。

要了解更多信息 AWS CloudFormation，请参阅什么是 AWS CloudFormation？ 在《AWS CloudFormation 用户指南》中。

AWS CDK 和抽象

使用 AWS CloudFormation，您必须定义资源配置方式的每一个细节。这样做的好处是可以完全控制您的基础架构。但是，这需要您学习、理解和创建强大的模板，其中包含资源配置详细信息以及资源之间的关系，例如权限和事件驱动的交互。

借 AWS CDK助，您可以同样控制自己的资源配置。但是， AWS CDK 还提供了强大的抽象功能，可以加快和简化基础架构开发过程。例如， AWS CDK 包括提供合理默认配置的构造和为你生成样板代码的辅助方法。 AWS CDK 还提供诸如 AWS CDK 命令行界面 (AWS CDK CLI) 之类的工具，可为您执行基础架构管理操作。

详细了解核心 AWS CDK 概念

与... 互动 AWS CDK

与一起使用时 AWS CDK，您将主要与 AWS 构造库和进行交互 AWS CDK CLI。

与... 一起开发 AWS CDK

AWS CDK 可以用任何支持的编程语言编写。你从一个 CDK 项目开始，该项目包含文件夹和文件（包括资产）的结构。在项目中，您可以创建一个 CDK 应用程序。在应用程序中，您可以定义一个堆栈，它直接代表一个 CloudFormation 堆栈。在堆栈中，您可以使用构造来定义 AWS 资源和属性。

使用进行部署 AWS CDK

您可以将 CDK 应用程序部署到 AWS 环境中。在部署之前，您必须执行一次性引导以准备您的环境。

了解更多信息

要了解有关 AWS CDK 核心概念的更多信息，请参阅本节的主题。

支持的编程语言

对以下通用编程语言 AWS Cloud Development Kit (AWS CDK) 具有一流的支持：

- TypeScript
- JavaScript
- Python

- Java
- C#
- Go

理论上也可以使用其他JVM和.NETCLR语言，但我们目前不提供官方支持。

> **Note**
>
> 除此Go之外，本指南目前不包括其他说明或代码示例the section called “在 Go 中”。

AWS CDK 是用一种语言开发的，TypeScript。为了支持其他语言， AWS CDK 使用名为的工具JSII来生成语言绑定。

我们尝试提供每种语言的常用惯例，以使开发 AWS CDK 尽可能自然和直观。例如，我们使用您的首选语言的标准存储库分发 C AWS onstruct Library 模块，而您则使用该语言的标准包管理器进行安装。方法和属性也使用您的语言推荐的命名模式进行命名。

以下是一些代码示例：

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

Python

```
bucket = s3.Bucket("MyBucket", bucket_name="my-bucket", versioned=True,
            website_redirect=s3.RedirectTarget(host_name="aws.amazon.com"))
```

Java

```
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket")
                      .versioned(true)
                      .websiteRedirect(new RedirectTarget.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true,
                      WebsiteRedirect = new RedirectTarget {
                              HostName = "aws.amazon.com"
                      }});
```

Go

```
bucket := awss3.NewBucket(scope, jsii.String("MyBucket"), &awss3.BucketProps {
 BucketName: jsii.String("my-bucket"),
 Versioned: jsii.Bool(true),
 WebsiteRedirect: &awss3.RedirectTarget {
  HostName: jsii.String("aws.amazon.com"),
 },
})
```

> **Note**
>
> 这些代码片段仅用于说明目的。它们不完整，无法按原样运行。

AWS 构造库使用每种语言的标准包管理工具进行分发NPM，包括PyPi、Maven、和NuGet。我们还提供每种语言的 AWS CDK API 参考版本。

为了帮助您以首选语言使用，本指南包括以下有关支持的语言的主题： AWS CDK

- the section called “在 TypeScript”
- the section called “在 JavaScript”

- the section called “在 Python 中”
- the section called “在 Java 中”
- the section called “在 C# 中”
- the section called “在 Go 中”

TypeScript是支持的第一种语言 AWS CDK，而且大部分 AWS CDK 示例代码都是用编写的 TypeScript。本指南包括一个专门介绍如何调整TypeScript AWS CDK 代码以用于其他支持的语言的主题。有关更多信息，请参阅 与其他 AWS CDK 语言TypeScript进行比较。

AWS CDK 项目

AWS Cloud Development Kit (AWS CDK) 项目代表包含您的 CDK 代码的文件和文件夹。内容将因您的编程语言而异。

您可以手动创建 AWS CDK 项目，也可以使用 AWS CDK 命令行界面 (AWS CDK CLI) `cdk init` 命令创建项目。在本主题中，我们将介绍 AWS CDK CLI 创建的文件和文件夹的项目结构和命名惯例。您可以自定义和组织 CDK 项目以满足您的需求。

> **Note**
>
> 随着时间的推移，创建的项目结构 AWS CDK CLI可能会因版本而异。

主题

- 通用文件和文件夹
- 特定语言的文件和文件夹

通用文件和文件夹

.git

如果您已`git`安装，则会 AWS CDK CLI自动为项目初始化Git存储库。该`.git`目录包含有关存储库的信息。

.gitignore

用于指定Git要忽略的文件和文件夹的文本文件。

README.md

文本文件，为您提供管理 AWS CDK 项目的基本指导和重要信息。根据需要修改此文件以记录有关您的 CDK 项目的重要信息。

cdk.json

的配置文件 AWS CDK。此文件提供 AWS CDK CLI有关如何运行应用程序的说明。

特定语言的文件和文件夹

以下文件和文件夹是每种支持的编程语言所独有的。

TypeScript

以下是使用`cdk init --language typescript`命令在`my-cdk-ts-project`目录中创建的示例项目：

```
my-cdk-ts-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-ts-project.ts
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-ts-project-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### my-cdk-ts-project.test.ts
### tsconfig.json
```

.npmignore

该文件指定在将包发布到npm注册表时要忽略哪些文件和文件夹。此文件与类似`.gitignore`，但特定于npm软件包。

bin/. my-cdk-ts-project ts

应用程序文件定义了您的 CDK 应用程序。CDK 项目可以包含一个或多个应用程序文件。应用程序文件存储在bin文件夹中。

以下是定义 CDK 应用程序的基本应用程序文件示例：

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MyCdkTsProjectStack } from '../lib/my-cdk-ts-project-stack';

const app = new cdk.App();
new MyCdkTsProjectStack(app, 'MyCdkTsProjectStack');
```

jest.config.js

的配置文件Jest。 Jest是一个流行的JavaScript测试框架。

lib/ my-cdk-ts-project-stack.ts

堆栈文件定义了你的 CDK 堆栈。在堆栈中，您可以使用构造定义 AWS 资源和属性。

以下是定义 CDK 堆栈的基本堆栈文件示例：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class MyCdkTsProjectStack extends cdk.Stack {
 constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

节点模块

包含项目依赖项的Node.js项目中的常用文件夹。

package-lock.j

与文件配合使用以管理依赖关系版本的元数据package.json文件。

package.j

Node.js项目中常用的元数据文件。此文件包含有关您的 CDK 项目的信息，例如项目名称、脚本定义、依赖关系和其他导入项目级信息。

test/ .test.ts my-cdk-ts-project

将创建一个测试文件夹，用于组织您的 CDK 项目的测试。还创建了一个示例测试文件。

在运行测试之前，您可以在中编写测试TypeScript并使用Jest来编译TypeScript代码。

tsconfig.json

TypeScript项目中使用的配置文件，用于指定编译器选项和项目设置。

JavaScript

以下是使用`cdk init --language javascript`命令在`my-cdk-js-project`目录中创建的示例项目：

```
my-cdk-js-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-js-project.js
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-js-project-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### my-cdk-js-project.test.js
```

.npmignore

该文件指定在将包发布到npm注册表时要忽略哪些文件和文件夹。此文件与类似`.gitignore`，但特定于npm软件包。

bin/ .js my-cdk-js-project

应用程序文件定义了您的 CDK 应用程序。CDK 项目可以包含一个或多个应用程序文件。应用程序文件存储在bin文件夹中。

以下是定义 CDK 应用程序的基本应用程序文件示例：

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyCdkJsProjectStack } = require('../lib/my-cdk-js-project-stack');

const app = new cdk.App();
new MyCdkJsProjectStack(app, 'MyCdkJsProjectStack');
```

jest.config.js

的配置文件Jest。 Jest是一个流行的JavaScript测试框架。

lib/-stack.js my-cdk-js-project

堆栈文件定义了你的 CDK 堆栈。在堆栈中，您可以使用构造定义 AWS 资源和属性。

以下是定义 CDK 堆栈的基本堆栈文件示例：

```
const { Stack, Duration } = require('aws-cdk-lib');

class MyCdkJsProjectStack extends Stack {
 constructor(scope, id, props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}

module.exports = { MyCdkJsProjectStack }
```

节点模块

包含项目依赖项的Node.js项目中的常用文件夹。

package-lock.j

与文件配合使用以管理依赖关系版本的元数据package.json文件。

package.j

Node.js项目中常用的元数据文件。此文件包含有关您的 CDK 项目的信息，例如项目名称、脚本定义、依赖关系和其他导入项目级信息。

test/ .test.js my-cdk-js-project

将创建一个测试文件夹，用于组织您的 CDK 项目的测试。还创建了一个示例测试文件。

在运行测试之前，您可以在中编写测试JavaScript并使用Jest来编译JavaScript代码。

Python

以下是使用`cdk init --language python`命令在`my-cdk-py-project`目录中创建的示例项目：

```
my-cdk-py-project
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### my_cdk_py_project
#   ### __init__.py
#   ### my_cdk_py_project_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
    ### __init__.py
    ### unit
```

.venv

CDK CLI 会自动为您的项目创建虚拟环境。该`.venv`目录指的是这个虚拟环境。

app.py

应用程序文件定义了您的 CDK 应用程序。CDK 项目可以包含一个或多个应用程序文件。

以下是定义 CDK 应用程序的基本应用程序文件示例：

```
#!/usr/bin/env python3
```

```
import os

import aws_cdk as cdk

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

app = cdk.App()
MyCdkPyProjectStack(app, "MyCdkPyProjectStack")

app.synth()
```

my_cdk_py_project

包含您的堆栈文件的目录。CDK 在此处CLI创建以下内容：

- `__init__.py` — 一个空的Python包定义文件。
- `my_cdk_py_project`— 定义您的 CDK 堆栈的文件。然后，您可以使用构造定义堆栈中的 AWS 资源和属性。

以下是堆栈文件的示例：

```
from aws_cdk import Stack

from constructs import Construct

class MyCdkPyProjectStack(Stack):
 def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
  super().__init__(scope, construct_id, **kwargs)

  # code that defines your resources and properties go here
```

requirements-dev.txt

文件类似于`requirements.txt`，但用于管理专门用于开发目的而不是生产目的的依赖关系。

requirements.txt

Python项目中用于指定和管理项目依赖关系的常用文件。

source.bat

Batch 文件用于Windows设置Python虚拟环境。

测试

包含您的 CDK 项目测试的目录。

以下是单元测试的示例：

```
import aws_cdk as core
import aws_cdk.assertions as assertions

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

def test_sqs_queue_created():
 app = core.App()
 stack = MyCdkPyProjectStack(app, "my-cdk-py-project")
 template = assertions.Template.from_stack(stack)

 template.has_resource_properties("AWS::SQS::Queue", {
  "VisibilityTimeout": 300
 })
```

Java

以下是使用`cdk init --language java`命令在`my-cdk-java-project`目录中创建的示例项目：

```
my-cdk-java-project
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
    ### main
    ### test
```

pom.xml

包含有关您的 CDK 项目的配置信息和元数据的文件。此文件是其中的一部分Maven。

src/main

包含您的应用程序和堆栈文件的目录。

以下是示例应用程序文件：

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class MyCdkJavaProjectApp {
 public static void main(final String[] args) {
  App app = new App();

  new MyCdkJavaProjectStack(app, "MyCdkJavaProjectStack", StackProps.builder()
   .build());

  app.synth();
 }
}
```

以下是示例堆栈文件：

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCdkJavaProjectStack extends Stack {
 public MyCdkJavaProjectStack(final Construct scope, final String id) {
  this(scope, id, null);
 }

 public MyCdkJavaProjectStack(final Construct scope, final String id, final
 StackProps props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

src/test

包含您的测试文件的目录。以下是 示例：

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.assertions.Template;
import java.io.IOException;

import java.util.HashMap;

import org.junit.jupiter.api.Test;

public class MyCdkJavaProjectTest {

 @Test
 public void testStack() throws IOException {
  App app = new App();
  MyCdkJavaProjectStack stack = new MyCdkJavaProjectStack(app, "test");

  Template template = Template.fromStack(stack);

  template.hasResourceProperties("AWS::SQS::Queue", new HashMap<String, Number>()
 {{
   put("VisibilityTimeout", 300);
  }});
 }
}
```

C#

以下是使用`cdk init --language csharp`命令在`my-cdk-csharp-project`目录中创建的示例项目：

```
my-cdk-csharp-project
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### MyCdkCsharpProject
    ### MyCdkCsharpProject.sln
```

src/ MyCdkCsharpProject

包含您的应用程序和堆栈文件的目录。

以下是示例应用程序文件：

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyCdkCsharpProject
{
 sealed class Program
 {
  public static void Main(string[] args)
  {
   var app = new App();
   new MyCdkCsharpProjectStack(app, "MyCdkCsharpProjectStack", new StackProps{});
   app.Synth();
  }
 }
}
```

以下是示例堆栈文件：

```
using Amazon.CDK;
using Constructs;

namespace MyCdkCsharpProject
{
 public class MyCdkCsharpProjectStack : Stack
 {
  internal MyCdkCsharpProjectStack(Construct scope, string id, IStackProps props
 = null) : base(scope, id, props)
  {
   // code that defines your resources and properties go here
  }
 }
}
```

此目录还包含以下内容：

- `GlobalSuppressions.cs`— 用于抑制项目中特定的编译器警告或错误的文件。
- `.csproj`— 基于 XML 的文件，用于定义项目设置、依赖关系和生成配置。

src/ .sln MyCdkCsharpProject

Microsoft Visual Studio Solution File用于组织和管理相关项目。

Go

以下是使用`cdk init --language go`命令在`my-cdk-go-project`目录中创建的示例项目：

```
my-cdk-go-project
### .git
### .gitignore
### README.md
### cdk.json
### go.mod
### my-cdk-go-project.go
### my-cdk-go-project_test.go
```

go.mod

包含模块信息的文件，用于管理Go项目的依赖关系和版本控制。

my-cdk-go-project.go

定义您的 CDK 应用程序和堆栈的文件。

以下是 示例：

```
package main
import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type MyCdkGoProjectStackProps struct {
 awscdk.StackProps
}

func NewMyCdkGoProjectStack(scope constructs.Construct, id string, props
 *MyCdkGoProjectStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
```

```
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)
 // The code that defines your resources and properties go here

  return stack
}

func main() {
 defer jsii.Close()
 app := awscdk.NewApp(nil)
 NewMyCdkGoProjectStack(app, "MyCdkGoProjectStack", &MyCdkGoProjectStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })
 app.Synth(nil)
}

func env() *awscdk.Environment {

 return nil
}
```

my-cdk-go-project_test.go

定义示例测试的文件。

以下是 示例：

```
package main

import (
 "testing"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/assertions"
 "github.com/aws/jsii-runtime-go"
)

func TestMyCdkGoProjectStack(t *testing.T) {

 // GIVEN
 app := awscdk.NewApp(nil)
```

```
 // WHEN
 stack := NewMyCdkGoProjectStack(app, "MyStack", nil)

 // THEN
 template := assertions.Template_FromStack(stack, nil)
 template.HasResourceProperties(jsii.String("AWS::SQS::Queue"),
 map[string]interface{}{
  "VisibilityTimeout": 300,
 })
}
```

AWS CDK 应用程序

AWS Cloud Development Kit (AWS CDK) 应用程序或应用程序是一个或多个 CDK 堆栈的集合。堆栈是一个或多个构造的集合，用于定义 AWS 资源和属性。因此，堆栈和构造的整体分组被称为 CDK 应用程序。

主题

- 定义应用程序
- 构造树
- 应用程序生命周期

定义应用程序

您可以通过在项目的应用程序文件中定义应用程序实例来创建应用程序。为此，您需要导入并使用App构造库中的 AWS 构造。该App构造不需要任何初始化参数。它是唯一可以用作根的构造。

AWS 构造库中的App和Stack类是唯一的构造。与其他结构相比，它们不会自行配置 AWS 资源。相反，它们用于为您的其他构造提供上下文。所有代表 AWS 资源的构造都必须在Stack构造的范围内直接或间接地定义。 Stack构造是在App构造的范围内定义的。

然后对应用程序进行合成，为您的堆栈创建 AWS CloudFormation 模板。以下是 示例：

TypeScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
```

```
app.synth();
```

JavaScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

Python

```
app = App()
MyFirstStack(app, "hello-cdk")
app.synth()
```

Java

```
App app = new App();
new MyFirstStack(app, "hello-cdk");
app.synth();
```

C#

```
var app = new App();
new MyFirstStack(app, "hello-cdk");
app.Synth();
```

Go

```
app := awscdk.NewApp(nil)

MyFirstStack(app, "MyFirstStack", &MyFirstStackProps{
  awscdk.StackProps{
    Env: env(),
  },
})

app.Synth(nil)
```

单个应用程序中的堆栈可以轻松引用彼此的资源和属性。推 AWS CDK 断堆栈之间的依赖关系，以便可以按正确的顺序部署堆栈。您只需一个`cdk deploy`命令即可在应用程序中部署任何或全部堆栈。

构造树

构造是使用传递给每个构造的scope参数在其他构造中定义的，该参数以App类为根。通过这种方式，AWS CDK 应用程序定义了称为构造树的构造层次结构。

这棵树的根是你的应用程序，它是该App类的一个实例。在应用程序中，您可以实例化一个或多个堆栈。在堆栈中，你可以实例化构造，这些构造本身可以实例化资源或其他构造，依此类推。

构造总是在另一个构造的范围内明确定义的，这会在构造之间建立关系。几乎总是应传递this（在 Python 中self）作为作用域，表示新构造是当前构造的子构造。预期的模式是你从中派生构造Construct，然后实例化它在构造函数中使用的构造。

显式传递作用域允许每个构造将自身添加到树中，这种行为完全包含在Construct基类中。它在支持的每种语言中的工作方式都是一样的， AWS CDK 并且不需要额外的自定义。

> **Important**
>
> 从技术上讲，除了实例化构造this时之外，还可以传递一些作用域。你可以在树中的任何地方添加构造，甚至可以在同一个应用程序的另一个堆栈中添加构造。例如，你可以编写一个 mixin 风格的函数，将构造添加到作为参数传入的作用域中。这里的实际困难在于，你无法轻易地确保为构造选择的 ID 在其他人的范围内是唯一的。这种做法还会使你的代码更难理解、维护和重用。在不诉诸滥用论点的情况下，找到一种表达意图的方法几乎总是更好的scope。

AWS CDK 使用从树根到每个子构造的路径中所有构造的 ID 来生成所需的唯一 ID。 AWS CloudFormation这种方法意味着构造 ID 只需要在其范围内保持唯一性，而不是像在原生版本中那样在整个堆栈中保持唯一性 AWS CloudFormation。但是，如果您将构造移到不同的作用域，则其生成的堆栈唯一 ID 会发生变化，并且 AWS CloudFormation 不会将其视为同一个资源。

构造树与您在 AWS CDK 代码中定义的构造是分开的。但是，它可以通过任何构造的node属性进行访问，该属性是对树中表示该构造的节点的引用。每个节点都是一个Node实例，其属性提供对树根以及该节点的父作用域和子节点的访问权限。

1. node.children— 构造的直接子代。
2. node.id— 构造在其作用域内的标识符。
3. node.path— 构造的完整路径，包括其所有父项的 ID。
4. node.root— 构造树（应用程序）的根。

5. `node.scope`— 构造的作用域（父级），如果节点是根，则为未定义。
6. `node.scopes`— 构造的所有父级，直到根部。
7. `node.uniqueId`— 树中此构造的唯一字母数字标识符（默认情况下，由哈希生成）。`node.path`

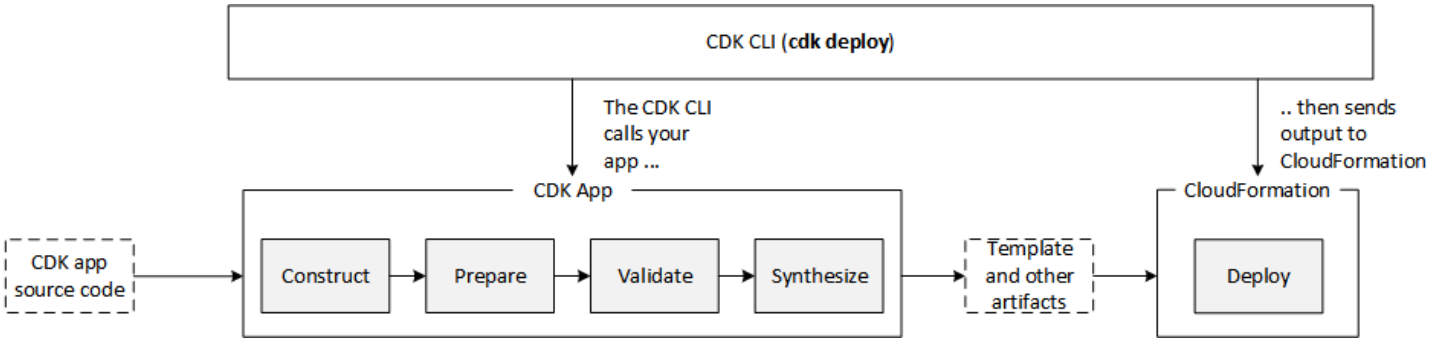
构造树定义了一种隐式顺序，在该顺序中，将构造合成最终 AWS CloudFormation 模板中的资源。其中一个资源必须在另一个资源之前创建， AWS CloudFormation 或者 AWS 构造库通常会推断依赖关系。然后，他们会确保按正确的顺序创建资源。

您也可以使用在两个节点之间添加显式依赖关系`node.addDependency()`。有关更多信息，请参阅 AWS CDK API 参考中的依赖关系。

AWS CDK 提供了一种简单的方法来访问构造树中的每个节点并对每个节点执行操作。有关更多信息，请参阅 the section called “方面”。

应用程序生命周期

部署 CDK 应用程序时，将进行以下几个阶段。这就是所谓的应用程序生命周期：



AWS CDK 应用程序在其生命周期中会经历以下几个阶段。

- 构造（或初始化）-您的代码实例化所有已定义的构造，然后将它们链接在一起。在这个阶段，所有的构造（应用程序、堆栈及其子构造）都被实例化并执行构造器链。您的大部分应用程序代码都是在此阶段执行的。
- 准备 — 所有实现了该`prepare`方法的构造都将参与最后一轮修改，以设置其最终状态。准备阶段是自动进行的。作为用户，您看不到此阶段的任何反馈。很少需要使用 “准备” 挂钩，通常不建议使用。在此阶段更改构造树时要非常小心，因为操作顺序可能会影响行为。
- 验证 — 所有实现了该`validate`方法的构造都可以进行自我验证，以确保它们处于可以正确部署的状态。在此阶段发生的任何验证失败时，您将收到通知。通常，我们建议尽快执行验证（通常是在收到一些输入后立即执行），并尽早抛出异常。尽早执行验证可以提高可靠性，因为堆栈跟踪将更加准确，并确保您的代码可以继续安全地执行。

- 合成 — 这是 AWS CDK 应用程序执行的最后阶段。它由对的调用触发`app.synth()`，它遍历构造树并在所有构造上调用该`synthesize`方法。实现的构造`synthesize`可以参与合成并向生成的云程序集发射部署工件。这些工件包括 AWS CloudFormation 模板、 AWS Lambda 应用程序包、文件和 Docker图像资产以及其他部署工件。 the section called “云端程序集”描述了此阶段的输出。在大多数情况下，您不需要实现该`synthesize`方法。
- 部署-在此阶段， AWS CDK CLI获取综合阶段生成的部署工件云组件，并将其部署到 AWS 环境中。它将资产上传到 Amazon S3 和 Amazon ECR，或者他们需要去的任何地方。然后，它开始 AWS CloudFormation 部署以部署应用程序并创建资源。

AWS CloudFormation 部署阶段开始时，您的 AWS CDK 应用程序已经完成并退出。这具有以下意义：

- AWS CDK 应用程序无法响应部署期间发生的事件，例如正在创建的资源或整个部署已完成。要在部署阶段运行代码，必须将其作为自定义资源注入到 AWS CloudFormation 模板中。有关向应用程序添加自定义资源的更多信息，请参阅AWS CloudFormation 模块或自定义资源示例。
- 该 AWS CDK 应用程序可能必须使用在运行时无法知道的值。例如，如果 AWS CDK 应用程序使用自动生成的名称定义了一个 Amazon S3 存储桶，而您检索了 `bucket.bucketName` (Python:`bucket_name`) 属性，则该值不是已部署存储桶的名称。相反，你会得到一个`Token`值。要确定特定值是否可用，请调用 `cdk.isUnresolved(value)` (Python:`is_unresolved`)。有关详细信息，请参阅 the section called “令牌”。

云端程序集

调用`app.synth()`是告诉从 AWS CDK 应用程序合成云程序集的原因。通常，您不会直接与云程序集交互。这些文件包含将应用程序部署到云环境所需的一切。例如，它包含应用程序中每个堆栈的 AWS CloudFormation 模板。它还包括您在应用程序中引用的任何文件资产或 Docker 镜像的副本。

有关如何格式化云程序集的详细信息，请参阅云装配规范。

要与您的 AWS CDK 应用程序创建的云程序集进行交互，您通常使用 AWS CDK CLI. 但是，任何可以读取云端汇编格式的工具都可用于部署您的应用程序。

运行你的应用程序

CDK CLI 需要知道如何执行您的 AWS CDK 应用程序。如果您使用`cdk init`命令从模板创建项目，则您的应用程序`cdk.json`文件将包含`app`密钥。此键为编写应用程序的语言指定必要的命令。如果您的语言需要编译，则命令行会在运行应用程序之前执行此步骤，因此您不能忘记执行此操作。

TypeScript

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-app.ts"
}
```

JavaScript

```
{
  "app": "node bin/my-app.js"
}
```

Python

```
{
    "app": "python app.py"
}
```

Java

```
{
  "app": "mvn -e -q compile exec:java"
}
```

C#

```
{
  "app": "dotnet run -p src/MyApp/MyApp.csproj"
}
```

Go

```
{
  "app": "go mod download && go run my-app.go"
}
```

如果您不是使用 CDK 创建项目CLI，或者想要覆盖中给出的命令行`cdk.json`，则可以在发出`cdk`命令时使用该`--app`选项。

```
$ cdk --app 'executable' cdk-command ...
```

该命令*####*部分表示应运行哪个命令来执行 CDK 应用程序。如图所示使用引号，因为此类命令包含空格。*cdk ##*是一个类似于synth或的子命令deploy，它告诉 CDK 你想用你的应用程序做CLI什么。接下来是该子命令所需的任何其他选项。

AWS CDK CLI也可以直接与已经合成的云组件进行交互。为此，请传递存储云程序集的目录--app。以下示例列出了存储在下的`./my-cloud-assembly`云组件中定义的堆栈。

```
$ cdk --app ./my-cloud-assembly ls
```

堆栈

AWS Cloud Development Kit (AWS CDK) 堆栈是一个或多个结构的集合，用于定义 AWS 资源。每个 CDK 堆栈代表您的 CDK 应用程序中的一个 AWS CloudFormation 堆栈。部署时，堆栈中的构造将作为一个单元（称为堆栈）进行 AWS CloudFormation 配置。要了解有关 AWS CloudFormation 堆栈的更多信息，请参阅《AWS CloudFormation 用户指南》中的使用堆栈。

由于 CDK 堆栈是通过 AWS CloudFormation 堆栈实现的，因此存在 AWS CloudFormation 配额和限制。要了解更多信息，请参阅AWS CloudFormation 配额。

主题

- 定义堆栈
- 使用 堆栈

定义堆栈

堆栈是在应用程序的上下文中定义的。您可以使用 AWS 构造库中的`Stack`类来定义堆栈。可以通过以下任何一种方式定义堆栈：

- 直接在应用程序的范围内。
- 由树中的任何构造间接获得。

以下示例定义了一个包含两个堆栈的 CDK 应用程序：

TypeScript

```
const app = new App();
```

```
new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

JavaScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

Python

```
app = App()

MyFirstStack(app, 'stack1')
MySecondStack(app, 'stack2')

app.synth()
```

Java

```
App app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.synth();
```

C#

```
var app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.Synth();
```

以下示例是在单独文件上定义堆栈的常用模式。在这里，我们扩展或继承Stack类并定义一个接受scopeid、和的构造函数props。然后，我们使用super接收到的、和scope，id调用基Stack类构造函数props。

TypeScript

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    //...
  }
}
```

JavaScript

```
class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    //...
  }
}
```

Python

```
class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # ...
```

Java

```
public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
```

```
        super(scope, id, props);

        // ...
    }
}
```

C#

```
public class HelloCdkStack : Stack
{
    public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        //...
    }
}
```

Go

```
func HelloCdkStack(scope constructs.Construct, id string, props *HelloCdkStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

  return stack
}
```

以下示例声明了一个名为的堆栈类MyFirstStack，其中包含一个 Amazon S3 存储桶。

TypeScript

```
class MyFirstStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

JavaScript

```
class MyFirstStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

Python

```
class MyFirstStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket")
```

Java

```
public class MyFirstStack extends Stack {
    public MyFirstStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyFirstStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        new Bucket(this, "MyFirstBucket");
    }
}
```

C#

```
public class MyFirstStack : Stack
{
    public MyFirstStack(Stack scope, string id, StackProps props = null) :
 base(scope, id, props)
    {
        new Bucket(this, "MyFirstBucket");
```

```
    }
}
```

Go

```
func MyFirstStack(scope constructs.Construct, id string, props *MyFirstStackProps)
 awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  s3.NewBucket(stack, jsii.String("MyFirstBucket"), &s3.BucketProps{})
  return stack
}
```

但是，此代码仅声明了一个堆栈。要将堆栈实际合成 AWS CloudFormation 模板并进行部署，必须对其进行实例化。而且，像所有 CDK 构造一样，它必须在某些上下文中实例化。App就是那个背景。

如果您使用的是标准 AWS CDK 开发模板，则您的堆栈将在您实例化对象的同一个文件中进行实例化。App

TypeScript

项目bin文件夹中以您的项目（例如hello-cdk.ts）命名的文件。

JavaScript

项目bin文件夹中以您的项目（例如hello-cdk.js）命名的文件。

Python

项目主目录app.py中的文件。

Java

例如 *ProjectName*App.javaHelloCdkApp.java，名为的文件嵌套在src/main目录的深处。

C#

例如src*ProjectName*，名为Program.cs下的文件src\HelloCdk\Program.cs。

堆栈 API

Stack 对象提供了丰富的 API，包括以下内容：

- `Stack.of(construct)`— 一种静态方法，它返回在其中定义构造的堆栈。如果你需要在可重复使用的构造中与堆栈进行交互，这很有用。如果在作用域中找不到堆栈，则调用将失败。
- `stack.stackName`(Python:`stack_name`)-返回堆栈的物理名称。如前所述，所有 AWS CDK 堆栈都有一个物理名称， AWS CDK 可以在合成过程中解析。
- `stack.region`和 `stack.account` — 分别返回此堆栈将部署到的 AWS 区域和账户。这些属性返回以下内容之一：
  - 定义堆栈时明确指定的账户或区域
  - 字符串编码的令牌，可解析为账户和区域的 AWS CloudFormation 伪参数，以表明此堆栈与环境无关

  有关如何确定堆栈环境的信息，请参阅the section called “环境”。
- `stack.addDependency(stack)`(Python: `stack.add_dependency(stack)` — 可用于明确定义两个堆栈之间的依赖顺序。同时部署多个堆栈时，cdk deploy命令会遵守此顺序。
- `stack.tags`— 返回TagManager可用于添加或移除堆栈级别标签的。此标签管理器会标记堆栈中的所有资源，并在通过创建堆栈时对堆栈本身进行标记 AWS CloudFormation。
- `stack.partition`、`stack.urlSuffix` (Python:`url_suffix`)、`stack.stackId` (Python:`notification_arn`) 和 `stack.notificationArn` (Python:) — 返回解析为相应 AWS CloudFormation 伪参数的标记，例如`{ "Ref": "AWS::Partition" }`。`stack_id`这些令牌与特定的堆栈对象相关联，因此 AWS CDK 框架可以识别跨堆栈引用。
- `stack.availabilityZones`(Python:`availability_zones`)-返回部署此堆栈的环境中可用的一组可用区。对于与环境无关的堆栈，这始终会返回一个包含两个可用区的数组。对于特定于环境的堆栈，会 AWS CDK 查询环境并返回您指定的区域中可用的确切可用区集。
- `stack.parseArn(arn)`和 `stack.formatArn(comps)` (Python:`parse_arn`,`format_arn`) — 可用于处理亚马逊资源名称 (ARN)。
- `stack.toJsonString(obj)`(Python:`to_json_string`)-可用于将任意对象格式化为可以嵌入到 AWS CloudFormation 模板中的 JSON 字符串。该对象可以包含标记、属性和引用，这些标记、属性和引用只能在部署期间解析。
- `stack.templateOptions`(Python:`template_options`)-用于为堆栈指定 AWS CloudFormation 模板选项，例如转换、描述和元数据。

使用 堆栈

要列出 CDK 应用程序中的所有堆栈，请使用命令。cdk ls前面的示例将输出以下内容：

```
stack1
stack2
```

堆栈作为 AWS CloudFormation 堆栈的一部分部署到 AWS 环境中。环境涵盖了特定的 AWS 账户 和 AWS 区域.

当您为具有多个堆栈的应用程序运行cdk synth命令时，云程序集会为每个堆栈实例包含一个单独的模板。即使这两个堆栈是同一个类的实例，它们也会将它们作为两个单独 AWS CDK 的模板发出。

您可以通过在cdk synth命令中指定堆栈名称来合成每个模板。以下示例合成了 st ack1 的模板。

```
$ cdk synth stack1
```

这种方法在概念上与通常使用 AWS CloudFormation 模板的方式不同，在模板中，模板可以多次部署并通过参数进行参数化。AWS CloudFormation尽管可以在中定义 AWS CloudFormation 参数 AWS CDK，但通常不建议使用这些参数，因为 AWS CloudFormation 参数只能在部署期间解析。这意味着您无法在代码中确定它们的值。

例如，要根据参数值有条件地将资源包含在应用程序中，您必须设置AWS CloudFormation 条件并使用该条件标记该资源。 AWS CDK 采用的方法是在合成时解析混凝土模板。因此，您可以使用 i f 语句来检查该值，以确定是应定义资源还是应应用某些行为。

> **Note**
>
> 在合成期间 AWS CDK 提供尽可能多的分辨率，以实现编程语言的惯用和自然使用。

像任何其他构造一样，堆栈可以组合成组。以下代码显示了由三个堆栈组成的服务示例：控制平面、数据平面和监控堆栈。服务结构定义了两次：一次用于测试环境，另一次用于生产环境。

TypeScript

```
import { App, Stack } from 'aws-cdk-lib';
import { Construct } from 'constructs';
```

```
interface EnvProps {
  prod: boolean;
}

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope: Construct, id: string, props?: EnvProps) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

JavaScript

```
const { App, Stack } = require('aws-cdk-lib');
const { Construct } = require('constructs');

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope, id, props) {

    super(scope, id);
```

```
    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");
  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

Python

```
from aws_cdk import App, Stack
from constructs import Construct

# imagine these stacks declare a bunch of related resources
class ControlPlane(Stack): pass
class DataPlane(Stack): pass
class Monitoring(Stack): pass

class MyService(Construct):

  def __init__(self, scope: Construct, id: str, *, prod=False):

    super().__init__(scope, id)

    # we might use the prod argument to change how the service is configured
    ControlPlane(self, "cp")
    DataPlane(self, "data")
    Monitoring(self, "mon")

app = App();
MyService(app, "beta")
MyService(app, "prod", prod=True)

app.synth()
```

Java

```
package com.myorg;
```

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.constructs.Construct;

public class MyApp {

    // imagine these stacks declare a bunch of related resources
    static class ControlPlane extends Stack {
        ControlPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class DataPlane extends Stack {
        DataPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class Monitoring extends Stack {
        Monitoring(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class MyService extends Construct {
        MyService(Construct scope, String id) {
            this(scope, id, false);
        }

        MyService(Construct scope, String id, boolean prod) {
            super(scope, id);

            // we might use the prod argument to change how the service is
 configured
            new ControlPlane(this, "cp");
            new DataPlane(this, "data");
            new Monitoring(this, "mon");
        }
    }

    public static void main(final String argv[]) {
        App app = new App();
```

```
        new MyService(app, "beta");
        new MyService(app, "prod", true);

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;

// imagine these stacks declare a bunch of related resources
public class ControlPlane : Stack {
    public ControlPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class DataPlane : Stack {
    public DataPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class Monitoring : Stack
{
    public Monitoring(Construct scope, string id=null) : base(scope, id) { }
}

public class MyService : Construct
{
    public MyService(Construct scope, string id, Boolean prod=false) : base(scope,
 id)
    {
        // we might use the prod argument to change how the service is configured
        new ControlPlane(this, "cp");
        new DataPlane(this, "data");
        new Monitoring(this, "mon");
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
        var app = new App();
        new MyService(app, "beta");
        new MyService(app, "prod", prod: true);
        app.Synth();
    }
}
```

该 AWS CDK 应用程序最终由六个堆栈组成，每个环境三个堆栈：

```
$ cdk ls

betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

AWS CloudFormation 堆栈的物理名称由 AWS CDK 基于堆栈在树中的构造路径自动确定。默认情况下，堆栈的名称源自`Stack`对象的构造 ID。但是，您可以使用 `stackName` prop（在 Python 中`stack_name`）指定显式名称，如下所示。

TypeScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

JavaScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

Python

```
MyStack(self, "not:a:stack:name", stack_name="this-is-stack-name")
```

Java

```
new MyStack(this, "not:a:stack:name", StackProps.builder()
    .StackName("this-is-stack-name").build());
```

C#

```
new MyStack(this, "not:a:stack:name", new StackProps
{
    StackName = "this-is-stack-name"
});
```

嵌套堆栈

该NestedStack构造为规避堆栈的 AWS CloudFormation 500 个资源限制提供了一种方法。嵌套堆栈仅算作包含它的堆栈中的一个资源。但是，它最多可以包含 500 个资源，包括额外的嵌套堆栈。

嵌套堆栈的作用域必须是Stack或NestedStack构造。嵌套堆栈不需要在其父堆栈中按词法声明。实例化嵌套堆栈时，只需要将父堆栈作为第一个参数 (scope) 传递。除了这个限制之外，在嵌套堆栈中定义构造的工作原理与在普通堆栈中定义构造完全相同。

在合成时，嵌套堆栈会合成到自己的 AWS CloudFormation 模板中，并在部署时将其上传到 AWS CDK 暂存桶。嵌套堆栈绑定到其父堆栈，不会被视为独立的部署工件。它们不是由列出的cdk list，也不能由其部署cdk deploy。

父堆栈和嵌套堆栈之间的引用会自动转换为生成的 AWS CloudFormation 模板中的堆栈参数和输出，就像任何跨堆栈引用一样。

> **Warning**
>
> 在部署嵌套堆栈之前，不会显示安全状态的变化。此信息仅针对顶级堆栈显示。

构造

构造是 AWS Cloud Development Kit (AWS CDK) 应用程序的基本组成部分。构造是应用程序中的一个组件，它代表一个或多个 AWS CloudFormation 资源及其配置。您可以通过导入和配置构造来逐步构建应用程序。

构造是您导入到 CDK 应用程序中的类。构造可从 AWS 构造库中获得。您也可以创建和分发自己的构造，或者使用第三方开发者创建的构造。

构造是构造编程模型 (CPM) 的一部分。它们可以与其他工具一起使用，例如 CDK for Terraform (cdkTF)、CDK for Kubernetes (cdk8s) 和。Projen

主题

AWS 构造图书馆

AWS 构造库包含由 AWS开发和维护的构造集合。它被组织成各种模块，这些模块包含代表上 AWS所有可用资源的结构。有关参考信息，请参阅 AWS CDK API 参考。

主要 CDK 包被称为`aws-cdk-lib`，它包含 AWS 构造库的大部分内容。它还包含诸如`Stack`和之类的基类`App`。

主 CDK 软件包的实际软件包名称因语言而异。

TypeScript

安装	`npm install aws-cdk-lib`
Import	`import * as cdk from 'aws-cdk-lib';`

JavaScript

安装	`npm install aws-cdk-lib`
Import	`const cdk = require('aws-cdk-lib');`

Python

安装	`python -m pip install aws-cdk-lib`

Import	`import aws_cdk as cdk`

Java

在`pom.xml`，添加	Group `##.amazon.awscdk` ；artifact `aws-cdk-lib`
Import	`import software.amazon.awscdk.App;`

C#

安装	dotnet add package Amazon.CDK.Lib
Import	`using Amazon.CDK;`

Go

安装	go get github.com/aws/aws-cdk-go/awscdk/v2
Import	`import ( "github.com/aws/aws-cdk-go/awscdk/v2" )`

> **Note**
>
> 如果您使用创建了 CDK 项目`cdk init`，则无需手动安装`aws-cdk-lib`。

AWS 构造库还包含带有`Construct`基类的`constructs`包。它包含在自己的软件包中，因为除了，其他基于构造的工具也使用它，包括适用于 Terraform 的 CDK 和 Kubernetes 的 CDK。 AWS CDK

许多第三方也发布了与... 兼容的构造。 AWS CDK访问 C onstruct Hub， AWS CDK 探索建筑合作伙伴生态系统。

构造关卡

构造库中的 AWS 构造分为三个级别。每个级别都提供了越来越高的抽象级别。抽象越高，配置越容易，所需的专业知识也越少。抽象越低，可用的自定义越多，需要更多的专业知识。

1 级 (L1) 构造

L1 构造，也称为 CFN 资源，是最低级别的构造，不提供抽象。每个 L1 构造都直接映射到单个 AWS CloudFormation 资源。使用 L1 构造，您可以导入代表特定 AWS CloudFormation 资源的构造。然后，您可以在构造实例中定义资源的属性。

当您熟悉 AWS CloudFormation 并需要完全控制 AWS 资源属性的定义时，L1 结构非常适合使用。

在 AWS 构造库中，L1 构造以开头命名`Cfn`，后面是它所代表的 AWS CloudFormation 资源的标识符。例如，该`CfnBucket`构造是代表`AWS::S3::Bucket` AWS CloudFormation 资源的 L1 构造。

L1 结构是根据AWS CloudFormation 资源规范生成的。如果资源存在于中 AWS CloudFormation，则该资源将 AWS CDK 作为 L1 结构在中可用。新资源或属性可能需要长达一周的时间才能在 Construct Librar AWS y 中发布。有关更多信息，请参阅《AWS CloudFormation 用户指南》中的AWS 资源和属性类型参考。

2 级 (L2) 构造

L2 构造，也称为精选构造，由 CDK 团队精心开发，通常是使用最广泛的构造类型。L2 构造直接映射到单个 AWS CloudFormation 资源，类似于 L1 构造。与 L1 结构相比，L2 构造通过直观的基于意图的 API 提供了更高级别的抽象。L2 结构包括合理的默认属性配置、最佳实践安全策略，并为您生成大量样板代码和粘合逻辑。

L2 构造还为大多数资源提供了辅助方法，使定义属性、权限、资源之间基于事件的交互等变得更加简单快捷。

该`s3.Bucket`类是亚马逊简单存储服务 (Amazon S3) 存储桶资源的 L2 结构示例。

AWS 构造库包含指定为稳定且可供生产使用的 L2 构造。对于正在开发的 L2 结构，它们被指定为实验性结构，并在单独的模块中提供。

3 级 (L3) 构造

L3 构造，也称为模式，是最高级别的抽象。每个 L3 结构可以包含一组资源，这些资源配置为协同工作以完成应用程序中的特定任务或服务。L3 结构用于为应用程序中的特定用例创建整个 AWS 架构。

为了提供完整的系统设计或大型系统的重要组成部分，L3 构造提供了自以为是的默认属性配置。它们是围绕解决问题和提供解决方案的特定方法而构建的。使用 L3 结构，您可以用最少的输入和代码快速创建和配置多个资源。

该`ecsPatterns.ApplicationLoadBalancedFargateService`类是 L3 结构的示例，该结构表示在亚马逊弹性容器 AWS Fargate 服务 (Amazon ECS) Container Service 集群上运行并以应用程序负载均衡器为前置的服务。

与 L2 构造类似，可供生产使用的 L3 构造包含在构造库中。 AWS 正在开发的软件以单独的模块提供。

定义结构

合成

组合是通过构造定义更高级别抽象的关键模式。高级构造可以由任意数量的较低级别构造构成。从自下而上的角度来看，您可以使用构造来组织要 AWS 部署的各个资源。你可以随心所欲地使用任何适合你目的的抽象，你可以根据需要使用任意数量的级别。

通过组合，您可以定义可重复使用的组件，并像任何其他代码一样共享它们。例如，团队可以定义一种结构，用于实现公司对 Amazon DynamoDB 表的最佳实践，包括备份、全局复制、自动扩展和监控。团队可以在内部与其他团队共享构建，也可以公开共享。

团队可以像使用任何其他库包一样使用构造。库更新后，开发人员可以访问新版本的改进和错误修复，这与任何其他代码库类似。

初始化

构造是在扩展 `Construct` 基类的类中实现的。您可以通过实例化类来定义构造。所有构造在初始化时都有三个参数：

- s@@ cop e — 构造的父级或所有者。这可以是堆栈，也可以是其他构造。作用域决定构造在构造树中的位置。通常，您应该为作用域传入 `this` (`self`in Python)，它代表当前对象。

- id — 在作用域内必须是唯一的标识符。该标识符充当构造中定义的所有内容的命名空间。它用于生成唯一标识符，例如资源名称和 AWS CloudFormation 逻辑 ID。

  标识符只需要在一个范围内是唯一的。这使您可以实例化和重用构造，而不必担心它们可能包含的构造和标识符，并且可以将构造组合成更高级别的抽象。此外，作用域可以同时引用一组构造。示例包括标记或指定构造的部署位置。
- props — 一组属性或关键字参数，视语言而定，用于定义构造的初始配置。更高级别的构造提供了更多的默认值，如果所有 prop 元素都是可选的，则可以完全省略 props 参数。

配置

大多数构造都接受定义构造配置的名称/值集合`props`作为其第三个参数（或者在 Python 中为关键字参数）。以下示例定义了一个启用了 AWS Key Management Service (AWS KMS) 加密和静态网站托管的存储桶。由于它没有明确指定加密密钥，因此该`Bucket`构造定义了一个新的`kms.Key`密钥并将其与存储桶相关联。

TypeScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

JavaScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

Python

```
s3.Bucket(self, "MyEncryptedBucket", encryption=s3.BucketEncryption.KMS,
    website_index_document="index.html")
```

Java

```
Bucket.Builder.create(this, "MyEncryptedBucket")
        .encryption(BucketEncryption.KMS_MANAGED)
```

```
        .websiteIndexDocument("index.html").build();
```

C#

```
new Bucket(this, "MyEncryptedBucket", new BucketProps
{
    Encryption = BucketEncryption.KMS_MANAGED,
    WebsiteIndexDocument = "index.html"
});
```

Go

```
awss3.NewBucket(stack, jsii.String("MyEncryptedBucket"), &awss3.BucketProps{
 Encryption: awss3.BucketEncryption_KMS,
 WebsiteIndexDocument: jsii.String("index.html"),
})
```

与构造交互

构造是扩展基本构造类的类。实例化构造后，构造对象会公开一组方法和属性，这些方法和属性允许你与该构造进行交互并将其作为对系统其他部分的引用传递。

该 AWS CDK 框架对构造的 API 没有任何限制。作者可以定义他们想要的任何 API。但是， AWS 构造库中包含的 AWS 构造（例如`s3.Bucket`）遵循准则和常见模式。这为所有 AWS 资源提供了一致的体验。

大多数 AWS 构造都有一组授权方法，您可以使用这些方法向委托人授予对该构造的 AWS Identity and Access Management (IAM) 权限。以下示例授予 IAM 组从 Amazon S3 存储桶中读取数据的`data-science`权限`raw-data`。

TypeScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

JavaScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
```

```
rawData.grantRead(dataScience);
```

Python

```
raw_data = s3.Bucket(self, 'raw-data')
data_science = iam.Group(self, 'data-science')
raw_data.grant_read(data_science)
```

Java

```
Bucket rawData = new Bucket(this, "raw-data");
Group dataScience = new Group(this, "data-science");
rawData.grantRead(dataScience);
```

C#

```
var rawData = new Bucket(this, "raw-data");
var dataScience = new Group(this, "data-science");
rawData.GrantRead(dataScience);
```

Go

```
 rawData := awss3.NewBucket(stack, jsii.String("raw-data"), nil)
 dataScience := awsiam.NewGroup(stack, jsii.String("data-science"), nil)
 rawData.GrantRead(dataScience, nil)
```

另一种常见的模式是 AWS 构造根据其他地方提供的数据设置资源的一个属性。属性可以包括亚马逊资源名称 (ARN)、名称或网址。

以下代码定义了一个 AWS Lambda 函数，并通过环境变量中队列的 URL 将其与亚马逊简单队列服务 (Amazon SQS) Simple Queue SQUEE Service 队列相关联。

TypeScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
```

```
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

JavaScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

Python

```
jobs_queue = sqs.Queue(self, "jobs")
create_job_lambda = lambda_.Function(self, "create-job",
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    code=lambda_.Code.from_asset("./create-job-lambda-code"),
    environment=dict(
        QUEUE_URL=jobs_queue.queue_url
    )
)
```

Java

```
final Queue jobsQueue = new Queue(this, "jobs");
Function createJobLambda = Function.Builder.create(this, "create-job")
                .handler("index.handler")
                .code(Code.fromAsset("./create-job-lambda-code"))
                .environment(java.util.Map.of(   // Map.of is Java 9 or later
                    "QUEUE_URL", jobsQueue.getQueueUrl())
                .build();
```

C#

```
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
```

```
{
    Runtime = Runtime.NODEJS_18_X,
    Handler = "index.handler",
    Code = Code.FromAsset(@".\create-job-lambda-code"),
    Environment = new Dictionary<string, string>
    {
        ["QUEUE_URL"] = jobsQueue.QueueUrl
    }
});
```

Go

```
createJobLambda := awslambda.NewFunction(stack, jsii.String("create-job"),
&awslambda.FunctionProps{
 Runtime: awslambda.Runtime_NODEJS_18_X(),
 Handler: jsii.String("index.handler"),
 Code:    awslambda.Code_FromAsset(jsii.String(".\\create-job-lambda-code"), nil),
 Environment: &map[string]*string{
  "QUEUE_URL": jsii.String(*jobsQueue.QueueUrl()),
 },
})
```

有关 AWS 构造库中最常见 API 模式的信息，请参阅the section called “资源”。

应用程序和堆栈结构

AWS 构造库中的App和Stack类是唯一的构造。与其他结构相比，它们不会自行配置 AWS 资源。相反，它们用于为您的其他构造提供上下文。所有代表 AWS 资源的构造都必须在Stack构造的范围内直接或间接地定义。 Stack构造是在App构造的范围内定义的。

要了解有关 CDK 应用程序的更多信息，请参阅AWS CDK 应用程序。要了解有关 CDK 堆栈的更多信息，请参阅。堆栈

以下示例定义了一个具有单个堆栈的应用程序。在堆栈中，使用 L2 结构来配置 Amazon S3 存储桶资源。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';
```

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

Python

```
from aws_cdk import App, Stack
import aws_cdk.aws_s3 as s3
from constructs import Construct

class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket", versioned=True)
```

```
app = App()
HelloCdkStack(app, "HelloCdkStack")
```

Java

HelloCdkStack.java文件中定义的堆栈：

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

在HelloCdkApp.java文件中定义的应用程序：

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.StackProps;

public class HelloCdkApp {
    public static void main(final String[] args) {
        App app = new App();

        new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdkApp
{
    internal static class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new HelloCdkStack(app, "HelloCdkStack");
            app.Synth();
        }
    }

    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps { Versioned = true });
        }
    }
}
```

Go

```
func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}
```

使用构造

使用 L1 构造

L1 构造直接映射到各个 AWS CloudFormation 资源。您必须提供资源所需的配置。

在这个例子中，我们使用 `CfnBucket` L1 构造创建一个`bucket`对象：

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

Python

```
bucket = s3.CfnBucket(self, "MyBucket", bucket_name="MyBucket")
```

Java

```
CfnBucket bucket = new CfnBucket.Builder().bucketName("MyBucket").build();
```

C#

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName= "MyBucket"
});
```

Go

```
 awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
  BucketName: jsii.String("MyBucket"),
```

```
 })
```

构造不是简单布尔值、字符串、数字或容器的属性在支持的语言中的处理方式有所不同。

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

Python

在 Python 中，这些属性由定义为 L1 构造内部类的类型表示。例如，a 的可选属性`CfnBucket`需要一个`cors_configuration`类型的`CfnBucket.CorsConfigurationProperty`封装器。在这里，我们在一个`CfnBucket`实例`cors_configuration`上进行定义。

```
bucket = CfnBucket(self, "MyBucket", bucket_name="MyBucket",
    cors_configuration=CfnBucket.CorsConfigurationProperty(
        cors_rules=[CfnBucket.CorsRuleProperty(
            allowed_origins=["*"],
            allowed_methods=["GET"]
        )]
    )
```

```
)
```

Java

在 Java 中，这些属性由定义为 L1 构造内部类的类型表示。例如，a 的可选属性CfnBucket需要一个corsConfiguration类型的CfnBucket.CorsConfigurationProperty封装器。在这里，我们在一个CfnBucket实例corsConfiguration上进行定义。

```
CfnBucket bucket = CfnBucket.Builder.create(this, "MyBucket")
                        .bucketName("MyBucket")
                        .corsConfiguration(new
 CfnBucket.CorsConfigurationProperty.Builder()
                            .corsRules(Arrays.asList(new
 CfnBucket.CorsRuleProperty.Builder()
                                .allowedOrigins(Arrays.asList("*"))
                                .allowedMethods(Arrays.asList("GET"))
                                .build()))
                            .build())
                        .build();
```

C#

在 C# 中，这些属性由定义为 L1 构造内部类的类型表示。例如，a 的可选属性CfnBucket需要一个CorsConfiguration类型的CfnBucket.CorsConfigurationProperty封装器。在这里，我们在一个CfnBucket实例CorsConfiguration上进行定义。

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName = "MyBucket",
    CorsConfiguration = new CfnBucket.CorsConfigurationProperty
    {
        CorsRules = new object[] {
            new CfnBucket.CorsRuleProperty
            {
                AllowedOrigins = new string[] { "*" },
                AllowedMethods = new string[] { "GET" },
            }
        }
    }
});
```

Go

在 Go 中，这些类型是使用 L1 构造的名称、下划线和属性名称命名的。
例如，a 的可选属性`CfnBucket`需要一个`CorsConfiguration`类型
的`CfnBucket_CorsConfigurationProperty`封装器。在这里，我们在一个`CfnBucket`实例`CorsConfiguration`上进行定义。

```
awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
 BucketName: jsii.String("MyBucket"),
 CorsConfiguration: &awss3.CfnBucket_CorsConfigurationProperty{
  CorsRules: []awss3.CorsRule{
   awss3.CorsRule{
    AllowedOrigins: jsii.Strings("*"),
    AllowedMethods: &[]awss3.HttpMethods{"GET"},
   },
  },
 },
})
```

Important

您不能将 L2 属性类型与 L1 构造一起使用，反之亦然。使用 L1 构造时，请务必使用为正在使用的 L1 构造定义的类型。不要使用其他 L1 构造中的类型（有些可能具有相同的名称，但类型不同）。
目前，我们的一些特定于语言的 API 引用在 L1 属性类型的路径中存在错误，或者根本没有记录这些类。我们希望尽快解决这个问题。同时，请记住，此类类型始终是与之一起使用的 L1 构造的内部类。

使用 L2 构造

在以下示例中，我们通过从 `Bucket`L2 构造中创建对象来定义 Amazon S3 存储桶：

TypeScript

```
import * as s3 from 'aws-cdk-lib/aws-s3';

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
```

```
});
```

JavaScript

```
const s3 = require('aws-cdk-lib/aws-s3');

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

Python

```
import aws_cdk.aws_s3 as s3

# "self" is HelloCdkStack
s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

```
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
                .versioned(true).build();
    }
}
```

C#

```
using Amazon.CDK.AWS.S3;

// "this" is HelloCdkStack
new Bucket(this, "MyFirstBucket", new BucketProps
```

```
{
    Versioned = true
});
```

Go

```
import (
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/jsii-runtime-go"
)

// stack is HelloCdkStack
awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })>
```

MyFirstBucket不是 AWS CloudFormation 创建的存储桶的名称。它是在 CDK 应用程序的上下文中赋予新构造的逻辑标识符。PhysicalNam e 值将用于命名资源。 AWS CloudFormation

使用第三方构造

C@@ onstruct Hub 是一种资源 AWS，可帮助您发现来自第三方和开源 CDK 社区的其他构造。

编写你自己的构造

除了使用现有构造之外，你还可以编写自己的构造，让任何人在自己的应用程序中使用它们。中的所有构造都是相等的。 AWS CDK构造库中的 AWS 构造与通过NPM、Maven或发布的第三方库中的构造相同。PyPI发布到贵公司内部软件包存储库的构造也将以同样的方式处理。

要声明新构造，请在constructs包中创建一个扩展 C on struct 基类的类，然后遵循初始化器参数的模式。

以下示例说明如何声明代表 Amazon S3 存储桶的结构。每当有人向其中上传文件时，S3 存储桶都会发送亚马逊简单通知服务 (Amazon SNS) Simple Notification Service 通知。

TypeScript

```
export interface NotifyingBucketProps {
  prefix?: string;
}
```

```
export class NotifyingBucket extends Construct {
  constructor(scope: Construct, id: string, props: NotifyingBucketProps = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {
  constructor(scope, id, props = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket }
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }
```

```
    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        Topic topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucketProps : BucketProps
{
    public string Prefix { get; set; }
}

public class NotifyingBucket : Construct
{
    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        var topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

```
type NotifyingBucketProps struct {
 awss3.BucketProps
 Prefix *string
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) awss3.Bucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 return bucket
}
```

> **Note**
>
> 我们的NotifyingBucket构造不是继承自Bucket而是继承自Construct。我们使用组合而不是继承来将 Amazon S3 存储桶和 Amazon SNS 主题捆绑在一起。通常，在开发 AWS CDK 构造时，组合比继承更受青睐。

NotifyingBucket构造函数具有典型的构造签名：scopeid、和props。最后一个参数是可选的（获取默认值{}），因为所有道具都是可选的。props（基Construct类不带props参数。）例如，你可以在你的应用程序中定义这个构造的实例props，而不必这样做：

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), nil)
```

或者你可以使用`props`（在 Java 中，一个附加参数）来指定要筛选的路径前缀，例如：

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket", prefix="images/")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket", "/images");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), &NotifyingBucketProps{
 Prefix: jsii.String("images/"),
})
```

通常，您还需要在构造中公开一些属性或方法。在你的构造后面隐藏一个话题并不是很有用，因为你的构造的用户无法订阅它。添加topic属性允许使用者访问内部主题，如以下示例所示：

TypeScript

```
export class NotifyingBucket extends Construct {
  public readonly topic: sns.Topic;

  constructor(scope: Construct, id: string, props: NotifyingBucketProps) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {

  constructor(scope, id, props) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
```

```
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket };
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None, **kwargs):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        self.topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(self.topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public Topic topic = null;

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);
```

```
        Bucket bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucket : Construct
{
    public readonly Topic topic;

    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

要在 Go 中做到这一点，我们需要一点额外的管道。我们的原始NewNotifyingBucket函数返回了一个awss3.Bucket. 我们需要通过创建NotifyingBucket结构Bucket来扩展以包含topic成员。然后，我们的函数将返回此类型。

```
type NotifyingBucket struct {
 awss3.Bucket
 topic awssns.Topic
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) NotifyingBucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
```

```
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 var nbucket NotifyingBucket
 nbucket.Bucket = bucket
 nbucket.topic = topic
 return nbucket
}
```

现在，消费者可以订阅该主题，例如：

TypeScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

JavaScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

Python

```
queue = sqs.Queue(self, "NewImagesQueue")
images = NotifyingBucket(self, prefix="Images")
images.topic.add_subscription(sns_sub.SqsSubscription(queue))
```

Java

```
NotifyingBucket images = new NotifyingBucket(this, "MyNotifyingBucket", "/images");
```

```
images.topic.addSubscription(new SqsSubscription(queue));
```

C#

```
var queue = new Queue(this, "NewImagesQueue");
var images = new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
images.topic.AddSubscription(new SqsSubscription(queue));
```

Go

```
queue := awssqs.NewQueue(stack, jsii.String("NewImagesQueue"), nil)
images := NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"),
&NotifyingBucketProps{
 Prefix: jsii.String("/images"),
})
images.topic.AddSubscription(awssnssubscriptions.NewSqsSubscription(queue, nil))
```

了解更多信息

以下视频全面概述了 CDK 结构，并说明了如何在 CDK 应用程序中使用它们。

CDK 结构详解

环境

环境是目标 AWS 账户 ，堆栈部署到 AWS 区域 该环境中。CDK 应用程序中的所有堆栈都与环境显式或隐式关联 ()。env

主题

- 配置环境
- 引导环境

配置环境

对于生产堆栈，我们建议您使用env属性为应用程序中的每个堆栈明确指定环境。以下示例为其两个不同的堆栈指定了不同的环境。

TypeScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

JavaScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

Python

```
env_EU = cdk.Environment(account="8373873873", region="eu-west-1")
env_USA = cdk.Environment(account="2383838383", region="us-west-2")

MyFirstStack(app, "first-stack-us", env=env_USA)
MyFirstStack(app, "first-stack-eu", env=env_EU)
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv("8373873873", "eu-west-1");
        Environment envUSA = makeEnv("2383838383", "us-west-2");
```

```
        new MyFirstStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyFirstStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment
    {
        Account = account,
        Region = region
    };
}

var envEU = makeEnv(account: "8373873873", region: "eu-west-1");
var envUSA = makeEnv(account: "2383838383", region: "us-west-2");

new MyFirstStack(app, "first-stack-us", new StackProps { Env=envUSA });
new MyFirstStack(app, "first-stack-eu", new StackProps { Env=envEU });
```

当您对目标账户和区域进行硬编码时，如前面的示例所示，堆栈将始终部署到该特定账户和区域。为了使堆栈可以部署到不同的目标，但要在合成时确定目标，您的堆栈可以使用 AWS CDK CLI 提供的两个环境变量：CDK_DEFAULT_ACCOUNT和CDK_DEFAULT_REGION。这些变量是根据使用--profile选项指定的 AWS 配置文件设置的，或者如果您未指定默认 AWS 配置文件，则根据默认配置文件进行设置。

以下代码片段显示了如何访问堆栈中从 AWS CDK CLI 传递的账户和区域。

TypeScript

通过 Node 的process对象访问环境变量。

> **Note**
>
> 您需要在process中使用该DefinitelyTyped模块 TypeScript。 cdk init为你安装这个模块。但是，如果您使用的是添加之前创建的项目，或者您没有使用来设置项目，则应手动安装此模块cdk init。
>
> ```
> npm install @types/node
> ```

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

通过 Node 的process对象访问环境变量。

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

Python

使用os模块的environ字典来访问环境变量。

```
import os
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ["CDK_DEFAULT_ACCOUNT"],
    region=os.environ["CDK_DEFAULT_REGION"]))
```

Java

System.getenv()用于获取环境变量的值。

```
public class MyApp {
```

```
    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

System.Environment.GetEnvironmentVariable()用于获取环境变量的值。

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

AWS 区域 使用区域代码指定。有关列表，请参阅区域终端节点。

完全不指定env属性与使用CDK_DEFAULT_ACCOUNT和CDK_DEFAULT_REGION指定属性的 AWS CDK 区别。前者意味着堆栈应合成一个与环境无关的模板。在这样的堆栈中定义的构造不能使用有关其环境的任何信息。例如，你不能编写像 vpc.fromLookup (Pythonfrom_lookup:) 这样的代码if (stack.region === 'us-east-1')或使用需要查询你的账户的框架工具。 AWS 在您指定明确的环境之前，这些功能根本不起作用；要使用它们，必须指定env。

当您使用CDK_DEFAULT_ACCOUNT和进入您的环境时CDK_DEFAULT_REGION，堆栈将部署在综合时由 AWS CDK CLI 确定的账户和区域中。这使依赖于环境的代码可以正常工作，但这也意味着合成后的模板可能会因其合成所在的机器、用户或会话而有所不同。在开发过程中，这种行为通常是可以接受的，甚至是可取的，但对于生产用途来说，它可能是一种反模式。

您可以使用任何有效的表达式随心所欲地进行设置env。例如，您可以编写堆栈以支持另外两个环境变量，以便在合成时覆盖账户和区域。我们CDK_DEPLOY_REGION在这里给它们CDK_DEPLOY_ACCOUNT起个名字，但你可以随心所欲地给它们起名字，因为它们不是由... 设置的 AWS CDK。在以下堆栈的环境中，如果设置了替代环境变量，则使用它们。如果未设置，则它们会回退到提供的默认环境 AWS CDK。

TypeScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

Python

```
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ.get("CDK_DEPLOY_ACCOUNT", os.environ["CDK_DEFAULT_ACCOUNT"]),
    region=os.environ.get("CDK_DEPLOY_REGION", os.environ["CDK_DEFAULT_REGION"])
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEPLOY_ACCOUNT") : account;
        region = (region == null) ? System.getenv("CDK_DEPLOY_REGION") : region;
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_ACCOUNT") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
```

```
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_REGION") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

以这种方式声明堆栈的环境后，你可以编写一个简短的脚本或批处理文件，如下所示，从命令行参数中设置变量，然后调用`cdk deploy`。除前两个参数之外的任何参数都将传递给`cdk deploy`并可用于指定命令行选项或堆栈。

macOS/Linux

```
#!/usr/bin/env bash
if [[ $# -ge 2 ]]; then
    export CDK_DEPLOY_ACCOUNT=$1
    export CDK_DEPLOY_REGION=$2
    shift; shift
    npx cdk deploy "$@"
    exit $?
else
    echo 1>&2 "Provide account and region as first two args."
    echo 1>&2 "Additional args are passed through to cdk deploy."
    exit 1
fi
```

将脚本另存为`cdk-deploy-to.sh`，然后执行`chmod +x cdk-deploy-to.sh`使其可执行。

Windows

```
@findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass
 %~dpn0.ps1 %*
@exit /B %ERRORLEVEL%
if ($args.length -ge 2) {
    $env:CDK_DEPLOY_ACCOUNT, $args = $args
    $env:CDK_DEPLOY_REGION,  $args = $args
    npx cdk deploy $args
    exit $lastExitCode
} else {
    [console]::error.writeline("Provide account and region as first two args.")
    [console]::error.writeline("Additional args are passed through to cdk deploy.")
    exit 1
```

```
}
```

该脚本的 Windows 版本用于提供 PowerShell 与 macOS/Linux 版本相同的功能。它还包含允许将其作为批处理文件运行的指令，以便可以轻松地从命令行调用。应将其另存为`cdk-deploy-to.bat`。该文件`cdk-deploy-to.ps1`将在调用批处理文件时创建。

然后，您可以编写其他调用 “deploy-to” 脚本的脚本来部署到特定环境（甚至每个脚本有多个环境）：

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-test.sh
./cdk-deploy-to.sh 123457689 us-east-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-test.bat
cdk-deploy-to 135792469 us-east-1 %*
```

部署到多个环境时，请考虑是否要在部署失败后继续部署到其他环境。如果第一个生产环境不成功，以下示例将避免部署到第二个生产环境。

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-prod.sh
./cdk-deploy-to.sh 135792468 us-west-1 "$@" || exit
./cdk-deploy-to.sh 246813579 eu-west-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-prod.bat
cdk-deploy-to 135792469 us-west-1 %* || exit /B
cdk-deploy-to 245813579 eu-west-1 %*
```

开发人员仍然可以使用普通`cdk deploy`命令部署到自己的 AWS 环境中进行开发。

如果您在实例化堆栈时没有指定环境，则表示该堆栈与环境无关。 AWS CloudFormation 从此类堆栈中合成的模板将尝试对与环境相关的属性（例如`stack.account`、`stack.region`和(`stack.availabilityZones`Python:)）使用部署时解析。`availability_zones`

使用部署cdk deploy与环境无关的堆栈时， AWS CDK CLI将使用指定的 AWS CLI 配置文件来确定部署位置。如果未指定配置文件，则使用默认配置文件。 AWS CDK CLI遵循类似于的协议， AWS CLI 用于确定在您的 AWS 账户中执行操作时要使用哪些 AWS 证书。有关详细信息，请参阅 the section called “AWS CDK 工具包”。

在与环境无关的堆栈中，任何使用可用区的结构都将看到两个可用区，从而允许将堆栈部署到任何区域。

引导环境

您必须引导要部署 CDK 堆栈的每个环境。引导可以为部署环境做好准备。要了解更多信息，请参阅正在引导。

正在引导

引导是为部署准备环境的过程。Bootstrapping 是一次性操作，您必须对部署资源的每个环境执行此操作。

主题

- 引导环境
- 如何引导
- 自定义引导
- 引导模板的差异
- 堆栈合成器
- 自定义合成
- 引导模板合约
- Security Hub 的调查结果

引导环境

> **Important**
>
> 存储在引导资源中的数据可能会产生 AWS 费用。

Bootstrapping 会在您的环境中配置资源，例如用于存储文件的亚马逊简单存储服务 (Amazon S3) 存储桶和授予执行部署所需权限的 AWS Identity and Access Management (IAM) 角色。这些资源在称为引导 AWS CloudFormation 堆栈的堆栈中进行配置。它通常被命名`CDKToolkit`。与任何 AWS CloudFormation 堆栈一样，它将在部署后出现在您环境的 AWS CloudFormation 控制台中。

> **Note**
>
> CDK v2 使用现代引导模板。v2 不支持 CDK v1 中的旧版模板。

环境是独立的。如果要部署到多个环境，则必须单独启动每个环境。

如果您尝试将 CDK 应用程序部署到尚未启动的环境中，则会收到一条错误消息，提醒您引导该环境。

使用 CDK Pipelines 进行引导

如果您使用CDK Pipelines部署到其他账户的环境中，并且会收到如下消息：

```
Policy contains a statement with one or more invalid principals
```

此错误消息表示其他环境中不存在相应的 IAM 角色。最有可能的原因是环境没有被引导。引导环境并重试。

> **Note**
>
> 如果环境已引导，请勿删除和重新创建环境的引导堆栈。删除引导堆栈将删除最初在环境中为支持 CDK 部署而配置的 AWS 资源。这将导致管道停止工作。相反，请尝试通过再次运行 CDK CLI `cdk bootstrap` 命令将引导堆栈更新到新版本。

如何引导

当您引导环境时，会将 AWS CloudFormation 模板部署到特定环境中。此模板在您的账户中预置资源，以便为您的环境做好部署准备。

引导模板接受自定义引导资源某些方面的参数。有关更多信息，请参阅 the section called “自定义引导”。

您可以通过以下任何一种方式进行引导：

- 使用 AWS CDK CLI 命令。cdk bootstrap这是最简单的方法，如果您只有几个环境需要引导，则效果很好。
- AWS CDK CLI使用其他部署工具 AWS CloudFormation 部署提供的模板。这允许您使用 AWS CloudFormation StackSets 或 AWS Control Tower 以及 AWS CloudFormation 控制台或 AWS CLI。在部署之前，您可以对模板进行少量修改。这种方法更加灵活，适用于大规模部署。

多次引导环境不是错误。如果您的引导环境已经被引导，则将在必要时升级其引导程序堆栈。否则，什么也不会发生。

使用 bootstraph AWS CDKCLI

使用`cdk bootstrap`命令引导一个或多个 AWS 环境。

以下示例引导了两个环境：

```
$ cdk bootstrap aws://ACCOUNT-NUMBER-1/REGION-1 aws://ACCOUNT-NUMBER-2/REGION-2 ...
```

以下示例显示了引导环境的多种方式。如第二个示例所示，在指定环境时，`aws://`前缀是可选的。

```
$ cdk bootstrap aws://123456789012/us-east-1
$ cdk bootstrap 123456789012/us-east-1 123456789012/us-west-1
```

当你运行时cdk bootstrap，CDK CLI 总是合成当前目录中的 CDK 应用程序。如果您未指定至少一个环境，CDK CLI 将引导应用程序中引用的所有环境。

对于与环境无关的堆栈，CDK CLI 将尝试根据默认来源确定环境。这可能是使用--profile选项、环境变量或默认 AWS CLI 源指定的环境。如果找到，则会引导环境。

例如，以下命令使用`prod` AWS 配置文件合成当前 AWS CDK 应用程序，然后引导其环境。

```
$ cdk bootstrap --profile prod
```

从模板引导 AWS CloudFormation

您可以通过获取和部署引导模板来引导 AWS CloudFormation 环境。

要在文件中获取此模板的副本`bootstrap-template.yaml`，请运行以下命令：

macOS/Linux

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

Windows

在 Windows 上， PowerShell 必须使用它来保留模板的编码。

```
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-
template.yaml"
```

该模板也可在AWS CDK GitHub 存储库中找到。

使用 CDK CLI 或您首选的模板部署机制部署此 AWS CloudFormation 模板。要使用 CDK CLI 进行部署，请运行cdk bootstrap --template ***TEMPLATE_FILENAME***。您也可以 AWS CLI 通过运行以下命令将其部署，或者使用 AWS CloudFormation 堆栈集一次性部署到一个或多个账户。

macOS/Linux

```
aws cloudformation create-stack \
  --stack-name CDKToolkit \
  --template-body file://path/to/bootstrap-template.yaml \
  --capabilities CAPABILITY_NAMED_IAM \
  --region us-west-1
```

Windows

```
aws cloudformation create-stack ^
  --stack-name CDKToolkit ^
  --template-body file://path/to/bootstrap-template.yaml ^
  --capabilities CAPABILITY_NAMED_IAM ^
```

```
  --region us-west-1
```

自定义引导

有两种方法可以自定义环境中资源的引导：

- 在命令中使用`cdk bootstrap`命令行参数。这使您可以修改模板的某些方面。
- 修改默认的引导模板并自行部署。这使您可以更全面地控制引导程序资源。

以下命令行选项与 CDK 一起使用时 CLIcdk bootstrap，可以对引导模板进行常用的调整：

- --bootstrap-bucket-name 覆盖 Amazon S3 存储桶的名称。可能需要更改您的 CDK 应用程序（请参阅the section called “堆栈合成器”）。
- --bootstrap-kms-key-id覆盖用于加密 S3 存储桶的密 AWS KMS 钥。
- --cloudformation-execution-policies指定应附加到堆栈部署 AWS CloudFormation 期间所扮演的部署角色的托管策略的 ARN。默认情况下，使用该`AdministratorAccess`策略部署堆栈时具有完全的管理员权限。

  策略 ARN 必须作为单个字符串参数传递，各个 ARN 用逗号分隔。例如：

  ```
  --cloudformation-execution-policies "arn:aws:iam::aws:policy/
  AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess".
  ```

  > **Important**
  >
  > 为避免部署失败，请确保您指定的策略足以满足您将在引导环境中执行的任何部署。

- --qualifier是添加到引导堆栈中所有资源的名称中的字符串。当您在同一环境中配置多个引导堆栈时，使用限定符可以避免资源名称冲突。默认值为`hnb659fds`（此值没有意义）。

  更改限定符还要求您的 CDK 应用程序将更改后的值传递给堆栈合成器。有关更多信息，请参阅 the section called “堆栈合成器”。
- --tags向引导堆栈添加一个或多个 AWS CloudFormation 标签。
- --trust列出了可能部署到正在引导的环境中的 AWS 帐户。

  在引导其他环境中的 CDK 流水线将部署到的环境时，请使用此标志。进行引导的帐户始终是可信的。

- --trust-for-lookup列出了可能从正在引导的环境中查找上下文信息的 AWS 帐户。

  使用此标志授予账户合成将部署到环境中的堆栈的权限，而无需实际授予他们直接部署这些堆栈的权限。
- --termination-protection防止删除引导堆栈。有关更多信息，请参阅《AWS CloudFormation 用户指南》中的保护堆栈不被删除。

> **Important**
>
> 现代引导模板可以有效地向`--trust`列表中的任何 AWS 账户授`--cloudformation-execution-policies`予隐含的权限。默认情况下，这会扩展对引导账户中任何资源的读写权限。请务必使用您熟悉的策略和可信帐户来配置引导堆栈。

自定义模板

当您需要的自定义功能超出CDK所CLI能提供的范围时，可以修改引导模板以满足您的需求。首先，使用--show-template选项获取模板。以下是 示例：

```
$ cdk bootstrap --show-template
```

您所做的任何修改都必须遵守引导模板合同。要确保您的自定义设置不会在以后被cdk bootstrap使用默认模板运行的用户意外覆盖，请更改模板参数的`BootstrapVariant`默认值。CDK CLI 只允许使用与当前部署的模板相同`BootstrapVariant`且相同或更高的模板覆盖引导堆栈。

然后，您可以按照中的说明部署修改后的模板the section called “从模板引导 AWS CloudFormation”，或者使用cdk bootstrap --template。

```
$ cdk bootstrap --template bootstrap-template.yaml
```

引导模板的差异

如前所述， AWS CDK v1 支持两个引导模板，即旧版和现代版。CDK v2 仅支持现代模板。作为参考，以下是这两个模板之间的高级区别。

功能	旧版（仅限 v1）	现代（v1 和 v2）
跨账户部署	不允许	已允许

功能	旧版（仅限 v1）	现代（v1 和 v2）
AWS CloudFormation 权限	使用当前用户的权限（由 AWS 配置文件、环境变量等决定）进行部署	使用配置引导堆栈时指定的权限进行部署（例如，使用）`--trust`
版本控制	只有一个版本的引导堆栈可用	Bootstrap 堆栈已有版本控制；可以在未来的版本中添加新资源， AWS CDK 应用程序可能需要最低版本
资源 *	Amazon S3 存储桶	Amazon S3 存储桶
		AWS KMS key
		IAM 角色
		亚马逊 ECR 存储库
		用于版本控制的 SSM 参数
资源命名	自动生成	确定性
存储桶加密	默认密钥	客户托管密钥

* 我们将根据需要向引导模板添加其他资源。

必须通过重新启动将使用旧版模板引导的环境升级为使用 CDK v2 的现代模板。在删除旧存储桶之前，至少重新部署环境中的所有 AWS CDK 应用程序一次。

堆栈合成器

您的 AWS CDK 应用程序需要了解其可用的引导资源才能成功合成可以部署的堆栈。堆栈合成器是一个控制堆栈模板合成方式的 AWS CDK 类。这包括它如何使用引导资源（例如，它如何引用存储在引导存储桶中的资产）。

AWS CDK的内置堆栈合成器被称为。`DefaultStackSynthesizer`它包括跨账户部署和 CDK Pipelines部署的功能。

当你使用属性实例化堆栈合成器时，你可以将堆栈合成器传递给堆栈。`synthesizer`

TypeScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

Python

```
MyStack(self, "MyStack",
    # stack properties
    synthesizer=DefaultStackSynthesizer(
        # synthesizer properties
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
    // stack properties
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    // synthesizer properties
    .build())
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
// stack properties
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
```

```
    {
        // synthesizer properties
    })
});
```

如果您不提供synthesizer属性，DefaultStackSynthesizer则使用。

自定义合成

根据您对引导模板所做的更改，您可能还需要自定义合成。DefaultStackSynthesizer可以使用如下所述的属性进行自定义。

如果这些属性都无法提供所需的自定义设置，则可以将合成器编写为实现IStackSynthesizer（可能源自）的类。DefaultStackSynthesizer

更改预选赛

将限定符添加到引导程序资源的名称中，以区分单独的引导程序堆栈中的资源。要在同一个环境（AWS 账户和区域）中部署两个不同版本的引导堆栈，堆栈必须具有不同的限定符。

此功能旨在在 CDK 本身的自动测试之间进行名称隔离。除非您可以非常精确地缩小分配给 AWS CloudFormation 执行角色的 IAM 权限，否则在单个账户中拥有两个不同的引导堆栈不会带来权限隔离的好处。因此，通常无需更改此值。

要更改限定符，请通过使用以下属性实例化合成器来配置DefaultStackSynthesizer其中一个：

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
```

```
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=DefaultStackSynthesizer(
        qualifier="MYQUALIFIER"
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    .qualifier("MYQUALIFIER")
    .build())
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        Qualifier = "MYQUALIFIER"
    })
});
```

或者通过将限定符配置为中的上下文密钥。`cdk.json`

```
{
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "MYQUALIFIER"
  }
}
```

更改资源名称

所有其他`DefaultStackSynthesizer`属性都与引导模板中的资源名称相关。只有在修改了引导模板并更改了资源名称或命名方案时，才需要提供这些属性中的任何一个。

所有属性都接受特殊占位符${Qualifier}${AWS::Partition}${AWS::AccountId}、和。${AWS::Region}这些占位符将分别替换为qualifier参数的值以及堆栈环境的 AWS 分区、账户 ID 和区域值。

以下示例显示了最常用的DefaultStackSynthesizer属性及其默认值，就像您在实例化合成器一样。有关完整列表，请参阅DefaultStackSynthesizerProps。

TypeScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
```

```
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,

})
```

JavaScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',
```

```
  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,
})
```

Python

```
DefaultStackSynthesizer(
  # Name of the S3 bucket for file assets
  file_assets_bucket_name="cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
  bucket_prefix="",

  # Name of the ECR repository for Docker image assets
  image_assets_repository_name="cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}",

  # ARN of the role assumed by the CLI and Pipeline to deploy here
  deploy_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
  deploy_role_external_id="",

  # ARN of the role used for file asset publishing (assumed from the CLI role)
  file_asset_publishing_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
  file_asset_publishing_external_id="",

  # ARN of the role used for Docker asset publishing (assumed from the CLI role)
  image_asset_publishing_role_arn="arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
  image_asset_publishing_external_id="",

  # ARN of the role passed to CloudFormation to execute the deployments
  cloud_formation_execution_role="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}",

  # ARN of the role used to look up context information in an environment
  lookup_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
```

```
  lookup_role_external_id="",

  # Name of the SSM parameter which describes the bootstrap stack version number
  bootstrap_stack_version_ssm_parameter="/cdk-bootstrap/${Qualifier}/version",

  # Add a rule to every template which verifies the required bootstrap stack version
  generate_bootstrap_version_rule=True,
)
```

Java

```
DefaultStackSynthesizer.Builder.create()
    // Name of the S3 bucket for file assets
    .fileAssetsBucketName("cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}")
    .bucketPrefix('')

    // Name of the ECR repository for Docker image assets
    .imageAssetsRepositoryName("cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}")

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    .deployRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}")
    .deployRoleExternalId("")

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    .fileAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}")
    .fileAssetPublishingExternalId("")

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    .imageAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}")
    .imageAssetPublishingExternalId("")

    // ARN of the role passed to CloudFormation to execute the deployments
    .cloudFormationExecutionRole("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}")

    .lookupRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}")
    .lookupRoleExternalId("")
```

```
    // Name of the SSM parameter which describes the bootstrap stack version number
    .bootstrapStackVersionSsmParameter("/cdk-bootstrap/${Qualifier}/version")

    // Add a rule to every template which verifies the required bootstrap stack
 version
    .generateBootstrapVersionRule(true)
.build()
```

C#

```
new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
{
    // Name of the S3 bucket for file assets
    FileAssetsBucketName = "cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
    BucketPrefix = "",

    // Name of the ECR repository for Docker image assets
    ImageAssetsRepositoryName = "cdk-${Qualifier}-container-assets-
${AWS::AccountId}-${AWS::Region}",

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    DeployRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
    DeployRoleExternalId = "",

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    FileAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
    FileAssetPublishingExternalId = "",

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    ImageAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
    ImageAssetPublishingExternalId = "",

    // ARN of the role passed to CloudFormation to execute the deployments
    CloudFormationExecutionRole = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-
${AWS::Region}",
```

```
    LookupRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
    LookupRoleExternalId = "",

    // Name of the SSM parameter which describes the bootstrap stack version number
    BootstrapStackVersionSsmParameter = "/cdk-bootstrap/${Qualifier}/version",

    // Add a rule to every template which verifies the required bootstrap stack
 version
    GenerateBootstrapVersionRule = true,
})
```

引导模板合约

引导堆栈的要求取决于所使用的堆栈合成器。如果您自己编写堆栈合成器，则可以完全控制合成器所需的引导资源以及合成器如何找到它们。

本节描述DefaultStackSynthesizer了对引导模板的期望。

版本控制

模板应包含用于创建具有众所周知名称的 SSM 参数的资源，以及用于反映模板版本的输出。

```
Resources:
  CdkBootstrapVersion:
    Type: AWS::SSM::Parameter
    Properties:
      Type: String
      Name:
        Fn::Sub: '/cdk-bootstrap/${Qualifier}/version'
      Value: 4
Outputs:
  BootstrapVersion:
    Value:
      Fn::GetAtt: [CdkBootstrapVersion, Value]
```

角色

DefaultStackSynthesizer需要五个 IAM 角色用于五个不同的目的。如果您未使用默认角色，则必须将要使用的角色的 ARN 告诉合成器。

角色如下：

- 部署角色由 AWS CDK Toolkit 承担，然后部署 AWS CodePipeline 到环境中。它`AssumeRolePolicy`控制谁可以部署到环境中。在模板中，您可以看到此角色所需的权限。
- 查找角色由 AWS CDK Toolkit 代替，用于在环境中执行上下文查找。它`AssumeRolePolicy`控制谁可以部署到环境中。可以在模板中看到此角色所需的权限。
- 文件发布角色和图像发布角色由 AWS CDK 工具包和 AWS CodeBuild 项目承担，用于将资源发布到环境中。它们分别用于写入 S3 存储桶和 ECR 存储库。这些角色需要对这些资源的写入权限。
- AWS CloudFormation 执行角色被传递给 AWS CloudFormation 以执行实际部署。它的权限是部署执行所依据的权限。权限作为列出托管策略 ARN 的参数传递到堆栈。

输出

该 AWS CDK 工具包要求引导堆栈中存在以下 CloudFormation 输出。

- `BucketName`: 文件资产存储桶的名称
- `BucketDomainName`: 域名格式的文件资产存储桶
- `BootstrapVersion`: 引导堆栈的当前版本

模板历史记录

bootstrap模板是版本化的，并且会随着时间的推移而自行演变。 AWS CDK 如果您提供自己的引导程序模板，请使用规范的默认模板使其保持最新状态。您需要确保您的模板能够继续使用所有 CDK 功能。

> **Note**
>
> 默认情况下，早期版本的引导模板 AWS KMS key 在每个引导环境中创建了。为避免对 KMS 密钥收费，请使用重新启动这些环境。`--no-bootstrap-customer-key`当前的默认值为无 KMS 密钥，这有助于避免这些费用。

本节包含每个版本中所做更改的列表。

模板版本	AWS CDK 版本	更改
1	1.40.0	包含存储桶、密钥、存储库和角色的模板的初始版本。

模板版本	AWS CDK 版本	更改
2	1.45.0	将资源发布角色拆分为单独的文件和图像发布角色。
3	1.46.0	添加`FileAssetKeyArn` 导出，以便能够向资产使用者添加解密权限。
4	1.61.0	AWS KMS 现在，权限通过 Amazon S3 是隐式的，不再需要`FileAsetKeyArn` 。添加`CdkBootstrapVersion` SSM 参数，这样就可以在不知道堆栈名称的情况下验证引导堆栈版本。
5	1.87.0	部署角色可以读取 SSM 参数。
6	1.108.0	添加独立于部署角色的查找角色。
6	1.109.0	为部署、文件发布和图像发布角色添加`aws-cdk:bootstrap-role` 标签。
7	1.110.0	部署角色无法再直接读取目标账户中的存储桶。（但是，此角色实际上是管理员，无论如何都可以随时使用其 AWS CloudFormation 权限使存储桶可读）。
8	1.114.0	查找角色对目标环境具有完全的只读权限，并且还有一个`aws-cdk:bootstrap-role` 标签。

模板版本	AWS CDK 版本	更改
9	2.1.0	修复了 Amazon S3 资产上传被常用加密 SCP 拒绝的问题。
10	2.4.0	现在，Amazon ECR ScanOnPush 已默认启用。
11	2.18.0	添加了允许 Lambda 从 Amazon ECR 存储库中提取数据的政策，使其能够在重启后幸存下来。
12	2.20.0	添加对实验的支持cdk import。
13	2.25.0	使引导创建的 Amazon ECR 存储库中的容器映像不可变。
14	2.34.0	默认情况下，在存储库级别关闭 Amazon ECR 图像扫描，以允许引导不支持图像扫描的区域。
15	2.60.0	无法标记 KMS 密钥。
16	2.69.0	解决了 Security Hub 查找 KMS.2 的问题。
17	2.72.0	解决了 Security Hub 发现的问题 ECR.3。
18	2.80.0	还原了针对版本 16 所做的更改，因为它们不适用于所有分区，因此不建议这样做。
19	2.106.1	恢复了对版本 18 所做的更改，其中 AccessControl 属性已从模板中移除。(#27964)

模板版本	AWS CDK 版本	更改
20	2.119.0	向 AWS CloudFormation 部署 IAM 角色添加`ssm:GetParameters` 操作。有关更多信息，请参阅 #28336.

Security Hub 的调查结果

如果您正在使用 AWS Security Hub，则可能会看到有关 AWS CDK Bootstrapping过程创建的某些资源的调查结果报告。Security Hub 的发现可帮助您找到资源配置，您应该仔细检查其准确性和安全性。我们已经通过 Sec AWS urity 审查了这些特定的资源配置，并确信它们不会构成安全问题。

[KMS.2] IAM 主体不应有允许对所有 KMS 密钥进行解密操作的 IAM 内联策略

部署角色（默认名称`cdk-hnb659fds-deploy-role-ACCOUNT-REGION`）有权读取存储在 Amazon S3 中的加密数据。该策略本身并未授予任何数据的权限：只能解密从 Amazon S3 读取的数据，并且只能从明确允许 Deploy 角色通过其存储桶策略读取数据的存储桶以及明确允许 Deploy 角色使用其密钥策略使用它们进行解密的密钥。此语句用于允许 Pip AWS CDK elines 执行跨账户部署。

为什么 Security Hub 会举报这个？ 该策略包含一个`Resource: *`组合`Condition`条款；Security Hub 正在标记。**这是必需的，因为在账户被引导时，Pipelin AWS CDK es 为 Artif CodePipeline act Bucket 创建的 AWS KMS 密钥尚不存在，因此我们无法引用其 ARN。此外，Security Hub 在其推理中未将该`Condition`条款包含在政策声明中。

如果我想修复这个发现怎么办？ 只要 AWS KMS 密钥上的资源策略不是不必要的宽松，当前的角色策略就不允许 Deploy 角色访问超过应有的数据。如果你仍然想摆脱这个发现，你可以通过以下两种方式之一自定义引导堆栈（使用上面概述的过程）来实现：

- 如果您不使用 AWS CDK 流水线进行跨账户部署：请`Sid: PipelineCrossAccountArtifactsBucket`从部署角色中移除带的语句；或者
- 如果您使用 AWS CDK 流水线进行跨账户部署：部署 AWS CDK 流水线后，请查找 Artifact Bucket 的 AWS KMS 密钥 ARN，并将语句的密钥 ARN 替换`Resource: *``Sid: PipelineCrossAccountArtifactsBucket`为实际的密钥 ARN。

资源

资源是您配置为在应用程序 AWS 服务 中使用的资源。资源是的功能 AWS CloudFormation。通过在 AWS CloudFormation 模板中配置资源及其属性，您可以部署 AWS CloudFormation 到以配置资源。使用 AWS Cloud Development Kit (AWS CDK)，您可以通过构造配置资源。然后，您部署您的 CDK 应用程序，其中包括合成 AWS CloudFormation 模板并部署 AWS CloudFormation 到以配置您的资源。

主题

使用构造配置资源

如中所述the section called “构造”， AWS CDK 提供了一个丰富的类库，其中包含代表所有 AWS 资源的AWS 构造（称为构造）。

要使用资源对应的构造创建资源实例，请将作用域作为第一个参数、构造的逻辑 ID 和一组配置属性（道具）传入。例如，以下是如何使用构造库中的 SQS .Queue 构造创建带 AWS KMS 加密功能的 Amazon SQUEU 队列。 AWS

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

Python

```
import aws_cdk.aws_sqs as sqs

sqs.Queue(self, "MyQueue", encryption=sqs.QueueEncryption.KMS_MANAGED)
```

Java

```
import software.amazon.awscdk.services.sqs.*;

Queue.Builder.create(this, "MyQueue").encryption(
        QueueEncryption.KMS_MANAGED).build();
```

C#

```
using Amazon.CDK.AWS.SQS;

new Queue(this, "MyQueue", new QueueProps
{
    Encryption = QueueEncryption.KMS_MANAGED
});
```

有些配置道具是可选的，而且在许多情况下具有默认值。在某些情况下，所有道具都是可选的，最后一个参数可以完全省略。

资源属性

AWS 构造库中的大多数资源都公开属性，这些属性在部署时由解析 AWS CloudFormation。属性以资源类的属性形式公开，并以类型名称为前缀。以下示例说明如何使用 (queueUrlPython:queue_url) 属性获取亚马逊 SQS 队列的网址。

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

Python

```
import aws_cdk.aws_sqs as sqs

queue = sqs.Queue(self, "MyQueue")
url = queue.queue_url # => A string representing a deploy-time value
```

Java

```
Queue queue = new Queue(this, "MyQueue");
String url = queue.getQueueUrl();    // => A string representing a deploy-time value
```

C#

```
var queue = new Queue(this, "MyQueue");
var url = queue.QueueUrl; // => A string representing a deploy-time value
```

有关如何将部署时属性 AWS CDK 编码为字符串的信息，请参阅the section called “令牌”。

引用资源

配置资源时，通常必须引用其他资源的属性。示例如下：

- 亚马逊弹性容器服务 (Amazon ECS) Service 资源需要对其运行的集群进行引用。
- 亚马逊 CloudFront 分发需要引用包含源代码的亚马逊简单存储服务 (Amazon S3) 存储桶。

您可以通过以下任一方式引用资源：

- 通过传递在 CDK 应用程序中定义的资源，无论是在同一个堆栈中还是在不同的堆栈中
- 通过传递代理对象，该代理对象引用您的 AWS 账户中定义的资源，该资源是根据资源的唯一标识符（例如 ARN）创建的

如果构造的属性代表另一种资源的构造，则其类型就是该构造的接口类型。例如，Amazon ECS 结构采用`cluster`的属性类型为`ecs.ICluster`。另一个例子是采用以下类型的属性 `sourceBucket` (Python:`source_bucket`) 的 CloudFront 分布结构`s3.IBucket`。

您可以直接传递在同一个 AWS CDK 应用程序中定义的适当类型的任何资源对象。以下示例定义了一个 Amazon ECS 集群，然后用它来定义 Amazon ECS 服务。

TypeScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

JavaScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

Python

```
cluster = ecs.Cluster(self, "Cluster")

service = ecs.Ec2Service(self, "Service", cluster=cluster)
```

Java

```
Cluster cluster = new Cluster(this, "Cluster");
Ec2Service service = new Ec2Service(this, "Service",
        new Ec2ServiceProps.Builder().cluster(cluster).build());
```

C#

```
var cluster = new Cluster(this, "Cluster");
```

```
var service = new Ec2Service(this, "Service", new Ec2ServiceProps { Cluster =
 cluster });
```

引用其他堆栈中的资源

您可以引用不同堆栈中的资源，前提是这些资源是在同一个应用程序中定义的，并且处于相同的 AWS 环境中。通常使用以下模式：

- 将对构造的引用存储为生成资源的堆栈的属性。（要获取对当前构造堆栈的引用，请使用`Stack.of(this)`。）
- 将此引用传递给堆栈的构造函数，该构造函数将资源用作参数或属性。然后，使用堆栈将其作为属性传递给任何需要它的构造。

以下示例定义了一个堆栈`stack1`。此堆栈定义一个 Amazon S3 存储桶，并存储对存储桶结构的引用作为堆栈的属性。然后，应用程序定义第二个堆栈`stack2`，该堆栈在实例化时接受一个存储桶。`stack2`例如，可以定义一个使用存储桶进行数据存储的 AWS Glue 表。

TypeScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

JavaScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
```

```
});
```

Python

```
prod = core.Environment(account="123456789012", region="us-east-1")

stack1 = StackThatProvidesABucket(app, "Stack1", env=prod)

# stack2 will take a property "bucket"
stack2 = StackThatExpectsABucket(app, "Stack2", bucket=stack1.bucket, env=prod)
```

Java

```
// Helper method to build an environment
static Environment makeEnv(String account, String region) {
    return Environment.builder().account(account).region(region)
            .build();
}

App app = new App();

Environment prod = makeEnv("123456789012", "us-east-1");

StackThatProvidesABucket stack1 = new StackThatProvidesABucket(app, "Stack1",
        StackProps.builder().env(prod).build());

// stack2 will take an argument "bucket"
StackThatExpectsABucket stack2 = new StackThatExpectsABucket(app, "Stack,",
        StackProps.builder().env(prod).build(), stack1.bucket);
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment { Account = account, Region = region };
}

var prod = makeEnv(account: "123456789012", region: "us-east-1");

var stack1 = new StackThatProvidesABucket(app, "Stack1", new StackProps { Env =
 prod });

// stack2 will take a property "bucket"
```

```
var stack2 = new StackThatExpectsABucket(app, "Stack2", new StackProps { Env = prod,
    bucket = stack1.Bucket});
```

如果 AWS CDK 确定资源位于相同的环境中，但位于不同的堆栈中，则它会自动合成生成堆栈中的 AWS CloudFormation 导出和使用堆栈ImportValue中的 Fn::，以将该信息从一个堆栈传输到另一个堆栈。

解决依赖关系死锁

引用来自不同堆栈中一个堆栈的资源会在两个堆栈之间产生依赖关系。这样可以确保它们按正确的顺序部署。部署堆栈后，这种依赖关系是具体的。之后，从使用堆栈中删除共享资源的使用可能会导致意外的部署失败。如果两个堆栈之间存在另一种依赖关系，迫使它们按相同的顺序部署，则会发生这种情况。如果直接由 CDK Toolkit 选择要先部署的生成堆栈，也可能在没有依赖关系的情况下发生。由于不再需要 AWS CloudFormation 导出，因此已将其从生成堆栈中删除，但是由于尚未部署更新，导出的资源仍在使用堆栈中使用。因此，部署生产者堆栈失败。

要打破这种僵局，请从使用堆栈中移除对共享资源的使用。（这将从生成堆栈中移除自动导出。）接下来，使用与自动生成的导出完全相同的逻辑 ID 将相同的导出内容手动添加到生成堆栈中。取消使用消费堆栈中共享资源的使用，然后部署两个堆栈。然后，移除手动导出（如果不再需要共享资源，则移除共享资源），然后重新部署两个堆栈。堆栈的`exportValue()`方法是为此目的创建手动导出的便捷方法。（参见链接方法参考中的示例。）

引用您 AWS 账户中的资源

假设你想在 AWS CDK 应用程序中使用 AWS 账户中已有的资源。这可能是通过控制台、 AWS SDK、直接使用 AWS CloudFormation或在其他 AWS CDK 应用程序中定义的资源。您可以将资源的 ARN（或其他标识属性或属性组）转换为代理对象。代理对象通过在资源的类上调用静态工厂方法来作为对资源的引用。

当您创建这样的代理时，外部资源不会成为您的 AWS CDK 应用程序的一部分。因此，您在 AWS CDK 应用程序中对代理所做的更改不会影响已部署的资源。但是，可以将代理传递给需要该类型资源的任何 AWS CDK 方法。

以下示例说明如何基于带有 ARN arn: aws: s3::: my-bucket-name 的现有存储桶和基于具有特定 ID 的现有 VPC 的 A mazon Virtual Private Cloud 来引用存储桶。

TypeScript

```
// Construct a proxy for a bucket by its name (must be same account)
```

```
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde',
});
```

JavaScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde'
});
```

Python

```
# Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.from_bucket_name(self, "MyBucket", "my-bucket-name")

# Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.from_bucket_arn(self, "MyBucket", "arn:aws:s3:::my-bucket-name")

# Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")
```

Java

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.fromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.fromBucketArn(this, "MyBucket",
        "arn:aws:s3:::my-bucket-name");
```

```
// Construct a proxy for an existing VPC from its attribute(s)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
        .vpcId("vpc-1234567890abcdef").build());
```

C#

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.FromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.FromBucketArn(this, "MyBucket", "arn:aws:s3:::my-bucket-name");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
{
    VpcId = "vpc-1234567890abcdef"
});
```

让我们仔细看看这个Vpc.fromLookup()方法。由于ec2.Vpc结构很复杂，因此您可以通过多种方式选择要与 CDK 应用程序一起使用的 VPC。为了解决这个问题，VPC 结构有一个fromLookup静态方法 (Python:from_lookup)，允许您在综合时通过查询您的 AWS 账户来查找所需的 Amazon VPC。

要使用Vpc.fromLookup()，合成堆栈的系统必须有权访问拥有 Amazon VPC 的账户。这是因为 CDK 工具包会在综合时查询账户以找到正确的 Amazon VPC。

此外，仅Vpc.fromLookup()适用于使用明确的账户和区域定义的堆栈（请参阅the section called “环境”）。如果 AWS CDK 尝试从与环境无关的堆栈中查找 Amazon VPC，则 CDK 工具包不知道要查询哪个环境才能找到该 VPC。

您必须提供足以唯一标识您 AWS 账户中的 VPC 的Vpc.fromLookup()属性。例如，只能有一个默认 VPC，因此将 VPC 指定为默认 VPC 就足够了。

TypeScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
```

```
  isDefault: true
});
```

Python

```
ec2.Vpc.from_lookup(self, "DefaultVpc", is_default=True)
```

Java

```
Vpc.fromLookup(this, "DefaultVpc", VpcLookupOptions.builder()
        .isDefault(true).build());
```

C#

```
Vpc.FromLookup(this, id = "DefaultVpc", new VpcLookupOptions { IsDefault = true });
```

您还可以使用该`tags`属性按标签查询 VPC。在创建 Amazon VPC 时，您可以使用 AWS CloudFormation 或向其添加标签 AWS CDK。创建标签后，您可以随时使用 AWS Management Console AWS CLI、或 AWS SDK 编辑标签。除了您自己添加的任何标签外，还 AWS CDK 会自动向其创建的所有 VPC 添加以下标签。

- 名称 — VPC 的名称。
- aws-cdk: subnet-name — 子网的名称。
- aws-cdk: subnet-type — 子网的类型：公有子网、私有子网或隔离子网。

TypeScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

Python

```
ec2.Vpc.from_lookup(self, "PublicVpc",
```

```
    tags={"aws-cdk:subnet-type": "Public"})
```

Java

```
Vpc.fromLookup(this, "PublicVpc", VpcLookupOptions.builder()
        .tags(java.util.Map.of("aws-cdk:subnet-type", "Public"))  // Java 9 or later
        .build());
```

C#

```
Vpc.FromLookup(this, id = "PublicVpc", new VpcLookupOptions
     { Tags = new Dictionary<string, string> { ["aws-cdk:subnet-type"] =
 "Public" });
```

的结果缓`Vpc.fromLookup()`存在项目`cdk.context.json`文件中。（请参见 the section called “上下文”。）将此文件提交给版本控制，这样您的应用程序就可以继续引用同一 Amazon VPC。即使您稍后更改了 VPC 的属性，从而导致选择了不同的 VPC，这也会起作用。如果您在无法访问定义VPC的 AWS 账户（例如 CDK Pipelines）的环境中部署堆栈，这一点尤其重要。

尽管你可以在任何使用 AWS CDK 应用程序中定义的类似资源的地方使用外部资源，但你无法对其进行修改。例如，在外部调用 `addToResourcePolicy` (Python:`add_to_resource_policy`) `s3.Bucket` 没有任何作用。

资源物理名称

中资源的逻辑名称不同 AWS CloudFormation 于部署 AWS Management Console 后在中显示的资源名称 AWS CloudFormation。他们 AWS CDK 称这些姓氏为物理名称。

例如， AWS CloudFormation 可能使用上一个示例中的逻辑 ID `Stack2MyBucket4DD88B4F` 和物理名称创建 Amazon S3 存储桶`stack2mybucket4dd88b4f-iuv1rbv9z3to`。

在创建代表资源的构造时，您可以使用属性 Name 指定物理<resourceType>名称。以下示例使用物理名称创建一个 Amazon S3 存储桶`my-bucket-name`。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name',
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name'
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket-name")
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName("my-bucket-name").build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps { BucketName = "my-bucket-
name" });
```

为资源分配物理名称有一些缺点 AWS CloudFormation。最重要的是，如果为资源分配了物理名称，则任何需要替换资源的已部署资源更改（例如对资源属性在创建后不可变的更改）都将失败。如果您最终处于这种状态，唯一的解决方案是删除 AWS CloudFormation 堆栈，然后重新部署 AWS CDK 应用程序。有关详细信息，请参阅AWS CloudFormation 文档。

在某些情况下，例如在创建具有跨环境引用的 AWS CDK 应用程序时，需要物理名称 AWS CDK 才能正常运行。在这些情况下，如果你不想费心自己想出一个真实的名字，你可以让它给你起 AWS CDK 名字。为此，请使用特殊值`PhysicalName.GENERATE_IF_NEEDED`，如下所示。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED,
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
```

```
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket",
                           bucket_name=core.PhysicalName.GENERATE_IF_NEEDED)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName(PhysicalName.GENERATE_IF_NEEDED).build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps
    { BucketName = PhysicalName.GENERATE_IF_NEEDED });
```

传递唯一的资源标识符

如上一节所述，应尽可能通过引用传递资源。但是，在某些情况下，您别无选择，只能通过资源的一个属性来引用资源。示例用例包括以下内容：

- 当你使用低级 AWS CloudFormation 资源时。
- 当您需要向 AWS CDK 应用程序的运行时组件公开资源时，例如通过环境变量引用 Lambda 函数时。

这些标识符可用作资源的属性，如下所示。

TypeScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

JavaScript

```
bucket.bucketName
```

```
lambdaFunc.functionArn
securityGroup.groupArn
```

Python

```
bucket.bucket_name
lambda_func.function_arn
security_group_arn
```

Java

Java AWS CDK 绑定使用获取器方法作为属性。

```
bucket.getBucketName()
lambdaFunc.getFunctionArn()
securityGroup.getGroupArn()
```

C#

```
bucket.BucketName
lambdaFunc.FunctionArn
securityGroup.GroupArn
```

以下示例说明如何将生成的存储桶名称传递给 AWS Lambda 函数。

TypeScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  },
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'Bucket');
```

```
new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "Bucket")

lambda.Function(self, "MyLambda", environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "Bucket");

Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Java 9 or later
                "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new Bucket(this, "Bucket");

new Function(this, "MyLambda", new FunctionProps
{
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

在资源之间授予权限

更高级别的结构通过提供简单的、基于意图的 API 来表达权限要求，从而实现最低权限权限。例如，许多 L2 结构提供授予方法，您可以使用这些方法向实体（例如 IAM 角色或用户）授予使用资源的权限，而无需手动创建 IAM 权限声明。

以下示例创建权限以允许 Lambda 函数的执行角色读取对象并将其写入特定 Amazon S3 存储桶。如果 Amazon S3 存储桶使用 AWS KMS 密钥加密，则此方法还会向 Lambda 函数的执行角色授予使用该密钥进行解密的权限。

TypeScript

```
if (bucket.grantReadWrite(func).success) {
  // ...
}
```

JavaScript

```
if ( bucket.grantReadWrite(func).success) {
  // ...
}
```

Python

```
if bucket.grant_read_write(func).success:
    # ...
```

Java

```
if (bucket.grantReadWrite(func).getSuccess()) {
    // ...
}
```

C#

```
if (bucket.GrantReadWrite(func).Success)
{
    // ...
}
```

grant 方法返回一个`iam.Grant`对象。使用`Grant`对象的`success`属性来确定是否有效应用了授权（例如，它可能未应用于外部资源）。您也可以使用`Grant`对象的 `assertSuccess` (Python:`assert_success`) 方法来强制授权已成功应用。

如果特定的授权方法不适用于特定的用例，则可以使用通用的授权方法来定义具有指定操作列表的新授权。

以下示例说明如何授予 Lambda 函数访问亚马逊 DynamoDB 操作的权限。`CreateBackup`

TypeScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

JavaScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

Python

```
table.grant(func, "dynamodb:CreateBackup")
```

Java

```
table.grant(func, "dynamodb:CreateBackup");
```

C#

```
table.Grant(func, "dynamodb:CreateBackup");
```

许多资源（例如 Lambda 函数）都需要在执行代码时扮演角色。配置属性使您可以指定`iam.IRole`。如果未指定角色，则该函数会自动创建专门用于此用途的角色。然后，您可以对资源使用授权方法向角色添加语句。

授予方法是使用较低级别的 API 构建的，用于处理 IAM 策略。策略被建模为PolicyDocument对象。使用`addToRolePolicy`方法 (Python:) 将语句直接添加到角色（或构造的附加角色`add_to_role_policy`），或者使用 (Python:) 方法向资源的`Bucket`策略`addToResourcePolicy`（例如策略`add_to_resource_policy`）中添加语句。

资源指标和警报

许多资源会发出可用于设置监控仪表板和警报的 CloudWatch 指标。更高级别的构造具有指标方法，允许您访问指标，而无需查找要使用的正确名称。

以下示例说明如何在 Amazon SQS 队列`ApproximateNumberOfMessagesNotVisible`的超过 100 时定义警报。

TypeScript

```
import * as cw from '@aws-cdk/aws-cloudwatch';
import * as sqs from '@aws-cdk/aws-sqs';
import { Duration } from '@aws-cdk/core';

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5),
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100,
  // ...
});
```

JavaScript

```
const cw = require('@aws-cdk/aws-cloudwatch');
const sqs = require('@aws-cdk/aws-sqs');
const { Duration } = require('@aws-cdk/core');

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5)
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100
  // ...
});
```

Python

```
import aws_cdk.aws_cloudwatch as cw
import aws_cdk.aws_sqs as sqs
from aws_cdk.core import Duration
```

```
queue = sqs.Queue(self, "MyQueue")
metric = queue.metric_approximate_number_of_messages_not_visible(
    label="Messages Visible (Approx)",
    period=Duration.minutes(5),
    # ...
)
metric.create_alarm(self, "TooManyMessagesAlarm",
    comparison_operator=cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    threshold=100,
    # ...
)
```

Java

```
import software.amazon.awscdk.core.Duration;
import software.amazon.awscdk.services.sqs.Queue;
import software.amazon.awscdk.services.cloudwatch.Metric;
import software.amazon.awscdk.services.cloudwatch.MetricOptions;
import software.amazon.awscdk.services.cloudwatch.CreateAlarmOptions;
import software.amazon.awscdk.services.cloudwatch.ComparisonOperator;

Queue queue = new Queue(this, "MyQueue");

Metric metric = queue
        .metricApproximateNumberOfMessagesNotVisible(MetricOptions.builder()
                .label("Messages Visible (Approx)")
                .period(Duration.minutes(5)).build());

metric.createAlarm(this, "TooManyMessagesAlarm", CreateAlarmOptions.builder()
                .comparisonOperator(ComparisonOperator.GREATER_THAN_THRESHOLD)
                .threshold(100)
                // ...
                .build());
```

C#

```
using cdk = Amazon.CDK;
using cw = Amazon.CDK.AWS.CloudWatch;
using sqs = Amazon.CDK.AWS.SQS;

var queue = new sqs.Queue(this, "MyQueue");
var metric = queue.MetricApproximateNumberOfMessagesNotVisible(new cw.MetricOptions
```

```
{
    Label = "Messages Visible (Approx)",
    Period = cdk.Duration.Minutes(5),
    // ...
});
metric.CreateAlarm(this, "TooManyMessagesAlarm", new cw.CreateAlarmOptions
{
    ComparisonOperator = cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    Threshold = 100,
    // ..
});
```

如果没有针对特定指标的方法，则可以使用通用指标方法手动指定指标名称。

也可以将指标添加到 CloudWatch 仪表板中。请参阅 CloudWatch。

网络流量

在许多情况下，您必须启用网络权限才能使应用程序正常运行，例如当计算基础设施需要访问持久层时。建立或侦听连接的资源会公开启用流量的方法，包括设置安全组规则或网络 ACL。

iconnecTable 资源有一个`connections`属性，即网络流量规则配置的网关。

您可以使用`allow`方法使数据能够在给定的网络路径上流动。以下示例启用与网络的 HTTPS 连接以及来自 Amazon EC2 Auto Scaling 组的传入连接`fleet2`。

TypeScript

```
import * as asg from '@aws-cdk/aws-autoscaling';
import * as ec2 from '@aws-cdk/aws-ec2';

const fleet1: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

JavaScript

```
const asg = require('@aws-cdk/aws-autoscaling');
const ec2 = require('@aws-cdk/aws-ec2');

const fleet1 = asg.AutoScalingGroup();

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2 = asg.AutoScalingGroup();
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

Python

```
import aws_cdk.aws_autoscaling as asg
import aws_cdk.aws_ec2 as ec2

fleet1 = asg.AutoScalingGroup( ... )

# Allow surfing the (secure) web
fleet1.connections.allow_to(ec2.Peer.any_ipv4(),
  ec2.Port(PortProps(from_port=443, to_port=443)))

fleet2 = asg.AutoScalingGroup( ... )
fleet1.connections.allow_from(fleet2, ec2.Port.all_traffic())
```

Java

```
import software.amazon.awscdk.services.autoscaling.AutoScalingGroup;
import software.amazon.awscdk.services.ec2.Peer;
import software.amazon.awscdk.services.ec2.Port;

AutoScalingGroup fleet1 = AutoScalingGroup.Builder.create(this, "MyFleet")
        /* ... */.build();

// Allow surfing the (secure) Web
fleet1.getConnections().allowTo(Peer.anyIpv4(),
        Port.Builder.create().fromPort(443).toPort(443).build());

AutoScalingGroup fleet2 = AutoScalingGroup.Builder.create(this, "MyFleet2")
        /* ... */.build();
```

```
fleet1.getConnections().allowFrom(fleet2, Port.allTraffic());
```

C#

```
using cdk = Amazon.CDK;
using asg = Amazon.CDK.AWS.AutoScaling;
using ec2 = Amazon.CDK.AWS.EC2;

// Allow surfing the (secure) Web
var fleet1 = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
 { /* ... */ });
fleet1.Connections.AllowTo(ec2.Peer.AnyIpv4(), new ec2.Port(new ec2.PortProps
  { FromPort = 443, ToPort = 443 });

var fleet2 = new asg.AutoScalingGroup(this, "MyFleet2", new
 asg.AutoScalingGroupProps { /* ... */ });
fleet1.Connections.AllowFrom(fleet2, ec2.Port.AllTraffic());
```

某些资源具有与之关联的默认端口。示例包括公共端口上的负载均衡器的侦听器，以及数据库引擎接受 Amazon RDS 数据库实例连接的端口。在这种情况下，您可以强制实施严格的网络控制，而不必手动指定端口。为此，请使用`allowDefaultPortFrom`和`allowToDefaultPort`方法 (Python:`allow_default_port_from`,`allow_to_default_port`)。

以下示例说明如何启用来自任意 IPV4 地址的连接以及来自 Auto Scaling 组的连接以访问数据库。

TypeScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

JavaScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

Python

```
listener.connections.allow_default_port_from_any_ipv4("Allow public access")
```

```
fleet.connections.allow_to_default_port(rds_database, "Fleet can access database")
```

Java

```
listener.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");

fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

C#

```
listener.Connections.AllowDefaultPortFromAnyIpv4("Allow public access");

fleet.Connections.AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

事件处理

有些资源可以充当事件源。使用addEventNotification方法 (Python:add_event_notification) 将事件目标注册到资源发出的特定事件类型。除此之外，addXxxNotification方法还提供了一种为常见事件类型注册处理程序的简单方法。

以下示例说明了如何在将对象添加到 Amazon S3 存储桶时触发 Lambda 函数。

TypeScript

```
import * as s3nots from '@aws-cdk/aws-s3-notifications';

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

JavaScript

```
const s3nots = require('@aws-cdk/aws-s3-notifications');

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

Python

```
import aws_cdk.aws_s3_notifications as s3_nots

handler = lambda_.Function(self, "Handler", ...)
bucket = s3.Bucket(self, "Bucket")
bucket.add_object_created_notification(s3_nots.LambdaDestination(handler))
```

Java

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.s3.notifications.LambdaDestination;

Function handler = Function.Builder.create(this, "Handler")/* ... */.build();
Bucket bucket = new Bucket(this, "Bucket");
bucket.addObjectCreatedNotification(new LambdaDestination(handler));
```

C#

```
using lambda = Amazon.CDK.AWS.Lambda;
using s3 = Amazon.CDK.AWS.S3;
using s3Nots = Amazon.CDK.AWS.S3.Notifications;

var handler = new lambda.Function(this, "Handler", new lambda.FunctionProps { .. });
var bucket = new s3.Bucket(this, "Bucket");
bucket.AddObjectCreatedNotification(new s3Nots.LambdaDestination(handler));
```

移除政策

维护永久性数据的资源，例如数据库、Amazon S3 存储桶和 Amazon ECR 注册表，都有移除政策。移除策略指示在包含永久对象的 AWS CDK 堆栈被销毁时是否将其删除。指定删除策略的值可通过 AWS CDK `core`模块中的`RemovalPolicy`枚举获得。

> **Note**
>
> 除了持久存储数据的资源之外，还可能有用于其他目的`removalPolicy`的资源。例如，Lambda 函数版本使用`removalPolicy`属性来确定在部署新版本时是否保留给定版本。与 Amazon S3 存储桶或 DynamoDB 表的删除策略相比，它们具有不同的含义和默认值。

值	意思
`RemovalPolicy.RETAIN`	Keep the contents of the resource when destroying the stack (default). The resource is orphaned from the stack and must be deleted manually. If you attempt to re-deploy the stack while the resource still exists, you will receive an error message due to a name conflict.
`RemovalPolicy. ##`	The resource will be destroyed along with the stack.

AWS CloudFormation 不会移除包含文件的 Amazon S3 存储桶，即使其删除策略设置为。`DESTROY`尝试这样做是 AWS CloudFormation 错误的。要在销毁存储桶之前从存储桶中 AWS CDK 删除所有文件，请将存储桶的`autoDeleteObjects`属性设置为`true`。

以下是创建 Amazon S3 存储桶`DESTROY`并将其`autoDeleteOjbects`设置为`RemovalPolicy`的示例`true`。

TypeScript

```
import * as cdk from '@aws-cdk/core';
import * as s3 from '@aws-cdk/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}
```

JavaScript

```
const cdk = require('@aws-cdk/core');
const s3 = require('@aws-cdk/aws-s3');
```

```
class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY,
            auto_delete_objects=True)
```

Java

```
software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.s3.*;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY)
```

```
                .autoDeleteObjects(true).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY,
        AutoDeleteObjects = true
    });
}
```

您也可以通过`applyRemovalPolicy()`方法将移除策略直接应用于底层 AWS CloudFormation 资源。此方法适用于某些在 L2 资源的 props 中没有`removalPolicy`属性的有状态资源。示例包括：

- AWS CloudFormation 堆栈
- Amazon Cognito 用户群体
- 亚马逊 DocumentDB 数据库实例
- 亚马逊 EC2 卷
- 亚马逊 OpenSearch 服务域名
- 亚马逊 FSx 文件系统
- Amazon SQS 队列

TypeScript

```
const resource = bucket.node.findChild('Resource') as cdk.CfnResource;
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

JavaScript

```
const resource = bucket.node.findChild('Resource');
```

```
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

Python

```
resource = bucket.node.find_child('Resource')
resource.apply_removal_policy(cdk.RemovalPolicy.DESTROY);
```

Java

```
CfnResource resource = (CfnResource)bucket.node.findChild("Resource");
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

C#

```

var resource = (CfnResource)bucket.node.findChild('Resource');
resource.ApplyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

> **Note**
>
> AWS CDK's `RemovalPolicy` 翻译为 AWS CloudFormation's。`DeletionPolicy`但是，中的默认设置 AWS CDK 是保留数据，这与 AWS CloudFormation 默认值相反。

标识符

在构建 AWS Cloud Development Kit (AWS CDK) 应用程序时，您将使用多种类型的标识符和名称。为了 AWS CDK 有效使用标识符并避免错误，了解标识符的类型非常重要。

标识符在创建时必须是唯一的；它们在您的 AWS CDK 应用程序中不必是全局唯一的。

如果您尝试在同一范围内创建具有相同值的标识符，则会 AWS CDK 引发异常。

主题

构造 ID

最常见的标识符是在实例化构造对象时作为第二个参数传递的标识符。id与所有标识符一样，此标识符只需要在创建它的范围内是唯一的，这是实例化构造对象时的第一个参数。

> **Note**
>
> 堆栈id的也是您在中用来引用堆栈的标识符the section called “AWS CDK 工具包”。

让我们来看一个示例，其中我们的应用程序中有两个带有标识符MyBucket的构造。第一个是在堆栈的作用域中定义的，标识符为Stack1。第二个是在堆栈的作用域中定义的，标识符为Stack2。因为它们是在不同的作用域中定义的，所以这不会造成任何冲突，而且它们可以毫无问题地共存于同一个应用程序中。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

class MyStack extends Stack {
  constructor(scope: Construct, id: string, props: StackProps = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MyStack extends Stack {
  constructor(scope, id, props = {}) {
    super(scope, id, props);
```

```
    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

Python

```
from aws_cdk import App, Construct, Stack, StackProps
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MyStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)
        s3.Bucket(self, "MyBucket")

app = App()
MyStack(app, 'Stack1')
MyStack(app, 'Stack2')
```

Java

```
// MyStack.java
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.services.s3.Bucket;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
```

```
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.App;

public class Main {
    public static void main(String[] args) {
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

C#

```
using Amazon.CDK;
using constructs;
using Amazon.CDK.AWS.S3;

public class MyStack : Stack
{
    public MyStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
    {
        new Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

路径

AWS CDK 应用程序中的构造形成了植根于该App类的层次结构。我们将从给定构造、其父构造、其祖父构造等到构造树根部的 ID 集合称为路径。

AWS CDK 通常将模板中的路径显示为字符串。关卡中的 ID 用斜线分隔，从根实例正下方的节点开始，根App实例通常是堆栈。例如，在前面的代码示例中，两个 Amazon S3 存储桶资源的路径是`Stack1/MyBucket`和`Stack2/MyBucket`。

您可以通过编程方式访问任何构造的路径，如以下示例所示。这会得到路径`myConstruct`（或者`my_construct`，正如 Python 开发者所写的那样）。由于 ID 在创建的范围内必须是唯一的，因此它们的路径在 AWS CDK 应用程序中始终是唯一的。

TypeScript

```
const path: string = myConstruct.node.path;
```

JavaScript

```
const path = myConstruct.node.path;
```

Python

```
path = my_construct.node.path
```

Java

```
String path = myConstruct.getNode().getPath();
```

C#

```
string path = myConstruct.Node.Path;
```

唯一 ID

AWS CloudFormation 要求模板中的所有逻辑 ID 都是唯一的。因此， AWS CDK 必须能够为应用程序中的每个构造生成唯一标识符。资源的路径是全局唯一的（从堆栈到特定资源的所有作用域的名称）。因此，通过连接路径的元素并添加一个 8 位数的哈希来 AWS CDK 生成必要的唯一标识符。（哈希值是区分不同路径（例如`A/B/C`和）所必需的`A/BC`，这会生成相同的 AWS CloudFormation 标识符。

AWS CloudFormation 标识符是字母数字，不能包含斜杠或其他分隔符。）AWS CDK 称此字符串为构造的唯一 ID。

通常，您的 AWS CDK 应用程序无需知道唯一 ID。但是，您可以通过编程方式访问任何构造的唯一 ID，如以下示例所示。

TypeScript

```
const uid: string = Names.uniqueId(myConstruct);
```

JavaScript

```
const uid = Names.uniqueId(myConstruct);
```

Python

```
uid = Names.unique_id(my_construct)
```

Java

```
String uid = Names.uniqueId(myConstruct);
```

C#

```
string uid = Names.Uniqueid(myConstruct);
```

该地址是另一种用于唯一区分 CDK 资源的唯一标识符。它源自路径的 SHA-1 哈希值，人类无法读取。但是，其长度恒定、相对较短（始终为 42 个十六进制字符）使其在 “传统” 唯一 ID 可能过长的情况下非常有用。某些构造可能会使用合成 AWS CloudFormation 模板中的地址而不是唯一 ID。同样，您的应用程序通常不需要知道其构造的地址，但您可以按如下方式检索构造的地址。

TypeScript

```
const addr: string = myConstruct.node.addr;
```

JavaScript

```
const addr = myConstruct.node.addr;
```

Python

```
addr = my_construct.node.addr
```

Java

```
String addr = myConstruct.getNode().getAddr();
```

C#

```
string addr = myConstruct.Node.Addr;
```

逻辑 ID

在为代表 AWS 资源的构造生成的 AWS CloudFormation 模板中，唯一 ID 用作资源的逻辑标识符（或逻辑名称）。

例如，在上一个示例中创建的 Amazon S3 存储桶会`Stack2`生成一个`AWS::S3::Bucket`资源。资源的逻辑 ID `Stack2MyBucket4DD88B4F` 位于生成的 AWS CloudFormation 模板中。（有关如何生成此标识符的详细信息，请参阅the section called “唯一 ID”。）

逻辑 ID 稳定性

避免在资源创建后更改其逻辑 ID。 AWS CloudFormation 通过资源的逻辑 ID 来标识资源。因此，如果您更改资源的逻辑 ID，则使用新的逻辑 ID AWS CloudFormation 创建新资源，然后删除现有资源。根据资源的类型，这可能会导致服务中断、数据丢失或两者兼而有之。

令牌

令牌表示只能在应用程序生命周期的稍后时间解析的值。例如，只有在合成模板时，才会分配您在CDK应用程序中定义的亚马逊简单存储服务 (Amazon S3) 存储桶 AWS CloudFormation 的名称。如果你打印这个`bucket.bucketName`属性（一个字符串），你会看到它包含如下内容：

```
${TOKEN[Bucket.Name.1234]}
```

这就是对代币进行 AWS CDK 编码的方式，该代币的价值在构造时尚不清楚，但稍后将可用。它们 AWS CDK 调用这些占位符标记。在本例中，它是一个编码为字符串的标记。

你可以像存储桶的名称一样传递这个字符串。在以下示例中，存储桶名称被指定为 AWS Lambda 函数的环境变量。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  }
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket")

fn = lambda_.Function(stack, "MyLambda",
        environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "MyBucket");

Function fn = Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Map.of requires Java 9+
            "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new s3.Bucket(this, "MyBucket");

var fn = new Function(this, "MyLambda", new FunctionProps {
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

当 AWS CloudFormation 模板最终合成时，标记将呈现为 AWS CloudFormation 内在函数`{ "Ref": "MyBucket" }`。在部署时， AWS CloudFormation 将此内部函数替换为创建的存储桶的实际名称。

主题

代币和代币编码

令牌是实现 iResolvable 接口的对象，该接口包含一个`resolve`方法。在合成过程中 AWS CDK 调用此方法以生成 AWS CloudFormation 模板的最终值。代币参与合成过程以生成任何类型的任意值。

> **Note**
>
> 你很少会直接使用该`IResolvable`界面。您很可能只会看到字符串编码版本的令牌。

其他函数通常只接受基本类型的参数，例如`string`或`number`。要在这些情况下使用标记，您可以在 CDK .Token 类上使用静态方法将它们编码为三种类型之一。

- `Token.asString`生成字符串编码（或调用 `.toString()` token 对象）
- `Token.asList`生成列表编码

- `Token.asNumber`生成数字编码

它们采用任意值（可以是）`IResolvable`，然后将它们编码为指定类型的原始值。

> **Important**
>
> 由于前面的任何一种类型都可能是编码标记，因此在解析或尝试读取其内容时要小心。例如，如果您尝试解析字符串以从中提取值，而该字符串是编码标记，则解析将失败。同样，如果您尝试查询数组的长度或使用数字执行数学运算，则必须首先验证它们不是经过编码的标记。

要检查值中是否有未解析的标记，请调用 `Token.isUnresolved` (Python:`is_unresolved`) 方法。

以下示例验证字符串值（可能是标记）的长度是否不超过 10 个字符。

TypeScript

```
if (!Token.isUnresolved(name) && name.length > 10) {
  throw new Error(`Maximum length for name is 10 characters`);
}
```

JavaScript

```
if ( !Token.isUnresolved(name) && name.length > 10) {
  throw ( new Error(`Maximum length for name is 10 characters`));
}
```

Python

```
if not Token.is_unresolved(name) and len(name) > 10:
    raise ValueError("Maximum length for name is 10 characters")
```

Java

```
if (!Token.isUnresolved(name) && name.length() > 10)
    throw new IllegalArgumentException("Maximum length for name is 10 characters");
```

C#

```
if (!Token.IsUnresolved(name) && name.Length > 10)
```

```
    throw new ArgumentException("Maximum length for name is 10 characters");
```

如果 name 是标记，则不执行验证，并且在生命周期的后期阶段（例如部署期间）仍可能发生错误。

> **Note**
>
> 您可以使用令牌编码来逃避类型系统。例如，您可以对在合成时生成数字值的标记进行字符串编码。如果您使用这些函数，则您有责任确保您的模板在合成后解析为可用状态。

字符串编码的标记

字符串编码的标记如下所示。

```
${TOKEN[Bucket.Name.1234]}
```

它们可以像常规字符串一样传递，也可以串联，如以下示例所示。

TypeScript

```
const functionName = bucket.bucketName + 'Function';
```

JavaScript

```
const functionName = bucket.bucketName + 'Function';
```

Python

```
function_name = bucket.bucket_name + "Function"
```

Java

```
String functionName = bucket.getBucketName().concat("Function");
```

C#

```
string functionName = bucket.BucketName + "Function";
```

如果您的语言支持，也可以使用字符串插值，如以下示例所示。

TypeScript

```
const functionName = `${bucket.bucketName}Function`;
```

JavaScript

```
const functionName = `${bucket.bucketName}Function`;
```

Python

```
function_name = f"{bucket.bucket_name}Function"
```

Java

```
String functionName = String.format("%sFunction". bucket.getBucketName());
```

C#

```
string functionName = $"${bucket.bucketName}Function";
```

避免以其他方式操纵字符串。例如，取一个字符串的子字符串很可能会破坏字符串标记。

列表编码的代币

列表编码的代币如下所示：

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

处理这些列表的唯一安全方法是将它们直接传递给其他构造。不能连接字符串列表形式的标记，也不能从令牌中提取元素。操作它们的唯一安全方法是使用诸如 fn.Select 之类的 AWS CloudFormation 内部函数。

数字编码的代币

数字编码的代币是一组微小的负浮点数，如下所示。

```
-1.8881545897087626e+289
```

与列表标记一样，您无法修改数字值，因为这样做可能会破坏数字标记。唯一允许的操作是将值传递给另一个构造。

懒惰的值

除了表示部署时间值（例如 AWS CloudFormation 参数）外，令牌还通常用于表示合成时的延迟值。这些值的最终值将在合成完成之前确定，但不会在构造值时确定。使用标记将文字字符串或数字值传递给另一个构造，而合成时的实际值可能取决于尚未进行的某些计算。

你可以使用Lazy类上的静态方法（例如 lazy.String 和 Laz y.Number）来构造表示合成时延迟值的标记。这些方法接受一个对象，该对象的produce属性是一个接受上下文参数并在调用时返回最终值的函数。

以下示例创建了一个 Auto Scaling 组，其容量是在创建后确定的。

TypeScript

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

JavaScript

```
let actualValue;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return (actualValue);
    }
  })
```

```
});

// At some later point
actualValue = 10;
```

Python

```
class Producer:
    def __init__(self, func):
        self.produce = func

actual_value = None

AutoScalingGroup(self, "Group",
    desired_capacity=Lazy.number_value(Producer(lambda context: actual_value))
)

# At some later point
actual_value = 10
```

Java

```
double actualValue = 0;

class ProduceActualValue implements INumberProducer {

    @Override
    public Number produce(IResolveContext context) {
        return actualValue;
    }
}

AutoScalingGroup.Builder.create(this, "Group")
    .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point
actualValue = 10;
```

C#

```
public class NumberProducer : INumberProducer
{
    Func<Double> function;
```

```
    public NumberProducer(Func<Double> function)
    {
        this.function = function;
    }

    public Double Produce(IResolveContext context)
    {
        return function();
    }
}

double actualValue = 0;

new AutoScalingGroup(this, "Group", new AutoScalingGroupProps
{
    DesiredCapacity = Lazy.NumberValue(new NumberProducer(() => actualValue))
});

// At some later point
actualValue = 10;
```

转换为 JSON

有时你想生成一个包含任意数据的 JSON 字符串，但你可能不知道这些数据是否包含标记。要对任何数据结构进行正确的 JSON 编码，无论其是否包含标记，都要使用方法堆栈。 toJsonString，如以下示例所示。

TypeScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

JavaScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

Python

```
stack = Stack.of(self)
string = stack.to_json_string(dict(value=bucket.bucket_name))
```

Java

```
Stack stack = Stack.of(this);
String stringVal = stack.toJsonString(java.util.Map.of(    // Map.of requires Java
 9+
        put("value", bucket.getBucketName())));
```

C#

```
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{
    ["value"] = bucket.BucketName
});
```

参数

参数是在部署时提供的自定义值。参数是的功能 AWS CloudFormation。由于 AWS Cloud Development Kit (AWS CDK) 综合了 AWS CloudFormation 模板，因此它还支持部署时间参数。

主题

- 关于参数
- 定义参数
- 使用参数
- 使用参数部署

关于参数

使用可以定义参数，然后可以在您创建的构造的属性中使用这些参数。 AWS CDK您也可以部署包含参数的堆栈。

使用 AWS CDK Toolkit 部署 AWS CloudFormation 模板时，您需要在命令行上提供参数值。如果您通过 AWS CloudFormation 控制台部署模板，则系统会提示您输入参数值。

一般而言，我们建议不要在中使用 AWS CloudFormation 参数 AWS CDK。将值传递到 AWS CDK 应用程序的常用方法是上下文值和环境变量。由于参数值在合成时不可用，因此在 CDK 应用程序中无法轻易地将其用于流量控制和其他用途。

> **Note**
>
> 要使用参数进行控制流，你可以使用`CfnCondition`构造，尽管与原生语`if`句相比，这很尴尬。

使用参数需要注意所编写的代码在部署时和综合时的行为。这使得你的 AWS CDK 申请更难理解和推理，在许多情况下，收效甚微。

通常，最好让你的 CDK 应用程序以明确定义的方式接受必要的信息，然后直接使用这些信息在 CDK 应用程序中声明构造。理想的 AWS CDK生成 AWS CloudFormation 模板是具体的，部署时无需指定任何值。

但是，在某些用例中， AWS CloudFormation 参数是独一无二的。例如，如果您有不同的团队来定义和部署基础架构，则可以使用参数使生成的模板更广泛地发挥作用。此外，由于 AWS CDK 支持 AWS CloudFormation 参数，您可以将 AWS CDK 与使用 AWS CloudFormation 模板的 AWS 服务（例如 Service Catalog）一起使用。这些 AWS 服务使用参数来配置正在部署的模板。

定义参数

使用`CfnParameter`类来定义参数。您至少需要为大多数参数指定类型和描述，尽管两者在技术上都是可选的。当提示用户在 AWS CloudFormation 控制台中输入参数值时，就会显示描述。有关可用类型的更多信息，请参阅类型。

> **Note**
>
> 您可以在任何范围内定义参数。但是，我们建议在堆栈级别定义参数，这样在重构代码时它们的逻辑 ID 就不会发生变化。

TypeScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
```

```
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

JavaScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

Python

```
upload_bucket_name = CfnParameter(self, "uploadBucketName", type="String",
    description="The name of the Amazon S3 bucket where uploaded files will be
 stored.")
```

Java

```
CfnParameter uploadBucketName = CfnParameter.Builder.create(this,
 "uploadBucketName")
        .type("String")
        .description("The name of the Amazon S3 bucket where uploaded files will be
 stored")
        .build();
```

C#

```
var uploadBucketName = new CfnParameter(this, "uploadBucketName", new
 CfnParameterProps
{
    Type = "String",
    Description = "The name of the Amazon S3 bucket where uploaded files will be
 stored"
});
```

使用参数

`CfnParameter`实例通过令牌向您的 AWS CDK 应用程序公开其价值。像所有标记一样，参数的标记是在合成时解析的。但是它解析为对 AWS CloudFormation 模板中定义的参数的引用（将在部署时解析），而不是对具体值的引用。

您可以将令牌作为Token类的实例进行检索，也可以使用字符串、字符串列表或数字编码进行检索。您的选择取决于要与参数一起使用的类或方法所需的值类型。

TypeScript

Property	kind of value
`value`	## class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

JavaScript

Property	kind of value
`value`	## class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

Python

Property	kind of value
`value`	## class instance
`value_as_list`	The token represented as a string list
`valu_as_number`	The token represented as a number
`value_as_string`	The token represented as a string

Java

Property	kind of value
`getValue ()`	## class instance
`getValueAs## ()`	The token represented as a string list
`getValueAs## ()`	The token represented as a number
`getValueAs### ()`	The token represented as a string

C#

Property	kind of value
`#`	## class instance
`ValueAsList`	The token represented as a string list
`ValueAsNumber`	The token represented as a number
`ValueAsString`	The token represented as a string

例如，要在`Bucket`定义中使用参数，请执行以下操作：

TypeScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

JavaScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

Python

```
bucket = Bucket(self, "myBucket",
```

```
    bucket_name=upload_bucket_name.value_as_string)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "myBucket")
        .bucketName(uploadBucketName.getValueAsString())
        .build();
```

C#

```
var bucket = new Bucket(this, "myBucket")
{
    BucketName = uploadBucketName.ValueAsString
};
```

使用参数部署

生成的包含参数的模板可以通过 AWS CloudFormation 控制台以通常的方式部署。系统会提示您输入每个参数的值。

AWS CDK 工具包（cdk命令行工具）还支持在部署时指定参数。您可以在命令行上的--parameters标志后面提供这些信息。您可以部署使用uploadBucketName参数的堆栈，如下例所示。

```
cdk deploy MyStack --parameters uploadBucketName=uploadbucket
```

要定义多个参数，请使用多个--parameters标志。

```
cdk deploy MyStack --parameters uploadBucketName=upbucket --parameters
 downloadBucketName=downbucket
```

如果要部署多个堆栈，则可以为每个堆栈的每个参数指定不同的值。为此，请在参数名称前加上堆栈名称和冒号。

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=uploadbucket --
parameters YourStack:uploadBucketName=upbucket
```

默认情况下， AWS CDK 会保留先前部署中的参数值，如果未明确指定，则在后续部署中使用这些值。使用该--no-previous-parameters标志要求指定所有参数。

标记

标签是信息性键值元素，您可以将其添加到应用程序的构造中。 AWS CDK 应用于给定构造的标签也适用于其所有可标记的子构造。标签包含在从您的应用程序合成的 AWS CloudFormation 模板中，并应用于其部署的 AWS 资源。您可以使用标签对资源进行标识和分类，以实现以下目的：

- 简化管理
- 成本分配
- 访问控制
- 您设计的任何其他目的

> **Tip**
>
> 有关如何对 AWS 资源使用标签的更多信息，请参阅AWS 白皮书中的为 AWS 资源添加标签的最佳实践。

主题

使用标签

该`Tags`类包括静态方法`of()`，通过该方法可以向指定构造添加标签或从中删除标签。

- `Tags.of(SCOPE).add()`将新标签应用于给定构造及其所有子构造。
- `Tags.of(SCOPE).remove()`从给定构造及其任何子构造中移除标签，包括子构造可能已应用于自己的标签。

> **Note**
>
> 标记是使用the section called “方面”实现的。Aspects 是一种将操作（例如标记）应用于给定作用域内的所有构造的方法。

以下示例将带有值的标签键应用于构造。

TypeScript

```
Tags.of(myConstruct).add('key', 'value');
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value');
```

Python

```
Tags.of(my_construct).add("key", "value")
```

Java

```
Tags.of(myConstruct).add("key", "value");
```

C#

```
Tags.Of(myConstruct).Add("key", "value");
```

以下示例从构造中删除标签密钥。

TypeScript

```
Tags.of(myConstruct).remove('key');
```

JavaScript

```
Tags.of(myConstruct).remove('key');
```

Python

```
Tags.of(my_construct).remove("key")
```

Java

```
Tags.of(myConstruct).remove("key");
```

C#

```
Tags.Of(myConstruct).Remove("key");
```

如果您使用的是Stage构造，请在Stage级别或更低级别应用标签。标签不会跨Stage界应用。

标记优先级

递归 AWS CDK 应用和删除标签。如果存在冲突，则优先级最高的标记操作获胜。（使用可选priority属性设置优先级。）如果两个操作的优先级相同，则最靠近构造树底部的标记操作获胜。默认情况下，应用标签的优先级为 100（直接添加到 AWS CloudFormation 资源的标签除外，其优先级为 50）。移除标签的默认优先级为 200。

以下内容将优先级为 300 的标签应用于构造。

TypeScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

Python

```
Tags.of(my_construct).add("key", "value", priority=300)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
        .priority(300).build());
```

C#

```
Tags.Of(myConstruct).Add("key", "value", new TagProps { Priority = 300 });
```

可选属性

标签支持properties微调标签应用于资源或从资源中删除标签的方式。所有其他属性均为可选。

`applyToLaunchedInstances`（Python:`apply_to_launched_instances`）

仅适用于 add ()。默认情况下，标签应用于在 Auto Scaling 组中启动的实例。将此属性设置为 false 可忽略在 Auto Scaling 组中启动的实例。

`includeResourceTypes`/`excludeResourceTypes`(Python:`include_resource_types`/`exclude_res`

使用它们可以根据资源类型仅对资源子集操作标签。 AWS CloudFormation 默认情况下，该操作应用于构造子树中的所有资源，但可以通过包含或排除某些资源类型来更改此值。如果同时指定了两者，则排除优先于包含。

`priority`

使用它来设置此操作相对于其他`Tags.add()`和`Tags.remove()`操作的优先级。较高的值优先于较低的值。添加操作的默认值为 100（直接应用于 AWS CloudFormation 资源的标签为 50），移除操作的默认值为 200。

以下示例将值为值且优先级为 100 的标签 tagnam e 应用于构造AWS::Xxx::Yyy中类型的资源。它不会将标签应用于在 Amazon EC2 Auto Scaling 组中启动的实例或该类型的资源AWS::Xxx::Zzz。（它们是两种任意但不同的 AWS CloudFormation 资源类型的占位符。）

TypeScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
```

```
  priority: 100,
});
```

JavaScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 100
});
```

Python

```
Tags.of(my_construct).add("tagname", "value",
    apply_to_launched_instances=False,
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=100)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
                .applyToLaunchedInstances(false)
                .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
                .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
                .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Add("tagname", "value", new TagProps
{
    ApplyToLaunchedInstances = false,
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

以下示例从构造中类型的资源AWS::Xxx::Yyy中删除优先级为 200 的标签 tagnam e，但未从类型的AWS::Xxx::Zzz资源中移除。

TypeScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200,
});
```

JavaScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200
});
```

Python

```
Tags.of(my_construct).remove("tagname",
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=200,)
```

Java

```
Tags.of((myConstruct).remove("tagname", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
        .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
        .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Remove("tagname", new TagProps
{
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

示例

以下示例将StackType带有值TheBest的标签键添加到在Stack命名中创建的任何资源中MarketingSystem。然后，它会再次将其从除 Amazon EC2 VPC 子网之外的所有资源中删除。结果是只有子网应用了标记。

TypeScript

```
import { App, Stack, Tags } from 'aws-cdk-lib';

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

JavaScript

```
const { App, Stack, Tags } = require('aws-cdk-lib');

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

Python

```
from aws_cdk import App, Stack, Tags

app = App();
the_best_stack = Stack(app, 'MarketingSystem')
```

```
# Add a tag to all constructs in the stack
Tags.of(the_best_stack).add("StackType", "TheBest")

# Remove the tag from all resources except subnet resources
Tags.of(the_best_stack).remove("StackType",
    exclude_resource_types=["AWS::EC2::Subnet"])
```

Java

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Tags;

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove("StackType", TagProps.builder()
        .excludeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
using Amazon.CDK;

var app = new App();
var theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.Of(theBestStack).Add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.Of(theBestStack).Remove("StackType", new TagProps
{
    ExcludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

以下代码实现了相同的结果。考虑哪种方法（包含或排除）可以使您的意图更加明确。

TypeScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
```

```
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

JavaScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

Python

```
Tags.of(the_best_stack).add("StackType", "TheBest",
    include_resource_types=["AWS::EC2::Subnet"])
```

Java

```
Tags.of(theBestStack).add("StackType", "TheBest", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
Tags.Of(theBestStack).Add("StackType", "TheBest", new TagProps {
    IncludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

标记单个构造

`Tags.of(scope).add(key, value)`是向中的构造添加标签的标准方法。 AWS CDK它的树木漫步行为以递归方式标记给定范围内的所有可标记资源，几乎总是你想要的。但是，有时你需要标记一个特定的任意构造（或多个构造）。

其中一种情况涉及应用标签，其值来自被标记的构造的某个属性。标准标记方法以递归方式将相同的键和值应用于作用域内所有匹配的资源。但是，这里每个标记构造的值可能不同。

标签是使用方面实现的，CDK 在您使用的指定范围内为每个构造调用`Tags.of(scope)`标签`visit()`的方法。我们可以`Tag.visit()`直接调用将标签应用于单个构造。

TypeScript

```
new cdk.Tag(key, value).visit(scope);
```

JavaScript

```
new cdk.Tag(key, value).visit(scope);
```

Python

```
cdk.Tag(key, value).visit(scope)
```

Java

```
Tag.Builder.create(key, value).build().visit(scope);
```

C#

```
new Tag(key, value).Visit(scope);
```

你可以标记作用域下的所有构造，但让标签的值从每个构造的属性中派生。为此，请编写一个方面并在该方面的`visit()`方法中应用标签，如前面的示例所示。然后，使用将宽高比添加到所需的范围`Aspects.of(scope).add(aspect)`。

以下示例将标签应用于堆栈中包含资源路径的每个资源。

TypeScript

```
class PathTagger implements cdk.IAspect {
  visit(node: IConstruct) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

JavaScript

```
class PathTagger {
  visit(node) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
```

```
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

Python

```
@jsii.implements(cdk.IAspect)
class PathTagger:
    def visit(self, node: IConstruct):
        cdk.Tag("aws-cdk-path", node.node.path).visit(node)

stack = MyStack(app)
cdk.Aspects.of(stack).add(PathTagger())
```

Java

```
final class PathTagger implements IAspect {
 public void visit(IConstruct node) {
  Tag.Builder.create("aws-cdk-path", node.getNode().getPath()).build().visit(node);
 }
}

stack stack = new MyStack(app);
Aspects.of(stack).add(new PathTagger());
```

C#

```
public class PathTagger : IAspect
{
    public void Visit(IConstruct node)
    {
        new Tag("aws-cdk-path", node.Node.Path).Visit(node);
    }
}

var stack = new MyStack(app);
Aspects.Of(stack).Add(new PathTagger);
```

> **Tip**
>
> 类中内置了条件标记的逻辑，包括优先Tag级、资源类型等。在对任意资源应用标签时，您可以使用这些功能；如果不满足条件，则不会应用标签。此外，该Tag类仅标记可标记的资源，因此在应用标签之前，您无需测试构造是否可标记。

资产

资产是可以捆绑到 AWS CDK 库和应用程序中的本地文件、目录或 Docker 镜像。例如，资产可能是包含 AWS Lambda 函数处理程序代码的目录。资产可以代表应用程序需要操作的任何工件。

以下教程视频全面概述了 CDK 资产，并说明了如何在基础设施即代码 (IaC) 中使用它们。

CDK 资产详解

您可以通过由特定 AWS 结构公开的 API 添加资产。例如，在定义 Lambda.Function 构造时，代码属性允许您传递资产（目录）。 `Function`使用 assets 来捆绑目录的内容并将其用于函数的代码。同样，ecs。 ContainerImage.fromasset 在定义 Amazon ECS 任务定义时使用从本地目录构建的 Docker 镜像。

详细资产

当您在应用程序中引用资产时，从您的应用程序中合成的云程序集包含带有 AWS CDK CLI 说明的元数据信息。说明包括在本地磁盘上的何处找到资产，以及根据资产类型（例如要压缩的目录 (zip) 或要构建的 Docker 映像）执行哪种类型的捆绑操作。

AWS CDK 生成资产的源哈希值。这可以在施工时用来确定资产的内容是否发生了变化。

默认情况下，会在源哈希下 AWS CDK 创建云装配目录中的资源副本`cdk.out`，默认为该副本。这样，云程序集是独立的，因此，如果将其移至其他主机进行部署，它仍然可以部署。有关详细信息，请参阅 the section called “云端程序集”。

AWS CDK 部署引用资产的应用程序（直接通过应用程序代码或通过库）时， AWS CDK CLI 会首先准备这些资产并将其发布到 Amazon S3 存储桶或 Amazon ECR 存储库。（S3 存储桶或存储库是在引导过程中创建的。）只有这样，才会部署堆栈中定义的资源。

本节介绍框架中可用的低级 API。

资产类型

AWS CDK 支持以下类型的资产：

亚马逊 S3 资产

这些是 AWS CDK 上传到 Amazon S3 的本地文件和目录。

Docker 映像

这些是 AWS CDK 上传到亚马逊 ECR 的 Docker 镜像。

以下各节将对这些资产类型进行说明。

亚马逊 S3 资产

您可以将本地文件和目录定义为资产，然后通过 aw s-s3-assets 模块将它们 AWS CDK 打包并上传到 A mazon S3。

以下示例定义了本地目录资产和文件资产。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
```

```
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

Python

```
import os.path
dirname = os.path.dirname(__file__)

from aws_cdk.aws_s3_assets import Asset

# Archived and uploaded to Amazon S3 as a .zip file
directory_asset = Asset(self, "SampleZippedDirAsset",
  path=os.path.join(dirname, "sample-asset-directory")
)

# Uploaded to Amazon S3 as-is
file_asset = Asset(self, 'SampleSingleFileAsset',
  path=os.path.join(dirname, 'file-asset.txt')
)
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.s3.assets.Asset;

// Directory where app was started
File startDir = new File(System.getProperty("user.dir"));

// Archived and uploaded to Amazon S3 as a .zip file
Asset directoryAsset = Asset.Builder.create(this, "SampleZippedDirAsset")
                .path(new File(startDir, "sample-asset-
directory").toString()).build();

// Uploaded to Amazon S3 as-is
Asset fileAsset = Asset.Builder.create(this, "SampleSingleFileAsset")
                .path(new File(startDir, "file-asset.txt").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.S3.Assets;

// Archived and uploaded to Amazon S3 as a .zip file
var directoryAsset = new Asset(this, "SampleZippedDirAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
var fileAsset = new Asset(this, "SampleSingleFileAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "file-asset.txt")
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

awss3assets.NewAsset(stack, jsii.String("SampleZippedDirAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "sample-asset-directory")),
})

awss3assets.NewAsset(stack, jsii.String("SampleSingleFileAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "file-asset.txt")),
})
```

在大多数情况下，您无需直接使用`aws-s3-assets`模块中的 API。支持资产（例如`aws-lambda`）的模块具有便捷方法，因此您可以使用资产。对于 Lambda 函数，fromAsSet () 静态方法允许您在本地文件系统中指定目录或.zip 文件。

Lambda 函数示例

一个常见的用例是使用处理程序代码作为 Amazon S3 资产创建 Lambda 函数。

以下示例使用 Amazon S3 资产在本地目录中定义 Python 处理程序`handler`。它还创建了一个以本地目录资产为属性的 Lambda 函数。`code`以下是该处理程序的 Python 代码。

```
def lambda_handler(event, context):
  message = 'Hello World!'
  return {
    'message': message
  }
```

主 AWS CDK 应用程序的代码应如下所示。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Constructs } from 'constructs';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as path from 'path';

export class HelloAssetStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const path = require('path');

class HelloAssetStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
```

```
      handler: 'index.lambda_handler'
    });
  }
}

module.exports = { HelloAssetStack }
```

Python

```
from aws_cdk import Stack
from constructs import Construct
from aws_cdk import aws_lambda as lambda_

import os.path
dirname = os.path.dirname(__file__)

class HelloAssetStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        lambda_.Function(self, 'myLambdaFunction',
            code=lambda_.Code.from_asset(os.path.join(dirname, 'handler')),
            runtime=lambda_.Runtime.PYTHON_3_6,
            handler="index.lambda_handler")
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class HelloAssetStack extends Stack {

    public HelloAssetStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public HelloAssetStack(final App scope, final String id, final StackProps props)
 {
        super(scope, id, props);
```

```
        File startDir = new File(System.getProperty("user.dir"));

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler").build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using System.IO;

public class HelloAssetStack : Stack
{
    public HelloAssetStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        new Function(this, "myLambdaFunction", new FunctionProps
        {
            Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(),
 "handler")),
            Runtime = Runtime.PYTHON_3_6,
            Handler = "index.lambda_handler"
        });
    }
}
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)
```

```
func HelloAssetStack(scope constructs.Construct, id string, props
 *HelloAssetStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  dirName, err := os.Getwd()
  if err != nil {
    panic(err)
  }

  awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
    Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler")),
 &awss3assets.AssetOptions{}),
    Runtime: awslambda.Runtime_PYTHON_3_6(),
    Handler: jsii.String("index.lambda_handler"),
  })

  return stack
}
```

该Function方法使用资源来捆绑目录的内容，并将其用于函数的代码。

> **Tip**
>
> Java .jar 文件是具有不同扩展名的 ZIP 文件。这些文件按原样上传到 Amazon S3，但是当作为 Lambda 函数部署时，它们包含的文件会被提取，这可能是你不想要的。为避免这种情况，请将.jar文件放在一个目录中，然后将该目录指定为资产。

部署时属性示例

Amazon S3 资产类型还公开了可在 AWS CDK 库和应用程序中引用的部署时属性。C AWS CDK LI 命令cdk synth将资产属性显示为 AWS CloudFormation 参数。

以下示例使用部署时属性将图像资产的位置作为环境变量传递到 Lambda 函数。（文件类型无关紧要；此处使用的 PNG 图像只是一个示例。）

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

Python

```
import os.path
```

```
import aws_cdk.aws_lambda as lambda_
from aws_cdk.aws_s3_assets import Asset

dirname = os.path.dirname(__file__)

image_asset = Asset(self, "SampleAsset",
    path=os.path.join(dirname, "images/my-image.png"))

lambda_.Function(self, "myLambdaFunction",
    code=lambda_.Code.asset(os.path.join(dirname, "handler")),
    runtime=lambda_.Runtime.PYTHON_3_6,
    handler="index.lambda_handler",
    environment=dict(
        S3_BUCKET_NAME=image_asset.s3_bucket_name,
        S3_OBJECT_KEY=image_asset.s3_object_key,
        S3_OBJECT_URL=image_asset.s3_object_url))
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.assets.Asset;

public class FunctionStack extends Stack {
    public FunctionStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset imageAsset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build())

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler")
                .environment(java.util.Map.of(    // Java 9 or later
                    "S3_BUCKET_NAME", imageAsset.getS3BucketName(),
```

```
                    "S3_OBJECT_KEY", imageAsset.getS3ObjectKey(),
                    "S3_OBJECT_URL", imageAsset.getS3ObjectUrl()))
                .build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png")
});

new Function(this, "myLambdaFunction", new FunctionProps
{
    Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), "handler")),
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionary<string, string>
    {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_OBJECT_URL"] = imageAsset.S3ObjectUrl
    }
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)
```

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

imageAsset := awss3assets.NewAsset(stack, jsii.String("SampleAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "images/my-image.png")),
})

awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
  Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler"))),
  Runtime: awslambda.Runtime_PYTHON_3_6(),
  Handler: jsii.String("index.lambda_handler"),
  Environment: &map[string]*string{
    "S3_BUCKET_NAME": imageAsset.S3BucketName(),
    "S3_OBJECT_KEY": imageAsset.S3ObjectKey(),
    "S3_URL": imageAsset.S3ObjectUrl(),
  },
})
```

权限

如果您直接通过 aws-s3-assets 模块、IAM 角色、用户或群组使用 Amazon S3 资产，并且需要在运行时读取资产，请通过 asset.grantRead 方法向这些资产授予 IAM 权限。

以下示例向 IAM 群组授予文件资产的读取权限。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

Python

```
from aws_cdk.aws_s3_assets import Asset
import aws_cdk.aws_iam as iam

import os.path
dirname = os.path.dirname(__file__)

        asset = Asset(self, "MyFile",
            path=os.path.join(dirname, "my-image.png"))

        group = iam.Group(self, "MyUserGroup")
        asset.grant_read(group)
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset asset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build();
```

```
        Group group = new Group(this, "MyUserGroup");
        asset.grantRead(group);   }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @"images\my-
image.png"))
});

var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsiam"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awss3assets.NewAsset(stack, jsii.String("MyFile"), &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "my-image.png")),
})

group := awsiam.NewGroup(stack, jsii.String("MyUserGroup"), &awsiam.GroupProps{})

asset.GrantRead(group)
```

Docker 镜像资产

AWS CDK 支持通过模块将本地 Docker 镜像捆绑为资产。aws-ecr-assets

以下示例定义了在本地构建并推送到 Amazon ECR 的 Docker 镜像。镜像从本地 Docker 上下文目录（带有 Dockerfile）构建，并通过 CL AWS CDK I 或应用程序的 CI/CD 管道上传到 Amazon ECR。这些图像可以在您的 AWS CDK 应用程序中自然引用。

TypeScript

```
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

Python

```
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))
```

Java

```
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
```

```
            .directory(new File(startDir, "my-image").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

该`my-image`目录必须包含一个 Dockerfile。 AWS CDK CLI 从构建 Docker 映像`my-image`，将其推送到 Amazon ECR 存储库，并将存储库的名称指定为堆栈的 AWS CloudFormation 参数。Docker 镜像资产类型公开了可以在 AWS CDK 库和应用程序中引用的部署时属性。C AWS CDK LI 命令`cdk synth`将资产属性显示为 AWS CloudFormation 参数。

亚马逊 ECS 任务定义示例

一个常见的用例是创建一台 Amazon 弹性云服务器TaskDefinition来运行 Docker 容器。以下示例指定了在本地 AWS CDK 构建并推送到 Amazon ECR 的 Docker 映像资产的位置。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecr_assets from 'aws-cdk-lib/aws-ecr-assets';
import * as path from 'path';

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const ecr_assets = require('aws-cdk-lib/aws-ecr-assets');
const path = require('path');

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

Python

```
import aws_cdk.aws_ecs as ecs
import aws_cdk.aws_ecr_assets as ecr_assets
```

```
import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

asset = ecr_assets.DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_docker_image_asset(asset))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder()
            .image(ContainerImage.fromDockerImageAsset(asset))
            .build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.Ecr.Assets;
```

```
var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromDockerImageAsset(asset)
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

taskDefinition := awsecs.NewTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.TaskDefinitionProps{
  MemoryMiB: jsii.String("1024"),
  Cpu: jsii.String("512"),
})

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

```
taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromDockerImageAsset(asset),
})
```

部署时属性示例

以下示例说明如何使用部署时间属性`repository`以及`imageUri`如何使用 AWS Fargate 启动类型创建 Amazon ECS 任务定义。请注意，Amazon ECR 存储库查询需要图片的标签，而不是其 URI，因此我们从资产 URI 的末尾截取它。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});
```

```
const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

Python

```
import aws_cdk.aws_ecs as ecs
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'my-image',
    directory=os.path.join(dirname, "..", "demo-image"))

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024, cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_ecr_repository(
        asset.repository, asset.image_uri.rpartition(":")[-1]))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
            .directory(new File(startDir, "demo-image").toString()).build();
```

```
FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset's image URI for use in ECR repo lookup
String imageUri = asset.getImageUri();
String imageTag = imageUri.substring(imageUri.lastIndexOf(":") + 1);

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
                asset.getRepository(), imageTag)).build());
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "demo-image")
});

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromEcrRepository(asset.Repository,
 asset.ImageUri.Split(":").Last())
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
```

```
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "demo-image")),
})

taskDefinition := awsecs.NewFargateTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.FargateTaskDefinitionProps{
  MemoryLimitMiB: jsii.Number(1024),
  Cpu: jsii.Number(512),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromEcrRepository(asset.Repository(),
 asset.ImageTag()),
})
```

生成参数示例

当 AWS CDK CLI 在部署期间构建映像时，您可以通过 buildArgs (Python:build_args) 属性选项为 Docker 构建步骤提供自定义的构建参数。

TypeScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

JavaScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
```

```
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

Python

```
asset = DockerImageAsset(self, "MyBulidImage",
    directory=os.path.join(dirname, "my-image"),
    build_args=dict(HTTP_PROXY="http://10.20.30.2:1234"))
```

Java

```
DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image"),
            .directory(new File(startDir, "my-image").toString())
            .buildArgs(java.util.Map.of(    // Java 9 or later
                "HTTP_PROXY", "http://10.20.30.2:1234"))
            .build();
```

C#

```
var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image"),
    BuildArgs = new Dictionary<string, string>
    {
        ["HTTP_PROXY"] = "http://10.20.30.2:1234"
    }
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
  BuildArgs: &map[string]*string{
    "HTTP_PROXY": jsii.String("http://10.20.30.2:1234"),
  },
```

```
})
```

权限

如果您使用支持 Docker 镜像资源的模块（例如 aws-ecs），则在您直接或通过使用资源时，它会为您 AWS CDK 管理权限。ContainerImage fromEcrRepository(Python:`from_ecr_repository`)。如果您直接使用 Docker 镜像资产，请确保使用者拥有拉取镜像的权限。

在大多数情况下，你应该使用 asset.repository.gran tPull 方法 (Python:. `grant_pull` 这会修改委托人的 IAM 策略，使其能够从此存储库中提取图像。 如果拉取图片的委托人不在同一个账户中，或者该 AWS 服务不在你的账户中扮演角色（例如 AWS CodeBuild），则必须授予对资源策略而不是委托人策略的拉取权限。 使用 asset.repository。 addToResource授予相应委托人权限的策略方法 (Python:`add_to_resource_policy`)。

AWS CloudFormation 资源元数据

> **Note**
>
> 本节仅与构造作者相关。在某些情况下，工具需要知道某个 CFN 资源正在使用本地资产。例如，您可以使用 AWS SAM CLI 在本地调用 Lambda 函数以进行调试。有关详细信息，请参阅 the section called “AWS SAM 整合”。

要启用此类用例，外部工具会查阅 AWS CloudFormation 资源上的一组元数据条目：

- `aws:asset:path`— 指向资产的本地路径。
- `aws:asset:property`-使用资产的资源属性的名称。

使用这两个元数据条目，工具可以识别资产是否由特定资源使用，并启用高级本地体验。

要将这些元数据条目添加到资源中，请使用 `asset.addResourceMetadata` (Python:`add_resource_metadata`) 方法。

权限

C AWS onstruct Library 使用一些常见的、广泛实施的习语来管理访问和权限。IAM 模块为您提供使用这些习语所需的工具。

AWS CDK AWS CloudFormation 用于部署更改。每个部署都涉及启动 AWS CloudFormation 部署的参与者（开发人员或自动化系统）。在此过程中，参与者将假设一个或多个 IAM 身份（用户或角色），并可选择将角色传递给 AWS CloudFormation。

如果您使用用户身份 AWS IAM Identity Center 进行身份验证，则单点登录提供商会提供短期会话证书，授权您充当预定义的 IAM 角色。要了解如何通过 IAM Identity Center 身份验证 AWS CDK 获取 AWS 证书，请参阅软件开发工具包和AWS 工具参考指南中的了解 IAM 身份中心身份验证。

主体

IAM 委托人是一个经过身份验证的 AWS 实体，代表可以调用 AWS API 的用户、服务或应用程序。C AWS onstruct Library 支持以多种灵活的方式指定委托人，以授予他们访问您的 AWS 资源的权限。

在安全环境中，“委托人” 一词专门指经过身份验证的实体，例如用户。群组和角色之类的对象并不代表用户（以及其他经过身份验证的实体），而是为了授予权限而间接标识他们。

例如，如果您创建一个 IAM 群组，则可以向该群组（以及其成员）授予对 Amazon RDS 表的写入权限。但是，群组本身不是委托人，因为它不代表单个实体（而且，您无法登录群组）。

在 CDK 的 IAM 库中，直接或间接标识委托人的类实现了接IPrincipal口，从而允许这些对象在访问策略中互换使用。但是，从安全意义上讲，并非所有人都是主体。这些对象包括：

1. IAM 资源Role，例如User、和 Group
2. 服务负责人 () `new iam.ServicePrincipal('service.amazonaws.com')`
3. 联邦校长 () `new iam.FederatedPrincipal('cognito-identity.amazonaws.com')`
4. 账户委托人 (`new iam.AccountPrincipal('0123456789012')`)
5. 规范用户主体 () `new iam.CanonicalUserPrincipal('79a59d[...]7ef2be')`
6. AWS Organizations 校长 () `new iam.OrganizationPrincipal('org-id')`
7. 任意 ARN 主体 () `new iam.ArnPrincipal(res.arn)`
8. 而且`iam.CompositePrincipal(principal1, principal2, ...)`要信任多个委托人

授权

每个表示可以访问的资源（例如 Amazon S3 存储桶或 Amazon DynamoDB 表）的结构都有向其他实体授予访问权限的方法。所有这些方法的名称都以 g rant 开头。

例如，Amazon S3 存储桶具有方法grantRead和 grantReadWrite (Python:grant_read,grant_read_write)，用于分别启用实体对存储桶的读取和读/写访问权限。该实体不必确切知道执行这些操作需要哪些 Amazon S3 IAM 权限。

授予方法的第一个参数始终为 igran Table 类型。此接口表示可以被授予权限的实体。也就是说，它表示具有角色的资源，例如 IAM 对象RoleUser、和Group。

也可以向其他实体授予权限。例如，在本主题的后面部分，我们将介绍如何向 CodeBuild项目授予对 Amazon S3 存储桶的访问权限。通常，关联角色是通过被授予访问权限的实体的role属性获得的。

使用执行角色的资源（例如）也会实现 lambda.FunctionIGrantable，因此您可以直接向他们授予访问权限，而不必向其角色授予访问权限。例如，如果bucket是 Amazon S3 存储桶，并且function是 Lambda 函数，则以下代码授予该函数对该存储桶的读取权限。

TypeScript

```
bucket.grantRead(function);
```

JavaScript

```
bucket.grantRead(function);
```

Python

```
bucket.grant_read(function)
```

Java

```
bucket.grantRead(function);
```

C#

```
bucket.GrantRead(function);
```

有时，必须在部署堆栈时应用权限。其中一种情况是您向 AWS CloudFormation 自定义资源授予对其他资源的访问权限。自定义资源将在部署期间被调用，因此它在部署时必须具有指定的权限。

另一种情况是，当服务验证您传递给它的角色是否应用了正确的策略时。（许多 AWS 服务机构这样做是为了确保您不会忘记设置策略。）在这种情况下，如果权限应用得太晚，部署可能会失败。

要强制在创建其他资源之前应用授予的权限，可以添加对授权本身的依赖关系，如下所示。尽管通常会丢弃授予方法的返回值，但实际上每个授权方法都会返回一个`iam.Grant`对象。

TypeScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

JavaScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

Python

```
grant = bucket.grant_read(function)
custom = CustomResource(...)
custom.node.add_dependency(grant)
```

Java

```
Grant grant = bucket.grantRead(function);
CustomResource custom = new CustomResource(...);
custom.node.addDependency(grant);
```

C#

```
var grant = bucket.GrantRead(function);
var custom = new CustomResource(...);
custom.node.AddDependency(grant);
```

角色

IAM 软件包包含一个代表 IAM 角色的`Role`结构。以下代码创建了一个信任 Amazon EC2 服务的新角色。

TypeScript

```
import * as iam from 'aws-cdk-lib/aws-iam';

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

JavaScript

```
const iam = require('aws-cdk-lib/aws-iam');

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com')   // required
});
```

Python

```
import aws_cdk.aws_iam as iam

role = iam.Role(self, "Role",
          assumed_by=iam.ServicePrincipal("ec2.amazonaws.com")) # required
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.iam.ServicePrincipal;

Role role = Role.Builder.create(this, "Role")
        .assumedBy(new ServicePrincipal("ec2.amazonaws.com")).build();
```

C#

```
using Amazon.CDK.AWS.IAM;

var role = new Role(this, "Role", new RoleProps
{
    AssumedBy = new ServicePrincipal("ec2.amazonaws.com"),   // required
});
```

您可以通过调用角色的方法 (Python:`add_to_policy`)，传入定义要添加的规则的[addToPolicyPolicyStatement](#)方法来为该角色添加权限。该语句将添加到角色的默认策略中；如果没有，则创建一个。

以下示例在授权服务为条件下，向角色添加操作`ec2:SomeAction`和`s3:AnotherAction`资源`bucket`和 `otherRole` (Python:`other_role`) 的`Deny`策略声明 AWS CodeBuild。

TypeScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com',
}}}));
```

JavaScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com'
}}}));
```

Python

```
role.add_to_policy(iam.PolicyStatement(
    effect=iam.Effect.DENY,
    resources=[bucket.bucket_arn, other_role.role_arn],
    actions=["ec2:SomeAction", "s3:AnotherAction"],
    conditions={"StringEquals": {
        "ec2:AuthorizedService": "codebuild.amazonaws.com"}}
))
```

Java

```
role.addToPolicy(PolicyStatement.Builder.create()
        .effect(Effect.DENY)
        .resources(Arrays.asList(bucket.getBucketArn(), otherRole.getRoleArn()))
```

```
        .actions(Arrays.asList("ec2:SomeAction", "s3:AnotherAction"))
        .conditions(java.util.Map.of(    // Map.of requires Java 9 or later
            "StringEquals", java.util.Map.of(
                "ec2:AuthorizedService", "codebuild.amazonaws.com")))
        .build());
```

C#

```
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.DENY,
    Resources = new string[] { bucket.BucketArn, otherRole.RoleArn },
    Actions = new string[] { "ec2:SomeAction", "s3:AnotherAction" },
    Conditions = new Dictionary<string, object>
    {
        ["StringEquals"] = new Dictionary<string, string>
        {
            ["ec2:AuthorizedService"] = "codebuild.amazonaws.com"
        }
    }
}));
```

在前面的示例中，我们使用 addToPolicy (Python:add_to_policy) 调用创建了一个新的PolicyStatement内联函数。您也可以传入现有的政策声明或您修改过的政策声明。该PolicyStatement对象有多种添加委托人、资源、条件和操作的方法。

如果您使用的构造需要角色才能正常运行，则可以执行以下操作之一：

- 在实例化构造对象时传入现有角色。
- 让构造为你创建一个新角色，信任相应的服务主体。以下示例使用这样的构造： CodeBuild 项目。

TypeScript

```
import * as codebuild from 'aws-cdk-lib/aws-codebuild';

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole: iam.IRole | undefined = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
```

```
  // trusting the codebuild.amazonaws.com service principal
  role: someRole,
});
```

JavaScript

```
const codebuild = require('aws-cdk-lib/aws-codebuild');

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole
});
```

Python

```
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
# if role is None, the Project creates a new default role,
# trusting the codebuild.amazonaws.com service principal
role=some_role)
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
```

```
        .role(someRole).build();
```

C#

```
using Amazon.CDK.AWS.CodeBuild;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
var someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
var project = new Project(this, "Project", new ProjectProps
{
    Role = someRole
});
```

创建对象后，角色（无论是传入的角色还是构造创建的默认角色）即可作为属性使用`role`。但是，此属性在外部资源上不可用。因此，这些构造有一个 `addToRolePolicy` (Python:`add_to_role_policy`) 方法。

如果构造是外部资源，则该方法不执行任何操作，否则它会调用该`role`属性的 `addToPolicy` (Python:`add_to_policy`) 方法。这为您省去了显式处理未定义案例的麻烦。

以下示例演示：

TypeScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,   // ... and so on defining the policy
}));
```

JavaScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');
```

```
// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW   // ... and so on defining the policy
}));
```

Python

```
# project is imported into the CDK application
project = codebuild.Project.from_project_name(self, 'Project', 'ProjectName')

# project is imported, so project.role is undefined, and this call has no effect
project.add_to_role_policy(iam.PolicyStatement(
  effect=iam.Effect.ALLOW,   # ... and so on defining the policy
)
```

Java

```
// project is imported into the CDK application
Project project = Project.fromProjectName(this, "Project", "ProjectName");

// project is imported, so project.getRole() is null, and this call has no effect
project.addToRolePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)   // .. and so on defining the policy
        .build());
```

C#

```
// project is imported into the CDK application
var project = Project.FromProjectName(this, "Project", "ProjectName");

// project is imported, so project.role is null, and this call has no effect
project.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW, // ... and so on defining the policy
}));
```

资源策略

中的 AWS一些资源（例如 Amazon S3 存储桶和 IAM 角色）也有资源策略。这些构造有一个addToResourcePolicy方法 (Python:add_to_resource_policy)，它以 a PolicyStatement 作为参数。添加到资源策略的每个策略声明都必须指定至少一个委托人。

在以下示例中，Amazon S3 存储桶向角色bucket授s3:SomeAction予了自身权限。

TypeScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

JavaScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

Python

```
bucket.add_to_resource_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["s3:SomeAction"],
    resources=[bucket.bucket_arn],
    principals=role))
```

Java

```
bucket.addToResourcePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)
        .actions(Arrays.asList("s3:SomeAction"))
        .resources(Arrays.asList(bucket.getBucketArn()))
        .principals(Arrays.asList(role))
        .build());
```

C#

```
bucket.AddToResourcePolicy(new PolicyStatement(new PolicyStatementProps
{
```

```
    Effect = Effect.ALLOW,
    Actions = new string[] { "s3:SomeAction" },
    Resources = new string[] { bucket.BucketArn },
    Principals = new IPrincipal[] { role }
}));
```

使用外部 IAM 对象

如果您在应用程序之外定义了 IAM 用户、委托人、群组或角色，则可以在 AWS CDK AWS CDK 应用程序中使用该 IAM 对象。为此，请使用其 ARN 或名称创建对其的引用。（使用用户、组和角色的名称。）然后，返回的引用可用于授予权限或构造策略声明，如前所述。

- 对于用户，请致电`User.fromUserArn()`或`User.fromUserName()`。`User.fromUserAttributes()`也可用，但目前提供的功能与`User.fromUserArn()`。
- 对于委托人，请实例化一个对象。`ArnPrincipal`
- 对于群组，请致电`Group.fromGroupArn()`或`Group.fromGroupName()`。
- 如需了解角色，请致电`Role.fromRoleArn()`或`Role.fromRoleName()`。

可以使用以下方法以类似的方式使用策略（包括托管策略）。您可以在任何需要 IAM 策略的地方使用对这些对象的引用。

- `Policy.fromPolicyName`
- `ManagedPolicy.fromManagedPolicyArn`
- `ManagedPolicy.fromManagedPolicyName`
- `ManagedPolicy.fromAwsManagedPolicyName`

> **Note**
>
> 与所有对外部 AWS 资源的引用一样，您无法在 CDK 应用程序中修改外部 IAM 对象。

运行时上下文

上下文值是可以与应用程序、堆栈或构造相关联的键值对。它们可以从文件（通常位于您的项目目录中）`cdk.json`或`cdk.context.json`在命令行中提供给您的应用程序。

CDK Toolkit 使用上下文来缓存合成期间从您的 AWS 账户中检索到的值。值包括您账户中的可用区域或 Amazon EC2 实例当前可用的亚马逊系统映像 (AMI) ID。由于这些值是由您的 AWS 账户提供的，因此在您的 CDK 应用程序运行之间，它们可能会发生变化。这使它们成为意外变化的潜在来源。CDK Toolkit 的缓存行为会为您的 CDK 应用程序会 “冻结” 这些值，直到您决定接受新值为止。

想象一下以下没有上下文缓存的场景。假设您指定了 “最新亚马逊 Linux” 作为亚马逊 EC2 实例的 AMI，并且发布了此 AMI 的新版本。然后，下次部署 CDK 堆栈时，已经部署的实例将使用过时（“错误”）的 AMI，需要升级。升级会导致将所有现有实例替换为新实例，这可能是意想不到的，也是不受欢迎的。

相反，CDK 会在您的项目`cdk.context.json`文件中记录您账户的可用 AMI，并将存储的值用于未来的合成操作。这样，AMI 列表就不再是潜在的变更来源。您还可以确保您的堆栈将始终合成相同的 AWS CloudFormation 模板。

并非所有上下文值都是来自您的 AWS 环境的缓存值。 the section called “功能标志”也是上下文值。您也可以创建自己的上下文值以供应用程序或构造使用。

上下文键是字符串。值可以是 JSON 支持的任何类型：数字、字符串、数组或对象。

> **Tip**
>
> 如果您的构造创建了自己的上下文值，请将库的包名称包含在其密钥中，这样它们就不会与其他包的上下文值发生冲突。

许多上下文值都与特定 AWS 环境相关联，并且给定的 CDK 应用程序可以部署在多个环境中。此类值的密钥包括 AWS 账户和区域，这样来自不同环境的值就不会发生冲突。

以下上下文密钥说明了使用的格式 AWS CDK，包括账户和区域。

```
availability-zones:account=123456789012:region=eu-central-1
```

> **Important**
>
> 缓存的上下文值由 AWS CDK 及其构造（包括您可能编写的构造）管理。请勿通过手动编辑文件来添加或更改缓存的上下文值。但是，`cdk.context.json`偶尔查看一下缓存了哪些值可能会很有用。不代表缓存值的上下文值应存储在的`context`键下`cdk.json`。这样，当缓存的值被清除时，它们就不会被清除。

上下文值的来源

上下文值可以通过六种不同的方式提供给您的 AWS CDK 应用程序：

- 自动从当前 AWS 账户中获取。
- 通过cdk命令的--context选项。（这些值始终是字符串。）
- 在项目`cdk.context.json`文件中。
- 在项目`cdk.json`文件的`context`密钥中。
- 在你的`~/.cdk.json`文件`context`密钥里。
- 在您的 AWS CDK 应用程序中使用该`construct.node.setContext()`方法。

项目文件`cdk.context.json`用于 AWS CDK 缓存从您的 AWS 账户中检索到的上下文值。这种做法可以避免在引入新的可用区等情况下对部署进行意外更改。 AWS CDK 不会将上下文数据写入列出的任何其他文件。

> **Important**
>
> 因为它们是应用程序状态的一部分，`cdk.context.json`必须`cdk.json`与应用程序的其余源代码一起提交给源代码控制。否则，在其他环境（例如 CI 管道）中的部署可能会产生不一致的结果。

上下文值的作用域仅限于创建它们的构造；它们对子构造可见，但对父母或兄弟姐妹不可见。由 AWS CDK Toolkit（cdk命令）设置的上下文值可以从文件或--context选项中自动设置。来自这些来源的上下文值是在App构造上隐式设置的。因此，它们对应用程序中每个堆栈中的每个构造都可见。

您的应用程序可以使用`construct.node.tryGetContext`方法读取上下文值。如果在当前构造或其任何父构造中找不到请求的条目，则结果为`undefined`。（或者，结果可能与您的语言相同，例如None在 Python 中。）

上下文方法

AWS CDK 支持多种上下文方法，使 AWS CDK 应用程序能够从 AWS 环境中获取上下文信息。例如，您可以使用 stack. AvailabilityZones 方法获取给定 AWS 账户和区域中可用的可用区域列表。

以下是上下文方法：

HostedZone.fromLooku

获取您账户中的托管区域。

Stack. 可用性区域

获取支持的可用区。

StringParameter.valueFromLookup

从当前区域的 Amazon EC2 Systems Manager 参数存储中获取一个值。

vpc.fromLookup

获取您账户中的现有 Amazon 虚拟私有云。

LookupMachineImage

在 Amazon Virtual Private Cloud 中查找计算机映像以用于 NAT 实例。

如果所需的上下文值不可用，则 AWS CDK 应用程序会通知 CDK Toolkit 缺少上下文信息。接下来，CLI 会向当前 AWS 账户查询信息，并将生成的上下文信息存储在`cdk.context.json`文件中。然后，它使用上下文值再次执行 AWS CDK 应用程序。

查看和管理上下文

使用cdk context命令查看和管理`cdk.context.json`文件中的信息。要查看此信息，请使用不带任何选项的cdk context命令。输出应如下所示。

```
Context found in cdk.json:

##############################################################################################
# # # Key                                                                  # Value
                                     #
##############################################################################################
# 1 # availability-zones:account=123456789012:region=eu-central-1 # [ "eu-central-1a",
 "eu-central-1b", "eu-central-1c" ]   #
##############################################################################################
# 2 # availability-zones:account=123456789012:region=eu-west-1    # [ "eu-west-1a",
 "eu-west-1b", "eu-west-1c" ]            #
##############################################################################################

Run cdk context --reset KEY_OR_NUMBER to remove a context key. If it is a cached value,
 it will be refreshed on the next cdk synth.
```

要删除上下文值，请运行cdk context --reset，指定该值的相应键或数字。以下示例删除了与前面示例中第二个键对应的值。此值表示欧洲（爱尔兰）地区的可用区列表。

```
cdk context --reset 2
```

```
Context value
availability-zones:account=123456789012:region=eu-west-1
reset. It will be refreshed on the next SDK synthesis run.
```

因此，如果您想更新到最新版本的 Amazon Linux AMI，请使用前面的示例对上下文值进行受控更新并将其重置。然后，重新合成并部署您的应用程序。

```
cdk synth
```

要清除应用程序的所有存储上下文值，请按如下方式运行cdk context --clear。

```
cdk context --clear
```

只能重置或清除存储在中的cdk.context.json上下文值。 AWS CDK 不涉及其他上下文值。因此，为了防止使用这些命令重置上下文值，可以将该值复制到cdk.json。

AWS CDK 工具包**--context**标志

在合成或部署期间，使用--context（-c简称）选项将运行时上下文值传递给您的 CDK 应用程序。

```
cdk synth --context key=value MyStack
```

要指定多个上下文值，请多次重复该--context选项，每次提供一个键值对。

```
cdk synth --context key1=value1 --context key2=value2 MyStack
```

合成多个堆栈时，指定的上下文值将传递给所有堆栈。要为单个堆栈提供不同的上下文值，要么对值使用不同的键，要么使用多个cdk synth或cdk deploy命令。

从命令行传递的上下文值始终是字符串。如果值通常是其他类型，则您的代码必须准备好转换或解析该值。您可能以其他方式提供非字符串上下文值（例如，在cdk.context.json）。要确保此类值按预期工作，请在转换之前确认该值是否为字符串。

示例

以下是使用 AWS CDK 上下文使用现有 Amazon VPC 的示例。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class ExistsVpcStack extends cdk.Stack {

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid,
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString(),
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const ec2 = require('aws-cdk-lib/aws-ec2');

class ExistsVpcStack extends cdk.Stack {

  constructor(scope, id, props) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid
```

```
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString()
    });
  }
}

module.exports = { ExistsVpcStack }
```

Python

```
import aws_cdk as cdk
import aws_cdk.aws_ec2 as ec2
from constructs import Construct

class ExistsVpcStack(cdk.Stack):

    def __init__(scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)

        vpcid = self.node.try_get_context("vpcid")
        vpc = ec2.Vpc.from_lookup(self, "VPC", vpc_id=vpcid)

        pubsubnets = vpc.select_subnets(subnetType=ec2.SubnetType.PUBLIC)

        cdk.CfnOutput(self, "publicsubnets",
            value=pubsubnets.subnet_ids.to_string())
```

Java

```
import software.amazon.awscdk.CfnOutput;

import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.ec2.VpcLookupOptions;
import software.amazon.awscdk.services.ec2.SelectedSubnets;
import software.amazon.awscdk.services.ec2.SubnetSelection;
import software.amazon.awscdk.services.ec2.SubnetType;
import software.constructs.Construct;
```

```
public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(Construct context, String id) {
        this(context, id, null);
    }

    public ExistsVpcStack(Construct context, String id, StackProps props) {
        super(context, id, props);

        String vpcId = (String)this.getNode().tryGetContext("vpcid");
        Vpc vpc = (Vpc)Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()
                .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()
                .subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
                .value(pubSubNets.getSubnetIds().toString()).build();

    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.EC2;
using Constructs;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        var vpcId = (string)this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
        {
            VpcId = vpcId
        });

        SelectedSubnets pubSubNets = vpc.SelectSubnets([new SubnetSelection
        {
            SubnetType = SubnetType.PUBLIC
        }]);
```

```
        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
}
```

您可以使用cdk diff来查看在命令行上传递上下文值的效果：

```
cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

```
Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
 {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
```

可以查看生成的上下文值，如下所示。

```
cdk context -j
```

```
{
  "vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-
east-1": {
    "vpcId": "vpc-0cb9c31031d0d3e22",
    "availabilityZones": [
      "us-east-1a",
      "us-east-1b"
    ],
    "privateSubnetIds": [
      "subnet-03ecfc033225be285",
      "subnet-0cded5da53180ebfa"
    ],
    "privateSubnetNames": [
      "Private"
    ],
    "privateSubnetRouteTableIds": [
      "rtb-0e955393ced0ada04",
      "rtb-05602e7b9f310e5b0"
    ],
    "publicSubnetIds": [
      "subnet-06e0ea7dd302d3e8f",
```

```
      "subnet-01fc0acfb58f3128f"
    ],
    "publicSubnetNames": [
      "Public"
    ],
    "publicSubnetRouteTableIds": [
      "rtb-00d1fdfd823c82289",
      "rtb-04bb1969b42969bcb"
    ]
  }
}
```

功能标志

AWS CDK 使用功能标志来启用发行版中可能存在的破坏行为。标志以`cdk.json`（或`~/.cdk.json`）中的the section called “上下文”值形式存储。`cdk context --reset`或`cdk context --clear`命令不会将其删除。

默认情况下，功能标记处于禁用状态。未指定标志的现有项目将在以后的 AWS CDK 版本中像以前一样继续运行。使用`cdk init`启用创建该项目的版本中所有可用功能的标志创建了新项目。编辑`cdk.json`以禁用您更喜欢以前行为的所有标志。您还可以在升级后添加标志以启用新行为 AWS CDK。

所有当前功能标志的列表可以在中的 AWS CDK GitHub 存储库中找到`FEATURE_FLAGS.md`。有关该版本`CHANGELOG`中添加的任何新功能标志的描述，请参阅给定版本中的。

恢复到 v1 行为

在 CDK v2 中，与 v1 相比，某些功能标志的默认值已更改。您可以将它们设置回来以恢复`false`到特定的 AWS CDK v1 行为。使用`cdk diff`命令检查对合成模板的更改，以查看是否需要这些标志中的任何一个。

`@aws-cdk/core:newStyleStackSynthesis`

使用新的堆栈合成方法，该方法假设引导资源具有众所周知的名称。需要现代引导，但反过来又允许通过CDK Pipelines进行CI/CD以及开箱即用的跨账户部署。

`@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId`

如果您的应用程序使用多个 Amazon API Gateway API 密钥并将其与使用计划相关联。

`@aws-cdk/aws-rds:lowercaseDbIdentifier`

如果您的应用程序使用 Amazon RDS 数据库实例或数据库集群，并明确指定这些实例或数据库集群的标识符。

`@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021`

如果您的应用程序在发行版中使用 TLS_V1_2_2019 安全策略。 Amazon CloudFront 默认情况下，CDK v2 使用安全策略 tlsv1.2_2021。

`@aws-cdk/core:stackRelativeExports`

如果您的应用程序使用多个堆栈，而您在另一个堆栈中引用来自一个堆栈的资源，则这将决定使用绝对路径还是相对路径来构造 AWS CloudFormation 导出。

`@aws-cdk/aws-lambda:recognizeVersionProps`

如果设置为`false`，则在检测 Lambda 函数是否已更改时，CDK 会包含元数据。由于不允许使用重复的版本，因此当只有元数据发生更改时，这可能会导致部署失败。如果您对应用程序中的所有 Lambda 函数进行了至少一次更改，则无需恢复此标志。

此处显示了在中恢复这些标志`cdk.json`的语法。

```
{
  "context": {
    "@aws-cdk/core:newStyleStackSynthesis": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021": false,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/core:stackRelativeExports": false,
    "@aws-cdk/aws-lambda:recognizeVersionProps": false
  }
}
```

方面

Aspects 是一种将操作应用于给定作用域内所有构造的方法。该方面可以修改结构，例如通过添加标签。或者它可以验证一些关于构造状态的信息，例如确保所有存储桶都已加密。

要将某个方面应用于同一个作用域内的构造和所有构造，请`Aspects.of(SCOPE).add()`使用新的方面进行调用，如以下示例所示。

TypeScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

JavaScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

Python

```
Aspects.of(my_construct).add(SomeAspect(...))
```

Java

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

C#

```
Aspects.Of(myConstruct).add(new SomeAspect(...));
```

Go

```
awscdk.Aspects_Of(stack).Add(awscdk.NewTag(...))
```

AWS CDK 使用方面来标记资源，但框架也可以用于其他目的。例如，您可以使用它来验证或更改由更高级别的构造为您定义的 AWS CloudFormation 资源。

细节方面

方面采用访客模式。一个方面是实现以下接口的类。

TypeScript

```
interface IAspect {
   visit(node: IConstruct): void;}
```

JavaScript

JavaScript 没有接口作为语言功能。因此，一个方面只是一个类的实例，该类的`visit`方法接受要操作的节点。

Python

Python 没有接口作为语言功能。因此，一个方面只是一个类的实例，该类的`visit`方法接受要操作的节点。

Java

```
public interface IAspect {
    public void visit(Construct node);
}
```

C#

```
public interface IAspect
{
    void Visit(IConstruct node);
}
```

Go

```
type IAspect interface {
  Visit(node constructs.IConstruct)
}
```

当您调用时`Aspects.of(SCOPE).add(...)`，该构造会将该方面添加到内部方面列表中。您可以通过获取列表`Aspects.of(SCOPE)`。

在准备阶段，按自上而下的顺序为构造及其每个子对象 AWS CDK 调用对象`visit`的方法。

该`visit`方法可以自由更改构造中的任何内容。在强类型语言中，在访问特定于构造的属性或方法之前，将接收到的构造转换为更具体的类型。

方面不会跨`Stage`构造边界传播，因为定义后`Stages`是自包含且不可变的。如果你想让`Stage`构造本身（或更低版本）访问构造内部的构造，请将各个方面应用于构造本身（或更低版本）。`Stage`

示例

以下示例验证堆栈中创建的所有存储桶是否都启用了版本控制。该方面为验证失败的构造添加了错误注释。这会导致synth操作失败并阻止部署生成的云组件。

TypeScript

```
class BucketVersioningChecker implements IAspect {
  public visit(node: IConstruct): void {
    // See that we're dealing with a CfnBucket
    if (node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || (!Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

JavaScript

```
class BucketVersioningChecker {
   visit(node) {
    // See that we're dealing with a CfnBucket
    if ( node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || !Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

Python

```
@jsii.implements(cdk.IAspect)
class BucketVersioningChecker:

  def visit(self, node):
    # See that we're dealing with a CfnBucket
    if isinstance(node, s3.CfnBucket):

      # Check for versioning property, exclude the case where the property
      # can be a token (IResolvable).
      if (not node.versioning_configuration or
              not Tokenization.is_resolvable(node.versioning_configuration)
                  and node.versioning_configuration.status != "Enabled"):
          Annotations.of(node).add_error('Bucket versioning is not enabled')

# Later, apply to the stack
Aspects.of(stack).add(BucketVersioningChecker())
```

Java

```
public class BucketVersioningChecker implements IAspect
{
    @Override
    public void visit(Construct node)
    {
        // See that we're dealing with a CfnBucket
        if (node instanceof CfnBucket)
        {
            CfnBucket bucket = (CfnBucket)node;
            Object versioningConfiguration = bucket.getVersioningConfiguration();
            if (versioningConfiguration == null ||
                    !Tokenization.isResolvable(versioningConfiguration.toString())
 &&
                    !versioningConfiguration.toString().contains("Enabled"))
                Annotations.of(bucket.getNode()).addError("Bucket versioning is not
 enabled");
        }
    }
}


// Later, apply to the stack
```

```
Aspects.of(stack).add(new BucketVersioningChecker());
```

C#

```
class BucketVersioningChecker : Amazon.Jsii.Runtime.Deputy.DeputyBase, IAspect
{
    public void Visit(IConstruct node)
    {
        // See that we're dealing with a CfnBucket
        if (node is CfnBucket)
        {
            var bucket = (CfnBucket)node;
            if (bucket.VersioningConfiguration is null ||
                    !Tokenization.IsResolvable(bucket.VersioningConfiguration) &&
                    !bucket.VersioningConfiguration.ToString().Contains("Enabled"))
                Annotations.Of(bucket.Node).AddError("Bucket versioning is not
 enabled");
        }
    }
}

// Later, apply to the stack
Aspects.Of(stack).add(new BucketVersioningChecker());
```

开始使用 AWS CDK

AWS Cloud Development Kit (AWS CDK) 通过安装 AWS CDK CLI并创建您的第一个 CDK 应用程序开始使用。

主题

先决条件

推荐资源

在开始使用之前 AWS CDK，我们建议您对以下内容有基本的了解：

- 简介 AWS CDK. 要了解更多信息，请参阅 那是什么 AWS CDK？。
- 背后的核心概念 AWS CDK. 要了解更多信息，请参阅AWS CDK 概念。
- 你 AWS 服务 想用它来管理的 AWS CDK。
- AWS Identity and Access Management。 有关更多信息，请参阅什么是 IAM？ 以及什么是 IAM 身份中心？
- AWS CloudFormation 因为 AWS CDK 利用该 AWS CloudFormation 服务来提供 CDK 中创建的资源。要了解更多信息，请参阅什么是 AWS CloudFormation？
- 您计划在中使用的支持的编程语言 AWS CDK。

准备好您的本地环境

所有 AWS CDK 开发者，无论您的首选语言如何，都需要 Node.js14.15.0 或更高版本。所有支持的编程语言都使用相同的后端，该后端在上运行Node.js。我们建议使用长期有效的支持版本。您的组织可能有不同的建议。

> **Important**
>
> Node.js 版本 13.0.0 到 13.6.0 与不兼容， AWS CDK 因为其依赖项存在兼容性问题。

其他先决条件取决于您开发 AWS CDK 应用程序时使用的语言，如下所示。

TypeScript

- TypeScript 3.8 或更高版本 (`npm -g install typescript`)

JavaScript

没有额外要求

Python

- Python 3.7 或更高版本包括`pip`和 `virtualenv`

Java

- Java 开发套件 (JDK) 8（又名 1.8）或更高版本
- Apache Maven 3.5 或更高版本

推荐使用 Java IDE（我们在本指南的某些示例中使用了 Eclipse）。IDE 必须能够导入 Maven 项目。检查以确保您的项目已设置为使用 Java 1.8。将 JAVA_HOME 环境变量设置为安装 JDK 的路径。

C#

.NET 酷睿 3.1 或更高版本，或.NET 6.0 或更高版本。

推荐 Visual Studio 2019（任何版本）或 Visual Studio

Go

转到 1.1.8 或更高版本。

有关更多详细信息，请参阅您的语言的 “先决条件” 部分：

- the section called “在 TypeScript”
- the section called “在 JavaScript”

- the section called “在 Python 中”
- the section called “在 Java 中”
- the section called “在 C# 中”
- the section called “在 Go 中”

> **第三方语言的弃用**
>
> 每种语言版本仅在EOL（使用寿命终止）之前受支持，如有更改，恕不另行通知。

步骤 1：创建一个 AWS 账户

如果您不熟 AWS悉，则必须注册 AWS 账户 并创建管理用户。有关更多信息，请参阅 IAM 用户指南中的使用 IAM 进行设置。

当您与之互动时 AWS，您可以指定您的 AWS 安全凭证来验证您的身份以及您是否有权访问所请求的资源。 AWS 使用安全证书对您的请求进行身份验证和授权。要了解更多信息，请参阅 IAM 用户指南中的AWS 安全证书。

步骤 2：配置编程访问权限

在本地环境 AWS CDK 中使用开发时，您将依靠与资源 AWS CDK CLI进行交互 AWS 服务 并管理您的 AWS 资源。要使用 AWS CDK CLI，必须配置编程访问权限。要详细了解配置编程访问权限的不同方式，请参阅 AWS SDK 和工具参考指南中的身份验证和访问权限。

对于雇主未提供身份验证方法的新用户，我们建议使用 AWS IAM Identity Center。此方法包括安装 AWS Command Line Interface (AWS CLI)、使用它进行配置和登录 AWS 访问门户。要使用 IAM Identity Center 配置编程访问权限，请参阅《软件开发工具包和AWS 工具参考指南》中的 IAM 身份中心身份验证。完成后，您的环境应包含以下元素：

- AWS CLI，用于在运行应用程序之前启动 AWS 访问门户会话。
- 一种共享 AWSconfig文件，其[default]配置文件包含一组配置值，可以从中引用 AWS CDK。要查找此文件的位置，请参阅《AWS SDK 和工具参考指南》中的共享文件的位置。
- 共享 config 文件设置了 region 设置。这将设置 AWS 请求 AWS 区域 的默认 AWS CDK 用途。
- 在向发送请求之前， AWS CDK 使用配置文件的 SSO 令牌提供程序配置来获取凭证。 AWS 该sso_role_name值是与 IAM Identity Center 权限集关联的 IAM 角色，应允许访问您的应用程序中 AWS 服务 使用的权限。

以下示例 config 文件展示了使用 SSO 令牌提供程序配置来设置的默认配置文件。配置文件的 sso_session 设置是指所指定的 sso-session 节。该sso-session部分包含启动 AWS 访问门户会话的设置。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

启动 AWS 访问门户会话

在访问之前 AWS 服务，您需要一个有效的 AWS 访问门户会话 AWS CDK 才能使用 IAM Identity Center 身份验证来解析证书。根据您配置的会话时长，您的访问权限最终将过期，并且 AWS CDK 会遇到身份验证错误。在中运行以下命令登录 AWS CLI AWS 访问门户。

```
aws sso login
```

如果您的 SSO 令牌提供程序配置使用命名配置文件而不是默认配置文件，则命令为aws sso login --profile *NAME*。使用--profile选项或AWS_PROFILE环境变量发出cdk命令时，也要指定此配置文件。

要测试是否已有活动会话，请运行以下 AWS CLI 命令。

```
aws sts get-caller-identity
```

对此命令的响应应该报告共享 config 文件中配置的 IAM Identity Center 账户和权限集。

> **Note**
>
> 如果您已经有一个有效的 AWS 访问门户会话并且aws sso login正在运行，则无需提供凭据。

登录过程可能会提示您允许 AWS CLI 访问您的数据。由于 AWS CLI 是在适用于 Python 的 SDK 之上构建的，因此权限消息可能包含botocore名称的变体。

步骤 3：安装 AWS CDKCLI

使用以下 Package Man Node ager 命令 AWS CDK CLI全局安装。

```
npm install -g aws-cdk
```

> **Note**
>
> 如果您遇到权限错误，并且拥有系统管理员访问权限，请尝试sudo npm install -g aws-cdk。

运行以下命令以验证安装是否成功。 AWS CDK CLI应该输出版本号：

```
cdk --version
```

如果您收到错误消息，请尝试 AWS CDK CLI通过运行以下命令来卸载：

```
npm uninstall -g aws-cdk
```

然后，重复步骤以重新安装。 AWS CDK CLI

如果您仍然收到错误消息，请从当前项目和全局node-modules文件夹中删除该node-modules文件夹。要找到此文件夹，请运行npm config get prefix。

AWS CDK CLI将从您在前面步骤中配置的来源获取安全证书。

> **Note**
>
> CDK 工具包 v2 适用于现有的 CDK v1 项目。但是，它无法初始化新的 CDK v1 项目。看看你the section called “新的先决条件”是否需要能够做到这一点。

第 4 步：引导您的环境

您计划向其部署资源的每个 AWS 环境都必须进行引导。

要引导，请运行以下命令：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

> **Tip**
>
> 如果您手边没有 AWS 账号，可以从 AWS Management Console. 或者，如果您 AWS CLI 安装了，则以下命令会显示您的默认帐户信息，包括账号。
>
> ```
> aws sts get-caller-identity
> ```
>
> 如果您在本地配置中创建了命名 AWS 配置文件，则可以使用该`--profile`选项来显示特定配置文件的帐户信息。以下示例说明如何显示产品配置文件的账户信息。
>
> ```
> aws sts get-caller-identity --profile prod
> ```
>
> 要显示默认区域，请使用`aws configure get`。
>
> ```
> aws configure get region
> aws configure get region --profile prod
> ```

可选 AWS CDK 工具

AWS Toolkit for Visual Studio Code是 Visual Studio Code 的开源插件，可帮助您在上创建、调试和部署应用程序 AWS。该工具包为开发 AWS CDK 应用程序提供了集成体验。它包括 AWS CDK Explorer 功能，用于列出您的 AWS CDK 项目并浏览 CDK 应用程序的各个组件。安装插件并详细了解如何使用 AWS CDK Explorer。

后续步骤

既然你已经安装了 AWS CDK CLI，那就用它来构建你的第一个 AWS CDK 应用程序。

要了解有关使用首选编程语言 AWS CDK 的更多信息，请参阅AWS CDK 在支持的编程语言中使用。

AWS CDK 是一个开源项目。要做出贡献，请参阅贡献 AWS Cloud Development Kit (AWS CDK)。

了解更多信息

要了解更多信息 AWS CDK，请参阅以下内容：

- CDK 研讨会 — 深入的动手研讨会。
- API 参考 — 浏览可供你 AWS 服务 使用的构造。
- C@@ onstruct Hub — 从 CDK 社区中查找构造。
- AWS CDK 示例-浏览 AWS CDK 项目的代码示例。

你的第一个 AWS CDK 应用程序

AWS Cloud Development Kit (AWS CDK) 通过构建您的第一个 CDK 应用程序开始使用。

在开始本教程之前，我们建议您完成以下操作：

- 那是什么 AWS CDK？有关简介，请参阅 AWS CDK。
- AWS CDK 概念要了解的核心概念，请参阅 AWS CDK。
- 请浏览先决条件和 AWS CDK 设置步骤，网址为开始使用 AWS CDK。

主题

关于本教程

在本教程中，您将创建和部署一个简单的 AWS CDK 应用程序。此应用程序包含一个堆栈，其中包含一个亚马逊简单存储服务 (Amazon S3) 存储桶资源。通过本教程，您将学到以下内容：

- AWS CDK 项目的结构。
- 如何创建 AWS CDK 应用程序。
- 如何使用 AWS 构造库来定义应用程序、堆栈和 AWS 资源。
- 如何使用 CDK CLI 合成、区分、部署和删除您的 CDK 应用程序。
- 如何修改和重新部署 CDK 应用程序以更新已部署的资源。

标准 AWS CDK 开发工作流程包括以下步骤：

1. 创建您的 AWS CDK 应用程序 — 在这里，您将使用提供的模板 AWS CDK CLI。
2. 定义堆栈和资源-使用构造在应用程序中定义堆栈和 AWS 资源。
3. 构建您的应用程序-此步骤是可选的。如有必要， AWS CDK CLI会自动执行此步骤。建议执行此步骤以识别语法和类型错误。
4. 合成堆栈 — 此步骤将为应用程序中的每个堆栈创建一个 AWS CloudFormation 模板。此步骤对于识别已定义 AWS 资源中的逻辑错误非常有用。
5. 部署您的应用程序-使用部署到您的 AWS 环境中 AWS CloudFormation 以配置您的资源。在部署过程中，您将发现您的应用程序存在任何权限问题。

通过典型的工作流程，您将返回并重复之前的步骤来修改或调试您的应用程序。

我们建议您对 AWS CDK 项目使用版本控制。

步骤 1：创建应用程序

CDK 应用程序应位于自己的目录中，并具有自己的本地模块依赖关系。在开发计算机上，创建一个新目录。以下是创建新`hello-cdk`目录的示例：

```
$ mkdir hello-cdk
$ cd hello-cdk
```

> **Important**
>
> 请务必完全按照此处所示命名您的项目目录*hello-cdk*。 AWS CDK 项目模板使用目录名称来命名生成的代码中的内容。如果您使用其他名称，则本教程中的代码将无法使用。

接下来，从您的新目录中，使用cdk init命令初始化应用程序。使用选项指定app模板和您的首选编程语言。`--language`以下是 示例：

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

创建应用程序后，还要输入以下两个命令。它们会激活应用程序的 Python 虚拟环境并安装 AWS CDK 核心依赖项。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

如果您使用的是 IDE，则现在可以打开或导入项目。例如，在 Eclipse 中，选择 “文件” > “导入” > “Maven” > “现有的 Maven 项目”。确保将项目设置设置为使用 Java 8 (1.8)。

C#

```
cdk init app --language csharp
```

如果您使用的是 Visual Studio，请在`src`目录中打开解决方案文件。

Go

```
cdk init app --language go
```

创建应用程序后，还要输入以下命令来安装该应用程序所需的 AWS 构造库模块。

```
go get
```

该cdk init命令会在hello-cdk目录中创建许多文件和文件夹，以帮助您整理 AWS CDK 应用程序的源代码。统称为 “您的 AWS CDK 项目”。花点时间来探索 CDK 项目。

如果您已Git安装，则使用cdk init创建的每个项目也会初始化为Git存储库。

第 2 步：构建应用程序

在大多数编程环境中，都是在进行更改后生成或编译代码。这不是必需的， AWS CDK 因为 CDK CLI 会自动执行此步骤。但是，当你想捕获 catch 语法和键入错误时，你仍然可以手动构建。以下是 示例：

TypeScript

```
npm run build
```

JavaScript

无需执行任何构建步骤。

Python

无需执行任何构建步骤。

Java

```
mvn compile -q
```

或者在 Eclipse 中按 Control-B（其他 Java IDE 可能会有所不同）

C#

```
dotnet build src
```

或者在 Visual Studio 中按 F6

Go

```
go build
```

第 3 步：在应用程序中列出堆栈

通过在应用程序中列出堆栈，验证您的应用程序是否已正确创建。运行以下命令：

```
cdk ls
```

应显示输出HelloCdkStack。如果您没有看到此输出，请确认您位于项目的正确工作目录中，然后重试。如果您仍然看不到您的堆栈，请重复the section called “步骤 1：创建应用程序”并重试。

第 4 步：添加 Amazon S3 存储桶

此时，您的 CDK 应用程序包含一个堆栈。接下来，您将在堆栈中定义亚马逊简单存储服务 (Amazon S3) 存储桶资源。为此，您将导入并使用构造库中的 `Bucket` L2 AWS 构造。

通过导入Bucket构造并定义您的 Amazon S3 存储桶资源来修改您的 CDK 应用程序。以下是 示例：

TypeScript

In `lib/hello-cdk-stack.ts`:

```
import * as cdk from 'aws-cdk-lib';
import { aws_s3 as s3 } from 'aws-cdk-lib';

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

JavaScript

In `lib/hello-cdk-stack.js`:

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

module.exports = { HelloCdkStack }
```

Python

In `hello_cdk/hello_cdk_stack.py`:

```
import aws_cdk as cdk
import aws_cdk.aws_s3 as s3

class HelloCdkStack(cdk.Stack):

    def __init__(self, scope: cdk.App, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        bucket = s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

In `src/main/java/com/myorg/HelloCdkStack.java`:

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.Bucket;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final App scope, final String id) {
        this(scope, id, null);
    }
```

```
    public HelloCdkStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

C#

In `src/HelloCdk/HelloCdkStack.cs`:

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdk
{
    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(App scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps
            {
                Versioned = true
            });
        }
    }
}
```

Go

In `hello-cdk.go`:

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)
```

```
type HelloCdkStackProps struct {
 awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}

func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)

 NewHelloCdkStack(app, "HelloCdkStack", &HelloCdkStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 app.Synth(nil)
}

func env() *awscdk.Environment {
 return nil
}
```

让我们仔细看看这个Bucket构造。像所有构造一样，该Bucket类采用三个参数：

- s@@ cop e — 将Stack类定义为Bucket构造的父类。所有定义 AWS 资源的构造都是在堆栈的范围内创建的。你可以在构造内部定义构造，从而创建层次结构（树）。在这里，在大多数情况下，作用域是this（selfinPython）。

- ID — 您的 AWS CDK 应用程序`Bucket`中的逻辑 ID。此 ID 加上基于存储桶在堆栈中的位置的哈希值，可在部署期间唯一标识存储桶。当您在应用程序中更新构造并重新部署以更新已部署的资源时，AWS CDK 也会引用此 ID。这里，你的逻辑 ID 是`MyFirstBucket`。存储桶也可以有一个名称，使用`bucketName`属性指定。这与逻辑 ID 不同。
- props — 一组定义存储桶属性的值。在这里，您将`versioned`属性定义为`true`，它允许对存储桶中的文件进行版本控制。

  在支持的语言中，道具的表示方式不同。 AWS CDK

  - 在 and 中 TypeScript JavaScript，`props`是单个参数，您可以传入一个包含所需属性的对象。
  - 在 Python 中，道具作为关键字参数传递。
  - 在 Java 中，提供了一个生成器来传递道具。有两个：一个用于`BucketProps`，第二个用于`Bucket`让你一步构建构造及其道具对象。这段代码使用后者。
  - 在 C# 中，你使用`BucketProps`对象初始化器实例化一个对象，并将其作为第三个参数传递。

  如果构造的 props 是可选的，则可以完全省略该`props`参数。

所有构造都采用相同的三个参数，因此在学习新结构时很容易保持定向。正如你所预料的那样，你可以对任何构造进行子类化以扩展它以满足你的需求，或者如果你想更改其默认值。

第 5 步：合成模板 AWS CloudFormation

合成应用程序的 AWS CloudFormation 模板，如下所示：

```
cdk synth
```

如果您的应用程序包含多个堆栈，则必须指定要合成哪些堆栈。由于您的应用程序包含单个堆栈，CDK CLI 会自动检测要合成的堆栈。

如果您不运行`cdk synth`，CDK CLI 将在您部署时自动执行此步骤。但是，我们建议您在每次部署之前运行此步骤。

> **Tip**
>
> 如果您收到诸如这样的错误`--app is required ...`，请检查您正在从中运行 CDK CLI 命令的目录。你应该在你的主应用程序目录中。

该cdk synth命令运行您的应用程序。这会为应用程序中的每个堆栈创建一个 AWS CloudFormation 模板。CDK CLI 将在命令行中显示模板的 YAML 格式版本，并将模板的 JSON 格式版本保存在目录中cdk.out。以下是命令行输出的片段，显示了 AWS CloudFormation 模板中定义的存储桶：

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
    Metadata:...
```

> **Note**
>
> 默认情况下，生成的每个模板都包含一个AWS::CDK::Metadata资源。该 AWS CDK 团队使用这些元数据来深入了解 AWS CDK 使用情况，并找到改进的方法。有关详细信息，包括如何选择退出版本报告，请参阅版本报告。

生成的模板可以通过 AWS CloudFormation 控制台或任何 AWS CloudFormation 部署工具进行部署。您也可以使用 CDK CLI 进行部署。在下一步中，您将使用 CDK CLI 进行部署。

第 6 步：部署您的堆栈

要 AWS CloudFormation 使用 CDK 部署 CDK 堆栈CLI，请运行以下命令：

```
cdk deploy
```

> **Important**
>
> 在部署之前，您必须对 AWS 环境进行一次性引导。有关说明，请参阅引导您的环境。

与之类似cdk synth，您无需指定 AWS CDK 堆栈，因为该应用程序包含单个堆栈。

如果您的代码有安全隐患，CDK CLI 将输出摘要。您需要确认它们才能继续部署。本教程中的应用程序没有这些含义。

运行后`cdk deploy`，CDK CLI 会在部署堆栈时显示进度信息。完成后，您可以前往AWS CloudFormation 控制台查看您的`HelloCdkStack`堆栈。您也可以前往 Amazon S3 控制台查看您的`MyFirstBucket`资源。

恭喜您！您已经使用部署了第一个堆栈 AWS CDK。接下来，您将修改您的应用程序并重新部署以更新您的资源。

第 7 步：修改您的应用程序

在此步骤中，您将修改您的 Amazon S3 存储桶，将其配置为在堆栈被删除时自动删除。此修改涉及更改存储桶的`RemovalPolicy`属性。您还将配置该`autoDeleteObjects`属性，将 CDK CLI 配置为在销毁存储桶之前从存储桶中删除对象。默认情况下， AWS CloudFormation 不会删除包含对象的 Amazon S3 存储桶。

使用以下示例修改您的资源：

TypeScript

更新 `lib/hello-cdk-stack.ts`

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

JavaScript

更新 `lib/hello-cdk-stack.js`

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

Python

更新 `hello_cdk/hello_cdk_stack.py`

```
bucket = s3.Bucket(self, "MyFirstBucket",
```

```
    versioned=True,
    removal_policy=cdk.RemovalPolicy.DESTROY,
    auto_delete_objects=True)
```

Java

更新 `src/main/java/com/myorg/HelloCdkStack.java`

```
Bucket.Builder.create(this, "MyFirstBucket")
        .versioned(true)
        .removalPolicy(RemovalPolicy.DESTROY)
        .autoDeleteObjects(true)
        .build();
```

C#

更新 `src/HelloCdk/HelloCdkStack.cs`

```
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true,
    RemovalPolicy = RemovalPolicy.DESTROY,
    AutoDeleteObjects = true
});
```

Go

更新 `hello-cdk.go`

```
   awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned:         jsii.Bool(true),
  RemovalPolicy:     awscdk.RemovalPolicy_DESTROY,
  AutoDeleteObjects: jsii.Bool(true),
 })
```

目前，您的代码更改尚未对已部署的 Amazon S3 存储桶资源进行任何直接更新。您的代码定义了资源的所需状态。要修改已部署的资源，您将使用 CDK 将所需状态合成CLI到新 AWS CloudFormation 模板中。然后，您将新 AWS CloudFormation 模板部署为更改集。变更集仅进行必要的更改以达到新的所需状态。

要查看这些更改，请使用`cdk diff`命令。运行以下命令：

```
cdk diff
```

CDK 会向您的 AWS 账户 账户CLI查询HelloCdkStack堆栈的最新 AWS CloudFormation 模板。然后，它将最新的模板与刚从您的应用程序中合成的模板进行比较。输出应与以下内容类似。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                    # Effect # Action                      # Principal
                  # Condition #
###############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole              #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                           #
                  #           #
#   # .Arn}                         #        #                           #
                  #           #
###############################################################################################
# + # ${MyFirstBucket.Arn}         # Allow  # s3:DeleteObject*            # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*       #        # s3:GetBucket*               #
 jectsCustomResourceProvider/ #           #
#   #                               #        # s3:GetObject*               # Role.Arn}
                  #           #
#   #                               #        # s3:List*                    #
                  #           #
###############################################################################################
IAM Policy Changes
###############################################################################################
#   # Resource                                            # Managed Policy ARN
                               #
###############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                 # ice-role/
AWSLambdaBasicExecutionRole"}                 #
###############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
```

```
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
     ## [-] Retain
     ## [+] Delete
```

此差异分为四个部分：

- IAM 声明变更和 IAM 政策变更 — 之所以出现这些权限变更，是因为您在 Amazon S3 存储桶上设置了该`AutoDeleteObjects`属性。自动删除功能使用自定义资源在删除存储桶本身之前删除存储桶中的对象。IAM 对象授予自定义资源的代码访问存储桶的权限。
- 参数- AWS CDK 使用这些条目来查找自定义资源的 AWS Lambda 函数资产。
- 资源-此堆栈中新的和已更改的资源。我们可以看到前面提到的 IAM 对象、自定义资源及其关联的 Lambda 函数正在添加中。我们还可以看到存储桶`DeletionPolicy`和`UpdateReplacePolicy`属性正在更新。它们允许将存储桶与堆栈一起删除，并用新的堆栈替换。

你可能会注意到，我们在 AWS CDK 应用程序`RemovalPolicy`中指定了，但在生成的 AWS CloudFormation 模板中却有一个`DeletionPolicy`属性。这是因为该属性 AWS CDK 使用了不同的名称。 AWS CDK 默认设置是在删除堆栈时保留存储桶，而 AWS CloudFormation 默认设置是将其删除。有关更多信息，请参阅 the section called “移除政策”。

要查看您的新 AWS CloudFormation 模板，您可以运行cdk synth。通过对 CDK 应用程序进行一些更改，与原始 AWS CloudFormation AWS CloudFormation 模板相比，您的新模板现在包含了许多额外的代码行。

接下来，通过运行以下命令来部署您的应用程序：

```
cdk deploy
```

AWS CDK 将告知您我们已经在差异中看到的安全策略变更。输入y以批准更改并部署更新的堆栈。CDK CLI 将部署您的堆栈以进行所需的更改。下面是一个示例输出：

```
HelloCdkStack: deploying...
[0%] start: Publishing
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
[100%] success: Published
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
HelloCdkStack: creating CloudFormation changeset...
 0/5 | 4:32:31 PM | UPDATE_IN_PROGRESS   | AWS::CloudFormation::Stack  | HelloCdkStack
 User Initiated
 0/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
        | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
 1/5 | 4:32:36 PM | UPDATE_COMPLETE      | AWS::S3::Bucket             | MyFirstBucket
 (MyFirstBucketB8884501)
 1/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
        | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092) Resource creation
 Initiated
 3/5 | 4:32:54 PM | CREATE_COMPLETE      | AWS::IAM::Role
        | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
```

```
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F) Resource creation
 Initiated
 3/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::Lambda::Function
      | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:57 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD) Resource creation Initiated
 4/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
 4/5 | 4:32:59 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
 5/5 | 4:33:06 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E) Resource creation Initiated
 5/5 | 4:33:06 PM | CREATE_COMPLETE      | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
 5/5 | 4:33:08 PM | UPDATE_COMPLETE_CLEA | AWS::CloudFormation::Stack  | HelloCdkStack
 6/5 | 4:33:09 PM | UPDATE_COMPLETE      | AWS::CloudFormation::Stack  | HelloCdkStack

 #  HelloCdkStack

Stack ARN:
arn:aws:cloudformation:REGION:ACCOUNT:stack/HelloCdkStack/UNIQUE-ID
```

第 8 步：销毁应用程序的资源

现在您已经完成了本教程，可以删除已部署的 AWS CloudFormation 堆栈以及与之关联的所有资源。这是一种很好的做法，可以最大限度地减少不必要的成本并保持环境清洁。运行以下命令：

```
cdk destroy
```

输入y以批准更改并删除您的堆栈。

> **Note**
>
> 如果您没有更改存储桶`RemovalPolicy`，则堆栈删除将成功完成，但存储桶将变为孤立存储桶（不再与堆栈关联）。

后续步骤

恭喜您！您已完成本教程，并已使用成功创建、修改和删除中的资源 AWS Cloud。 AWS CDK 您现在可以开始使用了 AWS CDK。

要了解有关使用首选编程语言 AWS CDK 的更多信息，请参阅AWS CDK 在支持的编程语言中使用。

有关其他资源，请参阅以下内容：

- 试试 CDK Workshop，深入了解更复杂的项目。
- 深入了解the section called “环境”、、、the section called “资产”the section called “权限”the section called “上下文”the section called “参数”、和等概念the section called “自定义构造”。
- 请参阅 API 参考资料，开始探索可用于您最喜欢的 AWS 服务的 CDK 结构。
- 访问 C onstruc t Hub，发现由 AWS 和其他人创建的构造。
- 浏览使用示例 AWS CDK.

AWS CDK 是一个开源项目。要做出贡献，请参阅 “贡献” AWS Cloud Development Kit (AWS CDK)。

从 AWS CDK v1 迁移到 v2 AWS CDK

的版本 2 AWS Cloud Development Kit (AWS CDK) 旨在让使用您首选的编程语言更轻松地将基础架构作为代码进行编写。本主题介绍的 v1 和 v2 之间的变化。 AWS CDK

> **Tip**
>
> 要识别使用 v1 部署的堆栈，请使用 aw scdk AWS CDK -v1-stack-finder 实用程序。

从 AWS CDK v1 到 CDK v2 的主要变化如下。

- AWS CDK v2 将 Con AWS struct 库的稳定部分（包括核心库）整合到一个包中。`aws-cdk-lib`开发人员不再需要为他们使用的个别 AWS 服务安装额外的软件包。这种单包方法还意味着您不必同步各种 CDK 库包的版本。

  代表中确切可用资源的 L1 (cfnxxxx) 结构始终被认为是稳定的 AWS CloudFormation，因此包含在中。`aws-cdk-lib`
- 实验模块不包括在内，我们仍在与社区合作开发新的 L2 或 L3 结构。`aws-cdk-lib`相反，它们是作为单独的软件包分发的。实验包以`alpha`后缀和语义版本号命名。语义版本号与它们兼容的 Construct L AWS ibrary 的第一个版本相匹配，还有一个`alpha`后缀。构造`aws-cdk-lib`在被指定为稳定版本后移入，从而允许主构造库遵守严格的语义版本控制。

  稳定性是在服务级别指定的。例如，如果我们开始为 Amazon 创建一个或多个 L2 构造，而在撰写本文时 AppFlow，这些构造只有 L1 结构，则它们首先出现在名为的模块中。`@aws-cdk/aws-appflow-alpha`然后，`aws-cdk-lib`当我们认为新结构可以满足客户的基本需求时，它们就会移动。

  一旦模块被指定为稳定模块并入其中`aws-cdk-lib`，就会使用下一个项目符号中描述的 “BetAN” 约定添加新的 API。

  每个实验模块的新版本都会随之发布 AWS CDK。但是，在大多数情况下，它们不需要保持同步。你可以随时升级`aws-cdk-lib`或实验模块。唯一的例外是，当两个或多个相关的实验模块相互依赖时，它们必须是相同的版本。
- 对于正在添加新功能的稳定模块，新 API（无论是全新的构造还是现有构造上的新方法或属性）都会在工作进行时获得`Beta1`后缀。（当需要进行重大更改时 `Beta2`、`Beta3`，其次是、等。）当 API 被指定为稳定版本时，会添加不带后缀的 API 版本。除最新方法（无论是测试版还是最终版）之外的所有方法都将被弃用。

例如，如果我们在构造中添加一个新方法`grantPower()`，它最初会显示为`grantPowerBeta1()`。如果需要进行重大更改（例如，新的必需参数或属性），则将命名该方法的下一个版本`grantPowerBeta2()`，依此类推。当工作完成并且 API 最终确定后，将添加该方法`grantPower()`（不带后缀），并弃用 BetAN 方法。

在下一个主要版本 (3.0) 发布之前，所有测试版 API 都将保留在构造库中，并且它们的签名不会改变。如果你使用它们，你会看到弃用警告，因此你应该尽早转到 API 的最终版本。但是，任何未来的 AWS CDK 2.x 版本都不会破坏你的应用程序。

- 该`Construct`类已与相关类型一起从中提取 AWS CDK 到单独的库中。这样做是为了支持将构造编程模型应用于其他领域的努力。如果您正在编写自己的构造或使用相关的 API，则必须将该`constructs`模块声明为依赖项，并对导入进行细微的更改。如果您使用的是高级功能，例如连接到 CDK 应用程序生命周期，则可能需要进行更多更改。有关完整详细信息，请参阅 RFC。
- AWS CDK v1.x 及其构造库中已弃用的属性、方法和类型已从 CDK v2 API 中完全删除。在大多数支持的语言中，这些 API 在 v1.x 下都会生成警告，因此您可能已经迁移到替换 API。CDK v1.x 中已弃用 API 的完整列表可在上找到。 GitHub
- 在 CDK v2 中，默认情况下，在 AWS CDK v1.x 中受功能标志限制的行为处于启用状态。不再需要早期的功能标志，而且在大多数情况下也不支持这些标志。在非常特殊的情况下，还有一些可以让你恢复到 CDK v1 的行为。有关更多信息，请参阅 the section called “更新功能标志”。
- 在 CDK v2 中，您部署到的环境必须使用现代引导堆栈进行引导。不再支持旧版引导堆栈（v1 下的默认堆栈）。CDK v2 还需要新版本的现代堆栈。要升级现有环境，请重新启动它们。无需再设置任何功能标志或环境变量即可使用现代 bootstrap 堆栈。

> **Important**
>
> 现代引导模板可以有效地向`--trust`列表中的任何 AWS 账户授`--cloudformation-execution-policies`予隐含的权限。默认情况下，这会扩展对引导账户中任何资源的读写权限。请务必使用您熟悉的策略和可信帐户配置引导堆栈。

新的先决条件

AWS CDK v2 的大多数要求与 v1.x 的要求相同。 AWS CDK 此处列出了其他要求。

- 对于 TypeScript 开发人员，需要 TypeScript 3.8 或更高版本。

- 在 CDK v2 中使用需要新版本的 CDK 工具包。现在 CDK v2 已正式上线，因此安装 CDK 工具包时，v2 是默认版本。它与 CDK v1 项目向后兼容，因此除非要创建 CDK v1 项目，否则无需继续安装早期版本。要升级，请发出`npm install -g aws-cdk`。

从 AWS CDK v2 开发者预览版升级

如果您使用的是 CDK v2 开发者预览版，则您的项目依赖于的发布候选版本 AWS CDK，例如。`2.0.0-rc1`将它们更新为`2.0.0`，然后更新项目中安装的模块。

TypeScript

`npm install` 或 `yarn install`

JavaScript

`npm install` 或 `yarn install`

Python

```
python -m pip install -r requirements.txt
```

Java

```
mvn package
```

C#

```
dotnet restore
```

Go

```
go get
```

更新依赖项后，发出 CDK Toolkit 更新`npm update -g aws-cdk`到发布版本的命令。

从 AWS CDK v1 迁移到 CDK v2

要将您的应用程序迁移到 AWS CDK v2，请先更新中的功能标志。`cdk.json`然后，根据需要更新应用程序的依赖关系和导入，以适应其编写的编程语言。

更新到最近的 v1

我们看到许多客户一步从旧版本的 AWS CDK v1 升级到最新版本的 v2。虽然做到这一点当然是可能的，但你既要在多年的变化中进行升级（不幸的是，这些变更可能不是所有人都进行了与今天相同数量的演化测试），也要跨具有新默认值和不同代码组织的版本进行升级。

为了获得最安全的升级体验并更轻松地诊断任何意外更改的来源，我们建议您将这两个步骤分开：首先升级到最新的 v1 版本，然后再切换到 v2。

更新功能标志

`cdk.json`如果存在以下 v1 功能标志，请将其删除，因为默认情况下，这些标志在 AWS CDK v2 中均处于活动状态。如果它们的旧效果对您的基础架构很重要，则需要对源代码进行更改。有关更多信息，GitHub请参阅上的旗帜列表。

- `@aws-cdk/core:enableStackNameDuplicates`
- `aws-cdk:enableDiffNoFail`
- `@aws-cdk/aws-ecr-assets:dockerIgnoreSupport`
- `@aws-cdk/aws-secretsmanager:parseOwnedSecretName`
- `@aws-cdk/aws-kms:defaultKeyPolicies`
- `@aws-cdk/aws-s3:grantWriteWithoutAcl`
- `@aws-cdk/aws-efs:defaultEncryptionAtRest`

可以将少量 v1 功能标志设置为`false`以恢复到特定的 AWS CDK v1 行为；有关完整参考，请参阅the section called “恢复到 v1 行为”或上的 GitHub 列表。

对于这两种类型的标志，请使用`cdk diff`命令检查对合成模板的更改，以查看对其中任何一个标志的更改是否会影响您的基础架构。

CDK 工具包兼容性

CDK v2 需要 CDK 工具包的 v2 或更高版本。此版本与 CDK v1 应用程序向后兼容。因此，无论项目使用 v1 还是 v2，您都可以在所有 AWS CDK 项目中使用一个全球安装的 CDK Toolkit 版本。一个例外是，CDK Toolkit v2 仅创建 CDK v2 项目。

如果您需要同时创建 v1 和 v2 CDK 项目，请不要全局安装 CDK Toolkit v2。（如果您已经安装了它，请将其删除:`npm remove -g aws-cdk`.）要调用 CDK 工具包，请`npx`根据需要使用运行 CDK 工具包的 v1 或 v2。

```
npx aws-cdk@1.x init app --language typescript
npx aws-cdk@2.x init app --language typescript
```

Tip

设置命令行别名，以便您可以使用cdk和cdk1命令调用所需版本的 CDK Toolkit。

macOS/Linux

```
alias cdk1="npx aws-cdk@1.x"
alias cdk="npx aws-cdk@2.x"
```

Windows

```
doskey cdk1=npx aws-cdk@1.x $*
doskey cdk=npx aws-cdk@2.x $*
```

更新依赖关系和导入

更新应用程序的依赖关系，然后安装新的软件包。最后，更新代码中的导入。

TypeScript

应用程序

对于 CDK 应用程序，请按`package.json`如下方式进行更新。移除对 v1 风格的单个稳定模块的依赖，并建立应用程序`aws-cdk-lib`所需的最低版本（此处为 2.0.0）。

实验构造在单独的、独立版本化的包中提供，其名称以结尾`alpha`并带有 alpha 版本号。alpha 版本号对应于`aws-cdk-lib`与之兼容的第一个版本。在这里，我们已固定`aws-codestar`到 v2.0.0-alpha.1。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
```

```
}
```

构造图书馆

对于构造库，请为应用程序建立aws-cdk-lib所需的最低版本（此处为 2.0.0），并按如下方式进行更新package.json。

请注意，这既aws-cdk-lib显示为对等依赖关系，也显示为开发依赖关系。

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

> **Note**
>
> 在发布兼容 v2 的库时，你应该对库的版本号进行主要版本提升，因为这对库使用者来说是一个重大变化。不可能用一个库同时支持 CDK v1 和 v2。要继续为仍在使用 v1 的客户提供支持，您可以并行维护早期版本，或者为 v2 创建新软件包。
> 您想继续支持 AWS CDK v1 客户多长时间由您决定。你可以从 CDK v1 本身的生命周期中汲取灵感，该生命周期于 2022 年 6 月 1 日进入维护阶段，并将于 2023 年 6 月 1 end-of-life 日到期。有关完整详细信息，请参阅AWS CDK 维护政策。

既有库又有应用程序

通过运行npm install或来安装新的依赖项yarn install。

将导入更改为Construct从新constructs模块、核心类型（例如App和Stack从的顶层）导入aws-cdk-lib，以及从下的命名空间中使用的服务的稳定构造库模块导入。aws-cdk-lib

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';                // core constructs
import { aws_s3 as s3 } from 'aws-cdk-lib';              // stable module
```

```
import * as codestar from '@aws-cdk/aws-codestar-alpha';  // experimental module
```

JavaScript

更新package.json如下。移除对 v1 风格的单个稳定模块的依赖，并建立应用程序aws-cdk-lib所需的最低版本（此处为 2.0.0）。

实验构造在单独的、独立版本化的包中提供，其名称以结尾alpha并带有 alpha 版本号。alpha 版本号对应于aws-cdk-lib与之兼容的第一个版本。在这里，我们已固定aws-codestar到 v2.0.0-alpha.1。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
}
```

通过运行npm install或来安装新的依赖项yarn install。

更改应用程序的导入以执行以下操作：

- Construct从新constructs模块导入
- 从顶层导入核心类型Stack，例如App和 aws-cdk-lib
- 从下的命名空间导入 AWS 构造库模块 aws-cdk-lib

```
const { Construct } = require('constructs');
const { App, Stack } = require('aws-cdk-lib');              // core constructs
const s3 = require('aws-cdk-lib').aws_s3;                   // stable module
const codestar = require('@aws-cdk/aws-codestar-alpha');    // experimental module
```

Python

更新requirements.txt或中的install_requires定义，setup.py如下所示。移除对 v1 风格的单个稳定模块的依赖。

实验构造在单独的、独立版本化的包中提供，其名称以结尾alpha并带有 alpha 版本号。alpha 版本号对应于aws-cdk-lib与之兼容的第一个版本。在这里，我们已固定aws-codestar到 v2.0.0alpha1。

```
install_requires=[
      "aws-cdk-lib>=2.0.0",
      "constructs>=10.0.0",
      "aws-cdk.aws-codestar-alpha>=2.0.0alpha1",
      # ...
],
```

> **Tip**
>
> 使用卸载已安装在应用程序虚拟环境中的任何其他版本的 AWS CDK 模块`pip uninstall`。然后使用安装新的依赖项`python -m pip install -r requirements.txt`。

更改应用程序的导入以执行以下操作：

- `Construct`从新`constructs`模块导入
- 从顶层导入核心类型`Stack`，例如`App`和 `aws_cdk`
- 从下的命名空间导入 AWS 构造库模块 `aws_cdk`

```
from constructs import Construct
from aws_cdk import App, Stack                    # core constructs
from aws_cdk import aws_s3 as s3                  # stable module
import aws_cdk.aws_codestar_alpha as codestar     # experimental module

# ...

class MyConstruct(Construct):
    # ...

class MyStack(Stack):
    # ...

s3.Bucket(...)
```

Java

在中`pom.xml`，删除稳定模块的所有`software.amazon.awscdk`依赖关系，并将其替换为对 `software.constructs` (for`Construct`) 和的依赖关系`software.amazon.awscdk`。

实验构造在单独的、独立版本化的包中提供，其名称以结尾alpha并带有 alpha 版本号。alpha 版本号对应于aws-cdk-lib与之兼容的第一个版本。在这里，我们已固定aws-codestar到 v2.0.0-alpha.1。

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0</version>
</dependency><dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>code-star-alpha</artifactId>
    <version>2.0.0-alpha.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
</dependency>
```

通过运行安装新的依赖项mvn package。

更改您的代码以执行以下操作：

- Construct从新software.constructs库导入
- 从中导入核心类App，比如Stack和 software.amazon.awscdk
- 从中导入服务构造 software.amazon.awscdk.services

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.App;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.codestar.alpha.GitHubRepository;
```

C#

升级 C# CDK 应用程序依赖关系的最直接方法是手动编辑.csproj文件。删除所有稳定的Amazon.CDK.*软件包引用，并将其替换为对Amazon.CDK.Lib和Constructs软件包的引用。

实验构造在单独的、独立版本化的包中提供，其名称以结尾`alpha`并带有 alpha 版本号。alpha 版本号对应于`aws-cdk-lib`与之兼容的第一个版本。在这里，我们已固定`aws-codestar`到 v2.0.0-alpha.1。

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0" />
<PackageReference Include="Amazon.CDK.AWS.Codestar.Alpha" Version="2.0.0-alpha.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

通过运行安装新的依赖项`dotnet restore`。

按如下所示更改源文件中的导入。

```
using Constructs;                   // for Construct class
using Amazon.CDK;                   // for core classes like App and Stack
using Amazon.CDK.AWS.S3;            // for stable constructs like Bucket
using Amazon.CDK.Codestar.Alpha;    // for experimental constructs
```

Go

将依赖项更新`go get`到最新版本并更新项目`.mod`文件的问题。

在部署之前测试已迁移的应用程序

在部署堆栈之前，`cdk diff`请使用检查资源是否有意外更改。预计不会对逻辑 ID 进行更改（导致资源替换）。

预期的变化包括但不限于：

- 对`CDKMetadata`资源的更改。
- 更新了资产哈希值。
- 与新式堆栈合成相关的更改。如果您的应用在 v1 中使用了旧版堆栈合成器，则适用。（CDK v2 不支持旧版堆栈合成器。）
- 添加`CheckBootstrapVersion`规则。

升级到 AWS CDK v2 本身通常不会导致意外更改。通常，它们是由先前被功能标志更改的过时行为的结果。这是从大约 1.85.x 之前的 CDK 版本升级的症状。升级到最新的 v1.x 版本时，您会看到相同的更改。通常，您可以通过执行以下操作来解决此问题：

1. 将您的应用程序升级到最新的 v1.x 版本
2. 移除功能标志
3. 根据需要修改您的代码
4. 部署
5. 升级到 v2

> **Note**
>
> 如果升级后的应用程序在两阶段升级后无法部署，请报告问题。

当您准备好在应用程序中部署堆栈时，可以考虑先部署一个副本，以便对其进行测试。最简单的方法是将其部署到不同的区域。但是，您也可以更改堆栈的 ID。测试完成后，请务必使用销毁测试副本cdk destroy。

故障排除

TypeScript **`'from' expected`**或者导入**`';' expected`**时出错

升级到 TypeScript 3.8 或更高版本。

运行 “cdk bootstrap”

如果你看到类似以下的错误：

```
#  MyStack failed: Error: MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not
 found. Has the environment been bootstrapped? Please run 'cdk bootstrap' (see https://
docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
    at CloudFormationDeployments.validateBootstrapStackVersion (.../aws-cdk/lib/api/
cloudformation-deployments.ts:323:13)
    at processTicksAndRejections (internal/process/task_queues.js:97:5)
MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not found. Has the environment
 been bootstrapped? Please run 'cdk bootstrap' (see https://docs.aws.amazon.com/cdk/
latest/guide/bootstrapping.html)
```

AWS CDK v2 需要更新的引导堆栈，而且，所有 v2 部署都需要引导资源。（使用 v1，您无需引导即可部署简单的堆栈。）有关完整的详细信息，请参阅 the section called “正在引导”。

查找 v1 堆栈

将 CDK 应用程序从 v1 迁移到 v2 时，您可能需要识别使用 v1 创建的已部署 AWS CloudFormation 堆栈。为此，请运行以下命令：

```
npx awscdk-v1-stack-finder
```

有关用法的详细信息，请参阅 awscdk-v1-stack-finder 自述文件。

将现有资源和 AWS CloudFormation 模板迁移到 AWS CDK

CDK Migrate 功能处于预览版 AWS CDK ，可能会发生变化。

使用 AWS Cloud Development Kit (AWS CDK) 命令行界面 (AWS CDK CLI) 将已部署的 AWS 资源、已部署的 AWS CloudFormation 堆栈和本地 AWS CloudFormation 模板迁移到 AWS CDK。

主题

迁移的工作原理

使用 AWS CDK CLI`cdk migrate`命令从以下来源迁移：

- 已部署的 AWS 资源。
- 已部署 AWS CloudFormation 堆栈。
- 本地 AWS CloudFormation 模板。

已部署的 AWS 资源

您可以从特定环境（ AWS 账户 和 AWS 区域 ）迁移未与 AWS CloudFormation 堆栈关联的已部署 AWS 资源。

利 AWS CDK CLI用 IaC 生成器服务扫描 AWS 环境中的资源以收集资源详细信息。要了解有关 IaC 生成器的更多信息，请参阅AWS CloudFormation 用户指南中的为现有资源生成模板。

收集资源详细信息后，将 AWS CDK CLI创建一个新的 CDK 应用程序，其中包含一个包含已迁移资源的堆栈。

已部署的 AWS CloudFormation 堆栈

您可以将单个 AWS CloudFormation 堆栈迁移到新 AWS CDK 应用程序中。 AWS CDK CLI将检索您的堆栈 AWS CloudFormation 模板并创建一个新的 CDK 应用程序。CDK 应用程序将由包含您迁移的堆栈的单个 AWS CloudFormation 堆栈组成。

本地 AWS CloudFormation 模板

您可以从本地 AWS CloudFormation 模板迁移。本地模板可能包含也可能不包含已部署的资源。AWS CDK CLI将创建一个新的 CDK 应用程序，其中包含一个包含您的资源的堆栈。

迁移后，您可以管理、修改和部署您的 CDK 应用程序， AWS CloudFormation 以配置或更新您的资源。

CDK 迁移的好处

从历史上看 AWS CDK ，将资源迁移到中一直是一个手动过程，需要时间 AWS CloudFormation 和专业知识 AWS CDK ，甚至需要开始。使用 CDK Migr AWS CDK CLI ate，可以在很短的时间内为您完成大部分迁移工作。CDK Migrate 可以让你快速开始使用开发和管理新的和现有的应用程序。 AWS CDK AWS

注意事项

一般注意事项

CDK 迁移与 CDK 导入

该`cdk import`命令可以将已部署的资源导入到新的或现有的 CDK 应用程序中。导入时，必须在您的应用程序中将每个资源手动定义为 L1 结构。我们建议使用一次`cdk import`将一个或多个资源导入到新的或现有的 CDK 应用程序中。要了解更多信息，请参阅将现有的资源导入到堆栈。

该`cdk migrate`命令将从已部署的资源、已部署的 AWS CloudFormation 堆栈或本地 AWS CloudFormation 模板迁移到新的 CDK 应用程序中。在迁移过程中， AWS CDK CLI用于将您的资源导入`cdk import`到新的 CDK 应用程序中。 AWS CDK CLI还会为您生成每种资源的 L1 结构。我们建议在从支持的迁移源导入到新 AWS CDK 应用程序`cdk migrate`时使用。

CDK Migrate 仅创建 L1 构造

新创建的 CDK 应用程序将仅包含 L1 结构。迁移后，您可以在应用程序中添加更高级别的结构。

CDK Migrate 创建包含单个堆栈的 CDK 应用程序

新创建的 CDK 应用程序将包含一个堆栈。

迁移已部署的资源时，所有迁移的资源都将包含在新 CDK 应用程序的单个堆栈中。

迁移 AWS CloudFormation 堆栈时，您只能在新的 CDK 应用程序中将单个 AWS CloudFormation 堆栈迁移到单个堆栈中。

迁移资产

项目资产（例如 AWS Lambda 代码）不会直接迁移到新的 CDK 应用程序中。迁移后，您可以指定资产值以将其包含在 CDK 应用程序中。

迁移有状态资源

在迁移有状态资源（例如数据库和亚马逊简单存储服务 (Amazon S3) Simple Storage S3S 存储桶）时，您通常希望迁移现有资源，而不是创建新资源。

要迁移和保留有状态资源，请执行以下操作：

- 确认您的有状态资源支持导入。有关更多信息，请参阅《AWS CloudFormation 用户指南》中的资源类型支持。
- 迁移后，验证新 CDK 应用程序中已迁移资源的逻辑 ID 是否与已部署资源的逻辑 ID 相匹配。
- 如果从 AWS CloudFormation 堆栈迁移，请验证新 CDK 应用程序中的堆栈名称是否与 AWS CloudFormation 堆栈匹配。
- 使用相同的 AWS 帐户和 AWS 区域 迁移的资源部署 CDK 应用程序。

从 AWS CloudFormation 模板迁移时的注意事项

CDK Migrate 支持单模板迁移

迁移 AWS CloudFormation 模板时，您可以选择单个模板进行迁移。不支持嵌套模板。

迁移带有内在函数的模板

从使用内部函数的 AWS CloudFormation 模板迁移时， AWS CDK CLI将尝试使用该类将您的逻辑迁移到 CDK 应用程序中。Fn要了解更多信息，请参阅 AWS Cloud Development Kit (AWS CDK) API 参考中的 Fn 类。

从已部署的资源迁移时的注意事项

扫描限制

在扫描环境中寻找资源时，IaC 生成器对其可以检索的数据有特定的限制，并且在扫描时有配额限制。要了解更多信息，请参阅AWS CloudFormation 用户指南中的注意事项。

先决条件

在使用该`cdk migrate`命令之前，请执行以下操作：

1. 使用建立身份验证 AWS。有关说明，请参阅步骤 2：配置编程访问权限。
2. 安装或升级 AWS CDK CLI。有关安装说明，请参阅步骤 3：安装 AWS CDKCLI。

开始使用 CDK 迁移

首先，从您选择的目录中运行该 AWS CDK CLI`cdk migrate`命令。根据您正在执行的迁移类型，提供必填选项和可选选项。

有关可与之配合使用的选项的完整列表和说明`cdk migrate`，请参阅cdk migrate 命令参考。

以下是您可能需要提供的一些重要选项。

堆栈名称

唯一需要的选项是`--stack-name`。使用此选项为迁移后将在 AWS CDK 应用程序中创建的堆栈指定名称。部署时，堆栈名称也将用作 AWS CloudFormation 堆栈的名称。

Language

`--language`用于指定新 CDK 应用程序的编程语言。

AWS 账户和 AWS 区域

从默认来源 AWS CDK CLI检索 AWS 账户和 AWS 区域 信息。有关更多信息，请参阅 步骤 2：配置编程访问权限。您可以使用`--account`和`--region`选项`cdk migrate`来提供其他值。

新 CDK 项目的输出目录

默认情况下， AWS CDK CLI将在您的工作目录中创建一个新的 CDK 项目，并使用您提供的值`--stack-name`来命名该项目文件夹。如果当前存在同名文件夹，则 AWS CDK CLI会覆盖该文件夹。

您可以使用`--output-path`选项为新 CDK 项目文件夹指定不同的输出路径。

迁移来源

提供一个选项来指定您要迁移的来源。

- `--from-path`— 从本地 AWS CloudFormation 模板迁移。
- `--from-scan`— 从 AWS 账户中已部署的资源迁移和 AWS 区域。
- `--from-stack`— 从 AWS CloudFormation 堆栈迁移。

根据您的迁移源，您可以提供其他选项来自定义`cdk migrate`命令。

从 AWS CloudFormation 堆栈迁移

要从已部署的 AWS CloudFormation 堆栈迁移，请提供`--from-stack`选项。使用提供已部署 AWS CloudFormation 堆栈的名称`--stack-name`。以下是 示例：

```
$ cdk migrate --from-stack --stack-name "myCloudFormationStack"
```

AWS CDK CLI将执行以下操作：

1. 检索已部署堆栈的 AWS CloudFormation 模板。
2. 运行`cdk init`以初始化新的 CDK 应用程序。
3. 在 CDK 应用程序中创建一个包含已迁移 AWS CloudFormation 堆栈的堆栈。

当您从已部署的 AWS CloudFormation 堆栈迁移时，会 AWS CDK CLI尝试将已部署的资源逻辑 ID 和已部署的 AWS CloudFormation 堆栈名称与新 CDK 应用程序中迁移的资源和堆栈进行匹配。

迁移后，您可以正常管理和修改您的 CDK 应用程序。部署时，由于 AWS CloudFormation 堆栈名称匹配， AWS CloudFormation 会将部署标识为 AWS CloudFormation 堆栈更新。将更新具有匹配逻辑 ID 的资源。有关部署的更多信息，请参阅管理和部署您的 CDK 应用程序。

从 AWS CloudFormation 模板迁移

CDK Migrate 支持从格式为`JSON`或`YAML`的 AWS CloudFormation 模板迁移。

要从本地 AWS CloudFormation 模板迁移，请使用`--from-path`选项并提供本地模板的路径。您还必须提供所需的`--stack-name`选项。以下是 示例：

```
$ cdk migrate --from-path "./template.json" --stack-name "myCloudFormationStack"
```

AWS CDK CLI将执行以下操作：

1. 检索您的本地 AWS CloudFormation 模板。
2. 运行`cdk init`以初始化新的 CDK 应用程序。
3. 在 CDK 应用程序中创建包含已迁移 AWS CloudFormation 模板的堆栈。

迁移后，您可以正常管理和修改您的 CDK 应用程序。部署时，您可以选择以下选项：

- 更新 AWS CloudFormation 堆栈-如果之前部署了本地 AWS CloudFormation 模板，则可以更新已部署的 AWS CloudFormation 堆栈。
- 部署新 AWS CloudFormation 堆栈-如果从未部署过本地 AWS CloudFormation 模板，或者您想使用先前部署的模板创建新堆栈，则可以部署新 AWS CloudFormation 堆栈。

从 AWS SAM 模板迁移

要从 AWS Serverless Application Model (AWS SAM) 模板迁移，必须先将其转换为 AWS CloudFormation 模板或部署以创建 AWS CloudFormation 堆栈。

要将 AWS SAM 模板转换为 AWS CloudFormation，可以使用 AWS SAM CLI`sam validate --debug`命令。在运行此命令之前`lint`，您可能需要`false`在`samconfig.toml`文件中将其设置为。

要转换为 AWS CloudFormation 堆栈，请使用部署 AWS SAM 模板 AWS SAM CLI。然后从已部署的堆栈迁移。

从已部署的资源迁移

要从已部署的 AWS 资源迁移，请提供`--from-scan`选项。您还必须提供所需的`--stack-name`选项。以下是 示例：

```
$ cdk migrate --from-scan --stack-name "myCloudFormationStack"
```

AWS CDK CLI将执行以下操作：

1. 扫描您的帐户以获取资源和财产详细信息 — AWS CDK CLI 利用IaC生成器来扫描您的帐户并收集详细信息。

2. 生成 AWS CloudFormation 模板-扫描后， AWS CDK CLI使用 IaC 生成器创建 AWS CloudFormation 模板。
3. 初始化新的 CDK 应用程序并迁移您的模板 — AWS CDK CLI 运行`cdk init`初始化新 AWS CDK 应用程序，并将您的 AWS CloudFormation 模板作为单个堆栈迁移到 CDK 应用程序中。

使用过滤器

默认情况下， AWS CDK CLI将扫描整个 AWS 环境并迁移不超过 IaC 生成器的最大配额限制的资源。您可以为过滤器提供过滤器 AWS CDK CLI，以指定将资源从您的账户迁移到新 CDK 应用程序所依据的标准。要了解更多信息，请参阅`--filter`。

使用 iaC 生成器扫描资源

根据您账户中的资源数量，扫描可能需要几分钟。扫描过程中将显示一个进度条。

支持的资源类型

AWS CDK CLI将迁移 IaC 生成器支持的资源。有关完整列表，请参阅《AWS CloudFormation 用户指南》中的资源类型支持。

解析只写属性

一些支持的资源包含只写属性。可以写入这些属性来配置属性，但不能被 IaC 生成器读 AWS CloudFormation 取或获取值。例如，出于安全考虑，用于指定数据库密码的属性可能是只写的。

在迁移期间扫描资源时，IaC 生成器将检测可能包含只写属性的资源，并将其归类为以下任何类型：

- `MUTUALLY_EXCLUSIVE_PROPERTIES`— 这些是特定资源的只写属性，可以互换，用途相似。配置您的资源需要其中一个互斥的属性。例如，`AWS::Lambda::Function`资源的`S3BucketImageUri`、和`ZipFile`属性是互斥的只写属性。其中任何一个都可用于指定函数资产，但必须使用一个。
- `MUTUALLY_EXCLUSIVE_TYPES`— 这些是必需的只写属性，可接受多种配置类型。例如，`AWS::ApiGateway::RestApi`资源的`Body`属性接受对象或字符串类型。
- `UNSUPPORTED_PROPERTIES`— 这些是只写属性，不属于其他两个类别。它们要么是可选属性，要么是接受对象数组的必需属性。

有关只写属性以及 IaC 生成器在扫描已部署资源和创建 AWS CloudFormation 模板时如何管理这些属性的更多信息，请参阅《用户指南》中的 IaC 生成器和只写属性。AWS CloudFormation

迁移后，您必须在新 CDK 应用程序中指定只写属性值。 AWS CDK CLI将在 CDK 项目的ReadMe文件中追加警告部分，以记录 IaC 生成器识别的所有只写属性。以下是 示例：

```
# Welcome to your CDK TypeScript project
...
## Warnings
### Write-only properties
Write-only properties are resource property values that can be written to but can't be
 read by AWS CloudFormation or CDK Migrate. For more information, see [IaC generator
 and write-only properties](https://docs.aws.amazon.com/AWSCloudFormation/latest/
UserGuide/generate-IaC-write-only-properties.html).

Write-only properties discovered during migration are organized here by resource ID and
 categorized by write-only property type. Resolve write-only properties by providing
 property values in your CDK app. For guidance, see [Resolve write-only properties]
(https://docs.aws.amazon.com/cdk/v2/guide/migrate.html#migrate-resources-writeonly).
### MyLambdaFunction
- **UNSUPPORTED_PROPERTIES**:
  - SnapStart/ApplyOn: Applying SnapStart setting on function resource type.Possible
 values: [PublishedVersions, None]
This property can be replaced with other types
  - Code/S3ObjectVersion: For versioned objects, the version of the deployment package
 object to use.
This property can be replaced with other exclusive properties
- **MUTUALLY_EXCLUSIVE_PROPERTIES**:
  - Code/S3Bucket: An Amazon S3 bucket in the same AWS Region as your function. The
 bucket can be in a different AWS account.
This property can be replaced with other exclusive properties
  - Code/S3Key: The Amazon S3 key of the deployment package.
This property can be replaced with other exclusive properties
```

- 警告按标题进行组织，标识与之关联的资源的逻辑 ID。
- 警告按类型分类。这些类型直接来自 IaC 生成器。

解析只写属性

1. 从 CDK 项目文件的 “警告” 部分确定要解决的只写属性。ReadMe在这里，您可以记下 CDK 应用程序中可能包含只写属性的资源，并识别发现的只写属性类型。

a. 对于MUTUALLY_EXCLUSIVE_PROPERTIES，请确定要在您的 AWS CDK 应用程序中配置哪个互斥属性。

b. 对于MUTUALLY_EXCLUSIVE_TYPES，请确定您将使用哪种可接受的类型来配置该属性。

c. 对于UNSUPPORTED_PROPERTIES，确定该属性是可选的还是必需的。然后，根据需要进行配置。

2. 使用 IaC 生成器和只写属性的指导来引用警告类型的含义。

3. 在您的 CDK 应用程序中，还将在应用程序的Props部分中指定要解析的只写属性值。请在此处提供正确的值。有关属性描述和指导，您可以参考 AWS CDK API 参考。

   以下是迁移后的 CDK 应用程序中该Props部分的示例，其中包含两个需要解析的只写属性：

```
export interface MyTestAppStackProps extends cdk.StackProps {
  /**
   * The Amazon S3 key of the deployment package.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Keym8P82: string;
  /**
   * An Amazon S3 bucket in the same AWS Region as your function. The bucket can be
 in a different AWS account.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Bucketzidw8: string;
}
```

解析完所有只写属性值后，就可以为部署做好准备了。

迁移.json 文件

迁移期间会在您的 AWS CDK 项目中 AWS CDK CLI创建一个migrate.json文件。此文件包含有关已部署资源的参考信息。首次部署 CDK 应用程序时，会 AWS CDK CLI使用此文件引用已部署的资源，将您的资源与新 AWS CloudFormation 堆栈关联起来，然后删除该文件。

管理和部署您的 CDK 应用程序

迁移到时 AWS CDK，新的 CDK 应用程序可能无法立即部署就绪。本主题介绍在管理和部署新 CDK 应用程序时需要考虑的操作项目。

准备部署

在部署之前，您必须准备好您的 CDK 应用程序。

合成您的应用程序

使用`cdk synth`命令将 CDK 应用程序中的堆栈合成一个模板。 AWS CloudFormation

如果您从已部署的 AWS CloudFormation 堆栈或模板迁移，则可以将合成后的模板与迁移的模板进行比较，以验证资源和属性值。

要了解有关 `cdk synth` 的更多信息，请参阅合成堆栈 。

执行差异

如果您从已部署的 AWS CloudFormation 堆栈迁移，则可以使用 cdk diff 命令与新 CDK 应用程序中的堆栈进行比较。

要了解有关 cdk 差异的更多信息，请参阅。比较堆栈

引导您的环境

如果您是第一次从 AWS 环境进行部署，请使用`cdk bootstrap`来准备您的环境。要了解更多信息，请参阅正在引导。

部署你的 CDK 应用程序

当您部署 CDK 应用程序时， AWS CDK CLI会利用该 AWS CloudFormation 服务来配置您的资源。资源在 CDK 应用程序中捆绑到单个堆栈中，并作为单个 AWS CloudFormation 堆栈进行部署。

根据您从何处迁移，您可以进行部署以创建新 AWS CloudFormation 堆栈或更新现有 AWS CloudFormation 堆栈。

部署以创建新 AWS CloudFormation 堆栈

如果您从已部署的资源迁移，则 AWS CDK CLI将在部署时自动创建一个新 AWS CloudFormation 堆栈。您部署的资源将包含在新 AWS CloudFormation 堆栈中。

如果您从从未部署过的本地 AWS CloudFormation 模板迁移，则 AWS CDK CLI将在部署时自动创建一个新 AWS CloudFormation 堆栈。

如果您从先前部署的已部署 AWS CloudFormation 堆栈或本地 AWS CloudFormation 模板迁移，则可以进行部署以创建新的 AWS CloudFormation 堆栈。要创建新堆栈，请执行以下操作：

- 部署到新 AWS 环境。这包括使用不同的 AWS 账户或部署到不同的账户 AWS 区域。
- 如果要将新堆栈部署到迁移后的堆栈或模板的相同 AWS 环境中，则必须将 CDK 应用程序中的堆栈名称修改为新值。您还必须修改 CDK 应用程序中资源的所有逻辑 ID。然后，您可以部署到同一环境以创建新堆栈和新资源。

部署以更新现有 AWS CloudFormation 堆栈

如果您从先前部署的已部署 AWS CloudFormation 堆栈或本地 AWS CloudFormation 模板迁移，则可以进行部署以更新现有 AWS CloudFormation 堆栈。

验证 CDK 应用程序中的堆栈名称是否与已 AWS CloudFormation 部署堆栈的堆栈名称相匹配，然后部署到相同的 AWS 环境中。

AWS CDK 在支持的编程语言中使用

使用使用支持的编程语言 AWS Cloud Development Kit (AWS CDK)来定义您的 AWS Cloud 基础架构。

主题

导入 AWS 构造库

AWS CDK 包括 AWS 构造库，这是按 AWS 服务组织的构造集合。该库的稳定构造是在单个模块中提供的，该模块由其TypeScript软件包名称调用：`aws-cdk-lib`。实际的软件包名称因语言而异。

TypeScript

安装	`npm ## aws-cdk-lib`
导入	`const cdk = require ('aws-cdk-lib')#`

JavaScript

安装	`npm ## aws-cdk-lib`
导入	`const cdk = require ('aws-cdk-lib')#`

Python

安装	`python -m pip ## aws-cdk-lib`
导入	`# aws_cdk ## cdk ##`

Java

添加到 `pom.xml`	Group `##.amazon.awscdk` ; artifact `aws-cdk-lib`
导入	`####.amazon.awscdk.app#` (for example)

C#

安装	`dotnet #### Amazon.cdk.lib`
导入	`## Amazon.cdk#`

`construct`基类和支持代码在`constructs`模块中。实验构造（API 仍在完善中）作为单独的模块分发。

AWS CDK API 参考资料

AWS CDK API 参考提供了库中构造（和其他组件）的详细文档。为每种支持的编程语言提供了 API 参考版本。

每个模块的参考资料分为以下几节。

- 概述：使用中的服务需要了解的入门材料 AWS CDK，包括概念和示例。
- 构造：表示一个或多个具体 AWS 资源的库类。这些是“精选”（L2）资源或模式（L3 资源），它们提供了具有合理默认值的高级接口。
- 类：非构造类，提供模块中构造使用的功能。
- 结构：定义复合值结构的数据结构（属性包），例如属性（构造的`props`参数）和选项。

- 接口：名称均以 “I” 开头的接口，定义相应构造或其他类的绝对最低功能。CDK 使用构造接口来表示在您的 AWS CDK 应用程序外部定义并由诸如`Bucket.fromBucketArn()`之类的方法引用的 AWS 资源。
- 枚举：用于指定某些构造参数的命名值的集合。使用枚举值允许 CDK 在合成过程中检查这些值的有效性。
- CloudFormation 资源：这些 L1 结构的名称以 “Cfn” 开头，它们完全代表规范中定义的资源。CloudFormation 它们是在每个 CDK 版本中根据该规范自动生成的。每个 L2 或 L3 结构都封装了一个或多个资源。 CloudFormation
- CloudFormation 属性类型：定义每个 CloudFormation 资源属性的命名值的集合。

接口与构造类的比较

它们以一种特定的方式 AWS CDK 使用接口，即使您熟悉接口作为编程概念，这种方式也可能并不明显。

AWS CDK 支持使用诸如之类的方法使用 CDK 应用程序外部定义的资源。`Bucket.fromBucketArn()`外部资源无法修改，也可能无法使用`Bucket`类等在您的 CDK 应用程序中定义的资源的所有可用功能。因此，接口代表 CDK 中针对给定 AWS 资源类型（包括外部资源）可用的最低限度功能。

那么，在 CDK 应用程序中实例化资源时，应始终使用具体的类，例如. `Bucket` 在您自己的构造中指定要接受的参数类型时，请使用接口类型，例如`IBucket`您准备好处理外部资源（也就是说，您无需更改它们）。如果您需要 CDK 定义的构造，请指定可以使用的最通用的类型。

有些接口是与特定类相关联的属性或选项包的最低版本，而不是构造。当子类化以接受要传递给父类的参数时，这样的接口可能很有用。如果您需要一个或多个其他属性，则需要从该接口或更具体的类型中实现或派生。

> **Note**
>
> 支持的某些编程语言 AWS CDK 没有接口功能。在这些语言中，接口只是普通的类。你可以通过它们的名字来识别它们，名称遵循首字母 “I” 的模式，然后是其他构造的名称（例如`IBucket`）。同样的规则适用。

管理依赖关系

您的 AWS CDK 应用程序或库的依赖项使用包管理工具进行管理。这些工具通常与编程语言一起使用。

通常， AWS CDK 支持该语言的标准或官方软件包管理工具（如果有）。否则， AWS CDK 将支持该语言最受欢迎或最广泛支持的语言。您也可以使用其他工具，特别是当它们与支持的工具配合使用时。但是，官方对其他工具的支持是有限的。

AWS CDK 支持以下软件包管理器：

Language	支持的软件包管理工具
TypeScript/JavaScript	NPM（Node Package Manager）或 Yarn
Python	PIP（适用于 Python 的 Package 安装程序）
Java	Maven
C#	NuGet
Go	Go 模块

使用 AWS CDK CLI`cdk init`命令创建新项目时，系统会自动指定 CDK 核心库和稳定构造的依赖关系。

有关管理支持的编程语言依赖关系的更多信息，请参阅以下内容：

- 在中管理依赖关系 TypeScript.
- 在中管理依赖关系 JavaScript.
- 在中管理依赖关系 Python.
- 在中管理依赖关系 Java.
- 在中管理依赖关系 C#.
- 在中管理依赖关系 Go.

与其他 AWS CDK 语言TypeScript进行比较

TypeScript 是开发 AWS CDK 应用程序时支持的第一种语言。因此，编写了大量的 CDK 示例代码。TypeScript如果您正在使用另一种语言进行开发，那么将 AWS CDK 代码的实现方式 TypeScript 与您选择的语言进行比较可能会很有用。这可以帮助你在整个文档中使用示例。

导入模块

TypeScript/JavaScript

TypeScript 支持导入整个命名空间或从命名空间导入单个对象。每个命名空间都包含用于给定 AWS 服务的构造和其他类。

```
// Import main CDK library as cdk
import * as cdk from 'aws-cdk-lib';   // ES6 import preferred in TS
const cdk = require('aws-cdk-lib');   // Node.js require() preferred in JS

// Import specific core CDK classes
import { Stack, App } from 'aws-cdk-lib';
const { Stack, App } = require('aws-cdk-lib');

// Import AWS S3 namespace as s3 into current namespace
import { aws_s3 as s3 } from 'aws-cdk-lib';   // TypeScript
const s3 = require('aws-cdk-lib/aws-s3');     // JavaScript

// Having imported cdk already as above, this is also valid
const s3 = cdk.aws_s3;

// Now use s3 to access the S3 types
const bucket = s3.Bucket(...);

// Selective import of s3.Bucket
import { Bucket } from 'aws-cdk-lib/aws-s3';        // TypeScript
const { Bucket } = require('aws-cdk-lib/aws-s3');   // JavaScript

// Now use Bucket to instantiate an S3 bucket
const bucket = Bucket(...);
```

Python

比如 TypeScript，Python 支持命名空间模块导入和选择性导入。Python 中的命名空间看起来像 aws_cdk。 xxx，其中 xxx 表示 AWS 服务名称，例如亚马逊 S3 的 s3。（这些示例中使用了 Amazon S3）。

```
# Import main CDK library as cdk
import aws_cdk as cdk

# Selective import of specific core classes
from aws_cdk import Stack, App

# Import entire module as s3 into current namespace
import aws_cdk.aws_s3 as s3

# s3 can now be used to access classes it contains
bucket = s3.Bucket(...)

# Selective import of s3.Bucket into current namespace
from aws_cdk.s3 import Bucket

# Bucket can now be used to instantiate a bucket
bucket = Bucket(...)
```

Java

Java 的导入工作方式与 TypeScript's 不同。每个 import 语句要么从给定包中导入单个类名，要么导入该包中定义的所有类（使用*）。可以单独使用类名（如果已导入）或包括其包在内的限定类名来访问类。

库的命名与 AWS 构造库类似`software.amazon.awscdk.services.xxx`（主库是`software.amazon.awscdk`）。 AWS CDK 软件包的 Maven 群组 ID 是`software.amazon.awscdk`。

```
// Make certain core classes available
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.App;

// Make all Amazon S3 construct library classes available
import software.amazon.awscdk.services.s3.*;

// Make only Bucket and EventType classes available
```

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that
 name
// does not conflict with another class)
Bucket bucket = Bucket.Builder.create(...).build();

// We can always use the qualified name of a class (including its package) even
 without an
// import directive
software.amazon.awscdk.services.s3.Bucket bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();

// Java 10 or later can use var keyword to avoid typing the type twice
var bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();
```

C#

在 C# 中，您可以使用`using`指令导入类型。有两种风格。一个允许你使用普通名称访问指定命名空间中的所有类型。对于另一个，你可以使用别名来引用命名空间本身。

包的命名与 AWS 构`Amazon.CDK.AWS.xxx`造库包类似。（核心模块是`Amazon.CDK`。）

```
// Make CDK base classes available under cdk
using cdk = Amazon.CDK;

// Make all Amazon S3 construct library classes available
using Amazon.CDK.AWS.S3;

// Now we can access any S3 type using its name
var bucket = new Bucket(...);

// Import the S3 namespace under an alias
using s3 = Amazon.CDK.AWS.S3;

// Now we can access an S3 type through the namespace alias
var bucket = new s3.Bucket(...);

// We can always use the qualified name of a type (including its namespace) even
 without a
```

```
// using directive
var bucket = new Amazon.CDK.AWS.S3.Bucket(...)
```

Go

每个 AWS 构造库模块都以 Go 包的形式提供。

```
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

// now instantiate a bucket
bucket := awss3.NewBucket(...)

// use aliases for brevity/clarity
import (
    cdk "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    s3  "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

bucket := s3.NewBucket(...)
```

实例化构造

AWS CDK 构造类在所有支持的语言中都具有相同的名称。大多数语言都使用new关键字来实例化类（Python 和 Go 不使用）。此外，在大多数语言中，关键字this指的是当前实例。（self按照惯例，Python 使用。）您应该将对当前实例的引用作为scope参数传递给您创建的每个构造。

AWS CDK 构造的第三个参数是props，一个包含构建构造所需的属性的对象。这个参数可能是可选的，但是当需要时，支持的语言会以惯用方式处理它。属性的名称也适用于该语言的标准命名模式。

TypeScript/JavaScript

```
// Instantiate default Bucket
const bucket = new s3.Bucket(this, 'MyBucket');

// Instantiate Bucket with bucketName and versioned properties
const bucket = new s3.Bucket(this, 'MyBucket', {
```

```
  bucketName: 'my-bucket',
   versioned: true,
});

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  websiteRedirect: {host: 'aws.amazon.com'}});
```

Python

Python 在实例化类时不使用new关键字。属性参数使用关键字参数表示，参数使用命名snake_case。

如果 props 值本身就是一组属性，则它由一个以该属性命名的类表示，该类接受子属性的关键字参数。

在 Python 中，当前实例作为第一个参数传递给方法，self按照惯例命名。

```
# Instantiate default Bucket
bucket = s3.Bucket(self, "MyBucket")

# Instantiate Bucket with bucket_name and versioned properties
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket", versioned=true)

# Instantiate Bucket with website_redirect, which has its own sub-properties
bucket = s3.Bucket(self, "MyBucket", website_redirect=s3.WebsiteRedirect(
            host_name="aws.amazon.com"))
```

Java

在 Java 中，props 参数由一个名为的类表示XxxxProps（例如，BucketProps用于Bucket构造的 props）。你可以使用生成器模式构建 props 参数。

每个XxxxProps类都有一个生成器。每个构造都有一个方便的构建器，可以一步构建 props 和构造，如以下示例所示。

道具的命名与中的相同 TypeScript，使用camelCase。

```
// Instantiate default Bucket
Bucket bucket = Bucket(self, "MyBucket");
```

```
// Instantiate Bucket with bucketName and versioned properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket").versioned(true)
                      .build();

# Instantiate Bucket with websiteRedirect, which has its own sub-properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .websiteRedirect(new websiteRedirect.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

在 C# 中，prop 是使用名为的类的对象初始化器指定的`XxxxProps`（例如，`BucketProps`用于`Bucket`构造的 props）。

道具的命名方式与类似 TypeScript，唯一的不同是使用`PascalCase`。

在实例化构造时使用`var`关键字很方便，因此无需键入类名两次。但是，您的本地代码风格指南可能会有所不同。

```
// Instantiate default Bucket
var bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with BucketName and Versioned properties
var bucket =  Bucket(self, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
var bucket = Bucket(self, "MyBucket", new BucketProps {
                      WebsiteRedirect = new WebsiteRedirect {
                              HostName = "aws.amazon.com"
                      }});
```

Go

要在 Go 中创建构造，请调用函数，`NewXxxxxx`其中`Xxxxxxx`是构造的名称。构造的属性被定义为一个结构。

在 Go 中，所有构造参数都是指针，包括数字、布尔值和字符串等值。使用诸如创建这些指针之类`jsii.String`的便捷函数。

```
// Instantiate default Bucket
bucket := awss3.NewBucket(stack, jsii.String("MyBucket"), nil)

// Instantiate Bucket with BucketName and Versioned properties
bucket1 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 BucketName: jsii.String("my-bucket"),
 Versioned:  jsii.Bool(true),
})

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
bucket2 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 WebsiteRedirect: &awss3.RedirectTarget{
  HostName: jsii.String("aws.amazon.com"),
 }})
```

访问成员

通常会引用构造和其他 AWS CDK 类的属性或属性，并将这些值用作构建其他构造的输入。前面描述的方法命名差异也适用于此处。此外，在 Java 中，不可能直接访问成员。相反，提供了一个获取器方法。

TypeScript/JavaScript

名字是`camelCase`。

```
bucket.bucketArn
```

Python

名字是`snake_case`。

```
bucket.bucket_arn
```

Java

为每个属性提供了一个 getter 方法；这些名称是`camelCase`。

```
bucket.getBucketArn()
```

C#

名字是PascalCase。

```
bucket.BucketArn
```

Go

名字是PascalCase。

```
bucket.BucketArn
```

枚举常量

枚举常量的作用域仅限于一个类，并且在所有语言中都具有大写名称和下划线（有时称为）。SCREAMING_SNAKE_CASE由于类名在除 Go 之外的所有支持的语言中也使用相同的大小写，因此这些语言中的限定枚举名称也相同。

```
s3.BucketEncryption.KMS_MANAGED
```

在 Go 中，枚举常量是模块命名空间的属性，其写法如下。

```
awss3.BucketEncryption_KMS_MANAGED
```

对象接口

AWS CDK 使用 TypeScript 对象接口来表示一个类实现了一组预期的方法和属性。您可以识别对象接口，因为它的名称以开头I。具体类使用implements关键字表示它实现的接口。

TypeScript/JavaScript

> **Note**
>
> JavaScript 没有界面功能。您可以忽略implements关键字及其后面的类名。

```
import { IAspect, IConstruct } from 'aws-cdk-lib';
```

```
class MyAspect implements IAspect {
  public visit(node: IConstruct) {
    console.log('Visited', node.node.path);
  }
}
```

Python

Python 没有接口功能。但是，对于这些来说， AWS CDK 你可以通过装饰你的类来表示接口的`@jsii.implements(interface)`实现。

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
  def visit(self, node: IConstruct) -> None:
    print("Visited", node.node.path)
```

Java

```
import software.amazon.awscdk.IAspect;
import software.amazon.awscdk.IConstruct;

public class MyAspect implements IAspect {
    public void visit(IConstruct node) {
        System.out.format("Visited %s", node.getNode().getPath());
    }
}
```

C#

```
using Amazon.CDK;

public class MyAspect : IAspect
{
    public void Visit(IConstruct node)
    {
        System.Console.WriteLine($"Visited ${node.Node.Path}");
    }
}
```

Go

Go 结构不需要显式声明它们实现了哪些接口。Go 编译器根据结构上可用的方法和属性来确定实现。例如，在以下代码中，`MyAspect`实现该`IAspect`接口，因为它提供了一种采用构造`Visit`的方法。

```
type MyAspect struct {
}

func (a MyAspect) Visit(node constructs.IConstruct) {
 fmt.Println("Visited", *node.Node().Path())
}
```

使用 AWS CDK in TypeScript

TypeScript 是完全支持的客户端语言，被认为是 AWS Cloud Development Kit (AWS CDK) 稳定的。TypeScript 使用 AWS CDK in 使用熟悉的工具，包括微软的 TypeScript 编译器 (`tsc`)、Node.js 和 Node Package Manager (`npm`)。如果你愿意，也可以使用 Yarn，尽管本指南中的示例使用 NPM。构成 AWS 构造库的模块通过 NPM 存储库 npmjs.org 分发。

您可以使用任何编辑器或 IDE。许多 AWS CDK 开发人员使用 V isual Studio Code（或其开源等效物 vsCodium），它具有出色的支持。 TypeScript

主题

- 开始使用 TypeScript
- 创建项目
- 使用本地tsc和 cdk
- 管理 AWS 构造库模块
- 在中管理依赖关系 TypeScript
- AWS CDK 中的成语 TypeScript
- 构建、合成和部署

开始使用 TypeScript

要使用 AWS CDK，您必须拥有 AWS 账户和凭证，并已安装 Node.js 和 AWS CDK Toolkit。请参阅开始使用 AWS CDK。

您还需要 TypeScript 自身（3.8 或更高版本）。如果您还没有，则可以使用进行安装npm。

```
npm install -g typescript
```

> **Note**
>
> 如果您遇到权限错误，并且在系统上拥有管理员访问权限，请尝试sudo npm install -g typescript。

随时了解 TypeScript 最新动态npm update -g typescript.

> **Note**
>
> 第三方语言弃用：语言版本仅在供应商或社区共享的 EOL（生命周期结束）之前才受支持，如有更改，恕不另行通知。

创建项目

您可以通过在空目录cdk init中调用来创建新 AWS CDK 项目。使用该--language选项并指定typescript：

```
mkdir my-project
cd my-project
cdk init app --language typescript
```

创建项目还会安装aws-cdk-lib模块及其依赖关系。

cdk init使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 TypeScript 标识符的形式；例如，名称不应以数字开头或包含空格。

使用本地tsc和 cdk

在大多数情况下，本指南假设您在全局安装 TypeScript CDK Toolkit (npm install -g typescript aws-cdk)，并且提供的命令示例（例如cdk synth）遵循此假设。这种方法可以很容

易地使两个组件保持最新状态，并且由于两者都对向后兼容性采取了严格的方法，因此始终使用最新版本通常风险很小。

有些团队更喜欢在每个项目中指定所有依赖关系，包括 TypeScript 编译器和 CDK Toolkit 等工具。这种做法允许您将这些组件固定到特定版本，并确保团队中的所有开发人员（以及您的 CI/CD 环境）都完全使用这些版本。这消除了可能的变更来源，有助于使构建和部署更加一致和可重复。

CDK 在 TypeScript 项目模板中包含两者的依赖关系 TypeScript 和 CDK Toolkit`package.json`，因此，如果您想使用这种方法，则无需对项目进行任何更改。你所需要做的就是使用稍微不同的命令来构建应用程序和发出`cdk`命令。

操作	使用全局工具	使用本地工具
初始化项目	`cdk init --language typescript`	`npx aws-cdk init --language typescript`
构建	`tsc`	`npm run build`
运行 CDK 工具包命令	`cdk ...`	`npm run cdk ...` or `npx aws-cdk ...`

`npx aws-cdk`运行当前项目中本地安装的 CDK Toolkit 版本（如果有），则回退到全局安装（如果有）。如果不存在全局安装，则`npx`下载 CDK Toolkit 的临时副本并运行该副本。您可以使用`@`语法指定 CDK 工具包的任意版本：pr `npx aws-cdk@2.0 --version` in `2.0.0` ts。

> **Tip**
>
> 设置别名，这样你就可以在安装本地 CDK Toolkit 时使用该`cdk`命令。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

管理 AWS 构造库模块

使用 Node Package Manager (npm) 安装和更新 C AWS onstruct Library 模块以供您的应用程序使用，以及您需要的其他软件包。（npm如果你愿意，你可以yarn改用。）npm还会自动安装这些模块的依赖关系。

大多数 AWS CDK 构造都位于名为 CDK 的主包中aws-cdk-lib，这是由创建的新项目中的默认依赖项。cdk init“实验性” AWS 构造库模块（其中更高级别的构造仍在开发中）被命名为。@aws-cdk/*SERVICE-NAME*-alpha服务名称带有 a ws- 前缀。如果您不确定某个模块的名称，请在 NPM 上进行搜索。

> **Note**
>
> CDK API 参考还显示了软件包名称。

例如，以下命令安装的实验模块 AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

某些服务的构造库支持位于多个命名空间中。例如，此外aws-route53，还有另外三个 Amazon Route 53 命名空间，aws-route53-targetsaws-route53-patterns、和。aws-route53resolver

您的项目的依赖关系在中维护package.json。您可以编辑此文件以将部分或全部依赖项锁定到特定版本，或者允许在特定条件下将其更新到较新的版本。要根据您在以下中指定的规则，将项目的 NPM 依赖项更新到允许的最新版本：package.json

```
npm update
```

在中 TypeScript，您可以将模块导入到代码中，其名称与使用 NPM 安装模块时使用的名称相同。在应用程序中导入 AWS CDK 类和 AWS 构造库模块时，我们建议采用以下做法。遵循这些准则将有助于使您的代码与其他 AWS CDK 应用程序保持一致并更易于理解。

- 请使用 ES6 风格的import指令，不是。require()
- 通常，从中导入单个类aws-cdk-lib。

```
import { App, Stack } from 'aws-cdk-lib';
```

- 如果您需要来自的许多类aws-cdk-lib，则可以使用命名空间别名来cdk代替导入各个类。避免两者兼而有之。

```
import * as cdk from 'aws-cdk-lib';
```

- 通常，导入 AWS 服务结构使用短命名空间别名。

```
import { aws_s3 as s3 } from 'aws-cdk-lib';
```

在中管理依赖关系 TypeScript

在 TypeScript CDK 项目中，依赖关系是在项目主目录package.json的文件中指定的。核心 AWS CDK 模块位于名为的单个NPM包中aws-cdk-lib。

当你使用安装软件包时npm install，NPM 会package.json为你记录该软件包。

如果你愿意，你可以用 Yarn 代替 NPM。但是，CDK 不支持 Yarn plug-and-play 模式，这是 Yarn 2 中的默认模式。将以下内容添加到您的项目.yarnrc.yml文件中以关闭此功能。

```
nodeLinker: node-modules
```

CDK 应用程序

以下是该cdk init --language typescript命令生成的示例package.json文件：

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
```

```
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

对于可部署的 CDK 应用程序，aws-cdk-lib必须在的dependencies部分中指定。package.json你可以使用尖号 (^) 版本号说明符来表示你将接受比指定版本更高的版本，前提是它们位于同一个主版本内。

对于实验构造，请为 alpha 构造库模块指定确切版本，这些模块的 API 可能会发生变化。请勿使用 ^ 或 ~，因为这些模块的更高版本可能会带来的 API 更改，从而导致您的应用程序中断。

在的devDependencies部分中指定测试您的应用程序所需的库和工具版本（例如，jest测试框架）package.json。（可选）使用 ^ 指定可接受更高版本的兼容版本。

第三方构造库

如果您正在开发构造库，请使用peerDependencies和devDependencies部分的组合来指定其依赖关系，如以下示例package.json文件所示。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
```

```
    "aws-cdk": "^2.14.0"
  }
}
```

在中peerDependencies，使用尖号 (^) 指定您的库aws-cdk-lib所使用的最低版本。这样可以最大限度地提高您的库与一系列 CDK 版本的兼容性。为 alpha 构造库模块指定确切版本，这些模块的 API 可能会发生变化。使用peerDependencies可以确保node_modules树中只有一个所有 CDK 库的副本。

在中devDependencies，指定测试所需的工具和库，也可以使用 ^ 表示可以接受更高版本的兼容版本。准确指定（不带 ^ 或 ~）你宣传的库aws-cdk-lib与之兼容的其他 CDK 软件包的最低版本和其他 CDK 软件包。这种做法可确保您的测试针对这些版本运行。这样，如果您无意中使用了仅在新版本中找到的功能，则您的测试可以捕捉到它。

> **Warning**
>
> peerDependencies仅由 NPM 7 及更高版本自动安装。如果您使用的是 NPM 6 或更早版本，或者使用的是 Yarn，则必须在中devDependencies包含依赖项的依赖关系。否则，它们将无法安装，并且您将收到有关未解决的对等依赖关系的警告。

安装和更新依赖关系

运行以下命令来安装项目的依赖项。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
```

```
yarn install --frozen-lockfile
```

要更新已安装的模块，可以使用前面的npm install和yarn upgrade命令。任一命令都会将软件包更新node_modules到满足中规则的最新版本package.json。但是，它们不会package.json自行更新，您可能需要这样做来设置新的最低版本。如果您在上托管软件包 GitHub，则可以将 De pendabot 版本更新配置为自动更新。package.json或者，请使用 npm-check-updates。

> **Important**
>
> 根据设计，当您安装或更新依赖项时，NPM 和 Yarn 会选择满足中指定要求的每个软件包的最新版本。package.json这些版本总是存在损坏的风险（无论是意外还是故意）。更新项目的依赖关系后，请进行全面测试。

AWS CDK 中的成语 TypeScript

道具

所有 C AWS onstruct Library 类都使用三个参数进行实例化：定义构造的作用域（构造树中的父级）、id 和 props。Argument props 是一组键/值对，构造使用这些键/值对来配置其创建的 AWS 资源。其他类和方法也使用 “属性包” 模式作为参数。

在中 TypeScript，的形状props是使用一个接口定义的，该接口告诉你必填参数和可选参数及其类型。这样的接口是为每种props参数定义的，通常特定于单个构造或方法。例如，Bucket 构造（在aws-cdk-lib/aws-s3 module）指定了一个符合BucketProps接口的props参数。

如果一个属性本身就是一个对象，例如的 WebsiteRedire ct 属性BucketProps，则该对象将拥有自己的接口，在这种情况下 RedirectTarget，其形状必须符合该接口。

如果要子类化一个 C AWS onstruct Library 类（或重写采用类似 props 的参数的方法），则可以从现有接口继承来创建一个新接口来指定代码所需的任何新道具。在调用父类或基方法时，通常可以传递收到的整个 props 参数，因为对象中提供但未在接口中指定的任何属性都将被忽略。

future 版本 AWS CDK 可能会巧合地添加一个新属性，其名称是你用于自己的财产。然后，将您收到的值向上传递继承链可能会导致意外行为。如果你的属性被移除或设置为，那么传递一份你收到的道具的浅层副本会更安全。undefined例如：

```
super(scope, name, {...props, encryptionKeys: undefined});
```

或者，请命名您的属性，使其清楚地表明它们属于您的构造。这样，它们就不太可能在 future AWS CDK 版本中与属性发生冲突。如果其中有很多，请使用一个适当命名的对象来存放它们。

缺失值

对象（例如 props）中的缺失值具有`undefined`中的值。 TypeScript该语言的3.7版本引入了简化处理这些值的运算符，从而更容易指定默认值，并在达到未定义的值时使用 “短路” 链接。有关这些功能的更多信息，请参阅 TypeScript 3.7 版本说明，特别是前两个功能，可选链接和 Nullish Coalescing。

构建、合成和部署

通常，在构建和运行应用程序时，您应该位于项目的根目录中。

Node.js 无法 TypeScript 直接运行；相反，您的应用程序会转换为 JavaScript 使用编 TypeScript译器`tsc`。然后执行生成的 JavaScript 代码。

每当需要运行您的应用程序时，它都会 AWS CDK 自动执行此操作。但是，手动编译以检查错误和运行测试可能很有用。要手动编译您的 TypeScript 应用程序，请发出`npm run build`。您也可以发出`npm run watch`进入监视模式的命令，在这种模式下，每当您保存对源文件所做的更改时，TypeScript编译器都会自动重建您的应用程序。

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- `cdk synth`：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- `cdk deploy`：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
```

```
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

使用 AWS CDK in JavaScript

JavaScript是完全支持的客户端语言，被认为是 AWS CDK 稳定的。在 in AWS Cloud Development Kit (AWS CDK) 中 JavaScript 使用熟悉的工具，包括 Node.js 和 Node Package Manager (npm)。如果你愿意，也可以使用 Yarn，尽管本指南中的示例使用 NPM。构成 AWS 构造库的模块通过 NPM 存储库 npmjs.org 分发。

您可以使用任何编辑器或 IDE。许多 AWS CDK 开发人员使用 V isual Studio Code（或其开源等效物 vsCodium），它有很好的支持。 JavaScript

主题

开始使用 JavaScript

要使用 AWS CDK，您必须拥有 AWS 账户和凭据并已安装 Node.js 和 AWS CDK Toolkit。请参阅 开始使用 AWS CDK。

JavaScript AWS CDK 除这些条件外，应用程序不需要其他先决条件。

> **Note**
>
> 第三方语言弃用：语言版本仅在供应商或社区共享的 EOL（生命周期结束）之前才受支持，如有更改，恕不另行通知。

创建项目

您可以通过在空目录`cdk init`中调用来创建新 AWS CDK 项目。使用该`--language`选项并指定`javascript`：

```
mkdir my-project
cd my-project
cdk init app --language javascript
```

创建项目还会安装`aws-cdk-lib`模块及其依赖关系。

`cdk init`使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 JavaScript 标识符的形式；例如，名称不应以数字开头或包含空格。

使用本地 cdk

在大多数情况下，本指南假设您全局安装 CDK Toolkit (`npm install -g aws-cdk`)，并且提供的命令示例（例如`cdk synth`）遵循此假设。这种方法可以轻松保持 CDK Toolkit 的最新状态，而且由于 CDK 对向后兼容采取了严格的方法，因此始终使用最新版本通常风险不大。

有些团队更喜欢在每个项目中指定所有依赖关系，包括诸如 CDK Toolkit 之类的工具。这种做法允许您将此类组件固定到特定版本，并确保团队中的所有开发人员（以及您的 CI/CD 环境）都完全使用这些版本。这消除了可能的变更来源，有助于使构建和部署更加一致和可重复。

CDK 在 JavaScript 项目模板中包含对 CDK Toolkit 的依赖关系`package.json`，因此，如果您想使用这种方法，则无需对项目进行任何更改。你所需要做的就是使用稍微不同的命令来构建应用程序和发出`cdk`命令。

操作	使用全球 CDK 工具包	使用本地 CDK 工具包
初始化项目	`cdk init--## javascrip t`	`npx aws-cdk init – ## javascript`

运行 CDK 工具包命令	`cdk...`	`npm ## cdk...` or `npx aws-cdk...`

`npx aws-cdk`运行当前项目中本地安装的 CDK Toolkit 版本（如果有），则回退到全局安装（如果有）。如果不存在全局安装，则`npx`下载 CDK Toolkit 的临时副本并运行该副本。您可以使用`@`语法指定 CDK 工具包的任意版本：pr `npx aws-cdk@1.120 --version` in `1.120.0` ts。

> **Tip**
>
> 设置别名，这样你就可以在安装本地 CDK Toolkit 时使用该`cdk`命令。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

管理 AWS 构造库模块

使用 Node Package Manager (npm) 安装和更新 C AWS onstruct Library 模块以供您的应用程序使用，以及您需要的其他软件包。（`npm`如果你愿意，你可以`yarn`改用。）`npm`还会自动安装这些模块的依赖关系。

大多数 AWS CDK 构造都位于名为 CDK 的主包中`aws-cdk-lib`，这是由创建的新项目中的默认依赖项。cdk init“实验性” AWS 构造库模块（其中更高级别的构造仍在开发中）被命名为。`aws-cdk-lib/`*`SERVICE-NAME`*`-alpha`服务名称带有 a ws- 前缀。如果您不确定某个模块的名称，请在 NPM 上进行搜索。

> **Note**
>
> CDK API 参考还显示了软件包名称。

例如，以下命令安装的实验模块 AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

某些服务的构造库支持位于多个命名空间中。例如，此外aws-route53，还有另外三个 Amazon Route 53 命名空间，aws-route53-targetsaws-route53-patterns、和。aws-route53resolver

您的项目的依赖关系在中维护package.json。您可以编辑此文件以将部分或全部依赖项锁定到特定版本，或者允许在特定条件下将其更新到较新的版本。要根据您在以下中指定的规则，将项目的 NPM 依赖项更新到允许的最新版本：package.json

```
npm update
```

在中 JavaScript，您可以将模块导入到代码中，其名称与使用 NPM 安装模块时使用的名称相同。在应用程序中导入 AWS CDK 类和 AWS 构造库模块时，我们建议采用以下做法。遵循这些准则将有助于使您的代码与其他 AWS CDK 应用程序保持一致并更易于理解。

- 使用require()，而不是 ES6 风格的指令import。Node.js 的旧版本不支持 ES6 导入，因此使用较旧的语法更具兼容性。（如果你真的想使用 ES6 导入，请使用 esm 来确保你的项目与所有支持的 Node.js 版本兼容。）
- 通常，从中导入单个类aws-cdk-lib。

  ```
  const { App, Stack } = require('aws-cdk-lib');
  ```

- 如果您需要来自的许多类aws-cdk-lib，则可以使用命名空间别名来cdk代替导入各个类。避免两者兼而有之。

  ```
  const cdk = require('aws-cdk-lib');
  ```

- 通常，使用短命名空间别名导入 AWS 构造库。

  ```
  const { s3 } = require('aws-cdk-lib/aws-s3');
  ```

在中管理依赖关系 JavaScript

在 JavaScript CDK 项目中，依赖关系是在项目主目录package.json的文件中指定的。核心 AWS CDK 模块位于名为的单个NPM包中aws-cdk-lib。

当你使用安装软件包时npm install，NPM 会package.json为你记录该软件包。

如果你愿意，你可以用 Yarn 代替 NPM。但是，CDK 不支持 Yarn plug-and-play 模式，这是 Yarn 2 中的默认模式。将以下内容添加到您的项目`.yarnrc.yml`文件中以关闭此功能。

```
nodeLinker: node-modules
```

CDK 应用程序

以下是该`cdk init --language typescript`命令生成的示例`package.json`文件。为生成的文件 JavaScript 类似，只是没有 TypeScript相关的条目。

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

对于可部署的 CDK 应用程序，`aws-cdk-lib`必须在的`dependencies`部分中指定。`package.json`你可以使用尖号 (^) 版本号说明符来表示你将接受比指定版本更高的版本，前提是它们位于同一个主版本内。

对于实验构造，请为 alpha 构造库模块指定确切版本，这些模块的 API 可能会发生变化。请勿使用 ^ 或 ~，因为这些模块的更高版本可能会带来的 API 更改，从而导致您的应用程序中断。

在的devDependencies部分中指定测试您的应用程序所需的库和工具版本（例如jest测试框架）package.json。（可选）使用 ^ 来指定可以接受更高版本的兼容版本。

第三方构造库

如果您正在开发构造库，请使用peerDependencies和devDependencies部分的组合来指定其依赖关系，如以下示例package.json文件所示。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

在中peerDependencies，使用尖号 (^) 指定您的库aws-cdk-lib所使用的最低版本。这样可以最大限度地提高您的库与一系列 CDK 版本的兼容性。为 alpha 构造库模块指定确切版本，这些模块的 API 可能会发生变化。使用peerDependencies可以确保node_modules树中只有一个所有 CDK 库的副本。

在中devDependencies，指定测试所需的工具和库，也可以使用 ^ 表示可以接受更高版本的兼容版本。准确指定（不带 ^ 或 ~）你宣传的库aws-cdk-lib与之兼容的其他 CDK 软件包的最低版本和其他 CDK 软件包。这种做法可确保您的测试针对这些版本运行。这样，如果您无意中使用了仅在新版本中找到的功能，则您的测试可以捕捉到它。

> **Warning**
>
> peerDependencies仅由 NPM 7 及更高版本自动安装。如果您使用的是 NPM 6 或更早版本，或者使用的是 Yarn，则必须在中devDependencies包含依赖项的依赖关系。否则，它们将无法安装，并且您将收到有关未解决的对等依赖关系的警告。

安装和更新依赖关系

运行以下命令来安装项目的依赖项。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

要更新已安装的模块，可以使用前面的npm install和yarn upgrade命令。任一命令都会将软件包更新node_modules到满足中规则的最新版本package.json。但是，它们不会package.json自行更新，您可能需要这样做来设置新的最低版本。如果您托管软件包 GitHub，则可以将 De pendabot 版本更新配置为自动更新。package.json或者，请使用 npm-check-updates。

> **Important**
>
> 根据设计，当您安装或更新依赖项时，NPM 和 Yarn 会选择满足中指定要求的每个软件包的最新版本。package.json这些版本总是存在损坏的风险（无论是意外还是故意）。更新项目的依赖关系后，请进行全面测试。

AWS CDK 中的成语 JavaScript

道具

所有 C AWS onstruct Library 类都使用三个参数进行实例化：定义构造的作用域（构造树中的父级）、id 和 props（构造函数用来配置其创建的资源的键/值对）。 AWS 其他类和方法也使用 “属性包” 模式作为参数。

使用具有良好 JavaScript 自动完成功能的 IDE 或编辑器将有助于避免拼写错误的属性名称。如果一个构造需要一个`encryptionKeys`属性，而你拼写了它`encryptionkeys`，那么在实例化构造时，你还没有传递你想要的值。如果该属性是必需的，则这可能会在合成时导致错误，或者如果该属性是可选的，则会导致该属性被静默忽略。在后一种情况下，你可能会得到一个你打算覆盖的默认行为。这里要特别小心。

对 AWS 构造库类进行子类化（或重写采用类似 props 的参数的方法）时，您可能需要接受其他属性供自己使用。这些值将被父类或重写方法忽略，因为在该代码中永远不会访问它们，因此您通常可以传递收到的所有道具。

future 版本 AWS CDK 可能会巧合地添加一个新属性，其名称是你用于自己的财产。然后，将您收到的值传递到继承链上可能会导致意外行为。如果你的属性被移除或设置为，那么传递一份你收到的道具的浅层副本会更安全。`undefined`例如：

```
super(scope, name, {...props, encryptionKeys: undefined});
```

或者，请命名您的属性，使其清楚地表明它们属于您的构造。这样，它们就不太可能在 future AWS CDK 版本中与属性发生冲突。如果其中有很多，请使用一个命名恰当的对象来存放它们。

缺失值

对象（例如`props`）中缺失的值具有`undefined`中的值 JavaScript。通常的技术适用于处理这些问题。例如，访问可能未定义的值的属性的常用惯用法如下：

```
// a may be undefined, but if it is not, it may have an attribute b
// c is undefined if a is undefined, OR if a doesn't have an attribute b
let c = a && a.b;
```

但是，a如果除此之外还有其他的 “假” 值`undefined`，则最好使测试更加明确。在这里，我们将利用这样一个事实，即`null`和`undefined`等于同时测试两者：

```
let c = a == null ? a : a.b;
```

> **Tip**
>
> Node.js 14.0 及更高版本支持新的运算符，这些运算符可以简化未定义值的处理。有关更多信息，请参阅可选的链接和无效合并提案。

合成和部署

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- `cdk synth`：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- `cdk deploy`：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

使用 TypeScript 示例 JavaScript

TypeScript是我们用来开发的语言 AWS CDK，也是开发应用程序时支持的第一种语言，因此编写了许多可用的 AWS CDK 代码示例 TypeScript。对于 JavaScript 开发人员来说，这些代码示例可能是一个很好的资源；你只需要删除代码中 TypeScript特定部分即可。

TypeScript 片段通常使用较新的ECMAScript `import` 和`export`关键字从其他模块导入对象，并声明要在当前模块之外提供的对象。Node.js 刚刚开始在其最新版本中支持这些关键词。根据您正在使用（或希望支持）的 Node.js 版本，您可以重写导入和导出以使用较旧的语法。

可以将导入替换为对`require()`函数的调用。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Bucket, BucketPolicy } from 'aws-cdk-lib/aws-s3';
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { Bucket, BucketPolicy } = require('aws-cdk-lib/aws-s3');
```

可以将导出分配给`module.exports`对象。

TypeScript

```
export class Stack1 extends cdk.Stack {
  // ...
}

export class Stack2 extends cdk.Stack {
  // ...
}
```

JavaScript

```
class Stack1 extends cdk.Stack {
  // ...
}

class Stack2 extends cdk.Stack {
```

```
  // ...
}

module.exports = { Stack1, Stack2 }
```

> **Note**
>
> 使用旧式导入和导出的另一种方法是使用该esm模块。

对导入和导出进行排序后，就可以深入研究实际的代码了。您可能会遇到以下常用功能 TypeScript ：

- 类型注释
- 接口定义
- 类型转换/转换
- 访问修饰符

可以为变量、类成员、函数参数和函数返回类型提供类型注释。对于变量、参数和成员，类型是在标识符后面加上冒号和类型来指定的。函数返回值遵循函数签名，由冒号和类型组成。

要将带类型注释的代码转换为 JavaScript，请删除冒号和类型。类成员中必须有一些值 JavaScript；undefined如果类成员中只有类型注释，则将其设置为 TypeScript。

TypeScript

```
var encrypted: boolean = true;

class myStack extends cdk.Stack {
    bucket: s3.Bucket;
    // ...
}

function makeEnv(account: string, region: string) : object {
    // ...
}
```

JavaScript

```
var encrypted = true;
```

```
class myStack extends cdk.Stack {
    bucket = undefined;
    // ...
}

function makeEnv(account, region) {
    // ...
}
```

在中 TypeScript，接口用于为必需属性和可选属性的捆绑包及其类型命名。然后，您可以使用接口名称作为类型注释。 TypeScript 将确保你用作函数参数的对象具有正确类型的必需属性。

```
interface myFuncProps {
    code: lambda.Code,
    handler?: string
}
```

JavaScript 没有接口功能，因此删除类型注释后，请完全删除接口声明。

当函数或方法返回通用类型（例如`object`），但您想将该值视为更具体的子类型以访问不属于更一般类型接口的属性或方法时， TypeScript 允许您使用`as`后跟类型或接口名称来转换值。 JavaScript 不支持（或不需要）这个，所以只需删除`as`和以下标识符即可。一种不太常见的强制转换语法是在方括号中使用类型名称`<LikeThis>`；这些强制转换也必须删除。

最后， TypeScript 支持类成员的访问修饰符`public`、`protected`和`private`。中的所有班级成员 JavaScript 都是公开的。只要在任何地方看到这些修饰符即可。

知道如何识别和删除这些 TypeScript 功能对于使简短的 TypeScript片段适应大有帮助. JavaScript 但是，以这种方式转换较长的 TypeScript 示例可能不切实际，因为它们更有可能使用其他 TypeScript 功能。对于这些情况，我们建议使用 Sucras e。例如，如果代码使用未定义的变量，Sucrase 就不会抱怨。`tsc`如果它在语法上是有效的，那么除了少数例外，Sucrase 可以将其翻译为。 JavaScript这使得它对于转换可能无法单独运行的片段特别有价值。

迁移到 TypeScript

TypeScript随着项目规模越来越大、越来越复杂，许多 JavaScript 开发人员开始使用。 TypeScript 是 JavaScript（所有 JavaScript 代码均为有效代码，因此无需对 TypeScript 代码进行任何更改）的超集，而且它也是一种支持的语言。 AWS CDK 类型注释和其他 TypeScript 功能是可选的，当你发现其

中的价值时，可以将其添加到你的 AWS CDK 应用程序中。 TypeScript 还可以让您在新 JavaScript 功能最终确定之前抢先体验这些新功能，例如可选的链接和空值合并，而且无需升级 Node.js。

TypeScript的 “基于形状” 的接口，它定义了对象中的必需和可选属性（及其类型）的捆绑包，允许您在编写代码时发现常见错误，并使您的 IDE 更容易提供强大的自动完成功能和其他实时编码建议。

编程 TypeScript 确实涉及一个额外的步骤：使用编 TypeScript 译器编译应用程序`tsc`。对于典型的 AWS CDK 应用程序，编译最多需要几秒钟。

将现有 JavaScript AWS CDK 应用程序迁移到的最简单方法 TypeScript 是使用创建新 TypeScript 项目`cdk init app --language typescript`，然后将源文件（以及任何其他必需的文件，例如 AWS Lambda 函数源代码等资产）复制到新项目。将 JavaScript 文件重命名为结尾`.ts`并开始开发 TypeScript。

AWS CDK 在 Python 中使用

Python 是一种完全支持的客户端语言 AWS Cloud Development Kit (AWS CDK) ，被认为是稳定的。AWS CDK 在 Python 中使用使用熟悉的工具，包括标准 Python 实现 (CPython)、虚拟环境和 Python 包安装程序`pip`。`virtualenv`构成 AWS 构造库的模块通过 pypi .org 分发。Python 版本的 eve AWS CDK n 使用了 Python 风格的标识符（例如，`snake_case`方法名称）。

您可以使用任何编辑器或 IDE。许多 AWS CDK 开发者使用 V isual Studio Code（或其开源等效物 vsCodium），它通过官方扩展为 Python 提供了良好的支持。Python 附带的 IDLE 编辑器足以开始使用。的 Python 模块 AWS CDK 确实有类型提示，这对于支持类型验证的 linting 工具或 IDE 非常有用。

主题

- 开始使用 Python
- 创建项目
- 管理 AWS 构造库模块
- 在中管理依赖关系 Python
- AWS CDK Python 中的成语
- 合成和部署

开始使用 Python

要使用 AWS CDK，您必须拥有 AWS 账户和凭据并已安装 Node.js 和 AWS CDK Toolkit。请参阅 开始使用 AWS CDK。

Python AWS CDK 应用程序需要 Python 3.6 或更高版本。如果您尚未安装，请在 python. org 上下载适用于您的操作系统的兼容版本。如果您运行的是 Linux，则您的系统可能带有兼容版本，或者您可以使用发行版的软件包管理器（yumapt、等）进行安装。Mac 用户可能会对 Homebrew 感兴趣，这是一款适用于 macOS 的 Linux 风格的软件包管理器。

> **Note**
>
> 第三方语言弃用：语言版本仅在供应商或社区共享的 EOL（生命周期结束）之前才受支持，如有更改，恕不另行通知。

还需要使用 Python 包安装程序和虚拟环境管理器。pip virtualenvWindows 安装的兼容 Python 版本包括这些工具。在 Linux 上pip，virtualenv可以在你的软件包管理器中作为单独的软件包提供。或者，您可以使用以下命令安装它们：

```
python -m ensurepip --upgrade
python -m pip install --upgrade pip
python -m pip install --upgrade virtualenv
```

如果遇到权限错误，请运行带有--user标志的上述命令，以便将模块安装在您的用户目录中，或者使用sudo获取在系统范围内安装模块的权限。

> **Note**
>
> Linux 发行版通常使用 Python 3.x 的可执行文件名称python3，并且python参考了 Python 2.x 的安装。有些发行版有一个可以安装的可选软件包，使python命令引用 Python 3。否则，您可以通过在项目的主目录cdk.json中进行编辑来调整用于运行应用程序的命令。

> **Note**
>
> 在 Windows 上，你可能需要使用py可执行文件（适用于 Windows 的 >Python 启动器pip）来调用 Python（和）。除其他外，启动器允许您轻松指定要使用哪个已安装的 Python 版本。

如果python在命令行键入内容后会显示一条关于从 Windows 应用商店安装 Python 的消息，即使安装了 Windows 版本的 Python，也要打开 Windows 的 “管理应用程序执行别名” 设置面板，关闭 Python 的两个应用程序安装程序条目。

创建项目

您可以通过在空目录`cdk init`中调用来创建新 AWS CDK 项目。使用该`--language`选项并指定`python`：

```
mkdir my-project
cd my-project
cdk init app --language python
```

`cdk init`使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 Python 标识符的形式；例如，它不应以数字开头或包含空格。

要使用新项目，请激活其虚拟环境。这允许将项目的依赖项安装在本地项目文件夹中，而不是全局安装。

```
source .venv/bin/activate
```

Note

您可将其视为用于激活虚拟环境的 Mac/Linux 命令。Python 模板包含一个批处理文件`source.bat`，该文件允许在 Windows 上使用相同的命令。传统的 Windows `.venv\Scripts\activate.bat` 命令也能起作用。
如果您使用 CDK Toolkit v1.70.0 或更早版本初始化 AWS CDK 项目，则您的虚拟环境位于目录`.env`中，而不是。`.venv`

Important

每当你开始处理项目时，都要激活该项目的虚拟环境。否则，您将无法访问安装在那里的模块，并且您安装的模块将进入Python全局模块目录（或者会导致权限错误）。

首次激活虚拟环境后，安装应用程序的标准依赖项：

```
python -m pip install -r requirements.txt
```

管理 AWS 构造库模块

使用 Python 包安装程序安装和更新 AWS 构造库模块以供您的应用程序使用，以及您需要的其他包。pip pip还会自动安装这些模块的依赖关系。如果您的系统无法识别pip为独立命令，请以 Python 模块的pip形式调用，如下所示：

```
python -m pip PIP-COMMAND
```

大多数 AWS CDK 构造都在。aws-cdk-lib实验模块位于名为 like 的单独模块中aws-cdk.*SERVICE-NAME*.alpha。服务名称包含 a ws 前缀。如果您不确定模块的名称，请在 Py PI 上进行搜索。例如，下面的命令安装该 AWS CodeStar 库。

```
python -m pip install aws-cdk.aws-codestar-alpha
```

某些服务的构造位于多个命名空间中。例如，此外aws-cdk.aws-route53，还有另外三个 Amazon Route 53 命名空间，名为aws-route53-targetsaws-route53-patterns、和。aws-route53resolver

> **Note**
>
> Python 版本的 CDK API 参考还显示了软件包的名称。

用于将 AWS 构造库模块导入 Python 代码的名称如下所示。

```
import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lambda_
```

在应用程序中导入 AWS CDK 类和 AWS 构造库模块时，我们建议采用以下做法。遵循这些准则将有助于使您的代码与其他 AWS CDK 应用程序保持一致并更易于理解。

- 通常，从顶层导入单个类aws_cdk。

  ```
  from aws_cdk import App, Construct
  ```

- 如果您需要来自的许多类aws_cdk，则可以使用命名空间别名来cdk代替导入单个类。避免两者兼而有之。

```
import aws_cdk as cdk
```

- 通常，使用短命名空间别名导入 AWS 构造库。

```
import aws_cdk.aws_s3 as s3
```

安装模块后，更新项目的requirements.txt文件，其中列出了项目的依赖关系。最好手动执行此操作，而不是使用pip freeze。 pip freeze捕获 Python 虚拟环境中安装的所有模块的当前版本，这在捆绑项目以便在其他地方运行时非常有用。

但是，通常，您requirements.txt应该只列出顶级依赖项（您的应用程序直接依赖的模块），而不应列出这些库的依赖关系。这种策略使更新依赖关系变得更加简单。

您可以进行编辑requirements.txt以允许升级；只需将==前面的版本号替~=换为允许升级到更高的兼容版本，或者完全删除版本要求以指定模块的最新可用版本。

requirements.txt经过适当编辑以允许升级，可以随时发出以下命令来升级项目中已安装的模块：

```
pip install --upgrade -r requirements.txt
```

在中管理依赖关系 Python

在 Python 中，你可以通过requirements.txt为应用程序或setup.py构造库输入依赖关系来指定依赖关系。然后使用 PIP 工具管理依赖关系。PIP 可通过以下方式之一调用：

```
pip command options
python -m pip command options
```

该python -m pip调用适用于大多数系统；pip要求 PIP 的可执行文件位于系统路径上。如果pip不起作用，请尝试将其替换为python -m pip。

该cdk init --language python命令为您的新项目创建虚拟环境。这使每个项目都有自己的依赖关系版本，还有一个基本requirements.txt文件。source .venv/bin/activate每次开始使用项目时，都必须通过运行来激活此虚拟环境。

CDK 应用程序

下面是一个 requirements.txt 示例文件。由于 PIP 没有依赖锁定功能，因此我们建议您使用 == 运算符为所有依赖项指定确切的版本，如下所示。

```
aws-cdk-lib==2.14.0
aws-cdk.aws-appsync-alpha==2.10.0a0
```

使用安装模块pip install不会自动将其添加到requirements.txt。你必须自己动手。如果要升级到依赖关系的更高版本，请在中编辑其版本号requirements.txt。

要在创建或编辑项目后安装或更新项目的依赖项requirements.txt，请运行以下命令：

```
python -m pip install -r requirements.txt
```

> **Tip**
>
> 该pip freeze命令以可写入文本文件的格式输出所有已安装依赖项的版本。这可以用作需求文件pip install -r。此文件便于将所有依赖项（包括传递依赖项）固定到您测试过的确切版本。为避免以后升级软件包时出现问题，请为此使用单独的文件，例如freeze.txt（不是requirements.txt）。然后，在升级项目的依赖关系时重新生成它。

第三方构造库

在库中，依赖关系是在中指定的setup.py，因此当应用程序使用软件包时，会自动下载传递依赖关系。否则，每个想要使用你的软件包的应用程序都需要将你的依赖项复制到它们的requirements.txt。此setup.py处显示了一个示例。

```
from setuptools import setup

setup(
  name='my-package',
  version='0.0.1',
  install_requires=[
    'aws-cdk-lib==2.14.0',
  ],
  ...
)
```

要使用软件包进行开发，请创建或激活虚拟环境，然后运行以下命令。

```
python -m pip install -e .
```

尽管 PIP 会自动安装传递依赖项，但任何一个软件包只能安装一个副本。选择依赖关系树中指定的最高版本；应用程序总是决定安装哪个版本的软件包。

AWS CDK Python 中的成语

语言冲突

在 Python 中，`lambda`是一个语言关键字，因此您不能将其用作 AWS Lambda 构造库模块或 Lambda 函数的名称。对于此类冲突，Python 惯例是在变量名中使用尾随下划线`lambda_`，如所示。

按照惯例， AWS CDK 构造的第二个参数是命名`id`的。在编写自己的堆栈和构造时，调用参数 `id` “阴影” Python 内置函数`id()`，该函数返回对象的唯一标识符。这个函数并不经常使用，但是如果你在构造中碰巧需要它，例如，可以重命名参数`construct_id`。

参数和属性

所有 C AWS onstruct Library 类都使用三个参数进行实例化：定义构造的作用域（构造树中的父级）、id 和 props（构造函数用来配置其创建的资源的键/值对）。其他类和方法也使用 “属性包” 模式作为参数。

s@@ cop e 和 id 应始终作为位置参数而不是关键字参数传递，因为如果构造接受名为 s co pe 或 id 的属性，它们的名称就会改变。

在 Python 中，道具以关键字参数的形式表示。如果参数包含嵌套的数据结构，则这些数据结构使用在实例化时采用自己的关键字参数的类来表示。同样的模式也适用于其他采用结构化参数的方法调用。

例如，在 Amazon S3 存储桶的`add_lifecycle_rule`方法中，该`transitions`属性是一个`Transition`实例列表。

```
bucket.add_lifecycle_rule(
  transitions=[
    Transition(
      storage_class=StorageClass.GLACIER,
      transition_after=Duration.days(10)
    )
```

```
  ]
)
```

扩展类或重写方法时，您可能希望出于自己的目的接受父类无法理解的其他参数。在这种情况下，你应该接受你不在乎使用**kwargs成语的论点，并使用仅限关键字的参数来接受你感兴趣的参数。在调用父级的构造函数或被重写的方法时，只传递它期望的参数（通常只**kwargs是）。传递父类或方法不期望的参数会导致错误。

```
class MyConstruct(Construct):
    def __init__(self, id, *, MyProperty=42, **kwargs):
        super().__init__(self, id, **kwargs)
        # ...
```

future 版本 AWS CDK 可能会巧合地添加一个新属性，其名称是你用于自己的财产。这不会给你的构造或方法的用户造成任何技术问题（由于你的属性不是 “向上链” 传递的，所以父类或被重写的方法只会使用默认值），但可能会引起混淆。你可以通过命名你的属性来避免这个潜在的问题，让它们明确属于你的构造。如果有许多新属性，请将它们捆绑到一个适当命名的类中，然后将其作为单个关键字参数传递。

缺失值

AWS CDK 用于None表示缺失值或未定义值。使用时**kwargs，如果未提供属性，则使用字典的get()方法提供默认值。避免使用kwargs[...]，因为这会增加KeyError缺失值。

```
encrypted = kwargs.get("encrypted")         # None if no property "encrypted" exists
encrypted = kwargs.get("encrypted", False)  # specify default of False if property is
 missing
```

某些 AWS CDK 方法（例如tryGetContext()获取运行时上下文值）可能会返回None，您需要明确检查这些方法。

使用接口

Python 不像其他一些语言那样具有接口功能，尽管它确实有类似的抽象基类。（如果你不熟悉界面，维基百科有一个很好的介绍。）TypeScript，实现时所 AWS CDK 用的语言确实提供了接口，而构造和其他 AWS CDK 对象通常需要一个附属于特定接口的对象，而不是从特定类继承的对象。因此，作为 JSII 层的一部分， AWS CDK 提供了自己的接口功能。

要表示一个类实现了特定的接口，你可以使用@jsii.implements装饰器：

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
    def visit(self, node: IConstruct) -> None:
        print("Visited", node.node.path)
```

类型陷阱

Python 使用动态类型，其中所有变量都可以指任何类型的值。参数和返回值可以用类型注释，但这些是“提示”，不是强制性的。这意味着在 Python 中，很容易将错误类型的值传递给 AWS CDK 构造。当JSII层（在Python和 TypeScript 核心之间进行转换）无法处理意外类型时，您可能会遇到运行时错误，而不是像静态类型语言 AWS CDK那样在构建过程中出现类型错误。

根据我们的经验，Python 程序员犯的类型错误往往属于这些类别。

- 在构造需要容器（Python 列表或字典）的地方传递单个值，反之亦然。
- 将与第 1 层 (`CfnXxxxxx`) 构造关联的类型的值传递给 L2 或 L3 构造，反之亦然。

AWS CDK Python 模块确实包含类型注释，因此您可以使用支持它们的工具来帮助处理类型。如果您使用的不是支持这些功能的 IDE（例如）PyCharm，则可能需要调用MyPy类型验证器作为构建过程中的一个步骤。还有一些运行时类型检查器可以改进与类型相关的错误的错误消息。

合成和部署

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- `cdk synth`：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- `cdk deploy`：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

AWS CDK 在 Java 中使用

Java 是完全支持的客户端语言 AWS CDK ，被认为是稳定的。你可以使用熟悉的工具用 Java 开发 AWS CDK 应用程序，包括 JDK（甲骨文的，或者像 Amazon Corretto 这样的 OpenJDK 发行版）和 Apache Maven。

AWS CDK 支持 Java 8 及更高版本。但是，我们建议您尽量使用最新版本，因为该语言的更高版本包含了特别便于开发 AWS CDK 应用程序的改进。例如，Java 9 引入了该`Map.of()`方法（一种声明哈希映射的便捷方法，该哈希映射将在中 TypeScript写成对象文字）。Java 10 使用`var`关键字引入了局部类型推断。

> **Note**
>
> 本开发人员指南中的大多数代码示例都适用于 Java 8。有几个示例使用`Map.of()`；这些示例包括注释，指出它们需要 Java 9。

您可以使用任何文本编辑器或可以读取 Maven 项目的 Java IDE 来处理您的 AWS CDK 应用程序。我们在本指南中提供了 Eclipse 提示，但是 IntelliJ IDEA 和其他 IDE 可以导入 Maven 项目，并且可以用来在 Java 中开发应用程序。 NetBeans AWS CDK

可以用 Jvm 托管的语言（例如 Kotlin、Groovy、Clojure 或 Scala）编写 AWS CDK 应用程序，但这种体验可能不是特别习惯，我们无法为这些语言提供任何支持。

主题

开始使用 Java

要使用 AWS CDK，您必须拥有 AWS 账户和凭证，并已安装 Node.js 和 AWS CDK Toolkit。请参阅开始使用 AWS CDK。

Java AWS CDK 应用程序需要 Java 8 (v1.8) 或更高版本。我们推荐 Amazon Cor retto，但你可以使用任何 OpenJDK 发行版或 Oracle 的 JDK。你还需要使用 Apache Maven 3.5 或更高版本。你也可以使用诸如 Gradle 之类的工具，但是 AWS CDK Toolkit 生成的应用程序框架是 Maven 项目。

> **Note**
>
> 第三方语言弃用：语言版本仅在供应商或社区共享的 EOL（生命周期结束）之前才受支持，如有更改，恕不另行通知。

创建项目

您可以通过在空目录`cdk init`中调用来创建新 AWS CDK 项目。使用该`--language`选项并指定`java`：

```
mkdir my-project
cd my-project
cdk init app --language java
```

`cdk init`使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 Java 标识符的形式；例如，它不应以数字开头或包含空格。

生成的项目包括对 `software.amazon.awscdk` Maven 包的引用。它及其依赖项由 Maven 自动安装。

如果您使用的是 IDE，则现在可以打开或导入项目。例如，在 Eclipse 中，选择 “文件” > “导入” > “Maven” > “现有的 Maven 项目”。确保将项目设置设置为使用 Java 8 (1.8)。

管理 AWS 构造库模块

使用 Maven 安装组`software.amazon.awscdk`中的 AWS 构造库软件包。大多数构造都在构件中，默认情况下`aws-cdk-lib`，该构件会添加到新的 Java 项目中。仍在开发更高级别 CDK 支持的服务的模块位于单独的 “实验性” 包中，并以其服务名称的简短版本（无或 AWS Amazon 前缀）命名。搜索 Maven 中央存储库以查找所有库 AWS CDK 和 AWS 构造模块库的名称。

> **Note**
>
> Java 版本的 CDK API 参考还显示了软件包名称。

某些服务的 AWS 构造库支持位于多个命名空间中。例如，Amazon Route 53 的功能分为`software.amazon.awscdk.route53`、`route53-patterns`、`route53resolver`和`route53-targets`。

主 AWS CDK 包以 Java 代码导入为`software.amazon.awscdk`。 AWS 构造库中各种服务的模块位于其下，其命名`software.amazon.awscdk.services`与它们的 Maven 包名称类似。例如，Amazon S3 模块的命名空间是`software.amazon.awscdk.services.s3`。

我们建议您为每个 Java 源文件中使用的每个 C AWS onstruct Library 类编写单独的 Java `import` 语句，并避免导入通配符。您始终可以在不使用`import`语句的情况下使用类型的完全限定名称（包括其命名空间）。

如果您的应用程序依赖于实验包，请编辑您的项目`pom.xml`并在`<dependencies>`容器中添加新`<dependency>`元素。例如，以下`<dependency>`元素指定了 CodeStar 实验构造库模块：

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>codestar-alpha</artifactId>
    <version>2.0.0-alpha.10</version>
</dependency>
```

> **Tip**
>
> 如果你使用 Java IDE，它可能具有管理 Maven 依赖关系的功能。但是，除非你完全确定 IDE 的功能与你手动操作的功能相匹配，否则我们建议你pom.xml直接进行编辑。

在中管理依赖关系 Java

在 Java 中，依赖关系是在 Maven 中指定pom.xml并使用 Maven 安装的。<dependencies>容器包含每个包的<dependency>元素。以下是典型的 pom.xml CDK Java 应用程序的部分。

```
<dependencies>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>aws-cdk-lib</artifactId>
        <version>2.14.0</version>
    </dependency>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>appsync-alpha</artifactId>
        <version>2.10.0-alpha.0</version>
    </dependency>
</dependencies>
```

> **Tip**
>
> 许多 Java IDE 都集成了 Maven 支持和可视化pom.xml编辑器，您可能会发现它们便于管理依赖关系。

Maven 不支持依赖项锁定。尽管可以在中指定版本范围pom.xml，但我们建议您始终使用精确的版本，以保持构建的可重复性。

Maven 会自动安装传递依赖项，但每个软件包只能安装一个副本。选择在 POM 树中指定的最高版本；应用程序总是决定安装哪个版本的软件包。

每当你构建 (mvn compile) 或打包 (mvn package) 项目时，Maven 都会自动安装或更新你的依赖项。每次运行 CDK Toolkit 时，它都会自动执行此操作，因此通常无需手动调用 Maven。

AWS CDK Java 中的成语

道具

所有 C AWS onstruct Library 类都使用三个参数进行实例化：定义构造的作用域（构造树中的父级）、id 和 props（构造函数用来配置其创建的资源的键/值对）。其他类和方法也使用 “属性包” 模式作为参数。

在 Java 中，道具是使用生成器模式表示的。每种构造类型都有相应的道具类型；例如，`Bucket`构造（代表 Amazon S3 存储桶）将的实例作为其道具。`BucketProps`

该`BucketProps`类（就像每个 Constru AWS ct Library 道具类一样）有一个名`Builder`为的内部类。该`BucketProps.Builder`类型提供了设置`BucketProps`实例各种属性的方法。每个方法都返回`Builder`实例，因此可以链接方法调用以设置多个属性。在链的尽头，你调`build()`用实际生成`BucketProps`对象。

```
Bucket bucket = new Bucket(this, "MyBucket", new BucketProps.Builder()
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build());
```

构造和其他以类似 props 的对象作为最终参数的类提供了捷径。该类有自己`Builder`的，可以一步实例化它和它的 props 对象。这样，你就不必显式实例化（例如）`BucketProps`和 aBucket，也不需要导入 props 类型。

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build();
```

从现有构造派生自己的构造时，可能需要接受其他属性。我们建议您遵循这些构建器模式。但是，这并不像子类化构造类那么简单。你必须自己提供这两个新`Builder`职业的活动部分。你可能更愿意让你的构造接受一个或多个额外的参数。如果参数是可选的，则应提供其他构造函数。

通用结构

在某些 API 中， AWS CDK 使用 JavaScript 数组或非类型化对象作为方法的输入。（例如，AWS CodeBuild参见`BuildSpec.fromObject()`的方法。）在 Java 中，这些对象表示为`java.util.Map<String, Object>`。如果值都是字符串，则可以使用`Map<String, String>`。

Java 不像其他一些语言那样提供为此类容器编写文字的方法。在 Java 9 及更高版本中 java.util.Map.of()，您可以使用其中一个调用方便地定义多达十个条目的映射。

```
java.util.Map.of(
    "base-directory", "dist",
    "files", "LambdaStack.template.json"
 )
```

要创建包含十个以上条目的地图，请使用java.util.Map.ofEntries()。

如果您使用的是 Java 8，则可以提供与这些方法类似的自己的方法。

JavaScript 数组在 Java List<String> 中以List<Object>或的形式表示。该方法便java.util.Arrays.asList于定义短 List s。

```
List<String> cmds = Arrays.asList("cd lambda", "npm install", "npm install typescript")
```

缺失值

在 Java 中，诸如道具之类的 AWS CDK 对象中的缺失值用表示。null在对任何可能的值进行任何操作之前，必须对其进行显式测试，以确保它包含一个值。nullJava 不像其他一些语言那样具有 “语法糖” 来帮助处理空值。你可能会发现 Apac ObjectUtil h defaultIfNulle 在某些情况下firstNonNull很有用。或者，也可以编写自己的静态辅助方法，以便更轻松地处理潜在的空值并提高代码的可读性。

构建、合成和部署

在运行您的应用程序之前 AWS CDK 会自动对其进行编译。但是，手动构建应用程序以检查错误和运行测试可能会很有用。您可以在 IDE 中执行此操作（例如，在 Eclipse 中按 Control-B），也可以在项目的根目录中mvn compile在命令提示符下发出命令来执行此操作。

在命令提示符mvn test下运行您编写的所有测试。

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- cdk synth：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- cdk deploy：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

AWS CDK 在 C# 中使用

.NET 是完全支持的客户端语言 AWS CDK ，被认为是稳定的。C# 是主要的.NET 语言，我们为其提供示例和支持。您可以选择使用其他.NET 语言（例如 Visual Basic 或 F#）编写 AWS CDK 应用程序，但 AWS 对在 CDK 中使用这些语言的支持有限。

您可以使用熟悉的工具（包括 Visual Studio、Visual Studio 代码、`dotnet`命令和 NuGet 包管理器）在 C# 中开发 AWS CDK 应用程序。构成 AWS 构造库的模块通过 nuget .org 分发。

我们建议在 Windows 上使用 Visual Studio 2019（任何版本）使用 C# 开发 AWS CDK 应用程序。

主题

开始使用 C#

要使用 AWS CDK，您必须拥有 AWS 账户和凭证，并已安装 Node.js 和 AWS CDK Toolkit。请参阅开始使用 AWS CDK。

C# AWS CDK 应用程序需要.NET Core v3.1 或更高版本，可在此处获得。

.NET 工具链包括`dotnet`一个用于构建和运行.NET 应用程序以及管理软件包的命令行工具。 NuGet 即使您主要在 Visual Studio 中工作，此命令也可用于批处理操作和安装 AWS 构造库包。

创建项目

您可以通过在空目录`cdk init`中调用来创建新 AWS CDK 项目。使用该`--language`选项并指定`csharp`：

```
mkdir my-project
cd my-project
cdk init app --language csharp
```

`cdk init`使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 C# 标识符的形式；例如，它不应以数字开头或包含空格。

生成的项目包括对`Amazon.CDK.Lib` NuGet 包的引用。它及其依赖项由自动安装 NuGet。

管理 AWS 构造库模块

.NET 生态系统使用 NuGet 软件包管理器。包含核心类和所有稳定的服务构造的 CDK 主包是。`Amazon.CDK.Lib`正在积极开发新功能的实验模块命名为`Amazon.CDK.AWS.`*`SERVICE-NAME`*`.Alpha`，其中服务名称是一个不带 AWS 或 Amazon 前缀的短名称。例如， AWS IoT 模块的 NuGet 软件包名称是`Amazon.CDK.AWS.IoT.Alpha`。如果您找不到想要的软件包，请搜索 Nuget .org。

> **Note**
>
> CDK API 参考的.NET 版本还显示了软件包名称。

某些服务的 AWS 构造库支持包含在多个模块中。例如， AWS IoT 有第二个名为的模块`Amazon.CDK.AWS.IoT.Actions.Alpha`。

在大多数 AWS CDK 应用程序中都需要 AWS CDK的主模块，在 C# 代码中导入为`Amazon.CDK`。AWS 构造库中各种服务的模块位于其下`Amazon.CDK.AWS`。例如，Amazon S3 模块的命名空间是`Amazon.CDK.AWS.S3`。

我们建议为 CDK 核心结构以及您在每个 C# 源文件中使用的每项 AWS 服务编写 C# `using` 指令。您可能会发现使用命名空间或类型的别名来帮助解决名称冲突很方便。您可以随时使用类型的完全限定名称（包括其命名空间），而不必使用`using`语句。

在中管理依赖关系 C#

在 C# AWS CDK 应用程序中，您可以使用管理依赖关系 NuGet。 NuGet 有四个标准的、基本上等同的接口。使用适合您的需求和工作风格的那个。您还可以使用兼容的工具，例如 Paket，MyGet甚至可以直接编辑`.csproj`文件。

NuGet 不允许您为依赖项指定版本范围。每个依赖项都固定到一个特定的版本。

更新依赖关系后，Visual Studio 将在下次生成时使用 NuGet 检索每个包的指定版本。如果您没有使用 Visual Studio，请使用dotnet restore命令更新您的依赖关系。

直接编辑项目文件

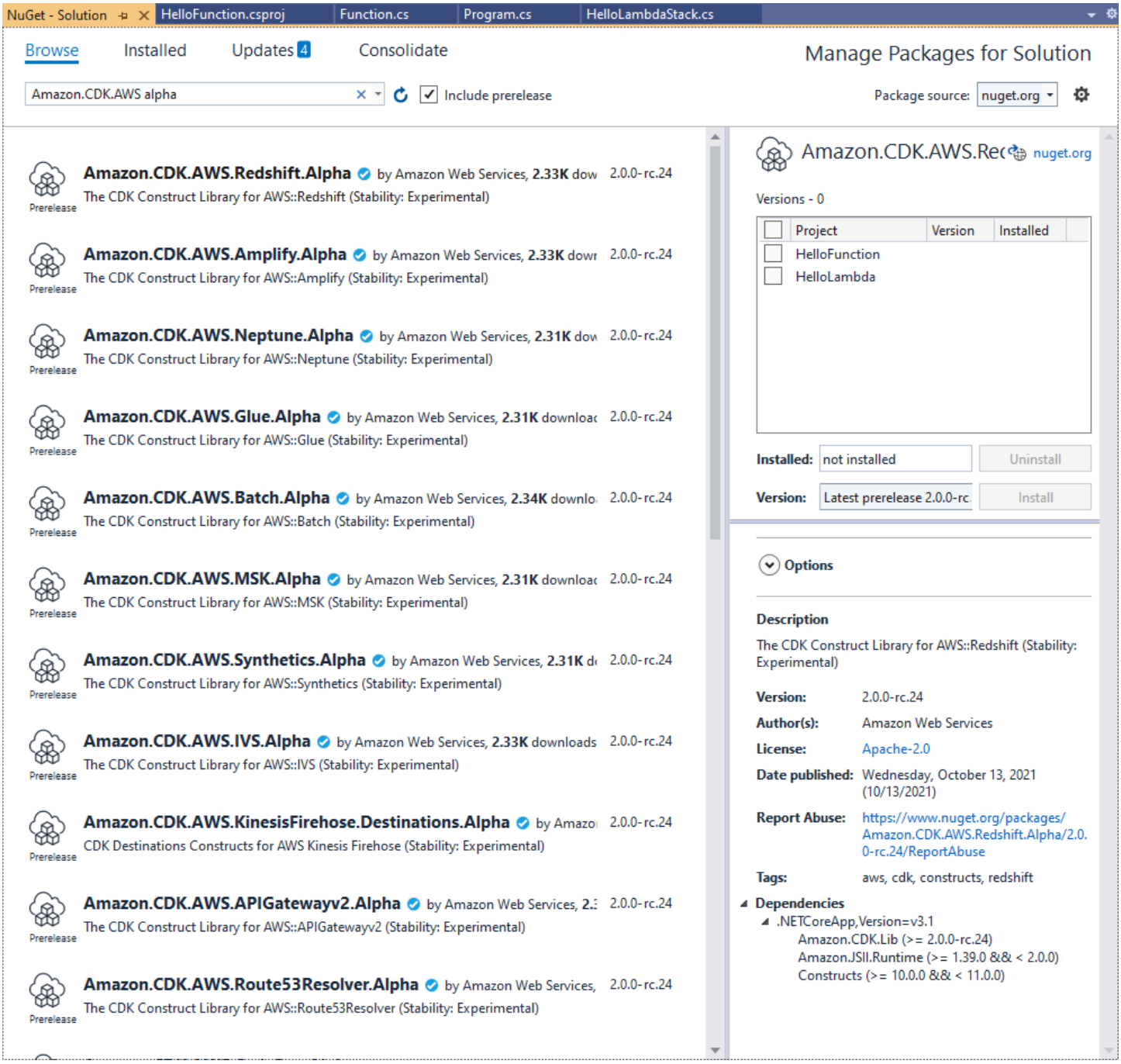
您的项目`.csproj`文件包含一个`<ItemGroup>`容器，该容器将您的依赖项列为`<PackageReference`元素。

```
<ItemGroup>
    <PackageReference Include="Amazon.CDK.Lib" Version="2.14.0" />
    <PackageReference Include="Constructs" Version="%constructs-version%" />
</ItemGroup>
```

Visual St NuGet udio

Visual Studio 的 NuGet 工具可通过 “工具” > “NuGet 包管理器” > “管理解决方案 NuGet 包” 进行访问。使用 “浏览” 选项卡查找要安装的 AWS 构造库软件包。您可以选择所需的版本，包括模块的预发行版本，然后将其添加到任何打开的项目中。

> **Note**
>
> 所有被视为 “实验性” 的 C AWS onstruct Library 模块（参见the section called “版本控制”）都被标记为预发行版， NuGet 并带有`alpha`名称后缀。



查看 “更新” 页面，安装软件包的新版本。

控制 NuGet 台

NuGet 控制台是一个 PowerShell基于 Visual Studio 项目的界面 NuGet ，可以在 Visual Studio 项目的上下文中运行。你可以在 Visual Studio 中通过选择 “工具” > “NuGet 软件包管理器” > “Packag e Man ager 控制台” 将其打开。有关使用此工具的更多信息，请参见在 Visual Studio 中使用 Package Manager 控制台安装和管理软件包。

该dotnet命令

该dotnet命令是处理 Visual Studio C# 项目的主要命令行工具。你可以从任何 Windows 命令提示符调用它。在其众多功能中，dotnet可以向 Visual Studio 项目添加 NuGet依赖关系。

假设你与 Visual Studio 项目 (.csproj) 文件位于同一个目录中，请发出如下命令来安装软件包。由于创建项目时会包含主 CDK 库，因此您只需要明确安装实验模块即可。实验模块要求您指定明确的版本号。

```
dotnet add package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

您可以从另一个目录发出该命令。为此，请在add关键字后面加上项目文件的路径或包含该文件的目录的路径。以下示例假设您位于 AWS CDK 项目的主目录中。

```
dotnet add src/PROJECT-DIR package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

要安装软件包的特定版本，请包括-v标志和所需的版本。

要更新软件包，请发出与安装软件包相同的dotnet add命令。同样，对于实验模块，您必须指定明确的版本号。

有关使用dotnet命令管理软件包的更多信息，请参阅使用 dotnet CLI 安装和管理软件包。

该nuget命令

nuget命令行工具可以安装和更新 NuGet 软件包。但是，它要求您的 Visual Studio 项目的设置方式与cdk init设置项目的方式不同。（技术细节nuget：处理Packages.config项目，同时cdk init创建新风格的PackageReference项目。）

我们不建议在由创建的 AWS CDK 项目中使用该nuget工具cdk init。如果您正在使用其他类型的项目，并且想要使用nuget，请参阅 NuGet CLI 参考。

AWS CDK C# 中的成语

道具

所有 C AWS onstruct Library 类都使用三个参数进行实例化：定义构造的作用域（构造树中的父级）、id 和 props（构造函数用来配置其创建的资源的键/值对）。其他类和方法也使用 “属性包” 模式作为参数。

在 C# 中，道具是使用 props 类型表示的。按照惯用的 C# 方式，我们可以使用对象初始化器来设置各种属性。这里我们使用`Bucket`构造创建一个 Amazon S3 存储桶；其对应的道具类型是`BucketProps`。

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
    Versioned = true
});
```

> **Tip**
>
> 将软件包`Amazon.JSII.Analyzers`添加到您的项目中，以便在 Visual Studio 中检查道具定义中的必填值。

在扩展类或重写方法时，您可能希望出于自己的目的接受父类无法理解的其他道具。为此，请对相应的 props 类型进行子类化并添加新属性。

```
// extend BucketProps for use with MimeBucket
class MimeBucketProps : BucketProps {
    public string MimeType { get; set; }
}

// hypothetical bucket that enforces MIME type of objects inside it
class MimeBucket : Bucket {
     public MimeBucket( readonly Construct scope, readonly string id, readonly
 MimeBucketProps props=null) : base(scope, id, props) {
         // ...
     }
}

// instantiate our MimeBucket class
var bucket = new MimeBucket(this, "MyBucket", new MimeBucketProps {
    Versioned = true,
    MimeType = "image/jpeg"
});
```

在调用父类的初始化器或重写的方法时，通常可以传递收到的 props。新类型与其父类型兼容，您添加的额外道具将被忽略。

future 版本 AWS CDK 可能会巧合地添加一个新属性，其名称是你用于自己的财产。这不会在使用你的构造或方法时造成任何技术问题（因为你的属性不是 “向上链” 传递的，所以父类或重写的方法只会

使用默认值），但它可能会给你的构造的用户造成混乱。你可以通过命名你的属性来避免这个潜在的问题，让它们明确属于你的构造。如果有许多新属性，请将它们捆绑到一个适当命名的类中，然后将它们作为单个属性传递。

通用结构

在某些 API 中， AWS CDK 使用 JavaScript 数组或非类型化对象作为方法的输入。（例如， AWS CodeBuild参见`BuildSpec.fromObject()`的方法。）在 C# 中，这些对象表示为`System.Collections.Generic.Dictionary<String, Object>`。如果值都是字符串，则可以使用`Dictionary<String, String>`。 JavaScript 在 C# 中，`string[]`数组表示为`object[]`或数组类型。

> **Tip**
>
> 您可以定义简短的别名，以便更轻松地使用这些特定的字典类型。
>
> ```
> using StringDict = System.Collections.Generic.Dictionary<string, string>;
> using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
> ```

缺失值

在 C# 中，诸如道具之类的 AWS CDK 对象中的缺失值用表示。`null`空条件成员访问运算符`?.`和空合并运算符便`??`于处理这些值。

```
// mimeType is null if props is null or if props.MimeType is null
string mimeType = props?.MimeType;

// mimeType defaults to text/plain. either props or props.MimeType can be null
string MimeType = props?.MimeType ?? "text/plain";
```

构建、合成和部署

在运行您的应用程序之前 AWS CDK 会自动对其进行编译。但是，手动构建应用程序以检查错误和运行测试可能会很有用。您可以通过在 Visual Studio 中按 F6 或`dotnet build src`从命令行发出命令来执行此操作，其中`src`是项目目录中包含 Visual Studio 解决方案 (`.sln`) 文件的目录。

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- `cdk synth`：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- `cdk deploy`：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

AWS CDK 在 Go 中使用

Go 是一种完全支持的客户端语言 AWS Cloud Development Kit (AWS CDK) ，被认为是稳定的。AWS CDK 在 Go 中使用熟悉的工具进行操作。事件的 Go 版本使用 G AWS CDK o 风格的标识符。

与 CDK 支持的其他语言不同，Go 不是传统的面向对象编程语言。Go 使用其他语言经常利用继承的组合。我们已经尝试尽可能使用惯用的 Go 方法，但是 CDK 在某些地方可能会有所不同。

本主题提供 AWS CDK 在 Go 中使用 Go 的指导。有关一个简单的 Go 项目的演练，请参阅公告博客文章。 AWS CDK

主题

- 在中管理依赖关系 Go
- AWS CDK 围棋中的成语
- 构建、合成和部署

开始使用 Go

要使用 AWS CDK，您必须拥有 AWS 账户和凭据并已安装 Node.js 和 AWS CDK Toolkit。请参阅 开始使用 AWS CDK。

的 Go 绑定 AWS CDK 使用标准的 Go 工具链，v1.18 或更高版本。您可以使用自己选择的编辑器。

> **Note**
>
> 第三方语言弃用：语言版本仅在供应商或社区共享的 EOL（生命周期结束）之前才受支持，如有更改，恕不另行通知。

创建项目

您可以通过在空目录`cdk init`中调用来创建新 AWS CDK 项目。使用该`--language`选项并指定`go`：

```
mkdir my-project
cd my-project
cdk init app --language go
```

`cdk init`使用项目文件夹的名称来命名项目的各种元素，包括类、子文件夹和文件。文件夹名称中的连字符将转换为下划线。但是，除此之外，名称应遵循 Go 标识符的形式；例如，它不应以数字开头或包含空格。

生成的项目包括对中核心 AWS CDK Go 模块的引用`go.mod`。`github.com/aws/aws-cdk-go/awscdk/v2`安装`go get`此模块和其他必需模块时出现问题。

管理 AWS 构造库模块

在大多数 AWS CDK 文档和示例中，“模块” 一词通常用于指 AWS 构造库模块，每个 AWS 服务一个或多个模块，这与该术语的惯用 Go 用法不同。CDK 构造库在一个 Go 模块中提供，其中包含单独的构造库模块，这些模块支持该模块中作为 Go 包提供的各种 AWS 服务。

某些服务的 AWS 构造库支持位于多个构造库模块（Go 包）中。例如，除了名为awsroute53patterns、和的主awsroute53包之外，Amazon Route 53 还有三个构造库模块awsroute53targets。awsroute53resolver

AWS CDK的核心包是你在大多数 AWS CDK 应用程序中都需要的，在 Go 代码中导入为github.com/aws/aws-cdk-go/awscdk/v2。Construct Library AWS 中各种服务的软件包位于github.com/aws/aws-cdk-go/awscdk/v2。例如，Amazon S3 模块的命名空间是github.com/aws/aws-cdk-go/awscdk/v2/awss3。

```
import (
        "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
        // ...
)
```

为要在应用程序中使用的服务导入构造库模块（Go 包）后，您可以使用例如访问该模块中的构造。awss3.Bucket

在中管理依赖关系 Go

在 Go 中，依赖关系版本是在中定义的go.mod。默认值类似go.mod于此处显示的值。

```
module my-package

go 1.16

require (
  github.com/aws/aws-cdk-go/awscdk/v2 v2.16.0
  github.com/aws/constructs-go/constructs/v10 v10.0.5
  github.com/aws/jsii-runtime-go v1.29.0
)
```

Package 名称（模块，用 Go 的话来说）由 URL 指定，并附加了所需的版本号。Go 的模块系统不支持版本范围。

发出go get命令以安装所有必需的模块并进行更新go.mod。要查看依赖项的可用更新列表，请发出go list -m -u all。

AWS CDK 围棋中的成语

字段和方法名称

字段和方法名称在 CDK 的起源语言中 TypeScript使用驼色大小写 (`likeThis`)。在 Go 中，它们遵循 Go 惯例，Pascal 大小写的 () 也是如此。`LikeThis`

清理

在您的`main`方法中，使用`defer jsii.Close()`来确保 CDK 应用程序自行清理。

缺失值和指针转换

在 Go 中，诸如属性包之类的 AWS CDK 对象中的缺失值用`nil`表示。Go 没有可为空的类型；唯一可以包含的类型`nil`是指针。因此，为了允许值是可选的，所有 CDK 属性、参数和返回值都是指针，即使对于原始类型也是如此。这既适用于必填值，也适用于可选值，因此，如果必填值后来变为可选值，则无需对类型进行重大更改。

传递文字值或表达式时，请使用以下辅助函数来创建指向这些值的指针。

- `jsii.String`
- `jsii.Number`
- `jsii.Bool`
- `jsii.Time`

为了保持一致性，我们建议您在定义自己的构造时类似地使用指针，尽管将构造`id`作为字符串而不是字符串指针接收似乎更方便。

在处理可选 AWS CDK 值（包括原始值和复杂类型）时，在对指针执行任何操作`nil`之前，应显式测试指针以确保它们不是。Go 没有像其他一些语言那样的 “语法糖” 来帮助处理空值或缺失值。但是，属性包和类似结构中的必填值可以保证存在（否则构造失败），因此无需对这些值进行检查`nil`。

构造和道具

代表一个或多个 AWS 资源及其关联属性的构造在 Go 中表示为接口。例如，`awss3.Bucket`是一个接口。每个构造都有一个工厂函数`awss3.NewBucket`，例如，返回一个实现相应接口的结构。

所有工厂函数都采用三个参数：`scope`在其中定义构造的（构造树中的父项）、一个`idprops`、以及构造用来配置其创建的资源的键/值对的捆绑包。其他地方也使用了 “属性包” 模式 AWS CDK。

在 Go 中，道具由每个构造的特定结构类型表示。例如，awss3.Bucket采用类型awss3.BucketProps为 props 的参数。使用结构体字面量来编写 props 参数。

```
var bucket = awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
    Versioned: jsii.Bool(true),
})
```

通用结构

在某些地方， AWS CDK 使用 JavaScript 数组或非类型化对象作为方法的输入。（例如， AWS CodeBuild参见BuildSpec.fromObject()的方法。）在 Go 中，这些对象分别表示为切片和空接口。

CDK 提供了可变参数辅助函数，例如jsii.Strings用于构建包含原始类型的切片。

```
jsii.Strings("One", "Two", "Three")
```

开发自定义结构

在 Go 中，编写新构造通常比扩展现有构造更简单。首先，定义一个新的结构类型，如果需要类似扩展的语义，则匿名嵌入一个或多个现有类型。为您要添加的任何新功能以及保存所需数据所需的字段编写方法。如果你的构造需要一个 props 接口，可以定义一个 props 接口。最后，编写一个工厂函数NewMyConstruct()来返回构造的实例。

如果你只是在现有构造上更改一些默认值，或者在实例化时添加一个简单的行为，那么你不需要所有这些管道。相反，写一个工厂函数来调用你正在 “扩展” 的构造的工厂函数。例如，在其他 CDK 语言中，您可以创建一个TypedBucket构造来强制执行 Amazon S3 存储桶中对象的类型，方法是覆盖该s3.Bucket类型，然后在新类型的初始化器中添加仅允许向存储桶添加指定文件扩展名的存储桶策略。在 Go 中，更容易简单地编写一个NewTypedBucket返回您已向其中添加了相应存储桶策略的s3.Bucket（使用实例化s3.NewBucket）。不需要新的构造类型，因为标准存储桶构造中已经提供了该功能；新的 “构造” 只是提供了一种更简单的配置方式。

构建、合成和部署

在运行您的应用程序之前 AWS CDK 会自动对其进行编译。但是，手动构建应用程序以检查错误和运行测试可能会很有用。你可以通过在项目的根目录go build下在命令提示符下发出命令来做到这一点。

在命令提示符go test下运行您编写的所有测试。

可以使用以下命令合成 AWS CDK 应用程序中定义的堆栈并单独或一起部署。通常，当你发布它们时，你应该在项目的主目录中。

- `cdk synth`：从应用程序中的一个或多个堆栈中 AWS CDK 合成一个 AWS CloudFormation 模板。
- `cdk deploy`：将您的 AWS CDK 应用程序中的一个或多个堆栈定义的资源部署到。 AWS

您可以在单个命令中指定要合成或部署的多个堆栈的名称。如果您的应用程序只定义了一个堆栈，则无需指定该堆栈。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

您也可以使用通配符 *（任意数量的字符）和?（任何单个字符），用于按模式识别堆栈。使用通配符时，请用引号将模式括起来。否则，在将文件传递到 T AWS CDK oolkit 之前，shell 可能会尝试将其扩展为当前目录中的文件名。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> 在部署堆栈之前，您无需显式合成堆栈；请`cdk deploy`执行此步骤以确保部署最新的代码。

有关该`cdk`命令的完整文档，请参阅the section called “AWS CDK 工具包”。

开发 AWS CDK 应用程序

开发 AWS Cloud Development Kit (AWS CDK) 应用程序。

主题

自定义构造库中的 AWS 构造

通过逃生舱口、原始覆盖和自定义资源自定义 AWS 构造库中的构造。

主题

使用逃生舱口

AWS 构造库提供了不同抽象级别的结构。

在最高级别上，您的 AWS CDK 应用程序及其中的堆栈本身就是整个云基础架构或其中的很大一部分的抽象。可以对它们进行参数化以将其部署在不同的环境中或满足不同的需求。

抽象是设计和实现云应用程序的强大工具。 AWS CDK 使您不仅可以利用其抽象进行构建，还可以创建新的抽象。以现有的开源 L2 和 L3 结构为指导，您可以构建自己的 L2 和 L3 结构，以反映自己组织的最佳实践和观点。

没有哪个抽象是完美的，即使是好的抽象也无法涵盖所有可能的用例。在开发过程中，您可能会发现一种几乎符合您需求的构造，需要进行小规模或大规模的自定义。

因此， AWS CDK 提供了突破构造模型的方法。这包括迁移到较低级别的抽象或完全不同的模型。逃生舱口可以让你逃离 AWS CDK 范式，并以适合你需求的方式对其进行自定义。然后，你可以将你的更改封装在一个新的结构中，以抽象出底层的复杂性，为其他开发者提供一个干净的 API。

以下是您可以使用逃生舱口的情况示例：

- AWS 服务功能可通过获得 AWS CloudFormation，但没有 L2 结构。
- AWS 服务功能可通过获得 AWS CloudFormation，并且该服务有 L2 结构，但这些结构尚未公开该功能。由于 L2 结构由 CDK 团队策划，因此它们可能无法立即用于新功能。
- 该功能还未完全 AWS CloudFormation 可用。

  要确定某项功能是否可通过提供 AWS CloudFormation，请参阅AWS 资源和属性类型参考。

为 L1 构造开发逃生舱口

如果 L2 构造不可用于服务，则可以使用自动生成的 L1 构造。这些资源可以通过以开头的名称进行识别`Cfn`，例如`CfnBucket`或`CfnRole`。您可以完全按照使用等效 AWS CloudFormation 资源的方式来实例化它们。

例如，要实例化启用分析功能的低级 Amazon S3 存储桶 L1，您需要编写如下内容。

TypeScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config',
      // ...
    }
  ]
});
```

JavaScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config'
      // ...
    }
  ]
});
```

Python

```
s3.CfnBucket(self, "MyBucket",
    analytics_configurations: [
        dict(id="Config",
             # ...
             )
    ]
)
```

Java

```
CfnBucket.Builder.create(this, "MyBucket")
    .analyticsConfigurations(Arrays.asList(java.util.Map.of(    // Java 9 or later
        "id", "Config", // ...
    ))).build();
```

C#

```
new CfnBucket(this, 'MyBucket', new CfnBucketProps {
    AnalyticsConfigurations = new Dictionary<string, string>
    {
        ["id"] = "Config",
        // ...
    }
});
```

在极少数情况下，你想定义一个没有相应CfnXxx类的资源。这可能是尚未在资源规范中发布的新 AWS CloudFormation 资源类型。在这种情况下，您可以cdk.CfnResource直接实例化并指定资源类型和属性。如以下示例所示。

TypeScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config',
        // ...
      }
    ]
  }
});
```

JavaScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config'
        // ...
      }
    ]
  }
});
```

Python

```
cdk.CfnResource(self, 'MyBucket',
  type="AWS::S3::Bucket",
  properties=dict(
    # Note the PascalCase here! These are CloudFormation identifiers.
    "AnalyticsConfigurations": [
      {
        "Id": "Config",
        # ...
      }
    ]
  }
```

```
)
```

Java

```
CfnResource.Builder.create(this, "MyBucket")
        .type("AWS::S3::Bucket")
        .properties(java.util.Map.of(    // Map.of requires Java 9 or later
            // Note the PascalCase here! These are CloudFormation identifiers
            "AnalyticsConfigurations", Arrays.asList(
                    java.util.Map.of("Id", "Config", // ...
                    ))))
        .build();
```

C#

```
new CfnResource(this, "MyBucket", new CfnResourceProps
{
    Type = "AWS::S3::Bucket",
    Properties = new Dictionary<string, object>
    {   // Note the PascalCase here! These are CloudFormation identifiers
        ["AnalyticsConfigurations"] = new Dictionary<string, string>[]
        {
            new Dictionary<string, string> {
                ["Id"] = "Config"
            }
        }
    }
});
```

为 L2 构造开发逃生舱口

如果 L2 构造缺少功能或您正在尝试解决问题，则可以修改由 L2 构造封装的 L1 构造。

所有 L2 构造都包含相应的 L1 构造。例如，高级`Bucket`构造封装了低级`CfnBucket`构造。由于与 AWS CloudFormation 资源直接`CfnBucket`对应，因此它公开了所有可用的 AWS CloudFormation功能。

访问 L1 构造的基本方法是使用 `construct.node.defaultChild` (Python:`default_child`)，将其转换为正确的类型（如有必要），然后修改其属性。再说一遍，让我们以 a 为例`Bucket`。

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }
];
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config'
    // ...
  }
];
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
    {
        "id": "Config",
        # ...
    }
]
```

Java

```
// Get the CloudFormation resource
```

```
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations(
        Arrays.asList(java.util.Map.of(    // Java 9 or later
            "Id", "Config", // ...
        ));
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List<object> {
    new Dictionary<string, string>
    {
        ["Id"] = "Config",
        // ...
    }
};
```

您也可以使用此对象来更改诸如`Metadata`和之类的 AWS CloudFormation 选项`UpdatePolicy`。

TypeScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

JavaScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

Python

```
cfn_bucket.cfn_options.metadata = {
    "MetadataKey": "MetadataValue"
}
```

Java

```
cfnBucket.getCfnOptions().setMetadata(java.util.Map.of(    // Java 9+
    "MetadataKey", "Metadatavalue"));
```

C#

```
cfnBucket.CfnOptions.Metadata = new Dictionary<string, object>
{
    ["MetadataKey"] = "Metadatavalue"
};
```

Un-scape 舱口

AWS CDK 还提供了上升抽象级别的功能，我们可以将其称为 “un-escape” 舱口。如果您有 L1 构造（例如）CfnBucket，则可以创建一个新的 L2 构造（在本例Bucket中）来封装 L1 构造。

当您创建 L1 资源但又想将其与需要 L2 资源的结构一起使用时，这很方便。当你想使用 L1 构造中没有的便捷方法时.grantXxxxx()，这也很有用。

您可以在 L2 类上使用名为（例如，A Bucket.fromCfnBucket() mazon S3 存储桶）的静态方法移.fromCfnXxxxx()至更高的抽象级别。L1 资源是唯一的参数。

TypeScript

```
b1 = new s3.CfnBucket(this, "buck09", { ... });
b2 = s3.Bucket.fromCfnBucket(b1);
```

JavaScript

```
b1 = new s3.CfnBucket(this, "buck09", { ...} );
b2 = s3.Bucket.fromCfnBucket(b1);
```

Python

```
b1 = s3.CfnBucket(self, "buck09", ...)
 b2 = s3.from_cfn_bucket(b1)
```

Java

```
CfnBucket b1 = CfnBucket.Builder.create(this, "buck09")
        // ....
                          .build();
IBucket b2 = Bucket.fromCfnBucket(b1);
```

C#

```
var b1 = new CfnBucket(this, "buck09", new CfnBucketProps { ... });
var v2 = Bucket.FromCfnBucket(b1);
```

从 L1 构造创建的 L2 构造是引用 L1 资源的代理对象，类似于根据资源名称、ARN 或查找创建的代理对象。对这些构造的修改不会影响最终合成的 AWS CloudFormation 模板（但是，由于您拥有 L1 资源，因此可以改为对其进行修改）。有关代理对象的更多信息，请参阅the section called “引用您 AWS 账户中的资源”。

为避免混淆，请勿创建引用同一 L1 构造的多个 L2 结构。例如，如果您`Bucket`使用上一节中的技术`CfnBucket`从中提取，则不应使用该方法调`Bucket.fromCfnBucket()`用来创建第二个`Bucket`实例`CfnBucket`。它实际上可以像你预期的那样工作（只有一个`AWS::S3::Bucket`是合成的），但它会使你的代码更难维护。

原始覆盖

如果 L1 构造中缺少某些属性，则可以使用原始覆盖来绕过所有输入。这也使得删除合成的属性成为可能。

使用其中一个`addOverride`方法 (Python:`add_override`)，如以下示例所示。

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');
```

```
// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild ;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Use dot notation to address inside the resource template fragment
cfn_bucket.add_override("Properties.VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_deletion_override("Properties.VersioningConfiguration.Status")

# use index (0 here) to address an element of a list
cfn_bucket.add_override("Properties.Tags.0.Value", "NewValue")
cfn_bucket.add_deletion_override("Properties.Tags.0")

# addPropertyOverride is a convenience function for paths starting with
 "Properties."
```

```
cfn_bucket.add_property_override("VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_property_deletion_override("VersioningConfiguration.Status")
cfn_bucket.add_property_override("Tags.0.Value", "NewValue")
cfn_bucket.add_property_deletion_override("Tags.0")
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.addDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.addOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.addDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.addPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.addPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.addPropertyDeletionOverride("Tags.0");
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment
cfnBucket.AddOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.AddOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.AddDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.AddPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.AddPropertyOverride("Tags.0.Value", "NewValue");
```

```
cfnBucket.AddPropertyDeletionOverride("Tags.0");
```

自定义资源

如果该功能无法通过直接 API 调用 AWS CloudFormation，而只能通过直接 API 调用，则必须编写 AWS CloudFormation 自定义资源来进行所需的 API 调用。您可以使用 AWS CDK 来编写自定义资源并将其封装到常规构造接口中。从你构造的消费者的角度来看，体验会让人感觉很原生。

构建自定义资源涉及编写一个 Lambda 函数来响应资源的CREATEUPDATE、和DELETE生命周期事件。如果您的自定义资源只需要进行一次 API 调用，请考虑使用AwsCustomResource。这使得在 AWS CloudFormation 部署期间可以执行任意 SDK 调用。否则，您应该编写自己的 Lambda 函数来执行需要完成的工作。

主题过于宽泛，无法在此处全面介绍，但是以下链接应该可以帮助您入门：

- 自定义资源
- 自定义资源示例
- 有关更完整的示例，请参阅 CDK 标准库中的DnsValidatedCertificate类。这是作为自定义资源实现的。

从环境变量中获取值

要获取环境变量的值，请使用如下代码。这段代码获取环境变量的值MYBUCKET。

TypeScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

JavaScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;
```

```
// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

Python

```
import os

# Raises KeyError if environment variable doesn't exist
bucket_name = os.environ["MYBUCKET"]

# Sets bucket_name to None if environment variable doesn't exist
bucket_name = os.getenv("MYBUCKET")

# Sets bucket_name to a default if env var doesn't exist
bucket_name = os.getenv("MYBUCKET", "DefaultName")
```

Java

```
// Sets bucketName to null if environment variable doesn't exist
String bucketName = System.getenv("MYBUCKET");

// Sets bucketName to a default if env var doesn't exist
String bucketName = System.getenv("MYBUCKET");
if (bucketName == null) bucketName = "DefaultName";
```

C#

```
using System;

// Sets bucket name to null if environment variable doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET");

// Sets bucket_name to a default if env var doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET") ?? "DefaultName";
```

使用一个 AWS CloudFormation 值

有关the section called “参数”在中使用 AWS CloudFormation 参数的信息，请参阅 AWS CDK。

要获取对现有 AWS CloudFormation 模板中资源的引用，请参阅the section called “导入 AWS CloudFormation 模板”。

导入现有 AWS CloudFormation 模板

使用`cloudformation-include.CfnInclude`构造将资源转换为 L1 结构，将资源从 AWS CloudFormation 模板导入到 AWS Cloud Development Kit (AWS CDK) 应用程序中。

导入后，您可以在应用程序中使用这些资源，方法与最初在 AWS CDK 代码中定义这些资源的方式相同。你也可以在更高级别 AWS CDK 的构造中使用这些 L1 结构。例如，这可以让你将 L2 权限授予方法与它们定义的资源一起使用。

该`cloudformation-include.CfnInclude`构造本质上是向 AWS CloudFormation 模板中的任何资源添加一个 AWS CDK API 封装器。使用此功能一次将现有 AWS CloudFormation 模板导入到 AWS CDK 一个片段中。通过这样做，您可以使用 AWS CDK 构造来管理现有资源，从而利用更高级别抽象的优势。您还可以使用此功能通过提供 AWS CDK 构造 API 将 AWS CloudFormation 模板出售给 AWS CDK 开发人员。

> **Note**
>
> AWS CDK 还包括 v1 `aws-cdk-lib.CfnInclude`，它以前用于相同的通用用途。但是，它缺少的许多功能`cloudformation-include.CfnInclude`。

主题

导入 AWS CloudFormation 模板

以下是示例 AWS CloudFormation 模板，我们将使用该模板在本主题中提供示例。复制并保存模板，`my-template.json`以便后续操作。完成这些示例后，您可以使用任何现有已部署的 AWS CloudFormation 模板进一步探索。您可以从 AWS CloudFormation 控制台获取它们。

```
{
  "Resources": {
    "MyBucket": {
```

```
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "MyBucket",
      }
    }
  }
}
```

你可以使用 JSON 或 YAML 模板。如果有的话，我们建议使用 JSON，因为 YAML 解析器接受的内容可能略有不同。

以下是如何使用将示例模板导入 AWS CDK 应用程序的示例`cloudformation-include`。模板是在 CDK 堆栈的上下文中导入的。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as cfninc from 'aws-cdk-lib/cloudformation-include';
import { Construct } from 'constructs';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const cfninc = require('aws-cdk-lib/cloudformation-include');

class MyStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
```

```
}

module.exports = { MyStack }
```

Python

```
import aws_cdk as cdk
from aws_cdk import cloudformation_include as cfn_inc
from constructs import Construct

class MyStack(cdk.Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        template = cfn_inc.CfnInclude(self, "Template",
            template_file="my-template.json")
```

Java

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.cloudformation.include.CfnInclude;
import software.constructs.Construct;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        CfnInclude template = CfnInclude.Builder.create(this, "Template")
         .templateFile("my-template.json")
         .build();
    }
}
```

C#

```
using Amazon.CDK;
```

```
using Constructs;
using cfnInc = Amazon.CDK.CloudFormation.Include;

namespace MyApp
{
    public class MyStack : Stack
    {
        internal MyStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            var template = new cfnInc.CfnInclude(this, "Template", new
 cfnInc.CfnIncludeProps
            {
                TemplateFile = "my-template.json"
            });
        }
    }
}
```

默认情况下，导入资源会保留模板中资源的原始逻辑 ID。此行为适用于将 AWS CloudFormation 模板导入到 AWS CDK，其中必须保留逻辑 ID。 AWS CloudFormation 需要这些信息才能将这些导入的资源识别为 AWS CloudFormation 模板中的相同资源。

如果您正在为模板开发 AWS CDK 构造封装器以便其他 AWS CDK 开发者可以使用，请改为 AWS CDK 生成新的资源 ID。通过这样做，可以在堆栈中多次使用该构造，而不会出现名称冲突。为此，请在导入模板`false`时将该`preserveLogicalIds`属性设置为。以下是 示例：

TypeScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    preserve_logical_ids=False)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .preserveLogicalIds(false)
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfn_inc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    PreserveLogicalIds = false
});
```

要将导入的资源置于您的 AWS CDK 应用程序的控制之下，请将堆栈添加到App：

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { MyStack } from '../lib/my-stack';

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { MyStack } = require('../lib/my-stack');

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

Python

```
import aws_cdk as cdk
from mystack.my_stack import MyStack

app = cdk.App()
MyStack(app, "MyStack")
```

Java

```
import software.amazon.awscdk.App;

public class MyApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyStack(app, "MyStack");
    }
}
```

C#

```
using Amazon.CDK;

namespace CdkApp
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyStack(app, "MyStack");
        }
    }
}
```

要验证堆栈中的 AWS 资源不会发生任何意想不到的更改，您可以执行差异。使用 AWS CDK CLIcdk diff命令并省略任何 AWS CDK特定的元数据。以下是 示例：

```
cdk diff --no-version-reporting --no-path-metadata --no-asset-metadata
```

导入 AWS CloudFormation 模板后，该 AWS CDK 应用程序应成为您导入资源的真实来源。要对资源进行更改，请在 AWS CDK 应用程序中对其进行修改，然后使用 AWS CDK CLIcdk deploy命令进行部署。

访问导入的资源

示例代码`template`中的名称代表导入的 AWS CloudFormation 模板。要从中访问资源，请使用对象`getResource()`的方法。要将返回的资源作为特定类型的资源进行访问，请将结果转换为所需的类型。在 Python 中或者，这不是必需 JavaScript的。以下是 示例：

TypeScript

```
const cfnBucket = template.getResource('MyBucket') as s3.CfnBucket;
```

JavaScript

```
const cfnBucket = template.getResource('MyBucket');
```

Python

```
cfn_bucket = template.get_resource("MyBucket")
```

Java

```
CfnBucket cfnBucket = (CfnBucket)template.getResource("MyBucket");
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
```

从这个例子中，现在`cfnBucket`是该`aws-s3.CfnBucket`类的实例。这是代表相应 AWS CloudFormation 资源的 L1 结构。你可以像对待任何其他同类资源一样对待它。例如，您可以使用属性获取其 ARN 值。`bucket.attrArn`

要改为将 L1 `CfnBucket` 资源封装在 L2 `aws-s3.Bucket`实例中，请使用静态方法`fromBucketArn()``fromBucketAttributes()`、或。`fromBucketName()`通常，该`fromBucketName()`方法最方便。以下是 示例：

TypeScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

JavaScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

Python

```
bucket = s3.Bucket.from_bucket_name(self, "Bucket", cfn_bucket.ref)
```

Java

```
Bucket bucket = (Bucket)Bucket.fromBucketName(this, "Bucket", cfnBucket.getRef());
```

C#

```
var bucket = (Bucket)Bucket.FromBucketName(this, "Bucket", cfnBucket.Ref);
```

其他 L2 构造也有类似的方法，用于从现有资源创建构造。

当你将 L1 构造封装在 L2 构造中时，它不会创建新的资源。从我们的示例来看，我们没有创建第二个 S3; 存储桶。相反，新`Bucket`实例封装了现有实例。`CfnBucket`

从示例中可以看出，现在`bucket`是一个 L2 `Bucket` 构造，其行为与任何其他 L2 构造类似。例如，您可以使用存储桶的便捷`grantWrite()`方法向 AWS Lambda 函数授予对该存储桶的写入权限。您不必手动定义必要的 AWS Identity and Access Management (IAM) 策略。以下是 示例：

TypeScript

```
bucket.grantWrite(lambdaFunc);
```

JavaScript

```
bucket.grantWrite(lambdaFunc);
```

Python

```
bucket.grant_write(lambda_func)
```

Java

```
bucket.grantWrite(lambdaFunc);
```

C#

```
bucket.GrantWrite(lambdaFunc);
```

替换参数

如果您的 AWS CloudFormation 模板包含参数，则可以在导入时使用`parameters`属性将其替换为构建时值。在以下示例中，我们将`UploadBucket`参数替换为代码中其他地方定义的存储桶的 ARN。AWS CDK

TypeScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
```

```
    template_file="my-template.json",
    parameters=dict(UploadBucket=bucket.bucket_arn)
)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .parameters(java.util.Map.of(    // Map.of requires Java 9+
   "UploadBucket", bucket.getBucketArn()))
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfnInc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    Parameters = new Dictionary<string, string>
    {
        { "UploadBucket", bucket.BucketArn }
    }
});
```

其他模板元素

您可以导入任何 AWS CloudFormation 模板元素，而不仅仅是资源。导入的元素将成为 AWS CDK 堆栈的一部分。要导入这些元素，请使用`CfnInclude`对象的以下方法：

- `getCondition()`— AWS CloudFormation 条件。
- `getHook()`— 用于蓝/绿部署的 AWS CloudFormation 挂钩。
- `getMapping()`— AWS CloudFormation 映射。
- `getOutput()`— AWS CloudFormation 输出。
- `getParameter()`— AWS CloudFormation 参数。
- `getRule()`— AWS Service Catalog 模板 AWS CloudFormation 规则。

这些方法中的每一个都返回一个表示特定 AWS CloudFormation 元素类型的类的实例。这些对象是可变的。您对它们所做的更改将显示在从 AWS CDK 堆栈生成的模板中。以下是从模板导入参数并修改其默认值的示例：

TypeScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

JavaScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

Python

```
param = template.get_parameter("MyParameter")
param.default = "AWS CDK"
```

Java

```
CfnParameter param = template.getParameter("MyParameter");
param.setDefaultValue("AWS CDK")
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
var param = template.GetParameter("MyParameter");
param.Default = "AWS CDK";
```

嵌套堆栈

您可以通过在导入嵌套堆栈的主模板时或稍后指定嵌套堆栈来导入它们。嵌套模板必须存储在本地文件中，但在主模板中作为NestedStack资源引用。此外， AWS CDK 代码中使用的资源名称必须与主模板中用于嵌套堆栈的名称相匹配。

鉴于主模板中的此资源定义，以下代码显示了如何双向导入引用的嵌套堆栈。

```
"NestedStack": {
  "Type": "AWS::CloudFormation::Stack",
  "Properties": {
    "TemplateURL": "https://my-s3-template-source.s3.amazonaws.com/nested-stack.json"
  }
```

TypeScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedTemplate', {
  templateFile: 'nested-template.json',
});
```

JavaScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedStack', {
  templateFile: 'my-nested-template.json',
});
```

Python

```
# include nested stack when importing main stack
main_template = cfn_inc.CfnInclude(self, "MainStack",
    template_file="main-template.json",
    load_nested_stacks=dict(NestedStack=
        cfn_inc.CfnIncludeProps(template_file="nested-template.json")))

# or add it some time after importing the main stack
```

```
nested_template = main_template.load_nested_stack("NestedStack",
    template_file="nested-template.json")
```

Java

```
CfnInclude mainTemplate = CfnInclude.Builder.create(this, "MainStack")
 .templateFile("main-template.json")
 .loadNestedStacks(java.util.Map.of(    // Map.of requires Java 9+
    "NestedStack", CfnIncludeProps.builder()
    .templateFile("nested-template.json").build()))
 .build();

// or add it some time after importing the main stack
IncludedNestedStack nestedTemplate = mainTemplate.loadNestedStack("NestedTemplate",
 CfnIncludeProps.builder()
  .templateFile("nested-template.json")
  .build());
```

C#

```
// include nested stack when importing main stack
var mainTemplate = new cfnInc.CfnInclude(this, "MainStack", new
 cfnInc.CfnIncludeProps
{
    TemplateFile = "main-template.json",
    LoadNestedStacks = new Dictionary<string, cfnInc.ICfnIncludeProps>
    {
        { "NestedStack", new cfnInc.CfnIncludeProps { TemplateFile = "nested-
template.json" } }
    }
});

// or add it some time after importing the main stack
var nestedTemplate = mainTemplate.LoadNestedStack("NestedTemplate", new
 cfnInc.CfnIncludeProps {
    TemplateFile = 'nested-template.json'
});
```

您可以使用任一方法导入多个嵌套堆栈。导入主模板时，您需要提供每个嵌套堆栈的资源名称与其模板文件之间的映射。此映射可以包含任意数量的条目。要在初始导入后执行此操作，请为每个嵌套堆栈调用`loadNestedStack()`一次。

导入嵌套堆栈后，您可以使用主模板的getNestedStack()方法对其进行访问。

TypeScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

JavaScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

Python

```
nested_stack = main_template.get_nested_stack("NestedStack").stack
```

Java

```
NestedStack nestedStack = mainTemplate.getNestedStack("NestedStack").getStack();
```

C#

```
var nestedStack = mainTemplate.GetNestedStack("NestedStack").Stack;
```

该getNestedStack()方法返回一个IncludedNestedStack实例。在此实例中，您可以通过stack属性访问 AWS CDK NestedStack实例，如示例所示。您也可以通过访问原始 AWS CloudFormation 模板对象includedTemplate，从中加载资源和其他 AWS CloudFormation 元素。

从 Systems Manager 参数存储库中获取值

AWS Cloud Development Kit (AWS CDK) 可以检索 AWS Systems Manager 参数存储属性的值。在合成过程中， AWS CDK 会生成一个令牌，该令牌 AWS CloudFormation 在部署期间由解析。

AWS CDK 支持检索普通值和安全值。您可以请求任何一种值的特定版本。对于纯值，您可以在检索最新版本的请求中省略该版本。对于安全值，您必须在请求安全属性的值时指定版本。

> **Note**
>
> 本主题介绍如何从 AWS Systems Manager 参数存储区读取属性。您也可以从中读取机密 AWS Secrets Manager （请参阅从中获取值 AWS Secrets Manager）。

主题

在部署时读取 Systems Manager 的值

要从 Systems Manager 参数存储区读取值，请使用valueForString参数和valueForSecureStringParameter方法。根据你想要的属性是纯字符串还是安全字符串值来选择方法。这些方法返回令牌，而不是实际值。该值在部署 AWS CloudFormation 期间由解析。以下是 示例：

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

Python

```
import aws_cdk.aws_ssm as ssm

# Get latest version or specified version of plain string attribute
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name")
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name", 1)

# Get specified version of secure string attribute
secure_string_token = ssm.StringParameter.value_for_secure_string_parameter(
    self, "my-secure-parameter-name", 1)   # must specify version
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

//Get latest version or specified version of plain string attribute
String latestStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name");       // latest version
String versionOfStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name", 1);    // version 1

//Get specified version of secure string attribute
String secureStringToken = StringParameter.valueForSecureStringParameter(
            this, "my-secure-parameter-name", 1);   // must specify version
```

C#

```
using Amazon.CDK.AWS.SSM;

// Get latest version or specified version of plain string attribute
var latestStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name");      // latest version
var versionOfStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name", 1);   // version 1

// Get specified version of secure string attribute
var secureStringToken = StringParameter.ValueForSecureStringParameter(
    this, "my-secure-parameter-name", 1);   // must specify version
```

目前支持此功能的 AWS 服务数量有限。

在合成时读取 Systems Manager 的值

有时，在合成时提供参数会很有用。通过这样做， AWS CloudFormation 模板将始终使用相同的值，而不是在部署期间解析该值。

要在合成时从 Systems Manager 参数存储区读取值，请使用valueFromLookup方法 (Python:`value_from_lookup`)。此方法将参数的实际值作为the section called “上下文”值返回。如果该值尚未缓存到命令行中`cdk.json`或未通过命令行传递，则会从当前 AWS 账户中检索该值。因此，必须使用明确的 AWS 环境信息来合成堆栈。

以下是 示例：

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

Python

```
import aws_cdk.aws_ssm as ssm

string_value = ssm.StringParameter.value_from_lookup(self, "my-plain-parameter-
name")
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

String stringValue = StringParameter.valueFromLookup(this, "my-plain-parameter-
name");
```

C#

```
using Amazon.CDK.AWS.SSM;

var stringValue = StringParameter.ValueFromLookup(this, "my-plain-parameter-name");
```

只能检索普通的 Systems Manager 字符串。无法检索安全字符串。将始终返回最新版本。无法请求特定版本。

> **Important**
>
> 检索到的值将最终出现在您的合成 AWS CloudFormation 模板中。这可能存在安全风险，具体取决于谁有权访问您的 AWS CloudFormation 模板以及模板的价值。通常，请勿将此功能用于密码、密钥或其他要保密的值。

将值写入 Systems Manager

您可以使用 AWS CLI AWS Management Console、或 AWS SDK 来设置 Systems Manager 的参数值。以下示例使用 ssm put-parameter CLI 命令。

```
aws ssm put-parameter --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --name "secure-parameter-name" --type "SecureString" --value
 "secure-parameter-value"
```

更新已存在的 SSM 值时，还要包括该`--overwrite`选项。

```
aws ssm put-parameter --overwrite --name "parameter-name" --type "String" --value
 "parameter-value"
aws ssm put-parameter --overwrite --name "secure-parameter-name" --type "SecureString"
 --value "secure-parameter-value"
```

从中获取值 AWS Secrets Manager

要使用 AWS CDK 应用程序 AWS Secrets Manager 中的值，请使用 fromSecretAttributes() 方法。它表示从 Secrets Manager 检索并在 AWS CloudFormation 部署时使用的值。以下是 示例：

TypeScript

```
import * as sm from "aws-cdk-lib/aws-secretsmanager";

export class SecretsManagerStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
```

JavaScript

```
const sm = require("aws-cdk-lib/aws-secretsmanager");

class SecretsManagerStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
  }
}

module.exports = { SecretsManagerStack }
```

Python

```
import aws_cdk.aws_secretsmanager as sm

class SecretsManagerStack(cdk.Stack):
```

```
    def __init__(self, scope: cdk.App, id: str, **kwargs):
      super().__init__(scope, name, **kwargs)

      secret = sm.Secret.from_secret_attributes(self, "ImportedSecret",
          secret_complete_arn="arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>",
          # If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
          # encryption_key=....
      )
```

Java

```
import software.amazon.awscdk.services.secretsmanager.Secret;
import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

public class SecretsManagerStack extends Stack {
    public SecretsManagerStack(App scope, String id) {
        this(scope, id, null);
    }

    public SecretsManagerStack(App scope, String id, StackProps props) {
        super(scope, id, props);

        Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret",
 SecretAttributes.builder()
            .secretCompleteArn("arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>")
             // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
             // .encryptionKey(...)
             .build());
    }
}
```

C#

```
using Amazon.CDK.AWS.SecretsManager;

public class SecretsManagerStack : Stack
{
    public SecretsManagerStack(App scope, string id, StackProps props) : base(scope,
 id, props) {
```

```
        var secret = Secret.FromSecretAttributes(this, "ImportedSecret", new
 SecretAttributes {
            SecretCompleteArn = "arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>"
            // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
            // encryptionKey = ...,
        });
    }
```

Tip

使用 AWS CLI create-secret CLI 命令从命令行创建密钥，例如在测试时：

```
aws secretsmanager create-secret --name ImportedSecret --secret-string
 mygroovybucket
```

该命令返回一个 ARN，您可以将其用于前面的示例。

创建实例后，您可以从`Secret`实例的`secretValue`属性中获取密钥的值。该值由`SecretValue`实例表示，这是一种特殊类型the section called “令牌”。因为它是一个标志，所以它只有在解决之后才有意义。您的 CDK 应用程序无需访问其实际价值。相反，应用程序可以将`SecretValue`实例（或其字符串或数字表示形式）传递给任何需要该值的 CDK 方法。

设置 CloudWatch 闹铃

使用 aws-cloudwatch 软件包设置亚马逊指标 CloudWatch 警报。 CloudWatch 您可以使用预定义的指标或创建自己的指标。

主题

- 使用现有指标
- 创建自己的指标
- 创建警报

使用现有指标

许 AWS 多 Construct Library 模块允许您通过将指标名称传递给具有指标的对象实例上的便捷方法来对现有指标设置警报。例如，给定一个 Amazon SQS 队列，你可以ApproximateNumberOfMessagesVisible从队列的 metric () 方法中获取指标：

TypeScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

JavaScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

Python

```
metric = queue.metric("ApproximateNumberOfMessagesVisible")
```

Java

```
Metric metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

C#

```
var metric = queue.Metric("ApproximateNumberOfMessagesVisible");
```

创建自己的指标

按如下方式创建您自己的指标，其中命名空间值应类似于 Amazon SQ S 队列的 AWS/ SQS。您还需要指定指标的名称和维度：

TypeScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

JavaScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

Python

```
metric = cloudwatch.Metric(
    namespace="MyNamespace",
    metric_name="MyMetric",
    dimensionsMap=dict(MyDimension="MyDimensionValue")
)
```

Java

```
Metric metric = Metric.Builder.create()
        .namespace("MyNamespace")
        .metricName("MyMetric")
        .dimensionsMap(java.util.Map.of(    // Java 9 or later
            "MyDimension", "MyDimensionValue"))
        .build();
```

C#

```
var metric = new Metric(this, "Metric", new MetricProps
{
    Namespace = "MyNamespace",
    MetricName = "MyMetric",
    Dimensions = new Dictionary<string, object>
    {
        { "MyDimension", "MyDimensionValue" }
    }
});
```

创建警报

有了现有指标或您定义的指标后，就可以创建警报。在此示例中，当您的指标在过去三个评估周期中有两个超过100个时，就会发出警报。您可以通过酒店使用比较，例如在警报中使用小

于。comparisonOperatorG reater-than-or-equal-to 是 AWS CDK 默认值，因此我们不需要指定它。

TypeScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2
});
```

Python

```
alarm = cloudwatch.Alarm(self, "Alarm",
    metric=metric,
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
import software.amazon.awscdk.services.cloudwatch.Alarm;
import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
        .metric(metric)
        .threshold(100)
        .evaluationPeriods(3)
        .datapointsToAlarm(2).build();
```

C#

```
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
    Metric = metric,
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

创建警报的另一种方法是使用指标的 createAlarm () 方法，该方法采用的属性与Alarm构造函数基本相同。您无需传递该指标，因为它已经为人所知。

TypeScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

Python

```
metric.create_alarm(self, "Alarm",
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
metric.createAlarm(this, "Alarm", new CreateAlarmOptions.Builder()
        .threshold(100)
```

```
        .evaluationPeriods(3)
        .datapointsToAlarm(2)
        .build());
```

C#

```
metric.CreateAlarm(this, "Alarm", new CreateAlarmOptions
{
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

保存和检索上下文变量值

可以在cdk.json文件中使用 AWS Cloud Development Kit (AWS CDK) CLI或指定上下文变量。然后，使用TryGetContext方法检索值。

主题

- 指定上下文变量
- 检索上下文变量值

指定上下文变量

您可以将上下文变量指定为 AWS CDK CLI命令的一部分，也可以在中指定cdk.json。

要创建命令行上下文变量，请使用 --contex t (-c) 选项，如下例所示。

```
cdk synth -c bucket_name=mygroovybucket
```

要在cdk.json文件中指定相同的上下文变量和值，请使用以下代码。

```
{
  "context": {
    "bucket_name": "myotherbucket"
  }
}
```

如果您同时使用 AWS CDK CLI和cdk.json文件指定上下文变量，则该 AWS CDK CLI值优先。

检索上下文变量值

要获取应用程序中上下文变量的值，请在构造的上下文中使用该`TryGetContext`方法。（也就是说`this`，何时或`self`在 Python 中，是某个构造的实例。）

在此示例中，我们检索`bucket_name`上下文变量的值。如果未定义请求的值，则`TryGetContext`返回`undefined`（`None`在 Python 中；`null`在 Java 和 C# `nil` 中；在 Go 中），而不是引发异常。

TypeScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

JavaScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

Python

```
bucket_name = self.node.try_get_context("bucket_name")
```

Java

```
String bucketName = (String)this.getNode().tryGetContext("bucket_name");
```

C#

```
var bucketName = this.Node.TryGetContext("bucket_name");
```

在构造的上下文之外，你可以从应用程序对象访问上下文变量，如下所示。

TypeScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name')
```

JavaScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

Python

```
app = cdk.App()
bucket_name = app.node.try_get_context("bucket_name")
```

Java

```
App app = App();
String bucketName = (String)app.getNode().tryGetContext("bucket_name");
```

C#

```
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

有关使用上下文变量的更多详细信息，请参阅the section called “上下文”。

使用 AWS CloudFormation 公共登记处的资源

AWS CloudFormation 公共注册表允许您管理公共扩展和私有扩展，例如可在中使用的资源、模块和挂钩 AWS 账户。您可以通过`CfnResource`构造在 AWS Cloud Development Kit (AWS CDK) 应用程序中使用公共资源扩展。

要了解有关 AWS CloudFormation 公共注册表的更多信息，请参阅《AWS CloudFormation 用户指南》中的使用 AWS CloudFormation 注册表。

由发布的所有公共扩展 AWS 均可供所有地区的所有账户使用，您无需采取任何行动。但是，您必须在要使用的每个账户和地区中激活要使用的每个第三方扩展程序。

> **Note**
>
> 当您 AWS CloudFormation 使用第三方资源类型时，将产生费用。费用基于您每月运行的处理程序操作数量和处理程序操作持续时间。有关完整详情，请参阅CloudFormation 定价。

要了解有关公共扩展的更多信息，请参阅《AWS CloudFormation 用户指南》 CloudFormation中的 “使用公共扩展”

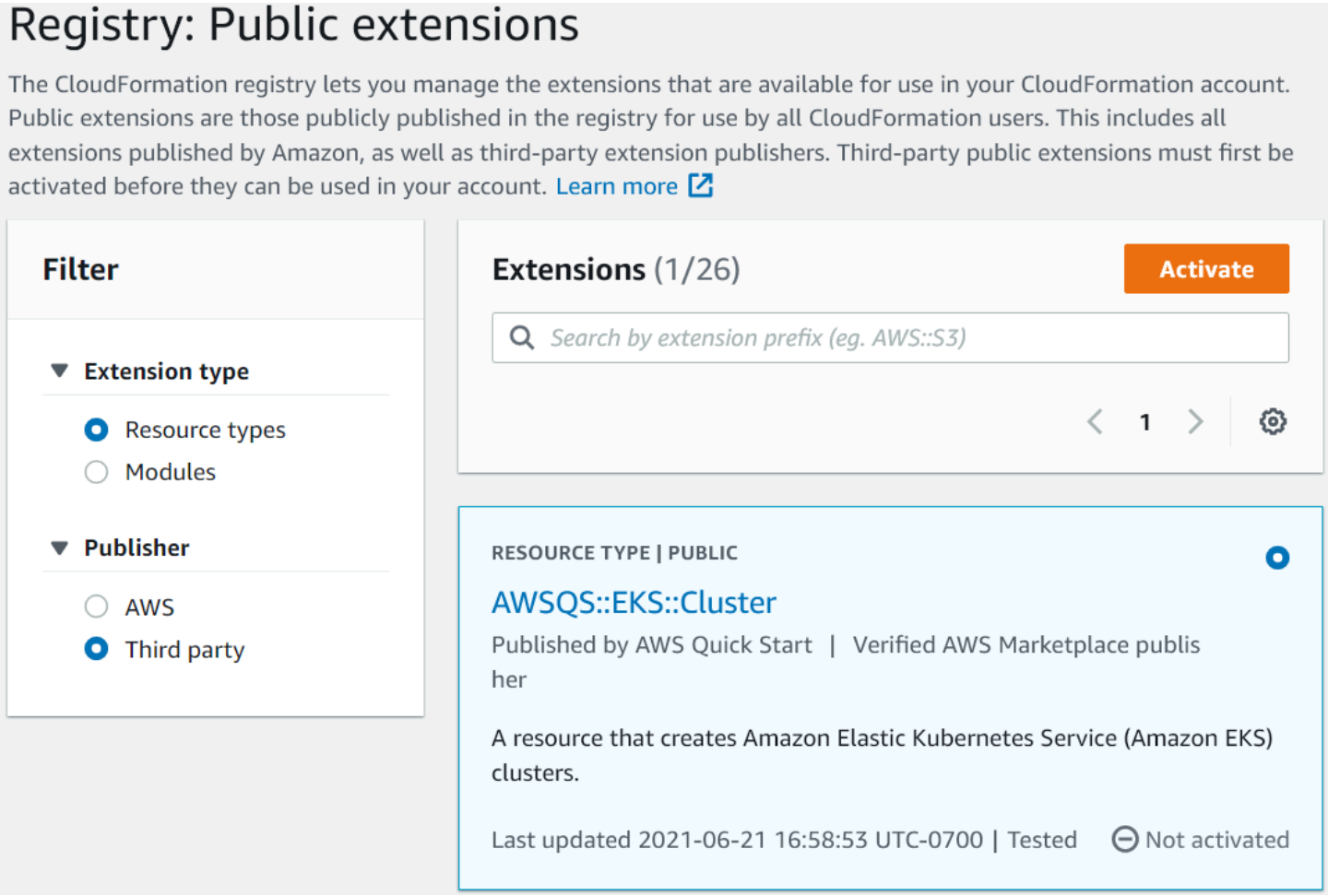
主题

- 在您的账户和地区中激活第三方资源
- 将 AWS CloudFormation 公共注册表中的资源添加到您的 CDK 应用程序

在您的账户和地区中激活第三方资源

由发布的扩展程序 AWS 不需要激活。它们始终适用于每个账户和地区。您可以通过 AWS Management Console AWS Command Line Interface、或通过部署特殊 AWS CloudFormation 资源来激活第三方扩展。

要通过激活第三方扩展程序 AWS Management Console 或查看有哪些资源可用



1. 登录您要使用该扩展程序的 AWS 账户，然后切换到要使用该扩展程序的区域。
2. 通过 “服务” 菜单导航到 CloudFormation 控制台。
3. 在导航栏上选择 “公共扩展”，然后激活 “发布者” 下的 “第三方” 单选按钮。随即会出现可用的第三方公共扩展列表。（您也可以选择AWS查看发布的公共扩展列表 AWS，但您无需激活它们。）

4. 浏览列表并找到您要激活的扩展程序。或者，搜索它，然后激活扩展卡片右上角的单选按钮。
5. 选择列表顶部的激活按钮以激活所选的扩展程序。此时将显示扩展程序的 “激活” 页面。
6. 在 “激活” 页面中，您可以覆盖扩展程序的默认名称并指定执行角色和日志配置。您还可以选择在新版本发布时是否自动更新扩展程序。根据需要设置这些选项后，请选择页面底部的激活扩展程序。

要使用激活第三方扩展程序 AWS CLI

- 使用 `activate-type` 命令。在标明的地方替换您要使用的自定义类型的 ARN。

  以下是 示例：

  ```
  aws cloudformation activate-type --public-type-arn public_extension_ARN --auto-update-activated
  ```

通过 CloudFormation 或 CDK 激活第三方扩展程序

- 部署类型为的资源`AWS::CloudFormation::TypeActivation`并指定以下属性：

  a. `TypeName`-类型的名称，例如`AWSQS::EKS::Cluster`。
  b. `MajorVersion`-你想要的扩展的主版本号。如果您想要最新版本，请省略。
  c. `AutoUpdate`-发行商发布新的次要版本时是否自动更新此扩展程序。（主要版本更新需要显式更改`MajorVersion`属性。）
  d. `ExecutionRoleArn`-运行此扩展程序的 IAM 角色的 ARN。
  e. `LoggingConfig`-扩展程序的日志配置。

  `TypeActivation`资源可以由 CDK 使用`CfnResource`构造部署。以下部分显示了实际扩展的内容。

将 AWS CloudFormation 公共注册表中的资源添加到您的 CDK 应用程序

使用该`CfnResource`构造将 AWS CloudFormation 公共注册表中的资源包含在您的应用程序中。这个构造在 CDK 的`aws-cdk-lib`模块中。

例如，假设有一个名为`MY::S5::UltimateBucket`的公共资源要在 AWS CDK 应用程序中使用。此资源采用一个属性：存储桶名称。相应的`CfnResource`实例化如下所示。

TypeScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

JavaScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

Python

```
ubucket = CfnResource(self, "MyUltimateBucket",
            type="MY::S5::UltimateBucket::MODULE",
            properties=dict(
                BucketName="UltimateBucket"))
```

Java

```
CfnResource.Builder.create(this, "MyUltimateBucket")
 .type("MY::S5::UltimateBucket::MODULE")
 .properties(java.util.Map.of(    // Map.of requires Java 9+
     "BucketName", "UltimateBucket"))
 .build();
```

C#

```
new CfnResource(this, "MyUltimateBucket", new CfnResourceProps
{
    Type = "MY::S5::UltimateBucket::MODULE",
    Properties = new Dictionary<string, object>
    {
        ["BucketName"] = "UltimateBucket"
```

```
    }
});
```

部署 AWS CDK 应用程序

部署 AWS Cloud Development Kit (AWS CDK) 应用程序。

主题

AWS CDK 在综合时进行策略验证

主题

在综合时进行策略验证

如果您或您的组织使用任何策略验证工具（例如AWS CloudFormation Guard或 OPA）来定义 AWS CloudFormation 模板的限制，则可以将它们与 AWS CDK at 综合时集成。通过使用相应的策略验证插件，您可以让 AWS CDK 应用程序在合成后立即根据您的策略对生成的 AWS CloudFormation 模板进行检查。如果有任何违规行为，合成将失败，并将报告打印到控制台。

AWS CDK 在合成时执行的验证可在部署生命周期的某一时刻验证控制，但它们不能影响合成之外发生的操作。示例包括直接在控制台中或通过服务 API 执行的操作。它们对合成后 AWS CloudFormation 模板的改变没有抵抗力。其他一些更权威地验证同一规则集的机制应该独立设置，例如AWS CloudFormation 钩子或。AWS Config尽管如此，在开发过程中评估规则集的能力仍然很有用，因为它可以提高检测速度和开发人员的工作效率。 AWS CDK

AWS CDK 策略验证的目标是最大限度地减少开发过程中所需的设置量，并使其尽可能简单。

> **Note**
>
> 此功能被认为是实验性的，插件 API 和验证报告的格式将来都可能发生变化。

主题

适用于应用程序开发人员

要在应用程序中使用一个或多个验证插件，请使用以下policyValidationBeta1属性Stage：

```
import { CfnGuardValidator } from '@cdklabs/cdk-validator-cfnguard';
const app = new App({
  policyValidationBeta1: [
    new CfnGuardValidator()
  ],
});
// only apply to a particular stage
const prodStage = new Stage(app, 'ProdStage', {
  policyValidationBeta1: [...],
});
```

合成后，将立即调用以这种方式注册的所有插件，以验证在您定义的范围内生成的所有模板。特别是，如果您在App对象中注册模板，则所有模板都将接受验证。

> **Warning**
>
> 除了修改云程序集之外，插件还可以执行 AWS CDK 应用程序所能做的任何事情。他们可以从文件系统读取数据，访问网络等。作为插件的使用者，你有责任验证其使用是否安全。

AWS CloudFormation Guard 插件

使用该CfnGuardValidator插件可以AWS CloudFormation Guard用来执行策略验证。该CfnGuardValidator插件内置了一组精选AWS Control Tower 的主动控件。当前的规则集可以在项目文档中找到。如中所述在综合时进行策略验证，我们建议组织使用AWS CloudFormation 挂钩设置更权威的验证方法。

对于AWS Control Tower客户，可以在整个组织中部署同样的主动控制措施。当您在 AWS Control Tower 环境中启用 AWS Control Tower 主动控制时，这些控件可以阻止部署通过 AWS CloudFormation部署的不合规资源。有关托管主动控制及其工作原理的更多信息，请参阅AWS Control Tower 文档。

这些 AWS CDK 捆绑的控件和托管 AWS Control Tower 的主动控制最好一起使用。在这种情况下，您可以使用与 AWS Control Tower 云环境中相同的主动控制来配置此验证插件。然后，您可以通过在`cdk synth`本地运行来快速确信您的 AWS CDK 应用程序将通过 AWS Control Tower 控制。

验证报告

当你合成 AWS CDK 应用程序时，将调用验证器插件并打印结果。示例报告如下所示。

```
Validation Report (CfnGuardValidator)
-------------------------------------
(Summary)
######################################
# Status    # failure                #
######################################
# Plugin    # CfnGuardValidator      #
######################################
(Violations)
Ensure S3 Buckets are encrypted with a KMS CMK (1 occurrences)
Severity: medium
  Occurrences:

    - Construct Path: MyStack/MyCustomL3Construct/Bucket
    - Stack Template Path: ./cdk.out/MyStack.template.json
    - Creation Stack:
        ###  MyStack (MyStack)
             # Library: aws-cdk-lib.Stack
             # Library Version: 2.50.0
             # Location: Object.<anonymous> (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:25:20)
             ###  MyCustomL3Construct (MyStack/MyCustomL3Construct)
                  # Library: N/A - (Local Construct)
                  # Library Version: N/A
                  # Location: new MyStack (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:15:20)
                  ###  Bucket (MyStack/MyCustomL3Construct/Bucket)
                       # Library: aws-cdk-lib/aws-s3.Bucket
                       # Library Version: 2.50.0
                       # Location: new MyCustomL3Construct (/home/johndoe/tmp/cdk-tmp-
app/src/main.ts:9:20)
    - Resource Name: my-bucket
    - Locations:
      > BucketEncryption/ServerSideEncryptionConfiguration/0/
ServerSideEncryptionByDefault/SSEAlgorithm
```

```
  Recommendation: Missing value for key `SSEAlgorithm` - must specify `aws:kms`
  How to fix:
    > Add to construct properties for `cdk-app/MyStack/Bucket`
      `encryption: BucketEncryption.KMS`

Validation failed. See above reports for details
```

默认情况下，报告将以人类可读的格式打印。如果你想要 JSON 格式的报告，请`@aws-cdk/core:validationReportJson`通过 CLI 启用它，或者将其直接传递给应用程序：

```
const app = new App({
  context: { '@aws-cdk/core:validationReportJson': true },
});
```

或者，您可以使用项目目录中的`cdk.json`或`cdk.context.json`文件来设置此上下文键值对（请参阅运行时上下文）。

如果您选择 JSON 格式，则 AWS CDK 会将策略验证报告打印到云汇编目录`policy-validation-report.json`中名为的文件中。对于默认的、人类可读的格式，报告将打印到标准输出。

对于插件作者

插件

AWS CDK 核心框架负责注册和调用插件，然后显示格式化的验证报告。该插件的职责是充当 AWS CDK 框架和策略验证工具之间的转换层。可以用支持的任何语言创建插件 AWS CDK。如果您正在创建可能被多种语言使用的插件，则建议您在中创建插件，`TypeScript`以便您可以使用 JSII 以每 AWS CDK 种语言发布插件。

创建插件

AWS CDK 核心模块和您的策略工具之间的通信协议由`IPolicyValidationPluginBeta1`接口定义。要创建新插件，必须编写一个实现此接口的类。你需要实现两件事：插件名称（通过覆盖`name`属性）和`validate()`方法。

框架将调用`validate()`，传递一个`IValidationContextBeta1`对象。要验证的模板的位置由给出`templatePaths`。该插件应返回一个实例`ValidationPluginReportBeta1`。此对象表示用户在合成结束时将收到的报告。

```
validate(context: IPolicyValidationContextBeta1): PolicyValidationReportBeta1 {
```

```
  // First read the templates using context.templatePaths...
  // ...then perform the validation, and then compose and return the report.
  // Using hard-coded values here for better clarity:
  return {
    success: false,
    violations: [{
      ruleName: 'CKV_AWS_117',
      description: 'Ensure that AWS Lambda function is configured inside a VPC',
      fix: 'https://docs.bridgecrew.io/docs/ensure-that-aws-lambda-function-is-
configured-inside-a-vpc-1',
      violatingResources: [{
        resourceName: 'MyFunction3BAA72D1',
        templatePath: '/home/johndoe/myapp/cdk.out/MyService.template.json',
        locations: 'Properties/VpcConfig',
      }],
    }],
  };
}
```

请注意，不允许插件修改云端程序集中的任何内容。任何这样做的尝试都将导致合成失败。

如果你的插件依赖于外部工具，请记住，有些开发者可能还没有在他们的工作站中安装该工具。为了最大限度地减少摩擦，我们强烈建议您在插件包中提供一些安装脚本，以实现整个过程的自动化。更好的是，将该脚本作为软件包安装的一部分运行。例如npm，您可以将其添加到package.json文件中的postinstall脚本中。

处理豁免

如果您的组织有处理豁免的机制，则可以将其作为验证器插件的一部分来实现。

说明可能的豁免机制的示例场景：

- 组织有一项规则，即除非在某些情况下，否则不允许使用公共 Amazon S3 存储桶。
- 开发者正在创建属于其中一种情况的 Amazon S3 存储桶，并请求豁免（例如创建工单）。
- 安全工具知道如何从注册豁免的内部系统中读取

在这种情况下，开发人员将在内部系统中请求例外，然后需要某种方式 “注册” 该异常。再加上警卫插件示例，你可以创建一个处理豁免的插件，方法是筛选出在内部票务系统中具有匹配豁免的违规行为。

有关实现示例，请参阅现有插件。

- @cdklabs/cdk-validator-cfnguard

使用 CDK Pipelines 进行持续集成和交付 (CI/CD)

使用构造库中的 CDK Pi pel AWS ines 模块来配置应用程序 AWS CDK 的持续交付。当你将 CDK 应用程序的源代码提交到 AWS CodeCommitGitHub AWS CodeStar、或，CDK Pipelines 可以自动构建、测试和部署你的新版本。

CDK Pipelines 是自我更新的。如果您添加应用程序阶段或堆栈，则管道会自动重新配置以部署这些新阶段或堆栈。

> **Note**
>
> CDK Pipelines 支持两个 API。一个是在CDK Pipelines开发者预览版中提供的原始API。另一个是现代 API，它包含了在预览阶段收到的 CDK 客户反馈。本主题中的示例使用现代 API。有关两个支持的 API 之间的区别的详细信息，请参阅 GitHubaw s-cdk 存储库中的 CDK Pipelines 原始 API。

主题

引导您的环境 AWS

在使用 CDK Pipelines 之前，必须引导 AWS 要部署堆栈的环境。

CDK 管道至少涉及两个环境。第一个环境是配置管道的地方。第二个环境是您要将应用程序堆栈或阶段部署到的环境（阶段是相关的堆栈组）。这些环境可能相同，但最佳实践建议是在不同的环境中将各个阶段彼此隔离开来。

> **Note**
>
> the section called “正在引导”有关通过引导创建的资源类型以及如何自定义引导堆栈的更多信息，请参阅。

使用 CDK Pipelines 进行持续部署需要在 CDK Toolkit 堆栈中包含以下内容：

- Amazon Simple Storage Service (Amazon S3) 存储桶。
- 亚马逊 ECR 存储库。
- IAM 角色为管道的各个部分提供所需的权限。

如有必要，CDK Toolkit 将升级您现有的引导堆栈或创建一个新的引导堆栈。

要引导可以配置 AWS CDK 管道的环境，请按以下示例`cdk bootstrap`所示调用。如有必要，通过`npx`命令调用 AWS CDK Toolkit 会临时安装它。它还将使用当前项目中安装的 Toolkit 版本（如果有）。

`--cloudformation-execution-policies`指定将来执行 CDK Pipelines 部署所依据的策略的 ARN。默认`AdministratorAccess`策略可确保您的管道可以部署所有类型的 AWS 资源。如果您使用此政策，请确保您信任构成 AWS CDK 应用程序的所有代码和依赖项。

大多数组织都要求对自动化可以部署哪些类型的资源进行更严格的控制。请咨询组织内的相应部门，以确定您的管道应使用的政策。

如果您的默认配置 AWS 文件包含必要的身份验证配置和 AWS 区域，则可以省略该`--profile`选项。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

要引导管道将要部署 AWS CDK 应用程序的其他环境，请改用以下命令。该`--trust`选项指示哪个其他账户应有权将 AWS CDK 应用程序部署到此环境中。对于此选项，请指定管道的 AWS 账户 ID。

同样，如果您的默认配置 AWS 文件包含必要的身份验证配置和 AWS 区域，则可以省略该--profile选项。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 \
    --trust PIPELINE-ACCOUNT-NUMBER
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 ^
    --trust PIPELINE-ACCOUNT-NUMBER
```

> **Tip**
>
> 仅使用管理凭据来引导和配置初始管道。之后，使用管道本身，而不是本地计算机来部署更改。

如果您要升级旧版的引导环境，则在创建新存储桶时，之前的 Amazon S3 存储桶将成为孤立存储桶。使用 Amazon S3 控制台手动将其删除。

初始化项目

创建一个新的空GitHub项目，然后将其克隆到您的工作站`my-pipeline`目录中。（我们在本主题中的代码示例使用 GitHub。 您也可以使用 AWS CodeStar 或 AWS CodeCommit。）

```
git clone GITHUB-CLONE-URL my-pipeline
cd my-pipeline
```

> **Note**
>
> 您可以`my-pipeline`为应用程序的主目录使用其他名称。但是，如果这样做，则必须在本主题的后面部分调整文件和类名。这是因为 AWS CDK Toolkit 的一些文件和类名基于主目录的名称。

克隆后，照常初始化项目。

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

创建应用程序后，还要输入以下两个命令。它们会激活应用程序的 Python 虚拟环境并安装 AWS CDK 核心依赖项。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

如果您使用的是 IDE，则现在可以打开或导入项目。例如，在 Eclipse 中，选择 “文件” > “导入” > “Maven” > “现有的 Maven 项目”。确保将项目设置设置为使用 Java 8 (1.8)。

C#

```
cdk init app --language csharp
```

如果您使用的是 Visual Studio，请在`src`目录中打开解决方案文件。

Go

```
cdk init app --language go
```

创建应用程序后，还要输入以下命令来安装应用程序所需的 AWS 构造库模块。

```
go get
```

Important

请务必将您的`cdk.json`和`cdk.context.json`文件提交到源代码管理中。上下文信息（例如从您的 AWS 账户中检索的功能标志和缓存值）是项目状态的一部分。在其他环境中，这些值可能会有所不同，这可能会导致结果发生意外变化。有关更多信息，请参阅 the section called “上下文”。

定义管道

您的CDK Pipelines应用程序将包含至少两个堆栈：一个代表管道本身，一个或多个堆栈代表通过管道部署的应用程序。堆栈也可以分为几个阶段，您可以使用这些阶段将基础架构堆栈的副本部署到不同的环境。现在，我们将考虑管道，稍后再深入研究它将部署的应用程序。

该构造`CodePipeline`是表示 AWS CodePipeline 用作其部署引擎的 CDK Pipeline 的构造。在堆栈`CodePipeline`中实例化时，需要定义管道的源位置（例如 GitHub 存储库）。您还可以定义用于构建应用程序的命令。

例如，以下内容定义了一个管道，其源存储在存储 GitHub 库中。它还包括 TypeScript CDK 应用程序的构建步骤。在指示的地方填写有关您的 GitHub 存储库的信息。

Note

默认情况下，管道 GitHub 使用存储在 Secrets Manager 中的名字`github-token`下的个人访问令牌进行身份验证。

您还需要更新管道堆栈的实例化以指定 AWS 账户和区域。

TypeScript

在lib/my-pipeline-stack.ts（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}
```

在bin/my-pipeline.ts（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
#!/usr/bin/env node
import * as cdk from 'aws-cdk-lib';
import { MyPipelineStack } from '../lib/my-pipeline-stack';

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

JavaScript

在lib/my-pipeline-stack.js（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}

module.exports = { MyPipelineStack }
```

在bin/my-pipeline.js（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyPipelineStack } = require('../lib/my-pipeline-stack');

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

Python

在my-pipeline/my-pipeline-stack.py（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]
                        )
                    )
```

In `app.py`:

```
#!/usr/bin/env python3
import aws_cdk as cdk
from my_pipeline.my_pipeline_stack import MyPipelineStack

app = cdk.App()
MyPipelineStack(app, "MyPipelineStack",
    env=cdk.Environment(account="111111111111", region="eu-west-1")
)

app.synth()
```

Java

在`src/main/java/com/myorg/MyPipelineStack.java`（如果您的项目文件夹未命名，可能会有所不同`my-pipeline`）：

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
```

```
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
             .pipelineName("MyPipeline")
             .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
             .build();
    }
}
```

在src/main/java/com/myorg/MyPipelineApp.java（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MyPipelineApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyPipelineStack(app, "PipelineStack", StackProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build());
```

```
        app.synth();
    }
}
```

C#

在src/MyPipeline/MyPipelineStack.cs（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
using Amazon.CDK;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });
        }
    }
}
```

在src/MyPipeline/Program.cs（如果您的项目文件夹未命名，可能会有所不同my-pipeline）：

```
using Amazon.CDK;

namespace MyPipeline
{
    sealed class Program
```

```
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyPipelineStack(app, "MyPipelineStack", new StackProps
            {
                Env = new Amazon.CDK.Environment {
                    Account = "111111111111", Region = "eu-west-1" }
            });

            app.Synth();
        }
    }
}
```

您必须手动部署一次管道。之后，管道会从源代码存储库中保持最新状态。因此，请确保存储库中的代码是您要部署的代码。检查您的更改并推送到 GitHub，然后部署：

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

> **Tip**
>
> 现在您已经完成了初始部署，您的本地 AWS 帐户不再需要管理权限。这是因为对您的应用程序的所有更改都将通过管道进行部署。你所需要做的就是推到 GitHub。

申请阶段

要定义可以同时添加到管道中的多堆栈 AWS 应用程序，请定义一个子类。`Stage`（这与 CDK Pipelines 模块`CdkStage`中的不同。）

该阶段包含构成应用程序的堆栈。如果堆栈之间存在依赖关系，则堆栈将按正确的顺序自动添加到管道中。相互不依赖的堆栈是并行部署的。您可以通过调用`stack1.addDependency(stack2)`在堆栈之间添加依赖关系。

阶段接受默认`env`参数，该参数将成为其中的堆栈的默认环境。（堆栈仍然可以指定自己的环境。）。

通过调addStage()用实例将应用程序添加到管道中Stage。可以多次实例化一个阶段并将其添加到管道中，以定义 DTAP 或多区域应用程序管道的不同阶段。

我们将创建一个包含简单 Lambda 函数的堆栈，并将该堆栈放在一个阶段中。然后，我们将向管道中添加舞台，以便可以对其进行部署。

TypeScript

创建新文件lib/my-pipeline-lambda-stack.ts来存放包含 Lambda 函数的应用程序堆栈。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { Function, InlineCode, Runtime } from 'aws-cdk-lib/aws-lambda';

export class MyLambdaStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}
```

创建新文件lib/my-pipeline-app-stage.ts来容纳我们的舞台。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from "constructs";
import { MyLambdaStack } from './my-pipeline-lambda-stack';

export class MyPipelineAppStage extends cdk.Stage {

    constructor(scope: Construct, id: string, props?: cdk.StageProps) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}
```

编辑lib/my-pipeline-stack.ts以将舞台添加到我们的管道中。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';
import { MyPipelineAppStage } from './my-pipeline-app-stage';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));
  }
}
```

JavaScript

创建新文件lib/my-pipeline-lambda-stack.js来存放包含 Lambda 函数的应用程序堆栈。

```
const cdk = require('aws-cdk-lib');
const { Function, InlineCode, Runtime } = require('aws-cdk-lib/aws-lambda');

class MyLambdaStack extends cdk.Stack {
    constructor(scope, id, props) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}

module.exports = { MyLambdaStack }
```

创建新文件lib/my-pipeline-app-stage.js来容纳我们的舞台。

```
const cdk = require('aws-cdk-lib');
const { MyLambdaStack } = require('./my-pipeline-lambda-stack');

class MyPipelineAppStage extends cdk.Stage {

    constructor(scope, id, props) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}

module.exports = { MyPipelineAppStage };
```

编辑lib/my-pipeline-stack.ts以将舞台添加到我们的管道中。

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');
const { MyPipelineAppStage } = require('./my-pipeline-app-stage');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));

  }
}

module.exports = { MyPipelineStack }
```

Python

创建新文件my_pipeline/my_pipeline_lambda_stack.py来存放包含 Lambda 函数的应用程序堆栈。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.aws_lambda import Function, InlineCode, Runtime

class MyLambdaStack(cdk.Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        Function(self, "LambdaFunction",
            runtime=Runtime.NODEJS_18_X,
            handler="index.handler",
            code=InlineCode("exports.handler = _ => 'Hello, CDK';")
        )
```

创建新文件my_pipeline/my_pipeline_app_stage.py来容纳我们的舞台。

```
import aws_cdk as cdk
from constructs import Construct
from my_pipeline.my_pipeline_lambda_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        lambdaStack = MyLambdaStack(self, "LambdaStack")
```

编辑my_pipeline/my-pipeline-stack.py以将舞台添加到我们的管道中。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep
from my_pipeline.my_pipeline_app_stage import MyPipelineAppStage

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

```
        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]))

        pipeline.add_stage(MyPipelineAppStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))
```

Java

创建新文件src/main/java/com.myorg/MyPipelineLambdaStack.java来存放包含 Lambda 函数的应用程序堆栈。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.InlineCode;

public class MyPipelineLambdaStack extends Stack {
    public MyPipelineLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineLambdaStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        Function.Builder.create(this, "LambdaFunction")
          .runtime(Runtime.NODEJS_18_X)
          .handler("index.handler")
          .code(new InlineCode("exports.handler = _ => 'Hello, CDK';"))
          .build();

    }
```

```
}
```

创建新文件src/main/java/com.myorg/MyPipelineAppStage.java来容纳我们的舞台。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.Stage;
import software.amazon.awscdk.StageProps;

public class MyPipelineAppStage extends Stage {
    public MyPipelineAppStage(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineAppStage(final Construct scope, final String id, final
 StageProps props) {
        super(scope, id, props);

        Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
    }

}
```

编辑src/main/java/com.myorg/MyPipelineStack.java以将舞台添加到我们的管道中。

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.StageProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
```

```
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
            .pipelineName("MyPipeline")
            .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
            .build();

        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build()));
    }
}
```

C#

创建新文件src/MyPipeline/MyPipelineLambdaStack.cs来存放包含 Lambda 函数的应用程序堆栈。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.Lambda;

namespace MyPipeline
{
    class MyPipelineLambdaStack : Stack
    {
        public MyPipelineLambdaStack(Construct scope, string id, StackProps
 props=null) : base(scope, id, props)
        {
            new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "index.handler",
```

```
                Code = new InlineCode("exports.handler = _ => 'Hello, CDK';")
            });
        }
    }
}
```

创建新文件src/MyPipeline/MyPipelineAppStage.cs来容纳我们的舞台。

```
using Amazon.CDK;
using Constructs;

namespace MyPipeline
{
    class MyPipelineAppStage : Stage
    {
        public MyPipelineAppStage(Construct scope, string id, StageProps
 props=null) : base(scope, id, props)
        {
            Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
        }
    }
}
```

编辑src/MyPipeline/MyPipelineStack.cs以将舞台添加到我们的管道中。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
```

```
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });

            pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
            {
                Env = new Environment
                {
                    Account = "111111111111", Region = "eu-west-1"
                }
            }));
        }
    }
}
```

添加的每个应用程序阶段都会addStage()导致添加一个相应的管道阶段，该阶段由addStage()调用返回的StageDeployment实例表示。您可以通过调用其或方法向舞台添加部署前addPre()或addPost()部署后操作。

TypeScript

```
// import { ManualApprovalStep } from 'aws-cdk-lib/pipelines';

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

    testingStage.addPost(new ManualApprovalStep('approval'));
```

JavaScript

```
// const { ManualApprovalStep } = require('aws-cdk-lib/pipelines');

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

testingStage.addPost(new ManualApprovalStep('approval'));
```

Python

```
# from aws_cdk.pipelines import ManualApprovalStep

testing_stage = pipeline.add_stage(MyPipelineAppStage(self, "testing",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))

testing_stage.add_post(ManualApprovalStep('approval'))
```

Java

```
// import software.amazon.awscdk.pipelines.StageDeployment;
// import software.amazon.awscdk.pipelines.ManualApprovalStep;

StageDeployment testingStage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

testingStage.addPost(new ManualApprovalStep("approval"));
```

C#

```
var testingStage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new
 StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

testingStage.AddPost(new ManualApprovalStep("approval"));
```

您可以向 Wav e 添加阶段以并行部署它们，例如，将阶段部署到多个账户或区域时。

TypeScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

JavaScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

Python

```
wave = pipeline.add_wave("wave")
wave.add_stage(MyApplicationStage(self, "MyAppEU",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))
wave.add_stage(MyApplicationStage(self, "MyAppUS",
    env=cdk.Environment(account="111111111111", region="us-west-1")))
```

Java

```
// import software.amazon.awscdk.pipelines.Wave;
final Wave wave = pipeline.addWave("wave");
wave.addStage(new MyPipelineAppStage(this, "MyAppEU", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
        .build()));
wave.addStage(new MyPipelineAppStage(this, "MyAppUS", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("us-west-1")
```

```
                .build())
        .build()));
```

C#

```
var wave = pipeline.AddWave("wave");
wave.AddStage(new MyPipelineAppStage(this, "MyAppEU", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));
wave.AddStage(new MyPipelineAppStage(this, "MyAppUS", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "us-west-1"
    }
}));
```

测试部署

您可以向 CDK Pipeline 添加步骤，以验证您正在执行的部署。例如，您可以使用 CDK Pipeline 库ShellStep来执行以下任务：

- 正在尝试访问由 Lambda 函数支持的新部署的 Amazon API Gateway
- 通过发出 AWS CLI 命令来检查已部署资源的设置

在最简单的形式中，添加验证操作如下所示：

TypeScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

JavaScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

Python

```
# stage was returned by pipeline.add_stage

stage.add_post(ShellStep("validate",
  commands=[''../tests/validate.sh'']
))
```

Java

```
// stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create("validate")
        .commands(Arrays.asList("'../tests/validate.sh'"))
        .build());
```

C#

```
// stage was returned by pipeline.addStage

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Commands = new string[] { "'../tests/validate.sh'" }
}));
```

许多 AWS CloudFormation 部署都会导致生成名称不可预测的资源。因此，CDK Pipelines提供了一种在部署后 AWS CloudFormation 读取输出的方法。这使得将（例如）负载均衡器生成的 URL 传递给测试操作成为可能。

要使用输出，请公开您感兴趣的`CfnOutput`对象。然后，将其传递到步骤的`envFromCfnOutputs`属性中，使其作为该步骤中的环境变量可用。

TypeScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

JavaScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

Python

```
# given a stack lb_stack that exposes a load balancer construct as load_balancer
self.load_balancer_address = cdk.CfnOutput(lb_stack, "LbAddress",
    value=f"https://{lb_stack.load_balancer.load_balancer_dns_name}/")

# pass the load balancer address to a shell step
stage.add_post(ShellStep("lbaddr",
    env_from_cfn_outputs={"lb_addr": lb_stack.load_balancer_address}
    commands=["echo $lb_addr"]))
```

Java

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = CfnOutput.Builder.create(lbStack, "LbAddress")
                            .value(String.format("https://%s/",
```

```
                                        lbStack.loadBalancer.loadBalancerDnsName))
                                .build();

stage.addPost(ShellStep.Builder.create("lbaddr")
    .envFromCfnOutputs(     // Map.of requires Java 9 or later
        java.util.Map.of("lbAddr", loadBalancerAddress))
    .commands(Arrays.asList("echo $lbAddr"))
    .build());
```

C#

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = new CfnOutput(lbStack, "LbAddress", new CfnOutputProps
{
    Value = string.Format("https://{0}/", lbStack.loadBalancer.LoadBalancerDnsName)
});

stage.AddPost(new ShellStep("lbaddr", new ShellStepProps
{
    EnvFromCfnOutputs = new Dictionary<string, CfnOutput>
    {
        {  "lbAddr", loadBalancerAddress }
    },
    Commands = new string[] { "echo $lbAddr" }
}));
```

你可以直接在中编写简单的验证测试ShellStep，但是当测试超过几行时，这种方法就会变得笨拙。对于更复杂的测试，您可以ShellStep通过inputs属性将其他文件（例如完整的 shell 脚本或其他语言的程序）引入到。输入可以是任何具有输出的步骤，包括源（例如 GitHub 存储库）或其他ShellStep来源。

如果文件可以直接用于测试（例如，如果它们本身是可执行文件），则可以从源存储库中引入这些文件。在此示例中，我们将 GitHub 存储库声明为source（而不是将其作为其中的一部分内联实例化）。CodePipeline然后，我们将这个文件集同时传递给管道和验证测试。

TypeScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
```

```
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

JavaScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

Python

```
source   = CodePipelineSource.git_hub("OWNER/REPO", "main")

pipeline =  CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=ShellStep("Synth",
```

```
                    input=source,
                    commands=["npm install -g aws-cdk",
                        "python -m pip install -r requirements.txt",
                        "cdk synth"]))

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

stage.add_post(ShellStep("validate", input=source,
    commands=["sh ../tests/validate.sh"],
))
```

Java

```
final CodePipelineSource source = CodePipelineSource.gitHub("OWNER/REPO", "main");

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(ShellStep.Builder.create("Synth")
                .input(source)
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(source)
        .commands(Arrays.asList("sh ../tests/validate.sh"))
        .build());
```

C#

```
var source = CodePipelineSource.GitHub("OWNER/REPO", "main");

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
```

```
    PipelineName = "MyPipeline",
    Synth = new ShellStep("Synth", new ShellStepProps
    {
        Input = source,
        Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
    })
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = source,
    Commands = new string[] { "sh ../tests/validate.sh" }
}));
```

如果需要编译测试（这是合成的一部分），则从合成步骤中获取其他文件是合适的。

TypeScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
```

```
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

JavaScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, "test", {
  env: { account: "111111111111", region: "eu-west-1" }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

Python

```
synth_step = ShellStep("Synth",
                input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                commands=["npm install -g aws-cdk",
                  "python -m pip install -r requirements.txt",
                  "cdk synth"])

pipeline   = CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=synth_step)

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

# run a script that was compiled during synthesis
stage.add_post(ShellStep("validate",
```

```
    input=synth_step,
    commands=["node test/validate.js"],
))
```

Java

```
final ShellStep synth = ShellStep.Builder.create("Synth")
                            .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                            .commands(Arrays.asList("npm install -g aws-cdk", "cdk
 synth"))
                            .build();

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(synth)
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(synth)
        .commands(Arrays.asList("node ./tests/validate.js"))
        .build());
```

C#

```
var synth = new ShellStep("Synth", new ShellStepProps
{
    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
    Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
});

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = synth
});
```

```
var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = synth,
    Commands = new string[] { "node ./tests/validate.js" }
}));
```

安全说明

任何形式的持续交付都有固有的安全风险。根据责任 AWS 共担模式，您应对 AWS 云端信息的安全负责。CDK Pipelines库通过整合安全的默认设置和建模最佳实践，让你抢先一步。

但是，就其本质而言，需要高访问权限才能实现其预期目的的图书馆无法确保完全安全。您的组织之外有许多攻击媒介。 AWS

特别要记住以下几点：

- 注意你所依赖的软件。审查您在管道中运行的所有第三方软件，因为它可能会更改已部署的基础架构。
- 使用依赖锁定来防止意外升级。CDK Pip `package-lock.json` elines尊重`yarn.lock`并确保你的依赖关系是你所期望的。
- CDK Pipelines在您自己的账户中创建的资源上运行，这些资源的配置由开发人员通过管道提交代码来控制。因此，CDK Pipelines本身无法抵御试图绕过合规性检查的恶意开发者。如果您的威胁模型包括编写 CDK 代码的开发人员，则应有外部合规机制，例如 AWS CloudFormation Hook（预防性）或 C AWS onfig（反应式）， AWS CloudFormation 执行角色无权禁用这些机制。
- 生产环境的凭证应该是短期的。在启动和初始配置之后，开发人员根本不需要拥有账户凭证。可以通过管道部署更改。一开始就不需要凭证，从而减少证书泄露的可能性。

故障排除

在开始使用CDK Pipelines时，通常会遇到以下问题。

管道：内部故障

```
CREATE_FAILED  | AWS::CodePipeline::Pipeline | Pipeline/Pipeline
Internal Failure
```

检查您的 GitHub 访问令牌。它可能丢失了，或者可能没有访问存储库的权限。

密钥：策略包含一个或多个无效委托人的声明

```
CREATE_FAILED | AWS::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.
```

其中一个目标环境尚未使用新的引导程序堆栈进行引导。确保所有目标环境都已引导。

堆栈处于 ROLLBACK_COMPLETE 状态，无法更新。

```
Stack STACK_NAME is in ROLLBACK_COMPLETE state and can not be updated. (Service:
AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request
ID: ...)
```

堆栈在之前的部署中失败，并且处于不可重试的状态。从 AWS CloudFormation 控制台中删除堆栈，然后重试部署。

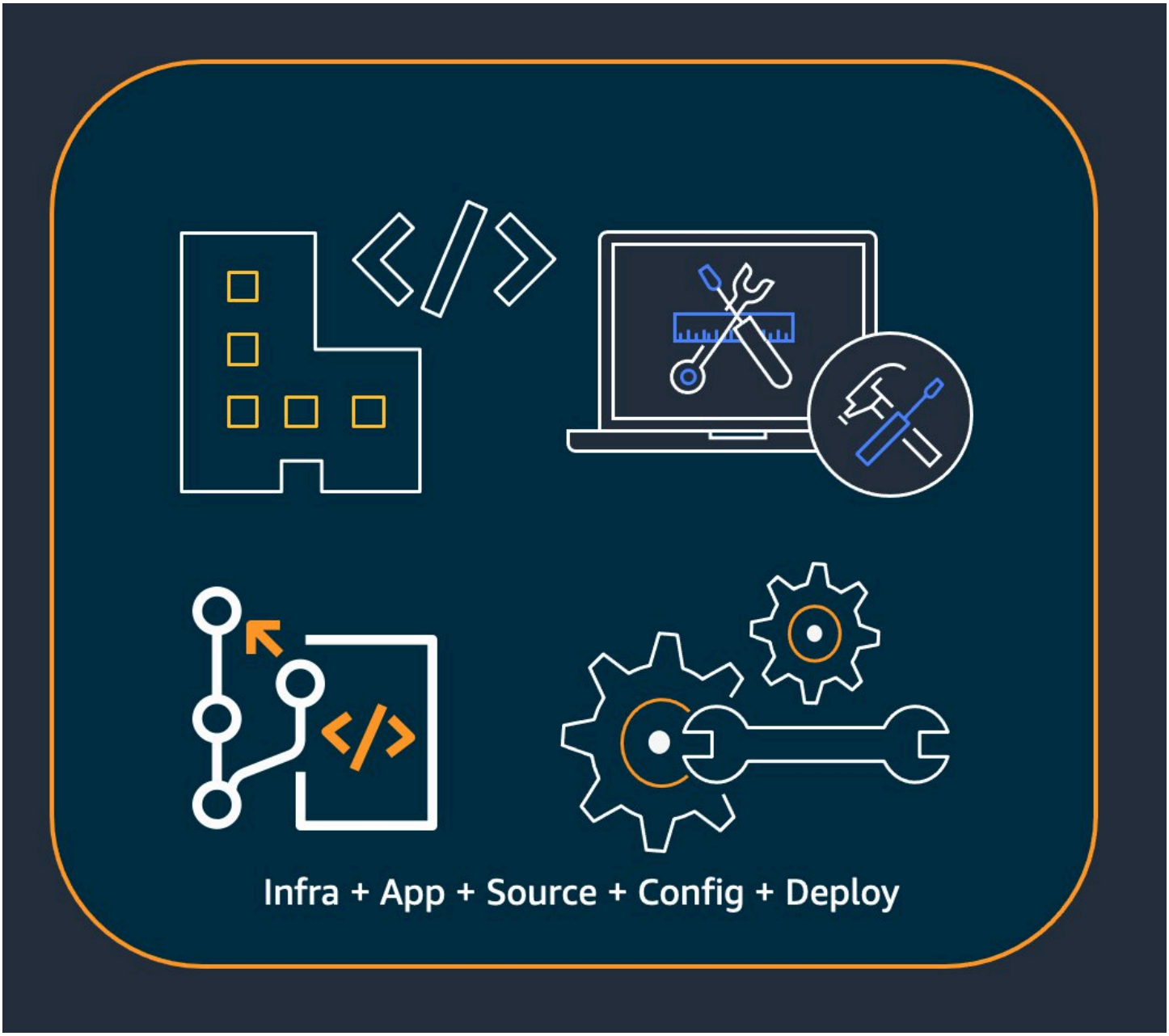
使用以下方法开发和部署云基础架构的最佳实践 AWS CDK

借助 AWS CDK，开发人员或管理员可以使用支持的编程语言来定义其云基础架构。CDK 应用程序应组织成逻辑单元，例如 API、数据库和监控资源，并且可以选择拥有用于自动部署的管道。逻辑单元应作为构造来实现，包括以下内容：

- 基础设施（例如 Amazon S3 存储桶、亚马逊 RDS 数据库或亚马逊 VPC 网络）
- 运行时代码（例如 AWS Lambda 函数）
- 配置代码

堆栈定义了这些逻辑单元的部署模型。有关 CDK 背后概念的更详细介绍，请参阅开始使用。

AWS CDK 这反映了对客户和内部团队需求的仔细考虑，也反映了对复杂云应用程序部署和持续维护期间经常出现的故障模式的仔细考虑。我们发现，故障通常与未经全面测试的应用程序的 out-of-band “” 更改有关，例如配置更改。因此，我们 AWS CDK 围绕一个模型开发了这个模型，在这个模型中，你的整个应用程序都是用代码定义的，不仅是业务逻辑，还有基础架构和配置。这样，就可以仔细审查提议的更改，在类似于生产的环境中进行不同程度的全面测试，如果出现问题，则可以完全撤消。



在部署时，会 AWS CDK 合成包含以下内容的云程序集：

- AWS CloudFormation 描述您在所有目标环境中的基础架构的模板
- 包含您的运行时代码及其支持文件的文件资产

使用 CDK，应用程序主版本控制分支中的每一次提交都可以代表应用程序的完整、一致、可部署的版本。然后，只要进行更改，就可以自动部署您的应用程序。

其背后的理念 AWS CDK 促成了我们推荐的最佳实践，我们将其分为四大类。

- the section called “组织最佳实践”
- the section called “编码最佳实践”
- the section called “构建最佳实践”
- the section called “应用程序最佳实践”

> **Tip**
>
> 在适用于 CDK 定义 AWS CloudFormation的基础架构的情况下，还要考虑您使用的各项 AWS 服务的最佳实践。

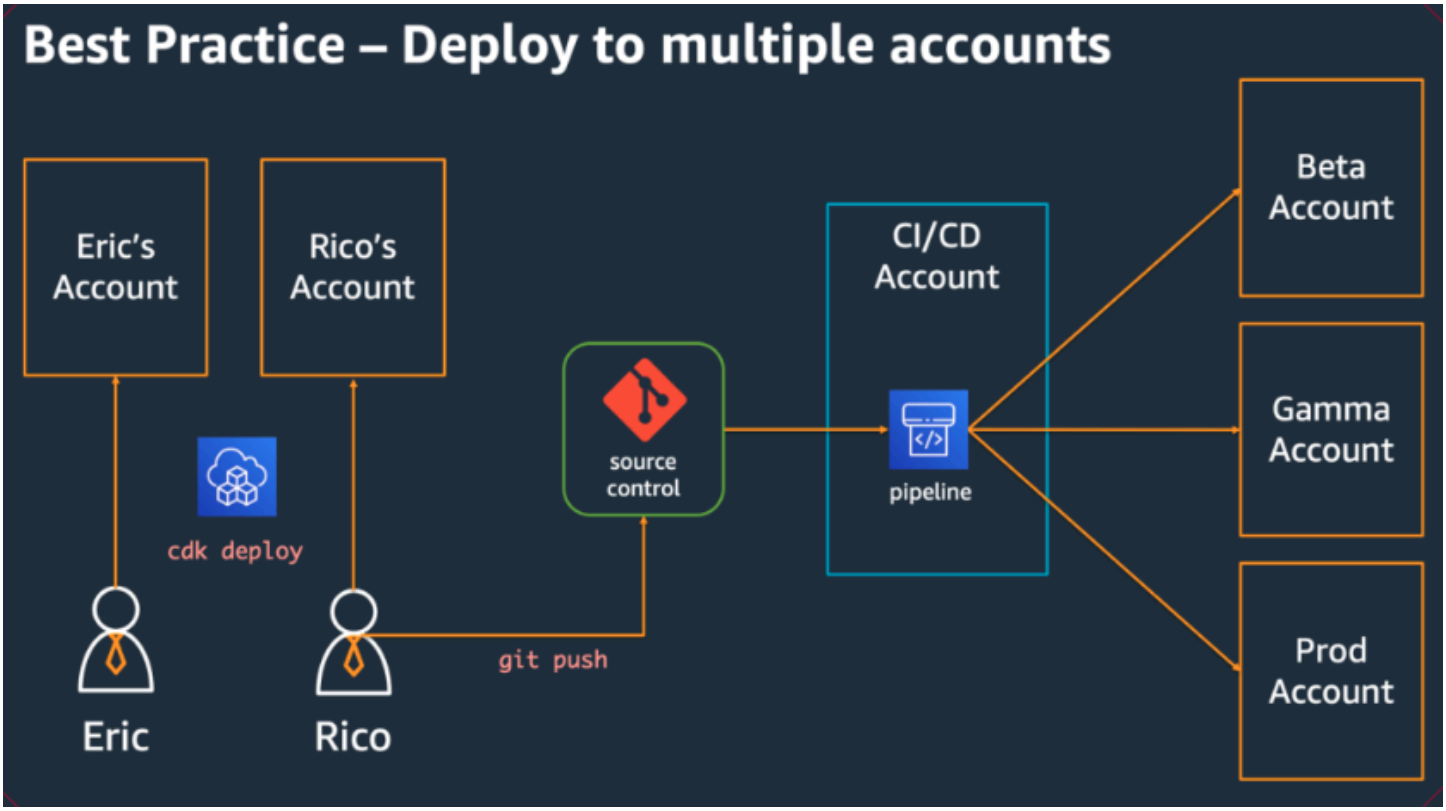
组织最佳实践

在 AWS CDK 采用的初始阶段，重要的是要考虑如何为组织做好准备，以取得成功。最好的做法是让一支专家团队负责培训和指导公司其他成员采用CDK。该团队的规模可能各不相同，从小型公司的一两个人到大型公司的成熟卓越云中心（CCoE）of Excellence）不等。该团队负责为贵公司的云基础设施制定标准和政策，并负责培训和指导开发人员。

CCoE 可能会就云基础设施应使用哪些编程语言提供指导。每个组织的细节会有所不同，但是良好的政策有助于确保开发人员能够了解和维护公司的云基础架构。

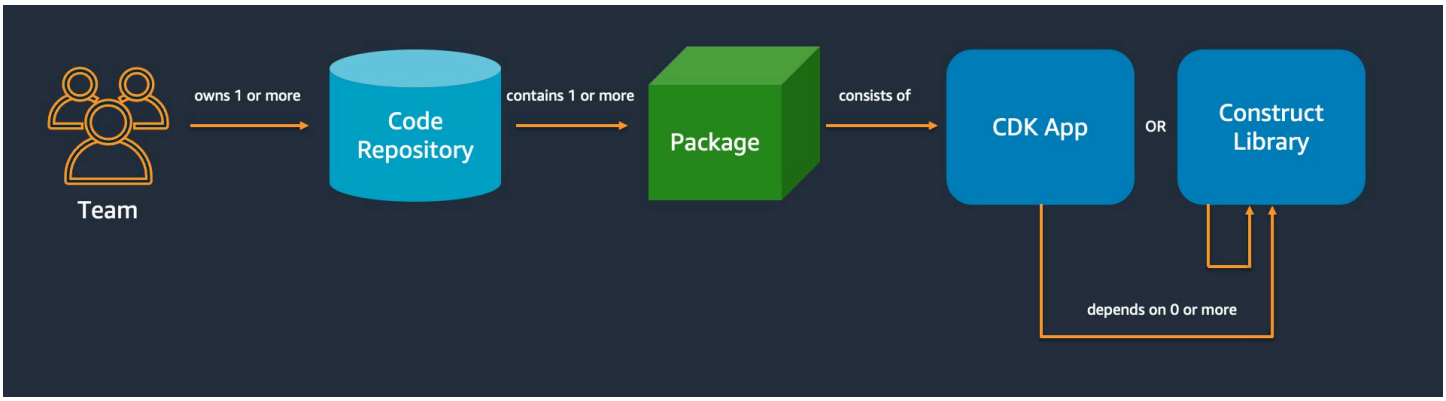
CCoE 还创建了一个 “着陆区”，用于定义您在其中的 AWS组织单位。landing zone 是一个基于最佳实践蓝图的预配置、安全、可扩展的多账户 AWS 环境。要整合构成您的 landing zone 的服务 AWS Control Tower，您可以使用，它通过单个用户界面配置和管理整个多账户系统。

开发团队应该能够使用自己的账户进行测试，并根据需要在这些账户中部署新资源。个人开发人员可以将这些资源视为他们自己的开发工作站的扩展。使用 CDK Pipelines， AWS CDK 可以通过 CI/CD 账户将应用程序部署到测试、集成和生产环境（每个环境都隔离在 AWS 自己的区域或账户中）。这是通过将开发者的代码合并到您组织的规范存储库中来完成的。



编码最佳实践

本节介绍组织 AWS CDK 代码的最佳实践。下图显示了团队与该团队的代码存储库、包、应用程序和构造库之间的关系。



从简单开始，只有在需要时才增加复杂性

我们大多数最佳实践的指导原则是让事情尽可能简单，但不要更简单。只有当您的要求要求采用更复杂的解决方案时，才会增加复杂性。借助 AWS CDK，您可以根据需要重构代码以支持新的要求。您不必预先为所有可能的场景进行架构设计。

与 Well-Architect AWS ed 框架保持一致

Wel AWS l-A rchitected Framework 将组件定义为根据需求共同交付的代码、配置 AWS 和资源。一个组件通常是技术所有权的单位，并且与其他组件分离。“工作负载” 一词用于标识一组共同带来业务价值的组件。工作量通常是业务和技术领导者沟通的详细程度。

AWS CDK 应用程序映射到由 Well-Architecte AWS d Framework 定义的组件。 AWS CDK 应用程序是一种编纂和交付 Well-Architected 云应用程序最佳实践的机制。您还可以通过构件存储库（例如）创建和共享作为可重复使用的代码库的组件 AWS CodeArtifact。

每个应用程序都从一个存储库中的单个软件包开始

单个软件包是您的 AWS CDK 应用程序的入口点。在这里，您可以定义如何以及在何处部署应用程序的不同逻辑单元。您还可以定义 CI/CD 管道来部署应用程序。应用程序的构造定义了解决方案的逻辑单元。

对于在多个应用程序中使用的构造，请使用其他包。（共享构造也应该有自己的生命周期和测试策略。）同一存储库中软件包之间的依赖关系由您的 repo 的构建工具管理。

尽管这是可能的，但我们不建议将多个应用程序放在同一个存储库中，尤其是在使用自动部署管道时。这样做会增加部署期间变更的 “爆炸半径”。当存储库中有多个应用程序时，对一个应用程序的更改会触发其他应用程序的部署（即使其他应用程序没有更改）。此外，一个应用程序的中断会阻止其他应用程序的部署。

根据代码生命周期或团队所有权将代码移入存储库

当软件包开始在多个应用程序中使用时，请将它们移动到自己的存储库中。这样，使用软件包的应用程序构建系统就可以引用这些包，也可以根据独立于应用程序生命周期的节奏对其进行更新。但是，起初将所有共享结构放在一个存储库中可能是有意义的。

此外，当不同的团队正在处理软件包时，请将它们移至自己的存储库。这有助于强制执行访问控制。

要跨存储库边界使用软件包，你需要一个私有软件包存储库，类似于 NPM 或 Maven Central PyPi，但位于组织内部。您还需要一个发布流程，用于构建、测试软件包并将其发布到私有软件包存储库。CodeArtifact可以托管大多数流行编程语言的软件包。

对包存储库中软件包的依赖由您的语言的包管理器管理，例如 TypeScript 或 JavaScript 应用程序的 NPM。你的包管理器可以帮助确保版本是可重复的。它通过记录您的应用程序所依赖的每个软件包的特定版本来实现此目的。它还允许您以受控的方式升级这些依赖关系。

共享软件包需要不同的测试策略。对于单个应用程序，将应用程序部署到测试环境并确认它仍然可以正常工作可能就足够了。但是共享软件包必须独立于消费应用程序进行测试，就好像它们是向公众发布一样。（您的组织可能会选择实际向公众发布一些共享软件包。）

请记住，构造可以是任意简单的，也可以是复杂的。A `Bucket` 是一个构造，但`CameraShopWebsite`也可以是一个构造。

基础架构和运行时代码位于同一个包中

除了生成用于部署基础设施的 AWS CloudFormation 模板外， AWS CDK 还捆绑了 Lambda 函数和 Docker 映像等运行时资产，并将它们与您的基础设施一起部署。这使得可以将定义基础架构的代码和实现运行时逻辑的代码组合到一个构造中。这是执行此操作的最佳实践。这两种代码不需要存放在单独的存储库中，甚至不需要放在单独的软件包中。

要将这两种代码一起发展，您可以使用一个独立的结构，该结构可以完整描述一项功能，包括其基础架构和逻辑。使用自包含结构，您可以隔离测试这两种代码，跨项目共享和重用代码，并同步对所有代码进行版本控制。

构建最佳实践

本节包含开发构造的最佳实践。构造是可重复使用的可组合模块，用于封装资源。它们是 AWS CDK 应用程序的基石。

使用构造建模，使用堆栈进行部署

堆栈是部署单位：堆栈中的所有内容都一起部署。因此，在使用多个 AWS 资源构建应用程序的高级逻辑单元时，请将每个逻辑单元表示为 `Construct`，而不是表示为`Stack`。仅使用堆栈来描述在各种部署场景中应如何构造和连接您的构造。

例如，如果您的逻辑单元之一是网站，则构成该网站的结构（例如 Amazon S3 存储桶、API Gateway、Lambda 函数或 Amazon RDS 表）应组合成一个高级结构。然后，应将该构造实例化为一个或多个堆栈进行部署。

通过使用构造进行构建，使用堆栈进行部署，可以提高基础架构的重复使用潜力，并使自己在部署方式上更加灵活。

使用属性和方法进行配置，而不是使用环境变量进行配置

在构造和堆栈中查找环境变量是一种常见的反模式。构造和堆栈都应接受属性对象，以便在代码中实现完全可配置性。否则会引入对运行代码的计算机的依赖，这会创建更多需要跟踪和管理的配置信息。

通常，应将环境变量查询限制在 AWS CDK 应用程序的顶层。它们还应该用于传递在开发环境中运行所需的信息。有关更多信息，请参阅 the section called “环境”。

对您的基础架构进行单元测试

要在构建时在所有环境中始终如一地运行全套单元测试，请避免在综合过程中进行网络查找，并使用代码对所有生产阶段进行建模。（这些最佳做法将在后面介绍。）如果任何一次提交总是生成相同的生成的模板，则可以信任您编写的单元测试，以确认生成的模板是否符合您的预期。有关更多信息，请参阅 测试结构。

不要更改有状态资源的逻辑 ID

更改资源的逻辑 ID 会导致该资源在下次部署时被新资源替换。对于数据库和 S3 存储桶等有状态资源，或者像 Amazon VPC 这样的持久性基础设施，这很少是你想要的。请谨慎对待任何可能导致 ID 更改的 AWS CDK 代码重构。编写断言有状态资源的逻辑 ID 保持静态的单元测试。逻辑 ID 源自`id`您在实例化构造时指定的以及构造在构造树中的位置。有关更多信息，请参阅 the section called “逻辑 ID”。

结构不足以保证合规性

许多企业客户为 L2 构造（代表具有内置合理默认值和最佳实践的单个 AWS 资源的 “精选” 结构）编写自己的包装器。这些封装程序强制执行安全最佳实践，例如静态加密和特定的 IAM 策略。例如，您可以创建一个`MyCompanyBucket`，然后在应用程序中使用它来代替通常的 Amazon S3 `Bucket` 结构。这种模式对于在软件开发生命周期的早期显示安全指导很有用，但不要依赖它作为唯一的强制执行手段。

取而代之的是，使用服务控制策略和权限边界等 AWS 功能在组织层面强制实施安全防护措施。在部署之前，使用the section called “方面”或诸如 CloudFormation Guard 之类的工具对基础架构元素的安全属性做出断言。 AWS CDK 用于它最擅长的事情。

最后，请记住，编写自己的 “L2+” 构造可能会使开发人员无法利用诸如AWS 解决方案构造之类的 AWS CDK 软件包或来自 Construct Hub 的第三方构造。这些包通常基于标准 AWS CDK 构造构造，无法使用您的包装器构造。

应用程序最佳实践

在本节中，我们将讨论如何编写 AWS CDK 应用程序，通过组合结构来定义 AWS 资源的连接方式。

在综合时做出决定

尽管 AWS CloudFormation 允许您在部署时做出决策（使用`Conditions{ Fn::If }`、和`Parameters`），并且允许您 AWS CDK 访问这些机制，但我们建议不要使用它们。与通用编程语言中可用的值相比，您可以使用的值类型和可以对其执行的操作类型是有限的。

相反，请尝试使用编程语言的语`if`句和其他功能在 AWS CDK 应用程序中做出所有决定，例如要实例化哪个构造。例如，一个常见的 CDK 成语，即遍历列表并用列表中每项的值实例化构造，根本不可能使用表达式。 AWS CloudFormation

AWS CloudFormation 将其视为 AWS CDK 用于强大云部署的实现细节，而不是语言目标。你不是在 TypeScript 用 Python 编写 AWS CloudFormation 模板，而是在编写恰好 CloudFormation 用于部署的 CDK 代码。

使用生成的资源名称，而不是物理名称

名字是一种宝贵的资源。每个名称只能使用一次。因此，如果您将表名或存储桶名称硬编码到基础设施和应用程序中，则不能在同一个账户中两次部署该基础架构。（我们这里所说的名称是由 Amazon S3 存储桶构造中的`bucketName`属性指定的名称。）

更糟糕的是，您无法对需要替换的资源进行更改。如果只能在创建资源时设置属性，例如 Amazon DynamoDB 表`KeySchema`的属性，则该属性是不可变的。更改此属性需要新的资源。但是，新资源必须具有相同的名称才能真正替换。但是，当现有资源仍在使用该名称时，它不能使用相同的名称。

更好的方法是尽可能少地指定名称。如果您省略资源名称，则 AWS CDK 会以不会导致问题的方式为您生成资源名称。假设你有一张表作为资源。然后，您可以将生成的表名作为环境变量传递到您的 AWS Lambda 函数中。在您的 AWS CDK 应用程序中，您可以将表名引用为`table.tableName`。或者，您可以在启动时在您的 Amazon EC2 实例上生成配置文件，或者将实际的表名写入 AWS Systems Manager 参数存储，以便您的应用程序可以从那里读取它。

如果你需要它的地方是另一个 AWS CDK 堆栈，那就更简单了。假设一个堆栈定义了资源，而另一个堆栈需要使用该资源，则以下情况适用：

- 如果两个堆栈位于同一个 AWS CDK 应用程序中，则在两个堆栈之间传递一个引用。例如，将对资源构造的引用保存为定义堆栈的属性 (`this.stack.uploadBucket = myBucket`)。然后，将该属性传递给需要资源的堆栈的构造函数。
- 当两个堆栈位于不同的 AWS CDK 应用程序中时，使用静态`from`方法根据其 ARN、名称或其他属性使用外部定义的资源。（例如，用`Table.fromArn()`于 DynamoDB 表）。使用`CfnOutput`构造在的输出中打印 ARN 或其他必填值`cdk deploy`，或者查看。 AWS Management Console或

者，第二个应用程序可以读取第一个应用程序生成的 CloudFormation模板并从该`Outputs`部分检索该值。

定义删除策略和日志保留

AWS CDK 试图通过默认使用保留您创建的所有内容的策略来防止数据丢失。例如，对于包含数据的资源（例如 Amazon S3 存储桶和数据库表），默认的移除策略是不在资源从堆栈中移除时将其删除。相反，资源是堆栈中的孤立资源。同样，CDK 的默认设置是永久保留所有日志。在生产环境中，这些默认设置很快就会导致存储大量您实际上并不需要的数据，并产生相应的 AWS 账单。

请仔细考虑您希望这些策略适用于每种生产资源，并相应地指定它们。the section called “方面”用于验证堆栈中的删除和日志策略。

根据部署要求将您的应用程序分成多个堆栈

对于您的应用程序需要多少堆栈，没有一成不变的规则。通常，您最终会根据自己的部署模式做出决定。请记住以下准则：

- 将尽可能多的资源放在同一个堆栈中通常更简单，因此，除非你知道要将它们分开，否则请将它们放在一起。
- 考虑将有状态资源（如数据库）与无状态资源保存在单独的堆栈中。然后，您可以在有状态堆栈上开启终止保护。这样，您就可以自由销毁或创建无状态堆栈的多个副本，而不会有数据丢失的风险。
- 有状态的资源对构造重命名更为敏感——重命名会导致资源替换。因此，不要将有状态的资源嵌套在可能被移动或重命名的构造中（除非状态丢失后可以重建，比如缓存）。这是将有状态资源放在自己的堆栈中的另一个很好的理由。

承诺`cdk.context.json`避免非确定性行为

确定性是成功 AWS CDK 部署的关键。无论何时将 AWS CDK 应用程序部署到给定环境，其结果都应基本相同。

由于您的 AWS CDK 应用程序是用通用编程语言编写的，因此它可以执行任意代码、使用任意库和进行任意网络调用。例如，在合成应用程序时，你可以使用 AWS SDK 从你的 AWS 账户中检索一些信息。要认识到，这样做会导致额外的凭据设置要求，增加延迟，并且每次运行`cdk synth`时都可能出现故障（无论多么小）。

在合成过程中，切勿修改您的 AWS 账户或资源。合成应用程序不应有副作用。只有在 AWS CloudFormation 模板生成之后，才应在部署阶段对基础架构进行更改。这样，如果出现问题， AWS

CloudFormation 可以自动回滚更改。要进行不容易在 AWS CDK 框架内进行的更改，请在部署时使用自定义资源执行任意代码。

即使是严格的只读调用也不一定是安全的。考虑一下如果网络调用返回的值发生变化会发生什么。这将影响您的基础架构的哪一部分？ 已经部署的资源会怎样？ 以下是两个示例情况，其中值的突然变化可能会导致问题。

- 如果您将 Amazon VPC 配置到指定区域的所有可用区域，并且部署当天可用区的数量为两个，则您的 IP 空间将被分成两半。如果在第二天 AWS 启动新的可用区，则之后的下一次部署会尝试将您的 IP 空间分成三分之二，要求重新创建所有子网。这可能是不可能的，因为你的 Amazon EC2 实例仍在运行，你必须手动清理它。
- 如果您查询最新的 Amazon Linux 计算机映像并部署 Amazon EC2 实例，而第二天又发布了新映像，则后续部署会选择新的 AMI 并替换您的所有实例。这可能不是你所期望发生的。

这些情况可能是有害的，因为 AWS-side 更改可能会在成功部署数月或数年后发生。突然间，你的部署“无缘无故” 失败了，你很久以前就忘记了自己做了什么以及为什么。

幸运的是， AWS CDK 包括一种名为上下文提供者的机制，用于记录非确定性值的快照。这允许 future 合成操作生成与首次部署时完全相同的模板。新模板中唯一的更改是您在代码中所做的更改。当你使用构造的`.fromLookup()`方法时，调用的结果会被缓存在中`cdk.context.json`。您应该将其与其余代码一起提交给版本控制，以确保 CDK 应用程序的未来执行使用相同的值。CDK Toolkit 包含用于管理上下文缓存的命令，因此您可以在需要时刷新特定条目。有关更多信息，请参阅 the section called “上下文”。

如果您需要一些没有原生 CDK 上下文提供程序的值（来自 AWS 或其他地方），我们建议您编写一个单独的脚本。该脚本应检索该值并将其写入文件，然后在您的 CDK 应用程序中读取该文件。仅当你想要刷新存储的值时才运行脚本，而不是作为常规生成过程的一部分。

让他们 AWS CDK 管理角色和安全组

借助 AWS CDK 构造库的`grant()`便捷方法，您可以创建 AWS Identity and Access Management 角色，使用最小范围的权限向另一个资源授予访问权限。例如，考虑如下所示的一行：

```
myBucket.grantRead(myLambda)
```

这一行向 Lambda 函数的角色添加了一个策略（该角色也是为您创建的）。这个角色及其政策有十几 CloudFormation 行你不必写。仅 AWS CDK 授予函数从存储桶读取所需的最低权限。

如果您要求开发人员始终使用安全团队创建的预定义角色，那么 AWS CDK 编码就会变得更加复杂。您的团队在设计应用程序时可能会失去很大的灵活性。更好的选择是使用服务控制策略和权限界限来确保开发人员保持在护栏之内。

用代码对所有生产阶段进行建模

在传统 AWS CloudFormation 场景中，您的目标是生成一个参数化的工件，以便在应用特定于这些环境的配置值后，可以将其部署到各种目标环境。在 CDK 中，你可以而且应该将该配置构建到你的源代码中。为您的生产环境创建一个堆栈，并为其他每个阶段创建一个单独的堆栈。然后，在代码中输入每个堆栈的配置值。使用诸如 S ecrets M an ager 和 System s Manager Parameter Store 之类的服务，使用这些资源的名称或 ARN 来获取你不想签入源代码管理的敏感值。

合成应用程序时，在`cdk.out`文件夹中创建的云程序集包含每个环境的单独模板。你的整个版本都是确定性的。您的应用程序没有任何 out-of-band更改，并且任何给定的提交始终会生成完全相同的 AWS CloudFormation 模板和随附的资产。这使得单元测试更加可靠。

测量一切

要在没有人为干预的情况下实现全面持续部署的目标，需要高度的自动化。只有通过大量的监控，才能实现自动化。要衡量已部署资源的各个方面，请创建指标、警报和仪表板。不要停下来衡量 CPU 使用率和磁盘空间之类的东西。还要记录您的业务指标，并使用这些衡量标准来自动执行部署决策，例如回滚。中的大多数 L2 结构 AWS CDK 都有方便的方法来帮助您创建指标，例如 Dynam odb.Table 类上的`metricUserErrors()`方法。

AWS CDK 参考

本节包含 AWS Cloud Development Kit (AWS CDK)的参考信息。

主题

- API 参考
- AWS CDK 版本控制

API 参考

API 参考包含有关 AWS 构造库和提供的其他 API 的信息 AWS Cloud Development Kit (AWS CDK)。AWS 构造库的大部分内容都包含在一个TypeScript名为:的包中`aws-cdk-lib`。实际的软件包名称因语言而异。为每种支持的编程语言提供了单独的 API 参考版本。

CDK API 参考被组织成子模块。每个 AWS 服务子模块都有一个或多个子模块。

每个子模块都有一个概述，其中包括有关如何使用其 API 的信息。例如，S3 概述演示了如何在亚马逊简单存储服务 (Amazon S3) 存储桶上设置默认加密。

AWS CDK 版本控制

本主题提供有关如何 AWS Cloud Development Kit (AWS CDK) 处理版本控制的参考信息。

版本号由三个数字版本部分组成：主版本。 未成年人。 补丁，并严格遵守语义版本控制模型。这意味着对稳定 API 的重大更改仅限于主要版本。

次要版本和补丁版本向后兼容。在具有相同主版本的先前版本中编写的代码可以升级到相同主版本中的新版本。它还将继续构建和运行，产生相同的输出。

主题

- AWS CDKCLI兼容性
- AWS 构造库版本控制
- 语言绑定稳定性

AWS CDKCLI兼容性

始终与语义上较低或相等版本号的构造库兼容。 AWS CDK CLI因此，在同一个主要版本 AWS CDK CLI中进行升级始终是安全的。

并不 AWS CDK CLI总是与语义上更高版本的构造库兼容。兼容性取决于两个组件是否采用相同的云装配架构版本。该 AWS CDK 框架在合成过程中生成云程序集， AWS CDK CLI然后将其用于部署。定义云组件格式的架构经过严格指定和版本控制。

AWS 使用给定云装配架构版本的构造库与使用该架构 AWS CDK CLI版本或更高版本的版本兼容。这可能包括早于给定构造库版本的版本。 AWS CDK CLI

当构造库所需的云装配版本与支持的版本不兼容时 AWS CDK CLI，您会收到如下错误消息：

```
Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but
 found 4.0.0.
    Please upgrade your CLI in order to interact with this app.
```

要解决此错误，请将更新 AWS CDK CLI到与所需云程序集版本兼容的版本或最新的可用版本。通常不建议使用替代方案（降级应用程序使用的构造库模块）。

> **Note**
>
> 有关云程序集架构的更多详细信息，请参阅云程序集版本控制。

AWS 构造库版本控制

C AWS onstruct Library 中的模块在从概念发展到成熟的 API 时会经历不同的阶段。在的后续版本中，不同的阶段提供不同程度的 API 稳定性 AWS CDK。

主 AWS CDK 库中的 API 是稳定的`aws-cdk-lib`，并且该库具有完全的语义版本。该软件包包括所有 AWS 服务的 AWS CloudFormation (L1) 构造和所有稳定的更高级别 (L2 和 L3) 模块。（它还包括核心 CDK 类，例如`App`和`Stack`）。在 CDK 的下一个主要版本发布之前，API 不会从该软件包中删除（尽管它们可能会被弃用）。任何单独的 API 都不会有重大变化。当需要进行重大更改时，将添加一个全新的 API。

对于已经包含在中的服务，`aws-cdk-lib`正在开发的新 API 使用`Beta`*N*后缀进行标识，后缀从 1 N 开始，随着新 API 的每一次重大更改而递增。 `Beta`*N*API 永远不会被删除，只会被弃用，因此您的现有

应用程序可以继续使用较新版本的。aws-cdk-lib当 API 被认为稳定时，会添加一个不带Beta*N*后缀的新 API。

当开始为以前只有 L1 API 的 AWS 服务开发更高级别（L2 或 L3）的 API 时，这些 API 最初是在单独的包中分发的。此类软件包的名称带有 “Alpha” 后缀，其版本与与之兼容的第一个版本相匹配aws-cdk-lib，即alpha子版本。当模块支持预期用例时，会将其 API 添加到aws-cdk-lib。

语言绑定稳定性

随着时间的推移，我们可能会增加 AWS CDK 对其他编程语言的支持。尽管所有语言中描述的 API 都相同，但 API 的表达方式因语言而异，并且可能会随着语言支持的发展而改变。出于这个原因，语言绑定在一段时间内被视为实验性的，直到它们被认为可以投入生产使用。

Language	Stability
TypeScript	Stable
JavaScript	Stable
Python	Stable
Java	Stable
C#/.NET	Stable
Go	Stable

AWS CDK 教程

本节包含的教程 AWS Cloud Development Kit (AWS CDK)。

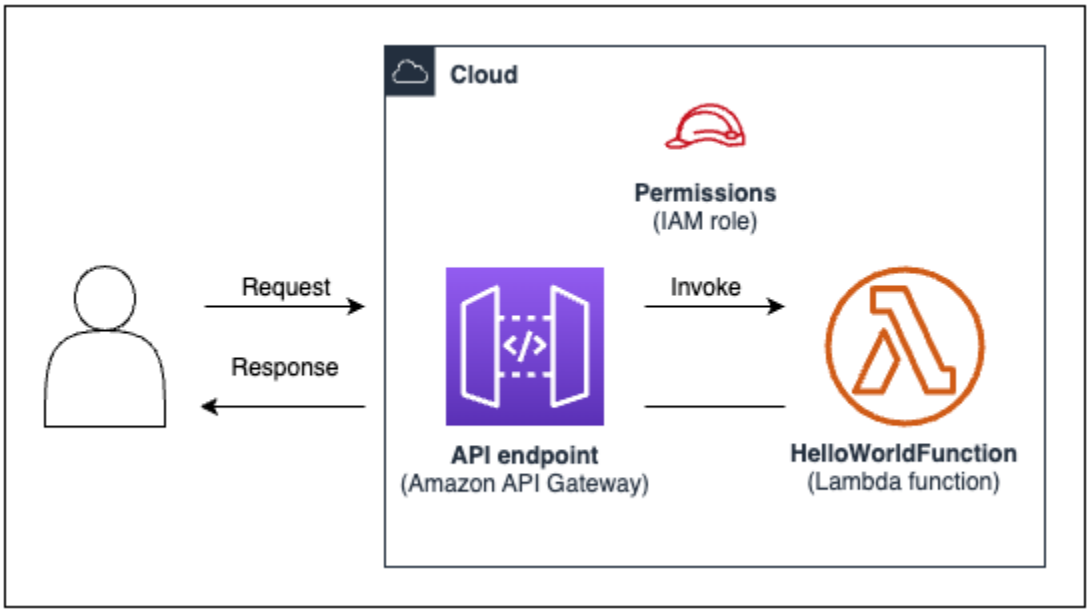
主题

创建无服务器 Hello World 应用程序

在本教程中，您将使用创建一个简单的无服务器Hello World应用程序，该应用程序通过创建以下内容来实现基本的 API 后端： AWS Cloud Development Kit (AWS CDK)

- Amazon API Gateway REST API — 提供一个 HTTP 终端节点，用于通过HTTP GET请求调用您的函数。
- AWS Lambda f unction — 使用HTTP端点调用时返回`Hello World!`消息的函数。
- 集成和权限-您的资源相互交互和执行操作（例如向 Amazon CloudWatch 写入日志）的配置详细信息和权限。

下图显示此应用程序的组件：



在本教程中，您将完成以下内容：

1. 创建 AWS CDK 项目。
2. 使用构造库中的 L2 构造定义 Lambda 函数和 API Gateway REST API。 AWS
3. 将您的应用程序部署到 AWS Cloud。
4. 在中与您的应用程序交互 AWS Cloud。
5. 从 AWS Cloud中删除示例应用程序。

先决条件

在开始本教程之前，请完成以下操作：

- 创建 AWS 账户 并安装并配置 AWS Command Line Interface (AWS CLI)。
- 安装Node.js和npm。
- 使用`npm install -g aws-cdk`在全球范围内安装 CDK 工具包。

有关更多信息，请参阅 开始使用 AWS CDK。

我们还建议您对以下内容有基本的了解：

- 那是什么 AWS CDK？以获取对 AWS CDK. 的基本介绍
- AWS CDK 概念有关核心概念的概述 AWS CDK。

步骤 1：创建 CDK 项目

在此步骤中，您将使用 AWS CDK CLI`cdk init`命令创建新的 CDK 项目。

创建 CDK 项目

1. 从您选择的起始目录中，创建并导航到计算机`cdk-hello-world`上名为的项目目录：

   ```
   $ mkdir cdk-hello-world && cd cdk-hello-world
   ```

2. 使用`cdk init`命令以您的首选编程语言创建新项目：

   TypeScript

   ```
   $ cdk init --language typescript
   ```

安装 AWS CDK 库：

```
$ npm install aws-cdk-lib constructs
```

JavaScript

```
$ cdk init --language javascript
```

安装 AWS CDK 库：

```
$ npm install aws-cdk-lib constructs
```

Python

```
$ cdk init --language python
```

激活虚拟环境：

```
$ source .venv/bin/activate
```

安装 AWS CDK 库和项目依赖项：

```
(.venv)$ python3 -m pip install -r requirements.txt
```

Java

```
$ cdk init --language java
```

安装 AWS CDK 库和项目依赖项：

```
$ mvn package
```

C#

```
$ cdk init --language csharp
```

安装 AWS CDK 库和项目依赖项：

```
$ dotnet restore src
```

Go

```
$ cdk init --language go
```

安装项目依赖关系：

```
$ go mod tidy
```

CDK CLI 创建了一个具有以下结构的项目：

TypeScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.ts
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### cdk-hello-world.test.ts
### tsconfig.json
```

JavaScript

```
cdk-hello-world
### .git
### .gitignore
```

```
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.js
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### cdk-hello-world.test.js
```

Python

```
cdk-hello-world
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### cdk_hello_world
#   ### __init__.py
#   ### cdk_hello_world_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
```

Java

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
#   ### main
#   #   ### java
#   #       ### com
```

```
#   #           ### myorg
#   #               ### CdkHelloWorldApp.java
#   #               ### CdkHelloWorldStack.java
### target
```

C#

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### CdkHelloWorld
    #   ### CdkHelloWorld.csproj
    #   ### CdkHelloWorldStack.cs
    #   ### GlobalSuppressions.cs
    #   ### Program.cs
    ### CdkHelloWorld.sln
```

Go

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk-hello-world.go
### cdk-hello-world_test.go
### cdk.json
### go.mod
```

CDK CLI 会自动创建包含单个堆栈的 CDK 应用程序。CDK 应用程序实例是根据App类创建的。以下是 CDK 应用程序文件的一部分：

TypeScript

位于`bin/cdk-hello-world.ts`：

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
```

```
import { CdkHelloWorldStack } from '../lib/cdk-hello-world-stack';

const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

JavaScript

位于`bin/cdk-hello-world.js`：

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { CdkHelloWorldStack } = require('../lib/cdk-hello-world-stack');
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

Python

位于`app.py`：

```
#!/usr/bin/env python3
import os
import aws_cdk as cdk
from cdk_hello_world.cdk_hello_world_stack import CdkHelloWorldStack

app = cdk.App()
CdkHelloWorldStack(app, "CdkHelloWorldStack",)
app.synth()
```

Java

位于`src/main/java/.../CdkHelloWorldApp.java`：

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;
```

```
public class JavaApp {
    public static void main(final String[] args) {
        App app = new App();

        new JavaStack(app, "JavaStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

位于`src/CdkHelloWorld/Program.cs`：

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CdkHelloWorld
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new CdkHelloWorldStack(app, "CdkHelloWorldStack", new StackProps
            {

            });
            app.Synth();
        }
    }
}
```

Go

位于`cdk-hello-world.go`：

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
```

```
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
    defer jsii.Close()
    app := awscdk.NewApp(nil)
    NewCdkHelloWorldStack(app, "CdkHelloWorldStack", &CdkHelloWorldStackProps{
        awscdk.StackProps{
            Env: env(),
        },
    })
    app.Synth(nil)
}

func env() *awscdk.Environment {
    return nil
}
```

第 2 步：创建您的 Lambda 函数

在您的 CDK 项目中，创建一个包含新`hello.js`文件的`lambda`目录。以下是 示例：

TypeScript

在项目的根目录下，运行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

JavaScript

在项目的根目录下，运行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Python

在项目的根目录下，运行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Java

在项目的根目录下，运行以下命令：

```
$ mkdir -p src/main/resources/lambda
$ cd src/main/resources/lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### src
    ### main
        ###resources
            ###lambda
                ###hello.js
```

C#

在项目的根目录下，运行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

Go

在项目的根目录下，运行以下命令：

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

现在应将以下内容添加到您的 CDK 项目中：

```
cdk-hello-world
### lambda
    ### hello.js
```

> **Note**
>
> 为了简化本教程，我们使用了适用于所有 CDK 编程语言的 JavaScript Lambda 函数。

通过在新创建的文件中添加以下内容来定义您的 Lambda 函数：

```
exports.handler = async (event) => {
    return {
        statusCode: 200,
        headers: { "Content-Type": "text/plain" },
        body: JSON.stringify({ message: "Hello, World!" }),
    };
```

```
};
```

第 3 步：定义您的构造

在此步骤中，您将使用 AWS CDK L2 结构来定义 Lambda 和 API Gateway 资源。

打开定义 CDK 堆栈的项目文件。您将修改此文件来定义您的构造。以下是您的起始堆栈文件示例：

TypeScript

位于`lib/cdk-hello-world-stack.ts`：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Your constructs will go here

  }
}
```

JavaScript

位于`lib/cdk-hello-world-stack.js`：

```
const { Stack, Duration } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigateway = require('aws-cdk-lib/aws-apigateway');

class CdkHelloWorldStack extends Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    // Your constructs will go here

  }
}

module.exports = { CdkHelloWorldStack }
```

Python

位于cdk_hello_world/cdk_hello_world_stack.py：

```
from aws_cdk import Stack
from constructs import Construct

class CdkHelloWorldStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

          // Your constructs will go here
```

Java

位于src/main/java/.../CdkHelloWorldStack.java：

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Your constructs will go here
    }
}
```

C#

位于src/CdkHelloWorld/CdkHelloWorldStack.cs：

```
using Amazon.CDK;
using Constructs;
```

```
namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Your constructs will go here
        }
    }
}
```

Go

位于cdk-hello-world.go：

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)
type CdkHelloWorldStackProps struct {
    awscdk.StackProps
}
func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)
    // Your constructs will go here
    return stack
}
func main() {
    // ...
}

func env() *awscdk.Environment {
    return nil
}
```

在此文件中，AWS CDK 正在执行以下操作：

- 您的 CDK 堆栈实例是从类中实例化的。Stack
- Constructs基类是作为堆栈实例的作用域或父类导入并提供的。

定义您的 Lambda 函数资源

要定义您的 Lambda 函数资源，请导入并使用构造aws-lambda库中的 L2 构造。 AWS

按如下方式修改您的堆栈文件：

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// Import Lambda L2 construct
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
// Import Lambda L2 construct
const lambda = require('aws-cdk-lib/aws-lambda');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);
```

```
    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}

module.exports = { CdkHelloWorldStack }
```

Python

```
from aws_cdk import (
    Stack,
    # Import Lambda L2 construct
    aws_lambda as _lambda,
)
# ...

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define the Lambda function resource
        hello_world_function = _lambda.Function(
            self,
            "HelloWorldFunction",
            runtime = _lambda.Runtime.NODEJS_20_X, # Choose any supported Node.js
 runtime
            code = _lambda.Code.from_asset("lambda"), # Points to the lambda
 directory
            handler = "hello.handler", # Points to the 'hello' file in the lambda
 directory
        )
```

> **Note**
>
> 我们以内置标识_lambda符lambda的形式导入aws_lambda模块。Python

Java

```
// ...
// Import Lambda L2 construct
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Define the Lambda function resource
        Function helloWorldFunction = Function.Builder.create(this,
 "HelloWorldFunction")
                .runtime(Runtime.NODEJS_20_X)  // Choose any supported Node.js
 runtime
                .code(Code.fromAsset("src/main/resources/lambda")) // Points to the
 lambda directory
                .handler("hello.handler")  // Points to the 'hello' file in the
 lambda directory
                .build();
    }
}
```

C#

```
// ...
// Import Lambda L2 construct
using Amazon.CDK.AWS.Lambda;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
```

```
            // Define the Lambda function resource
            var helloWorldFunction = new Function(this, "HelloWorldFunction", new
 FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X, // Choose any supported Node.js
 runtime
                Code = Code.FromAsset("lambda"), // Points to the lambda directory
                Handler = "hello.handler" // Points to the 'hello' file in the
 lambda directory
            });
        }
    }
}
```

Go

```
package main

import (
    // ...
    // Import Lambda L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    helloWorldFunction := awslambda.NewFunction(stack,
 jsii.String("HelloWorldFunction"), &awslambda.FunctionProps{
        Runtime: awslambda.Runtime_NODEJS_20_X(), // Choose any supported Node.js
 runtime
        Code:    awslambda.Code_FromAsset(jsii.String("lambda")), // Points to the
 lambda directory
```

```
        Handler: jsii.String("hello"), // Points to the 'hello' file in the lambda
 directory
    })

    return stack
}

// ...
```

在这里，您可以创建 Lambda 函数资源并定义以下属性：

- `runtime`— 函数运行的环境。在这里，我们使用Node.js版本20.x。
- `code`— 本地计算机上函数代码的路径。
- `handler`— 包含您的函数代码的特定文件的名称。

定义你的 API Gateway REST API 资源

要定义您的 API Gateway REST API 资源，请导入并使用构造库中的 [aws-apigateway](#) L2 AWS 构造。

按如下方式修改您的堆栈文件：

TypeScript

```
// ...
//Import API Gateway L2 construct
import * as apigateway from 'aws-cdk-lib/aws-apigateway';


export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });
```

```
    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  }
}
```

JavaScript

```
// ...
// Import API Gateway L2 construct
const apigateway = require('aws-cdk-lib/aws-apigateway');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });

    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  };
};

// ...
```

Python

```
from aws_cdk import (
    # ...
    # Import API Gateway L2 construct
    aws_apigateway as apigateway,
)
from constructs import Construct

class CdkHelloWorldStack(Stack):
```

```
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # ...

        # Define the API Gateway resource
        api = apigateway.LambdaRestApi(
            self,
            "HelloWorldApi",
            handler = hello_world_function,
            proxy = False,
        )

        # Define the '/hello' resource with a GET method
        hello_resource = api.root.add_resource("hello")
        hello_resource.add_method("GET")
```

Java

```
// ...
// Import API Gateway L2 construct
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.apigateway.Resource;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // ...

        // Define the API Gateway resource
        LambdaRestApi api = LambdaRestApi.Builder.create(this, "HelloWorldApi")
                .handler(helloWorldFunction)
                .proxy(false) // Turn off default proxy integration
                .build();

        // Define the '/hello' resource and its GET method
```

```
        Resource helloResource = api.getRoot().addResource("hello");
        helloResource.addMethod("GET");
    }
}
```

C#

```
// ...
// Import API Gateway L2 construct
using Amazon.CDK.AWS.APIGateway;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
           // ...

            // Define the API Gateway resource
            var api = new LambdaRestApi(this, "HelloWorldApi", new
 LambdaRestApiProps
            {
                Handler = helloWorldFunction,
                Proxy = false
            });

            // Add a '/hello' resource with a GET method
            var helloResource = api.Root.AddResource("hello");
            helloResource.AddMethod("GET");
        }
    }
}
```

Go

```
// ...

import (
    // ...
    // Import Api Gateway L2 construct
```

```
    "github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    // ...

    // Define the API Gatweay resource
    api := awsapigateway.NewLambdaRestApi(stack, jsii.String("HelloWorldApi"),
 &awsapigateway.LambdaRestApiProps{
        Handler: helloWorldFunction,
        Proxy: jsii.Bool(false),
    })

    // Add a '/hello' resource with a GET method
    helloResource := api.Root().AddResource(jsii.String("hello"))
    helloResource.AddMethod(jsii.String("GET"))

    return stack
}

// ...
```

在这里，您将创建 API Gateway REST API 资源以及以下内容：

- 与您的 REST API Lambda 函数之间的集成，允许 API 调用您的函数。这包括创建 Lambda 权限资源。
- 添加`hello`到 API 端点根目录的新资源或名为的路径。这将创建一个新的终端节点，为您的基础/`hello`增光添彩URL。
- `hello`资源的 GET 方法。向/`hello`终端节点发送 GET 请求时，将调用 Lambda 函数并返回其响应。

步骤 4：为部署做好应用程序准备

在此步骤中，必要时使用 AWS CDK CLIcdk synth命令进行构建并执行基本验证，从而为部署应用程序做好准备。

如有必要，请构建您的应用程序：

TypeScript

在项目的根目录下，运行以下命令：

```
$ npm run build
```

JavaScript

不需要建筑。

Python

不需要建筑。

Java

在项目的根目录下，运行以下命令：

```
$ mvn package
```

C#

在项目的根目录下，运行以下命令：

```
$ dotnet build src
```

Go

在项目的根目录下，运行以下命令：

```
$ go build
```

运行cdk synth以根据你的 CDK 代码合成一个 AWS CloudFormation 模板。通过使用 L2 构造，便于您的 Lambda 函数之间的交互所需的许多配置详细信息REST API都是由 AWS CloudFormation 为您预置的。 AWS CDK

在项目的根目录下，运行以下命令：

```
$ cdk synth
```

> **Note**
>
> 如果您收到类似以下的错误，请确认您在cdk-hello-world目录中并重试：
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

如果成功， AWS CDK CLI将在命令提示符下以YAML格式输出 AWS CloudFormation 模板。JSON格式化的模板也保存在cdk.out目录中。

以下是 AWS CloudFormation 模板的输出示例：

AWS CloudFormation 模板

```
Resources:
  HelloWorldFunctionServiceRoleunique-identifier:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/ServiceRole/Resource
  HelloWorldFunctionunique-identifier:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
```

```
          Fn::Sub: cdk-unique-identifier-assets-${AWS::AccountId}-${AWS::Region}
        S3Key: unique-identifier.zip
      Handler: hello.handler
      Role:
        Fn::GetAtt:
          - HelloWorldFunctionServiceRoleunique-identifier
          - Arn
      Runtime: nodejs20.x
    DependsOn:
      - HelloWorldFunctionServiceRoleunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/Resource
      aws:asset:path: asset.unique-identifier
      aws:asset:is-bundled: false
      aws:asset:property: Code
  HelloWorldApiunique-identifier:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: HelloWorldApi
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Resource
  HelloWorldApiDeploymentunique-identifier:
    Type: AWS::ApiGateway::Deployment
    Properties:
      Description: Automatically created by the RestApi construct
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    DependsOn:
      - HelloWorldApihelloGETunique-identifier
      - HelloWorldApihellounique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Deployment/Resource
  HelloWorldApiDeploymentStageprod012345ABC:
    Type: AWS::ApiGateway::Stage
    Properties:
      DeploymentId:
        Ref: HelloWorldApiDeploymentunique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
      StageName: prod
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/DeploymentStage.prod/Resource
  HelloWorldApihellounique-identifier:
    Type: AWS::ApiGateway::Resource
```

```
    Properties:
      ParentId:
        Fn::GetAtt:
          - HelloWorldApiunique-identifier
          - RootResourceId
      PathPart: hello
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/Resource
  HelloWorldApihelloGETApiPermissionCdkHelloWorldStackHelloWorldApiunique-identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApi9E278160
            - /
            - Ref: HelloWorldApiDeploymentStageprodunique-identifier
            - /GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETApiPermissionTestCdkHelloWorldStackHelloWorldApiunique-
identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
```

```
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApiunique-identifier
            - /test-invoke-stage/GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.Test.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETunique-identifier:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS_PROXY
        Uri:
          Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - ":apigateway:"
              - Ref: AWS::Region
              - :lambda:path/2015-03-31/functions/
              - Fn::GetAtt:
                  - HelloWorldFunctionunique-identifier
                  - Arn
              - /invocations
      ResourceId:
        Ref: HelloWorldApihellounique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/Resource
  CDKMetadata:
```

```
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:unique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Outputs:
  HelloWorldApiEndpointunique-identifier:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: HelloWorldApiunique-identifier
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
          - Ref: HelloWorldApiDeploymentStageprodunique-identifier
          - /
Conditions:
  CDKMetadataAvailable:
    Fn::Or:
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - af-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-1
          - Fn::Equals:
              - Ref: AWS::Region
```

```
              - ap-southeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ca-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-northwest-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-2
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-3
          - Fn::Equals:
              - Ref: AWS::Region
              - il-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - sa-east-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
```

```
              - us-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-2
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-2
Parameters:
  BootstrapVersion:
    Type: AWS::SSM::Parameter::Value<String>
    Default: /cdk-bootstrap/hnb659fds/version
    Description: Version of the CDK Bootstrap resources in this environment,
 automatically retrieved from SSM Parameter Store. [cdk:skip]
Rules:
  CheckBootstrapVersion:
    Assertions:
      - Assert:
          Fn::Not:
            - Fn::Contains:
                - - "1"
                  - "2"
                  - "3"
                  - "4"
                  - "5"
                - Ref: BootstrapVersion
        AssertDescription: CDK bootstrap stack version 6 required. Please run 'cdk
 bootstrap' with a recent version of the CDK CLI.
```

通过使用 L2 构造，您可以定义一些属性来配置资源，并使用辅助方法将它们集成在一起。 AWS CDK 配置配置应用程序所需的大部分 AWS CloudFormation 资源和属性。

步骤 5：部署应用程序

在此步骤中，您将使用 AWS CDK CLIcdk deploy命令部署应用程序。 AWS CDK 与该 AWS CloudFormation 服务配合使用以配置您的资源。

> **Important**
>
> 在部署之前，您必须对 AWS 环境进行一次性引导。有关说明，请参阅引导您的环境。

在项目的根目录下，运行以下命令。如果出现提示，请确认更改：

```
$ cdk deploy

#  Synthesis time: 2.44s

...

Do you wish to deploy these changes (y/n)? y
```

部署完成后， AWS CDK CLI将输出您的终端节点 URL。复制此 URL 以进行下一步操作。以下是 示例：

```
...
#  HelloWorldStack

#  Deployment time: 45.37s

Outputs:
HelloWorldStack.HelloWorldApiEndpointunique-identifier = https://<api-id>.execute-
api.<region>.amazonaws.com/prod/
Stack ARN:
arn:aws:cloudformation:region:account-id:stack/HelloWorldStack/unique-identifier
...
```

第 6 步：与您的应用程序交互

在此步骤中，您将向 API 终端节点发起 GET 请求并收到您的 Lambda 函数响应。

找到上一步中的终端节点 URL 并添加/hello路径。然后，使用浏览器或命令提示符向您的终端节点发送 GET 请求。以下是 示例：

```
$ curl https://<api-id>.execute-api.<region>.amazonaws.com/prod/hello
{"message":"Hello World!"}%
```

恭喜，您已使用! 成功创建、部署应用程序并与之 AWS CDK交互

第 7 步：删除您的应用程序

在此步骤中，您将使用从中删除您的应用程序 AWS Cloud。 AWS CDK CLI

要删除您的应用程序，请运行cdk destroy。出现提示时，确认您删除应用程序的请求：

```
$ cdk destroy
Are you sure you want to delete: CdkHelloWorldStack (y/n)? y
CdkHelloWorldStack: destroying... [1/1]
...
 #  CdkHelloWorldStack: destroyed
```

故障排除

错误：{“消息”：“内部服务器错误”}%

在调用已部署的 Lambda 函数时，您会收到此错误。出现此错误的原因可能有多种。

进一步排除故障

使用 AWS CLI 来调用您的 Lambda 函数。

1. 修改您的堆栈文件以捕获已部署的 Lambda 函数名称的输出值。以下是 示例：

```
...

class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    new CfnOutput(this, 'HelloWorldFunctionName', {
      value: helloWorldFunction.functionName,
      description: 'JavaScript Lambda function'
    });

    // Define the API Gateway resource
    // ...
```

2. 再次部署您的应用程序。 AWS CDK CLI将输出您部署的 Lambda 函数名称的值：

```
$ cdk deploy

#  Synthesis time: 0.29s
```

```
...
 #  CdkHelloWorldStack

#  Deployment time: 20.36s

Outputs:
...
CdkHelloWorldStack.HelloWorldFunctionName = CdkHelloWorldStack-
HelloWorldFunctionunique-identifier
...
```

3. 使用在中调 AWS CLI 用您的 Lambda 函数 AWS Cloud 并将响应输出到文本文件：

```
$ aws lambda invoke --function-name CdkHelloWorldStack-HelloWorldFunctionunique-
identifier output.txt
```

4. output.txt查看您的结果。

可能的原因：您的堆栈文件中对 API Gateway 资源的定义不正确。

如果output.txt显示成功的 Lambda 函数响应，则问题可能出在您如何定义 API Gateway REST API 上。直接 AWS CLI 调用您的 Lambda，而不是通过您的终端节点。检查您的代码以确保它与本教程相匹配。然后，再次部署。

可能的原因：您的堆栈文件中对 Lambda 资源的定义不正确。

如果output.txt返回错误，则问题可能与您定义 Lambda 函数的方式有关。检查您的代码以确保它与本教程相匹配。然后再次部署。

创建包含多个堆栈的应用程序

您可以创建包含多个堆栈的 AWS Cloud Development Kit (AWS CDK) 应用程序。部署 AWS CDK 应用程序时，每个堆栈都会变成自己的 AWS CloudFormation 模板。您也可以使用 AWS CDK CLIcdk deploy命令单独合成和部署每个堆栈。

本教程涵盖以下内容：

- 如何扩展Stack类以接受新的属性或参数。
- 如何使用属性来确定堆栈包含哪些资源及其配置。
- 如何从这个类中实例化多个堆栈。

本主题中的示例使用了一个名为 `encryptBucket` (Python:`encrypt_bucket`) 的布尔属性。它指示是否应对 Amazon S3 存储桶进行加密。如果是，堆栈将使用由 AWS Key Management Service (AWS KMS) 管理的密钥启用加密。该应用程序创建了该堆栈的两个实例，一个带有加密，一个没有加密。

主题

开始前的准备工作

首先，如果你还没有安装 Node.js 和 AWS CDK 命令行工具。有关详细信息，请参阅 开始使用 AWS CDK。

接下来，通过在命令行输入以下命令来创建 AWS CDK 项目。

TypeScript

```
mkdir multistack
cd multistack
cdk init --language=typescript
```

JavaScript

```
mkdir multistack
cd multistack
cdk init --language=javascript
```

Python

```
mkdir multistack
cd multistack
cdk init --language=python
source .venv/bin/activate
pip install -r requirements.txt
```

Java

```
mkdir multistack
cd multistack
cdk init --language=java
```

你可以将生成的 Maven 项目导入到你的 Java IDE 中。

C#

```
mkdir multistack
cd multistack
cdk init --language=csharp
```

你可以在 Visual Studio `src/Pipeline.sln` 中打开该文件。

添加可选参数

`Stack`构造函数的`props`参数满足了接口`StackProps`。在此示例中，我们希望堆栈接受一个额外的属性来告诉我们是否要加密 Amazon S3 存储桶。我们应该创建一个包含该属性的接口或类。这样，编译器就可以确保该属性具有布尔值，并在您的 IDE 中为其启用自动完成功能。

因此，在 IDE 或编辑器中打开指定的源文件，然后添加新的接口、类或参数。更改后的代码应如下所示。我们添加的行以粗体显示。

TypeScript

文件：`lib/multistack-stack.ts`

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultiStackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here
```

```
  }
}
```

JavaScript

文件：lib/multistack-stack.js

JavaScript 没有接口功能；我们不需要添加任何代码。

```
const cdk = require('aws-cdk-stack');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your stack goes here
  }
}

module.exports = { MultistackStack }
```

Python

multistack/multistack_stack.py 文件：

Python 没有接口功能，因此我们将通过添加关键字参数来扩展堆栈以接受新属性。

```
import aws_cdk as cdk
from constructs import Construct

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines your stack goes here
```

Java

src/main/java/com/myorg/MultistackStack.java 文件：

在 Java 中扩展 props 类型比我们真正想说的要复杂得多。相反，编写堆栈的构造函数以接受可选的布尔参数。因为`props`是一个可选参数，所以我们将编写一个额外的构造函数来让你跳过它。它将默认为`false`。

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;

import software.amazon.awscdk.services.s3.Bucket;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id, boolean
 encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    public MultistackStack(final Construct scope, final String id, final StackProps
 props,
            final boolean encryptBucket) {
        super(scope, id, props);

        // The code that defines your stack goes here
    }
}
```

C#

`src/Multistack/MultistackStack.cs` 文件：

```
using Amazon.CDK;
using constructs;

namespace Multistack
{
```

```
    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }


    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, MultiStackProps props) :
 base(scope, id, props)
        {
            // The code that defines your stack goes here
        }
    }
}
```

新属性是可选的。如果 encryptBucket (Python:encrypt_bucket) 不存在，则其值为undefined，或本地等效值。默认情况下，存储桶将处于未加密状态。

定义堆栈类

现在让我们使用我们的新属性来定义我们的堆栈类。使代码如下所示。您需要添加或更改的代码以粗体显示。

TypeScript

文件：lib/multistack-stack.ts

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from constructs;
import * as s3 from 'aws-cdk-lib/aws-s3';

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultistackProps) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
```

```
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if (props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}
```

JavaScript

文件：lib/multistack-stack.js

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if ( props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}

module.exports = { MultistackStack }
```

Python

文件：multistack/multistack_stack.py

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add a Boolean property "encryptBucket" to the stack constructor.
        # If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
        # Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if encrypt_bucket:
            s3.Bucket(self, "MyGroovyBucket",
                      encryption=s3.BucketEncryption.KMS_MANAGED,
                      removal_policy=cdk.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                     removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

文件：src/main/java/com/myorg/MultistackStack.java

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.RemovalPolicy;

import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id,
            boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
```

```
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    // main constructor
    public MultistackStack(final Construct scope, final String id,
            final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is
        // unencrypted. Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if (encryptBucket) {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .encryption(BucketEncryption.KMS_MANAGED)
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        } else {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        }
    }
}
```

C#

文件：src/Multistack/MultistackStack.cs

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace Multistack
{

    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }

    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, IMultiStackProps props =
 null) : base(scope, id, props)
```

```
        {
            // Add a Boolean property "EncryptBucket" to the stack constructor.
            // If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
            // Encrypted bucket uses KMS-managed keys (SSE-KMS).
            if (props?.EncryptBucket ?? false)
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    Encryption = BucketEncryption.KMS_MANAGED,
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
            else
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
        }
    }
}
```

创建两个堆栈实例

现在我们将添加代码来实例化两个单独的堆栈。和以前一样，粗体显示的代码行是你需要添加的代码行。删除现有`MultistackStack`定义。

TypeScript

文件：`bin/multistack.ts`

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MultistackStack } from '../lib/multistack-stack';

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
```

```
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

JavaScript

文件：bin/multistack.js

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { MultistackStack } = require('../lib/multistack-stack');

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

Python

文件：./app.py

```
#!/usr/bin/env python3

import aws_cdk as cdk

from multistack.multistack_stack import MultistackStack

app = cdk.App()
```

```
MultistackStack(app, "MyWestCdkStack",
                env=cdk.Environment(region="us-west-1"),
                encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
                env=cdk.Environment(region="us-east-1"),
                encrypt_bucket=True)

app.synth()
```

Java

文件：src/main/java/com/myorg/MultistackApp.java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MultistackApp {
    public static void main(final String argv[]) {
        App app = new App();

        new MultistackStack(app, "MyWestCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-west-1")
                        .build())
                .build(), false);

        new MultistackStack(app, "MyEastCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-east-1")
                        .build())
                .build(), true);

        app.synth();
    }
}
```

C#

文件：src/Multistack/Program.cs

```
using Amazon.CDK;

namespace Multistack
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-west-1" },
                EncryptBucket = false
            });

            new MultistackStack(app, "MyEastCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-east-1" },
                EncryptBucket = true
            });

            app.Synth();
        }
    }
}
```

此代码使用MultistackStack类上的新 encryptBucket (Python:encrypt_bucket) 属性来实例化以下内容：

- 一个堆栈，其中包含该us-east-1 AWS 地区已加密 Amazon S3 存储桶。
- 一个堆栈，其中包含该us-west-1 AWS 地区未加密的 Amazon S3 存储桶。

合成并部署堆栈

现在，您可以通过应用程序部署堆栈。首先，为 MyEastCdkStack —stack in us-east-1 合成一个 AWS CloudFormation 模板。这是带有加密 S3 存储桶的堆栈。

```
$ cdk synth MyEastCdkStack
```

要将此堆栈部署到您的 AWS 账户，请发出以下命令之一。第一个命令使用您的默认 AWS 配置文件来获取部署堆栈的证书。第二个使用您指定的配置文件。将 *PROFILE_N* AME 替换为包含部署到该地区的相应凭据的 AWS CLI 配置文件名称。us-east-1 AWS

```
cdk deploy MyEastCdkStack
```

```
cdk deploy MyEastCdkStack --profile=PROFILE_NAME
```

清理

为避免对部署的资源收费，请使用以下命令销毁堆栈。

```
cdk destroy MyEastCdkStack
```

如果堆栈的存储桶中存储了任何内容，则销毁操作将失败。如果您只按照本主题中的说明进行操作，则不应该这样做。但是，如果您确实在存储桶中放了一些东西，则必须在销毁堆栈之前删除存储桶中的内容。（请勿删除存储桶本身。）使用 AWS Management Console 或删除存储桶中的内容。 AWS CLI

示例

本主题包含以下示例：

- 创建无服务器 Hello World 应用程序使用 Lambda、API Gateway 和 Amazon S3 创建无服务器应用程序。
- 使用创建 AWS Fargate 服务 AWS CDK使用镜像创建 Amazon ECS Fargate 服务。 DockerHub

使用创建 AWS Fargate 服务 AWS CDK

此示例将向您介绍如何创建在亚马逊弹性容器服务 (Amazon ECS) Service 集群上运行的 AWS Fargate 服务，该集群前面是来自亚马逊 ECR 上映像的面向互联网的应用程序负载均衡器。

Amazon ECS 是一项高度可扩展的快速容器管理服务，它可轻松运行、停止和管理群集上的 Docker 容器。您可以使用 Fargate 启动类型启动服务或任务，将集群托管在由 Amazon ECS 管理的无服务器基础设施上。为了获得更多控制权，您可以在使用亚马逊 EC2 启动类型管理的亚马逊弹性计算云 (Amazon EC2) 实例集群上托管任务。

本教程向您展示如何使用 Fargate 启动类型启动某些服务。如果您使用创建了 Fargate 服务，那么您就知道要完成该任务需要遵循许多步骤。 AWS Management Console AWS 有几个教程和文档主题可以指导你创建 Fargate 服务，包括：

- 如何部署 Docker 容器- AWS
- 使用 Amazon ECS 进行设置
- 使用 Fargate 开始使用 Amazon ECS

此示例在代码中创建了一个类似的 Fargate 服务。 AWS CDK

本教程中使用的 Amazon ECS 结构通过提供以下好处来帮助您使用 AWS 服务：

- 自动配置负载均衡器。
- 自动为负载均衡器打开安全组。这使负载均衡器无需您明确创建安全组即可与实例通信。
- 自动排序服务与附加到目标组的负载均衡器之间的依赖关系，其中 AWS CDK 强制执行在创建实例之前创建监听器的正确顺序。
- 自动配置自动伸缩组的用户数据。这将创建正确的配置以将集群关联到 AMI。

- 尽早验证参数组合。这样可以更早地暴露 AWS CloudFormation 问题，从而节省部署时间。例如，根据任务的不同，很容易错误配置内存设置。以前，在部署应用程序之前，您不会遇到错误。但是现在，当你合成应用程序时， AWS CDK 可以检测到配置错误并发出错误。
- 如果您使用来自亚马逊 ECR 的图片，则会自动添加亚马逊弹性容器注册表 (Amazon ECR) Container Registry 的权限。
- 自动缩放。 AWS CDK 提供了一种方法，这样您就可以在使用 Amazon EC2 集群时自动扩展实例。当您在 Fargate 集群中使用实例时，会自动发生这种情况。

  此外，当自动扩展尝试终止实例，但任务正在运行或计划在该实例上时，可以 AWS CDK 防止实例被删除。

  以前，您必须创建一个 Lambda 函数才能使用此功能。
- 提供资产支持，以便您可以一步将源从您的计算机部署到 Amazon ECS。以前，要使用应用程序源，您必须执行几个手动步骤，例如上传到 Amazon ECR 和创建 Docker 映像。

有关详细信息，请参阅 ECS。

> **Important**
>
> 我们将要使用的`ApplicationLoadBalancedFargateService`结构包括许多 AWS 组件，即使您不使用它们，如果将其中一些组件留在您的 AWS 账户中，也会产生不小的费用。完成此示例后，请务必清理 (cdk destroy)。

创建目录并初始化 AWS CDK

让我们先创建一个存放 AWS CDK 代码的目录，然后在该目录中创建一个 AWS CDK 应用程序。

TypeScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language typescript
```

JavaScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
```

```
cdk init --language javascript
```

Python

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language python
source .venv/bin/activate
pip install -r requirements.txt
```

Java

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language java
```

现在，您可以将 Maven 项目导入 IDE。

C#

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language csharp
```

您现在可以在 Visual Studio `src/MyEcsConstruct.sln` 中打开了。

运行应用程序并确认它创建了一个空堆栈。

```
cdk synth
```

创建 Fargate 服务

使用 Amazon ECS 运行容器任务有两种不同的方法：

- 使用`Fargate`启动类型，其中 Amazon ECS 为您管理运行容器的物理机。
- 使用`EC2`启动类型，您可以在其中进行管理，例如指定自动缩放。

在本示例中，我们将创建一个在面向互联网的应用程序负载均衡器前面的 ECS 集群上运行的 Fargate 服务。

将以下 AWS 构造库模块导入添加到指定的文件中。

TypeScript

文件：lib/my_ecs_construct-stack.ts

```
import * as ec2 from "aws-cdk-lib/aws-ec2";
import * as ecs from "aws-cdk-lib/aws-ecs";
import * as ecs_patterns from "aws-cdk-lib/aws-ecs-patterns";
```

JavaScript

文件：lib/my_ecs_construct-stack.js

```
const ec2 = require("aws-cdk-lib/aws-ec2");
const ecs = require("aws-cdk-lib/aws-ecs");
const ecs_patterns = require("aws-cdk-lib/aws-ecs-patterns");
```

Python

文件：my_ecs_construct/my_ecs_construct_stack.py

```
from aws_cdk import (aws_ec2 as ec2, aws_ecs as ecs,
                     aws_ecs_patterns as ecs_patterns)
```

Java

文件：src/main/java/com/myorg/MyEcsConstructStack.java

```
import software.amazon.awscdk.services.ec2.*;
import software.amazon.awscdk.services.ecs.*;
import software.amazon.awscdk.services.ecs.patterns.*;
```

C#

文件：src/MyEcsConstruct/MyEcsConstructStack.cs

```
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;
```

将构造函数末尾的注释替换为以下代码。

TypeScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

JavaScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

Python

```
        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is True
```

Java

```
        Vpc vpc = Vpc.Builder.create(this, "MyVpc")
                             .maxAzs(3)  // Default is all AZs in region
                             .build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                             .vpc(vpc).build();

        // Create a load-balanced Fargate service and make it public
        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                        .cluster(cluster)           // Required
                        .cpu(512)                   // Default is 256
                         .desiredCount(6)            // Default is 1
                         .taskImageOptions(
                                 ApplicationLoadBalancedTaskImageOptions.builder()
                                         .image(ContainerImage.fromRegistry("amazon/
amazon-ecs-sample"))
                                         .build())
                         .memoryLimitMiB(2048)       // Default is 512
                         .publicLoadBalancer(true)   // Default is true
                         .build();
```

C#

```
            var vpc = new Vpc(this, "MyVpc", new VpcProps
            {
```

```
                MaxAzs = 3 // Default is all AZs in region
            });

            var cluster = new Cluster(this, "MyCluster", new ClusterProps
            {
                Vpc = vpc
            });

            // Create a load-balanced Fargate service and make it public
            new ApplicationLoadBalancedFargateService(this, "MyFargateService",
                new ApplicationLoadBalancedFargateServiceProps
                {
                    Cluster = cluster,          // Required
                    DesiredCount = 6,           // Default is 1
                    TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
                    {
                        Image = ContainerImage.FromRegistry("amazon/amazon-ecs-
sample")
                    },
                    MemoryLimitMiB = 2048,      // Default is 256
                    PublicLoadBalancer = true   // Default is true
                }
            );
```

保存并确保它运行并创建堆栈。

```
cdk synth
```

堆栈有数百行，所以我们不在这里显示。堆栈应包含一个默认实例、三个可用区的私有子网和公有子网以及一个安全组。

部署堆栈。

```
cdk deploy
```

AWS CloudFormation 显示有关它在部署您的应用程序时所执行的数十个步骤的信息。

这就是创建由 Fargate 提供支持的 Amazon ECS 服务来运行 Docker 镜像非常容易。

清理

为避免意 AWS 外冲锋，请在完成本练习后销毁你的 AWS CDK 堆栈。

```
cdk destroy
```

AWS CDK 例子

有关使用你最喜欢的支持的编程语言的 AWS CDK 堆栈和应用程序的更多AWS CDK 示例，请参阅上GitHub的 Examples 存储库。

AWS CDK 工具

本节包含有关下列 AWS CDK 工具的信息。

主题

AWS CDK 工具包（**cdk**命令）

AWS CDK Toolkit（cdk即 CLI 命令）是与您的 AWS CDK 应用程序交互的主要工具。它执行您的应用程序，查询您定义的应用程序模型，并生成和部署由生成的 AWS CloudFormation 模板。 AWS CDK它还提供了其他可用于创建和处理 AWS CDK 项目的功能。本主题包含有关 CDK 工具包的常见用例的信息。

该 AWS CDK 工具包与 Node Package Manager 一起安装。在大多数情况下，我们建议在全球范围内进行安装。

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

> **Tip**
>
> 如果您经常使用多个版本的 AWS CDK，请考虑在单个 CDK 项目中安装相应版本的 AWS CDK Toolkit。为此，npm install请在命令-g中省略。然后使用npx aws-cdk来调用它。如果存在本地版本，则运行本地版本；如果不存在，则回退到全局版本。

工具包命令

所有 CDK Toolkit 命令都以开头cdk，后面是子命令（listsynthesize、deploy、等）。有些子命令具有等效的较短版本（lssynth、等）。选项和参数按任意顺序跟在子命令之后。这里汇总了可用命令。

命令	函数
`cdk list` (`ls`)	列出应用程序中的堆栈
`cdk synthesize` (`synth`)	合成并打印一个或多个指定堆栈的 CloudFormation 模板
`cdk bootstrap`	部署 CDK Toolkit 暂存堆栈；请参阅 the section called “正在引导”
`cdk deploy`	部署一个或多个指定的堆栈
`cdk destroy`	摧毁一个或多个指定的堆栈
`cdk diff`	将指定的堆栈及其依赖关系与已部署的堆栈或本地 CloudFormation模板进行比较
`cdk import`	使用 CloudFormation 资源导入将现有资源引入 CDK 管理的堆栈
`cdk metadata`	显示有关指定堆栈的元数据
`cdk init`	使用指定模板在当前目录下创建新的 CDK 项目
`cdk context`	管理缓存的上下文值
`cdk docs` (`doc`)	在浏览器中打开 CDK API 参考
`cdk doctor`	检查您的 CDK 项目是否存在潜在问题

有关每个命令的可用选项，请参见the section called “内置帮助”。

指定选项及其值

命令行选项以两个连字符 (`--`) 开头。一些常用的选项具有以单个连字符开头的单字母同义词（例如，`--app`具有同义词）。`-a` AWS CDK Toolkit 命令中选项的顺序并不重要。

所有选项都接受一个值，该值必须位于选项名称之后。该值可以用空格或等号`=`与名称分隔。以下两个选项是等效的。

```
--toolkit-stack-name MyBootstrapStack
--toolkit-stack-name=MyBootstrapStack
```

有些选项是标志（布尔值）。您可以指定`true`或`false`作为它们的值。如果您不提供值，则该值将被视为`true`。您也可以在选项名称前加上前缀`no-`以暗示`false`。

```
# sets staging flag to true
--staging
--staging=true
--staging true

# sets staging flag to false
--no-staging
--staging=false
--staging false
```

可以多次指定几个选项 `--context` `--parameters`、`--plugin`、`--tags`和`--trust`，以指定多个值。在 CDK Toolkit 帮助中注明这些内容是有`[array]`键入的。例如：

```
cdk bootstrap --tags costCenter=0123 --tags responsibleParty=jdoe
```

内置帮助

该 AWS CDK 工具包已集成帮助。您可以通过发出以下命令来查看有关该实用程序的一般帮助和所提供的子命令列表：

```
cdk --help
```

例如`deploy`，要查看特定子命令的帮助，请在`--help`标记之前指定该子命令。

```
cdk deploy --help
```

显示 AWS CDK 工具包版本的问题`cdk version`。在请求支持时提供此信息。

版本报告

为了深入了解 AWS CDK 是如何使用的， AWS CDK 应用程序使用的构造是通过使用标识为`AWS::CDK::Metadata`的资源来收集和报告的。此资源已添加到 AWS CloudFormation 模板中，

可以轻松查看。这些信息还可用于使用存在已知安全性或可靠性问题的结构来识别堆栈。 AWS 它还可用于与用户联系以提供重要信息。

> **Note**
>
> 在 1.93.0 版本之前， AWS CDK 报告的是合成过程中加载的模块的名称和版本，而不是堆栈中使用的构造。

默认情况下，它们会 AWS CDK 报告堆栈中使用的以下 NPM 模块中构造的使用情况：

- AWS CDK 核心模块
- AWS 构造库模块
- AWS 解决方案构造模块
- AWS 渲染农场部署套件模块

该`AWS::CDK::Metadata`资源如下所示。

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Analytics:
 "v2:deflate64:H4sIAND9SGAAAzXKSw5AMBAA0L1b2PdzBYnEAdio3RglglY60zQi7u6TWL/
XKmNUlxeQSOKwaPTBqrNhwEWU3hGHiCzK0dWWfAxoL/Fd8mvk+QkS/0X6BdjnCdgmOOQKWz
+AqqLDt2Y3YMnLYWwAAAA="
```

该`Analytics`属性是堆栈中构造的 gzip、base64 编码、前缀编码列表。

要选择退出版本报告，请使用以下方法之一：

- 使用带有`--no-version-reporting`参数的`cdk`命令可以选择退出单个命令。

  ```
  cdk --no-version-reporting synth
  ```

  请记住， AWS CDK Toolkit 在部署之前会合成新的模板，因此您还应该在`cdk deploy`命令中`--no-version-reporting`添加新模板。
- 在`./cdk.json`或中设置versionReporting为 false `~/.cdk.json`。除非您通过在单个命令`--version-reporting`上指定来选择加入，否则会选择退出。

```
{
  "app": "...",
  "versionReporting": false
}
```

使用进行身份验证 AWS

您可以通过不同的方式配置对 AWS 资源的编程访问权限，具体取决于环境和可用的 AWS 访问权限。

要选择您的身份验证方法并针对 CDK Toolkit 进行配置，请参阅 AWS SDK 和工具参考指南中的身份验证和访问权限。

对于在本地开发的新用户，如果雇主没有向他们提供身份验证方法，推荐的方法是设置 AWS IAM Identity Center。此方法包括安装 AWS CLI 以便于配置和定期登录 AWS 访问门户。如果选择此方法，则在完成 AWS SDK 和工具参考指南中的 IAM Identity Center 身份验证程序后，您的环境应包含以下元素：

- AWS CLI，用于在运行应用程序之前启动 AWS 访问门户会话。
- 一种共享 AWSconfig文件，其[default]配置文件包含一组配置值，可以从中引用 AWS CDK。要查找此文件的位置，请参阅《AWS SDK 和工具参考指南》中的共享文件的位置。
- 共享 config 文件设置了 region 设置。这将设置 AWS CDK 和CDK工具包 AWS 区域 在 AWS 请求中使用的默认值。
- CDK Toolkit 使用配置文件的 SSO 令牌提供程序配置来获取凭证，然后再向发送请求。 AWS 该sso_role_name值是与 IAM Identity Center 权限集关联的 IAM 角色，应允许访问您的应用程序中 AWS 服务 使用的权限。

  以下示例 config 文件展示了使用 SSO 令牌提供程序配置来设置的默认配置文件。配置文件的 sso_session 设置是指所指定的 sso-session 节。该sso-session部分包含启动 AWS 访问门户会话的设置。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
```

```
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

启动 AWS 访问门户会话

在访问之前 AWS 服务，您需要对 CDK Toolkit 进行有效的 AWS 访问门户会话，才能使用 IAM Identity Center 身份验证来解析证书。根据您配置的会话时长，您的访问权限最终将过期，CDK Toolkit 将遇到身份验证错误。在中运行以下命令登录 AWS CLI AWS 访问门户。

```
aws sso login
```

如果您的 SSO 令牌提供程序配置使用命名配置文件而不是默认配置文件，则命令为`aws sso login --profile NAME`。使用--profile选项或`AWS_PROFILE`环境变量发出cdk命令时，也要指定此配置文件。

要测试是否已有活动会话，请运行以下 AWS CLI 命令。

```
aws sts get-caller-identity
```

对此命令的响应应该报告共享 `config` 文件中配置的 IAM Identity Center 账户和权限集。

> **Note**
>
> 如果您已经有一个有效的 AWS 访问门户会话并且`aws sso login`正在运行，则无需提供凭据。
>
> 登录过程可能会提示您允许 AWS CLI 访问您的数据。由于 AWS CLI 是在适用于 Python 的 SDK 之上构建的，因此权限消息可能包含`botocore`名称的变体。

指定区域和其他配置

CDK Toolkit 需要知道您要部署到的 AWS 区域以及如何进行 AWS身份验证。这是部署操作和合成期间检索上下文值所必需的。您的账户和地区共同构成了环境。

可以使用环境变量或在配置文件中指定区域。这些变量和文件与其他 AWS 工具（例如和各种 AWS SDK）使用的变量 AWS CLI 和文件相同。CDK 工具包按以下顺序查找此信息。

- 环境变量 AWS_DEFAULT_REGION
- 在标准 AWS config文件中定义的命名配置文件，并使用cdk命令上的--profile选项指定。
- 标准 AWS config文件的[default]部分。

除了在该[default]部分中指定 AWS 身份验证和区域外，您还可以添加一个或多个[profile *NAME*]部分，其中 *NAM* E 是配置文件的名称。有关命名配置文件的更多信息，请参阅 AWS SDK 和工具参考指南中的共享配置和凭据文件。

标准 AWS config文件位于 ~/.aws/config (macOS/Linux) 或 %USERPROFILE%\.aws\config (Windows)。有关详细信息和备用位置，请参阅 AWS SDK 和工具参考指南中的共享配置和凭据文件的位置

合成过程中会使用您在 AWS CDK 应用程序中使用堆栈env属性指定的环境。它用于生成特定于环境的 AWS CloudFormation 模板，在部署期间，它会覆盖通过上述方法之一指定的账户或区域。有关更多信息，请参阅 the section called “环境”。

> **Note**
>
> AWS CDK 使用与其他 AWS 工具和软件开发工具包相同的源文件中的凭证，包括。AWS Command Line Interface但是，它们的行为 AWS CDK 可能与这些工具略有不同。它使用引擎盖 AWS SDK for JavaScript 下方。有关为设置凭据的完整详细信息 AWS SDK for JavaScript，请参阅设置凭据。

您可以选择使用--role-arn（或-r）选项来指定应用于部署的 IAM 角色的 ARN。此角色必须由所使用的 AWS 账户担任。

指定应用程序命令

CDK Toolkit 的许多功能都需要合成一个或多个 AWS CloudFormation 模板，这反过来又需要运行您的应用程序。 AWS CDK 支持用多种语言编写的程序。因此，它使用配置选项来指定运行应用程序所需的确切命令。可以通过两种方式指定此选项。

首先，也是最常见的，可以使用文件中的app密钥进行指定cdk.json。它位于 AWS CDK 项目的主目录中。使用创建新项目时，CDK 工具包提供了相应的命令。cdk init例如，以下是cdk.json来自一个新 TypeScript 项目的内容。

```
{
```

```
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

尝试运行您的应用程序时，CDK Toolkit 会在当前工作目录`cdk.json`中查找。因此，你可以在项目的主目录中打开一个 shell 来发出 CDK Toolkit 命令。

如果找不到应用程序密钥，CDK Toolkit 还会在`~/.cdk.json`（即您的主目录中）中`./cdk.json`查找该密钥。如果您通常使用相同语言的 CDK 代码，则在此处添加 app 命令会很有用。

如果您位于其他目录中，或者要使用中以外的命令运行应用程序`cdk.json`，请使用`--app`（或`-a`）选项进行指定。

```
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

部署时，您还可以将包含合成云程序集的目录（例如）指定为的值。`cdk.out` --app指定的堆栈是从此目录部署的；应用程序不是合成的。

指定堆栈

许多 CDK Toolkit 命令（例如`cdk deploy`）都适用于应用程序中定义的堆栈。如果您的应用程序仅包含一个堆栈，那么如果您没有明确指定堆栈，CDK Toolkit 会假设您的意思是那个堆栈。

否则，您必须指定要使用的堆栈。您可以通过在命令行中按照 ID 单独指定所需的堆栈来实现此目的。回想一下，ID 是实例化堆栈时第二个参数指定的值。

```
cdk synth PipelineStack LambdaStack
```

您也可以使用通配符来指定与模式匹配的 ID。

- `?`匹配任何单个字符
- `*`匹配任意数量的字符（`*`单独匹配所有堆栈）
- `**`匹配层次结构中的所有内容

您也可以使用该--all选项来指定所有堆栈。

如果你的应用程序使用 CDK Pipelines，CDK 工具包会将你的堆栈和阶段理解为一个层次结构。此外，该--all选项和`*`通配符仅匹配顶级堆栈。要匹配所有堆栈，请使用`**`。也可以`**`用来表示特定层次结构下的所有堆栈。

使用通配符时，请用引号将模式括起来，或者使用转义通配符。\如果不这样做，你的 shell 可能会尝试将模式扩展为当前目录中的文件名。充其量，这不会达到你的预期；在最坏的情况下，你可以部署你本来不打算部署的堆栈。在 Windows 上，这并不是绝对必要的，因为这cmd.exe不会扩展通配符，但还是不错的做法。

```
cdk synth "*Stack"    # PipelineStack, LambdaStack, etc.
cdk synth 'Stack?'    # StackA, StackB, Stack1, etc.
cdk synth \*          # All stacks in the app, or all top-level stacks in a CDK
 Pipelines app
cdk synth '**'        # All stacks in a CDK Pipelines app
cdk synth 'PipelineStack/Prod/**'   # All stacks in Prod stage in a CDK Pipelines app
```

> **Note**
>
> 您指定堆栈的顺序不一定是堆栈的处理顺序。在决定堆栈的处理顺序时， AWS CDK Toolkit 会考虑堆栈之间的依赖关系。例如，假设一个堆栈使用另一个堆栈产生的值（例如第二个堆栈中定义的资源的 ARN）。在这种情况下，由于这种依赖关系，第二个堆栈在第一个堆栈之前合成。您可以使用堆栈的addDependency()方法在堆栈之间手动添加依赖关系。

引导您的环境 AWS

使用 CDK 部署堆栈需要配置特殊的专用 AWS CDK 资源。该cdk bootstrap命令会为您创建必要的资源。只有在部署需要这些专用资源的堆栈时，才需要引导。有关详细信息，请参阅 the section called “正在引导”。

```
cdk bootstrap
```

如果在没有参数的情况下发出（如下所示），则该cdk bootstrap命令将合成当前应用程序并引导其堆栈将部署到的环境。如果应用程序包含与环境无关的堆栈，而这些堆栈没有明确指定环境，则会引导默认账户和区域，或者使用指定的环境。--profile

在应用程序之外，您必须明确指定要引导的环境。您也可以这样做来引导未在您的应用程序或本地 AWS 配置文件中指定的环境。必须为指定的账户和地区配置证书（例如在~/.aws/credentials）。您可以指定包含所需凭据的配置文件。

```
cdk bootstrap ACCOUNT-NUMBER/REGION # e.g.
cdk bootstrap 1111111111/us-east-1
```

```
cdk bootstrap --profile test 1111111111/us-east-1
```

> **Important**
>
> 部署此类堆栈的每个环境（账户/区域组合）都必须单独引导。

您可能会因为在引导资源中 AWS CDK 存储的内容而产生 AWS 费用。此外，如果您使用`--bootstrap-customer-key`，则会创建 AWS KMS 密钥，这也会对每个环境产生费用。

> **Note**
>
> 默认情况下，早期版本的引导模板创建了 KMS 密钥。为避免收费，请使用重新启动。`--no-bootstrap-customer-key`

> **Note**
>
> CDK Toolkit v2 不支持 CDK v1 中默认使用的原始引导模板（称为旧版模板）。

> **Important**
>
> 现代引导模板可以有效地向`--trust`列表中的任何 AWS 账户授`--cloudformation-execution-policies`予隐含的权限。默认情况下，这会扩展对引导账户中任何资源的读取和写入权限。请务必使用您熟悉的策略和可信帐户来配置引导堆栈。

创建新应用程序

要创建新应用程序，请为其创建一个目录，然后在目录内发出`cdk init`。

```
mkdir my-cdk-app
cd my-cdk-app
cdk init TEMPLATE --language LANGUAGE
```

支持的语言（*##*）有：

代码	Language
typescript	TypeScript
javascript	JavaScript
python	Python
java	Java
csharp	C#

##是一个可选的模板。如果所需的模板是 app（默认），则可以省略它。可用的模板有：

模板	描述
app（默认）	创建一个空 AWS CDK 应用程序。
sample-app	使用包含亚马逊 SQS 队列和亚马逊 SNS 主题的堆栈创建 AWS CDK 应用程序。

模板使用项目文件夹的名称为新应用程序中的文件和类生成名称。

列出堆栈

要查看 AWS CDK 应用程序中堆栈的 ID 列表，请输入以下等效命令之一：

```
cdk list
cdk ls
```

如果您的应用程序包含 CDK Pipelines 堆栈，CDK Toolkit 会根据堆栈名称在管道层次结构中的位置将堆栈名称显示为路径。（例如PipelineStackPipelineStack/Prod、和PipelineStack/Prod/MyService。）

如果您的应用程序包含许多堆栈，则可以指定要列出的堆栈的全部或部分堆栈 ID。有关更多信息，请参阅 the section called “指定堆栈”。

添加--long标志以查看有关堆栈的更多信息，包括堆栈名称及其环境（AWS 账户和区域）。

合成堆栈

该cdk synthesize命令（几乎总是缩写synth）将应用程序中定义的堆栈合成一个 CloudFormation 模板。

```
cdk synth         # if app contains only one stack
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*"     # all stacks in app
```

> **Note**
>
> CDK Toolkit 实际上是在大多数操作（例如部署或比较堆栈时）之前运行您的应用程序并合成新的模板。默认情况下，这些模板存储在cdk.out目录中。该cdk synth命令仅打印一个或多个指定堆栈生成的模板。

cdk synth --help有关所有可用选项，请参阅。以下部分将介绍一些最常用的选项。

指定上下文值

使用--context或-c选项将运行时上下文值传递给您的 CDK 应用程序。

```
# specify a single context value
cdk synth --context key=value MyStack

# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

部署多个堆栈时，通常会将指定的上下文值传递给所有堆栈。如果需要，可以通过在上下文值前加上堆栈名称前缀来为每个堆栈指定不同的值。

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

指定显示格式

默认情况下，合成后的模板以 YAML 格式显示。添加标--json志，改为以 JSON 格式显示它。

```
cdk synth --json MyStack
```

指定输出目录

添加 `--output` (`-o`) 选项，将合成后的模板写入除之外的`cdk.out`目录。

```
cdk synth --output=~/templates
```

部署堆栈

`cdk deploy`子命令将一个或多个指定的堆栈部署到您的账户。 AWS

```
cdk deploy        # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"    # all stacks in app
```

> **Note**
>
> 在部署任何内容之前，CDK Toolkit 会运行您的应用程序并合成新的 AWS CloudFormation 模板。因此，可以与一起使用的大多数命令行选项`cdk synth`（例如`--context`）也可以与一起使用`cdk deploy`。

`cdk deploy --help`有关所有可用选项，请参阅。以下部分将介绍一些最有用的选项。

跳过合成

该cdk deploy命令通常会在部署之前合成应用程序的堆栈，以确保部署反映应用程序的最新版本。如果您知道自上次代码以来没有更改过代码cdk synth，则可以在部署时取消冗余合成步骤。为此，请在--app选项中指定项目的`cdk.out`目录。

```
cdk deploy --app cdk.out StackOne StackTwo
```

禁用回滚

AWS CloudFormation 能够回滚更改，从而实现原子部署。这意味着它们要么成功，要么整体失败。AWS CDK 继承了此功能，因为它可以合成和部署 AWS CloudFormation 模板。

Rollback 可确保您的资源始终处于一致状态，这对于生产堆栈至关重要。但是，当你仍在开发基础架构时，有些故障是不可避免的，而回滚失败的部署可能会减慢你的速度。

因此，CDK Toolkit 允许您通过在命令中添加`--no-rollback`来禁用回滚。`cdk deploy`使用此标志，失败的部署不会被回滚。相反，在出现故障的资源之前部署的资源会保留在原处，而下一次部署则从出现故障的资源开始。等待部署的时间将大大减少，而将更多的时间花在开发基础架构上。

热交换

使用带的`--hotswap`标志`cdk deploy`尝试直接更新您的 AWS 资源，而不是生成 AWS CloudFormation 更改集并进行部署。如果无法进行热交换，则 AWS CloudFormation 部署将回退到部署。

目前热交换支持 Lambda 函数、Step Functions 状态机和 Amazon ECS 容器镜像。该`--hotswap`标志还会禁用回滚（即暗示`--no-rollback`）。

> **Important**
>
> 不建议在生产部署中使用热插拔。

观看模式

CDK Toolkit 的监视模式（cdk deploy --watch或cdk watch简称）会持续监视您的 CDK 应用程序的源文件和资产是否有更改。当检测到更改时，它会立即部署指定的堆栈。

默认情况下，这些部署使用标志，该`--hotswap`标志可快速跟踪对 Lambda 函数的更改的部署。AWS CloudFormation 如果您更改了基础架构配置，它还会回退到通过部署。要`cdk watch`始终执行完整 AWS CloudFormation 部署，请将`--no-hotswap`标志添加到`cdk watch`。

`cdk watch`正在执行部署时所做的任何更改都将合并到一个部署中，该部署将在正在进行的部署完成后立即开始。

监视模式使用项目中的`"watch"`密钥`cdk.json`来确定要监视哪些文件。默认情况下，这些文件是您的应用程序文件和资产，但可以通过修改`"watch"`密钥中的`"include"`和`"exclude"`条目来更改这些文件和资产。以下`cdk.json`文件显示了这些条目的示例。

```
{
  "app": "mvn -e -q compile exec:java",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
```

```
}
```

cdk watch在合成之前执行来自cdk.json的"build"命令来构建您的应用程序。如果您的部署需要任何命令来构建或打包您的 Lambda 代码（或不在您的 CDK 应用程序中的任何其他内容），请将其添加到此处。

和键中可以使用 Git 风格的通配符**，包括*和。"watch" "build"每个路径都是相对于的父目录进行解释的cdk.json。的默认值include为**/*，表示项目根目录中的所有文件和目录。 exclude是可选的。

> **Important**
>
> 对于生产部署，建议不要使用监视模式。

指定 AWS CloudFormation 参数

该 AWS CDK 工具包支持在部署时指定 AWS CloudFormation 参数。你可以在命令行上的--parameters标志后面提供这些信息。

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

要定义多个参数，请使用多个--parameters标志。

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

如果要部署多个堆栈，则可以为每个堆栈的每个参数指定不同的值。为此，请在参数名称前加上堆栈名称和冒号。否则，将相同的值传递给所有堆栈。

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

默认情况下， AWS CDK 会保留先前部署中的参数值，如果未明确指定，则在以后的部署中使用这些值。使用该--no-previous-parameters标志要求指定所有参数。

指定输出文件

如果您的堆栈声明了 AWS CloudFormation 输出，则这些输出通常会在部署结束时显示在屏幕上。要将它们写入 JSON 格式的文件中，请使用标--outputs-file志。

```
cdk deploy --outputs-file outputs.json MyStack
```

与安全相关的更改

为了保护您免受影响安全状况的意外更改， AWS CDK Toolkit 会提示您在部署与安全相关的更改之前批准这些更改。您可以指定需要批准的更改级别：

```
cdk deploy --require-approval LEVEL
```

##可以是以下之一：

租期	含义
never	无需批准
any-change	任何 IAM 或 security-group-related 变更都需要获得批准
broadening （默认）	添加 IAM 声明或流量规则时需要批准；移除不需要批准

也可以在cdk.json文件中配置该设置。

```
{
  "app": "...",
  "requireApproval": "never"
}
```

比较堆栈

该cdk diff命令将应用程序中定义的堆栈（及其依赖项）的当前版本与已部署的版本或已保存的 AWS CloudFormation 模板进行比较，并显示更改列表。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                    # Effect # Action                      # Principal
                  # Condition #
###############################################################################################
```

```
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole               #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                              #
                    #           #
#   # .Arn}                          #        #                              #
                    #           #
#############################################################################################
# + # ${MyFirstBucket.Arn}         # Allow  # s3:DeleteObject*             # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*       #        # s3:GetBucket*                #
 jectsCustomResourceProvider/ #           #
#   #                              #        # s3:GetObject*                # Role.Arn}
                    #           #
#   #                              #        # s3:List*                     #
                    #           #
#############################################################################################
IAM Policy Changes
#############################################################################################
#   # Resource                                                  # Managed Policy ARN
                              #
#############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                       # ice-role/
AWSLambdaBasicExecutionRole"}                #
#############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
```

```
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
     ## [-] Retain
     ## [+] Delete
```

要将您的应用程序堆栈与现有部署进行比较，请执行以下操作：

```
cdk diff MyStack
```

要将应用的堆栈与已保存的 CloudFormation 模板进行比较，请执行以下操作：

```
cdk diff --template ~/stacks/MyStack.old MyStack
```

将现有的资源导入到堆栈

您可以使用`cdk import`命令将特定 AWS CDK 堆栈的资源置于 CloudFormation 的管理之下。如果您要迁移到堆栈 AWS CDK，或者要在堆栈之间移动资源或更改其逻辑 ID，则此功能非常有用。`cdk import`使用 CloudFormation 资源导入。请在此处查看可以导入的资源列表。

要将现有资源导入 AWS CDK 堆栈，请执行以下步骤：

- 确保资源当前未由任何其他 CloudFormation 堆栈管理。如果是，请先将移除策略设置为`RemovalPolicy.RETAIN`资源当前所在的堆栈中，然后执行部署。然后，从堆栈中移除资源并执行另一次部署。此过程将确保资源不再由管理， CloudFormation 但不会将其删除。

- 运行 `acdk diff`，确保要将资源导入到的 AWS CDK 堆栈中没有待处理的更改。“导入” 操作中唯一允许的更改是添加要导入的新资源。
- 为要导入堆栈的资源添加结构。例如，如果您想导入 Amazon S3 存储桶，请添加类似的内容`new s3.Bucket(this, 'ImportedS3Bucket', {});`。请勿对任何其他资源进行任何修改。

  您还必须确保将资源当前的状态精确地建模到定义中。以存储桶为例，请务必包含 AWS KMS 密钥、生命周期策略以及与存储桶相关的任何其他内容。否则，后续更新操作可能无法达到预期效果。

  您可以选择是否包含物理存储桶名称。我们通常建议不要在资源定义中包含 AWS CDK 资源名称，这样可以更轻松地多次部署资源。
- 运行 `cdk import` *`STACKNAME`*。
- 如果您的模型中没有资源名称，CLI 将提示您传入要导入的资源的实际名称。之后，开始导入。
- `cdk import`报告成功后，资源现在由 AWS CDK 和管理 CloudFormation。您随后对 AWS CDK 应用程序中的资源属性（构造配置）所做的任何更改都将应用于下一次部署。
- 要确认 AWS CDK 应用程序中的资源定义是否与资源的当前状态相匹配，您可以启动CloudFormation 偏差检测操作。

此功能目前不支持将资源导入嵌套堆栈。

配置 (`cdk.json`)

许多 CDK Toolkit 命令行标志的默认值可以存储在项目`cdk.json`.`cdk.json`文件中或用户目录的文件中。以下是按字母顺序对支持的配置设置的引用。

键	注意事项	CDK 工具包选项
`app`	执行 CDK 应用程序的命令。	--app
`assetMetadata`	如果是`false`，CDK 不会向使用资产的资源添加元数据。	--no-asset-metadata
`bootstrapKmsKeyId`	覆盖用于加密 Amazon S3 部署存储桶的密 AWS KMS 钥的 ID。	--bootstrap-kms-key-id

键	注意事项	CDK 工具包选项
build	在合成之前编译或构建 CDK 应用程序的命令。不允许进入~/.cdk.json 。	--build
browser	用于启动cdk docs子命令的 Web 浏览器的命令。	--browser
context	请参阅 the section called “上下文”。配置文件中的上下文值不会被删除。cdk context --clear（CDK 工具包将缓存的上下文值放入其中cdk.context.json 。）	--context
debug	如果是true，CDK Toolkit 会发出对调试有用的更多详细信息。	--debug
language	用于初始化新项目的语言。	--language
lookups	如果是false，则不允许进行上下文查找。如果需要执行任何上下文查找，则合成将失败。	--no-lookups
notices	如果false，则禁止显示有关安全漏洞、回归和不支持的版本的消息。	--no-notices
output	合成云程序集将发射到的目录的名称（默认"cdk.out" ）。	--output
outputsFile	将已部署堆栈的 AWS CloudFormation 输出写入的文件（采用 JSON 格式）。	--outputs-file

键	注意事项	CDK 工具包选项
`pathMetadata`	如果是`false`，则不会将 CDK 路径元数据添加到合成模板中。	--no-path-metadata
`plugin`	JSON 数组，用于指定扩展 CDK 的软件包名称或本地路径	--plugin
`profile`	用于指定区域和账户凭证的默认 AWS 配置文件的名称。	--profile
`progress`	如果设置为`"events"`，CDK Toolkit 将显示部署期间的所有 AWS CloudFormation 事件，而不是进度条。	--progress
`requireApproval`	安全更改的默认批准级别。请参阅 the section called “与安全相关的更改”。	--require-approval
`rollback`	如果是`false`，则失败的部署不会被回滚。	--no-rollback
`staging`	如果是`false`，则不会将资源复制到输出目录（用于对源文件进行本地调试 AWS SAM）。	--no-staging
`tags`	包含堆栈标签（键值对）的 JSON 对象。	--tags
`toolkitBucketName`	用于部署 Lambda 函数和容器映像等资产的 Amazon S3 存储桶的名称（请参阅。the section called “引导您的环境 AWS”	--toolkit-bucket-name

键	注意事项	CDK 工具包选项
`toolkitStackName`	引导堆栈的名称（请参阅the section called “引导您的环境 AWS”。	--toolkit-stack-name
`versionReporting`	如果`false`是，则选择退出版本报告。	--no-version-reporting
`watch`	JSON 对象包含`"include"` 和用于指示哪些文件在更改时应该（或不应该）触发项目重建的`"exclude"` 密钥。请参阅 the section called “观看模式”。	--watch

cdk migrate 命令参考

AWS Cloud Development Kit (AWS CDK) 命令行界面 (CLI) `cdk migrate` 命令的参考。有关使用的更多信息`cdk migrate`，请参阅将现有资源和 AWS CloudFormation 模板迁移到 AWS CDK。

该`cdk migrate`命令将已部署的 AWS 资源、 AWS CloudFormation 堆栈和本地 AWS CloudFormation 模板迁移到。 AWS CDK

主题

- 使用量
- Options

使用量

```
$ cdk migrate <options>
```

Options

必填选项

`--stack-name` *STRING*

迁移后将在 CDK 应用程序中创建的 AWS CloudFormation 堆栈的名称。

必需：是

条件选项

`--from-path` *PATH*

要迁移的 AWS CloudFormation 模板的路径。提供此选项以指定本地模板。

必填：条件性。如果从本地 AWS CloudFormation 模板迁移，则为必填项。

`--from-scan` *STRING*

从 AWS 环境迁移已部署的资源时，使用此选项来指定是应启动新的扫描，还是 AWS CDK CLI应使用上次成功的扫描。

必填：条件性。从已部署的 AWS 资源迁移时为必填项。

可接受的值：`most-recent`，`new`

`--from-stack`

提供此选项以从已部署的 AWS CloudFormation 堆栈迁移。`--stack-name`用于指定已部署 AWS CloudFormation 堆栈的名称。

必填：条件性。如果从已部署的 AWS CloudFormation 堆栈迁移，则为必填项。

可选选项

`--account` *STRING*

要从中检索 AWS CloudFormation 堆栈模板的账户。

必需：否

默认：从默认来源 AWS CDK CLI获取账户信息。

`--compress`

提供此选项可将生成的 CDK 项目压缩成ZIP文件。

必需：否

`--filter` *ARRAY*

用于从 AWS 账户迁移已部署的资源和 AWS 区域。此选项指定筛选器来确定要迁移哪些已部署的资源。

此选项接受键值对数组，其中 ke y 代表过滤器类型，值代表要过滤的值。

以下是可接受的密钥：

- `resource-identifier`— 资源的标识符。值可以是资源逻辑或物理 ID。例如，`resource-identifier="ClusterName"`。
- `resource-type-prefix`— AWS CloudFormation 资源类型前缀。例如，指定`resource-type-prefix="AWS::DynamoDB::"`筛选所有亚马逊 DynamoDB 资源。
- `tag-key`— 资源标签的密钥。例如，`tag-key="myTagKey"`。
- `tag-value`— 资源标签的值。例如，`tag-value="myTagValue"`。

为AND条件逻辑提供多个键值对。以下示例筛选了标记myTagKey为标签键的任何 DynamoDB 资源:. `--filter resource-type-prefix="AWS::DynamoDB::", tag-key="myTagKey"`

在单个命令中为OR条件逻辑多次提供该`--filter`选项。以下示例筛选任何属于 DynamoDB 资源或标记为标签密钥myTagKey的资源:. `--filter resource-type-prefix="AWS::DynamoDB::" --filter tag-key="myTagKey"`

必需：否

`--language` *STRING*

迁移期间创建的 CDK 项目使用的编程语言。

必需：否

可接受的值：typescriptpython、java、、csharp、go。

默认值：typescript

`--output-path` *PATH*

已迁移的 CDK 项目的输出路径。

必需：否

默认：默认情况下， AWS CDK CLI将使用您当前的工作目录。

`--region` *`STRING`*

AWS 区域 要从中检索 AWS CloudFormation 堆栈模板。

必需：否

默认：从默认来源 AWS CDK CLI获取 AWS 区域 信息。

AWS Visual Studio 代码工具包

Visual Studio Code 的AWS Toolk it 是 Visual Studio 代码的开源插件，它使创建、调试和部署应用程序变得更加容易 AWS。该工具包为开发 AWS CDK 应用程序提供了集成体验。它包括 AWS CDK 资源管理器功能，用于列出您的 AWS CDK 项目并浏览 CDK 应用程序的各个组件。安装 AWS 工具包并详细了解如何使用 AWS CDK 资源管理器。

AWS SAM 整合

AWS CDK 和 AWS Serverless Application Model (AWS SAM) 可以协同工作，让您在本地构建和测试 CDK 中定义的无服务器应用程序。有关完整信息，请参阅 AWS SAM 开发人员指南AWS Cloud Development Kit (AWS CDK)中的。要安装 SAM CLI，请参阅安装 AWS SAM CLI。

测试结构

使用后 AWS CDK，您的基础架构可以像您编写的任何其他代码一样具有可测试性。测试 AWS CDK 应用程序的标准方法使用断言模块和流行 AWS CDK的测试框架，例如 Jest fo JavaScript r and 和 Python 的 TypeScript Pyt est。

您可以为 AWS CDK 应用程序编写两类测试。

- 细粒度的断言测试生成 AWS CloudFormation 模板的特定方面，例如 “此资源具有此值的此属性”。这些测试可以检测回归。当你使用测试驱动开发来开发新功能时，它们也很有用。（你可以先写一个测试，然后通过写一个正确的实现来让它通过。）细粒度断言是最常用的测试。
- 快照测试根据先前存储的基线 AWS CloudFormation 模板测试合成后的模板。Snapshot 测试允许您自由重构，因为您可以确保重构后的代码与原始代码完全相同。如果这些更改是有意的，您可以接受新的基线，用于今后的测试。但是，CDK 升级也可能导致合成模板发生变化，因此您不能仅仅依靠快照来确保实现正确。

> **Note**
>
> 本主题中用作示例的 TypeScript、Python 和 Java 应用程序的完整版本可在上找到 GitHub。

开始使用

为了说明如何编写这些测试，我们将创建一个包含 AWS Step Functions 状态机和 AWS Lambda 函数的堆栈。Lambda 函数订阅了 Amazon SNS 主题，只需将消息转发到状态机即可。

首先，使用 CDK 工具包创建一个空的 CDK 应用程序项目并安装我们需要的库。我们将使用的构造都在 CDK 主包中，这是使用 CDK Toolkit 创建的项目中的默认依赖项。但是，您必须安装测试框架。

TypeScript

```
mkdir state-machine && cd state-machine
cdk init --language=typescript
npm install --save-dev jest @types/jest
```

为您的测试创建目录。

```
mkdir test
```

编辑项目package.json以告诉 NPM 如何运行 Jest，并告诉 Jest 要收集什么类型的文件。必要的更改如下。

- 向该分scripts区添加新test密钥
- 将 Jest 及其类型添加到该部分 devDependencies
- 添加带有moduleFileExtensions声明的新jest顶级密钥

这些更改显示在以下大纲中。将新文本放在中指示的位置package.json。“...” 占位符表示文件中不应更改的现有部分。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "@types/jest": "^24.0.18",
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

JavaScript

```
mkdir state-machine && cd state-machine
cdk init --language=javascript
npm install --save-dev jest
```

为您的测试创建目录。

```
mkdir test
```

编辑项目`package.json`以告诉 NPM 如何运行 Jest，并告诉 Jest 要收集什么类型的文件。必要的更改如下。

- 向该分`scripts`区添加新`test`密钥
- 将 Jest 添加到该部分 `devDependencies`
- 添加带有`moduleFileExtensions`声明的新`jest`顶级密钥

这些更改显示在以下大纲中。将新文本放在中指示的位置`package.json`。“...” 占位符表示文件中不应更改的现有部分。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

Python

```
mkdir state-machine && cd state-machine
cdk init --language=python
source .venv/bin/activate
python -m pip install -r requirements.txt
python -m pip install -r requirements-dev.txt
```

Java

```
mkdir state-machine && cd-state-machine
cdk init --language=java
```

在你首选的 Java IDE 中打开该项目。（在 Eclipse 中，使用 “文件” > “导入” > “现有 Maven 项目”。）

C#

```
mkdir state-machine && cd-state-machine
cdk init --language=csharp
```

在 Visual Studio 中打开 `src\StateMachine.sln`。

在 “解决方案资源管理器” 中右键单击该解决方案，然后选择 “添加” > “新建项目”。搜索 msteSt C# 并添加适用于 C# 的 msteSt 测试项目。（默认名称`TestProject1`没问题。）

右键单击`TestProject1`并选择 “添加” > “项目引用”，然后添加`StateMachine`项目作为参考。

示例堆栈

以下是将在本主题中测试的堆栈。如前所述，它包含一个 Lambda 函数和一个 Step Functions 状态机，并接受一个或多个亚马逊 SNS 主题。Lambda 函数订阅了 Amazon SNS 主题并将其转发到状态机。

您无需做任何特别的事情即可使该应用程序具有可测试性。实际上，这个 CDK 堆栈与本指南中的其他示例堆栈没有任何重要区别。

TypeScript

将以下代码放入`lib/state-machine-stack.ts`：

```
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import * as sns_subscriptions from "aws-cdk-lib/aws-sns-subscriptions";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as sfn from "aws-cdk-lib/aws-stepfunctions";
import { Construct } from "constructs";

export interface StateMachineStackProps extends cdk.StackProps {
  readonly topics: sns.Topic[];
}

export class StateMachineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: StateMachineStackProps) {
    super(scope, id, props);

    // In the future this state machine will do some work...
```

```
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}
```

JavaScript

将以下代码放入`lib/state-machine-stack.js`：

```
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const sns_subscriptions = require("aws-cdk-lib/aws-sns-subscriptions");
const lambda = require("aws-cdk-lib/aws-lambda");
const sfn = require("aws-cdk-lib/aws-stepfunctions");

class StateMachineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
```

```
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}

module.exports = { StateMachineStack }
```

Python

将以下代码放入`state_machine/state_machine_stack.py`：

```
from typing import List

import aws_cdk.aws_lambda as lambda_
import aws_cdk.aws_sns as sns
import aws_cdk.aws_sns_subscriptions as sns_subscriptions
import aws_cdk.aws_stepfunctions as sfn
import aws_cdk as cdk

class StateMachineStack(cdk.Stack):
    def __init__(
        self,
        scope: cdk.Construct,
        construct_id: str,
        *,
        topics: List[sns.Topic],
        **kwargs
    ) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # In the future this state machine will do some work...
        state_machine = sfn.StateMachine(
            self, "StateMachine", definition=sfn.Pass(self, "StartState")
        )
```

```
        # This Lambda function starts the state machine.
        func = lambda_.Function(
            self,
            "LambdaFunction",
            runtime=lambda_.Runtime.NODEJS_18_X,
            handler="handler",
            code=lambda_.Code.from_asset("./start-state-machine"),
            environment={
                "STATE_MACHINE_ARN": state_machine.state_machine_arn,
            },
        )
        state_machine.grant_start_execution(func)

        subscription = sns_subscriptions.LambdaSubscription(func)
        for topic in topics:
            topic.add_subscription(subscription)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.sns.ITopicSubscription;
import software.amazon.awscdk.services.sns.Topic;
import software.amazon.awscdk.services.sns.subscriptions.LambdaSubscription;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;

import java.util.Collections;
import java.util.List;

public class StateMachineStack extends Stack {
    public StateMachineStack(final Construct scope, final String id, final
 List<Topic> topics) {
        this(scope, id, null, topics);
    }
```

```
    public StateMachineStack(final Construct scope, final String id, final
 StackProps props, final List<Topic> topics) {
        super(scope, id, props);

        // In the future this state machine will do some work...
        final StateMachine stateMachine = StateMachine.Builder.create(this,
 "StateMachine")
                .definition(new Pass(this, "StartState"))
                .build();

        // This Lambda function starts the state machine.
        final Function func = Function.Builder.create(this, "LambdaFunction")
                .runtime(Runtime.NODEJS_18_X)
                .handler("handler")
                .code(Code.fromAsset("./start-state-machine"))
                .environment(Collections.singletonMap("STATE_MACHINE_ARN",
 stateMachine.getStateMachineArn()))
                .build();
        stateMachine.grantStartExecution(func);

        final ITopicSubscription subscription = new LambdaSubscription(func);
        for (final Topic topic : topics) {
            topic.addSubscription(subscription);
        }
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.AWS.SNS.Subscriptions;
using Constructs;

using System.Collections.Generic;

namespace AwsCdkAssertionSamples
{
    public class StateMachineStackProps : StackProps
    {
        public Topic[] Topics;
```

```
    }

    public class StateMachineStack : Stack
    {

        internal StateMachineStack(Construct scope, string id,
 StateMachineStackProps props = null) : base(scope, id, props)
        {
            // In the future this state machine will do some work...
            var stateMachine = new StateMachine(this, "StateMachine", new
 StateMachineProps
            {
                Definition = new Pass(this, "StartState")
            });

            // This Lambda function starts the state machine.
            var func = new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "handler",
                Code = Code.FromAsset("./start-state-machine"),
                Environment = new Dictionary<string, string>
                {
                    { "STATE_MACHINE_ARN", stateMachine.StateMachineArn }
                }
            });
            stateMachine.GrantStartExecution(func);

            foreach (Topic topic in props?.Topics ?? new Topic[0])
            {
                var subscription = new LambdaSubscription(func);
            }

        }
    }
}
```

我们将修改应用程序的主入口点，这样我们就不会实际实例化堆栈。我们不想意外部署它。我们的测试将创建一个应用程序和一个用于测试的堆栈实例。与测试驱动开发结合使用时，这是一种有用的策略：在启用部署之前，请确保堆栈通过所有测试。

TypeScript

In `bin/state-machine.ts`:

```
#!/usr/bin/env node
import * as cdk from "aws-cdk-lib";

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

JavaScript

In `bin/state-machine.js`:

```
#!/usr/bin/env node
const cdk = require("aws-cdk-lib");

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

Python

In `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk  as cdk

app = cdk.App()

# Stacks are intentionally not created here -- this application isn't meant to
# be deployed.

app.synth()
```

Java

```
package software.amazon.samples.awscdkassertionssamples;
```

```
import software.amazon.awscdk.App;

public class SampleApp {
    public static void main(final String[] args) {
        App app = new App();

        // Stacks are intentionally not created here -- this application isn't meant
 to be deployed.

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;

namespace AwsCdkAssertionSamples
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();

            // Stacks are intentionally not created here -- this application isn't
 meant to be deployed.

            app.Synth();
        }
    }
}
```

Lambda 函数

我们的示例堆栈包括一个启动状态机的 Lambda 函数。我们必须提供此函数的源代码，这样 CDK 才能将其作为创建 Lambda 函数资源的一部分进行捆绑和部署。

- 在应用程序的主目录`start-state-machine`中创建文件夹。

- 在此文件夹中，至少创建一个文件。例如，您可以将以下代码保存在中`start-state-machines/index.js`。

```
exports.handler = async function (event, context) {
 return 'hello world';
};
```

  但是，任何文件都可以使用，因为我们实际上不会部署堆栈。

运行测试

以下是用于在 AWS CDK 应用程序中运行测试的命令供参考。这些命令与您在使用相同测试框架的任何项目中运行测试时使用的命令相同。对于需要构建步骤的语言，请将其包括在内以确保您的测试已编译。

TypeScript

```
tsc && npm test
```

JavaScript

```
npm test
```

Python

```
python -m pytest
```

Java

```
mvn compile && mvn test
```

C#

生成您的解决方案 (F6) 以发现测试，然后运行测试（“测试” > “运行所有测试”）。要选择要运行的测试，请打开测试资源管理器（“测试” > “测试资源管理器”）。

或者：

```
dotnet test src
```

细粒度的断言

使用细粒度断言测试堆栈的第一步是合成堆栈，因为我们正在根据生成的模板编写断言。 AWS CloudFormation

我们StateMachineStackStack要求我们将 Amazon SNS 主题传递给状态机。因此，在我们的测试中，我们将创建一个单独的堆栈来包含该主题。

通常，在编写 CDK 应用程序时，可以在堆栈的构造函数中对 Amazon SNS 主题进行子类Stack化和实例化。在我们的测试中，我们Stack直接实例化，然后将此堆栈作为作用域传递，将其附加到堆栈中。Topic这在功能上是等效的，而且不那么冗长。它还有助于使仅用于测试的堆栈与您打算部署的堆栈 “看起来不同”。

TypeScript

```
import { Capture, Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import { StateMachineStack } from "../lib/state-machine-stack";

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const stateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
```

```
}
```

JavaScript

```
const { Capture, Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const { StateMachineStack } = require("../lib/state-machine-stack");

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const StateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
```

Python

```
from aws_cdk import aws_sns as sns
import aws_cdk as cdk
from aws_cdk.assertions import Template

from app.state_machine_stack import StateMachineStack

def test_synthesizes_properly():
    app = cdk.App()

    # Since the StateMachineStack consumes resources from a separate stack
    # (cross-stack references), we create a stack for our SNS topics to live
```

```
    # in here. These topics can then be passed to the StateMachineStack later,
    # creating a cross-stack reference.
    topics_stack = cdk.Stack(app, "TopicsStack")

    # Create the topic the stack we're testing will reference.
    topics = [sns.Topic(topics_stack, "Topic1")]

    # Create the StateMachineStack.
    state_machine_stack = StateMachineStack(
        app, "StateMachineStack", topics=topics  # Cross-stack reference
    )

    # Prepare the stack for assertions.
    template = Template.from_stack(state_machine_stack)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import software.amazon.awscdk.assertions.Capture;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.services.sns.Topic;

import java.util.*;

import static org.assertj.core.api.Assertions.assertThat;

public class StateMachineStackTest {
    @Test
    public void testSynthesizesProperly() {
        final App app = new App();

        // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
        // for our SNS topics to live in here. These topics can then be passed to
 the StateMachineStack later, creating a
        // cross-stack reference.
        final Stack topicsStack = new Stack(app, "TopicsStack");
```

```
        // Create the topic the stack we're testing will reference.
        final List<Topic> topics =
 Collections.singletonList(Topic.Builder.create(topicsStack, "Topic1").build());

        // Create the StateMachineStack.
        final StateMachineStack stateMachineStack = new StateMachineStack(
                app,
                "StateMachineStack",
                topics // Cross-stack reference
        );

        // Prepare the stack for assertions.
        final Template template = Template.fromStack(stateMachineStack)
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest
    {
        [TestMethod]
        public void TestMethod1()
        {
            var app = new App();

            // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
            // for our SNS topics to live in here. These topics can then be passed
 to the StateMachineStack later, creating a
            // cross-stack reference.
            var topicsStack = new Stack(app, "TopicsStack");
```

```
            // Create the topic the stack we're testing will reference.
            var topics = new Topic[] { new Topic(topicsStack, "Topic1") };

            // Create the StateMachineStack.
            var StateMachineStack = new StateMachineStack(app, "StateMachineStack",
 new StateMachineStackProps
            {
                Topics = topics
            });

            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // test will go here
        }
    }
}
```

现在我们可以断言 Lambda 函数和亚马逊 SNS 订阅已创建。

TypeScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

JavaScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

Python

```
# Assert that we have created the function with the correct properties
    template.has_resource_properties(
        "AWS::Lambda::Function",
        {
            "Handler": "handler",
            "Runtime": "nodejs14.x",
        },
    )

    # Assert that we have created a subscription
    template.resource_count_is("AWS::SNS::Subscription", 1)
```

Java

```
        // Assert it creates the function with the correct properties...
        template.hasResourceProperties("AWS::Lambda::Function", Map.of(
                "Handler", "handler",
                "Runtime", "nodejs14.x"
        ));

         // Creates the subscription...
        template.resourceCountIs("AWS::SNS::Subscription", 1);
```

C#

```
            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // Assert it creates the function with the correct properties...
            template.HasResourceProperties("AWS::Lambda::Function", new StringDict {
                { "Handler", "handler"},
                { "Runtime", "nodejs14x" }
            });

            // Creates the subscription...
            template.ResourceCountIs("AWS::SNS::Subscription", 1);
```

我们的 Lambda 函数测试断言函数资源的两个特定属性具有特定的值。默认情况下，该`hasResourceProperties`方法对合成 CloudFormation 模板中给出的资源属性执行部分匹配。此测试要求所提供的属性存在并具有指定的值，但资源也可以具有其他未经过测试的属性。

我们的 Amazon SNS 断言合成模板包含订阅，但没有包含订阅本身。我们加入这个断言主要是为了说明如何断言资源数量。该`Template`类提供了更具体的方法来针对 CloudFormation 模板的`Resources`、`Outputs`和`Mapping`部分编写断言。

匹配器

的默认部分匹配行为`hasResourceProperties`可以使用`Match`类中的匹配器进行更改。

匹配器的范围从宽松 (`Match.anyValue`) 到严格 ()。`Match.objectEquals`它们可以嵌套以将不同的匹配方法应用于资源属性的不同部分。例如，`Match.anyValue`结合使用`Match.objectEquals`和，我们可以更全面地测试状态机的 IAM 角色，同时不需要为可能更改的属性设置特定值。

TypeScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
  "AWS::IAM::Role",
  Match.objectEquals({
    AssumeRolePolicyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "sts:AssumeRole",
          Effect: "Allow",
          Principal: {
            Service: {
              "Fn::Join": [
                "",
                ["states.", Match.anyValue(), ".amazonaws.com"],
              ],
            },
          },
        },
      ],
    },
  })
);
```

JavaScript

```
    // Fully assert on the state machine's IAM role with matchers.
    template.hasResourceProperties(
      "AWS::IAM::Role",
      Match.objectEquals({
        AssumeRolePolicyDocument: {
          Version: "2012-10-17",
          Statement: [
            {
              Action: "sts:AssumeRole",
              Effect: "Allow",
              Principal: {
                Service: {
                  "Fn::Join": [
                    "",
                    ["states.", Match.anyValue(), ".amazonaws.com"],
                  ],
                },
              },
            },
          ],
        },
      })
    );
```

Python

```
from aws_cdk.assertions import Match

    # Fully assert on the state machine's IAM role with matchers.
    template.has_resource_properties(
        "AWS::IAM::Role",
        Match.object_equals(
            {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Action": "sts:AssumeRole",
                            "Effect": "Allow",
                            "Principal": {
                                "Service": {
```

```
                                "Fn::Join": [
                                    "",
                                    [
                                        "states.",
                                        Match.any_value(),
                                        ".amazonaws.com",
                                    ],
                                ],
                            },
                        },
                    },
                ],
            },
        }
    ),
)
```

Java

```
        // Fully assert on the state machine's IAM role with matchers.
        template.hasResourceProperties("AWS::IAM::Role", Match.objectEquals(
                Collections.singletonMap("AssumeRolePolicyDocument", Map.of(
                        "Version", "2012-10-17",
                        "Statement", Collections.singletonList(Map.of(
                                "Action", "sts:AssumeRole",
                                "Effect", "Allow",
                                "Principal", Collections.singletonMap(
                                        "Service", Collections.singletonMap(
                                                "Fn::Join", Arrays.asList(
                                                        "",
                                                        Arrays.asList("states.",
Match.anyValue(), ".amazonaws.com")
                                                )
                                        )
                                )
                        ))
                ))
        ));
```

C#

```
            // Fully assert on the state machine's IAM role with matchers.
            template.HasResource("AWS::IAM::Role", Match.ObjectEquals(new ObjectDict
```

```
            {
                { "AssumeRolePolicyDocument", new ObjectDict
                    {
                        { "Version", "2012-10-17" },
                        { "Action", "sts:AssumeRole" },
                        { "Principal", new ObjectDict
                            {
                                { "Version", "2012-10-17" },
                                { "Statement", new object[]
                                    {
                                        new ObjectDict {
                                            { "Action", "sts:AssumeRole" },
                                            { "Effect", "Allow" },
                                            { "Principal", new ObjectDict
                                                {
                                                    { "Service", new ObjectDict
                                                        {
                                                            { "", new object[]
                                                                { "states",
Match.AnyValue(), ".amazonaws.com" }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }));
```

许多 CloudFormation 资源都包含以字符串形式表示的序列化 JSON 对象。`Match.serializedJson()`匹配器可用于匹配此 JSON 中的属性。

例如，Step Functions 状态机是使用基于 JSON 的 A ma zon States 语言中的字符串定义的。我们将使用`Match.serializedJson()`来确保我们的初始状态是唯一的步骤。同样，我们将使用嵌套匹配器将不同类型的匹配应用于对象的不同部分。

TypeScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
    })
  ),
});
```

JavaScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
    })
```

```
      ),
    });
```

Python

```
    # Assert on the state machine's definition with the serialized_json matcher.
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                # Match.object_equals() is the default, but specify it here for
clarity
                Match.object_equals(
                    {
                        "StartAt": "StartState",
                        "States": {
                            "StartState": {
                                "Type": "Pass",
                                "End": True,
                                # Make sure this state doesn't provide a next state
--
                                # we can't provide both Next and set End to true.
                                "Next": Match.absent(),
                            },
                        },
                    }
                )
            ),
        },
    )
```

Java

```
        // Assert on the state machine's definition with the Match.serializedJson()
matcher.
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                        Match.objectEquals(Map.of(
                                "StartAt", "StartState",
                                "States", Collections.singletonMap(
```

```
                                        "StartState", Map.of(
                                                "Type", "Pass",
                                                "End", true,
                                                // Make sure this state doesn't
provide a next state -- we can't provide
                                                // both Next and set End to true.
                                                "Next", Match.absent()
                                        )
                                )
                        ))
                )
        ));
```

C#

```
            // Assert on the state machine's definition with the
Match.serializedJson() matcher
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                    Match.ObjectEquals(new ObjectDict {
                        { "StartAt", "StartState" },
                        { "States", new ObjectDict
                        {
                            { "StartState", new ObjectDict {
                                { "Type", "Pass" },
                                { "End", "True" },
                                // Make sure this state doesn't provide a next state
-- we can't provide
                                // both Next and set End to true.
                                { "Next", Match.Absent() }
                            }}
                        }}
                    })
                )}});
```

捕获

测试属性以确保它们遵循特定的格式，或者与其他属性具有相同的值，而无需事先知道它们的确切值，这通常很有用。该assertions模块在其Capture类中提供了此功能。

通过指定一个Capture实例来代替中的值hasResourceProperties，该值将保留在Capture对象中。可以使用对象as的方法（包括、和）检索实际捕获的值 asNumber() asString()asObject，并进行测试。与匹配器Capture一起使用可指定要捕获的值在资源属性（包括序列化的 JSON 属性）中的确切位置。

以下示例进行测试，以确保状态机的起始状态名称以开头Start。它还会测试该状态是否存在于计算机的状态列表中。

TypeScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
      StartAt: startAtCapture,
      States: statesCapture,
    })
  ),
});

// Assert that the start state starts with "Start".
expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the
// state machine definition.
expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

JavaScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
```

```
          StartAt: startAtCapture,
          States: statesCapture,
        })
      ),
    });

    // Assert that the start state starts with "Start".
    expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

    // Assert that the start state actually exists in the states object of the
    // state machine definition.
    expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

Python

```
import re

    from aws_cdk.assertions import Capture

    # ...

    # Capture some data from the state machine's definition.
    start_at_capture = Capture()
    states_capture = Capture()
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                Match.object_like(
                    {
                        "StartAt": start_at_capture,
                        "States": states_capture,
                    }
                )
            ),
        },
    )

    # Assert that the start state starts with "Start".
    assert re.match("^Start", start_at_capture.as_string())

    # Assert that the start state actually exists in the states object of the
    # state machine definition.
```

```
    assert start_at_capture.as_string() in states_capture.as_object()
```

Java

```
        // Capture some data from the state machine's definition.
        final Capture startAtCapture = new Capture();
        final Capture statesCapture = new Capture();
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        Match.objectLike(Map.of(
                                "StartAt", startAtCapture,
                                "States", statesCapture
                        ))
                )
        ));

        // Assert that the start state starts with "Start".
        assertThat(startAtCapture.asString()).matches("^Start.+");

        // Assert that the start state actually exists in the states object of the
state machine definition.
        assertThat(statesCapture.asObject()).containsKey(startAtCapture.asString());
```

C#

```
            // Capture some data from the state machine's definition.
            var startAtCapture = new Capture();
            var statesCapture = new Capture();
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    new ObjectDict
                    {
                        { "StartAt", startAtCapture },
                        { "States", statesCapture }
                    }
                )}
            });

            Assert.IsTrue(startAtCapture.ToString().StartsWith("Start"));
```

```
Assert.IsTrue(statesCapture.AsObject().ContainsKey(startAtCapture.ToString()));
```

快照测试

在快照测试中，您可以将整个合成 CloudFormation 模板与先前存储的基线（通常称为 “主模板”）模板进行比较。与细粒度的断言不同，快照测试在捕捉回归时没有用。这是因为快照测试适用于整个模板，而除了代码更改之外的事情可能会导致合成结果出现微小（或 not-so-small）差异。这些更改甚至可能不会影响您的部署，但它们仍会导致快照测试失败。

例如，您可以更新 CDK 结构以纳入新的最佳实践，这可能会导致综合资源或其组织方式发生变化。或者，您可以将 CDK 工具包更新为报告其他元数据的版本。对上下文值的更改也会影响合成后的模板。

但是，只要你保持所有其他可能影响合成模板的因素不变，快照测试对重构有很大帮助。如果您所做的更改无意中更改了模板，您将立即知道。如果更改是故意的，只需接受新模板作为基准即可。

例如，如果我们有这样的`DeadLetterQueue`构造：

TypeScript

```
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}
```

JavaScript

```
class DeadLetterQueue extends sqs.Queue {

  constructor(scope, id) {
```

```
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}

module.exports = { DeadLetterQueue }
```

Python

```
class DeadLetterQueue(sqs.Queue):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        self.messages_in_queue_alarm = cloudwatch.Alarm(
            self,
            "Alarm",
            alarm_description="There are messages in the Dead Letter Queue.",
            evaluation_periods=1,
            threshold=1,
            metric=self.metric_approximate_number_of_messages_visible(),
        )
```

Java

```
public class DeadLetterQueue extends Queue {
    private final IAlarm messagesInQueueAlarm;

    public DeadLetterQueue(@NotNull Construct scope, @NotNull String id) {
        super(scope, id);

        this.messagesInQueueAlarm = Alarm.Builder.create(this, "Alarm")
                .alarmDescription("There are messages in the Dead Letter Queue.")
                .evaluationPeriods(1)
                .threshold(1)
                .metric(this.metricApproximateNumberOfMessagesVisible())
                .build();
```

```
    }

    public IAlarm getMessagesInQueueAlarm() {
        return messagesInQueueAlarm;
    }
}
```

C#

```
namespace AwsCdkAssertionSamples
{
    public class DeadLetterQueue : Queue
    {
        public IAlarm messagesInQueueAlarm;

        public DeadLetterQueue(Construct scope, string id) : base(scope, id)
        {
            messagesInQueueAlarm = new Alarm(this, "Alarm", new AlarmProps
            {
                AlarmDescription = "There are messages in the Dead Letter Queue.",
                EvaluationPeriods = 1,
                Threshold = 1,
                Metric = this.MetricApproximateNumberOfMessagesVisible()
            });
        }
    }
}
```

我们可以这样测试它：

TypeScript

```
import { Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import { DeadLetterQueue } from "../lib/dead-letter-queue";

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
```

```
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

JavaScript

```
const { Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const { DeadLetterQueue } = require("../lib/dead-letter-queue");

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

Python

```
import aws_cdk_lib as cdk
from aws_cdk_lib.assertions import Match, Template

from app.dead_letter_queue import DeadLetterQueue

def snapshot_test():
    stack = cdk.Stack()
    DeadLetterQueue(stack, "DeadLetterQueue")

    template = Template.from_stack(stack)
    assert template.to_json() == snapshot
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import au.com.origin.snapshots.Expect;
import software.amazon.awscdk.assertions.Match;
```

```
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.Stack;

import java.util.Collections;
import java.util.Map;

public class DeadLetterQueueTest {
    @Test
    public void snapshotTest() {
        final Stack stack = new Stack();
        new DeadLetterQueue(stack, "DeadLetterQueue");

        final Template template = Template.fromStack(stack);
        expect.toMatchSnapshot(template.toJSON());
    }
}
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest

    [TestClass]
    public class DeadLetterQueueTest
    {
    [TestMethod]
        public void SnapshotTest()
        {
            var stack = new Stack();
            new DeadLetterQueue(stack, "DeadLetterQueue");

            var template = Template.FromStack(stack);
```

```
            return Verifier.Verify(template.ToJSON());
        }
    }
}
```

测试小贴士

请记住，您的测试将与它们测试的代码一样长，并且它们将被读取和修改的频率相同。因此，花点时间考虑如何最好地写出它们是值得的。

不要复制和粘贴设置行或常见断言。取而代之的是，将此逻辑重构为夹具或辅助函数。使用能反映每项测试实际测试内容的好名字。

不要试图在一次测试中做太多事情。最好是，测试应该只测试一种行为。如果你不小心破坏了这种行为，那么只有一个测试会失败，测试的名称应该会告诉你失败了什么。但是，这更是一个值得努力的理想；有时你会不可避免地（或无意中）编写测试多个行为的测试。出于我们已经描述的原因，快照测试特别容易出现此问题，因此请谨慎使用它们。

安全性 AWS Cloud Development Kit (AWS CDK)

云安全性一直是 Amazon Web Services (AWS) 的重中之重。作为 AWS 客户，您可以受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。安全是双方 AWS 的共同责任。责任共担模式将其描述为云的安全性和云中的安全性。

云安全 — AWS 负责保护运行 AWS 云中提供的所有服务的基础架构，并为您提供可以安全使用的服务。我们的安全责任是重中之重 AWS，作为AWS 合规计划的一部分，第三方审计师定期测试和验证我们安全的有效性。

云端安全 — 您的责任由您使用的 AWS 服务以及其他因素决定，包括数据的敏感性、组织的要求以及适用的法律和法规。

通过其支持的特定 Amazon Web Services (AWS) 服务， AWS CDK 遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

主题

的身份和访问管理 AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) AWS 服务 可帮助管理员安全地控制对 AWS 资源的访问权限。IAM 管理员控制谁可以进行身份验证（登录）和授权（拥有权限）使用 AWS 资源。您可以使用 IAM AWS 服务 ，无需支付额外费用。

受众

您的使用方式 AWS Identity and Access Management (IAM) 会有所不同，具体取决于您所做的工作 AWS。

服务用户-如果您 AWS 服务 曾经完成工作，则您的管理员会为您提供所需的凭证和权限。当你使用更多 AWS 功能来完成工作时，你可能需要额外的权限。了解如何管理访问权限有助于您向管理员请求适合的权限。

服务管理员-如果您负责公司的 AWS 资源，则可能拥有对 AWS 资源的完全访问权限。您的工作是确定您的服务用户应该访问哪些资源 AWS 服务 和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。

IAM 管理员：如果您是 IAM 管理员，您可能希望了解如何编写策略以管理对 AWS 服务的访问权限的详细信息。

使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份或通过担 AWS 账户根用户任 IAM 角色进行身份验证（登录 AWS）。

您可以使用通过身份源提供的凭据以 AWS 联合身份登录。 AWS IAM Identity Center （IAM Identity Center）用户、贵公司的单点登录身份验证以及您的 Google 或 Facebook 凭据就是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当你使用联合访问 AWS 时，你就是在间接扮演一个角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录的更多信息 AWS，请参阅《AWS 登录 用户指南》中的如何登录到您 AWS 账户的。

要 AWS 以编程方式访问，请 AWS 提供软件开发套件 (SDK) 和命令行接口 (CLI)，以便使用您的凭据对请求进行加密签名。 AWS CDK如果您不使用 AWS 工具，则必须自己签署请求。有关使用推荐的方法自行对请求签名的更多信息，请参阅《AWS 一般参考》中的 签名版本 4 签名流程。

无论使用何种身份验证方法，您可能需要提供其他安全信息。例如， AWS 建议您使用多重身份验证 (MFA) 来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的 多重身份验证 和《IAM 用户指南》中的 在 AWS中使用多重身份验证（MFA）。

AWS 账户 root 用户

创建时 AWS 账户，首先要有一个登录身份，该身份可以完全访问账户中的所有资源 AWS 服务 和资源。此身份被称为 AWS 账户 root 用户，使用您创建帐户时使用的电子邮件地址和密码登录即可访问该身份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关要求您以根用户身份登录的任务的完整列表，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，要求人类用户（包括需要管理员访问权限的用户）使用与身份提供商的联合身份验证 AWS 服务 通过临时证书进行访问。

联合身份是指您的企业用户目录、Web 身份提供商、Identity C enter 目录中的用户，或者任何使用 AWS 服务 通过身份源提供的凭据进行访问的用户。 AWS Directory Service当联合身份访问时 AWS 账户，他们将扮演角色，角色提供临时证书。

要集中管理访问权限，建议您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中创建用户和群组，也可以连接并同步到您自己的身份源中的一组用户和群组，以便在您的所有 AWS 账户 和应用程序中使用。有关 IAM Identity Center 的信息，请参阅《AWS IAM Identity Center 用户指南》中的什么是 IAM Identity Center？。

IAM 用户和群组

I AM 用户是您 AWS 账户 内部对个人或应用程序具有特定权限的身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，我们建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的对于需要长期凭证的使用场景定期轮换访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅 IAM 用户指南中的何时创建 IAM 用户（而不是角色）。

IAM 角色

I AM 角色是您内部具有特定权限 AWS 账户 的身份。它类似于 IAM 用户，但与特定人员没有关联。您可以 AWS Management Console 通过切换角色在中临时担任 IAM 角色。您可以通过调用 AWS CLI 或 AWS API 操作或使用自定义 URL 来代入角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色。

具有临时凭证的 IAM 角色在以下情况下很有用：

- 联合用户访问 – 要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关联合身份验证的角色的信息，请参阅《IAM 用户指南》中的 为第三方身份提供商创建角色。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。
- 临时 IAM 用户权限 – IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。

- 跨账户存取 – 您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些资源 AWS 服务，您可以将策略直接附加到资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。
- 跨服务访问 — 有些 AWS 服务 使用其他 AWS 服务服务中的功能。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon EC2 中运行应用程序或在 Amazon S3 中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 服务角色——服务角色是服务代表您在账户中执行操作而代入的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务委派权限的角色。
  - 服务相关角色-服务相关角色是一种链接到的服务角色。 AWS 服务服务可以代入代表您执行操作的角色。服务相关角色出现在您的中 AWS 账户 ，并且归服务所有。IAM 管理员可以查看但不能编辑服务相关角色的权限。
- 在 A@@ mazon EC2 上运行的应用程序 — 您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时证书。这优先于在 EC2 实例中存储访问密钥。要向 EC2 实例分配 AWS 角色并使其可供其所有应用程序使用，您需要创建附加到该实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色为 Amazon EC2 实例上运行的应用程序授予权限。

要了解是使用 IAM 角色还是 IAM 用户，请参阅 IAM 用户指南中的何时创建 IAM 角色（而不是用户）。

的合规性验证 AWS Cloud Development Kit (AWS CDK)

通过其支持的特定 Amazon Web Services (AWS) 服务， AWS CDK 遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

AWS 服务的安全性与合规性由第三方审计机构作为多项 AWS 合规计划的一部分进行评估。其中包括 SOC、PCI、FedRAMP、HIPAA 等。 AWS 在 “按合规计划划分的范围内的 AWS 服务” 中提供了经常更新的特定合规计划范围内的AWS 服务列表。

您可以使用 Artif AWS act 下载第三方审计报告。有关更多信息，请参阅中的下载报告 AWS Artifact。

有关 AWS 合规计划的更多信息，请参阅AWS 合规计划。

您在使用 AWS CDK 访问 AWS 服务时的合规责任取决于数据的敏感性、组织的合规目标以及适用的法律和法规。如果您在使用 AWS 服务时必须符合 HIPAA、PCI 或 FedRAMP 等标准，请提供资源来帮助： AWS

- 安全与合规性快速入门指南 — 部署指南，讨论架构注意事项，并提供在上部署以安全为重点和以合规为重点的基准环境的步骤。 AWS
- AWS 合规资源 — 可能适用于您所在行业和所在地区的工作手册和指南的集合。
- AWS Config – 评估资源配置对内部实践、行业指南和法规的遵循情况的服务。
- AWS Security Hub— 全面了解您的安全状态 AWS ，可帮助您检查自己是否符合安全行业标准和最佳实践。

的韧性 AWS Cloud Development Kit (AWS CDK)

Amazon Web Services (AWS) 全球基础设施是围绕 AWS 区域和可用区构建的。

AWS 区域提供多个物理隔离和隔离的可用区，这些可用区通过低延迟、高吞吐量和高度冗余的网络相连。

利用可用区，您可以设计和操作在可用区之间无中断地自动实现故障转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅AWS 全球基础设施。

通过其支持的特定 Amazon Web Services (AWS) 服务， AWS CDK 遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

基础设施安全 AWS Cloud Development Kit (AWS CDK)

通过其支持的特定 Amazon Web Services (AWS) 服务， AWS CDK 遵循分担责任模式。有关 AWS 服务安全信息，请参阅AWS 服务安全文档页面和合规计划合 AWS 规工作范围内的AWS 服务。

常见 AWS CDK 问题疑难解答

本主题介绍如何对以下问题进行故障排除 AWS CDK。

更新后 AWS CDK， AWS CDK 工具包 (CLI) 报告与 AWS 构造库不匹配

AWS CDK 工具包（提供cdk命令）的版本必须至少等于主 AWS 构造库模块的版本aws-cdk-lib。该工具包旨在向后兼容。该工具包的最新 2.x 版本可以与该库的任何 1.x 或 2.x 版本一起使用。因此，我们建议您在全球范围内安装此组件并使其保持最新状态。

```
npm update -g aws-cdk
```

如果您需要使用该 AWS CDK 工具包的多个版本，请在您的项目文件夹中本地安装该工具包的特定版本。

如果您使用的是 TypeScript 或 JavaScript，则您的项目目录中已经包含 CDK Toolkit 的版本化本地副本。

如果您使用的是其他语言，请使用npm安装 AWS CDK Toolkit，省略该-g标志并指定所需的版本。例如：

```
npm install aws-cdk@2.0
```

要运行本地安装的 AWS CDK Toolkit，请使用命令npx aws-cdk而不是仅使用命令cdk。例如：

```
npx aws-cdk deploy MyStack
```

`npx aws-cdk`运行 AWS CDK 工具包的本地版本（如果存在）。当项目没有本地安装时，它会回退到全局版本。你可能会发现设置一个 shell 别名`cdk`很方便，以确保总是以这种方式调用。

macOS/Linux

```
alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

（返回列表）

部署我的 AWS CDK 堆栈时，我收到一个**`NoSuchBucket`**错误

您的 AWS 环境尚未引导，因此没有用于在部署期间存放资源的 Amazon S3 存储桶。您可以使用以下命令创建暂存存储桶和其他必需的资源：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

为避免产生意外 AWS 费用， AWS CDK 不会自动引导任何环境。您必须显式引导要部署的每个环境。

默认情况下，引导资源是在当前 AWS CDK 应用程序中的堆栈使用的一个或多个区域中创建的。或者，它们是在您的本地 AWS 个人资料（设置为`aws configure`）中指定的区域中使用该个人资料的帐户创建的。您可以在命令行上指定不同的账户和区域，如下所示。（如果您不在应用程序目录中，则必须指定账户和区域。）

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

有关更多信息，请参阅 the section called “正在引导”。

（返回列表）

部署我的 AWS CDK 堆栈时，我收到一条**`forbidden: null`**消息

您部署的堆栈需要引导资源，但使用的 IAM 角色或账户没有写入权限。（部署包含资产或合成大于 50K 的 AWS CloudFormation 模板的堆栈时，会使用暂存存储桶。）使用有权`s3:*`对错误消息中提及的存储桶执行操作的账户或角色。

（返回列表）

合成 AWS CDK 堆栈时，我会收到消息 **--app is required either in command-line, in cdk.json or in ~/.cdk.json**

这条消息通常意味着当你发布问题时，你不在 AWS CDK 项目的主目录中`cdk synth`。此目录`cdk.json`中的文件由`cdk init`命令创建，包含运行（从而合成）AWS CDK 应用程序所需的命令行。例如，对于一个 TypeScript 应用程序，默认值是`cdk.json`这样的：

```
{
  "app": "npx ts-node bin/my-cdk-app.ts"
}
```

我们建议仅在项目的主目录中发出`cdk`命令，这样 AWS CDK 工具包就可以在`cdk.json`那里找到并成功运行您的应用程序。

如果由于某种原因这不切实际， AWS CDK Toolkit 会在另外两个位置查找应用程序的命令行：

- `cdk.json`在你的主目录中
- 在`cdk synth`命令本身上使用`-a`选项

例如，您可以按如下方式合成来自 TypeScript 应用程序的堆栈。

```
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

（返回列表）

合成 AWS CDK 堆栈时，我收到错误消息，因为 AWS CloudFormation 模板包含的资源太多

AWS CDK 生成并部署 AWS CloudFormation 模板。 AWS CloudFormation 对堆栈可以包含的资源数量有硬性限制。有了这个 AWS CDK，你可以比预期的更快地突破这个限制。

> **Note**
>
> 在撰写本文时， AWS CloudFormation 资源限制为 500。有关当前资源限制，请参阅AWS CloudFormation 配额。

C AWS onstruct Library 的更高级别、基于意图的构造会自动配置日志、密钥管理、授权和其他目的所需的任何辅助资源。例如，向一个资源授予另一个资源的访问权限会生成相关服务进行通信所需的任何 IAM 对象。

根据我们的经验，在现实世界中使用基于意图的构造会导致每个构造需要 1-5 个 AWS CloudFormation 资源，尽管情况可能有所不同。对于无服务器应用程序，通常每个 API 端点有 5-8 个 AWS 资源。

模式代表更高的抽象级别，允许你用更少的代码定义更多的 AWS 资源。例如the section called “ECS”，中的 AWS CDK 代码生成了 50 多个 AWS CloudFormation 资源，而只定义了三个结构！

超过 AWS CloudFormation 资源限制是 AWS CloudFormation 合成过程中的错误。如果您的堆栈超过限制的 80%，则会发出警告。 AWS CDK 您可以通过在堆栈上设置`maxResources`属性来使用不同的限制，也可以通过设置为 0 `maxResources` 来禁用验证。

> **Tip**
>
> 您可以使用以下实用程序脚本获得合成输出中资源的精确计数。（由于每个 AWS CDK 开发者都需要 Node.js，因此脚本是用编写的 JavaScript。）

```
// rescount.js - count the resources defined in a stack
// invoke with: node rescount.js <path-to-stack-json>
// e.g. node rescount.js cdk.out/MyStack.template.json

import * as fs from 'fs';
const path = process.argv[2];

if (path) fs.readFile(path, 'utf8', function(err, contents) {
  console.log(err ? `${err}` :
  `${Object.keys(JSON.parse(contents).Resources).length} resources defined in
 ${path}`);
}); else console.log("Please specify the path to the stack's output .json
 file");
```

当堆栈的资源数量接近限制时，可以考虑重新架构以减少堆栈包含的资源数量：例如，通过组合一些 Lambda 函数，或者将堆栈分成多个堆栈。CDK 支持堆栈之间的引用，因此你可以用任何对你来说最有意义的方式将应用程序的功能分成不同的堆栈。

> **Note**
>
> AWS CloudFormation 专家们经常建议使用嵌套堆栈作为资源限制的解决方案。通过`NestedStack`构造 AWS CDK 支持这种方法。

（返回列表）

我为我的 Auto Scaling 组或 VPC 指定了三个（或更多）可用区，但它只部署在两个可用区中

要获取您请求的可用区数量，请在堆栈的`env`属性中指定账户和区域。如果未同时指定两者，则默认情况下 AWS CDK，会将堆栈合成为与环境无关的堆栈。然后，您可以使用将堆栈部署到特定区域 AWS CloudFormation。由于某些区域只有两个可用区，因此与环境无关的模板使用的可用区不超过两个。

> **Note**
>
> 过去，区域偶尔会推出只有一个可用区。与环境无关的 AWS CDK 堆栈无法部署到此类区域。但是，在撰写本文时，所有 AWS 区域都至少有两个可用区。

您可以通过覆盖堆栈的`availablilityZones`(Python:`availability_zones`) 属性来显式指定要使用的区域，从而更改此行为。

有关在合成时指定堆栈的账户和区域，同时保留部署到任何区域的灵活性的更多信息，请参阅the section called “环境”。

（返回列表）

我的 S3 存储桶、DynamoDB 表或其他资源在我发布时没有被删除 **`cdk destroy`**

默认情况下，可以包含用户数据的资源的`removalPolicy` (Python:`removal_policy`) 属性为`RETAIN`，并且在堆栈被销毁时不会删除该资源。相反，资源是堆栈中的孤立资源。然后，您必须在堆栈销毁后手动删除资源。在你这样做之前，重新部署堆栈会失败。这是因为部署期间创建的新资源的名称与孤立资源的名称冲突。

如果您将资源的移除策略设置为`DESTROY`，则该资源将在堆栈被销毁时被删除。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
```

```
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

```
software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.*;
import software.constructs;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY
    });
}
```

> **Note**
>
> AWS CloudFormation 无法删除非空的 Amazon S3 存储桶。如果您将 Amazon S3 存储桶的移除策略设置为DESTROY，并且该存储桶包含数据，则尝试销毁该堆栈将失败，因为该存储桶无法删除。您可以通过将存储桶的 `autoDeleteObjects` prop 设置为，在尝试销毁存储桶之前 AWS CDK 删除存储桶中的对象`true`。

（返回列表）

和 jsii 的 OpenPGP 密钥 AWS CDK

本主题包含和 jsii 的当前和历史 OpenPGP 密钥 AWS CDK 。

当前密钥

应使用这些密钥来验证 AWS CDK 和 jsii 的当前版本。

AWS CDK OpenPGP 密钥

密钥 ID：	0x42b9cf2286cd987A
类型：	RSA
尺码:	4096/4096
已创建：	2022-07-05
过期：	2026-07-04
用户标识：	AWS Cloud 开发套件 Kit < aws-cdk@amazon.com >
钥匙指纹：	69B5 2D5B A295 1D11 FA65 413B 42B9 CF22 86CD 987A 987A

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGLEgOsBEADCoAMwvnszMLybJ+AD9cHhVyX6+rYIUEXYSgVnfkl6Z7qawIwv
wgd/a5fEs9Kiz2XJmfwS9Rxb4d+0+Y1ls1A+gnpw9FMLcZlqkC9KLnS2MqvuXWLB
t3z4kjZaL9fQ+58PoD4gy/M2hDg6gZrYqR3gtJuw8FcFpb/1KlkzRQUM8eAMFxf2
TyfjP0V0tSHwcB+84oushX7fUXVMyc3+OHsCPOe/WBFMIlWgKA+n33JKIQlUUC8f
kCWBAsAFupil0lCveT6mZu5slNRlc1I3iBLjUZ3/MtLygfqAMKwUVXeawtDvRIZe
PrAFc2NyODEhly2JG6K0FW7eIcvBqR3rg8U49t9Y74ELTM0kKnfd+flvq35xWqQC
0zghnk3kDppRTN4zWBgTKiCMxBcsHXGOoGn57t4B9VY9Zy3vkeySigeiwl/Tw9nJ
PE0SRnwEc/HnjTTfX+GTG1aQVE0xSVyZ4m5ymRNCu6+rNH8lKwo5FujlXJ+GXPkp
```

```
qT+Lx6Ix/Ny7PaoweWxwtZUkLRS4pWUsg0yotZrGyIbS+X3yMEG8WBTFI9hf6HTq
0ryfi5/TsBrdrGKqWB99EC9xYEGgtHp4fKO5X0ynOagVOhf0jSe8t1uyuJPGb2Gc
MQagSys5xMhdG/ZnEY4Cb+JDtH/4jc3tca0+4Z5RQ7kF9IhCncFtrbjJbwARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJixIDrAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
F4AACgkQQrnPIobNmHo2qg//Zt9p/kN1DevflzxWKouUX0AS7UmUtRYXu5k/EEbu
wkYNHpUr7+lZ+Me5YyjcIpt6UwuG9cW4SvwuxIfXucyKAWiwEbydCQauvnrYDxDa
J6Yr/ntk7Sii6An9re99qic3IsvX+xlUXh+qJ/34ooP/1PHziCMqykvW/DwAIyhx
2qvTXy+9+OlOWSUbhkCnNz5XKb4XQGq73DqalZX1nH4dG6fckZmYRX+dpw2njfTw
ZLdZ7bkrfiL84FI4A21RfSbEU4s4ngiV17lZ9ivilBKTbDv3da7+yc919M7C5N4J
yrlxvtyYNDoqKAD2WYZAnpEbG/shu3f56RyOJd56tXGwl9nKPh+F9y+379XthSwA
xZTURFtjWf7wWHaDZadU0DKi+Oeeszjg2f/VJaGmmS8PIg7q6GiSHHpqHqNvACHm
ZXMw12QFd3qt3xu0JMmE11ZC5VBgblwpkQTrO04Sq1rOpJwXI9ODMS/ZEhAIoYmT
OR7ouknlAx6mj9fwpavWDAAJHLdVUMYBZTXiQYFzDvx51ivvTRWkB1zTJcFdqShY
B37+Jz2jLDNdMrcHk2yfVp/VvfbxKcexg8wEwrrtQUslTUenl5jBZJouoz/wW81s
Y4U1nCPCdTK5/C7JCKzR2gVnCpe6uaxAWkkM2feQhjqJZkTC4cFVgBT+4M6WcT1r
yq4=
=ahbs
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP 密钥

密钥 ID：	0x056c4e15dae3d8d9
类型：	RSA
尺码:	4096/4096
已创建：	2022-07-05
过期：	2026-07-04
用户标识：	AWS JSII 团队 < aws-jsii@amazon.com >
钥匙指纹：	1E07 31D4 57E5 FE87 87E5 530A 056C 4E15 DAE3 D8D9

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGLEgOkBEAD27EPVG9g2mHQ3+M6tF6le+tfhARJ2EV7m7NKIrTdSlCZATLWn
AVLlxG1unW34NlkKZbcbR86gAxRnnAhuEhPuloU/S5wAqPGbRiFl58YjYZDNJw6U
1SSMpE4O1sfjxv9yAbiRihLYtvksyHHZmaDhYner2aK1PdeWu+BKq/tjfm3Yzsd2
uuVEduJ72YoQk/29dEiGOHfT+2kUKxUX+0tJSJ9MGlEf4NtQE4WLzrT6Xqb2SG4+
alIiIVxIEi0XKDn7n8ZLjFwfJwOYxVYLtEUkqFWM8e8vgoc9/nYc+vDXZVED2g3Z
FWrwSnDSXbQpnMa2cLhD4xLpDHUS3i2p7r3dkJQGLo/5JGOopLibrOAbYZ72izhu
H/TuPFogSz0mNFPglrWdnLF04UIjIq420+06V4WQZC9n55Zjcbki/OhnC3B9pAdU
tiy8zg070bWq45dPGf5STkPPn7G8A2zmKefyO51iLIi26ZzW78siB+FvcGRhdg25
39sHJ1cmrTeC+B+k4KeV5sQ/m3UucimrZnk1xdaiVp8mWzRqWb8bB6Rs8K9RMrMV
tFBOKOBAT2QxOQtRGAantVgm193E1T1cmNpD0FKAKkDdPs64rKBEwFiHxccXHbah
eMd1weVwn3AKFD6uAm8ZRMV+dyssfcQxqpo/kfT1XpA6cQeOmGDOcKBfdwARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
YsSA6QIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEAVsThXa
49jZjU4QANoyqOJUT4gRrXshE3N0mW5Ad4i8Ke09GA62HyvTtfbsA+2nkNVGJpXm
sFMzdaFO95Q65RkLS9vW4nhhjXBEc2XYNCt2AnARudA/41ykjDPwU112z9ZTB9he
y4ItIeNGpHvMWr51fihl0y2nkpODOBeiv44jscLbHyOmZfki1f5fuIu2U2IbUGK3
5FtYyeHcgRHnpYkzLuzK4PfayOywqQPJ7M9DWrHf+v5Cu4ZCZDOIKfzF+ew7MWwc
6KaoWHCYbFpX8jxFppbGsSFOQ8Sl2quoP0TLz9Wsq70Khi6C2P8JI6lm0HRLO+1M
jFbQxNOwAcN3k4HSwunAjXBlmT/6oc1RsdBdpXBaZ2AWseIXwSYZqNXp+5L179uZ
vSiD3DSSUqLJbdQRVOsJi3/87V5QU59byq2dToHveRjtSbVnK0TkTx9Zlgkcpjv M
BwHNqWhratV6af2Upjq2YQ0fdSB42f3pgopInxNJPMvlAb+cCfr0Pfwu7ge7UooQ
WHTxbpCvwtn/HNctMGpWscO02WsWgoYVjnVFay/XphE77pQ9rRUkhMe6VKXfxj/n
OCZJKrydluIIwR8vvONNqO+QwZ1xDEhO7MaSZlOm1AuUZIXFPgaWQkPZHKiiwFA/
QWnL/+shuRtMH2geTjkev198Jgb5HyXFm4SyYtZferQROyliEhik
=BuGv
-----END PGP PUBLIC KEY BLOCK-----
```

历史密钥

这些密钥可用于在 2022-07-05 之前验证 AWS CDK 和 jsii 的版本。

> **Important**
>
> 新密钥是在以前的密钥过期之前创建的。因此，在任何给定时刻，可能有多个密钥有效。从工件创建之日起，密钥就用于对其进行签名，因此在密钥的有效性重叠的地方使用最近发布的密钥。

AWS CDK OpenPGP 密钥 (2022-04-07)

> **Note**
>
> 2022-07-05 之后，此密钥未被用于签署 AWS CDK 文物。

密钥 ID：	0x015584281f44a3c3
类型：	RSA
尺码:	4096/4096
已创建：	2022-04-07
过期：	2026-04-06
用户标识：	AWS Cloud 开发套件 Kit < aws-cdk@amazon.com >
钥匙指纹：	EAE1 1A24 82B0 AA86 456E 6C67 0155 8428 1F44 A3C3

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGJPLgUBEADtlR5jQtxtBmROQvmWlPOViqqnJNhk0dULc3tXnq8NS/l6X81r
wHk+/CHG5kBunwvM0qaqLFRC6z9NnnNDxEHcTi47n+OAjWyDM6unxxWOPz8Dfaps
Uq/ZWa4by292ZeqRC9Ir2wdrizb69JbRjeshBwlJDAS/qtqCAqBRH/f7Zw7QSD6/
XTxyIy+KOVjZwFPFNHMRQ/NmgUc/Rfxsa0pUjk1YAj/AkvQlwwD8DEnASoBhO0DP
QonZxouLqIpgp4LsGo8TZdQv30ocIj0C9DuYUiUXWlCPlYPgDj6IWf3rgpMQ6nB9
wC9lx4t/L3Zg1HUD52y8aymndmbdHVn90mzlNg4XWyc58rioYrEk57YwbDnea/Kk
Hv4kVHZRfJ4/OFPyqs5ex1X3X6rb07VvA1tfLgPywO9XF2Xws8YWOWcEobaWTcnb
AzyVC6wKya8rEQzxkYJ6UkJlhDB6g6bZwIpsI2zlimG+kSBsyFvE2oRYMS0cXPqU
o+tX0+4TvxEyW3RrUQzQHIpqXrb0X1Q8Z2idPn5dwsipDEa4gsFXtrSXmbB/0Cee
eJVvKWQAsxol3+NE9L/yozq3cz5PWh0SSbmCLRcs78lMJ23MmzbMWV7BWC9DXdY+
TywY5IkDUPjGCKlD8VlrI3TgC222bH6qaua6LYCiTtRtvpDYuJNAlUjhawARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJiTy4FAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
```

```
F4AACgkQAVWEKB9Eo8NpbxAAiBF0kR/lVw3vuam60mk4l0iGMVsP8Xq6g/buzbEO
2MEB4Ftk04qOnoa+93S0ZiLR9PqxrwsGSp4ADDX3Vtc4uxwzUlKUi1ywEhQ1cwyL
YHQI3Hd75K1J81ozMEu6qJH+yF0TtTDZMeZHtH/XvuIYJW3Lx4o5ZFlsEegFPAgX
YCCpUS+k9qC6M8g2VjcltQJpyjGswsKm6FWaKHW+B9dfjdOHlImB9E2jaknJ8eoY
zb9zHgFANluMzpZ6rYVSiCuXiEgYmazQWCvlPcMOP7nX+1hq1z11LMqeSnfE09gX
H+OYho9cMEJkb1dzx1H9MRpylFIn9tL+2iCp4UPJjnqi6uawWyLZ2tp4G11haqQq
1yAh69u233I8GZKFUySzjHwH5qWGRgBTjrZ6FdcjSS2w/wMkVKuCPkWtdvo/TJrm
msCd1Reye8SEKYqrs0ujTwmlvWmUZmO06AdUjo1kWiBKeslTJrWEuG7Yk4pFOoA4
dsaq83gxpOJNVCh6M3y4DLNrvl7dhF95NwTWMROPj2otw7NIjF4/cdzve2+P7YNN
pVAtyCtTJdD3eZbQPVaL3T8cf1VGqt6++pnLGnWJ0+X3TyvfmTohdJvN3TE+tq7A
7cprDX/q9c56HaXdJzVpxEzuf/YC+JuYKeHwsX3QouDhyRg3PsigdZES/02Wr8so
l6U=
=MQI4
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP 密钥 (2022-04-07)

> **Note**
>
> 在 2022-07-05 之后，此密钥未被用来签署 jsii 工件。

密钥 ID：	0x985f5bc974b79356
类型：	RSA
尺码:	4096/4096
已创建：	2022-04-07
过期：	2026-04-06
用户标识：	AWS JSII 团队 < aws-jsii@amazon.com >
钥匙指纹：	35A7 1785 8FA6 282D C5AC CD95 985F 5BC9 74B7 9356

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGJPLewBEADHH4TXup/gOlHrKDZRbj8MvsMTdM6eDteA6/c32UYV/YsK9rDA
jN8Jv/xlfosOebcHrfnFpHF9VTkmjuOpN695XdwMrW/NvlEPISTGEJf21x6ZTQ2r
1xWFYzC3sl3FZmvj9XAXTmygdv+XM3TqsFgZeCaBkZVdiLbQf+FhYrovUlgotb5D
YiCQI3ofV5QTE+141jhO5Pkd3ZIoBG+P826LaT8NXhwS0o1XqVk39DCZNoFshNmR
WFZpkVCTHyv5ZhVey1NWXnD8opO375htGNV4AeSmSIH9YkURD1g5F+2t7RiosKFo
kJrfPmUjhHn8IFpReGc8qmMMZX0WaV3t+VAWfOHGGyrXDfQ4xz1VCot75C2+qypM
+qhwOAOOP0zA7CfI96ULZzSH/j8HuQk3O0DsUCybpMuKEazEMxP3tgGtRerwDaFG
jQvAlK8Rbq3v8buBI6YJuXTwSzJE8KLjleUiTFumE6WP4rsAvlP/5rBvubeMfa3n
NIMm5Rkl36Z+jt3e2Z2ZqWDPpBRta8m7QHccrZhkvqu3YC3Gl6kdnm4Vio3Xfpg2
qtWhIQutQ6DmItewV+weQHas3hl88RPJtSrfWWIIMkpbF7Y4vbX9xcnsYCLlp2Mz
tWbbnU+EWATNSsufml/Kdnu9iEEuLmeovE11I69nwjNOq9P+GJ3r/FUb2wARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
Yk8t7AIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEJhfW8l0
t5NWo64P/2y7gcMRylLLW/wbrCjton2O4+YRocwQxKm1cBml9FVDUR5967YczNuu
EwEOfH/Pu3UAlrBfKAfxPNhKchLwYiOBNh2Wk5UUxRcldNHTLb5jn5gxCeWNAsl/
Tc46qY+ObdBMdOf2Vu33UCOg83WLbg1bfBoA8Bm1cd0XObtLGucu606EBt1dBrKq
9UTcbJfuGivY2Xjy5r4kEiMHBoLKcFrSo2Mm7VtYlE4Mabjyj9+orqUio7qxOl60
aa7Psa6rMvs1Ip9IOrAdG7o5Y29tQpeINH0R1/u47BrlTEAgG63Dfy49w2h/1g0G
c9KPXVuN55OWRIu0hsiySDMk/2ERsF348TU3NURZltnCOxp6pHlbPJIxRVTNa9Cn
f8tbLB3y3HfA80516g+qwNYIYiqksDdV2bz+VbvmCWcO+FellDZli831gyMGa5JJ
rq7d0lEr6nqjcnKiVwItTQXyFYmKTAXweQtVC72g1sd3oZIyqa7T8pvhWpKXxoJV
WP+OPBhGg/JEVC9sguhuv53tzVwayrNwb54JxJsD2nemfhQm1Wyvb2bPTEaJ3mrv
mhPUvXZj/I9rgsEq3L/sm2Xjy09nra4o3oe3bhEL8nOj11wkIodi17VaGP0y+H3s
I5zB5UztS6dy+cH+J7DoRaxzVzq7qtH/ZY2quClt30wwqDHUX1ef
=+iYX
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK OpenPGP 密钥 (2018-06-19)

密钥 ID：	0x0566a784e17f3870
类型：	RSA
尺码:	4096/4096
已创建：	2018-06-19
过期：	2022-06-18
用户标识：	AWS CDK 团队 < aws-cdk@amazon.com >

钥匙指纹：	E88B E3B6 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFsovE8BEADEFVCHeAVPvoQgsjVu9FPUczxy9P+2zGIT/MLI3/vPLiULQwRy
IN2oxyBNDtcDToNa/fTkW3Ev0NTP4V1h+uBoKDZD/p+dTmSDRfByECMI0sGZ3UsG
Ohhyl2Of44s0sL8gdLtDnqSRLf+ZrfT3gpgUnplW7VitkwLxr78jDpW4QD8p8dZ9
WNm3JgB55jyPgaJKqA1Ln4Vduni/1XkrG42nxrrU71uUdZPvPZ2ELLJa6n0/raG8
jq3le+xQh45gAIs6PGaAgy7jAsfbwkGTBHjjujITAY1DwvQH5iS31OaCM9n4JNpc
xGZeJAVYTLilznf2QtS/a50t+ZOmpq67Ssp2j6qYpiumm0Lo9q3K/R4/yF0FZ8SL
1TuNX0ecXEptiMVUfTiqrLsANg18EPtLZZOYW+ZkbcVytkDpiqj7bMwA7mI7zGCJ
1gjaTbcEmOmVdQYS1G6ZptwbTtvrgA6AfnZxX1HUxLRQ7tT/wvRtABfbQKAh85Ff
a3U9W4oC3c1MP5IyhNV1Wo8Zm0flZiZc0iZnojTtSG6UbcxNNL4Q8e08FWjhungj
yxSsIBnQO1Aeo1N4BbzlI+n9iaXVDUN7Kz1QEyS4PNpjvUyrUiQ+a9C5sRA7WP+x
IEOaBBGpoAXB3oLsdTNO6AcwcDd9+r2NlXlhWC4/uH2YHQUIegPqHmPWxwARAQAB
tCFBV1MgQ0RLIFRlYW0gPGF3cy1jZGtAYW1hem9uLmNvbT6JAj8EEwEIACkFAlso
vE8CGy8FCQeEzgAHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRAFZqeE4X84
cLGxD/0XHnhoR2xvz38GM8HQlwlZy9W1wVhQKmNDQUavw8Zx7+iRR3m7nq3xM7Qq
BDbcbKSg1lVLSBQ6H2V6vRpysOhkPSH1nN2dO8DtvSKIPcxK48+1x7lmO+ksSs/+
oo1UvOmTDaRzOitYh3kOGXHHXk/l11GtF2FGQzYssX5iM4PHcjBsK1unThs56IMh
OJeZezEYzBaskTu/ytRJ236bPP2kZIEXfzAvhmTytuXWUXEftxOxc6fIAcYiKTha
aofG7WyR+Fvb1j5gNLcbY552QMxa23NZd5cSZH7468WEW1SGJ3AdLA7k5xvsPPOC
2YvQFD+vUOZ1JJuu6B5rHkiEMhRTLklkvqXEShTxuXiCp7iTOo6TBCmrWAT4eQr7
htLmqlXrgKi8qPkWmRdXXG+MQBzI/UyZq2q8KC6cx2md1PhANmeefhiM7FZZfeNM
WLonWfh8gVCsNH5h8WJ9fxsQCADd3Xxx3NelS2zDYBPRoaqZEEBbgUP6LnWFprA2
EkSlc/RoDqZCpBGgcoy1FFWvV/ZLgNU6OTQlYH6oYOWiylSJnaTDyurrktsxJI6d
4gdsFb6tqwTGecuUPvvZaEuvhWExLxAebhu780FdAPXgVTX+YCLI2zf+dWQvkFQf
80RE7ayn7BsiaLzFBVux/zz/WgvudsZX18r8tDiVQBL51ORmqw==
=0wuQ
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP 密钥 (2018-08-06)

密钥 ID：	0x1c7ace4cb2a1b93a
类型：	RSA

尺码:	4096/4096
已创建：	2018-08-06
过期：	2022-08-05
用户标识：	AWS JSII 团队 < aws-jsii@amazon.com >
钥匙指纹：	85EF 6522 4CE2 1E8C 72DB 28EC 1C7A CE4C B2A1 B93A B2A1 B93A

选择 “复制” 图标以复制以下 OpenPGP 密钥：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFtoSs0BEAD6WweLD0B26h0F7Jo9iR6tVQ4PgQBK1Va5H/eP+A2Iqw79UyxZ
WNzHYhzQ5MjYYI1SgcPavXy5/LV1N8HJ7QzyKszybnLYpNTLPYArWE8ZM9ZmjvIR
p1GzwnVBGQfoOlxyeutE9T5ZkAn45dTS5jlno4unji4gHjnwXKf2nP1APU2CZfdK
8vDpLOgj9LeeGlerYNbx+7xtY/I+csFIQvK09FPLSNMJQLlkBhY0r6Rt9ZQG+653
tJn+AUjyM237w0UIX1IqyYc5IONXu8HklPGu0NYuX9AY/63Ak2Cyfj0w/PZlvueQ
noQNM3j0nkOEsTOEXCyaLQw9iBKpxvLnm5RjMSODDCkj8c9uu0LHr7J4EOtgt2S1
pem7Y/c/N+/Z+Ksg9fP8fVTfYwRPvdI1x2sCiRDfLoQSG9tdrN5VwPFi4sGV04sI
x7Al8Vf/OBjAGZrDaJgM/gVvb9SKAQUA6t3ofeP14gDrS0eYodEXZ+lamnxFglxF
Sn8NRC4JFNmkXSUaTNGUdFf//F0D69PRNT8CnFfmniGj0CphN5037PCA2LC/Buq2
3+K6mTPkCcCHYPC/SwItp/xIDAQsGuDc1i1SfDYXrjsK7uOuwC5jLA9X6wZ/jgXQ
4umRRJBAV1aW8b1+yfaYYCO2AfXXO6caObv8IvH7Pc4leC2DoqylD3KklQARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQgAKQUC
W2hKzQIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEBx6zkyy
obk6B34P/iNb5QjKyhT0glZiq1wK7tuDDRpR6fC/sp6Jd/GhaNjO4BzlDbUPSjW5
950VT+qwaHXbIma/QVP7EIRztfwWy7m8eOodjpiu7JyJprhwG9nocXiNsLADcMoH
BvabkDRWXWIWSurq2wbcFMlTVwxjHPIQs6kt2oojpzP985CDS/KTzyjow6/gfMim
DLdhSSbDUM34STEgew79L2sQzL7cvM/N59k+AGyEMHZDXHkEw/Bge5Ovz50YOnsp
lisH4BzPRIw7uWqPlkVPzJKwMuo2WvMjDfgbYLbyjfvs5mqDxT2GTwAx/rd2taU6
iSqP0QmLM54BtTVVdoVXZSmJyTmXAAGlITq8ECZ/coUW9K2pUSgVuWyu63lktFP6
MyCQYRmXPh9aSd4+ielteXM9Y39snlyLgEJBhMxioZXVO2oszwluPuhPoAp4ekwj
/umVsBf6As6PoAchg7Qzr+lRZGmV9YTJOgDn2Z7jf/7tOes0g/mdiXTQMSGtp/Fp
ggnifTBx3iXkrQhqHlwtam8XTHGHy3MvX17ZslNuB8Pjh+07hhCxv0VUVZPUHJqJ
ZsLa398LMteQ8UMxwJ3t06jwDWAd7mbr2tatIilLHtWWBFoCwBh1XLe/03ENCpDp
njZ7OsBsBK2nVVcN0H2v5ey0T1yE93o6r7xOwCwBiVp5skTCRUob
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK 开发者指南历史记录

有关版本的信息，请参阅 AWS CDK 版本。大约每周更新一次。 AWS CDK 维护版本可能会在每周发布之间发布，以解决关键问题。每个版本都包含匹配的 AWS CDK 工具包 (CDK CLI)、 AWS 构造库和 API 参考。本指南的更新通常与 AWS CDK 版本不同步。

> **Note**
>
> 下表列出了重要的文档里程碑。我们会持续修复错误并改进内容。

变更	说明	日期
添加 CDK 迁移功能的文档	使用 AWS CDK CLI`cdk migrate`命令将已部署的 AWS 资源、已部署的 AWS CloudFormation 堆栈和本地 AWS CloudFormation 模板迁移到。 AWS CDK有关更多信息，请参阅迁移到 AWS CDK。	2024 年 2 月 2 日
IAM 最佳实践更新	更新了指南，使其符合 IAM 最佳实践。有关更多信息，请参阅 IAM 安全最佳实践。	2023 年 3 月 23 日
文档 `cdk.json`	添加`cdk.json`配置值文档。	2022 年 4 月 20 日
依赖关系管理	添加有关使用管理依赖关系的主题 AWS CDK。	2022 年 4 月 7 日
从 Java 示例中删除双大括号	将此反模式全部替换为 Java 9 `Map.of`。	2022 年 3 月 9 日
AWS CDK v2 版本	《 AWS CDK 开发者指南》第 2 版已发布。CDK v1 的@@文档历史记录。	2021 年 12 月 4 日

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。