aws

Real-Time Streaming User Guide

Amazon IVS

Amazon IVS: Real-Time Streaming User Guide

Table of Contents

什么是 Amazon IVS 实时流式传输？

Amazon Interactive Video Service（IVS）实时流式传输为您提供将实时音频和视频添加到应用程序所需的一切支持。

强度：

- 实时延迟：构建适用于延迟敏感型用例的应用程序，帮助您的观众与 IVS 实时流式传输保持连接状态并进行互动。实时流式传输内容从主机传送到观众的延迟不到 300 毫秒。
- 高并发：通过 IVS 实时流式传输打造大规模交互式视频体验。最多可向 10,000 名观众传送实时流式传输，支持多达 12 台主机控制虚拟舞台。
- 移动平台优化：IVS 实时流式传输针对移动平台用例进行了优化，可满足各种设备和网络功能。通过集成适用于 Android 和 iOS 的 Amazon IVS 广播 SDK，您的用户可以作为主机或观众进行互动，在移动平台上获得高质量的实时流式传输体验。

使用案例：

- 嘉宾席位：创建允许主机“登上舞台”邀请嘉宾的应用程序，将观众转变为主机参与实时互动。
- 对战（VS）模式：通过并行竞赛提供体验，并允许观众实时观看主机之间的比赛。
- 音频聊天室：邀请听众作为嘉宾参与对话，在音频聊天室中展开更深入的互动交流。
- 实时视频带货：让带货成为交互式视频活动，并通过实时延迟还原视频的清晰度和完整性。

除此处的产品文档外，请参阅 https://ivs.rocks/，此网站专用于浏览已发布内容（演示、代码示例、博客文章）、估算成本并通过现场演示体验 Amazon IVS。

全球解决方案、区域控制

流传输和查看面向全球

您可以使用 Amazon IVS 向全球的查看者进行流传输：

- 当您进行流传输时，Amazon IVS 会自动在您附近的位置提取视频。
- 观众可以在全球范围内观看您的实时流。

换一种说法是，“数据层面”是全球性的。数据层面是指流传输/提取和查看。

控制分区域进行

虽然 Amazon IVS 数据层面是全球性的，但“控制层面”是区域性的。控制面板是指 Amazon IVS 控制台、API 和资源（舞台）。

换句话说，Amazon IVS 是一种“区域性 AWS 服务”。即每个区域中的 Amazon IVS 资源都独立于其他区域中的类似资源。例如，您在一个区域中创建的舞台与您在其他区域中创建的舞台无关。

当您使用资源（例如，创建舞台）时，必须指定创建资源的区域。随后，当您管理资源时，您必须从创建资源的同一区域执行此操作。

如果您使用的是...	您可以通过以下方式指定区域...
Amazon IVS 控制台	使用导航栏右上角的 Select a Region (选择区域) 下拉菜单。
Amazon IVS API	使用合适的服务终端节点。请参阅 Amazon IVS Real-Time Streaming API Reference。https://docs.aws.amazon.com/ivs/latest/RealTimeAPIReference/Welcome.html （如果您通过 SDK 访问 API，请设置 SDK 的 `region` 参数。请参阅用于在 AWS 上进行构建的工具。）
AWS CLI	或者： • 将 `--region <aws-region>` 附加到您的 CLI 命令。 • 将区域放入本地 AWS 配置文件中。

请记住，无论在哪个区域创建舞台，您都可以从任何地方流式传输到 Amazon IVS，从而便于观众从任何地方观看。

IVS 实时流式传输入门

本文档将引导您完成将 Amazon IVS 实时流式传输集成到应用程序所涉及的步骤。

主题

简介

先决条件

首次使用实时流式传输之前，请完成以下任务。有关说明，请参阅 Getting Started with IVS Low-Latency Streaming。

- 创建 AWS 账户
- 设置根用户和管理用户

其他参考资料

- IVS Web Broadcast SDK Reference
- IVS Android Broadcast SDK Reference
- IVS iOS Broadcast SDK Reference
- IVS Real-Time Streaming API Reference

实时流式传输术语

租期	描述
舞台	一个虚拟空间，参与者可以在其中实时交换视频。
Host	向舞台发送本地视频的参与者。
查看者	接收主机视频的参与者。
参与者	以主机或观众身份连接到舞台的用户。
参与者令牌	参与者加入舞台时对其进行身份验证的令牌。
广播 SDK	允许参与者发送和接收视频的客户端库。

步骤概述

1. the section called “设置 IAM 权限” – 创建 AWS Identity and Access Management（IAM）policy，以授予用户基本权限，并将此策略分配给用户。
2. 创建舞台 – 创建一个虚拟空间，参与者可以在其中实时交换视频。
3. 分发参与者令牌 – 向参与者发送令牌，以便他们可以加入舞台。
4. 集成 IVS 广播 SDK – 将广播 SDK 添加到应用程序，以便参与者能够发送和接收视频：the section called “Web”、the section called “Android” 和 the section called “iOS”。
5. 发布和订阅视频 – 将您的视频发送到舞台并接收来自其他主机的视频：the section called “Web”、the section called “Android” 和 the section called “iOS”。

设置 IAM 权限

接下来，您必须创建 Amazon Identity and Access Management（IAM）policy，以授予用户一组基本权限（例如，创建 Amazon IVS 舞台和创建参与者令牌），并将该策略分配给用户。您可以在创建新用户时分配权限，也可以向现有用户添加权限。这两项程序如下。

有关更多信息（例如，了解 IAM 用户和策略、如何将策略附加到用户以及如何限制用户可以使用 Amazon IVS 执行的操作），请参见：

- IAM 用户指南中的创建 IAM 用户
- Amazon IVS 安全性中关于 IAM 和“IVS 的托管式策略”的信息。
- Amazon IVS 安全性中的 IAM 信息

您可以对 Amazon IVS 使用现有的 AWS 托管式策略，也可以创建新策略，该策略可自定义想要授予一组用户、组或角色的权限。下面介绍了这两种方法。

使用 IVS 权限的现有策略

在大多数情况下，您需要对 Amazon IVS 使用 AWS 托管式策略。IVS 安全性的 IVS 的托管式策略部分对其进行了全面描述。

- 使用 `IVSReadOnlyAccess` AWS 托管式策略，您的应用程序开发人员可以访问所有 IVS Get 和 List API 端点（低延迟和实时直播均适用）。
- 使用 `IVSFullAccess` AWS 托管式策略，您的应用程序开发人员可以访问所有 IVS API 端点（低延迟和实时直播均适用）。

可选：为 Amazon IVS 权限创建自定义策略

按照以下步骤进行操作：

1. 登录 Amazon 管理控制台，并通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 在导航窗格中，选择策略，然后选择创建策略。这时将会打开指定权限窗口。
3. 在指定权限窗口中，选择 JSON 选项卡，然后复制下列 IVS 策略并粘贴到策略编辑器文本区域。[该策略不包括所有 Amazon IVS 操作。您可以根据需要添加/删除（允许/拒绝）端点访问权限。有关 IVS 端点的详细信息，请参阅 IVS Real-Time Streaming API Reference。]

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "ivs:CreateStage",
            "ivs:CreateParticipantToken",
            "ivs:GetStage",
            "ivs:GetStageSession",
            "ivs:ListStages",
            "ivs:ListStageSessions",
            "ivs:CreateEncoderConfiguration",
            "ivs:GetEncoderConfiguration",
            "ivs:ListEncoderConfigurations",
            "ivs:GetComposition",
            "ivs:ListCompositions",
            "ivs:StartComposition",
            "ivs:StopComposition"
          ],
          "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "cloudwatch:DescribeAlarms",
            "cloudwatch:GetMetricData",
            "s3:DeleteBucketPolicy",
            "s3:GetBucketLocation",
            "s3:GetBucketPolicy",
            "s3:PutBucketPolicy",
            "servicequotas:ListAWSDefaultServiceQuotas",
            "servicequotas:ListRequestedServiceQuotaChangeHistoryByQuota",
            "servicequotas:ListServiceQuotas",
            "servicequotas:ListServices",
            "servicequotas:ListTagsForResource"
         ],
         "Resource": "*"
      }
   ]
}
```

4. 继续在指定权限窗口中，选择下一步（滚动到窗口底部即可看到此按钮）。这时将打开检查并创建窗口。
5. 在检查并创建窗口中，输入策略名称，此外还可以选择添加描述。记下策略名称，因为您在创建用户时需要使用该名称（如下文所述）。选择 Create policy（创建策略）（位于窗口底部）。
6. 您将返回到 IAM 控制台窗口，您应该会在该窗口中看到一条横幅，确认您的新策略已创建。

创建新用户并添加权限

IAM 用户访问密钥

IAM 访问密钥包含一个访问密钥 ID 和一个秘密访问密钥。它们用于对您向 Amazon 发出的编程请求进行签名。如果没有访问密钥，您可以从 Amazon 管理控制台创建。作为最佳实践，请勿创建根用户访问密钥。

仅当创建访问密钥时，您才能查看或下载秘密访问密钥。以后您无法恢复它们。但是，您随时可以创建新的访问密钥；您必须拥有执行所需 IAM 操作的权限。

请务必安全地存储访问密钥。切勿与第三方共享（即使查询似乎来自 Amazon）。有关更多信息，请参阅《 IAM 用户指南》中的管理 IAM 用户的访问密钥。

过程

按照以下步骤进行操作：

1. 在导航窗格中，选择用户，然后选择创建用户。这时将会打开指定用户详细信息窗口。
2. 在指定用户详细信息窗口中：
   a. 在用户详细信息下，键入要创建的新用户名。
   b. 选中授予用户访问亚马逊云科技管理控制台的权限。
   c. 在控制台密码下，选择自动生成的密码。
   d. 选中用户下次登录时必须修改密码旁的复选框。
   e. 请选择 Next（下一步）。这时将会打开设置权限窗口。
3. 在设置权限下，选择直接附加策略。这时将会打开权限策略窗口。
4. 在搜索框中，输入 IVS 策略名称（AWS 托管式策略或您之前创建的自定义策略）。找到该策略后，选中复选框以选择该策略。
5. 选择下一步（位于窗口底部）。这时将打开检查并创建窗口。

6. 在检查并创建窗口中，确认所有用户详细信息均正确无误，然后选择创建用户（位于窗口底部）。
7. 这时将会打开找回密码窗口，其中包含您的控制台登录详细信息。妥善保存好此信息，以备将来参考。完成后，选择返回用户列表。

向现有用户添加权限

按照以下步骤进行操作：

1. 登录 Amazon 管理控制台，并通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 在导航窗格中，选择用户，然后选择要更新的现有用户名。（单击选择用户名；不要选中选择框。）
3. 在摘要页面的权限选项卡中，选择添加权限。这时将会打开添加权限窗口。
4. 选择 Attach existing policies directly（直接附加现有策略）。这时将会打开权限策略窗口。
5. 在搜索框中，输入 IVS 策略名称（AWS 托管式策略或您之前创建的自定义策略）。找到该策略后，选中复选框以选择该策略。
6. 选择下一步（位于窗口底部）。这时将会打开检查窗口。
7. 在检查窗口中，选择添加权限（位于窗口底部）。
8. 在 Summary（摘要）页面上，确认已添加 IVS 策略。

创建舞台

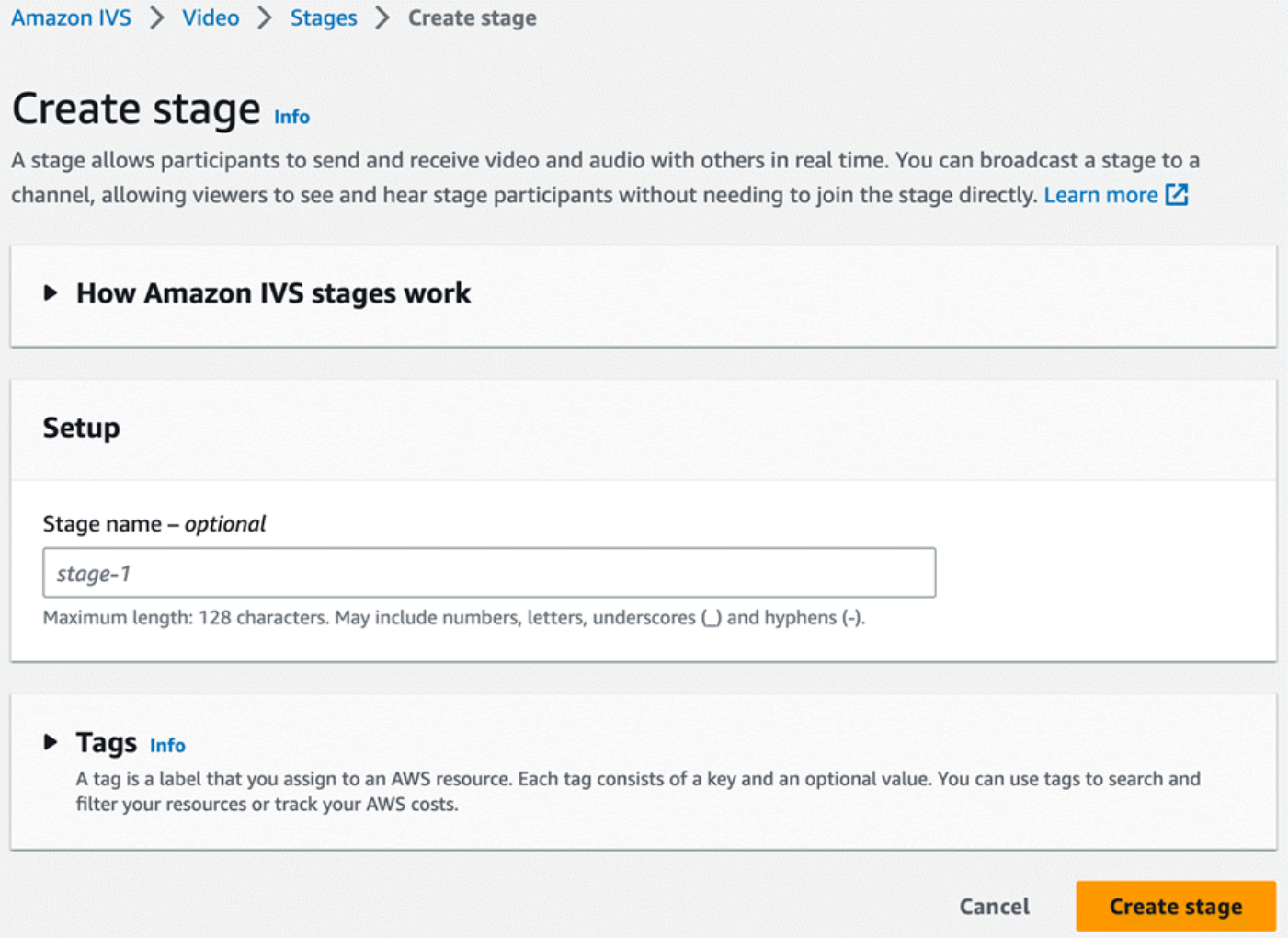
舞台是一个虚拟空间，参与者可以在其中实时交换视频。它是实时流式传输 API 的基础资源。您可以使用控制台或 CreateStage 终端节点创建阶段。

我们建议尽可能为每个逻辑会话创建一个新舞台，使用后将其删除，而不是保留旧舞台以备可能的重复使用。如果不清理过时资源（不可重复使用的旧舞台），您很可能会更快地达到最大舞台数量的限制。

控制台说明

1. 打开 Amazon IVS 控制台。

   （您还可通过亚马逊云科技管理控制台访问 Amazon IVS 控制台。）
2. 请在左侧导航窗格中选择舞台，然后选择创建舞台。此时将显示创建舞台窗口。



3. （可选）输入舞台名称。请选择创建舞台以创建舞台。此时将显示新舞台的舞台详细信息页面。

CLI 说明

要安装 AWS CLI，请参阅安装或更新 AWS CLI 的最新版本。

现在，您可以使用 CLI 创建和管理资源。舞台 API 在 ivs-realtime 命名空间下。例如，要创建舞台，以执行以下操作：

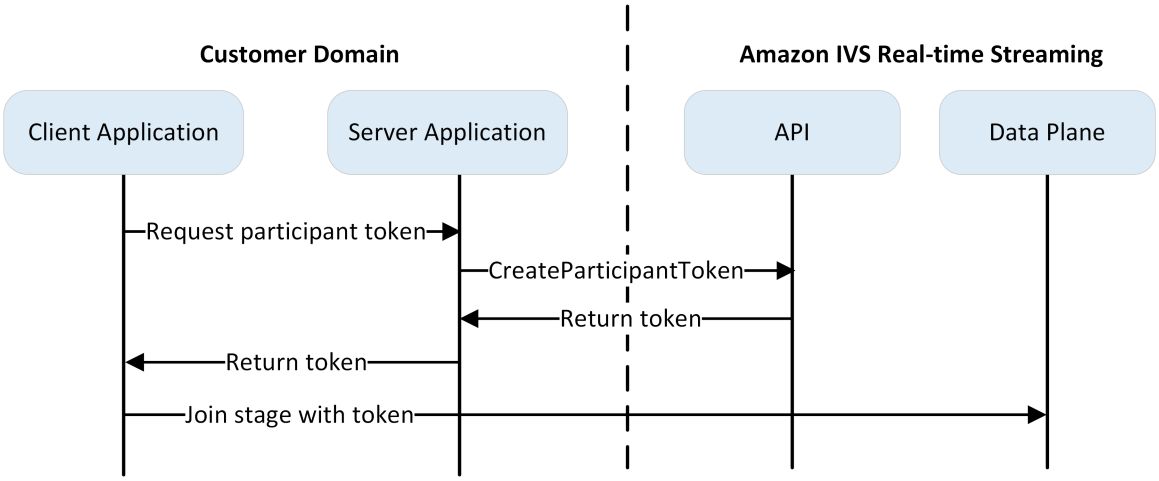
```
aws ivs-realtime create-stage --name "test-stage"
```

响应如下：

```
{
   "stage": {
      "arn": "arn:aws:ivs:us-west-2:376666121854:stage/VSWjvX5XOkU3",
```

```
        "name": "test-stage"
    }
}
```

分发参与者令牌



现在您拥有了舞台，您需要创建令牌并将其分发给参与者，以便他们能够加入舞台并开始发送和接收视频。

如上所示，客户端应用程序要求您的服务器应用程序提供令牌，服务器应用程序 CreateParticipantToken 使用 AWS SDK 或 SigV4 签名请求进行调用。由于 AWS 凭证用于调用 API，因此应在安全的服务器端应用程序中生成令牌，而不是在客户端应用程序中。

创建参与者令牌时，您可以选择指定该令牌启用的功能。默认功能为 PUBLISH 和 SUBSCRIBE，该功能允许参与者发送和接收音频和视频，但您可以发布具有子集功能的令牌。例如，您可以为监管人发布仅具有 SUBSCRIBE 功能的令牌。在这种情况下，监管人可以看到正在发送视频但不发送自己视频的参与者。

您可以通过控制台或 CLI 创建参与者令牌以进行测试和开发，但您很可能希望在生产环境中使用 AWS SDK 创建令牌。

您需要一种将令牌从服务器分发到每个客户端（例如，通过 API 请求）的方法。我们不提供此功能。在本指南中，您只需遵循以下步骤，即可将令牌复制并粘贴到客户端代码。

重要：将令牌视为不透明；也就是说，不要根据令牌内容构建功能。令牌的格式未来可能会发生变化。

控制台说明

1. 导航到您在上一步骤中创建的舞台。

2. 请选择创建参与者令牌。创建参与者令牌窗口出现。
3. 输入要与令牌关联的用户 ID。这可以是任何 UTF-8 编码的文本。
4. 请选择创建参与者令牌。
5. 复制令牌。重要提示：请务必保存令牌；IVS 不存储令牌，您之后也无法检索令牌。

CLI 说明

使用 AWS CLI 创建令牌需要您先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。注意：使用 AWS CLI 生成令牌适合用于测试目的，但对于生产用途而言，我们建议您使用 AWS SDK 在服务器端生成令牌（请参阅以下说明）。

1. 使用舞台 ARN 运行 `create-participant-token` 命令。包括以下任何或所有功能：`"PUBLISH"`、`"SUBSCRIBE"`。

```
aws ivs-realtime create-participant-token --stage-arn arn:aws:ivs:us-
west-2:376666121854:stage/VSWjvX5XOkU3 --capabilities '["PUBLISH", "SUBSCRIBE"]'
```

2. 这将返回一个参与者令牌：

```
{
    "participantToken": {
        "capabilities": [
            "PUBLISH",
            "SUBSCRIBE"
        ],
        "expirationTime": "2023-06-03T07:04:31+00:00",
        "participantId": "tU06DT5jCJeb",
        "token":
 "eyJhbGciOiJLTVMiLCJ0eXAiOiJKV1QifQ.eyJleHAiOjE2NjE1NDE0MjAsImp0aSI6ImpGcFdtdmVFTm9sUyIsInJ
TaKjllW9Qac6c5xBrdAk"    }
}
```

3. 保存该令牌。您需要此令牌才能加入舞台并发送和接收视频。

AWS SDK 说明

您可以使用 AWS SDK 创建令牌。以下是 AWS SDK 的使用说明 JavaScript。

重要提示：此代码必须在服务器端执行，并将其输出内容传递给客户端。

先决条件：要使用下面的代码示例，您需要安装 aws-sd client-ivs-realtime k/ 软件包。有关详细信息，请参阅适用的 AWS 开发工具包入门 JavaScript。

```
import { IVSRealTimeClient, CreateParticipantTokenCommand } from "@aws-sdk/client-ivs-realtime";

const ivsRealtimeClient = new IVSRealTimeClient({ region: 'us-west-2' });
const stageArn = 'arn:aws:ivs:us-west-2:123456789012:stage/L210UYabcdef';
const createStageTokenRequest = new CreateParticipantTokenCommand({
  stageArn,
});
const response = await ivsRealtimeClient.send(createStageTokenRequest);
console.log('token', response.participantToken.token);
```

集成 IVS 广播 SDK

IVS 提供适用于 Web、Android 和 iOS 的广播 SDK，您可以将其集成到应用程序中。广播 SDK 用于发送和接收视频。在这一部分，我们编写了一个简单的应用程序，可让两个或多个参与者进行实时交互。以下步骤将指导您创建名为的应用程序 BasicRealTime。完整的应用程序代码已开启 CodePen 并且 GitHub：

- Web：https://codepen.io/amazon-ivs/pen/ZEqgrpo/cbe7ac3b0ecc8c0f0a5c0dc9d6d36433
- 安卓：https://github.com/aws-samples/ amazon-ivs-real-time-streaming-android-samples
- iOS：https://github.com/aws-samples/ amazon-ivs-real-time-streaming-ios-samples

Web

设置文件

首先，创建一个文件夹和一个初始 HTML 和 JS 文件来设置文件：

```
mkdir realtime-web-example
cd realtime-web-example
touch index.html
touch app.js
```

您可以使用脚本标签或 npm 安装广播 SDK。为简单起见，示例使用了脚本标签，但如果您稍后选择使用 npm，也能轻易修改。

使用脚本标签

网络广播 SDK 作为 JavaScript 库分发，可通过 https://web-broadcast.live-video.net/1.8.0/ amazon-ivs-web-broadcast .js 进行检索。

通过 <script> 标签加载时，该库会在窗口作用域中公开一个名为 IVSBroadcastClient 的全局变量。

使用 npm

安装 npm 程序包：

```
npm install amazon-ivs-web-broadcast
```

您现在可以访问 IVS BroadcastClient 对象了：

```
const { Stage } = IVSBroadcastClient;
```

Android

创建 Android 项目

1. 在 Android Studio 中，创建新项目。
2. 选择空白视图活动。

   注意：在某些较旧版本的 Android Studio 中，基于视图的活动称为空白活动。如果您的 Android Studio 窗口显示空白活动，并且确实不显示空白视图活动，选择空白活动。否则，请勿选择空白活动，因为将使用 View API（而不是 Jetpack Compose）。
3. 给项目起一个名称，然后选择完成。

安装广播 SDK

要将 Amazon IVS Android 广播库添加到您的 Android 开发环境中，请将该库添加到您模块的 build.gradle 文件，如此处所示（适用于最新版本的 Amazon IVS 广播 SDK）。在较新的项目中，mavenCentral 存储库可能已经包含在您的 settings.gradle 文件中，如果是这种情况，您可以省略 repositories 数据块。对于我们的示例，我们还需要在 android 数据块中启用数据绑定。

```
android {
    dataBinding.enabled true
}

repositories {
    mavenCentral()
}

dependencies {
     implementation 'com.amazonaws:ivs-broadcast:1.14.1:stages@aar'
}
```

如要手动安装 SDK，也可从以下位置下载最新版本：

https://search.maven.org/artifact/com.amazonaws/ivs-broadcast

iOS

创建 iOS 项目

1. 创建新 Xcode 项目。
2. 对于平台，选择 iOS。
3. 对于应用程序，选择应用程序。
4. 输入应用程序的商品名称，然后选择下一步。
5. 选择（导航到）保存项目的目录，然后选择创建。

接下来您需要引入 SDK。我们建议您通过集成广播 SDK CocoaPods。或者，您可以手动将框架添加至项目。这两种方法如下所述。

推荐：安装广播 SDK (CocoaPods)

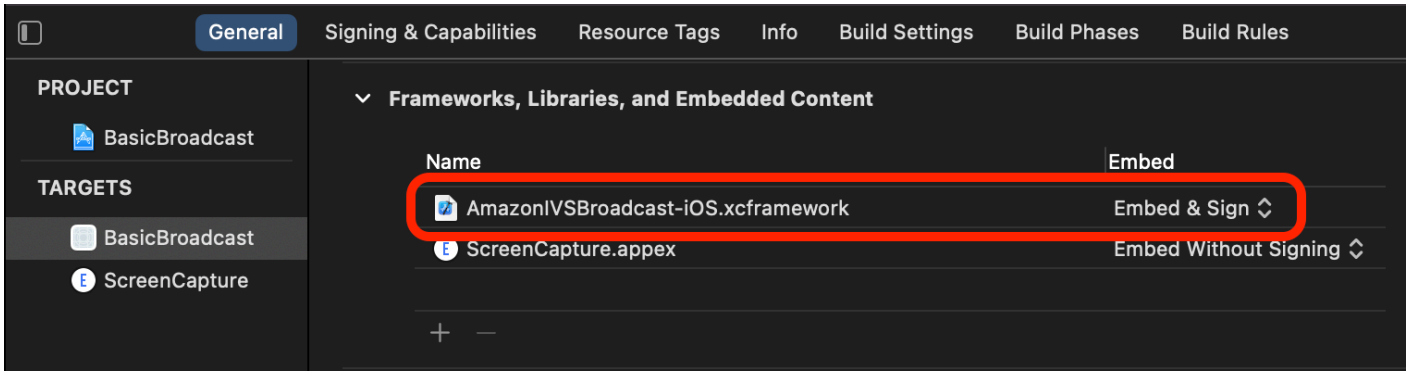
假设您的项目名称是 `BasicRealTime`，使用下列内容在项目文件夹中创建 `Podfile`，然后运行 `pod install`：

```
target 'BasicRealTime' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!
```

```
  # Pods for BasicRealTime
  pod 'AmazonIVSBroadcast/Stages'
end
```

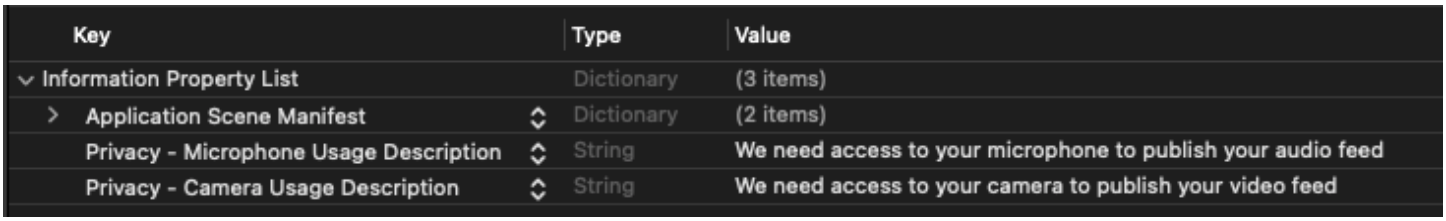
替代方法：手动安装框架

1. 从 https://broadcast.live-video.net/1.14.1/AmazonIVSBroadcast-Stages.xcframework.zip 下载最新版本。
2. 提取归档的内容。`AmazonIVSBroadcast.xcframework` 包含适用于设备和模拟器的开发工具包。
3. 通过以下方法嵌入 `AmazonIVSBroadcast.xcframework`：将其拖动到应用程序目标 General（常规）选项卡上的 Frameworks, Libraries, and Embedded Content（框架、库和嵌入式内容）部分：



配置权限

您需要更新项目的 `Info.plist`，以便为 `NSCameraUsageDescription` 和 `NSMicrophoneUsageDescription` 添加两个新条目。对于这些值，请提供面向用户的说明，解释您的应用程序为何要求访问相机和麦克风。



发布和订阅视频

有关网页、安卓和 iOS 的详细信息，请参阅下文。

Web

创建 HTML 样板

首先，创建 HTML 样板，并将该库作为脚本标签导入：

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Import the SDK -->
  <script src="https://web-broadcast.live-video.net/1.8.0/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>

<!-- TODO - fill in with next sections -->
<script src="./app.js"></script>

</body>
</html>
```

接受令牌输入并添加“加入/离开”按钮

使用输入控件在此处填充正文。它们将令牌作为输入，然后设置加入和离开按钮。通常，应用程序会从应用程序的 API 请求令牌，但在本示例中，您需要将令牌复制并粘贴到令牌输入中。

```
<h1>IVS Real-Time Streaming</h1>
<hr />

<label for="token">Token</label>
<input type="text" id="token" name="token" />
<button class="button" id="join-button">Join</button>
<button class="button" id="leave-button" style="display: none;">Leave</button>
<hr />
```

添加媒体容器元素

这些元素将为本地和远程参与者保留媒体。添加脚本标签来加载在 app.js 中定义的应用程序逻辑。

```
<!-- Local Participant -->
<div id="local-media"></div>

<!-- Remote Participants -->
<div id="remote-media"></div>

<!-- Load Script -->
<script src="./app.js"></script>
```

这样就完成了 HTML 页面，在浏览器中加载 index.html 时您应该会看到以下内容：

创建 app.js

开始定义 app.js 文件的内容。首先，从 SDK 的全局导入所有必需的属性：

```
const {
  Stage,
  LocalStageStream,
  SubscribeType,
  StageEvents,
  ConnectionState,
  StreamType
} = IVSBroadcastClient;
```

创建应用程序变量

建立变量以保存对加入和离开按钮 HTML 元素的引用，并存储应用程序的状态：

```
let joinButton = document.getElementById("join-button");
```

```
let leaveButton = document.getElementById("leave-button");

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
let cameraStageStream;
let micStageStream;
```

创建 joinStage 1：定义函数并验证输入

joinStage 函数获取输入令牌，创建与舞台的连接，然后开始发布从 getUserMedia 中检索的视频和音频。

首先，定义函数并验证状态和令牌输入。我们将在接下来的几个部分中完善此功能。

```
const joinStage = async () => {
  if (connected || joining) {
    return;
  }
  joining = true;

  const token = document.getElementById("token").value;

  if (!token) {
    window.alert("Please enter a participant token");
    joining = false;
    return;
  }

  // Fill in with the next sections
};
```

创建 joinStage 2：发布媒体

以下是将发布到舞台的媒体：

```
async function getCamera() {
  // Use Max Width and Height
  return navigator.mediaDevices.getUserMedia({
```

```
    video: true,
    audio: false
  });
}

async function getMic() {
  return navigator.mediaDevices.getUserMedia({
    video: false,
    audio: true
  });
}

// Retrieve the User Media currently set on the page
localCamera = await getCamera();
localMic = await getMic();

// Create StageStreams for Audio and Video
cameraStageStream = new LocalStageStream(localCamera.getVideoTracks()[0]);
micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);
```

创建 joinStage 3：定义舞台策略并创建舞台

这个舞台策略是决策逻辑的核心，SDK 将使用此策略来决定要发布什么内容和订阅哪些参与者。有关此函数用途的更多信息，请参阅 Strategy。

这个策略很简单。加入舞台后，发布刚刚检索的流，并订阅每个远程参与者的音频和视频：

```
const strategy = {
  stageStreamsToPublish() {
    return [cameraStageStream, micStageStream];
  },
  shouldPublishParticipant() {
    return true;
  },
  shouldSubscribeToParticipant() {
    return SubscribeType.AUDIO_VIDEO;
  }
};

stage = new Stage(token, strategy);
```

创建 JoinStage 4：处理舞台事件和渲染媒体

舞台会发出许多事件。需要侦
听 STAGE_PARTICIPANT_STREAMS_ADDED 和 STAGE_PARTICIPANT_LEFT，以将媒体渲染到页面和从页面中移除媒体。事件中列出了一组更详尽的事件。

请注意，我们在这里创建了四个帮助程序函数，以帮助管理必要的 DOM 元
素：setupParticipant、teardownParticipant、createVideoEl 和 createContainer。

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
  connected = state === ConnectionState.CONNECTED;

  if (connected) {
    joining = false;
    joinButton.style = "display: none";
    leaveButton.style = "display: inline-block";
  }
});

stage.on(
  StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED,
  (participant, streams) => {
    console.log("Participant Media Added: ", participant, streams);

    let streamsToDisplay = streams;

    if (participant.isLocal) {
      // Ensure to exclude local audio streams, otherwise echo will occur
      streamsToDisplay = streams.filter(
        (stream) => stream.streamType === StreamType.VIDEO
      );
    }

    const videoEl = setupParticipant(participant);
    streamsToDisplay.forEach((stream) =>
      videoEl.srcObject.addTrack(stream.mediaStreamTrack)
    );
  }
);

stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
  console.log("Participant Left: ", participant);
  teardownParticipant(participant);
```

```
});


// Helper functions for managing DOM

function setupParticipant({ isLocal, id }) {
  const groupId = isLocal ? "local-media" : "remote-media";
  const groupContainer = document.getElementById(groupId);

  const participantContainerId = isLocal ? "local" : id;
  const participantContainer = createContainer(participantContainerId);
  const videoEl = createVideoEl(participantContainerId);

  participantContainer.appendChild(videoEl);
  groupContainer.appendChild(participantContainer);

  return videoEl;
}

function teardownParticipant({ isLocal, id }) {
  const groupId = isLocal ? "local-media" : "remote-media";
  const groupContainer = document.getElementById(groupId);
  const participantContainerId = isLocal ? "local" : id;

  const participantDiv = document.getElementById(
    participantContainerId + "-container"
  );
  if (!participantDiv) {
    return;
  }
  groupContainer.removeChild(participantDiv);
}

function createVideoEl(id) {
  const videoEl = document.createElement("video");
  videoEl.id = id;
  videoEl.autoplay = true;
  videoEl.playsInline = true;
  videoEl.srcObject = new MediaStream();
  return videoEl;
}

function createContainer(id) {
  const participantContainer = document.createElement("div");
```

```
  participantContainer.classList = "participant-container";
  participantContainer.id = id + "-container";

  return participantContainer;
}
```

创建 joinStage 5：加入舞台

通过最终加入舞台来完成 `joinStage` 函数吧！

```
try {
  await stage.join();
} catch (err) {
  joining = false;
  connected = false;
  console.error(err.message);
}
```

创建 leaveStage

定义 `leaveStage` 函数，以调用离开按钮。

```
const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;
};
```

初始化输入事件处理程序

把 `app.js` 文件添加到最后一个函数。加载页面时会立即调用此函数，并建立用于加入和离开舞台的事件处理程序。

```
const init = async () => {
  try {
    // Prevents issues on Safari/FF so devices are not blank
    await navigator.mediaDevices.getUserMedia({ video: true, audio: true });
  } catch (e) {
    alert(
```

```
      "Problem retrieving media! Enable camera and microphone permissions."
    );
  }

  joinButton.addEventListener("click", () => {
    joinStage();
  });

  leaveButton.addEventListener("click", () => {
    leaveStage();
    joinButton.style = "display: inline-block";
    leaveButton.style = "display: none";
  });
};

init(); // call the function
```

运行应用程序并提供令牌

这时您可以在本地或与其他人共享网页，只需打开页面，输入参与者令牌并加入舞台即可。

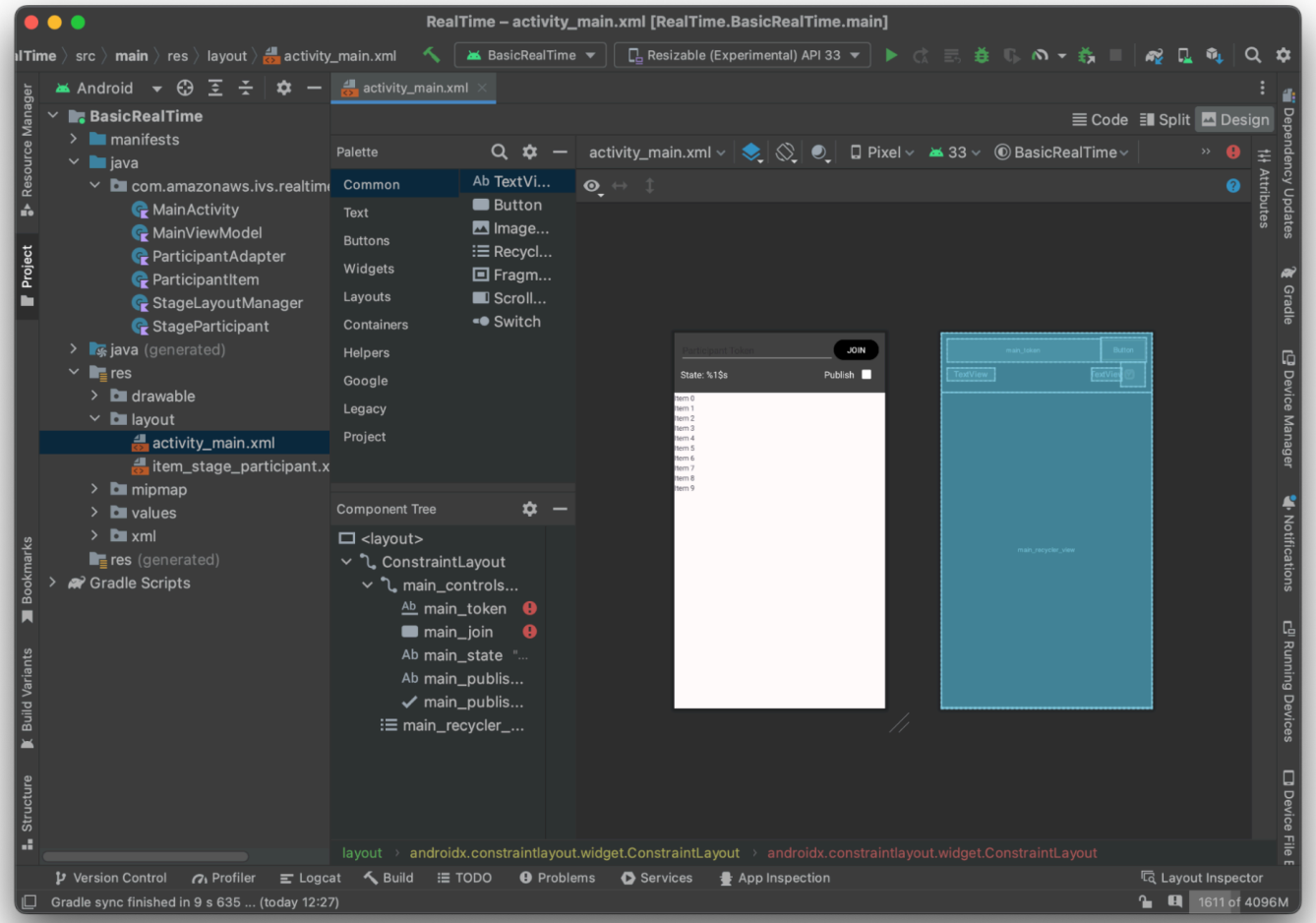
接下来做什么？

有关涉及 npm、React 等的更多详细示例，请参阅 IVS 广播 SDK：网络指南（实时直播功能指南）。

Android

创建视图

首先使用自动创建的 `activity_main.xml` 文件为应用程序创建一个简单的布局。此布局包含要添加到令牌的 `EditText`、加入 `Button`、显示舞台状态的 `TextView` 和切换发布的 `CheckBox`。



以下是视图背后的 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:keepScreenOn="true"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".BasicActivity">

        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/main_controls_container"
            android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"
            android:background="@color/cardview_dark_background"
            android:padding="12dp"
            app:layout_constraintTop_toTopOf="parent">

            <EditText
                android:id="@+id/main_token"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                android:autofillHints="@null"
                android:backgroundTint="@color/white"
                android:hint="@string/token"
                android:imeOptions="actionDone"
                android:inputType="text"
                android:textColor="@color/white"
                app:layout_constraintEnd_toStartOf="@id/main_join"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent" />

            <Button
                android:id="@+id/main_join"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:backgroundTint="@color/black"
                android:text="@string/join"
                android:textAllCaps="true"
                android:textColor="@color/white"
                android:textSize="16sp"
                app:layout_constraintBottom_toBottomOf="@+id/main_token"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintStart_toEndOf="@id/main_token" />

            <TextView
                android:id="@+id/main_state"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/state"
                android:textColor="@color/white"
                android:textSize="18sp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toBottomOf="@id/main_token" />

            <TextView
```

```
                android:id="@+id/main_publish_text"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@string/publish"
                android:textColor="@color/white"
                android:textSize="18sp"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintEnd_toStartOf="@id/main_publish_checkbox"
                app:layout_constraintTop_toBottomOf="@id/main_token" />

            <CheckBox
                android:id="@+id/main_publish_checkbox"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:buttonTint="@color/white"
                android:checked="true"
                app:layout_constraintBottom_toBottomOf="@id/main_publish_text"
                app:layout_constraintEnd_toEndOf="parent"
                app:layout_constraintTop_toTopOf="@id/main_publish_text" />

        </androidx.constraintlayout.widget.ConstraintLayout>

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/main_recycler_view"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:layout_constraintTop_toBottomOf="@+id/main_controls_container"
            app:layout_constraintBottom_toBottomOf="parent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
<layout>
```

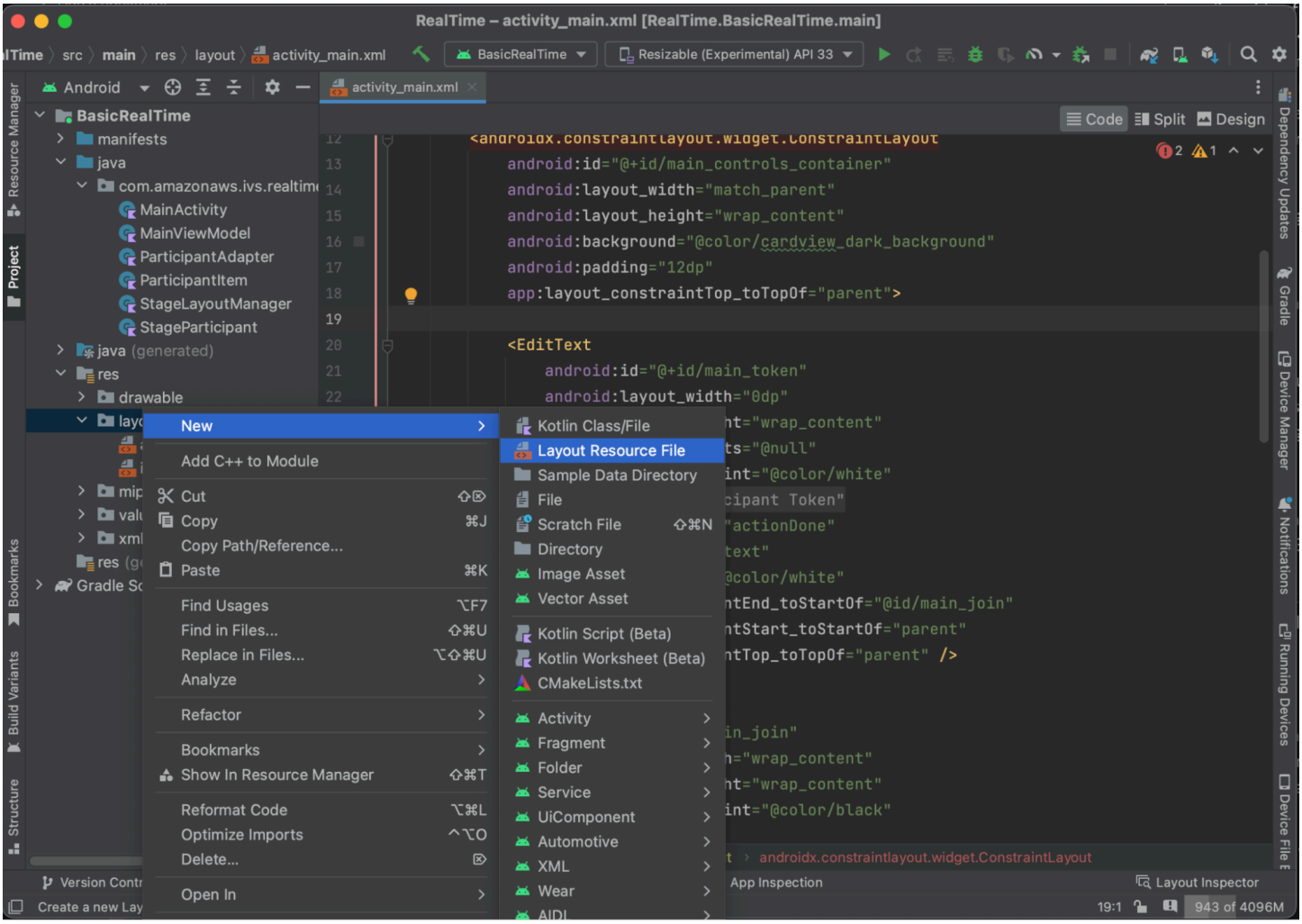
我们在此处引用了几个字符串 ID，因此现在将创建整个 `strings.xml` 文件：

```
<resources>
    <string name="app_name">BasicRealTime</string>
    <string name="join">Join</string>
    <string name="leave">Leave</string>
    <string name="token">Participant Token</string>
    <string name="publish">Publish</string>
    <string name="state">State: %1$s</string>
</resources>
```

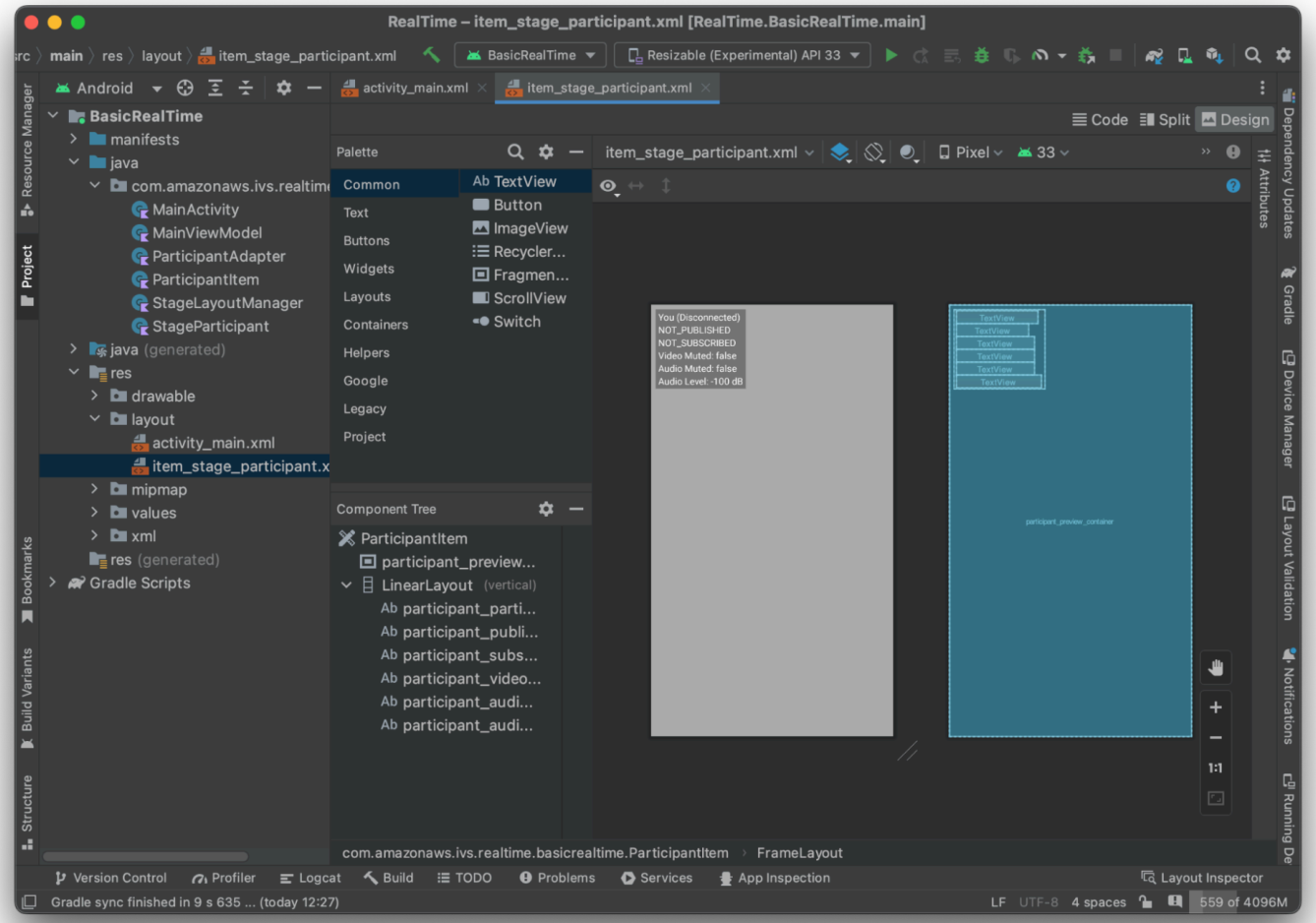
将 XML 中的这些视图链接到 MainActivity.kt：

```
import android.widget.Button
import android.widget.CheckBox
import android.widget.EditText
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

private lateinit var checkboxPublish: CheckBox
private lateinit var recyclerView: RecyclerView
private lateinit var buttonJoin: Button
private lateinit var textViewState: TextView
private lateinit var editTextToken: EditText

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    checkboxPublish = findViewById(R.id.main_publish_checkbox)
    recyclerView = findViewById(R.id.main_recycler_view)
    buttonJoin = findViewById(R.id.main_join)
    textViewState = findViewById(R.id.main_state)
    editTextToken = findViewById(R.id.main_token)
}
```

现在我们为 RecyclerView 创建一个项目视图。要执行此操作，右键单击 res/layout 目录，然后选择创建 > 布局资源文件。将此新文件命名为 item_stage_participant.xml。



此项目的布局很简单：它包含用于渲染参与者视频流的视图和用于显示参与者有关信息的标签列表：



以下是 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<com.amazonaws.ivs.realtime.basicrealtime.ParticipantItem xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <FrameLayout
        android:id="@+id/participant_preview_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:background="@android:color/darker_gray" />
```

```
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:background="#50000000"
        android:orientation="vertical"
        android:paddingLeft="4dp"
        android:paddingTop="2dp"
        android:paddingRight="4dp"
        android:paddingBottom="2dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <TextView
            android:id="@+id/participant_participant_id"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="You (Disconnected)" />

        <TextView
            android:id="@+id/participant_publishing"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="NOT_PUBLISHED" />

        <TextView
            android:id="@+id/participant_subscribed"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="NOT_SUBSCRIBED" />

        <TextView
            android:id="@+id/participant_video_muted"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
```

```
            tools:text="Video Muted: false" />

        <TextView
            android:id="@+id/participant_audio_muted"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="Audio Muted: false" />

        <TextView
            android:id="@+id/participant_audio_level"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@android:color/white"
            android:textSize="16sp"
            tools:text="Audio Level: -100 dB" />

    </LinearLayout>

</com.amazonaws.ivs.realtime.basicrealtime.ParticipantItem>
```

这个 XML 文件扩大了还没有创建的类 `ParticipantItem`。由于 XML 包含完整的命名空间，因此请务必将此 XML 文件更新到您的命名空间。创建这个类并设置视图，但暂时将其保留为空。

创建一个新的 Kotlin 类，`ParticipantItem`：

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.content.Context
import android.util.AttributeSet
import android.widget.FrameLayout
import android.widget.TextView
import kotlin.math.roundToInt

class ParticipantItem @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0,
    defStyleRes: Int = 0,
) : FrameLayout(context, attrs, defStyleAttr, defStyleRes) {

    private lateinit var previewContainer: FrameLayout
```

```
    private lateinit var textViewParticipantId: TextView
    private lateinit var textViewPublish: TextView
    private lateinit var textViewSubscribe: TextView
    private lateinit var textViewVideoMuted: TextView
    private lateinit var textViewAudioMuted: TextView
    private lateinit var textViewAudioLevel: TextView

    override fun onFinishInflate() {
        super.onFinishInflate()
        previewContainer = findViewById(R.id.participant_preview_container)
        textViewParticipantId = findViewById(R.id.participant_participant_id)
        textViewPublish = findViewById(R.id.participant_publishing)
        textViewSubscribe = findViewById(R.id.participant_subscribed)
        textViewVideoMuted = findViewById(R.id.participant_video_muted)
        textViewAudioMuted = findViewById(R.id.participant_audio_muted)
        textViewAudioLevel = findViewById(R.id.participant_audio_level)
    }
}
```

权限

要使用相机和麦克风，您需要向用户请求权限。我们为此遵循标准权限流程：

```
override fun onStart() {
    super.onStart()
    requestPermission()
}

private val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions())
 { permissions ->
        if (permissions[Manifest.permission.CAMERA] == true &&
 permissions[Manifest.permission.RECORD_AUDIO] == true) {
            viewModel.permissionGranted() // we will add this later
        }
    }

private val permissions = listOf(
    Manifest.permission.CAMERA,
    Manifest.permission.RECORD_AUDIO,
)

private fun requestPermission() {
```

```
    when {
        this.hasPermissions(permissions) -> viewModel.permissionGranted() // we will
 add this later
        else -> requestPermissionLauncher.launch(permissions.toTypedArray())
    }
}

private fun Context.hasPermissions(permissions: List<String>): Boolean {
    return permissions.all {
        ContextCompat.checkSelfPermission(this, it) ==
 PackageManager.PERMISSION_GRANTED
    }
}
```

应用程序状态

我们的应用程序会在 a 中跟踪本地参与者`MainViewModel.kt`，状态将传回`MainActivity`使用 Kotlin 的StateFlow。

创建一个新的 Kotlin 类 `MainViewModel`：

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.app.Application
import androidx.lifecycle.AndroidViewModel

class MainViewModel(application: Application) : AndroidViewModel(application),
 Stage.Strategy, StageRenderer {

}
```

在 `MainActivity.kt` 中管理视图模型：

```
import androidx.activity.viewModels

private val viewModel: MainViewModel by viewModels()
```

要使用 `AndroidViewModel` 还有这些 Kotlin `ViewModel` 扩展，您需要将以下内容添加到模块的 `build.gradle` 文件：

```
implementation 'androidx.core:core-ktx:1.10.1'
implementation "androidx.activity:activity-ktx:1.7.2"
```

```
implementation 'androidx.appcompat:appcompat:1.6.1'
implementation 'com.google.android.material:material:1.10.0'
implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"

def lifecycle_version = "2.6.1"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
```

RecyclerView 适配器

创建一个简单的 RecyclerView.Adapter 子类来跟踪参与者并更新舞台活动中的 RecyclerView。但首先，要一个代表参与者的类。创建一个新的 Kotlin 类 StageParticipant：

```
package com.amazonaws.ivs.realtime.basicrealtime

import com.amazonaws.ivs.broadcast.Stage
import com.amazonaws.ivs.broadcast.StageStream

class StageParticipant(val isLocal: Boolean, var participantId: String?) {
    var publishState = Stage.PublishState.NOT_PUBLISHED
    var subscribeState = Stage.SubscribeState.NOT_SUBSCRIBED
    var streams = mutableListOf<StageStream>()

    val stableID: String
        get() {
            return if (isLocal) {
                "LocalUser"
            } else {
                requireNotNull(participantId)
            }
        }
}
```

将在接下来要创建的 ParticipantAdapter 中使用此类。首先定义类并创建一个变量来跟踪参与者：

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.view.LayoutInflater
import android.view.ViewGroup
```

```
import androidx.recyclerview.widget.RecyclerView

class ParticipantAdapter : RecyclerView.Adapter<ParticipantAdapter.ViewHolder>() {

    private val participants = mutableListOf<StageParticipant>()
```

在实现其余的覆盖之前，还必须定义 RecyclerView.ViewHolder：

```
class ViewHolder(val participantItem: ParticipantItem) :
 RecyclerView.ViewHolder(participantItem)
```

如此，便可以实现标准 RecyclerView.Adapter 覆盖：

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val item = LayoutInflater.from(parent.context)
        .inflate(R.layout.item_stage_participant, parent, false) as ParticipantItem
    return ViewHolder(item)
}

override fun getItemCount(): Int {
    return participants.size
}

override fun getItemId(position: Int): Long =
    participants[position]
        .stableID
        .hashCode()
        .toLong()

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    return holder.participantItem.bind(participants[position])
}

override fun onBindViewHolder(holder: ViewHolder, position: Int, payloads:
 MutableList<Any>) {
    val updates = payloads.filterIsInstance<StageParticipant>()
    if (updates.isNotEmpty()) {
        updates.forEach { holder.participantItem.bind(it) // implemented later }
    } else {
        super.onBindViewHolder(holder, position, payloads)
    }
}
```

最后，我们添加了新方法，参与者发生更改时，将从 MainViewModel 中调用这些新方法。这些方法是适配器上的标准 CRUD 操作。

```
fun participantJoined(participant: StageParticipant) {
    participants.add(participant)
    notifyItemInserted(participants.size - 1)
}

fun participantLeft(participantId: String) {
    val index = participants.indexOfFirst { it.participantId == participantId }
    if (index != -1) {
        participants.removeAt(index)
        notifyItemRemoved(index)
    }
}

fun participantUpdated(participantId: String?, update: (participant: StageParticipant)
 -> Unit) {
    val index = participants.indexOfFirst { it.participantId == participantId }
    if (index != -1) {
        update(participants[index])
        notifyItemChanged(index, participants[index])
    }
}
```

返回 MainViewModel，需要创建并保留对此适配器的引用：

```
internal val participantAdapter = ParticipantAdapter()
```

阶段状态

还需要跟踪 MainViewModel 内的某些舞台状态。现在来定义这些属性：

```
private val _connectionState = MutableStateFlow(Stage.ConnectionState.DISCONNECTED)
val connectionState = _connectionState.asStateFlow()

private var publishEnabled: Boolean = false
    set(value) {
        field = value
        // Because the strategy returns the value of `checkboxPublish.isChecked`, just
 call `refreshStrategy`.
        stage?.refreshStrategy()
```

```
    }

private var deviceDiscovery: DeviceDiscovery? = null
private var stage: Stage? = null
private var streams = mutableListOf<LocalStageStream>()
```

要在加入舞台之前查看自己的预览，立即创建本地参与者：

```
init {
    deviceDiscovery = DeviceDiscovery(application)

    // Create a local participant immediately to render our camera preview and
 microphone stats
    val localParticipant = StageParticipant(true, null)
    participantAdapter.participantJoined(localParticipant)
}
```

确保在清理 ViewModel 时也清理这些资源。立即覆盖 onCleared()，以便不会忘记清理这些资源。

```
override fun onCleared() {
    stage?.release()
    deviceDiscovery?.release()
    deviceDiscovery = null
    super.onCleared()
}
```

现在，一旦授予权限，就会填充本地 streams 属性，实施之前调用的 permissionsGranted 方法：

```
internal fun permissionGranted() {
    val deviceDiscovery = deviceDiscovery ?: return
    streams.clear()
    val devices = deviceDiscovery.listLocalDevices()
    // Camera
    devices
        .filter { it.descriptor.type == Device.Descriptor.DeviceType.CAMERA }
        .maxByOrNull { it.descriptor.position == Device.Descriptor.Position.FRONT }
        ?.let { streams.add(ImageLocalStageStream(it)) }
    // Microphone
    devices
        .filter { it.descriptor.type == Device.Descriptor.DeviceType.MICROPHONE }
        .maxByOrNull { it.descriptor.isDefault }
```

```
        ?.let { streams.add(AudioLocalStageStream(it)) }

    stage?.refreshStrategy()

    // Update our local participant with these new streams
    participantAdapter.participantUpdated(null) {
        it.streams.clear()
        it.streams.addAll(streams)
    }
}
```

实施舞台 SDK

三个核心概念构成了实时功能的基础：舞台、策略和渲染器。设计目标是最大限度地减少构建有效产品所需的客户端逻辑量。

Stage.Strategy

`Stage.Strategy` 实施很简单：

```
override fun stageStreamsToPublishForParticipant(
    stage: Stage,
    participantInfo: ParticipantInfo
): MutableList<LocalStageStream> {
    // Return the camera and microphone to be published.
    // This is only called if `shouldPublishFromParticipant` returns true.
    return streams
}

override fun shouldPublishFromParticipant(stage: Stage, participantInfo:
 ParticipantInfo): Boolean {
    return publishEnabled
}

override fun shouldSubscribeToParticipant(stage: Stage, participantInfo:
 ParticipantInfo): Stage.SubscribeType {
    // Subscribe to both audio and video for all publishing participants.
    return Stage.SubscribeType.AUDIO_VIDEO
}
```

总而言之，要根据内部 `publishEnabled` 状态发布内容，如果要发布，将发布之前收集的流。最后，对于此示例，我们将始终订阅其他参与者并接收他们的音频和视频。

StageRenderer

考虑到函数的数量，尽管 StageRenderer 包含更多的代码，但是它实施起来也相当简单。此渲染器中的一般方法是，SDK 通知参与者发生更改时更新 ParticipantAdapter。在某些情况下，我们会以不同的方式处理本地参与者，因为我们决定自己管理这些参与者，这样他们就可以在加入舞台之前看到自己的相机预览。

```
override fun onError(exception: BroadcastException) {
    Toast.makeText(getApplication(), "onError ${exception.localizedMessage}",
 Toast.LENGTH_LONG).show()
    Log.e("BasicRealTime", "onError $exception")
}

override fun onConnectionStateChanged(
    stage: Stage,
    connectionState: Stage.ConnectionState,
    exception: BroadcastException?
) {
    _connectionState.value = connectionState
}

override fun onParticipantJoined(stage: Stage, participantInfo: ParticipantInfo) {
    if (participantInfo.isLocal) {
        // If this is the local participant joining the stage, update the participant
 with a null ID because we
        // manually added that participant when setting up our preview
        participantAdapter.participantUpdated(null) {
            it.participantId = participantInfo.participantId
        }
    } else {
        // If they are not local, add them normally
        participantAdapter.participantJoined(
            StageParticipant(
                participantInfo.isLocal,
                participantInfo.participantId
            )
        )
    }
}

override fun onParticipantLeft(stage: Stage, participantInfo: ParticipantInfo) {
    if (participantInfo.isLocal) {
```

```
        // If this is the local participant leaving the stage, update the ID but keep
 it around because
        // we want to keep the camera preview active
        participantAdapter.participantUpdated(participantInfo.participantId) {
            it.participantId = null
        }
    } else {
        // If they are not local, have them leave normally
        participantAdapter.participantLeft(participantInfo.participantId)
    }
}

override fun onParticipantPublishStateChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    publishState: Stage.PublishState
) {
    // Update the publishing state of this participant
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.publishState = publishState
    }
}

override fun onParticipantSubscribeStateChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    subscribeState: Stage.SubscribeState
) {
    // Update the subscribe state of this participant
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.subscribeState = subscribeState
    }
}

override fun onStreamsAdded(stage: Stage, participantInfo: ParticipantInfo, streams:
 MutableList<StageStream>) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, add these new streams to that participant's streams
 array.
    participantAdapter.participantUpdated(participantInfo.participantId) {
```

```
        it.streams.addAll(streams)
    }
}

override fun onStreamsRemoved(stage: Stage, participantInfo: ParticipantInfo, streams:
 MutableList<StageStream>) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, remove these streams from that participant's streams
 array.
    participantAdapter.participantUpdated(participantInfo.participantId) {
        it.streams.removeAll(streams)
    }
}

override fun onStreamsMutedChanged(
    stage: Stage,
    participantInfo: ParticipantInfo,
    streams: MutableList<StageStream>
) {
    // We don't want to take any action for the local participant because we track
 those streams locally
    if (participantInfo.isLocal) {
        return
    }
    // For remote participants, notify the adapter that the participant has been
 updated. There is no need to modify
    // the `streams` property on the `StageParticipant` because it is the same
 `StageStream` instance. Just
    // query the `isMuted` property again.
    participantAdapter.participantUpdated(participantInfo.participantId) {}
}
```

实现自定义 RecyclerView LayoutManager

安排不同数量的参与者可能很复杂。您希望参与者占据整个父视图的框架，但是不想单独处理每个参与者配置。为了简单起见，将逐步实施 RecyclerView.LayoutManager。

创建另一个新类 StageLayoutManager，它应该扩展 GridLayoutManager。此类旨在根据基于流的行/列布局中的参与者数量计算每个参与者的布局。每行的高度与其他行相同，但每行列的宽度各不相同。有关如何自定义该行为的说明，请参阅 layouts 变量上方的代码注释。

```
package com.amazonaws.ivs.realtime.basicrealtime

import android.content.Context
import androidx.recyclerview.widget.GridLayoutManager
import androidx.recyclerview.widget.RecyclerView

class StageLayoutManager(context: Context?) : GridLayoutManager(context, 6) {

    companion object {
        /**
         * This 2D array contains the description of how the grid of participants
 should be rendered
         * The index of the 1st dimension is the number of participants needed to
 active that configuration
         * Meaning if there is 1 participant, index 0 will be used. If there are 5
 participants, index 4 will be used.
         *
         * The 2nd dimension is a description of the layout. The length of the array is
 the number of rows that
         * will exist, and then each number within that array is the number of columns
 in each row.
         *
         * See the code comments next to each index for concrete examples.
         *
         * This can be customized to fit any layout configuration needed.
         */
        val layouts: List<List<Int>> = listOf(
            // 1 participant
            listOf(1), // 1 row, full width
            // 2 participants
            listOf(1, 1), // 2 rows, all columns are full width
            // 3 participants
            listOf(1, 2), // 2 rows, first row's column is full width then 2nd row's
 columns are 1/2 width
            // 4 participants
            listOf(2, 2), // 2 rows, all columns are 1/2 width
            // 5 participants
            listOf(1, 2, 2), // 3 rows, first row's column is full width, 2nd and 3rd
 row's columns are 1/2 width
```

```
            // 6 participants
            listOf(2, 2, 2), // 3 rows, all column are 1/2 width
            // 7 participants
            listOf(2, 2, 3), // 3 rows, 1st and 2nd row's columns are 1/2 width, 3rd
row's columns are 1/3rd width
            // 8 participants
            listOf(2, 3, 3),
            // 9 participants
            listOf(3, 3, 3),
            // 10 participants
            listOf(2, 3, 2, 3),
            // 11 participants
            listOf(2, 3, 3, 3),
            // 12 participants
            listOf(3, 3, 3, 3),
        )
    }

    init {
        spanSizeLookup = object : SpanSizeLookup() {
            override fun getSpanSize(position: Int): Int {
                if (itemCount <= 0) {
                    return 1
                }
                // Calculate the row we're in
                val config = layouts[itemCount - 1]
                var row = 0
                var curPosition = position
                while (curPosition - config[row] >= 0) {
                    curPosition -= config[row]
                    row++
                }
                // spanCount == max spans, config[row] = number of columns we want
                // So spanCount / config[row] would be something like 6 / 3 if we want
3 columns.
                // So this will take up 2 spans, with a max of 6 is 1/3rd of the view.
                return spanCount / config[row]
            }
        }
    }

    override fun onLayoutChildren(recycler: RecyclerView.Recycler?, state:
RecyclerView.State?) {
        if (itemCount <= 0 || state?.isPreLayout == true) return
```

```
        val parentHeight = height
        val itemHeight = parentHeight / layouts[itemCount - 1].size // height divided
 by number of rows.

        // Set the height of each view based on how many rows exist for the current
 participant count.
        for (i in 0 until childCount) {
            val child = getChildAt(i) ?: continue
            val layoutParams = child.layoutParams as RecyclerView.LayoutParams
            if (layoutParams.height != itemHeight) {
                layoutParams.height = itemHeight
                child.layoutParams = layoutParams
            }
        }
        // After we set the height for all our views, call super.
        // This works because our RecyclerView can not scroll and all views are always
 visible with stable IDs.
        super.onLayoutChildren(recycler, state)
    }

    override fun canScrollVertically(): Boolean = false
    override fun canScrollHorizontally(): Boolean = false
}
```

回到 MainActivity.kt 中，我们需要为 RecyclerView 设置适配器和布局管理器：

```
// In onCreate after setting recyclerView.
recyclerView.layoutManager = StageLayoutManager(this)
recyclerView.adapter = viewModel.participantAdapter
```

挂接 UI 操作

即将完成所有操作；只需要挂接几个 UI 操作。

首先让 MainActivity 观察 MainViewModel 的 StateFlow 变更：

```
// At the end of your onCreate method
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.CREATED) {
        viewModel.connectionState.collect { state ->
            buttonJoin.setText(if (state == ConnectionState.DISCONNECTED) R.string.join
 else R.string.leave)
```

```
            textViewState.text = getString(R.string.state, state.name)
        }
    }
}
```

接下来，将侦听器添加到“加入”按钮和“发布”复选框中：

```
buttonJoin.setOnClickListener {
    viewModel.joinStage(editTextToken.text.toString())
}
checkboxPublish.setOnCheckedChangeListener { _, isChecked ->
    viewModel.setPublishEnabled(isChecked)
}
```

上述两个事件调用正在实施的 MainViewModel 中的功能：

```
internal fun joinStage(token: String) {
    if (_connectionState.value != Stage.ConnectionState.DISCONNECTED) {
        // If we're already connected to a stage, leave it.
        stage?.leave()
    } else {
        if (token.isEmpty()) {
            Toast.makeText(getApplication(), "Empty Token", Toast.LENGTH_SHORT).show()
            return
        }
        try {
            // Destroy the old stage first before creating a new one.
            stage?.release()
            val stage = Stage(getApplication(), token, this)
            stage.addRenderer(this)
            stage.join()
            this.stage = stage
        } catch (e: BroadcastException) {
            Toast.makeText(getApplication(), "Failed to join stage
 ${e.localizedMessage}", Toast.LENGTH_LONG).show()
            e.printStackTrace()
        }
    }
}

internal fun setPublishEnabled(enabled: Boolean) {
    publishEnabled = enabled
```

```
}
```

渲染参与者

最后，需要将从 SDK 收到的数据渲染到之前创建的参与者项目上。我们已经完成了 RecyclerView 逻辑，所以只需要在 ParticipantItem 中实施 bind API。

首先添加空函数，然后逐步进行操作：

```
fun bind(participant: StageParticipant) {

}
```

首先，处理简易状态、参与者 ID、发布状态和订阅状态。对于这些，直接更新 TextViews：

```
val participantId = if (participant.isLocal) {
    "You (${participant.participantId ?: "Disconnected"})"
} else {
    participant.participantId
}
textViewParticipantId.text = participantId
textViewPublish.text = participant.publishState.name
textViewSubscribe.text = participant.subscribeState.name
```

接下来，更新音频和视频的静音状态。要进入静音状态，需要找到来自流数组的 ImageDevice 和 AudioDevice。要优化性能，需要记住最后连接的设备 ID。

```
// This belongs outside the `bind` API.
private var imageDeviceUrn: String? = null
private var audioDeviceUrn: String? = null

// This belongs inside the `bind` API.
val newImageStream = participant
    .streams
    .firstOrNull { it.device is ImageDevice }
textViewVideoMuted.text = if (newImageStream != null) {
    if (newImageStream.muted) "Video muted" else "Video not muted"
} else {
    "No video stream"
}

val newAudioStream = participant
```

```
    .streams
    .firstOrNull { it.device is AudioDevice }
textViewAudioMuted.text = if (newAudioStream != null) {
    if (newAudioStream.muted) "Audio muted" else "Audio not muted"
} else {
    "No audio stream"
}
```

最后，渲染 imageDevice 的预览：

```
if (newImageStream?.device?.descriptor?.urn != imageDeviceUrn) {
    // If the device has changed, remove all subviews from the preview container
    previewContainer.removeAllViews()
    (newImageStream?.device as? ImageDevice)?.let {
        val preview = it.getPreviewView(BroadcastConfiguration.AspectMode.FIT)
        previewContainer.addView(preview)
        preview.layoutParams = FrameLayout.LayoutParams(
            FrameLayout.LayoutParams.MATCH_PARENT,
            FrameLayout.LayoutParams.MATCH_PARENT
        )
    }
}
imageDeviceUrn = newImageStream?.device?.descriptor?.urn
```

然后显示来自 audioDevice 的音频统计数据：

```
if (newAudioStream?.device?.descriptor?.urn != audioDeviceUrn) {
    (newAudioStream?.device as? AudioDevice)?.let {
        it.setStatsCallback { _, rms ->
            textViewAudioLevel.text = "Audio Level: ${rms.roundToInt()} dB"
        }
    }
}
audioDeviceUrn = newAudioStream?.device?.descriptor?.urn
```

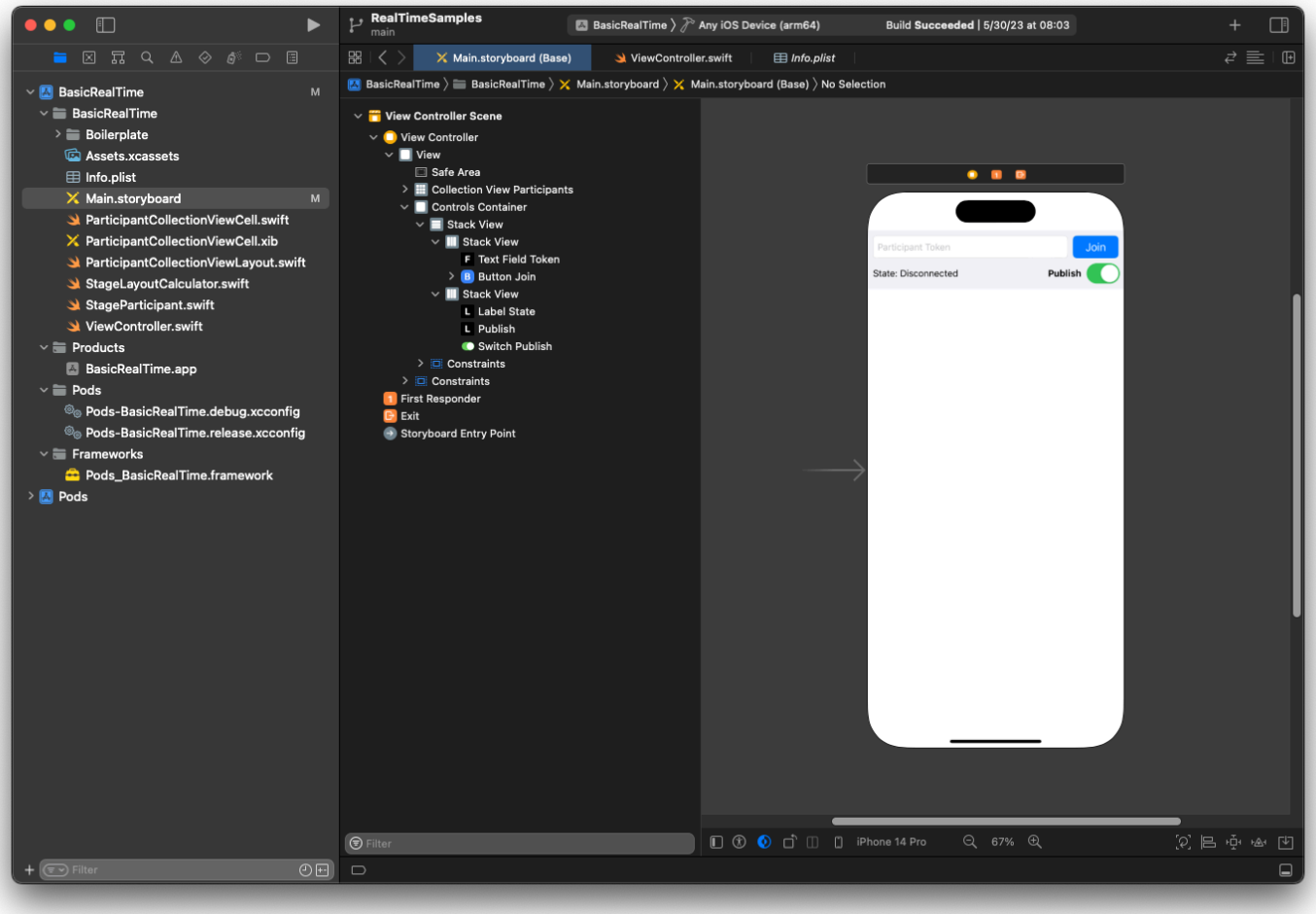
iOS

创建视图

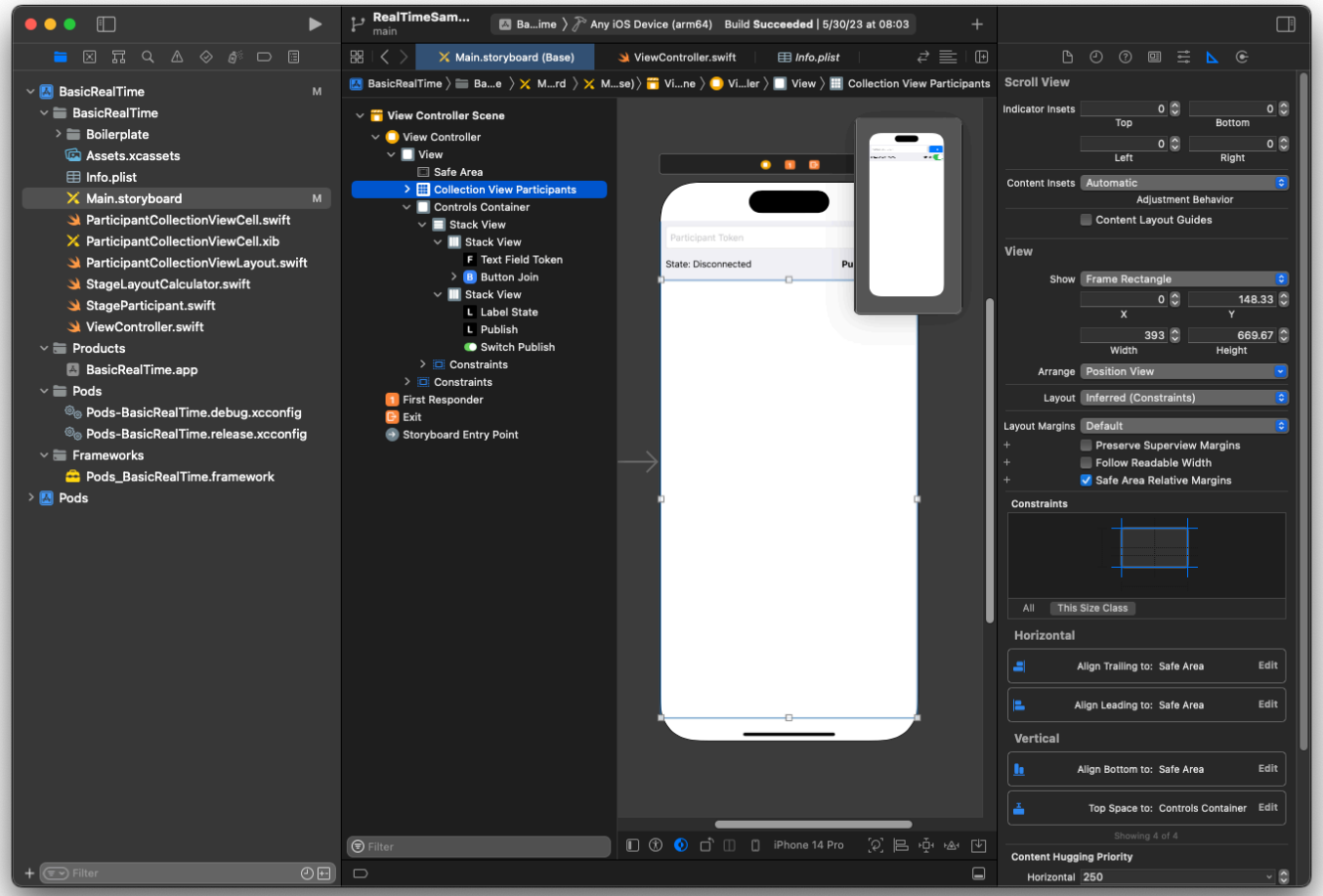
首先使用自动创建的 ViewController.swift 文件来导入 AmazonIVSBroadcast，然后添加一些要链接的 @IBOutlets：

```
import AmazonIVSBroadcast

class ViewController: UIViewController {

    @IBOutlet private var textFieldToken: UITextField!
    @IBOutlet private var buttonJoin: UIButton!
    @IBOutlet private var labelState: UILabel!
    @IBOutlet private var switchPublish: UISwitch!
    @IBOutlet private var collectionViewParticipants: UICollectionView!
```

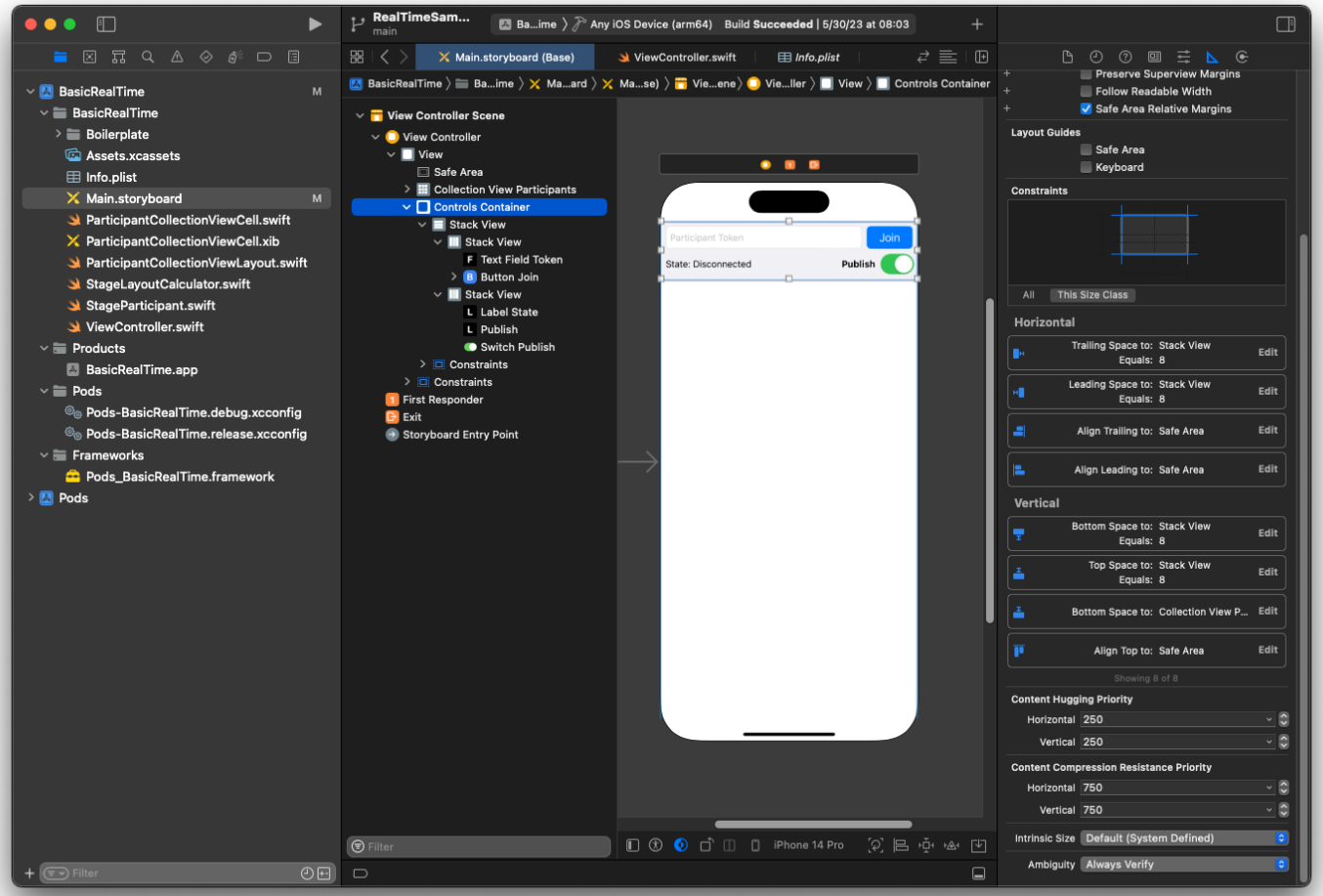
现在创建这些视图，然后在 Main.storyboard 中将其链接起来。以下是将使用的视图结构：



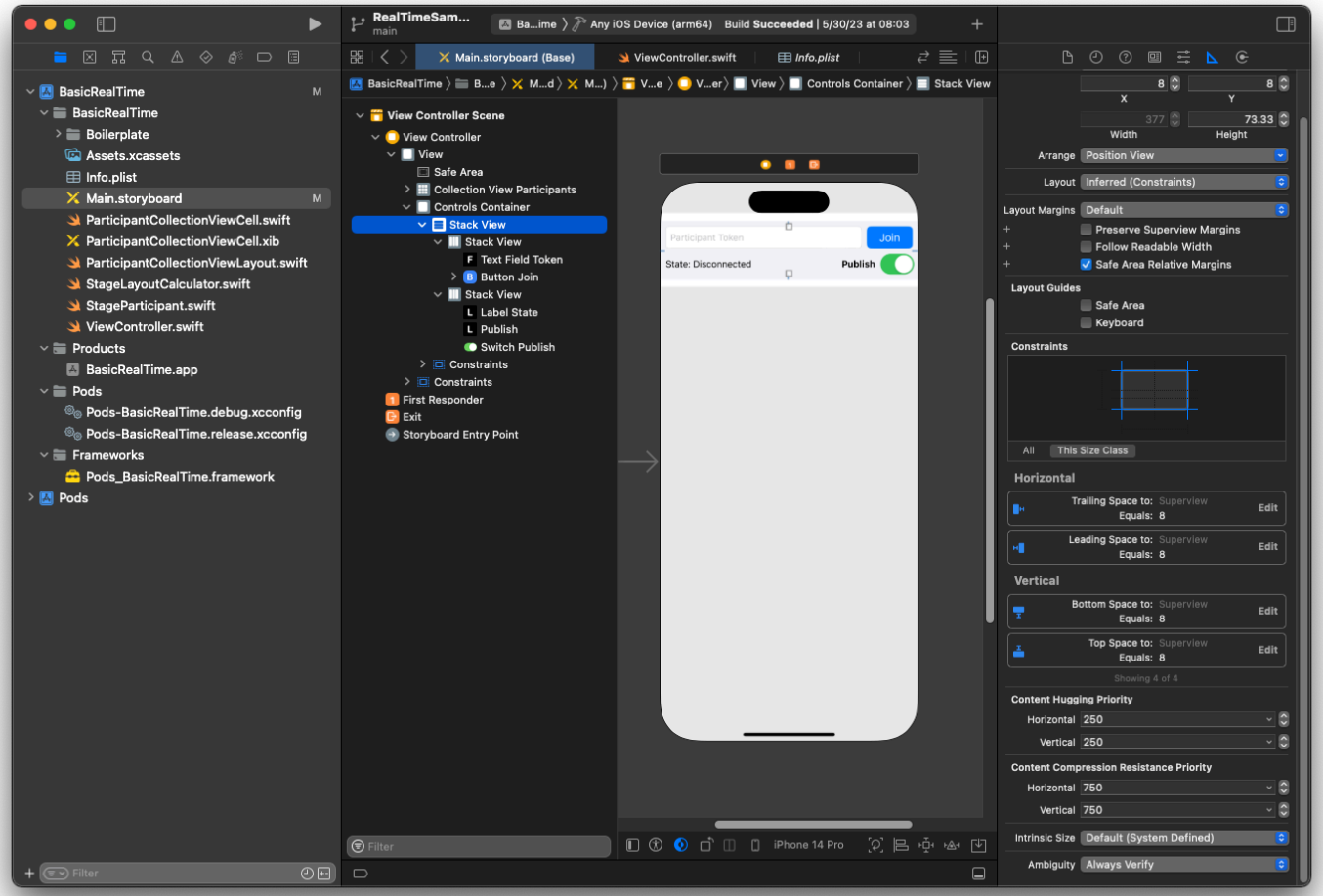
为了进行 AutoLayout 配置，我们需要自定义三个视图。第一个视图是集合视图参与者 (UICollectionView)。将开头、结尾和底部绑定到安全区域。同时将顶部绑定到控件容器。



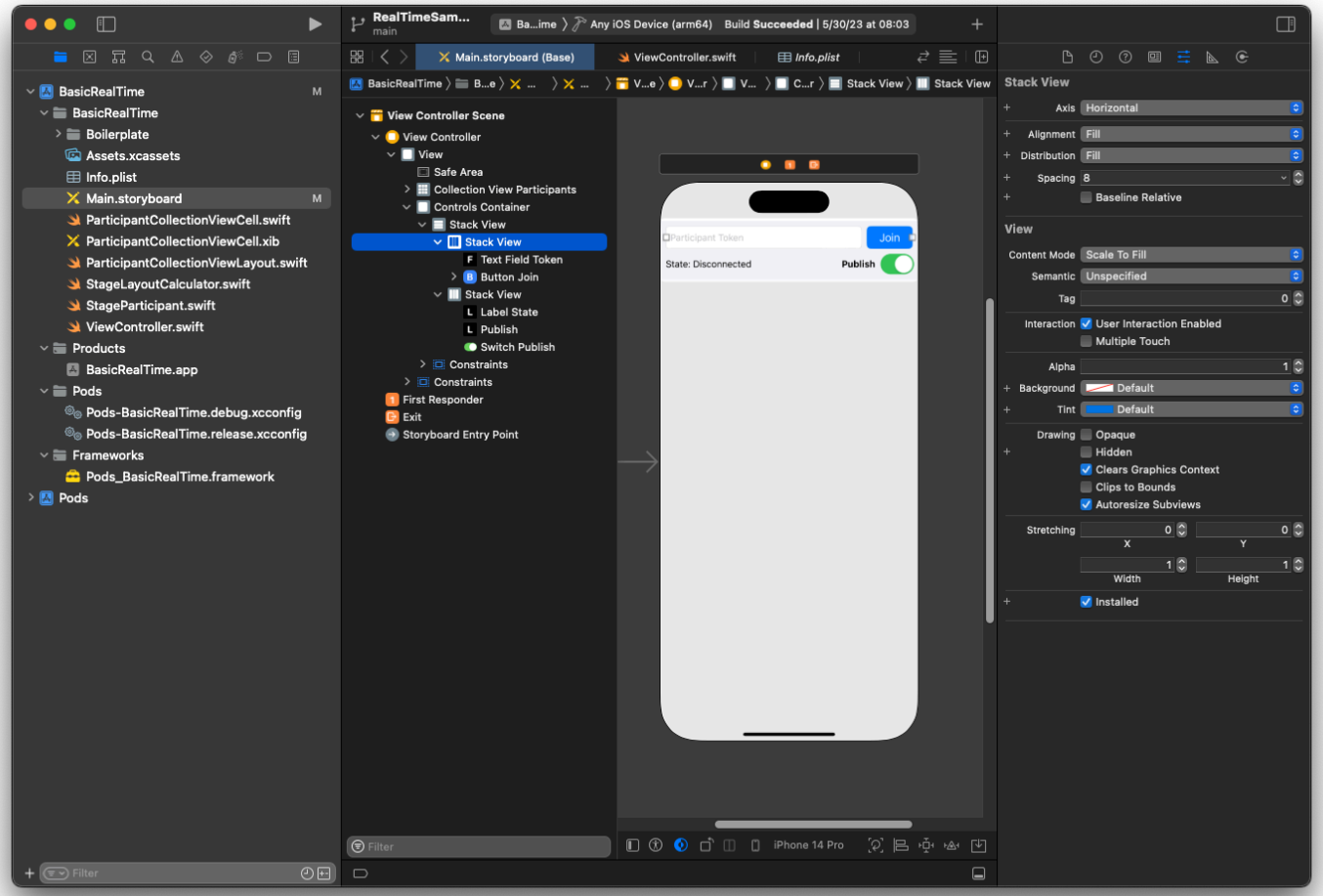
第二个视图是控件容器。将开头、结尾和顶部绑定到安全区域：



第三个也是最后一个视图是垂直堆栈视图。将顶部、开头、结尾和底部绑定到超级视图。对于样式，将间距设置为 8 而不是 0。



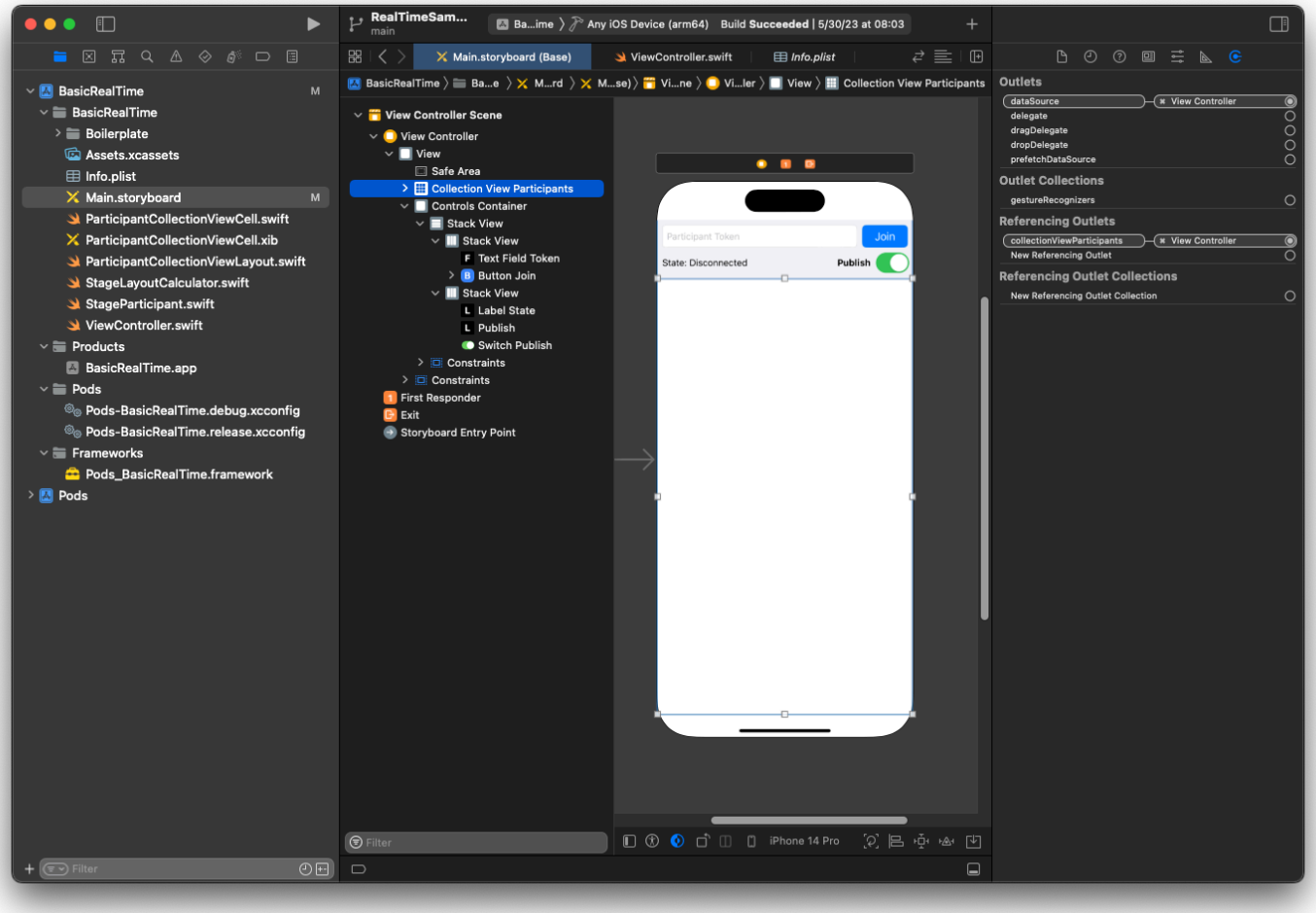
用户界面StackViews将处理其余视图的布局。对于所有三个 UI StackViews，使用填充作为对齐和分布。



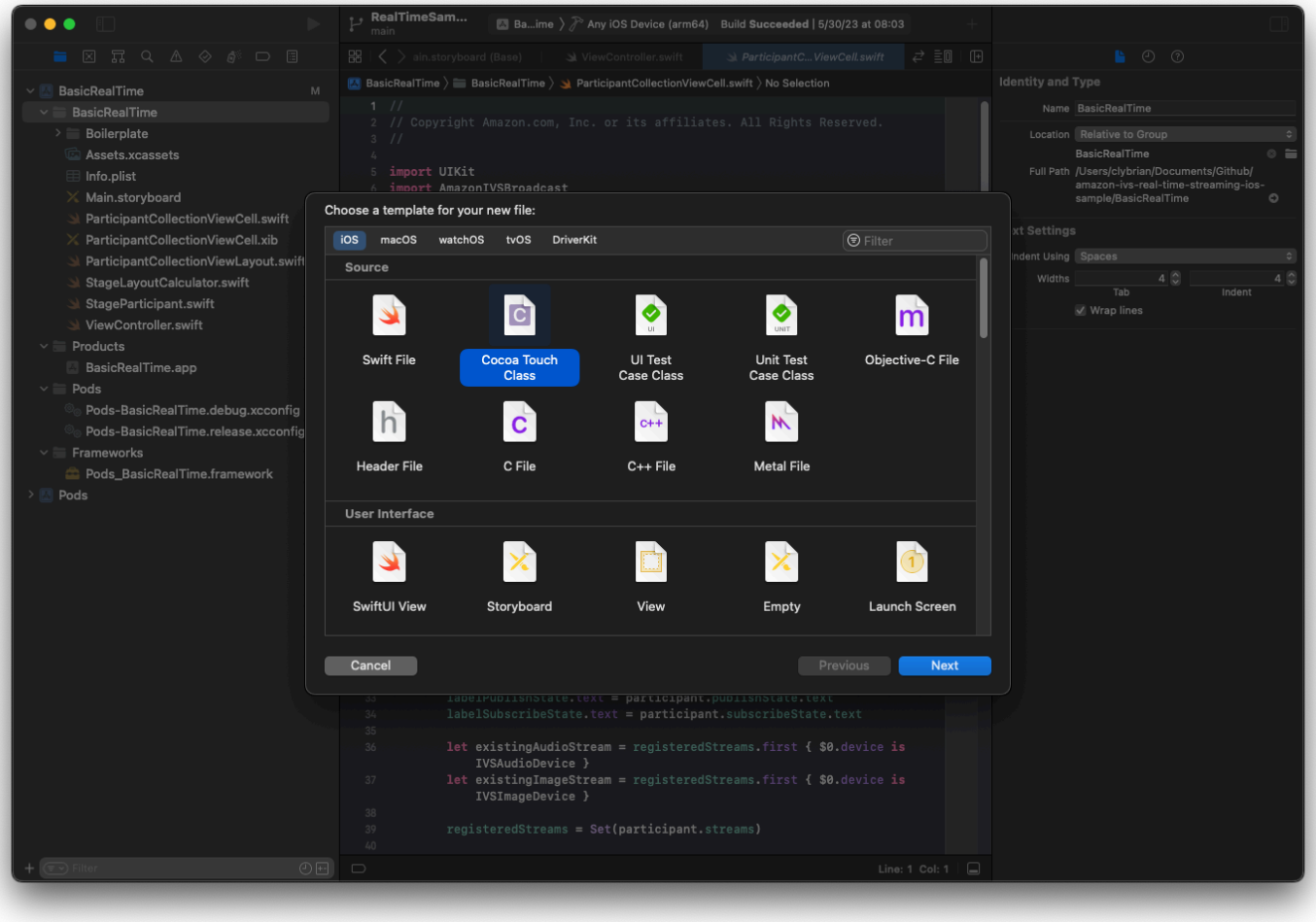
最后，将这些观点链接到 `ViewController`。从上面绘制以下视图：

- 将文本字段加入绑定到 `textFieldToken`。
- 将按钮加入绑定到 `buttonJoin`。
- 将标签状态绑定到 `labelState`。
- 将切换发布绑定到 `switchPublish`。
- 将集合视图参与者绑定到 `collectionViewParticipants`。

也可以利用这段时间将集合视图参与者项的 `dataSource` 设置为所属 `ViewController`：



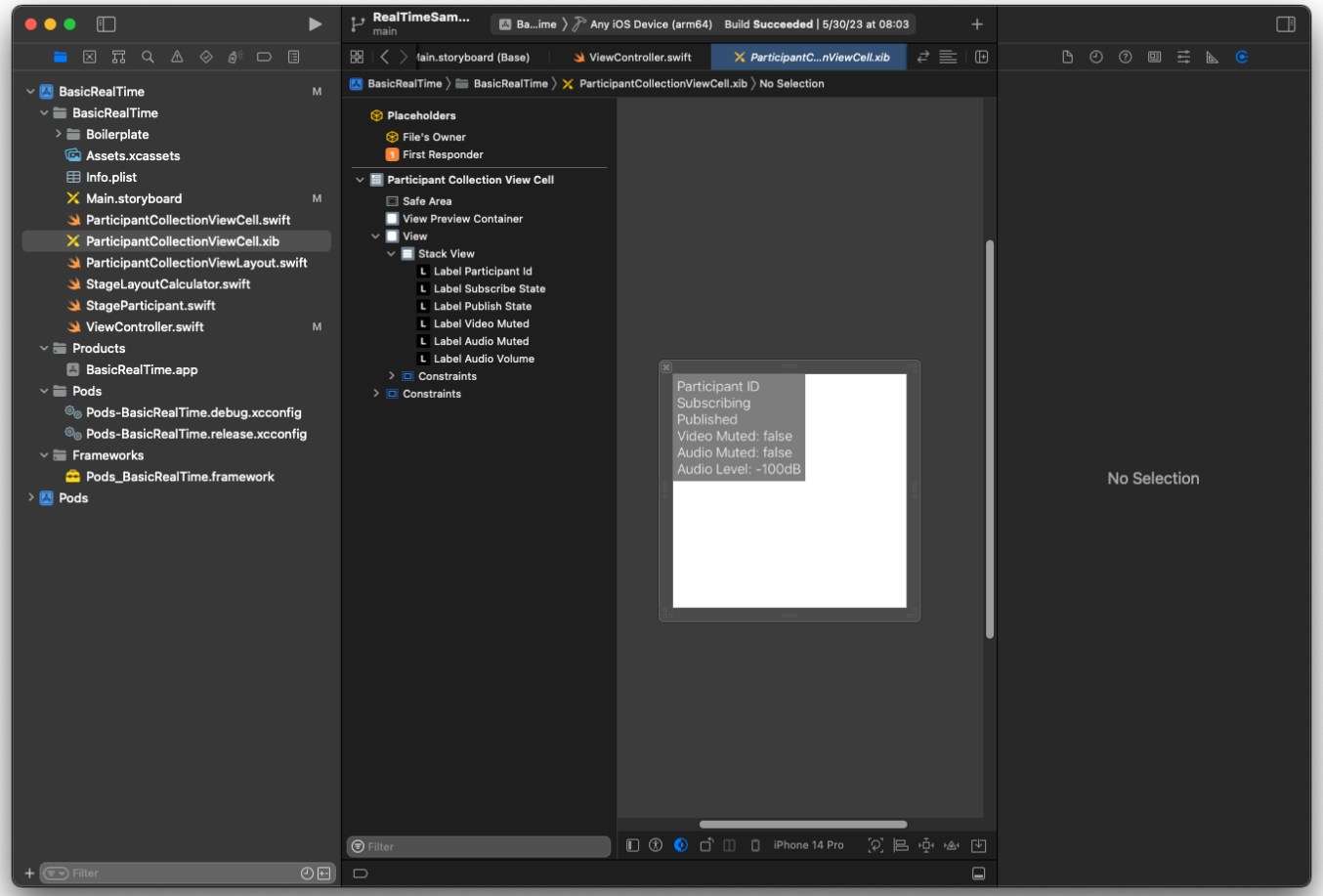
现在创建 `UICollectionViewCell` 子类以在其中渲染参与者。首先创建一个新的 Cocoa Touch 类文件：



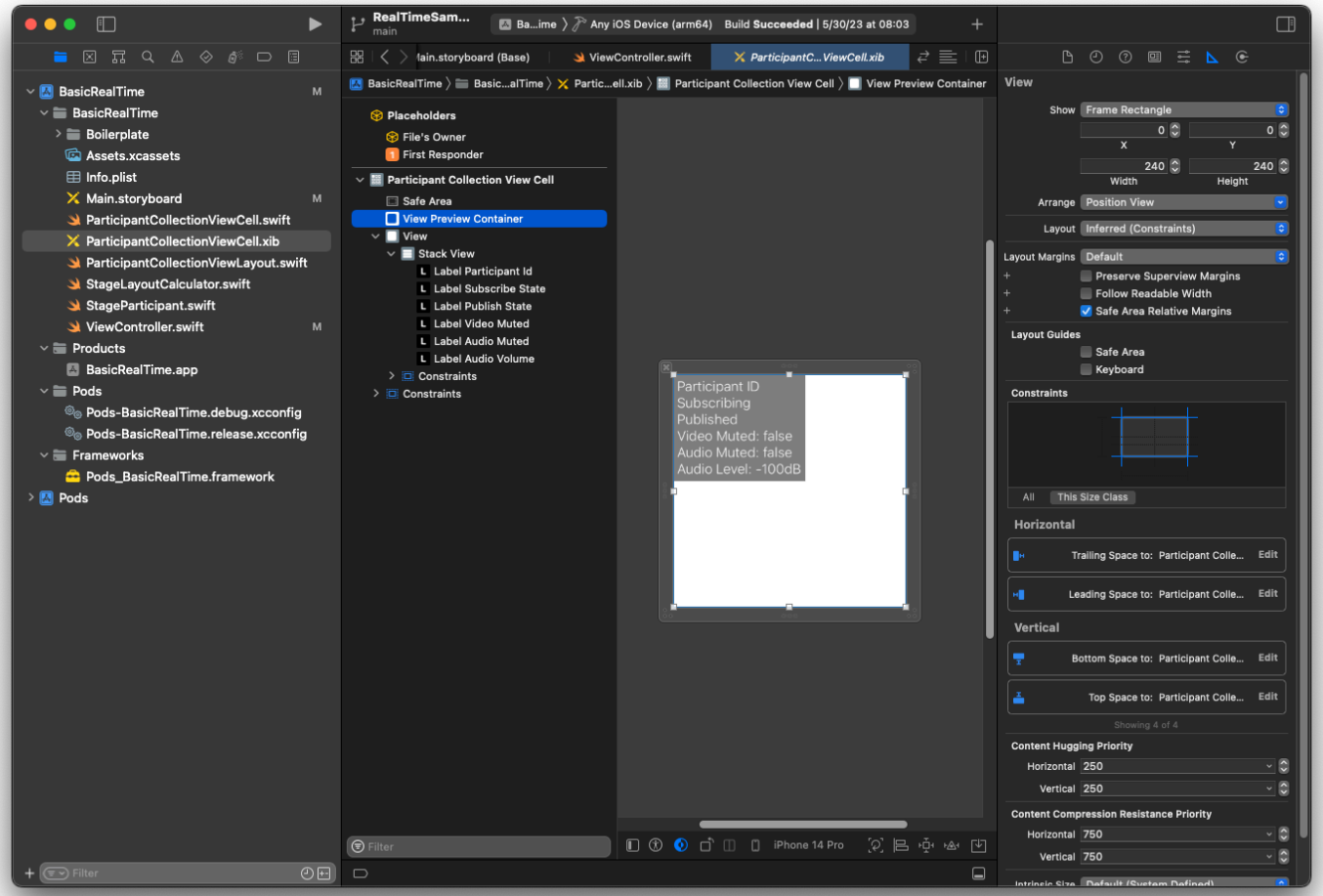
将其命名为 `ParticipantUICollectionViewCell` 并使其成为 Swift 中 `UICollectionViewCell` 的子类。再次从 Swift 文件开始，创建要链接的 `@IBOutlets`：

```
import AmazonIVSBroadcast

class ParticipantCollectionViewCell: UICollectionViewCell {

    @IBOutlet private var viewPreviewContainer: UIView!
    @IBOutlet private var labelParticipantId: UILabel!
    @IBOutlet private var labelSubscribeState: UILabel!
    @IBOutlet private var labelPublishState: UILabel!
    @IBOutlet private var labelVideoMuted: UILabel!
    @IBOutlet private var labelAudioMuted: UILabel!
    @IBOutlet private var labelAudioVolume: UILabel!
```

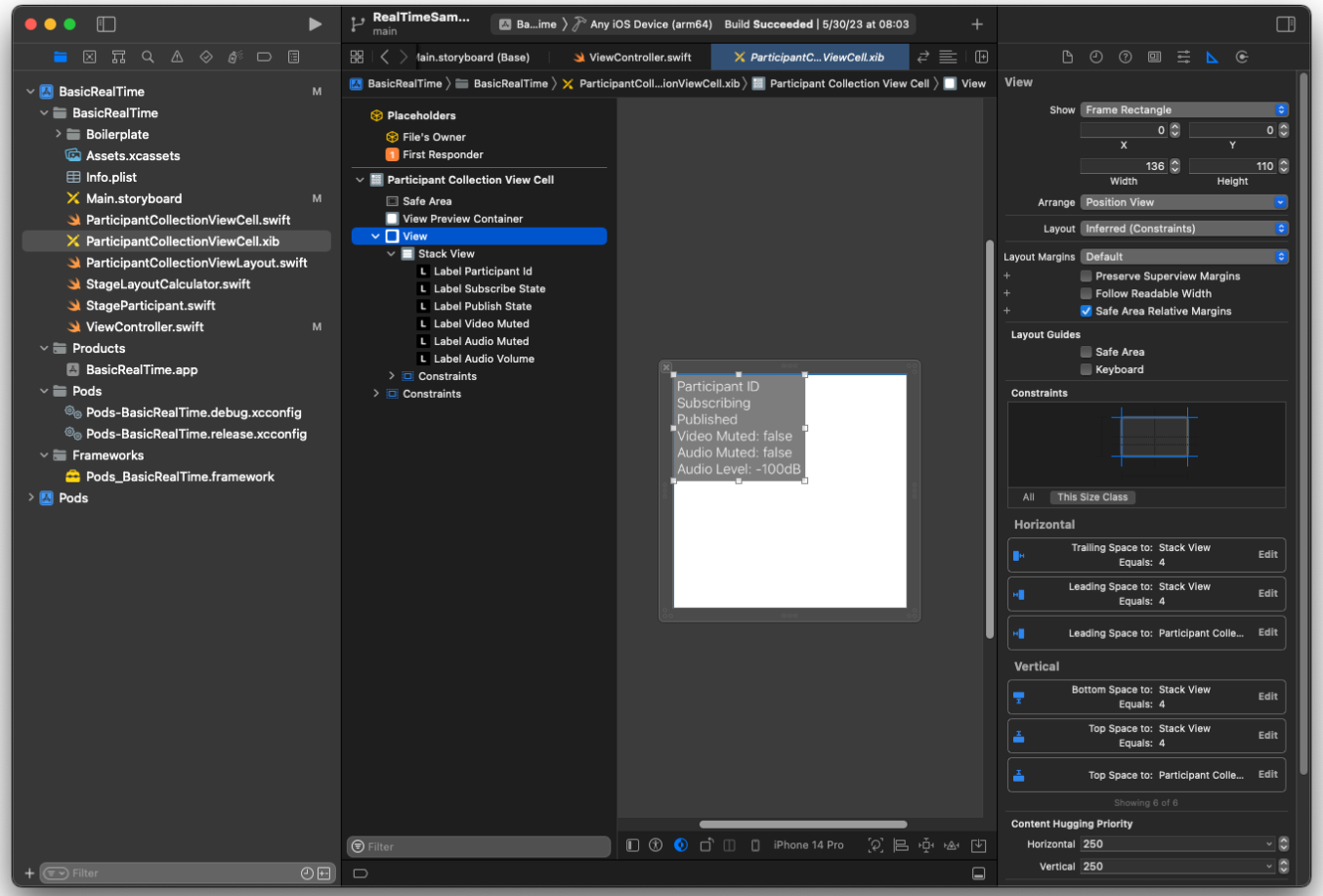
在关联的 XIB 文件中，创建此视图层次结构：



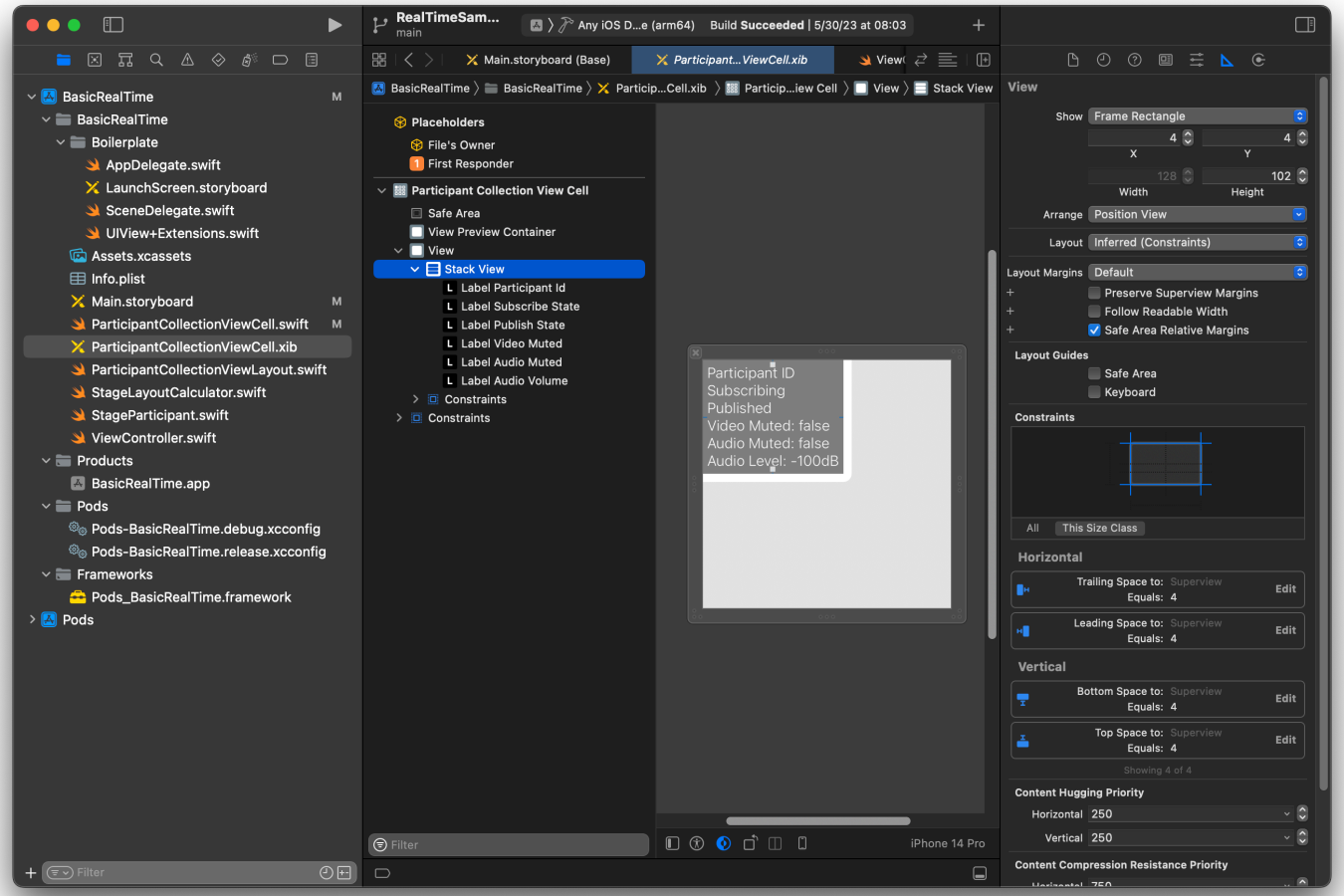
对于 AutoLayout，我们将再次修改三个视图。第一个视图是视图预览容器。将结尾、开头、顶部和底部设置为参与者集合视图单元格。



第二个视图是视图。将开头和顶部设置为参与者集合视图单元格并将值更改为 4。



第三个视图是堆栈视图。将结尾、开头、顶部和底部设置为超级视图并将值更改为 4。



权限和空闲计时器

返回 `ViewController`，禁用系统空闲计时器，以防止设备在使用应用程序时进入睡眠状态：

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    // Prevent the screen from turning off during a call.
    UIApplication.shared.isIdleTimerDisabled = true
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    UIApplication.shared.isIdleTimerDisabled = false
}
```

接下来，向系统请求相机和麦克风权限：

```
private func checkPermissions() {
    checkOrGetPermission(for: .video) { [weak self] granted in
        guard granted else {
            print("Video permission denied")
            return
        }
        self?.checkOrGetPermission(for: .audio) { [weak self] granted in
            guard granted else {
                print("Audio permission denied")
                return
            }
            self?.setupLocalUser() // we will cover this later
        }
    }
}

private func checkOrGetPermission(for mediaType: AVMediaType, _ result: @escaping
 (Bool) -> Void) {
    func mainThreadResult(_ success: Bool) {
        DispatchQueue.main.async {
            result(success)
        }
    }
    switch AVCaptureDevice.authorizationStatus(for: mediaType) {
    case .authorized: mainThreadResult(true)
    case .notDetermined:
        AVCaptureDevice.requestAccess(for: mediaType) { granted in
            mainThreadResult(granted)
        }
    case .denied, .restricted: mainThreadResult(false)
    @unknown default: mainThreadResult(false)
    }
}
```

应用程序状态

需要使用之前创建的布局文件配置 `collectionViewParticipants`：

```
override func viewDidLoad() {
    super.viewDidLoad()
    // We render everything to exactly the frame, so don't allow scrolling.
    collectionViewParticipants.isScrollEnabled = false
```

```
    collectionViewParticipants.register(UINib(nibName: "ParticipantCollectionViewCell",
 bundle: .main), forCellWithReuseIdentifier: "ParticipantCollectionViewCell")
}
```

为了代表每个参与者，创建了一个名为 StageParticipant 的简单结构。这可以包含在 ViewController.swift 文件中，也可以创建一个新文件。

```
import Foundation
import AmazonIVSBroadcast

struct StageParticipant {
    let isLocal: Bool
    var participantId: String?
    var publishState: IVSParticipantPublishState = .notPublished
    var subscribeState: IVSParticipantSubscribeState = .notSubscribed
    var streams: [IVSStageStream] = []

    init(isLocal: Bool, participantId: String?) {
        self.isLocal = isLocal
        self.participantId = participantId
    }
}
```

为了追踪这些参与者，我们将他们的数组作为私有财产保留在 ViewController 中：

```
private var participants = [StageParticipant]()
```

此属性将用于为之前从故事情节链接的 UICollectionViewDataSource 提供支持：

```
extension ViewController: UICollectionViewDataSource {

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection
 section: Int) -> Int {
        return participants.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
 IndexPath) -> UICollectionViewCell {
        if let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
 "ParticipantCollectionViewCell", for: indexPath) as? ParticipantCollectionViewCell {
            cell.set(participant: participants[indexPath.row])
```

```
            return cell
        } else {
            fatalError("Couldn't load custom cell type
 'ParticipantCollectionViewCell'")
        }
    }

}
```

要在加入舞台之前查看自己的预览，立即创建本地参与者：

```
override func viewDidLoad() {
    /* existing UICollectionView code */
    participants.append(StageParticipant(isLocal: true, participantId: nil))
}
```

这会导致在应用程序运行后立即渲染代表本地参与者的参与者单元格。

用户希望在加入舞台之前能够看到自己，所以接下来要实施之前从权限处理代码中调用的 `setupLocalUser()` 方法。将相机和麦克风引用存储为 `IVSLocalStageStream` 对象。

```
private var streams = [IVSLocalStageStream]()
private let deviceDiscovery = IVSDeviceDiscovery()

private func setupLocalUser() {
    // Gather our camera and microphone once permissions have been granted
    let devices = deviceDiscovery.listLocalDevices()
    streams.removeAll()
    if let camera = devices.compactMap({ $0 as? IVSCamera }).first {
        streams.append(IVSLocalStageStream(device: camera))
        // Use a front camera if available.
        if let frontSource = camera.listAvailableInputSources().first(where:
 { $0.position == .front }) {
            camera.setPreferredInputSource(frontSource)
        }
    }
    if let mic = devices.compactMap({ $0 as? IVSMicrophone }).first {
        streams.append(IVSLocalStageStream(device: mic))
    }
    participants[0].streams = streams
    participantsChanged(index: 0, changeType: .updated)
}
```

在这里，通过 SDK 找到设备的相机和麦克风，并将它们存储在本地 streams 对象中，然后将第一个参与者（之前创建的本地参与者）的 streams 数组分配给 streams。最后使用 index 0 和 updated 的 changeType 调用 participantsChanged。此函数是一个帮助程序函数，用于使用精美的动画更新 UICollectionView。它看起来像这样：

```
private func participantsChanged(index: Int, changeType: ChangeType) {
    switch changeType {
    case .joined:
        collectionViewParticipants?.insertItems(at: [IndexPath(item: index, section:
 0)])
    case .updated:
        // Instead of doing reloadItems, just grab the cell and update it ourselves. It
 saves a create/destroy of a cell
        // and more importantly fixes some UI flicker. We disable scrolling so the
 index path per cell
        // never changes.
        if let cell = collectionViewParticipants?.cellForItem(at: IndexPath(item:
 index, section: 0)) as? ParticipantCollectionViewCell {
            cell.set(participant: participants[index])
        }
    case .left:
        collectionViewParticipants?.deleteItems(at: [IndexPath(item: index, section:
 0)])
    }
}
```

暂时不用担心 cell.set；稍后会讨论这个问题，但这就是我们将根据参与者渲染单元格内容的地方。

ChangeType 是简单的枚举：

```
enum ChangeType {
    case joined, updated, left
}
```

最后，要跟踪舞台是否已连接。我们使用简单的 bool 进行跟踪，其将在用户界面自行更新时自动更新。

```
private var connectingOrConnected = false {
    didSet {
        buttonJoin.setTitle(connectingOrConnected ? "Leave" : "Join", for: .normal)
```

```
        buttonJoin.tintColor = connectingOrConnected ? .systemRed : .systemBlue
    }
}
```

实施舞台 SDK

三个核心概念构成了实时功能的基础：舞台、策略和渲染器。设计目标是最大限度地减少构建有效产品所需的客户端逻辑量。

IVS StageStrategy

`IVSStageStrategy` 实施很简单：

```
extension ViewController: IVSStageStrategy {
    func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream] {
        // Return the camera and microphone to be published.
        // This is only called if `shouldPublishParticipant` returns true.
        return streams
    }

    func stage(_ stage: IVSStage, shouldPublishParticipant participant:
 IVSParticipantInfo) -> Bool {
        // Our publish status is based directly on the UISwitch view
        return switchPublish.isOn
    }

    func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
        // Subscribe to both audio and video for all publishing participants.
        return .audioVideo
    }
}
```

总而言之，只有在发布开关处于“打开”位置时才会发布，如果要发布，将发布之前收集的流。最后，对于此示例，我们将始终订阅其他参与者并接收他们的音频和视频。

IVS StageRenderer

考虑到函数的数量，尽管 `IVSStageRenderer` 包含更多的代码，但是它实施起来也相当简单。此渲染器中的一般方法是，SDK 通知参与者发生更改时更新 `participants` 数组。在某些情况下，我们

会以不同的方式处理本地参与者，因为我们决定自己管理这些参与者，这样他们就可以在加入舞台之前看到自己的相机预览。

```
extension ViewController: IVSStageRenderer {

    func stage(_ stage: IVSStage, didChange connectionState: IVSStageConnectionState,
 withError error: Error?) {
        labelState.text = connectionState.text
        connectingOrConnected = connectionState != .disconnected
    }

    func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo) {
        if participant.isLocal {
            // If this is the local participant joining the Stage, update the first
 participant in our array because we
            // manually added that participant when setting up our preview
            participants[0].participantId = participant.participantId
            participantsChanged(index: 0, changeType: .updated)
        } else {
            // If they are not local, add them to the array as a newly joined
 participant.
            participants.append(StageParticipant(isLocal: false, participantId:
 participant.participantId))
            participantsChanged(index: (participants.count - 1), changeType: .joined)
        }
    }

    func stage(_ stage: IVSStage, participantDidLeave participant: IVSParticipantInfo)
 {
        if participant.isLocal {
            // If this is the local participant leaving the Stage, update the first
 participant in our array because
            // we want to keep the camera preview active
            participants[0].participantId = nil
            participantsChanged(index: 0, changeType: .updated)
        } else {
            // If they are not local, find their index and remove them from the array.
            if let index = participants.firstIndex(where: { $0.participantId ==
 participant.participantId }) {
                participants.remove(at: index)
                participantsChanged(index: index, changeType: .left)
            }
        }
```

```
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
publishState: IVSParticipantPublishState) {
        // Update the publishing state of this participant
        mutatingParticipant(participant.participantId) { data in
            data.publishState = publishState
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
subscribeState: IVSParticipantSubscribeState) {
        // Update the subscribe state of this participant
        mutatingParticipant(participant.participantId) { data in
            data.subscribeState = subscribeState
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo,
didChangeMutedStreams streams: [IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
those streams locally
        if participant.isLocal { return }
        // For remote participants, notify the UICollectionView that they have updated.
There is no need to modify
        // the `streams` property on the `StageParticipant` because it is the same
`IVSStageStream` instance. Just
        // query the `isMuted` property again.
        if let index = participants.firstIndex(where: { $0.participantId ==
participant.participantId }) {
            participantsChanged(index: index, changeType: .updated)
        }
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didAdd streams:
[IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
those streams locally
        if participant.isLocal { return }
        // For remote participants, add these new streams to that participant's streams
array.
        mutatingParticipant(participant.participantId) { data in
            data.streams.append(contentsOf: streams)
        }
```

```
    }

    func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didRemove streams:
 [IVSStageStream]) {
        // We don't want to take any action for the local participant because we track
 those streams locally
        if participant.isLocal { return }
        // For remote participants, remove these streams from that participant's
 streams array.
        mutatingParticipant(participant.participantId) { data in
            let oldUrns = streams.map { $0.device.descriptor().urn }
            data.streams.removeAll(where: { stream in
                return oldUrns.contains(stream.device.descriptor().urn)
            })
        }
    }

    // A helper function to find a participant by its ID, mutate that participant, and
 then update the UICollectionView accordingly.
    private func mutatingParticipant(_ participantId: String?, modifier: (inout
 StageParticipant) -> Void) {
        guard let index = participants.firstIndex(where: { $0.participantId ==
 participantId }) else {
            fatalError("Something is out of sync, investigate if this was a sample app
 or SDK issue.")
        }

        var participant = participants[index]
        modifier(&participant)
        participants[index] = participant
        participantsChanged(index: index, changeType: .updated)
    }
}
```

此代码使用扩展将连接状态转换为用户友好文本：

```
extension IVSStageConnectionState {
    var text: String {
        switch self {
        case .disconnected: return "Disconnected"
        case .connecting: return "Connecting"
        case .connected: return "Connected"
        @unknown default: fatalError()
```

```
        }
    }
}
```

实现自定义 UI CollectionViewLayout

安排不同数量的参与者可能很复杂。您希望参与者占据整个父视图的框架，但是不想单独处理每个参与者配置。为了简单起见，将逐步实施 UICollectionViewLayout。

创建另一个新文件 ParticipantCollectionViewLayout.swift，它应该扩展 UICollectionViewLayout。此类将使用另一个名为 StageLayoutCalculator 的类，我们很快就会介绍它。类接收每个参与者的计算框架值，然后生成必要的 UICollectionViewLayoutAttributes 对象。

```
import Foundation
import UIKit

/**
 Code modified from https://developer.apple.com/documentation/uikit/views_and_controls/
collection_views/layouts/customizing_collection_view_layouts?language=objc
 */
class ParticipantCollectionViewLayout: UICollectionViewLayout {

    private let layoutCalculator = StageLayoutCalculator()

    private var contentBounds = CGRect.zero
    private var cachedAttributes = [UICollectionViewLayoutAttributes]()

    override func prepare() {
        super.prepare()

        guard let collectionView = collectionView else { return }

        cachedAttributes.removeAll()
        contentBounds = CGRect(origin: .zero, size: collectionView.bounds.size)

        layoutCalculator.calculateFrames(participantCount:
 collectionView.numberOfItems(inSection: 0),
                                         width: collectionView.bounds.size.width,
                                         height: collectionView.bounds.size.height,
                                         padding: 4)
        .enumerated()
        .forEach { (index, frame) in
```

```
            let attributes = UICollectionViewLayoutAttributes(forCellWith:
IndexPath(item: index, section: 0))
            attributes.frame = frame
            cachedAttributes.append(attributes)
            contentBounds = contentBounds.union(frame)
        }
    }

    override var collectionViewContentSize: CGSize {
        return contentBounds.size
    }

    override func shouldInvalidateLayout(forBoundsChange newBounds: CGRect) -> Bool {
        guard let collectionView = collectionView else { return false }
        return !newBounds.size.equalTo(collectionView.bounds.size)
    }

    override func layoutAttributesForItem(at indexPath: IndexPath) ->
UICollectionViewLayoutAttributes? {
        return cachedAttributes[indexPath.item]
    }

    override func layoutAttributesForElements(in rect: CGRect) ->
[UICollectionViewLayoutAttributes]? {
        var attributesArray = [UICollectionViewLayoutAttributes]()

        // Find any cell that sits within the query rect.
        guard let lastIndex = cachedAttributes.indices.last, let firstMatchIndex =
binSearch(rect, start: 0, end: lastIndex) else {
            return attributesArray
        }

        // Starting from the match, loop up and down through the array until all the
attributes
        // have been added within the query rect.
        for attributes in cachedAttributes[..<firstMatchIndex].reversed() {
            guard attributes.frame.maxY >= rect.minY else { break }
            attributesArray.append(attributes)
        }

        for attributes in cachedAttributes[firstMatchIndex...] {
            guard attributes.frame.minY <= rect.maxY else { break }
            attributesArray.append(attributes)
        }
```

```
        return attributesArray
    }

    // Perform a binary search on the cached attributes array.
    func binSearch(_ rect: CGRect, start: Int, end: Int) -> Int? {
        if end < start { return nil }

        let mid = (start + end) / 2
        let attr = cachedAttributes[mid]

        if attr.frame.intersects(rect) {
            return mid
        } else {
            if attr.frame.maxY < rect.minY {
                return binSearch(rect, start: (mid + 1), end: end)
            } else {
                return binSearch(rect, start: start, end: (mid - 1))
            }
        }
    }
}
```

更重要的是 StageLayoutCalculator.swift 类。此类旨在根据基于流的行/列布局中的参与者数量计算每个参与者的框架。每行的高度与其他行相同，但每行列的宽度各不相同。有关如何自定义该行为的说明，请参阅 layouts 变量上方的代码注释。

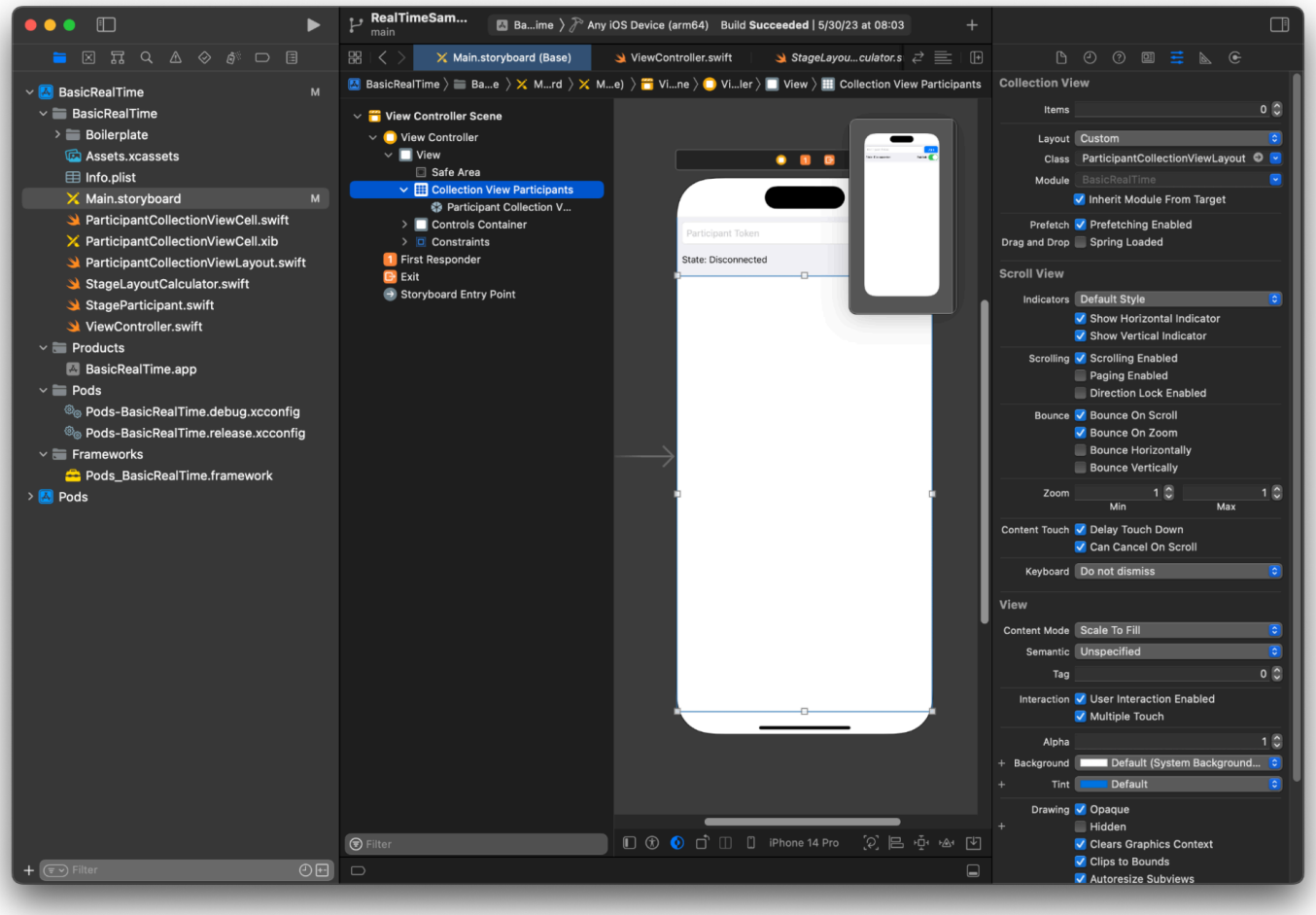
```
import Foundation
import UIKit

class StageLayoutCalculator {

    /// This 2D array contains the description of how the grid of participants should
 be rendered
    /// The index of the 1st dimension is the number of participants needed to active
 that configuration
    /// Meaning if there is 1 participant, index 0 will be used. If there are 5
 participants, index 4 will be used.
    ///
    /// The 2nd dimension is a description of the layout. The length of the array is
 the number of rows that
    /// will exist, and then each number within that array is the number of columns in
 each row.
```

```
    ///
    /// See the code comments next to each index for concrete examples.
    ///
    /// This can be customized to fit any layout configuration needed.
    private let layouts: [[Int]] = [
        // 1 participant
        [ 1 ], // 1 row, full width
        // 2 participants
        [ 1, 1 ], // 2 rows, all columns are full width
        // 3 participants
        [ 1, 2 ], // 2 rows, first row's column is full width then 2nd row's columns
are 1/2 width
        // 4 participants
        [ 2, 2 ], // 2 rows, all columns are 1/2 width
        // 5 participants
        [ 1, 2, 2 ], // 3 rows, first row's column is full width, 2nd and 3rd row's
columns are 1/2 width
        // 6 participants
        [ 2, 2, 2 ], // 3 rows, all column are 1/2 width
        // 7 participants
        [ 2, 2, 3 ], // 3 rows, 1st and 2nd row's columns are 1/2 width, 3rd row's
columns are 1/3rd width
        // 8 participants
        [ 2, 3, 3 ],
        // 9 participants
        [ 3, 3, 3 ],
        // 10 participants
        [ 2, 3, 2, 3 ],
        // 11 participants
        [ 2, 3, 3, 3 ],
        // 12 participants
        [ 3, 3, 3, 3 ],
    ]

    // Given a frame (this could be for a UICollectionView, or a Broadcast Mixer's
canvas), calculate the frames for each
    // participant, with optional padding.
    func calculateFrames(participantCount: Int, width: CGFloat, height: CGFloat,
padding: CGFloat) -> [CGRect] {
        if participantCount > layouts.count {
            fatalError("Only \(layouts.count) participants are supported at this time")
        }
        if participantCount == 0 {
            return []
```

```
        }
        var currentIndex = 0
        var lastFrame: CGRect = .zero

        // If the height is less than the width, the rows and columns will be flipped.
        // Meaning for 6 participants, there will be 2 rows of 3 columns each.
        let isVertical = height > width

        let halfPadding = padding / 2.0

        let layout = layouts[participantCount - 1] // 1 participant is in index 0, so
`-1`.
        let rowHeight = (isVertical ? height : width) / CGFloat(layout.count)

        var frames = [CGRect]()
        for row in 0 ..< layout.count {
            // layout[row] is the number of columns in a layout
            let itemWidth = (isVertical ? width : height) / CGFloat(layout[row])
            let segmentFrame = CGRect(x: (isVertical ? 0 : lastFrame.maxX) +
halfPadding,
                                      y: (isVertical ? lastFrame.maxY : 0) +
halfPadding,
                                      width: (isVertical ? itemWidth : rowHeight) -
padding,
                                      height: (isVertical ? rowHeight : itemWidth) -
padding)

            for column in 0 ..< layout[row] {
                var frame = segmentFrame
                if isVertical {
                    frame.origin.x = (itemWidth * CGFloat(column)) + halfPadding
                } else {
                    frame.origin.y = (itemWidth * CGFloat(column)) + halfPadding
                }
                frames.append(frame)
                currentIndex += 1
            }

            lastFrame = segmentFrame
            lastFrame.origin.x += halfPadding
            lastFrame.origin.y += halfPadding
        }
        return frames
    }
```

```
}
```

返回 Main.storyboard，确保将 UICollectionView 的布局类设置为刚刚创建的类：



挂接 UI 操作

即将完成所有操作，还需要创建几个 IBActions。

首先处理加入按钮。它根据 connectingOrConnected 的值做出不同响应。已连接时，它就会离开舞台。如果断开连接，它会从令牌 UITextField 中读取文本，并使用此 IVSStage 文本创建一个新文本。然后，添加 ViewController 作为 strategy、errorDelegate 和 IVSStage 的渲染器，最后异步加入舞台。

```
@IBAction private func joinTapped(_ sender: UIButton) {
    if connectingOrConnected {
        // If we're already connected to a Stage, leave it.
```

```
        stage?.leave()
    } else {
        guard let token = textFieldToken.text else {
            print("No token")
            return
        }
        // Hide the keyboard after tapping Join
        textFieldToken.resignFirstResponder()
        do {
            // Destroy the old Stage first before creating a new one.
            self.stage = nil
            let stage = try IVSStage(token: token, strategy: self)
            stage.errorDelegate = self
            stage.addRenderer(self)
            try stage.join()
            self.stage = stage
        } catch {
            print("Failed to join stage - \(error)")
        }
    }
}
```

需要挂接的另一个 UI 操作是发布开关：

```
@IBAction private func publishToggled(_ sender: UISwitch) {
    // Because the strategy returns the value of `switchPublish.isOn`, just call
 `refreshStrategy`.
    stage?.refreshStrategy()
}
```

渲染参与者

最后，需要将从 SDK 收到的数据渲染到之前创建的参与者单元格上。我们已经完成了 UICollectionView 逻辑，所以只需要在 ParticipantCollectionViewCell.swift 中实施 set API。

首先添加 empty 函数，然后逐步进行操作：

```
func set(participant: StageParticipant) {

}
```

首先，处理简易状态、参与者 ID、发布状态和订阅状态。对于这些，直接更新 UILabels：

```
labelParticipantId.text = participant.isLocal ? "You (\(participant.participantId ??
 "Disconnected"))" : participant.participantId
labelPublishState.text = participant.publishState.text
labelSubscribeState.text = participant.subscribeState.text
```

发布和订阅枚举的文本属性来自本地扩展：

```
extension IVSParticipantPublishState {
    var text: String {
        switch self {
        case .notPublished: return "Not Published"
        case .attemptingPublish: return "Attempting to Publish"
        case .published: return "Published"
        @unknown default: fatalError()
        }
    }
}

extension IVSParticipantSubscribeState {
    var text: String {
        switch self {
        case .notSubscribed: return "Not Subscribed"
        case .attemptingSubscribe: return "Attempting to Subscribe"
        case .subscribed: return "Subscribed"
        @unknown default: fatalError()
        }
    }
}
```

接下来，更新音频和视频的静音状态。要进入静音状态，需要找到来自 streams 数组的 IVSImageDevice 和 IVSAudioDevice。要优化性能，需要记住最后连接的设备。

```
// This belongs outside `set(participant:)`
private var registeredStreams: Set<IVSStageStream> = []
private var imageDevice: IVSImageDevice? {
    return registeredStreams.lazy.compactMap { $0.device as? IVSImageDevice }.first
}
private var audioDevice: IVSAudioDevice? {
    return registeredStreams.lazy.compactMap { $0.device as? IVSAudioDevice }.first
}
```

```
// This belongs inside `set(participant:)`
let existingAudioStream = registeredStreams.first { $0.device is IVSAudioDevice }
let existingImageStream = registeredStreams.first { $0.device is IVSImageDevice }

registeredStreams = Set(participant.streams)

let newAudioStream = participant.streams.first { $0.device is IVSAudioDevice }
let newImageStream = participant.streams.first { $0.device is IVSImageDevice }

// `isMuted != false` covers the stream not existing, as well as being muted.
labelVideoMuted.text = "Video Muted: \(newImageStream?.isMuted != false)"
labelAudioMuted.text = "Audio Muted: \(newAudioStream?.isMuted != false)"
```

最后，要渲染 imageDevice 的预览并显示 audioDevice 的音频统计数据：

```
if existingImageStream !== newImageStream {
    // The image stream has changed
    updatePreview() // We'll cover this next
}

if existingAudioStream !== newAudioStream {
    (existingAudioStream?.device as? IVSAudioDevice)?.setStatsCallback(nil)
    audioDevice?.setStatsCallback( { [weak self] stats in
        self?.labelAudioVolume.text = String(format: "Audio Level: %.0f dB", stats.rms)
    })
    // When the audio stream changes, it will take some time to receive new stats.
 Reset the value temporarily.
    self.labelAudioVolume.text = "Audio Level: -100 dB"
}
```

要创建的最后一个函数是 updatePreview()，它将参与者的预览添加到视图中：

```
private func updatePreview() {
    // Remove any old previews from the preview container
    viewPreviewContainer.subviews.forEach { $0.removeFromSuperview() }
    if let imageDevice = self.imageDevice {
        if let preview = try? imageDevice.previewView(with: .fit) {
            viewPreviewContainer.addSubviewMatchFrame(preview)
        }
    }
}
```

上面的步骤在 UIView 上使用了帮助程序函数来让嵌入子视图变得更易于操作：

```
extension UIView {
    func addSubviewMatchFrame(_ view: UIView) {
        view.translatesAutoresizingMaskIntoConstraints = false
        self.addSubview(view)
        NSLayoutConstraint.activate([
            view.topAnchor.constraint(equalTo: self.topAnchor, constant: 0),
            view.bottomAnchor.constraint(equalTo: self.bottomAnchor, constant: 0),
            view.leadingAnchor.constraint(equalTo: self.leadingAnchor, constant: 0),
            view.trailingAnchor.constraint(equalTo: self.trailingAnchor, constant: 0),
        ])
    }
}
```

监控 Amazon IVS Real-Time Streaming

什么是舞台会话？

当第一个参与者加入舞台时，舞台会话开始，最后一个参与者停止发布到舞台的几分钟后，舞台会话结束。舞台会话将事件和参与者分成短期会话，帮助调试持续时间较长的舞台。

查看舞台会话和参与者

控制台说明

1. 打开 Amazon IVS 控制台。

   （您还可通过 AWS Management Console 访问 Amazon IVS 控制台。）
2. 在导航窗格中，选择舞台。（如果导航窗格已折叠，请首先选择汉堡包图标以将其打开。）
3. 选择舞台以跳转至该舞台的详细信息页面。
4. 向下滚动页面，直到看到舞台会话部分，然后选择一个舞台会话以查看该舞台的详细信息页面。
5. 要查看会话中的参与者，请向下滚动，直到看到参与者部分，然后选择一个参与者以查看该参与者的详细信息页面，包括 Amazon CloudWatch 指标的图表。

查看参与者的事件

舞台中的参与者状态发生变化（例如加入舞台或在尝试发布到舞台时遇到错误）时，会发送事件。并非所有错误都会导致发生事件；例如，客户端网络错误和令牌签名错误不会作为事件发送。要处理客户端应用程序中的这些错误，请使用 IVS 广播 SDK。

控制台说明

1. 按照上面的说明导航到参与者详细信息页面。
2. 向下滚动，直到看到事件部分。此部分将显示参与者事件的有序列表。请参阅将 Amazon EventBridge 与 Amazon IVS 配合使用，了解为参与者发布的事件的详细信息。

CLI 说明

使用 AWS CLI 访问舞台会话事件是一种高级选项，需要先在计算机上下载并配置 CLI。有关详细信息，请参阅 AWS Command Line Interface 用户指南。

1. 列出所有舞台会话以查找某个舞台会话：

```
aws ivs-realtime list-stage-sessions --stage-arn <arn>
```

2. 列出某个舞台会话的所有参与者以查找某个参与者：

```
aws ivs-realtime list-participants --stage-arn <arn> -session-id <sessionId>
```

3. 列出某个舞台会话和参与者的所有事件：

```
aws ivs-realtime list-participant-events --stage-arn <arn> --session-id <sessionId>
 --participant-id <participantId>
```

以下为 `list-participant-events` 调用的示例响应：

```
{
    "events": [
        {
            "eventTime": "2023-04-04T22:48:41+00:00",
            "name": "JOINED",
            "participantId": "AdRezBl021t0"
        },
        {
            "eventTime": "2023-04-04T22:48:41+00:00",
            "name": "SUBSCRIBE_STARTED",
            "participantId": "AdRezBl021t0",
            "remoteParticipantId": "Ou5b5n5XLMdC"
        },
        {
            "eventTime": "2023-04-04T22:49:45+00:00",
            "name": "SUBSCRIBE_STOPPED",
            "participantId": "AdRezBl021t0",
            "remoteParticipantId": "Ou5b5n5XLMdC"
        },
        {
            "eventTime": "2023-04-04T22:49:45+00:00",
```

```
            "name": "LEFT",
            "participantId": "AdRezBl021t0"
        }
    ]
}
```

访问 CloudWatch 指标

要使 CloudWatch 指标可用，必须使用以下 IVS 广播 SDK 版本：Web 1.5.0 或更高版本、Android 1.12.0 或更高版本、iOS 1.12.0 或更高版本。

CloudWatch 控制台说明

1. 访问 https://console.aws.amazon.com/cloudwatch/ 打开 CloudWatch 控制台。
2. 在侧导航栏中，展开 Metrics（指标）下拉菜单，然后选择 All metrics（所有指标）。
3. 在浏览选项卡上，使用左侧未标记的下拉菜单，选择您的“主”区域，即创建通道的区域。有关区域的详细信息，请参阅全球解决方案，区域控制。有关支持区域的列表，请参阅亚马逊云科技一般参考中的 Amazon IVS 页面。
4. 在浏览选项卡的底部，选择 IVSRealTime 命名空间。
5. 执行下列操作之一：
   a. 在搜索栏中，输入资源 ID（是 ARN `arn:::ivs:stage/<resource id>` 的一部分）。

      然后选择 IVSRealTime > Stage 指标。

   b. 如果 IVSRealTime 显示为 Amazon 命名空间下的一个可选服务，选择该服务。如果您使用 Amazon IVS 实时直播功能并将指标发送给 Amazon CloudWatch，则将会列出 IVS。（如果 IVSRealTime 未列出，则说明您没有任何 Amazon IVS 指标。）

      然后根据需要选择维度分组；可用维度将在下面的 CloudWatch 指标 中列出。
6. 选择要添加到图表的指标。可用维度将在下面的 CloudWatch 指标 中列出。

您还可以从流会话的详细信息页面访问流会话的 CloudWatch 图表，方法是选择 View in CloudWatch（在 CloudWatch 中查看）按钮。

CLI 说明

您也可以使用 Amazon CLI 访问指标。这需要首先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。

然后，使用 Amazon CLI 访问 Amazon IVS 实时直播功能指标：

- 在命令提示符下，运行：

```
aws cloudwatch list-metrics --namespace AWS/IVSRealTime
```

有关更多信息，请参阅 Amazon CloudWatch 用户指南中的使用 Amazon CloudWatch 指标。

CloudWatch 指标：IVS 实时直播功能

Amazon IVS 在 Amazon/IVSRealTime 命名空间中提供了以下指标。

要使 CloudWatch 指标可用，必须使用 Web 广播 SDK 1.5.2 或更高版本。

该维度可能的有效值如下：

- `Stage` 维度是一个资源 ID（ARN `arn:::stage/<resource id>` 的一部分）。
- `Participant` 维度是一个 `participantID`。
- 对于“video”的 `MediaType`，`SimulcastLayer` 为“hi”、“mid”、“low”或“no-rid”，对于“audio”的 `MediaType`，则为“disabled”。该值也可以为空。
- `MediaType` 维度为“视频”或“音频”（字符串）。

指标	维度	说明
`DownloadPacketLoss`	`Stage`	每个样本代表给定订阅用户从 IVS 服务器下载时的丢包百分比。 单位：百分比 有效统计数据：平均值、最大值、最小值 –（分别为）在配置的时间间隔内掉帧的平均数、最大数或最小数。
`DownloadPacketLoss`	`Stage,Participant`	对于同时也是发布者的订阅用户，按参与者筛选 `DownloadPacketLoss` 。样本代表订阅用户从 IVS 服务器下载时的丢包百分比。仅当参与者同时也是发布者时，才会发出样本。 单位：百分比

指标	维度	说明
		有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内丢帧的平均数、最大数或最小数。
DroppedFrames	Stage	每个样本代表给定订阅用户的丢帧百分比。 单位：百分比 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内丢帧的平均数、最大数或最小数。
DroppedFrames	Stage,Participant	对于同时也是发布者的订阅用户，按参与者筛选 DroppedFrames 。样本表示订阅参与者与舞台中所有发布者之间掉帧的百分比。仅当参与者同时也是发布者时，才会发出样本。 单位：百分比 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内丢帧的平均数、最大数或最小数。
PublishBitrate	Stage	发出的样本表示给定发布者发送视频和音频数据的总速率（所有联播层的总和）。 单位：每秒比特数 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内比特率的平均数、最大数或最小数。
PublishBitrate	Stage, Participant, SimulcastLayer, MediaType	PublishBitrate 按参与者、联播层和媒体类型筛选。联播层 ID 由广播 SDK 设置。禁用联播时，此层 ID 将设置为“已禁用”。媒体类型为视频或音频。 单位：每秒比特数 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内比特率的平均数、最大数或最小数。

指标	维度	说明
Publishers	Stage	发布到舞台的参与者人数。 单位：计数 有效统计数据：平均值、最大值、最小值
PublishResolution	Stage, Participant, SimulcastLayer, MediaType	帧宽和帧高两者中较小者的像素数。例如，对于大小为 1920x1080 的横向帧，PublishResolution 为 1080。对于大小为 720x1280 的竖向帧，PublishResolution 为 720。 单位：计数 有效统计数据：平均值、最大值、最小值
SubscribeBitrate	Stage	发出的样本代表给定订阅用户同时接收视频和音频数据的总速率。 单位：每秒比特数 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内比特率的平均数、最大数或最小数。
SubscribeBitrate	Stage,Participant, MediaType	对于同时也是发布者的订阅用户，按参与者筛选 SubscribeBitrate 。样本表示给定订阅用户的接收给定 MediaType 的比特率。仅当订阅参与者发布时才会发出样本。 单位：每秒比特数 有效统计数据：平均值、最大值、最小值 – （分别为）在配置的时间间隔内比特率的平均数、最大数或最小数。
Subscribers	Stage	订阅该舞台的参与者人数。请注意，主动发布和订阅的参与者才被视为发布者和订阅用户。 单位：计数 有效统计数据：平均值、最大值、最小值

IVS 广播 SDK（实时流式传输）

Amazon Interactive Video Services（IVS）实时流式传输广播 SDK 适用于使用 Amazon IVS 构建应用程序的开发人员。此开发工具包旨在利用 Amazon IVS 架构，并将实现 Amazon IVS 的持续改进和新功能。作为本机广播开发工具包，它旨在最大限度地减少对应用程序以及用户有权访问应用程序所在设备的性能影响。

请注意，广播 SDK 用于发送和接收视频；也就是说，您对主机和观众使用相同的 SDK。无需使用单独的玩家 SDK。

您的应用程序可以利用 Amazon IVS 广播开发工具包的主要功能：

- 高质量的流式传输 - 广播开发工具包支持高质量的流式传输。从相机捕获视频并以高达 720p 的分辨率进行编码。
- 自动比特率调整 - 智能手机用户是移动的，因此他们的网络条件会在整个广播过程中发生变化。Amazon IVS 广播开发工具包会自动调整视频比特率，以适应不断变化的网络条件。
- 支持纵向和横向 - 无论您的用户如何持有其设备，图像都会显示为顶部朝上并正确缩放。广播 SDK 支持纵向和横向画布大小。当用户从配置的方向旋转设备时，它会自动管理宽高比。
- 安全流式传输 - 使用 TLS 对用户的广播进行加密，因此他们可以保护其流的安全。
- 外部音频设备 - Amazon IVS 广播开发工具包支持音频插孔、USB 和蓝牙 SCO 外接麦克风。

平台要求

本机平台

平台	受支持的版本
Android	9.0 和更高版本 – 请注意，客户可以使用 5.0 版本进行构建，但不能使用实时流式传输功能。
iOS	14 和更高版本

IVS 至少支持 4 个主要 iOS 版本和 6 个主要 Android 版本。我们当前版本的支持可能会超出这些最低限度。如果主要版本不再受支持，将至少提前 3 个月通过 SDK 发布说明通知客户。

桌面浏览器

浏览器	支持的平台	受支持的版本
Chrome	Windows、macOS	两个主要版本（当前版本和最新版本）
Firefox	Windows、macOS	两个主要版本（当前版本和最新版本）
边缘	（Windows 8.1 和更高版本）	两个主要版本（当前版本和最新版本） 不包括 Edge Legacy
Safari	macOS	两个主要版本（当前版本和最新版本）

移动浏览器（iOS 和 Android）

浏览器	支持的平台	受支持的版本
Chrome	iOS、Android	两个主要版本（当前版本和最新版本）
Firefox	Android	两个主要版本（当前版本和最新版本）
Safari	iOS	两个主要版本（当前版本和最新版本）

已知限制条件

- 在所有移动设备上，由于视频伪影和黑屏问题，我们建议不要同时发布/订阅四个或更多参与者。如果您需要更多参与者，请配置仅限音频发布和订阅。
- 出于性能考虑和可能发生的崩溃，我们建议不要合成舞台并将其广播到 Android 移动网络上的频道。如果需要广播功能，请集成 IVS 实时流式 Android 广播 SDK。

Webviews

Web 广播 SDK 不支持 Webviews 或 Weblike 环境（电视、控制台等）。有关移动实施，请参阅适用于 Android 和 iOS 的 Real-Time Streaming Broadcast SDK Guide。

所需设备访问

广播开发工具包需要访问设备的摄像头和麦克风，包括设备内置的摄像头和麦克风以及通过蓝牙、USB 或音频插孔连接的摄像头和麦克风。

支持

广播 SDK 在不断改进。请参阅 Amazon IVS 发布说明了解可用版本和已修复问题。如果合适，请在联系支持部门之前更新您的广播开发工具包版本，看看这是否解决了您的问题。

版本控制

Amazon IVS 广播开发工具包使用语义化版本。

在此讨论中，假设：

- 最新版本是 4.1.3。
- 先前主要版本的最新版本为 3.2.4。
- 版本 1.x 最新版本是 1.5.6。

最新版本的次要版本已添加向后兼容的新功能。在本例中，版本 4.2.0 已添加新功能。

最新版本的补丁版本已添加向后兼容、次要错误修复。在这里，版本 4.1.4 已添加次要错误修复。

向后兼容、主要错误修复处理方式不同；将在以下几个版本中添加：

- 最新版本补丁版本。在本例中是版本 4.1.4。
- 先前次要版本的补丁版本。在本例中是版本 3.2.5。
- 最新版本 1.x 版本的补丁版本。在本例中是版本 1.5.7。

主要错误修复由 Amazon IVS 产品团队定义。典型示例包括关键安全更新和客户所需的其他选定修复。

注意：在上面的例子中，发布的版本递增但不会跳过任何数字（例如，从 4.1.3 到 4.1.4）。实际上，一个或多个补丁编号可能保留在内部而不发布，因此发布版本可以从 4.1.3 增加到 4.1.6。

IVS 广播 SDK：Web 指南（实时流式传输）

IVS 低延迟实时流式传输 Web 广播 SDK 为开发人员提供了在 Web 上构建交互式实时体验的工具。此 SDK 适用于使用 Amazon IVS 构建 Web 应用程序的开发人员。

Web 广播 SDK 让参与者能够发送和接收视频。SDK 支持以下操作：

- 加入舞台
- 向舞台中的其他参与者发布媒体
- 舞台中其他参与者订阅媒体
- 管理和监控发布到舞台的视频和音频
- 获取每个对等连接的 WebRTC 统计信息
- 所有操作均来自 IVS 低延迟流式传输 Web 广播 SDK

最新版本的网络广播 SDK：1.8.0（版本说明）

参考文档：有关亚马逊 IVS 网络广播软件开发工具包中可用的最重要方法的信息，请参阅 https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference。请确保选择最新版本的 SDK。

示例代码：以下示例可以帮助您快速开始使用 SDK：

- HTML 和 JavaScript
- React

平台要求：有关支持的平台列表，请参阅 Amazon IVS 广播 SDK

开始使用

导入

实时构建基块与根广播模块位于不同的命名空间中。

使用脚本标签

使用相同的脚本导入，可以在全局对象 `IVSBroadcastClient` 上找到以下示例中定义的类和枚举：

```
const { Stage, SubscribeType } = IVSBroadcastClient;
```

使用 npm

也可以从程序包模块中导入类、枚举和类型：

```
import { Stage, SubscribeType, LocalStageStream } from 'amazon-ivs-web-broadcast'
```

请求权限

您的应用程序必须请求权限才能访问用户的摄像头和麦克风，并且必须使用 HTTPS 发送请求。（这不是 Amazon IVS 特有的；需要访问摄像头和麦克风的任何网站都需要请求权限。）

以下示例函数显示了如何请求和获取音频和视频设备的权限：

```
async function handlePermissions() {
   let permissions = {
       audio: false,
       video: false,
   };
   try {
       const stream = await navigator.mediaDevices.getUserMedia({ video: true, audio:
 true });
       for (const track of stream.getTracks()) {
           track.stop();
       }
       permissions = { video: true, audio: true };
   } catch (err) {
       permissions = { video: false, audio: false };
       console.error(err.message);
   }
   // If we still don't have permissions after requesting them display the error
 message
   if (!permissions.video) {
       console.error('Failed to get video permissions.');
   } else if (!permissions.audio) {
       console.error('Failed to get audio permissions.');
   }
}
```

有关更多信息，请参阅权限 API 和MediaDevices。 getUserMedia()。

列出可用的设备

要查看哪些设备可供捕获，请查询浏览器的 MediaDevices.enumerateDevices () 方法：

```
const devices = await navigator.mediaDevices.enumerateDevices();
window.videoDevices = devices.filter((d) => d.kind === 'videoinput');
window.audioDevices = devices.filter((d) => d.kind === 'audioinput');
```

MediaStream 从设备中检索

获取可用设备列表后，您可以从任意数量的设备中检索媒体流。例如，您可以使用 `getUserMedia()` 方法从摄像头中检索媒体流。

如果您想指定从哪个设备捕获媒体流，可以在媒体限制的 `audio` 或 `video` 部分明确设置 `deviceId`。或者，您可以省略 `deviceId`，让用户从浏览器提示中选择他们的设备。

您还可以使用 `width` 和 `height` 限制来指定理想的摄像头分辨率。（请在此处阅读有关这些限制的更多信息。）SDK 会自动应用与您的最大广播分辨率相对应的宽度和高度限制；但是，最好自己也应用这些限制，以确保将源添加到 SDK 后源宽高比不会改变。

要进行实时流式传输，请确保将媒体分辨率限制为 720p。具体而言，宽度和高度的`getUserMedia`和`getDisplayMedia`约束值相乘时不得超过 921600 (1280*720)。

```
const videoConfiguration = {
  maxWidth: 1280,
  maxHeight: 720,
  maxFramerate: 30,
}

window.cameraStream = await navigator.mediaDevices.getUserMedia({
   video: {
       deviceId: window.videoDevices[0].deviceId,
       width: {
           ideal: videoConfiguration.maxWidth,
       },
       height: {
           ideal:videoConfiguration.maxHeight,
       },
   },
});
window.microphoneStream = await navigator.mediaDevices.getUserMedia({
```

```
    audio: { deviceId: window.audioDevices[0].deviceId },
});
```

发布和订阅

概念

三个核心概念构成了实时功能的基础：舞台、策略和事件。设计目标是最大限度地减少构建有效产品所需的客户端逻辑量。

舞台

Stage 类是主机应用程序和 SDK 之间交互的主要点。此类表示舞台，用于加入和退出舞台。创建和加入舞台需要控制面板上有效的未过期令牌字符串（表示为 token）。加入和退出舞台很简单：

```
const stage = new Stage(token, strategy)

try {
    await stage.join();
} catch (error) {
    // handle join exception
}

stage.leave();
```

策略

StageStrategy 接口为主机应用程序提供了一种方法，可以将所需的舞台状态传递给 SDK。需要实现三项函数：shouldSubscribeToParticipant、shouldPublishParticipant 和 stageStreamsToPublish。下面将进行详述。

要使用已定义的策略，请将该策略传递给 Stage 构造函数。以下是应用程序的完整示例，该应用程序使用策略将参与者的网络摄像头发布到舞台并订阅所有参与者。以下部分详细说明了每个必需策略函数的用途。

```
const devices = await navigator.mediaDevices.getUserMedia({
    audio: true,
    video: {
         width: { max: 1280 },
         height: { max: 720 },
     }
```

```
});
const myAudioTrack = new LocalStageStream(devices.getAudioTracks()[0]);
const myVideoTrack = new LocalStageStream(devices.getVideoTracks()[0]);

// Define the stage strategy, implementing required functions
const strategy = {
   audioTrack: myAudioTrack,
   videoTrack: myVideoTrack,

   // optional
   updateTracks(newAudioTrack, newVideoTrack) {
      this.audioTrack = newAudioTrack;
      this.videoTrack = newVideoTrack;
   },

   // required
   stageStreamsToPublish() {
      return [this.audioTrack, this.videoTrack];
   },

   // required
   shouldPublishParticipant(participant) {
      return true;
   },

   // required
   shouldSubscribeToParticipant(participant) {
      return SubscribeType.AUDIO_VIDEO;
   }
};

// Initialize the stage and start publishing
const stage = new Stage(token, strategy);
await stage.join();


// To update later (e.g. in an onClick event handler)
strategy.updateTracks(myNewAudioTrack, myNewVideoTrack);
stage.refreshStrategy();
```

订阅参与者

```
shouldSubscribeToParticipant(participant: StageParticipantInfo): SubscribeType
```

当远程参与者加入舞台，SDK 会向主机应用程序查询该参与者的所需订阅状态。选项为 NONE、AUDIO_ONLY 和 AUDIO_VIDEO。为该函数返回值时，主机应用程序无需担心发布状态、当前订阅状态或舞台连接状态。如果返回 AUDIO_VIDEO，则 SDK 会等到远程参与者发布后再订阅，并在整个过程中通过发射事件来更新主机应用程序。

以下是实施示例：

```
const strategy = {

   shouldSubscribeToParticipant: (participant) => {
      return SubscribeType.AUDIO_VIDEO;
   }

   // ... other strategy functions
}
```

完整实施此功能，适用于始终希望所有参与者都能看到对方的主机应用程序；例如，视频聊天应用程序。

也可以进行更高级的实施。根据服务器提供的属性，使用 ParticipantInfo 上的 userInfo 属性有选择地订阅参与者：

```
const strategy = {

   shouldSubscribeToParticipant(participant) {
      switch (participant.info.userInfo) {
         case 'moderator':
            return SubscribeType.NONE;
         case 'guest':
            return SubscribeType.AUDIO_VIDEO;
         default:
            return SubscribeType.NONE;
      }
   }
   // . . . other strategies properties
}
```

此操作用于创建舞台，在该舞台中，监管人可以监视所有来宾，而不会被来宾看见或听见。主机应用程序可以使用其他业务逻辑，让监管人看到彼此，但对来宾不可见。

发布

```
shouldPublishParticipant(participant: StageParticipantInfo): boolean
```

连接到舞台后，SDK 会查询主机应用程序以查看特定参与者是否应该发布。仅对有权根据提供的令牌进行发布的本地参与者调用此操作。

以下是实施示例：

```
const strategy = {

   shouldPublishParticipant: (participant) => {
      return true;
   }

   // . . . other strategies properties
}
```

适用于用户总想发布的标准视频聊天应用程序。用户可以将音频和视频静音或取消静音，以便立即隐藏或被看见/听见。（他们也可以使用发布/取消发布，但这要慢得多。对于经常需要更改可见性的使用场景，静音/取消静音更可取。）

选择要发布的流

```
stageStreamsToPublish(): LocalStageStream[];
```

这项操作用于在发布时确定应发布的音频和视频流。稍后将在 Publish a Media Stream 中对此进行更详细的介绍。

更新策略

此策略是动态的：可以随时更改从上述任何函数返回的值。例如，如果主机应用程序希望最终用户点击按钮之前不要发布，则可以从 shouldPublishParticipant（类似于 hasUserTappedPublishButton）返回一个变量。当该变量根据最终用户的交互而发生变化时，调用 stage.refreshStrategy() 发送信号到 SDK，表明 SDK 应该查询策略以获取最新值，仅应用已更改的内容。如果 SDK 发现 shouldPublishParticipant 值已更改，则会启动发布流程。如果 SDK 查询和所有函数返回的值与之前相同，则 refreshStrategy 调用不会修改舞台。

如果 shouldSubscribeToParticipant 的返回值从 AUDIO_VIDEO 更改为 AUDIO_ONLY，则如果之前存在视频流，将删除所有返回值已更改的参与者的视频流。

通常，舞台使用该策略来最有效地应用以前和当前策略之间的差异，而主机应用程序无需担心正确管理该策略所需的所有状态。因此，可以将调用 stage.refreshStrategy() 视为一种只需少量计算的操作，因为除非策略发生变化，否则该调用什么都不会做。

事件

Stage 实例是事件发射器。使用 stage.on()，将舞台状态传递给主机应用程序。事件完全可以支持主机应用程序界面的更新。事件如下所示：

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participant, state) =>
 {})
stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participant, state) =>
 {})
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {})
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_REMOVED, (participant, streams) => {})
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {})
```

对于其中大多数事件，提供相应的 ParticipantInfo。

预计事件提供的信息不会影响策略的返回值。例如，调用 STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED 时，shouldSubscribeToParticipant 的返回值预计不会改变。如果主机应用程序想要订阅特定参与者，则无论该参与者的发布状态如何，它都应返回所需的订阅类型。SDK 负责确保根据舞台状态在正确的时间执行策略的期望状态。

发布媒体流

使用与上面 MediaStream 从设备检索中概述的相同步骤检索麦克风和摄像头等本地设备。在示例中，我们使用 MediaStream 创建用于 SDK 发布的 LocalStageStream 对象列表：

```
try {
    // Get stream using steps outlined in document above
    const stream = await getMediaStreamFromDevice();

    let streamsToPublish = stream.getTracks().map(track => {
        new LocalStageStream(track)
    });

    // Create stage with strategy, or update existing strategy
```

```
    const strategy = {
        stageStreamsToPublish: () => streamsToPublish
    }
}
```

发布屏幕共享

除了用户的网络摄像头外，应用程序通常还需要发布屏幕共享。发布屏幕共享需要创建带有唯一令牌的额外 Stage。

```
// Invoke the following lines to get the screenshare's tracks
const media = await navigator.mediaDevices.getDisplayMedia({
   video: {
      width: {
         max: 1280,
      },
      height: {
         max: 720,
      }
   }
});
const screenshare = { videoStream: new LocalStageStream(media.getVideoTracks()[0]) };
const screenshareStrategy = {
   stageStreamsToPublish: () => {
      return [screenshare.videoStream];
   },
   shouldPublishParticipant: (participant) => {
      return true;
   },
   shouldSubscribeToParticipant: (participant) => {
      return SubscribeType.AUDIO_VIDEO;
   }
}
const screenshareStage = new Stage(screenshareToken, screenshareStrategy);
await screenshareStage.join();
```

显示并删除参与者

订阅完成后，您将通过 STAGE_PARTICIPANT_STREAMS_ADDED 事件接收一组 StageStream 对象。该活动还为您提供参与者信息，以在显示媒体流时提供帮助：

```
stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
```

```
    const streamsToDisplay = streams;

    if (participant.isLocal) {
        // Ensure to exclude local audio streams, otherwise echo will occur
        streamsToDisplay = streams.filter(stream => stream.streamType ===
 StreamType.VIDEO)
    }

    // Create or find video element already available in your application
    const videoEl = getParticipantVideoElement(participant.id);

    // Attach the participants streams
    videoEl.srcObject = new MediaStream();
    streamsToDisplay.forEach(stream =>
 videoEl.srcObject.addTrack(stream.mediaStreamTrack));
})
```

当参与者停止发布或取消订阅流时，将使用已删除的流来调用 STAGE_PARTICIPANT_STREAMS_REMOVED 函数。主机应用程序应将其用作信号，从 DOM 中删除参与者的视频流。

在所有可能删除流的场景中都会调用 STAGE_PARTICIPANT_STREAMS_REMOVED，包括：

- 远程参与者停止发布。
- 本地设备取消订阅或将订阅从 AUDIO_VIDEO 更改为 AUDIO_ONLY。
- 远程参与者退出舞台。
- 本地参与者退出舞台。

由于在所有场景中都会调用 STAGE_PARTICIPANT_STREAMS_REMOVED，因此在远程或本地退出操作期间，从用户界面中删除参与者无需自定义业务逻辑。

静音和取消静音媒体流

LocalStageStream 对象具有控制流是否静音的 setMuted 函数。可以在 stageStreamsToPublish 策略函数返回之前或之后在流上调用此函数。

重要提示：如果在调用 refreshStrategy 后 stageStreamsToPublish 返回了新的 LocalStageStream 对象实例，将对舞台应用新流对象的静音状态。创建新 LocalStageStream 实例时要小心，务必保持预期的静音状态。

监控远程参与者媒体静音状态

当参与者更改其视频或音频的静音状态时，已更改的流列表会触发 STAGE_STREAM_MUTE_CHANGED 事件。使用 StageStream 上的 isMuted 属性相应地更新您的用户界面：

```
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {
   if (stream.streamType === 'video' && stream.isMuted) {
       // handle UI changes for video track getting muted
   }
})
```

此外，您还可以查看StageParticipantInfo有关音频还是视频已静音的状态信息：

```
stage.on(StageEvents.STAGE_STREAM_MUTE_CHANGED, (participant, stream) => {
   if (participant.videoStopped || participant.audioMuted) {
       // handle UI changes for either video or audio
   }
})
```

获取 WebRTC 统计信息

要获取发布流或订阅流的最新 WebRTC 统计信息，请使用 StageStream 上的 getStats。这是一种异步方法，您可以使用该方法通过 await 或链接承诺来检索统计信息。您将得到 RTCStatsReport，一个包含所有标准统计信息的字典。

```
try {
   const stats = await stream.getStats();
} catch (error) {
   // Unable to retrieve stats
}
```

优化媒体

为了获得最佳性能，建议对 getUserMedia 和 getDisplayMedia 调用采取以下限制：

```
const CONSTRAINTS = {
    video: {
        width: { ideal: 1280 }, // Note: flip width and height values if portrait is
 desired
        height: { ideal: 720 },
```

```
        framerate: { ideal: 30 },
    },
};
```

您可以通过传递给 LocalStageStream 构造函数的附加选项进一步约束媒体：

```
const localStreamOptions = {
    minBitrate?: number;
    maxBitrate?: number;
    maxFramerate?: number;
    simulcast: {
        enabled: boolean
    }
}
const localStream = new LocalStageStream(track, localStreamOptions)
```

在以上代码中：

- minBitrate 设置浏览器应使用的最小比特率。但是，低复杂度的视频流可能会导致编码器低于此比特率。
- maxBitrate 设置浏览器不应超过的此流的最大比特率。
- maxFramerate 设置浏览器不应超过的此流的最大帧率。
- simulcast 选项仅在基于 Chromium 的浏览器上可用。它允许发送流的三个渲染层。
  - 这允许服务器根据网络限制选择发送给其他参与者的版本。
  - simulcast 与 maxBitrate 和/或 maxFramerate 值一起指定时，预计会根据这些值配置最高渲染层，前提是 maxBitrate 不低于内部 SDK 第二最高层 900 kbps 的默认 maxBitrate 值。
  - 如果与第二最高层的默认值相比，maxBitrate 被定过低，将禁用 simulcast。
  - 如果不通过让 shouldPublishParticipant 返回 false、调用 refreshStrategy、让 shouldPublishParticipant 返回 true 并再次调用 refreshStrategy 的组合操作来重新发布媒体，则无法打开和关闭 simulcast。

获取参与者属性

如果您在 CreateParticipantToken 端点请求中指定属性，则可以在 StageParticipantInfo 属性中看到这些属性：

```
stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
   console.log(`Participant ${participant.id} info:`, participant.attributes);
```

```
})
```

处理网络问题

本地设备的网络连接中断时，SDK 会内部尝试重新连接，无需用户执行任何操作。在某些情况下，SDK 无法重新连接，则需要用户操作。

一般来说，可以通过 STAGE_CONNECTION_STATE_CHANGED 事件来处理舞台状态：

```
stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
   switch (state) {
      case StageConnectionState.DISCONNECTED:
         // handle disconnected UI
         break;
      case StageConnectionState.CONNECTING:
         // handle establishing connection UI
         break;
      case StageConnectionState.CONNECTED:
         // SDK is connected to the Stage
         break;
      case StageConnectionState.ERRORED:
         // unrecoverable error detected, please re-instantiate
         Break;
})
```

通常，成功加入舞台后遇到错误则表明 SDK 已断开连接且未能成功重新建立连接。创建新的 Stage 对象，并在网络条件改善时尝试加入。

将舞台广播到 IVS 通道

要广播舞台，请创建一个单独的 IVSBroadcastClient 会话，然后按照上述用 SDK 进行广播的常规说明进行操作。通过 STAGE_PARTICIPANT_STREAMS_ADDED 公开的 StageStream 列表可用于检索参与者媒体流，这些媒体流可以应用于广播流的构成，如下所示：

```
// Setup client with preferred settings
const broadcastClient = getIvsBroadcastClient();

stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
    streams.forEach(stream => {
        const inputStream = new MediaStream([stream.mediaStreamTrack]);
        switch (stream.streamType) {
            case StreamType.VIDEO:
```

```
                broadcastClient.addVideoInputDevice(inputStream, `video-
${participant.id}`, {
                    index: DESIRED_LAYER,
                    width: MAX_WIDTH,
                    height: MAX_HEIGHT
                });
                break;
            case StreamType.AUDIO:
                broadcastClient.addAudioInputDevice(inputStream, `audio-
${participant.id}`);
                break;
        }
    })
})
```

或者，您可以合成舞台并将其广播到 IVS 低延迟通道，以覆盖更多的观众。请参阅 IVS Low-Latency Streaming User Guide 中的 Enabling Multiple Hosts on an Amazon IVS Stream。

已知问题和解决方法

- 在不调用 `stage.leave()` 的情况下关闭浏览器标签页或退出浏览器时，用户仍然会出现在会话中，伴有长达 10 秒的静帧或黑屏。

  解决办法：尚无。

- 对于会话开始后的用户加入，Safari 会话会间歇性出现黑屏。

  解决方法：刷新浏览器并重新连接会话。

- Safari 无法从切换网络中正常恢复。

  解决方法：刷新浏览器并重新连接会话。

- 开发人员控制台重复出现 `Error: UnintentionalError at StageSocket.onClose` 错误。

  解决方法：每个参与者令牌只能创建一个舞台。当使用相同的参与者令牌创建多个 `Stage` 实例时，无论该实例位于一台设备上还是在多台设备上，都会发生此错误。

- 您可能无法维持`StageParticipantPublishState.PUBLISHED`状态，并且在收听`StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED`事件时可能会收到重复的`StageParticipantPublishState.ATTEMPTING_PUBLISH`状态。

  解决办法：调用`getUserMedia`或时，将视频分辨率限制为 720p。`getDisplayMedia`具体而言，宽度和高度的`getUserMedia`和`getDisplayMedia`约束值相乘时不得超过 921600 (1280*720)。

Safari 限制

- 拒绝权限提示需要在操作系统级别重置 Safari 网站设置中的权限。
- Safari 本身无法像 Firefox 或 Chrome 那样有效地检测所有设备。例如，未检测到 OBS 虚拟摄像头。

Firefox 限制

- Firefox 屏幕共享需要启用系统权限。启用它们后，用户必须重新启动 Firefox 才能正常运行；否则，如果权限被视为被阻止，浏览器将抛出NotFoundError异常。
- 缺少 `getCapabilities` 方法。这意味着用户无法获得媒体轨道的分辨率或宽高比。请参阅此 bugzilla 主题帖。
- 缺少几个 `AudioContext` 属性；例如，延迟和通道数。这可能会给想要操作音轨的高级用户带来问题。
- 在 MacOS 上，来自 `getUserMedia` 的摄像头画面被限制为 4:3 的宽高比。请参阅 bugzilla 主题帖 1 和 bugzilla 主题帖 2。
- `getDisplayMedia` 不支持音频捕获。请参阅此 bugzilla 主题帖。
- 屏幕捕获的帧率不理想（大约 15fps？）。请参阅此 bugzilla 主题帖。

移动 Web 限制

- getDisplayMedia移动设备不支持屏幕共享。

  解决办法：尚无。
- 在不调用 `leave()` 的情况下关闭浏览器时，参与者需要花 15-30 秒才能离开。

  解决办法：增加一个鼓励用户正确断开连接的 UI。
- 后台应用程序会导致发布视频的操作停止。

  解决办法：在发布者暂停时显示 UI 列表。
- 在 Android 设备上取消相机静音后，视频帧率会下降大约 5 秒。

  解决办法：尚无。

- 对于 iOS 16.0，视频源在旋转时会被拉长。

  解决办法：显示概述此已知操作系统问题的 UI。

- 切换音频输入设备会自动切换音频输出设备。

  解决办法：尚无。

- 将浏览器设置为背景会导致发布流变黑并仅产生音频。

  解决办法：尚无。这是出于安全考虑。

错误处理

本节概述错误条件、Web 广播 SDK 如何向应用程序报告错误条件，以及在遇到这些错误时应用程序应采取的措施。有如下四类错误：

```
try {
  stage = new Stage(token, strategy);
} catch (e) {
  // 1) stage instantiation errors
}

try {
  await stage.join();
} catch (e) {
  // 2) stage join errors
}

stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participantInfo, state)
 => {
  if (state === StageParticipantPublishState.ERRORED) {
    // 3) stage publish errors
  }
});

stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participantInfo,
 state) => {
  if (state === StageParticipantSubscribeState.ERRORED) {
    // 4) stage subscribe errors
  }
});
```

阶段实例化错误

阶段实例化不会远程验证令牌，但它会检查一些可以在客户端验证的基本令牌问题。因此，SDK 可能会引发错误。

参与者令牌格式不正确

当阶段令牌格式不正确时，就会发生这种情况。实例化阶段时，SDK 会引发错误并显示以下消息：“解析阶段令牌时出错。”

操作：创建有效令牌并重试实例化。

阶段加入错误

这些是最初尝试加入阶段时可能出现的错误。

已删除阶段

当加入已删除的阶段（与令牌相关联）时，就会发生这种情况。S `join` DK 方法会引发错误，并显示以下消息：“10 秒InitialConnectTimedOut 后”。

操作：使用新阶段创建有效令牌并重试加入。

参与者令牌已过期

当令牌过期时，就会发生这种情况。`join` SDK 方法会引发错误，并显示以下消息：“令牌已过期且不再有效。”

操作：创建新令牌并重试加入。

参与者令牌无效或已撤销

当令牌无效或撤销/断开连接时，就会发生这种情况。S `join` DK 方法会引发错误，并显示以下消息：“10 秒InitialConnectTimedOut 后”。

操作：创建新令牌并重试加入。

令牌已断开连接

当阶段令牌没有格式错误但被阶段服务器拒绝时，就会发生这种情况。S `join` DK 方法会引发错误，并显示以下消息：“10 秒InitialConnectTimedOut 后”。

操作：创建有效令牌并重试加入。

初始加入时出现网络错误

当 SDK 无法联系阶段服务器建立连接时，就会发生这种情况。S join DK 方法会引发错误，并显示以下消息：“10 秒InitialConnectTimedOut 后”。

操作：等待设备连接恢复，然后重试加入。

已加入时出现网络错误

如果设备的网络连接中断，SDK 可能会失去与阶段服务器的连接。您可能会在控制台中看到错误，因为 SDK 无法再访问后端服务。向 https://broadcast.stats.live-video.net 执行 POST 操作会失败。

如果您正在发布和/或订阅，您将在控制台中看到与尝试发布/订阅相关的错误。

在内部，SDK 将尝试使用指数回退策略重新连接。

操作：等待设备连接恢复。如果要发布或订阅，请刷新策略以确保重新发布您的媒体流。

发布和订阅错误

发布错误：发布状态

当发布失败时，SDK 会报告 ERRORED。这可能是由于网络条件或发布者的某个阶段已满负荷而发生的情况。

```
stage.on(StageEvents.STAGE_PARTICIPANT_PUBLISH_STATE_CHANGED, (participantInfo, state)
 => {
  if (state === StageParticipantPublishState.ERRORED) {
      // Handle
  }
});
```

操作：刷新策略以尝试重新发布您的媒体流。

订阅错误

SDK 会在订阅失败时报告 ERRORED。这可能是由于网络条件或订阅者的某个阶段已满负荷而发生的情况。

```
stage.on(StageEvents.STAGE_PARTICIPANT_SUBSCRIBE_STATE_CHANGED, (participantInfo,
 state) => {
  if (state === StageParticipantSubscribeState.ERRORED) {
    // 4) stage subscribe errors
  }
```

```
});
```

操作：刷新策略以尝试新的订阅。

IVS 广播 SDK：Android 指南（实时流式传输）

IVS 实时流式传输 Android 广播 SDK 可让参与者在 Android 设备上发送和接收视频。

`com.amazonaws.ivs.broadcast` 软件包实现了本文档中所描述的接口。SDK 支持以下操作：

- 加入舞台
- 向舞台中的其他参与者发布媒体
- 舞台中其他参与者订阅媒体
- 管理和监控发布到舞台的视频和音频
- 获取每个对等连接的 WebRTC 统计信息
- 所有操作均来自 IVS 低延迟流式传输 Android 广播 SDK

最新版本的安卓广播 SDK：1.14.1（发布说明）

参考文档：有关亚马逊 IVS 安卓广播 SDK 中可用的最重要方法的信息，请参阅参考文档 https://aws.github.io/ amazon-ivs-broadcast-docs /1.14.1/android/。

示例代码：参见 Android 示例存储库，网址为 GitHub：https://github.com/aws-samples/ amazon-ivs-broadcast-android-sample。

平台要求：Android 9.0 及更高版本。

开始使用

安装库

要将 Amazon IVS Android 广播库添加到您的 Android 开发环境中，请将该库添加到您模块的 `build.gradle` 文件，如此处所示（适用于最新版本的 Amazon IVS 广播开发工具包）。

```
repositories {
    mavenCentral()
}

dependencies {
```

```
    implementation 'com.amazonaws:ivs-broadcast:1.14.1:stages@aar'
}
```

在清单中添加以下权限，以允许 SDK 启用和禁用对讲电话：

```
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
```

如要手动安装 SDK，也可从以下位置下载最新版本：

https://search.maven.org/artifact/com.amazonaws/ivs-broadcast

请务必下载附加了 `-stages` 的 `aar`。

请求权限

您的应用必须请求权限才能访问用户摄像头和麦克风。（这并非特定于 Amazon IVS；需要访问摄像头和麦克风的任何应用程序都需要这样做。）

我们在此处检查用户是否已授予权限，如果没有，对他们提出要求：

```
final String[] requiredPermissions =
        { Manifest.permission.CAMERA, Manifest.permission.RECORD_AUDIO };

for (String permission : requiredPermissions) {
    if (ContextCompat.checkSelfPermission(this, permission)
                != PackageManager.PERMISSION_GRANTED) {
        // If any permissions are missing we want to just request them all.
        ActivityCompat.requestPermissions(this, requiredPermissions, 0x100);
        break;
    }
}
```

在这里，我们得到用户的响应：

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode,
            permissions, grantResults);
    if (requestCode == 0x100) {
        for (int result : grantResults) {
```

```
            if (result == PackageManager.PERMISSION_DENIED) {
                return;
            }
        }
        setupBroadcastSession();
    }
}
```

发布和订阅

概念

三个核心概念构成了实时功能的基础：舞台、策略和渲染器。设计目标是最大限度地减少构建有效产品所需的客户端逻辑量。

舞台

Stage 类是主机应用程序和 SDK 之间交互的主要点。此类表示舞台，用于加入和退出舞台。创建和加入舞台需要控制面板上有效的未过期令牌字符串（表示为 token）。加入和退出舞台很简单。

```
Stage stage = new Stage(context, token, strategy);

try {
 stage.join();
} catch (BroadcastException exception) {
 // handle join exception
}

stage.leave();
```

也可以将 StageRenderer 附加到 Stage 类：

```
stage.addRenderer(renderer); // multiple renderers can be added
```

策略

Stage.Strategy 接口为主机应用程序提供了一种方法，可以将所需的舞台状态传递给 SDK。需要实现三项函数：shouldSubscribeToParticipant、shouldPublishFromParticipant 和 stageStreamsToPublishForParticipant。下面将进行详述。

订阅参与者

```
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo);
```

当远程参与者加入舞台，SDK 会向主机应用程序查询该参与者的所需订阅状态。选项为 NONE、AUDIO_ONLY 和 AUDIO_VIDEO。为该函数返回值时，主机应用程序无需担心发布状态、当前订阅状态或舞台连接状态。如果返回 AUDIO_VIDEO，则 SDK 会等到远程参与者发布后再订阅，并在整个过程中通过渲染器更新主机应用程序。

以下是实施示例：

```
@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 return Stage.SubscribeType.AUDIO_VIDEO;
}
```

完整实施此功能，适用于始终希望所有参与者都能看到对方的主机应用程序；例如，视频聊天应用程序。

也可以进行更高级的实施。根据服务器提供的属性，使用 ParticipantInfo 上的 userInfo 属性有选择地订阅参与者：

```
@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 switch(participantInfo.userInfo.get(“role”)) {
  case “moderator”:
   return Stage.SubscribeType.NONE;
  case “guest”:
   return Stage.SubscribeType.AUDIO_VIDEO;
  default:
   return Stage.SubscribeType.NONE;
 }
}
```

此操作用于创建舞台，在该舞台中，监管人可以监视所有来宾，而不会被来宾看见或听见。主机应用程序可以使用其他业务逻辑，让监管人看到彼此，但对来宾不可见。

发布

```
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo);
```

连接到舞台后，SDK 会查询主机应用程序以查看特定参与者是否应该发布。仅对有权根据提供的令牌进行发布的本地参与者调用此操作。

以下是实施示例：

```
@Override
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 return true;
}
```

适用于用户总想发布的标准视频聊天应用程序。用户可以将音频和视频静音或取消静音，以便立即隐藏或被看见/听见。（他们也可以使用发布/取消发布，但这要慢得多。对于经常需要更改可见性的使用场景，静音/取消静音更可取。）

选择要发布的流

```
@Override
List<LocalStageStream> stageStreamsToPublishForParticipant(@NonNull Stage stage,
 @NonNull ParticipantInfo participantInfo);
}
```

这项操作用于在发布时确定应发布的音频和视频流。稍后将在 Publish a Media Stream 中对此进行更详细的介绍。

更新策略

此策略是动态的：可以随时更改从上述任何函数返回的值。例如，如果主机应用程序希望最终用户点击按钮之前不要发布，则可以从 `shouldPublishFromParticipant`（类似于 `hasUserTappedPublishButton`）返回一个变量。当该变量根据最终用户的交互而发生变化时，调用 `stage.refreshStrategy()` 发送信号到 SDK，表明 SDK 应该查询策略以获取最新值，仅应用已更改的内容。如果 SDK 发现 `shouldPublishFromParticipant` 值已更改，它将启动发布流程。如果 SDK 查询和所有函数返回的值与之前相同，则 `refreshStrategy` 调用将不会对舞台进行任何修改。

如果 shouldSubscribeToParticipant 的返回值从 AUDIO_VIDEO 更改为 AUDIO_ONLY，则如果之前存在视频流，将删除所有返回值已更改的参与者的视频流。

通常，舞台使用该策略来最有效地应用以前和当前策略之间的差异，而主机应用程序无需担心正确管理该策略所需的所有状态。因此，可以将调用 stage.refreshStrategy() 视为一种只需少量计算的操作，因为除非策略发生变化，否则该调用什么都不会做。

渲染器

StageRenderer 接口将舞台状态传递给主机应用程序。渲染器提供的事件通常完全可以支持主机应用程序界面的更新。渲染器提供以下函数：

```
void onParticipantJoined(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo);

void onParticipantLeft(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo);

void onParticipantPublishStateChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull Stage.PublishState publishState);

void onParticipantSubscribeStateChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull Stage.SubscribeState subscribeState);

void onStreamsAdded(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo,
 @NonNull List<StageStream> streams);

void onStreamsRemoved(@NonNull Stage stage, @NonNull ParticipantInfo participantInfo,
 @NonNull List<StageStream> streams);

void onStreamsMutedChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull List<StageStream> streams);

void onError(@NonNull BroadcastException exception);

void onConnectionStateChanged(@NonNull Stage stage, @NonNull Stage.ConnectionState
 state, @Nullable BroadcastException exception);
```

对于其中大多数方法，提供相应的 Stage 和 ParticipantInfo。

预计渲染器提供的信息不会影响策略的返回值。例如，调用 onParticipantPublishStateChanged 时，shouldSubscribeToParticipant 的返回值预计

不会改变。如果主机应用程序想要订阅特定参与者，则无论该参与者的发布状态如何，它都应返回所需的订阅类型。SDK 负责确保根据舞台状态在正确的时间执行策略的期望状态。

可以将 StageRenderer 附加到舞台类：

```
stage.addRenderer(renderer); // multiple renderers can be added
```

请注意，只有发布参与者才会触发 onParticipantJoined，每当参与者停止发布或退出舞台会话时，都会触发 onParticipantLeft。

发布媒体流

通过 DeviceDiscovery 发现内置麦克风和摄像头等本地设备。以下示例演示如何选择前置摄像头和默认麦克风，然后将它们作为 LocalStageStreams 返回，由 SDK 发布：

```
DeviceDiscovery deviceDiscovery = new DeviceDiscovery(context);

List<Device> devices = deviceDiscovery.listLocalDevices();
List<LocalStageStream> publishStreams = new ArrayList<LocalStageStream>();

Device frontCamera = null;
Device microphone = null;

// Create streams using the front camera, first microphone
for (Device device : devices) {
 Device.Descriptor descriptor = device.getDescriptor();
 if (!frontCamera && descriptor.type == Device.Descriptor.DeviceType.Camera &&
 descriptor.position = Device.Descriptor.Position.FRONT) {
  front Camera = device;
 }
 if (!microphone && descriptor.type == Device.Descriptor.DeviceType.Microphone) {
  microphone = device;
 }
}

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera);
AudioLocalStageStream microphoneStream = new AudioLocalStageStream(microphoneDevice);

publishStreams.add(cameraStream);
publishStreams.add(microphoneStream);

// Provide the streams in Stage.Strategy
```

```
@Override
@NonNull List<LocalStageStream> stageStreamsToPublishForParticipant(@NonNull Stage
 stage, @NonNull ParticipantInfo participantInfo) {
 return publishStreams;
}
```

显示并删除参与者

订阅完成后，您将通过渲染器的 onStreamsAdded 函数接收一组 StageStream 对象。您可以从 ImageStageStream 检索预览：

```
ImagePreviewView preview = ((ImageStageStream)stream).getPreview();

// Add the view to your view hierarchy
LinearLayout previewHolder = findViewById(R.id.previewHolder);
preview.setLayoutParams(new LinearLayout.LayoutParams(
  LinearLayout.LayoutParams.MATCH_PARENT,
  LinearLayout.LayoutParams.MATCH_PARENT));
previewHolder.addView(preview);
```

您可以从 AudioStageStream 检索音频级别的统计信息：

```
((AudioStageStream)stream).setStatsCallback((peak, rms) -> {
 // handle statistics
});
```

当参与者停止发布或取消订阅时，将使用已删除的流来调用 onStreamsRemoved 函数。主机应用程序应将其用作信号，从视图层次结构中删除参与者的视频流。

在所有可能删除流的场景中都会调用 onStreamsRemoved，包括：

- 远程参与者停止发布。
- 本地设备取消订阅或将订阅从 AUDIO_VIDEO 更改为 AUDIO_ONLY。
- 远程参与者退出舞台。
- 本地参与者退出舞台。

由于在所有场景中都会调用 onStreamsRemoved，因此在远程或本地退出操作期间，从用户界面中删除参与者无需自定义业务逻辑。

静音和取消静音媒体流

LocalStageStream 对象具有控制流是否静音的 setMuted 函数。可以在 streamsToPublishForParticipant 策略函数返回之前或之后在流上调用此函数。

重要提示：如果在调用 refreshStrategy 后 streamsToPublishForParticipant 返回了新的 LocalStageStream 对象实例，将对舞台应用新流对象的静音状态。创建新 LocalStageStream 实例时要小心，务必保持预期的静音状态。

监控远程参与者媒体静音状态

当参与者更改其视频或音频流的静音状态时，将使用已更改的流列表调用渲染器 onStreamMutedChanged 函数。使用 StageStream 上的 getMuted 方法相应地更新您的用户界面。

```
@Override
void onStreamsMutedChanged(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo, @NonNull List<StageStream> streams) {
 for (StageStream stream : streams) {
  boolean muted = stream.getMuted();
  // handle UI changes
 }
}
```

获取 WebRTC 统计信息

要获取发布流或订阅流的最新 WebRTC 统计信息，请使用 StageStream 上的 requestRTCStats。收集完成后，您将通过 StageStream.Listener（可在 StageStream 上设置）收到统计信息。

```
stream.requestRTCStats();

@Override
void onRTCStats(Map<String, Map<String, String>> statsMap) {
 for (Map.Entry<String, Map<String, string>> stat : statsMap.entrySet()) {
  for(Map.Entry<String, String> member : stat.getValue().entrySet()) {
   Log.i(TAG, stat.getKey() + " has member " + member.getKey() + " with value " +
 member.getValue());
  }
 }
}
```

获取参与者属性

如果您在 CreateParticipantToken 端点请求中指定属性，则可以在 ParticipantInfo 属性中看到这些属性：

```
@Override
void onParticipantJoined(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 for (Map.Entry<String, String> entry : participantInfo.userInfo.entrySet()) {
  Log.i(TAG, “attribute: “ + entry.getKey() + “ = “ + entry.getValue());
 }
}
```

在后台继续会话

应用程序进入后台时，您可能需要停止发布或仅订阅其他远程参与者的音频。要实现此目的，请更新 Strategy 实施以停止发布，然后订阅 AUDIO_ONLY（或者 NONE，如果适用）。

```
// Local variables before going into the background
boolean shouldPublish = true;
Stage.SubscribeType subscribeType = Stage.SubscribeType.AUDIO_VIDEO;

// Stage.Strategy implementation
@Override
boolean shouldPublishFromParticipant(@NonNull Stage stage, @NonNull ParticipantInfo
 participantInfo) {
 return shouldPublish;
}

@Override
Stage.SubscribeType shouldSubscribeToParticipant(@NonNull Stage stage, @NonNull
 ParticipantInfo participantInfo) {
 return subscribeType;
}

// In our Activity, modify desired publish/subscribe when we go to background, then
 call refreshStrategy to update the stage
@Override
void onStop() {
 super.onStop();
 shouldPublish = false;
 subscribeTpye = Stage.SubscribeType.AUDIO_ONLY;
```

```
 stage.refreshStrategy();
}
```

通过联播启用/禁用分层编码

发布媒体流时，SDK 会传输高质量和低质量的视频流，因此即使下行带宽有限，远程参与者也可以订阅此流。默认情况下，联播分层编码为启用状态。您可以使用 StageVideoConfiguration.Simulcast 类将其禁用：

```
// Disable Simulcast
StageVideoConfiguration config = new StageVideoConfiguration();
config.simulcast.setEnabled(false);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

视频配置限制

SDK 不支持使用 StageVideoConfiguration.setSize(BroadcastConfiguration.Vec2 size) 强制纵向模式或横向模式。在纵向中，较小的尺寸为宽度；在横向中，较小的尺寸为高度。这意味着以下两个对 setSize 的调用会对视频配置产生相同的影响：

```
StageVideo Configuration config = new StageVideo Configuration();

config.setSize(BroadcastConfiguration.Vec2(720f, 1280f);
config.setSize(BroadcastConfiguration.Vec2(1280f, 720f);
```

处理网络问题

本地设备的网络连接中断时，SDK 会内部尝试重新连接，无需用户执行任何操作。在某些情况下，SDK 无法重新连接，则需要用户操作。有两个与网络连接中断有关的主要错误：

- 错误代码 1400，消息：“PeerConnection 由于未知网络错误而丢失”
- 错误代码 1300，消息：“重试次数已用完”

如果收到第一个错误但没有收到第二个错误，则 SDK 仍在连接该舞台，并将尝试自动重新建立连接。作为一种保护措施，您可以在不更改策略方法的返回值的情况下调用 refreshStrategy，以触发手动重新连接。

如果收到第二个错误，则 SDK 的重新连接尝试已失败，本地设备不再连接到舞台。在这种情况下，请尝试在重新建立网络连接后调用 `join`，以重新加入舞台。

通常，成功加入舞台后遇到错误则表明 SDK 未能成功重新建立连接。创建新的 `Stage` 对象，并在网络条件改善时尝试加入。

使用蓝牙麦克风

要使用蓝牙麦克风设备进行发布，必须启动蓝牙 SCO 连接：

```
Bluetooth.startBluetoothSco(context);
// Now bluetooth microphones can be used
…
// Must also stop bluetooth SCO
Bluetooth.stopBluetoothSco(context);
```

已知问题和解决方法

- 当 Android 设备进入睡眠状态然后唤醒时，预览可能会处于冻结状态。

  解决方法：创建并使用新的 `Stage`。
- 当一个参与者使用另一个参与者正在使用的令牌加入时，第一个连接将断开，不会出现具体错误提示。

  解决办法：尚无。
- 有一个问题比较少见，即发布者正在发布，但订阅用户收到的发布状态是 `inactive`。

  解决方法：尝试退出然后加入会话。如果问题仍然存在，请为发布者创建新令牌。
- 在舞台会话期间，通常在持续时间较长的通话中，可能会间歇性地出现罕见的音频失真问题。

  解决方法：遇到音频失真问题的参与者可以退出并重新加入会话，也可以取消发布并重新发布音频，以修复问题。
- 发布到舞台时不支持外接麦克风。

  解决方法：不要使用通过 USB 连接的外接麦克风发布到舞台。
- 不支持使用 `createSystemCaptureSources` 屏幕共享发布到舞台。

  解决方法：使用自定义图像输入源和自定义音频输入源手动管理系统捕获。

- 当从父级中删除 ImagePreviewView 时（例如，在父级调用 removeView()），会立即释放 ImagePreviewView。将其添加到另一个父视图时，ImagePreviewView 不显示任何帧。

  解决方法：使用 getPreview 请求再次预览。
- 使用搭载 Android 12 的 Samsung Galaxy S22/+ 加入舞台时，可能会遇到 1401 错误，显示本地设备无法加入舞台或加入舞台但没有音频。

  解决方法：升级到 Android 13。
- 在 Android 13 上使用 Nokia X20 加入舞台时，可能无法打开相机并引发异常。

  解决办法：尚无。
- 采用 MediaTek Helio 芯片组的设备可能无法正确呈现远程参与者的视频。

  解决办法：尚无。
- 在少数设备上，设备操作系统选择的麦克风可能与通过 SDK 选择的麦克风不同。这是因为 Amazon IVS 广播 SDK 无法控制 VOICE_COMMUNICATION 音频路由的定义方式，因为定义方式因不同的设备制造商而异。

  解决办法：尚无。
- 某些 Android 视频编码器不能配置小于 176x176 的视频大小。配置较小的尺寸会导致错误并阻止流式传输。

  解决办法：请勿将视频大小配置为小于 176x176。

错误处理

致命错误与非致命错误

错误对象带有值为 BroadcastException 的“is fatal”布尔字段。

通常，致命错误与阶段服务器的连接有关（连接无法建立，或者连接丢失且无法恢复）。应用程序应重新创建阶段并重新加入，可能使用新令牌或在设备连接恢复后重新加入。

非致命错误通常与发布/订阅状态有关，由 SDK 处理，SDK 会重试发布/订阅操作。

可以检查如下属性：

```
try {
  stage.join(...)
```

```
} catch (e: BroadcastException) {
  If (e.isFatal) {
    // the error is fatal
```

加入错误

令牌格式不正确

当阶段令牌格式不正确时，就会发生这种情况。

SDK 在调用 `stage.join` 时引发 Java 异常，其中错误代码 = 1000，fatal = true。

操作：创建有效令牌并重试加入。

令牌已过期

当阶段令牌过期时，就会发生这种情况。

SDK 在调用 `stage.join` 时引发 Java 异常，其中错误代码 = 1001，fatal = true。

操作：创建新令牌并重试加入。

令牌无效或已撤销

当阶段令牌没有格式错误但被阶段服务器拒绝时，就会发生这种情况。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调用 `onConnectionStateChanged` 时引发异常，其中错误代码 = 1026，fatal = true。

操作：创建有效令牌并重试加入。

初始加入时出现网络错误

当 SDK 无法联系阶段服务器建立连接时，就会发生这种情况。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调用 `onConnectionStateChanged` 时引发异常，其中错误代码 = 1300，fatal = true。

操作：等待设备连接恢复，然后重试加入。

已加入时出现网络错误

如果设备的网络连接中断，SDK 可能会失去与阶段服务器的连接。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调用 onConnectionStateChanged 时引发异常，其中错误代码 = 1300，fatal = true。

操作：等待设备连接恢复，然后重试加入。

发布/订阅错误

初次

有如下几种错误：

- MultihostSessionOfferCreationFailPublish (1020)
- MultihostSessionOfferCreationFailSubscribe (1021)
- MultihostSessionNoIceCandidates (1022)
- MultihostSessionStageAtCapacity (1024)
- SignallingSessionCannotRead (1201)
- SignallingSessionCannotSend (1202)
- SignallingSessionBadResponse (1203)

通过应用程序提供的阶段渲染器异步报告这些情况。

SDK 会在有限的次数内重试该操作。在重试期间，发布/订阅状态为 ATTEMPTING_PUBLISH/ATTEMPTING_SUBSCRIBE。如果重试成功，则状态将更改为 PUBLISHED/SUBSCRIBED。

SDK 调用 onError 时引发相关的错误代码，并且 fatal = false。

操作：无需执行任何操作，因为 SDK 会自动重试。或者，应用程序可以刷新策略以强制进行更多重试。

已经建立，然后失败

发布或订阅在建立后可能会失败，很可能是由于网络错误所致。“对等连接由于未知的网络错误中断”的错误代码为 1400。

通过应用程序提供的阶段渲染器异步报告此情况。

SDK 会重试发布/订阅操作。在重试期间，发布/订阅状态为 ATTEMPTING_PUBLISH/ATTEMPTING_SUBSCRIBE。如果重试成功，则状态将更改为 PUBLISHED/SUBSCRIBED。

SDK 调用 onError 时引发相关的错误，其中错误代码 = 1400，fatal = false。

操作：无需执行任何操作，因为 SDK 会自动重试。或者，应用程序可以刷新策略以强制进行更多重试。如果连接完全丢失，指向阶段的连接也可能失败。

IVS 广播 SDK：iOS 指南（实时流式传输）

IVS 实时流式传输 iOS 广播 SDK 让参与者能在 iOS 设备上发送和接收视频。

`AmazonIVSBroadcast` 模块实施了本文档中所描述的接口。支持以下操作：

- 加入舞台
- 向舞台中的其他参与者发布媒体
- 舞台中其他参与者订阅媒体
- 管理和监控发布到舞台的视频和音频
- 获取每个对等连接的 WebRTC 统计信息
- 所有操作均来自 IVS 低延迟流式传输 iOS 广播 SDK

最新版本的 iOS 广播 SDK：1.14.1（版本说明）

参考文档：有关亚马逊 IVS iOS 广播 SDK 中可用的最重要方法的信息，请参阅参考文档 https://aws.github.io/ amazon-ivs-broadcast-docs /1.14.1/ios/。

示例代码：参见 iOS 示例存储库，网址为 GitHub：https://github.com/aws-samples/ amazon-ivs-broadcast-ios-sample。

平台要求：iOS 14 或更高版本

开始使用

安装库

我们建议您通过集成广播 SDK CocoaPods。（或者，您可以手动将框架添加至项目。）

推荐：集成广播 SDK (CocoaPods)

实时功能作为 iOS 低延迟流式传输广播 SDK 的子规格发布。这样客户就可以根据自己的功能需求选择包含或排除该功能。包括该功能的程序包会更大。

发行版以该 CocoaPods 名称发布`AmazonIVSBroadcast`。将此依赖项添加至您的 Podfile 中：

```
pod 'AmazonIVSBroadcast/Stages'
```

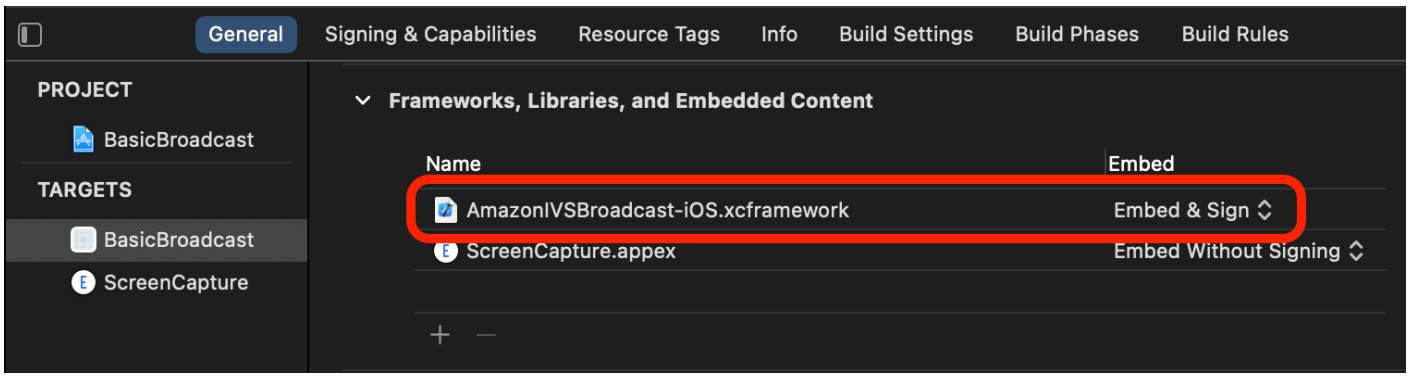
运行 `pod install`，开发工具包将在 `.xcworkspace` 中可用。

重要提示：IVS 实时流式传输广播 SDK（即有舞台子规范）包含 IVS 低延迟流式传输广播 SDK 的所有功能。不可能将两个 SDK 集成到同一个项目中。如果您将舞台子规范 via CocoaPods 添加到项目中，请务必删除 Podfile 中包含的所有其他行。`AmazonIVSBroadcast`例如，在 Podfile 中不要同时包含这两行：

```
pod 'AmazonIVSBroadcast'
pod 'AmazonIVSBroadcast/Stages'
```

替代方法：手动安装框架

1. 从 https://broadcast.live-video.net/1.14.1/AmazonIVSBroadcast-Stages.xcframework.zip 下载最新版本。
2. 提取归档的内容。`AmazonIVSBroadcast.xcframework` 包含适用于设备和模拟器的开发工具包。
3. 通过以下方法嵌入 `AmazonIVSBroadcast.xcframework`：将其拖动到应用程序目标的 General（常规）选项卡中的 Frameworks, Libraries, and Embedded Content（框架、库和嵌入式内容）部分中。



请求权限

您的应用必须请求权限才能访问用户摄像头和麦克风。（这并非特定于 Amazon IVS；需要访问摄像头和麦克风的任何应用程序都需要这样做。）

我们在此处检查用户是否已授予权限，如果没有，对他们提出要求：

```
switch AVCaptureDevice.authorizationStatus(for: .video) {
case .authorized: // permission already granted.
case .notDetermined:
```

```
    AVCaptureDevice.requestAccess(for: .video) { granted in
        // permission granted based on granted bool.
    }
case .denied, .restricted: // permission denied.
@unknown default: // permissions unknown.
}
```

如果您希望分别访问摄像头和麦克风，则需要对 .video 和 .audio 媒体类型进行此操作。

您还需要将 NSCameraUsageDescription 和 NSMicrophoneUsageDescription 的条目添加到 Info.plist。否则，尝试请求权限时，您的应用程序将崩溃。

禁用应用程序空闲计时器

您可以自由选择，但我们建议您这样做。它可以防止您的设备在使用广播开发工具包时进入睡眠状态，这会中断广播。

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    UIApplication.shared.isIdleTimerDisabled = true
}
override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    UIApplication.shared.isIdleTimerDisabled = false
}
```

发布和订阅

概念

三个核心概念构成了实时功能的基础：舞台、策略和渲染器。设计目标是最大限度地减少构建有效产品所需的客户端逻辑量。

舞台

IVSStage 类是主机应用程序和 SDK 之间交互的主要点。此类表示舞台，用于加入和退出舞台。创建或加入舞台需要控制面板上有效的未过期令牌字符串（表示为 token）。加入和退出舞台很简单。

```
let stage = try IVSStage(token: token, strategy: self)

try stage.join()
```

```
stage.leave()
```

也可以将 IVSStageRenderer 和 IVSErrorDelegate 附加到 IVSStage 类：

```
let stage = try IVSStage(token: token, strategy: self)
stage.errorDelegate = self
stage.addRenderer(self) // multiple renderers can be added
```

策略

IVSStageStrategy 协议为主机应用程序提供了一种方法，可以将所需的舞台状态传递给 SDK。需要实现三项函数：shouldSubscribeToParticipant、shouldPublishParticipant 和 streamsToPublishForParticipant。下面将进行详述。

订阅参与者

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType
```

当远程参与者加入舞台时，SDK 会向主机应用程序查询该参与者的所需订阅状态。选项为 .none、.audioOnly 和 .audioVideo。为该函数返回值时，主机应用程序无需担心发布状态、当前订阅状态或舞台连接状态。如果返回 .audioVideo，则 SDK 会等到远程参与者发布后再订阅，并在整个过程中通过渲染器更新主机应用程序。

以下是实施示例：

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    return .audioVideo
}
```

完整实施此功能，适用于始终希望所有参与者都能看到对方的主机应用程序；例如，视频聊天应用程序。

也可以进行更高级的实施。根据服务器提供的属性，使用 IVSParticipantInfo 上的 attributes 属性有选择地订阅参与者：

```
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    switch participant.attributes["role"] {
```

```
    case "moderator": return .none
    case "guest": return .audioVideo
    default: return .none
    }
}
```

此操作用于创建舞台，在该舞台中，监管人可以监视所有来宾，而不会被来宾看见或听见。主机应用程序可以使用其他业务逻辑，让监管人看到彼此，但对来宾不可见。

发布

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool
```

连接到舞台后，SDK 会查询主机应用程序以查看特定参与者是否应该发布。仅对有权根据提供的令牌进行发布的本地参与者调用此操作。

以下是实施示例：

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool {
    return true
}
```

适用于用户总想发布的标准视频聊天应用程序。用户可以将音频和视频静音或取消静音，以便立即隐藏或被看见/听见。（他们也可以使用发布/取消发布，但这要慢得多。对于经常需要更改可见性的使用场景，静音/取消静音更可取。）

选择要发布的流

```
func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream]
```

这项操作用于在发布时确定应发布的音频和视频流。稍后将在 Publish a Media Stream 中对此进行更详细的介绍。

更新策略

此策略是动态的：可以随时更改从上述任何函数返回的值。例如，如果主机应用程序希望最终用户点击按钮之前不要发布，则可以从 `shouldPublishParticipant`（类似于 `hasUserTappedPublishButton`）返回一个变量。当该变量根据最终用户的交互而发生变化时，调

用 stage.refreshStrategy() 发送信号到 SDK，表明 SDK 应该查询策略以获取最新值，仅应用已更改的内容。如果 SDK 发现 shouldPublishParticipant 值已更改，它将启动发布流程。如果 SDK 查询和所有函数返回的值与之前相同，则 refreshStrategy 调用不会对阶段进行任何修改。

如果 shouldSubscribeToParticipant 的返回值从 .audioVideo 更改为 .audioOnly，则如果之前存在视频流，将删除所有返回值已更改的参与者的视频流。

通常，舞台使用该策略来最有效地应用以前和当前策略之间的差异，而主机应用程序无需担心正确管理该策略所需的所有状态。因此，可以将调用 stage.refreshStrategy() 视为一种只需少量计算的操作，因为除非策略发生变化，否则该调用什么都不会做。

渲染器

IVSStageRenderer 协议将舞台状态传递给主机应用程序。渲染器提供的事件通常完全可以支持主机应用程序界面的更新。渲染器提供以下函数：

```
func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo)

func stage(_ stage: IVSStage, participantDidLeave participant: IVSParticipantInfo)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange publishState:
 IVSParticipantPublishState)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChange
 subscribeState: IVSParticipantSubscribeState)

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didAdd streams:
 [IVSStageStream])

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didRemove streams:
 [IVSStageStream])

func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChangeMutedStreams
 streams: [IVSStageStream])

func stage(_ stage: IVSStage, didChange connectionState: IVSStageConnectionState,
 withError error: Error?)
```

预计渲染器提供的信息不会影响策略的返回值。例如，调用 participant:didChangePublishState 时，shouldSubscribeToParticipant 的返回值预计不会改变。如果主机应用程序想要订阅特定参与者，则无论该参与者的发布状态如何，它都应返回所需的订阅类型。SDK 负责确保根据舞台状态在正确的时间执行策略的期望状态。

请注意，只有发布参与者才会触发 participantDidJoin，每当参与者停止发布或退出舞台会话时，都会触发 participantDidLeave。

发布媒体流

通过 IVSDeviceDiscovery 发现内置麦克风和摄像头等本地设备。以下示例演示如何选择前置摄像头和默认麦克风，然后将它们作为 IVSLocalStageStreams 返回，由 SDK 发布：

```
let devices = IVSDeviceDiscovery.listLocalDevices()

// Find the camera virtual device, choose the front source, and create a stream
let camera = devices.compactMap({ $0 as? IVSCamera }).first!
let frontSource = camera.listAvailableInputSources().first(where: { $0.position
 == .front })!
camera.setPreferredInputSource(frontSource)
let cameraStream = IVSLocalStageStream(device: camera)

// Find the microphone virtual device, enable echo cancellation, and create a stream
let microphone = devices.compactMap({ $0 as? IVSMicrophone }).first!
microphone.isEchoCancellationEnabled = true
let microphoneStream = IVSLocalStageStream(device: microphone)

// This is a function on IVSStageStrategy
func stage(_ stage: IVSStage, streamsToPublishForParticipant participant:
 IVSParticipantInfo) -> [IVSLocalStageStream] {
    return [cameraStream, microphoneStream]
}
```

显示并删除参与者

订阅完成后，您将通过渲染器的 didAddStreams 函数接收一组 IVSStageStream 对象。要预览或接收有关该参与者的音频级别统计信息，您可以从流中访问底层 IVSDevice 对象：

```
if let imageDevice = stream.device as? IVSImageDevice {
    let preview = imageDevice.previewView()
    /* attach this UIView subclass to your view */
} else if let audioDevice = stream.device as? IVSAudioDevice {
    audioDevice.setStatsCallback( { stats in
        /* process stats.peak and stats.rms */
    })
}
```

当参与者停止发布或取消订阅时，将使用已删除的流来调用 didRemoveStreams 函数。主机应用程序应将其用作信号，从视图层次结构中删除参与者的视频流。

在所有可能删除流的场景中都会调用 didRemoveStreams，包括：

- 远程参与者停止发布。
- 本地设备取消订阅或将订阅从 .audioVideo 更改为 .audioOnly。
- 远程参与者退出舞台。
- 本地参与者退出舞台。

由于在所有场景中都会调用 didRemoveStreams，因此在远程或本地退出操作期间，从用户界面中删除参与者无需自定义业务逻辑。

静音和取消静音媒体流

IVSLocalStageStream 对象具有控制流是否静音的 setMuted 函数。可以在 streamsToPublishForParticipant 策略函数返回之前或之后在流上调用此函数。

重要提示：如果在调用 refreshStrategy 后 streamsToPublishForParticipant 返回了新的 IVSLocalStageStream 对象实例，将对舞台应用新流对象的静音状态。创建新 IVSLocalStageStream 实例时要小心，务必保持预期的静音状态。

监控远程参与者媒体静音状态

当参与者更改其视频或音频流的静音状态时，将使用一组已更改的流调用渲染器 didChangeMutedStreams 函数。使用 IVSStageStream 上的 isMuted 属性相应地更新您的用户界面：

```
func stage(_ stage: IVSStage, participant: IVSParticipantInfo, didChangeMutedStreams
 streams: [IVSStageStream]) {
    streams.forEach { stream in
        /* stream.isMuted */
    }
}
```

创建舞台配置

要自定义舞台视频配置的值，请使用 IVSLocalStageStreamVideoConfiguration：

```
let config = IVSLocalStageStreamVideoConfiguration()
try config.setMaxBitrate(900_000)
try config.setMinBitrate(100_000)
try config.setTargetFramerate(30)
try config.setSize(CGSize(width: 360, height: 640))
config.degradationPreference = .balanced
```

获取 WebRTC 统计信息

要获取发布流或订阅流的最新 WebRTC 统计信息，请使用 IVSStageStream 上的 requestRTCStats。收集完成后，您将通过 IVSStageStreamDelegate（可在 IVSStageStream 上设置）收到统计信息。要持续收集 WebRTC 统计信息，请在 Timer 上调用此函数。

```
func stream(_ stream: IVSStageStream, didGenerateRTCStats stats: [String : [String :
 String]]) {
    for stat in stats {
      for member in stat.value {
         print("stat \(stat.key) has member \(member.key) with value \(member.value)")
      }
   }
}
```

获取参与者属性

如果您在 CreateParticipantToken 端点请求中指定属性，则可以在 IVSParticipantInfo 属性中看到这些属性：

```
func stage(_ stage: IVSStage, participantDidJoin participant: IVSParticipantInfo) {
    print("ID: \(participant.participantId)")
    for attribute in participant.attributes {
        print("attribute: \(attribute.key)=\(attribute.value)")
    }
}
```

在后台继续会话

应用程序进入后台时，您可以继续在舞台上听到远程音频，但无法继续发送自己的图像和音频。您需要更新 IVSStrategy 实施以停止发布，然后订阅 .audioOnly（或者 .none，如果适用）。

```
func stage(_ stage: IVSStage, shouldPublishParticipant participant: IVSParticipantInfo)
 -> Bool {
    return false
}
func stage(_ stage: IVSStage, shouldSubscribeToParticipant participant:
 IVSParticipantInfo) -> IVSStageSubscribeType {
    return .audioOnly
}
```

然后调用 `stage.refreshStrategy()`。

通过联播启用/禁用分层编码

发布媒体流时，SDK 会传输高质量和低质量的视频流，因此即使下行带宽有限，远程参与者也可以订阅此流。默认情况下，联播分层编码为启用状态。您可以通过 `IVSSimulcastConfiguration` 将其来禁用：

```
// Disable Simulcast
let config = IVSLocalStageStreamVideoConfiguration()
config.simulcast.enabled = false

let cameraStream = IVSLocalStageStream(device: camera, configuration: config)

// Other Stage implementation code
```

将舞台广播到 IVS 通道

要广播舞台，请创建一个单独的 `IVSBroadcastSession`，然后按照上述用 SDK 进行广播的常规说明进行操作。`IVSStageStream` 上的 `device` 属性将是上面代码片段中所示的 `IVSImageDevice` 或 `IVSAudioDevice`；这些属性可以连接到 `IVSBroadcastSession.mixer`，从而以可自定义的布局广播整个舞台。

或者，您可以合成舞台并将其广播到 IVS 低延迟通道，以覆盖更多的观众。请参阅 IVS Low-Latency Streaming User Guide 中的 Enabling Multiple Hosts on an Amazon IVS Stream。

iOS 如何选择相机分辨率和帧率

由广播 SDK 管理的摄像机可优化其分辨率和帧速率（或 FPS）frames-per-second，以最大限度地减少热量产生和能耗。本节介绍如何选择分辨率和帧率以帮助主机应用程序针对其使用案例进行优化。

当使用 IVSCamera 创建 IVSLocalStageStream 时，会根据帧率 IVSLocalStageStreamVideoConfiguration.targetFramerate 和分辨率 IVSLocalStageStreamVideoConfiguration.size 优化相机。调用 IVSLocalStageStream.setConfiguration 会使用更新的值更新相机。

相机预览

如果您在不将 IVSCamera 连接到 IVSBroadcastSession 或 IVSStage 时创建其预览，则默认分辨率为 1080p，帧率为 60 fps。

广播舞台

使用 IVSBroadcastSession 广播 IVSStage 时，SDK 会尝试使用符合两个会话标准的分辨率和帧率来优化相机。

例如，如果广播配置的帧率设置为 15 FPS，分辨率设置为 1080p，而舞台的帧率为 30 FPS，分辨率为 720p，则 SDK 将选择帧率为 30 FPS、分辨率为 1080p 的相机配置。IVSBroadcastSession 会从相机中每隔一帧删除一帧，然后 IVSStage 将 1080p 的图像缩减到 720p。

如果主机应用程序计划将 IVSBroadcastSession 和 IVSStage 与相机结合使用，则建议相应配置的 targetFramerate 和 size 属性相匹配。不匹配可能会导致相机在捕获视频时自行重新配置，这将导致视频样本传输出现短暂延迟。

如果具有相同的值不符合主机应用程序的使用案例，则首先创建更高质量的相机将防止相机在添加较低质量会话时自行重新配置。例如，如果您以 1080p 和 30 FPS 进行广播，然后加入设为 720p 和 30 FPS 的舞台，则相机不会自行重新配置，视频将不间断地继续播放。这是因为 720p 小于或等于 1080p 而 30 FPS 小于或等于 30 FPS。

任意帧率、分辨率和宽高比

大多数相机硬件可以完全匹配常见格式，例如 30 FPS 时的 720p 或 60 FPS 时的 1080p。但是，您无法完全匹配所有格式。广播 SDK 根据以下规则（按优先顺序排列）选择相机配置：

1. 分辨率的宽度和高度大于或等于所需的分辨率，但是在此限制范围内，宽度和高度尽可能小。
2. 帧率大于或等于所需的帧率，但是在此限制范围内，帧率尽可能低。
3. 宽高比与所需的宽高比相匹配。
4. 如果有多种匹配格式，则使用视野最大的格式。

以下是两个示例：

- 主机应用程序正在尝试以 120 FPS 的帧率按 4k 进行广播。所选相机在 60 FPS 时仅支持 4k 或在 120 FPS 时仅支持 1080p。所选格式在 60 FPS 时将为 4k，因为分辨率规则的优先级高于帧率规则。
- 请求了一种不规则的分辨率，即 1910x1070。相机将使用 1920x1080。注意：选择 1921x1080 之类的分辨率会导致相机纵向扩展到下一个可用分辨率（例如 2592x1944），这会导致 CPU 和内存带宽损失。

Android 的情况怎么样？

Android 不会像 iOS 那样即时调整其分辨率或帧率，因此这不会影响 Android 广播 SDK。

已知问题和解决方法

- 更改蓝牙音频路由是不可预测的。如果您在会话中连接新设备，iOS 可能会自动更改输入路由，也可能不会。此外，无法在同一时间连接的多个蓝牙耳机之间进行选择。常规广播和舞台会话中均会发生此现象。

  解决办法：如果您打算使用蓝牙耳机，请在开始广播或进入舞台之前进行连接，并在整个会话期间保持连接状态。
- 使用 iPhone 14、iPhone 14 Plus、iPhone 14 Pro 或 iPhone 14 Pro Max 的参与者可能会导致其他参与者出现音频回声问题。

  解决方法：使用受影响设备的参与者可以使用耳机来防止其他参与者出现回声问题。
- 当一个参与者使用另一个参与者正在使用的令牌加入时，第一个连接将断开，不会出现具体错误提示。

  解决办法：尚无。
- 有一个问题比较少见，即发布者正在发布，但订阅用户收到的发布状态是 `inactive`。

  解决方法：尝试退出然后加入会话。如果问题仍然存在，请为发布者创建新令牌。
- 当参与者发布或订阅时，即使网络稳定，也可能会收到代码为 1400 的错误，表明由于网络问题而断开连接。

  解决方法：尝试重新发布/重新订阅。
- 在舞台会话期间，通常在持续时间较长的通话中，可能会间歇性地出现罕见的音频失真问题。

  解决方法：遇到音频失真问题的参与者可以退出并重新加入会话，也可以取消发布并重新发布音频，以修复问题。

错误处理

致命错误与非致命错误

错误对象带有“is fatal”布尔字段。这是 `IVSBroadcastErrorIsFatalKey` 下包含布尔值的字典条目。

通常，致命错误与阶段服务器的连接有关（连接无法建立，或者连接丢失且无法恢复）。应用程序应重新创建阶段并重新加入，可能使用新令牌或在设备连接恢复后重新加入。

非致命错误通常与发布/订阅状态有关，由 SDK 处理，SDK 会重试发布/订阅操作。

可以检查如下属性：

```
let nsError = error as NSError
if nsError.userInfo[IVSBroadcastErrorIsFatalKey] as? Bool == true {
  // the error is fatal
}
```

加入错误

令牌格式不正确

当阶段令牌格式不正确时，就会发生这种情况。

SDK 抛出一个 Swift 异常，错误代码 = 1000，IVS BroadcastErrorIsFatalKey = 是。

操作：创建有效令牌并重试加入。

令牌已过期

当阶段令牌过期时，就会发生这种情况。

SDK 抛出一个 Swift 异常，错误代码 = 1001，IVS BroadcastErrorIsFatalKey = 是。

操作：创建新令牌并重试加入。

令牌无效或已撤销

当阶段令牌没有格式错误但被阶段服务器拒绝时，就会发生这种情况。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调`stage(didChange connectionState, withError error)`用时错误代码 = 1026，IVS BroadcastErrorIsFatalKey = 是。

操作：创建有效令牌并重试加入。

初始加入时出现网络错误

当 SDK 无法联系阶段服务器建立连接时，就会发生这种情况。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调`stage(didChange connectionState, withError error)`用时错误代码 = 1300，IVS BroadcastErrorIsFatalKey = YES。

操作：等待设备连接恢复，然后重试加入。

已加入时出现网络错误

如果设备的网络连接中断，SDK 可能会失去与阶段服务器的连接。通过应用程序提供的阶段渲染器异步报告此情况。

SDK 调`stage(didChange connectionState, withError error)`用时错误代码 = 1300，IVS BroadcastErrorIsFatalKey 值 = 是。

操作：等待设备连接恢复，然后重试加入。

发布/订阅错误

初次

有如下几种错误：

- MultihostSessionOfferCreationFailPublish (1020)
- MultihostSessionOfferCreationFailSubscribe (1021)
- MultihostSessionNoIceCandidates (1022)
- MultihostSessionStageAtCapacity (1024)
- SignallingSessionCannotRead (1201)
- SignallingSessionCannotSend (1202)
- SignallingSessionBadResponse (1203)

通过应用程序提供的阶段渲染器异步报告这些情况。

SDK 会在有限的次数内重试该操作。在重试期间，发布/订阅状态为 `ATTEMPTING_PUBLISH/ATTEMPTING_SUBSCRIBE`。如果重试成功，则状态将更改为 `PUBLISHED/SUBSCRIBED`。

SDK 调IVSErrorSourceDelegate:didEmitError用时会显示相关的错误代码，IVS BroadcastErrorIsFatalKey = 否。

操作：无需执行任何操作，因为 SDK 会自动重试。或者，应用程序可以刷新策略以强制进行更多重试。

已经建立，然后失败

发布或订阅在建立后可能会失败，很可能是由于网络错误所致。“对等连接由于未知的网络错误中断”的错误代码为 1400。

通过应用程序提供的阶段渲染器异步报告此情况。

SDK 会重试发布/订阅操作。在重试期间，发布/订阅状态为 ATTEMPTING_PUBLISH/ATTEMPTING_SUBSCRIBE。如果重试成功，则状态将更改为 PUBLISHED/SUBSCRIBED。

SDK 调didEmitError用时错误代码 = 1400，IVS BroadcastErrorIsFatalKey = NO。

操作：无需执行任何操作，因为 SDK 会自动重试。或者，应用程序可以刷新策略以强制进行更多重试。如果连接完全丢失，指向阶段的连接也可能失败。

IVS 广播 SDK：自定义图像源（实时流式传输）

自定义图像输入源允许应用程序向广播 SDK 提供自己的图像输入，而不仅限于预设相机。自定义图像源可以是简单的半透明水印或静态“马上回来”等场景，也可以允许应用程序执行额外的自定义处理，例如向相机添加美颜滤镜。

当您使用自定义图像输入源对摄像机进行自定义控制时（例如使用需要访问相机的美颜滤镜库）时，广播 SDK 不再负责管理相机。相反，应用程序负责正确处理相机的生命周期。请参阅官方平台文档，以了解应用程序应如何管理相机。

Android

创建 DeviceDiscovery 会话后，请创建一个图像输入源：

```
CustomImageSource imageSource = deviceDiscovery.createImageInputSource(new
 BroadcastConfiguration.Vec2(1280, 720));
```

此方法将返回一个 CustomImageSource，这是标准的 Android Surface 支持的图像源。子类 SurfaceSource 支持大小调整和轮换。您还可以创建 ImagePreviewView 来显示其内容预览。

检索底层 Surface：

```
Surface surface = surfaceSource.getInputSurface();
```

此 Surface 可以用作图像创建器（例如 Camera2、OpenGL ES 和其他库）的输出缓冲区。最简单的使用场景是直接在 Surface 画布中绘制静态位图或颜色。但是，许多库（例如 beauty-filter 库）提供了一种方法，允许应用程序指定外部 Surface 来进行渲染。您可以用这种方法将此 Surface 传递到滤镜库，从而让库输出处理后的帧以便广播会话进行流式传输。

此 CustomImageSource 可以包装在 LocalStageStream 中并由 StageStrategy 返回以发布到 Stage。

iOS

创建 DeviceDiscovery 会话后，请创建一个图像输入源：

```
let customSource = broadcastSession.createImageSource(withName: "customSourceName")
```

此方法将返回一个 IVSCustomImageSource，这是一个允许应用程序手动提交 CMSampleBuffers 的图像源。有关支持的像素格式，请参阅 iOS 广播 SDK 参考；指向最新版本的链接详见最新广播 SDK 发行版的 Amazon IVS 版本注释。

提交到自定义源的样本将流式传输到舞台：

```
customSource.onSampleBuffer(sampleBuffer)
```

对于串流视频，请在回调中使用此方法。例如，假设您使用的是相机，则每次从 AVCaptureSession 收到一个新的样本缓冲时，应用程序可以将样本缓冲转发到自定义图像源。如果需要，应用程序可以在将样本提交到自定义图像源之前执行进一步的处理（例如美颜滤镜）。

IVSCustomImageSource 可以包装在 IVSLocalStageStream 中并由 IVSStageStrategy 返回以发布到 Stage。

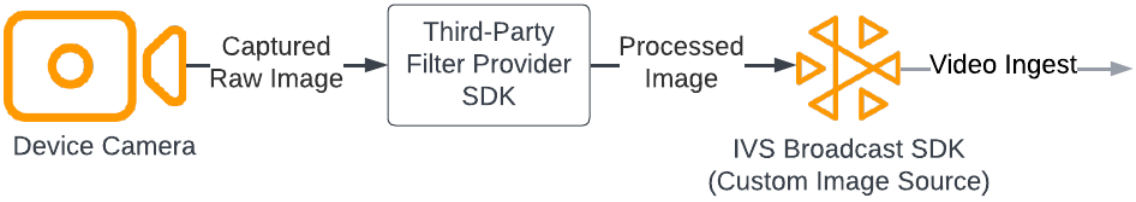
IVS 广播 SDK：第三方相机滤镜（实时直播功能）

本指南假设您已经熟悉自定义图像源以及将 IVS 实时流式广播 SDK 集成到您的应用程序中。

相机滤镜使直播创作者能够增强或改变他们的面部或背景外观。这有可能提高观众的参与度、吸引观众并增强直播体验。

集成第三方相机滤镜

通过将滤镜 SDK 的输出馈送到自定义图像输入源，您可以将第三方相机滤镜 SDK 与 IVS 广播 SDK 集成。自定义图像输入源允许应用程序向广播 SDK 提供自己的图像输入。第三方滤镜提供商的 SDK 可以管理相机的生命周期，以处理来自相机的图像、应用滤镜效果，并以可传递到自定义图像源的格式将其输出。



请参阅第三方滤镜提供者的文档，了解将应用了滤镜效果的相机帧转换为可以传递到自定义图像输入源的格式的内置方法。该流程因所使用的 IVS 广播 SDK 版本而异：

- Web — 滤镜提供者必须能够将其输出渲染到画布元素。然后，可以使用 captureStream 方法返回画布内容的 MediaStream。然后，可以将 MediaStream 转换为 LocalStageStream 的实例并发布到舞台。
- Android — 滤镜提供者的 SDK 可以将帧渲染到 IVS 广播 SDK 提供的安卓 `Surface`，也可以将帧转换为位图。如果使用位图，则可以通过解锁并写入画布，将其渲染到自定义图像源提供的底层 `Surface`。
- iOS — 第三方滤镜提供者的 SDK 必须提供应用了滤镜效果的相机帧作为 `CMSampleBuffer`。有关如何在处理相机图像之后获取 `CMSampleBuffer` 作为最终输出的信息，请参阅第三方滤镜提供者 SDK 的文档。

BytePlus

Android

安装和设置 BytePlus Effects SDK

有关如何安装、初始化和设置 BytePlus Effects SDK 的详细信息，请参阅 BytePlus Android 访问指南。

设置自定义图像源

初始化 SDK 后，将经过处理并应用了滤镜效果的相机帧馈送到自定义图像输入源。为此，请创建 `DeviceDiscovery` 对象的实例并创建自定义图像源。请注意，当您使用自定义图像输入源对相机进行自定义控制时，广播 SDK 不再负责管理相机。相反，应用程序负责正确处理相机的生命周期。

Java

```
var deviceDiscovery = DeviceDiscovery(applicationContext)
var customSource = deviceDiscovery.createImageInputSource( BroadcastConfiguration.Vec2(
720F, 1280F
))
var surface: Surface = customSource.inputSurface
var filterStream = ImageLocalStageStream(customSource)
```

将输出转换为位图并馈送到自定义图像输入源

为了使来自 BytePlus Effect SDK 的应用了滤镜效果的相机帧直接转发到 IVS 广播 SDK，请将纹理的 BytePlus Effects SDK 的输出转换为位图。处理图像时，SDK 会调用 onDrawFrame() 方法。onDrawFrame() 方法是 Android 的 GLSurfaceView.Renderer 界面中的一种公共方法。在 BytePlus 提供的 Android 示例应用程序中，在每个相机帧上都调用此方法；它输出纹理。同时，您可以使用逻辑来补充 onDrawFrame() 方法，将此纹理转换为位图并将其馈送到自定义图像输入源。如以下代码示例中所示，使用 BytePlus SDK 提供的 transferTextureToBitmap 方法进行此转换。此方法由来自 BytePlus Effects SDK 的 com.bytedance.labcv.core.util.ImageUtil 库提供，如以下代码示例中所示。然后，您可以将生成的位图写入 Surface 的画布以渲染到 CustomImageSource 的底层 Android Surface。多次连续调用 onDrawFrame() 会生成一系列位图，组合后会形成视频流。

Java

```
import com.bytedance.labcv.core.util.ImageUtil;
...
protected ImageUtil imageUtility;
...


@Override
public void onDrawFrame(GL10 gl10) {
  ...
  // Convert BytePlus output to a Bitmap
  Bitmap outputBt = imageUtility.transferTextureToBitmap(output.getTexture(),ByteEffect

  Constants.TextureFormat.Texture2D,output.getWidth(), output.getHeight());

  canvas = surface.lockCanvas(null);
  canvas.drawBitmap(outputBt, 0f, 0f, null);
  surface.unlockCanvasAndPost(canvas);
```

DeepAR

Android

有关如何将 DeepAR SDK 与 Android IVS 广播 SDK 集成的详细信息，请参阅 Deepar 的 Android 集成指南。

iOS

有关如何将 DeepAR SDK 与 iOS IVS 广播 SDK 集成的详细信息，请参阅 Deepar 的 iOS 集成指南。

Snap

Web

本节假设您已经熟悉使用 Web 广播 SDK 发布和订阅视频。

要集成 Snap 的 Camera Kit SDK 与 IVS 实时流式 Web 广播 SDK，您需要：

1. 安装 Camera Kit SDK 和 Webpack。（我们的示例使用 Webpack 作为捆绑程序，但您可以使用自己选择的任何捆绑程序。）
2. 创建 `index.html`。
3. 添加安装元素。
4. 显示和设置参与者。
5. 显示连接的相机和麦克风。
6. 创建 Camera Kit 会话。
7. 获取并应用镜头。
8. 将 Camera Kit 会话的输出渲染到画布。
9. 为 Camera Kit 提供用于渲染的媒体来源并发布 `LocalStageStream`。
10. 创建 Webpack 配置文件。

这些步骤中的各自的介绍如下。

安装 Camera Kit SDK 和 Webpack

```
npm i @snap/camera-kit webpack webpack-cli
```

创建 index.html

接下来，创建 HTML 样板，并将 Web 广播 SDK 作为脚本标签导入。在下面的代码中，请务必将 <SDK version> 替换为您正在使用的广播 SDK 版本。

JavaScript

```
<!--
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */
-->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <title>Amazon IVS Real-Time Streaming Web Sample (HTML and JavaScript)</title>

  <!-- Fonts and Styling -->
  <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:300,300italic,700,700italic" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/
normalize.css" />
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/milligram/1.4.1/
milligram.css" />
  <link rel="stylesheet" href="./index.css" />

  <!-- Stages in Broadcast SDK -->
  <script src="https://web-broadcast.live-video.net/<SDK version>/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>
  <!-- Introduction -->
  <header>
    <h1>Amazon IVS Real-Time Streaming Web Sample (HTML and JavaScript)</h1>

    <p>This sample is used to demonstrate basic HTML / JS usage. <b><a href="https://
docs.aws.amazon.com/ivs/latest/userguide/multiple-hosts.html">Use the AWS CLI</
a></b> to create a <b>Stage</b> and a corresponding <b>ParticipantToken</b>.
 Multiple participants can load this page and put in their own tokens. You can <b><a
```

```
 href="https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#glossary"
 target="_blank">read more about stages in our public docs.</a></b></p>
  </header>
  <hr />

  <!-- Setup Controls -->

  <!-- Local Participant -->

  <hr style="margin-top: 5rem"/>

  <!-- Remote Participants -->

  <!-- Load all Desired Scripts -->

</body>

</html>
```

添加安装元素

创建 HTML，用于选择相机和麦克风并指定参与者令牌：

JavaScript

```
<!-- Setup Controls -->
  <div class="row">
    <div class="column">
      <label for="video-devices">Select Camera</label>
      <select disabled id="video-devices">
        <option selected disabled>Choose Option</option>
      </select>
    </div>
    <div class="column">
      <label for="audio-devices">Select Microphone</label>
      <select disabled id="audio-devices">
        <option selected disabled>Choose Option</option>
      </select>
    </div>
    <div class="column">
      <label for="token">Participant Token</label>
      <input type="text" id="token" name="token" />
    </div>
    <div class="column" style="display: flex; margin-top: 1.5rem">
```

```
      <button class="button" style="margin: auto; width: 100%" id="join-button">Join
 Stage</button>
    </div>
    <div class="column" style="display: flex; margin-top: 1.5rem">
      <button class="button" style="margin: auto; width: 100%" id="leave-button">Leave
 Stage</button>
    </div>
  </div>
```

在其下方添加其他 HTML 以显示来自本地和远程参与者的相机源：

JavaScript

```
 <!-- Local Participant -->
<div class="row local-container">
    <canvas id="canvas"></canvas>

    <div class="column" id="local-media"></div>
    <div class="static-controls hidden" id="local-controls">
      <button class="button" id="mic-control">Mute Mic</button>
      <button class="button" id="camera-control">Mute Camera</button>
    </div>
  </div>


  <hr style="margin-top: 5rem"/>

  <!-- Remote Participants -->
  <div class="row">
    <div id="remote-media"></div>
  </div>
```

加载其他逻辑，包括用于设置相机的辅助方法和捆绑的 JavaScript 文件。（在本节的后面部分，您将创建这些 JavaScript 文件并将它们捆绑到单个文件中，这样您就可以将 Camera Kit 作为模块导入。捆绑的 JavaScript 文件将包含设置 Camera Kit、应用镜头以及将相机源及所应用镜头发布到舞台的逻辑。）

JavaScript

```
<!-- Load all Desired Scripts -->
  <script src="./helpers.js"></script>
  <script src="./media-devices.js"></script>
  <!-- <script type="module" src="./stages-simple.js"></script> -->
```

```
  <script src="./dist/bundle.js"></script>
```

显示和设置参与者

接下来，创建 `helpers.js`，其中包含用于显示和设置参与者的辅助方法：

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

function setupParticipant({ isLocal, id }) {
  const groupId = isLocal ? 'local-media' : 'remote-media';
  const groupContainer = document.getElementById(groupId);

  const participantContainerId = isLocal ? 'local' : id;
  const participantContainer = createContainer(participantContainerId);
  const videoEl = createVideoEl(participantContainerId);

  participantContainer.appendChild(videoEl);
  groupContainer.appendChild(participantContainer);

  return videoEl;
}

function teardownParticipant({ isLocal, id }) {
  const groupId = isLocal ? 'local-media' : 'remote-media';
  const groupContainer = document.getElementById(groupId);
  const participantContainerId = isLocal ? 'local' : id;

  const participantDiv = document.getElementById(
    participantContainerId + '-container'
  );
  if (!participantDiv) {
    return;
  }
  groupContainer.removeChild(participantDiv);
}

function createVideoEl(id) {
  const videoEl = document.createElement('video');
  videoEl.id = id;
  videoEl.autoplay = true;
  videoEl.playsInline = true;
```

```
  videoEl.srcObject = new MediaStream();
  return videoEl;
}

function createContainer(id) {
  const participantContainer = document.createElement('div');
  participantContainer.classList = 'participant-container';
  participantContainer.id = id + '-container';

  return participantContainer;
}
```

显示已连接的相机和麦克风

接下来，创建 media-devices.js，其中包含用于显示连接到设备的相机和麦克风的辅助方法：

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

/**
 * Returns an initial list of devices populated on the page selects
 */
async function initializeDeviceSelect() {
  const videoSelectEl = document.getElementById('video-devices');
  videoSelectEl.disabled = false;

  const { videoDevices, audioDevices } = await getDevices();
  videoDevices.forEach((device, index) => {
    videoSelectEl.options[index] = new Option(device.label, device.deviceId);
  });

  const audioSelectEl = document.getElementById('audio-devices');

  audioSelectEl.disabled = false;
  audioDevices.forEach((device, index) => {
    audioSelectEl.options[index] = new Option(device.label, device.deviceId);
  });
}

/**
 * Returns all devices available on the current device
 */
```

```
async function getDevices() {
  // Prevents issues on Safari/FF so devices are not blank
  await navigator.mediaDevices.getUserMedia({ video: true, audio: true });

  const devices = await navigator.mediaDevices.enumerateDevices();
  // Get all video devices
  const videoDevices = devices.filter((d) => d.kind === 'videoinput');
  if (!videoDevices.length) {
    console.error('No video devices found.');
  }

  // Get all audio devices
  const audioDevices = devices.filter((d) => d.kind === 'audioinput');
  if (!audioDevices.length) {
    console.error('No audio devices found.');
  }

  return { videoDevices, audioDevices };
}

async function getCamera(deviceId) {
  // Use Max Width and Height
  return navigator.mediaDevices.getUserMedia({
    video: {
      deviceId: deviceId ? { exact: deviceId } : null,
    },
    audio: false,
  });
}

async function getMic(deviceId) {
  return navigator.mediaDevices.getUserMedia({
    video: false,
    audio: {
      deviceId: deviceId ? { exact: deviceId } : null,
    },
  });
}
```

创建 Camera Kit 会话

创建 `stages.js`，其中包含将镜头应用于相机视频源并将该视频源发布到舞台的逻辑。在本文件的第一部分，我们导入广播 SDK 和 Camera Kit Web SDK，并初始化将用于每个 SDK 的变量。我们通过

在启动 Camera Kit Web SDK 后调用 `createSession` 来创建 Camera Kit 会话。请注意，画布元素对象会传递到会话；这会告诉 Camera Kit 渲染到该画布中。

Java

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

// All helpers are expose on 'media-devices.js' and 'dom.js'
// const { setupParticipant } = window;
// const { initializeDeviceSelect, getCamera, getMic } = window;
// require('./helpers.js');
// require('./media-devices.js');

const {
  Stage,
  LocalStageStream,
  SubscribeType,
  StageEvents,
  ConnectionState,
  StreamType,
} = IVSBroadcastClient;

import {
  bootstrapCameraKit,
  createMediaStreamSource,
  Transform2D,
} from '@snap/camera-kit';

let cameraButton = document.getElementById('camera-control');
let micButton = document.getElementById('mic-control');
let joinButton = document.getElementById('join-button');
let leaveButton = document.getElementById('leave-button');

let controls = document.getElementById('local-controls');
let videoDevicesList = document.getElementById('video-devices');
let audioDevicesList = document.getElementById('audio-devices');

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
```

```
let cameraStageStream;
let micStageStream;

const liveRenderTarget = document.getElementById('canvas');

const init = async () => {
  await initializeDeviceSelect();

  const cameraKit = await bootstrapCameraKit({
    apiToken: INSERT_API_TOKEN_HERE,
  });

  const session = await cameraKit.createSession({ liveRenderTarget });
```

获取并应用镜头

要获取镜头，请输入您的镜头组 ID，它可在 Camera Kit 开发人员门户中找到。在此示例中，我们通过应用返回的镜头数组中的第一个镜头来简化操作。

JavaScript

```
const { lenses } = await cameraKit.lensRepository.loadLensGroups([
    INSERT_LENS_GROUP_ID_HERE,
  ]);

  session.applyLens(lenses[0]);
```

将 Camera Kit 会话的输出渲染到画布

使用 captureStream 方法返回画布内容的 MediaStream。画布将包含已应用镜头的相机视频源的视频流。此外，为用于将相机和麦克风静音的按钮添加事件侦听器，以及用于加入和离开舞台的事件侦听器。在加入舞台的事件侦听器中，我们传入 Camera Kit 会话以及来自画布的 MediaStream，这样它就可以发布到舞台。

JavaScript

```
const snapStream = liveRenderTarget.captureStream();

  cameraButton.addEventListener('click', () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? 'Show Camera' : 'Hide Camera';
  });
```

```
  micButton.addEventListener('click', () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
    micButton.innerText = isMuted ? 'Unmute Mic' : 'Mute Mic';
  });

  joinButton.addEventListener('click', () => {
    joinStage(session, snapStream);
  });

  leaveButton.addEventListener('click', () => {
    leaveStage();
  });
};
```

为 Camera Kit 提供用于渲染的媒体来源并发布 LocalStageStream

要发布应用了镜头的视频流，请创建一个调用 `setCameraKitSource` 的函数，用于传入之前从画布上捕获的 `MediaStream`。画布中的 `MediaStream` 目前没有执行任何操作，因为我们还没有整合本地相机源。我们可以通过调用 `getCamera` 辅助方法并将其分配给 `localCamera` 来整合本地相机源。然后，我们可以将本地相机源（通过 `localCamera`）和会话对象传递给 `setCameraKitSource`。`setCameraKitSource` 函数通过调用 `createMediaStreamSource` 将我们的本地相机源转换为 CameraKit 的媒体源。然后，转换 CameraKit 的媒体源以生成前置相头的镜像。然后，镜头效果应用于媒体源，并通过调用 `session.play()` 将其渲染到输出画布。

现在，将镜头应用于从画布捕获的 `MediaStream` 后，我们可以继续将其发布到舞台。为此，我们使用来自 `MediaStream` 的视频轨道创建 `LocalStageStream`。然后，可以将 `LocalStageStream` 的实例传递到要发布的 `StageStrategy`。

JavaScript

```
async function setCameraKitSource(session, mediaStream) {
  const source = createMediaStreamSource(mediaStream);
  await session.setSource(source);
  source.setTransform(Transform2D.MirrorX);
  session.play();
}

const joinStage = async (session, snapStream) => {
  if (connected || joining) {
    return;
```

```
  }
  joining = true;

  const token = document.getElementById('token').value;

  if (!token) {
    window.alert('Please enter a participant token');
    joining = false;
    return;
  }

  // Retrieve the User Media currently set on the page
  localCamera = await getCamera(videoDevicesList.value);
  localMic = await getMic(audioDevicesList.value);
  await setCameraKitSource(session, localCamera);
  // Create StageStreams for Audio and Video
  // cameraStageStream = new LocalStageStream(localCamera.getVideoTracks()[0]);
  cameraStageStream = new LocalStageStream(snapStream.getVideoTracks()[0]);
  micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);

  const strategy = {
    stageStreamsToPublish() {
      return [cameraStageStream, micStageStream];
    },
    shouldPublishParticipant() {
      return true;
    },
    shouldSubscribeToParticipant() {
      return SubscribeType.AUDIO_VIDEO;
    },
  };
```

下面的其余代码用于创建和管理我们的舞台：

JavaScript

```
stage = new Stage(token, strategy);

  // Other available events:
  // https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#events

  stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
    connected = state === ConnectionState.CONNECTED;
```

```
    if (connected) {
      joining = false;
      controls.classList.remove('hidden');
    } else {
      controls.classList.add('hidden');
    }
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
    console.log('Participant Joined:', participant);
  });

  stage.on(
    StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED,
    (participant, streams) => {
      console.log('Participant Media Added: ', participant, streams);

      let streamsToDisplay = streams;

      if (participant.isLocal) {
        // Ensure to exclude local audio streams, otherwise echo will occur
        streamsToDisplay = streams.filter(
          (stream) => stream.streamType === StreamType.VIDEO
        );
      }

      const videoEl = setupParticipant(participant);
      streamsToDisplay.forEach((stream) =>
        videoEl.srcObject.addTrack(stream.mediaStreamTrack)
      );
    }
  );

  stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
    console.log('Participant Left: ', participant);
    teardownParticipant(participant);
  });

  try {
    await stage.join();
  } catch (err) {
    joining = false;
    connected = false;
    console.error(err.message);
```

```
  }
};

const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;

  cameraButton.innerText = 'Hide Camera';
  micButton.innerText = 'Mute Mic';
  controls.classList.add('hidden');
};

init();
```

创建 Webpack 配置文件

创建 `webpack.config.js` 并添加以下代码。这捆绑了上面的逻辑，使您能够利用 import 语句来使用 Camera Kit。

JavaScript

```
const path = require('path');
module.exports = {
  entry: ['./stage.js'],
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

最后，运行 `npm run build` 以按照 Webpack 配置文件中的定义捆绑您的 JavaScript。您随后可以从网络服务器提供 HTML 和 JavaScript。例如，您可以使用 Python 的 HTTP 服务器并打开 localhost:8000 来查看结果：

```
# Run this from the command line and the directory containing index.html
python3 -m http.server -d ./
```

Android

要将 Snap 的 Camera Kit SDK 与 IVS Android 广播 SDK 集成，您必须安装 Camera Kit SDK，初始化 Camera Kit 会话，应用镜头并将 Camera Kit 会话的输出馈送到自定义图像输入源。

要安装 Camera Kit SDK，请将以下内容添加到您的模块的 `build.gradle` 文件中。将 `$cameraKitVersion` 替换为最新的 Camera Kit SDK 版本。

Java

```
implementation "com.snap.camerakit:camerakit:$cameraKitVersion"
```

初始化并获取 `cameraKitSession`。Camera Kit 还为 Android 的 CameraX API 提供了便捷的包装器，因此您无需编写复杂的逻辑即可将 CameraX 与 Camera Kit 一起使用。您可以将 `CameraXImageProcessorSource` 对象用作 ImageProcessor 的来源，这样您就可以开始相机预览流式帧了。

Java

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);

       setContentView(R.layout.activity_main);

       // Camera Kit support implementation of ImageProcessor that is backed by
CameraX library:
       // https://developer.android.com/training/camerax
       CameraXImageProcessorSource imageProcessorSource = new
CameraXImageProcessorSource(
           this /*context*/, this /*lifecycleOwner*/
       );
       imageProcessorSource.startPreview(true /*cameraFacingFront*/);

       cameraKitSession = Sessions.newBuilder(this)
               .imageProcessorSource(imageProcessorSource)
               .attachTo(findViewById(R.id.camerakit_stub))
               .build();
   }
```

获取并应用镜头

您可以在 Camera Kit 开发人员门户的轮盘中配置镜头及其顺序：

Java

```
// Fetch lenses from repository and apply them
 // Replace LENS_GROUP_ID with Lens Group ID from https://camera-kit.snapchat.com
cameraKitSession.getLenses().getRepository().get(new Available(LENS_GROUP_ID),
 available -> {
     Log.d(TAG, "Available lenses: " + available);
     Lenses.whenHasFirst(available, lens ->
 cameraKitSession.getLenses().getProcessor().apply(lens, result -> {
          Log.d(TAG,  "Apply lens [" + lens + "] success: " + result);
      }));
});
```

要进行广播，请将处理后的帧发送到自定义图像源的底层 Surface。使用 DeviceDiscovery 对象并创建 CustomImageSource 以返回 SurfaceSource。然后，您可以将 CameraKit 会话的输出渲染到 SurfaceSource 提供的底层 Surface。

Java

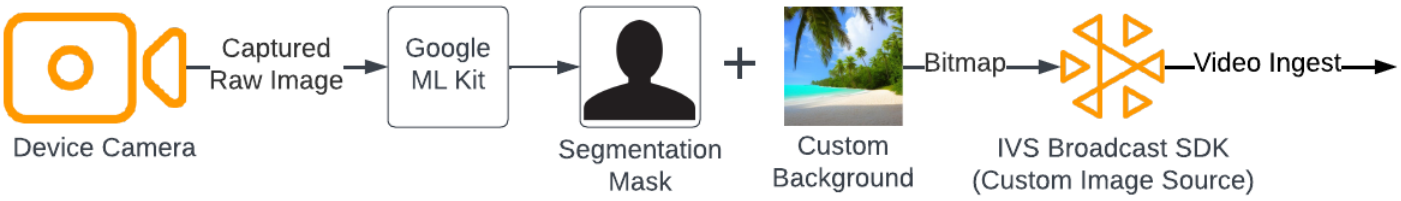
```
val publishStreams = ArrayList<LocalStageStream>()

val deviceDiscovery = DeviceDiscovery(applicationContext)
val customSource =
 deviceDiscovery.createImageInputSource(BroadcastConfiguration.Vec2(720f, 1280f))

cameraKitSession.processor.connectOutput(outputFrom(customSource.inputSurface))
val customStream = ImageLocalStageStream(customSource)

// After rendering the output from a Camera Kit session to the Surface, you can
// then return it as a LocalStageStream to be published by the Broadcast SDK
val customStream: ImageLocalStageStream = ImageLocalStageStream(surfaceSource)
publishStreams.add(customStream)

@Override
fun stageStreamsToPublishForParticipant(stage: Stage, participantInfo:
 ParticipantInfo): List<LocalStageStream> = publishStreams
```

背景替换

背景替换是一种相机滤镜，它使直播创作者能够更改其背景。如下图所示，替换背景涉及：

1. 从实时相机源中获取相机图像。

2. 使用 Google 机器学习套件将其分为前景和背景分量。
3. 将生成的分割遮罩与自定义背景图像相结合。
4. 将其传递给自定义图像源进行广播。



Web

本节假设您已经熟悉使用 Web 广播 SDK 发布和订阅视频。

要使用自定义图像替换直播的背景，请使用带有 MediaPipe Image Segmenter 的自拍分割模型。这是一种机器学习模型，用于识别视频帧中的哪些像素位于前景或背景。然后，您可以使用该模型的结果来替换直播的背景，方法是将视频源中的前景像素复制到表示新背景的自定义图像中。

要将背景替换与 IVS 实时流式 Web 广播 SDK 集成，您需要：

1. 安装 MediaPipe 和 Webpack。（我们的示例使用 Webpack 作为捆绑程序，但您可以使用自己选择的任何捆绑程序。）
2. 创建 `index.html`。
3. 添加媒体元素。
4. 添加脚本标签。
5. 创建 `app.js`。
6. 加载自定义背景图像。
7. 创建 `ImageSegmenter` 的实例。
8. 将视频源渲染到画布上。
9. 创建背景替换逻辑。
10. 创建 Webpack 配置文件。
11. 捆绑您的 JavaScript 文件。

安装 MediaPipe 和 Webpack

首先，请安装 `@mediapipe/tasks-vision` 和 `webpack` npm 包。下面的示例使用 Webpack 作为 JavaScript 捆绑程序；如果愿意，您可以使用不同的捆绑程序。

JavaScript

```
npm i @mediapipe/tasks-vision webpack webpack-cli
```

请务必更新您的 `package.json` 以指定 `webpack` 作为构建脚本：

JavaScript

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
```

创建 index.html

接下来，创建 HTML 样板，并将 Web 广播 SDK 作为脚本标签导入。在下面的代码中，请务必将 `<SDK version>` 替换为您正在使用的广播 SDK 版本。

JavaScript

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Import the SDK -->
  <script src="https://web-broadcast.live-video.net/<SDK version>/amazon-ivs-web-
broadcast.js"></script>
</head>

<body>

</body>
</html>
```

添加媒体元素

接下来，在正文标签内添加一个视频元素和两个画布元素。视频元素将包含您的实时相机源，并将用作 MediaPipe Image Segmenter 的输入。第一个画布元素将用于渲染将要广播的源的预览。第二个画布元素将用于渲染将用作背景的自定义图像。由于带有自定义图像的第二个画布仅用作以编程方式将像素从其复制到最终画布的来源，因此在视图中被隐藏。

JavaScript

```
<div class="row local-container">
      <video id="webcam" autoplay style="display: none"></video>
    </div>
    <div class="row local-container">
      <canvas id="canvas" width="640px" height="480px"></canvas>

      <div class="column" id="local-media"></div>
      <div class="static-controls hidden" id="local-controls">
        <button class="button" id="mic-control">Mute Mic</button>
        <button class="button" id="camera-control">Mute Camera</button>
      </div>
    </div>
    <div class="row local-container">
      <canvas id="background" width="640px" height="480px" style="display: none"></
canvas>
    </div>
```

添加脚本标签

添加脚本标签以加载捆绑的 JavaScript 文件，该文件将包含用于进行背景替换的代码并将其发布到舞台：

```
<script src="./dist/bundle.js"></script>
```

创建 app.js

接下来，创建一个 JavaScript 文件以获取在 HTML 页面中创建的画布和视频元素的元素对象。导入 `ImageSegmenter` 和 `FilesetResolver` 模块。`ImageSegmenter` 模块将用于执行分割任务。

JavaScript

```
const canvasElement = document.getElementById("canvas");
```

```
const background = document.getElementById("background");
const canvasCtx = canvasElement.getContext("2d");
const backgroundCtx = background.getContext("2d");
const video = document.getElementById("webcam");

import { ImageSegmenter, FilesetResolver } from "@mediapipe/tasks-vision";
```

接下来，创建一个调用 `init()` 的函数，用于从用户的摄像机中检索 MediaStream，并在每次摄像机画面完成加载时调用回调函数。为按钮添加事件侦听器以加入和离开舞台。

请注意，在加入舞台时，我们会传入一个名为 `segmentationStream` 的变量。这是从画布元素捕获的视频流，包含叠加在代表背景的自定义图像上的前景图像。稍后，此自定义流将用于创建 `LocalStageStream` 的实例，该实例可以发布到舞台。

JavaScript

```
const init = async () => {
  await initializeDeviceSelect();

  cameraButton.addEventListener("click", () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? "Show Camera" : "Hide Camera";
  });

  micButton.addEventListener("click", () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
    micButton.innerText = isMuted ? "Unmute Mic" : "Mute Mic";
  });

  localCamera = await getCamera(videoDevicesList.value);
  const segmentationStream = canvasElement.captureStream();

  joinButton.addEventListener("click", () => {
    joinStage(segmentationStream);
  });

  leaveButton.addEventListener("click", () => {
    leaveStage();
  });
};
```

加载自定义背景图像

在 init 函数的底部，添加用于调用名为 initBackgroundCanvas 的函数的代码，该函数从本地文件加载自定义图像并将其渲染到画布上。我们将在下一个步骤中定义此函数。将从用户相机检索到的 MediaStream 分配给视频对象。稍后，该视频对象将传递到 Image Segmenter。此外，还要设置一个名为 renderVideoToCanvas 的函数作为回调函数，以便在视频帧加载完毕时调用。我们将在稍后的步骤中定义此函数。

JavaScript

```
initBackgroundCanvas();

  video.srcObject = localCamera;
  video.addEventListener("loadeddata", renderVideoToCanvas);
```

让我们实现 initBackgroundCanvas 函数，它从本地文件加载图像。在此示例中，我们使用一张海滩的图像作为自定义背景。包含自定义图像的画布将从显示画面中隐藏，因为您将把它与包含相机源的画布元素的前景像素合并。

JavaScript

```
const initBackgroundCanvas = () => {
  let img = new Image();
  img.src = "beach.jpg";

  img.onload = () => {
    backgroundCtx.clearRect(0, 0, canvas.width, canvas.height);
    backgroundCtx.drawImage(img, 0, 0);
  };
};
```

创建 ImageSegmenter 实例

接下来，创建 ImageSegmenter 的实例，该实例将对图像进行分割并返回结果作为遮罩。在创建 ImageSegmenter 的实例时，您将使用自拍分割模型。

JavaScript

```
const createImageSegmenter = async () => {
  const audio = await FilesetResolver.forVisionTasks("https://cdn.jsdelivr.net/npm/
@mediapipe/tasks-vision@0.10.2/wasm");
```

```
  imageSegmenter = await ImageSegmenter.createFromOptions(audio, {
    baseOptions: {
      modelAssetPath: "https://storage.googleapis.com/mediapipe-models/image_segmenter/
selfie_segmenter/float16/latest/selfie_segmenter.tflite",
      delegate: "GPU",
    },
    runningMode: "VIDEO",
    outputCategoryMask: true,
  });
};
```

将视频源渲染到画布上

接下来，创建将视频源渲染到其他画布元素的函数。我们需要将视频源渲染到画布上，这样我们就可以使用 Canvas 2D API 从画布中提取前景像素。在执行此操作时，我们还会将视频帧传递给我们的 `ImageSegmenter` 实例，使用 segmentforVideo 方法在视频帧中分割前景与背景。当 segmentforVideo 方法返回时，它会调用我们的自定义回调函数 `replaceBackground` 来进行背景替换。

JavaScript

```
const renderVideoToCanvas = async () => {
  if (video.currentTime === lastWebcamTime) {
    window.requestAnimationFrame(renderVideoToCanvas);
    return;
  }
  lastWebcamTime = video.currentTime;
  canvasCtx.drawImage(video, 0, 0, video.videoWidth, video.videoHeight);

  if (imageSegmenter === undefined) {
    return;
  }

  let startTimeMs = performance.now();

  imageSegmenter.segmentForVideo(video, startTimeMs, replaceBackground);
};
```

创建背景替换逻辑

创建 `replaceBackground` 函数，它可将自定义背景图像与相机视频源中的前景合并，以替换背景。该函数首先从此前创建的两个画布元素中检索自定义背景图像和视频源的底层像素数据。然后，它会遍

历 ImageSegmenter 提供的遮罩，该遮罩指示前景中有哪些像素。当它遍历遮罩时，会有选择地将包含用户相机源的像素复制到相应的背景像素数据中。完成后，它会对最终的像素数据进行转换，并将前景复制到背景上，然后将其绘制到画布上。

JavaScript

```
function replaceBackground(result) {
  let imageData = canvasCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  let backgroundData = backgroundCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  const mask = result.categoryMask.getAsFloat32Array();
  let j = 0;

  for (let i = 0; i < mask.length; ++i) {
    const maskVal = Math.round(mask[i] * 255.0);

    j += 4;
  // Only copy pixels on to the background image if the mask indicates they are in the
 foreground
    if (maskVal < 255) {
      backgroundData[j] = imageData[j];
      backgroundData[j + 1] = imageData[j + 1];
      backgroundData[j + 2] = imageData[j + 2];
      backgroundData[j + 3] = imageData[j + 3];
    }
  }

 // Convert the pixel data to a format suitable to be drawn to a canvas
  const uint8Array = new Uint8ClampedArray(backgroundData.buffer);
  const dataNew = new ImageData(uint8Array, video.videoWidth, video.videoHeight);
  canvasCtx.putImageData(dataNew, 0, 0);
  window.requestAnimationFrame(renderVideoToCanvas);
}
```

作为参考，下面是包含上述所有逻辑的完整 app.js 文件：

JavaScript

```
/*! Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. SPDX-License-
Identifier: Apache-2.0 */

// All helpers are expose on 'media-devices.js' and 'dom.js'
```

```
const { setupParticipant } = window;

const { Stage, LocalStageStream, SubscribeType, StageEvents, ConnectionState,
 StreamType } = IVSBroadcastClient;
const canvasElement = document.getElementById("canvas");
const background = document.getElementById("background");
const canvasCtx = canvasElement.getContext("2d");
const backgroundCtx = background.getContext("2d");
const video = document.getElementById("webcam");

import { ImageSegmenter, FilesetResolver } from "@mediapipe/tasks-vision";

let cameraButton = document.getElementById("camera-control");
let micButton = document.getElementById("mic-control");
let joinButton = document.getElementById("join-button");
let leaveButton = document.getElementById("leave-button");

let controls = document.getElementById("local-controls");
let audioDevicesList = document.getElementById("audio-devices");
let videoDevicesList = document.getElementById("video-devices");

// Stage management
let stage;
let joining = false;
let connected = false;
let localCamera;
let localMic;
let cameraStageStream;
let micStageStream;
let imageSegmenter;
let lastWebcamTime = -1;

const init = async () => {
  await initializeDeviceSelect();

  cameraButton.addEventListener("click", () => {
    const isMuted = !cameraStageStream.isMuted;
    cameraStageStream.setMuted(isMuted);
    cameraButton.innerText = isMuted ? "Show Camera" : "Hide Camera";
  });

  micButton.addEventListener("click", () => {
    const isMuted = !micStageStream.isMuted;
    micStageStream.setMuted(isMuted);
```

```
    micButton.innerText = isMuted ? "Unmute Mic" : "Mute Mic";
  });

  localCamera = await getCamera(videoDevicesList.value);
  const segmentationStream = canvasElement.captureStream();

  joinButton.addEventListener("click", () => {
    joinStage(segmentationStream);
  });

  leaveButton.addEventListener("click", () => {
    leaveStage();
  });

  initBackgroundCanvas();

  video.srcObject = localCamera;
  video.addEventListener("loadeddata", renderVideoToCanvas);
};

const joinStage = async (segmentationStream) => {
  if (connected || joining) {
    return;
  }
  joining = true;

  const token = document.getElementById("token").value;

  if (!token) {
    window.alert("Please enter a participant token");
    joining = false;
    return;
  }

  // Retrieve the User Media currently set on the page
  localMic = await getMic(audioDevicesList.value);

  cameraStageStream = new LocalStageStream(segmentationStream.getVideoTracks()[0]);
  micStageStream = new LocalStageStream(localMic.getAudioTracks()[0]);

  const strategy = {
    stageStreamsToPublish() {
      return [cameraStageStream, micStageStream];
    },
```

```
    shouldPublishParticipant() {
      return true;
    },
    shouldSubscribeToParticipant() {
      return SubscribeType.AUDIO_VIDEO;
    },
  };

  stage = new Stage(token, strategy);

  // Other available events:
  // https://aws.github.io/amazon-ivs-web-broadcast/docs/sdk-guides/stages#events
  stage.on(StageEvents.STAGE_CONNECTION_STATE_CHANGED, (state) => {
    connected = state === ConnectionState.CONNECTED;

    if (connected) {
      joining = false;
      controls.classList.remove("hidden");
    } else {
      controls.classList.add("hidden");
    }
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_JOINED, (participant) => {
    console.log("Participant Joined:", participant);
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_STREAMS_ADDED, (participant, streams) => {
    console.log("Participant Media Added: ", participant, streams);

    let streamsToDisplay = streams;

    if (participant.isLocal) {
      // Ensure to exclude local audio streams, otherwise echo will occur
      streamsToDisplay = streams.filter((stream) => stream.streamType ===
StreamType.VIDEO);
    }

    const videoEl = setupParticipant(participant);
    streamsToDisplay.forEach((stream) =>
videoEl.srcObject.addTrack(stream.mediaStreamTrack));
  });

  stage.on(StageEvents.STAGE_PARTICIPANT_LEFT, (participant) => {
```

```
    console.log("Participant Left: ", participant);
    teardownParticipant(participant);
  });

  try {
    await stage.join();
  } catch (err) {
    joining = false;
    connected = false;
    console.error(err.message);
  }
};

const leaveStage = async () => {
  stage.leave();

  joining = false;
  connected = false;

  cameraButton.innerText = "Hide Camera";
  micButton.innerText = "Mute Mic";
  controls.classList.add("hidden");
};

function replaceBackground(result) {
  let imageData = canvasCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  let backgroundData = backgroundCtx.getImageData(0, 0, video.videoWidth,
 video.videoHeight).data;
  const mask = result.categoryMask.getAsFloat32Array();
  let j = 0;

  for (let i = 0; i < mask.length; ++i) {
    const maskVal = Math.round(mask[i] * 255.0);

    j += 4;
    if (maskVal < 255) {
      backgroundData[j] = imageData[j];
      backgroundData[j + 1] = imageData[j + 1];
      backgroundData[j + 2] = imageData[j + 2];
      backgroundData[j + 3] = imageData[j + 3];
    }
  }
  const uint8Array = new Uint8ClampedArray(backgroundData.buffer);
```

```
  const dataNew = new ImageData(uint8Array, video.videoWidth, video.videoHeight);
  canvasCtx.putImageData(dataNew, 0, 0);
  window.requestAnimationFrame(renderVideoToCanvas);
}

const createImageSegmenter = async () => {
  const audio = await FilesetResolver.forVisionTasks("https://cdn.jsdelivr.net/npm/
@mediapipe/tasks-vision@0.10.2/wasm");

  imageSegmenter = await ImageSegmenter.createFromOptions(audio, {
    baseOptions: {
      modelAssetPath: "https://storage.googleapis.com/mediapipe-models/image_segmenter/
selfie_segmenter/float16/latest/selfie_segmenter.tflite",
      delegate: "GPU",
    },
    runningMode: "VIDEO",
    outputCategoryMask: true,
  });
};

const renderVideoToCanvas = async () => {
  if (video.currentTime === lastWebcamTime) {
    window.requestAnimationFrame(renderVideoToCanvas);
    return;
  }
  lastWebcamTime = video.currentTime;
  canvasCtx.drawImage(video, 0, 0, video.videoWidth, video.videoHeight);

  if (imageSegmenter === undefined) {
    return;
  }

  let startTimeMs = performance.now();

  imageSegmenter.segmentForVideo(video, startTimeMs, replaceBackground);
};

const initBackgroundCanvas = () => {
  let img = new Image();
  img.src = "beach.jpg";

  img.onload = () => {
    backgroundCtx.clearRect(0, 0, canvas.width, canvas.height);
    backgroundCtx.drawImage(img, 0, 0);
```

```
  };
};

createImageSegmenter();
init();
```

创建 Webpack 配置文件

将此配置添加到要捆绑 `app.js` 的 Webpack 配置文件中，这样导入调用就会起作用：

JavaScript

```
const path = require("path");
module.exports = {
  entry: ["./app.js"],
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "dist"),
  },
};
```

捆绑您的 JavaScript 文件

```
npm run build
```

从包含 `index.html` 的目录中启动简单 HTTP 服务器并打开 `localhost:8000` 以查看结果：

```
python3 -m http.server -d ./
```

Android

要替换直播中的背景，可以使用 Google 机器学习套件的自拍分割 API。自拍分割 API 接受相机图像作为输入，并返回一个遮罩，该遮罩为图像的每个像素提供置信度分数，指示图像是在前景还是背景中。根据置信度分数，您可以从背景图像或前景图像检索相应的像素颜色。此流程一直持续到检查完遮罩中的所有置信度分数为止。结果是一个新像素颜色数组，其中包含前景像素以及来自背景图像的像素。

要将背景替换与 IVS 实时流式 Android 广播 SDK 集成，您需要：

1. 安装 CameraX 库和 Google 机器学习套件。
2. 初始化样板变量。

3. 创建自定义图像源。
4. 管理相机帧。
5. 将相机帧传递到 Google 机器学习套件。
6. 将相机帧前景叠加到您的自定义背景上。
7. 将新图像馈送到自定义图像源。

安装 CameraX 库和 Google 机器学习套件

要从实时相机源中提取图像，请使用 Android 的 CameraX 库。要安装 CameraX 库和 Google 机器学习套件，请将以下内容添加到您的模块的 `build.gradle` 文件中。分别将 `${camerax_version}` 和 `${google_ml_kit_version}` 替换为最新版本的 CameraX 和 Google 机器学习套件库。

Java

```
implementation "com.google.mlkit:segmentation-selfie:${google_ml_kit_version}"
implementation "androidx.camera:camera-core:${camerax_version}"
implementation "androidx.camera:camera-lifecycle:${camerax_version}"
```

导入以下库：

Java

```
import androidx.camera.core.CameraSelector
import androidx.camera.core.ImageAnalysis
import androidx.camera.core.ImageProxy
import androidx.camera.lifecycle.ProcessCameraProvider
import com.google.mlkit.vision.segmentation.selfie.SelfieSegmenterOptions
```

初始化样板变量

初始化 `ImageAnalysis` 的实例和 `ExecutorService` 的实例：

Java

```
private lateinit var binding: ActivityMainBinding
private lateinit var cameraExecutor: ExecutorService
private var analysisUseCase: ImageAnalysis? = null
```

在 STREAM_MODE 中初始化 Segmenter 实例：

Java

```
private val options =
        SelfieSegmenterOptions.Builder()
            .setDetectorMode(SelfieSegmenterOptions.STREAM_MODE)
            .build()

private val segmenter = Segmentation.getClient(options)
```

创建自定义图像源

在活动的 onCreate 方法中，创建 DeviceDiscovery 对象的实例并创建自定义图像源。自定义图像源提供的 Surface 将收到最终图像，前景叠加在自定义背景图像上。然后，您将使用自定义图像源创建 ImageLocalStageStream 的实例。然后，可以将 ImageLocalStageStream（在此例中名为 filterStream）的实例发布到舞台。有关设置舞台的说明，请参阅 IVS Android 广播 SDK 指南。最后，还要创建一个用于管理相机的线程。

Java

```
var deviceDiscovery = DeviceDiscovery(applicationContext)
var customSource = deviceDiscovery.createImageInputSource( BroadcastConfiguration.Vec2(
720F, 1280F
))
var surface: Surface = customSource.inputSurface
var filterStream = ImageLocalStageStream(customSource)

cameraExecutor = Executors.newSingleThreadExecutor()
```

管理相机帧

接下来，创建一个函数来初始化相机。此功能使用 CameraX 库从实时相机源中提取图像。首先，创建调用 cameraProviderFuture 的 ProcessCameraProvider 的实例。此对象表示获取摄像机提供者操作的未来结果。然后，将项目中的图像作为位图加载。此示例使用海滩图像作为背景，但它可以是您想使用的任何图像。

然后，向 cameraProviderFuture 添加一个侦听器。当相机可用时，或者在获取相机提供者的过程中出现错误时，系统会通知侦听器。

Java

```
private fun startCamera(surface: Surface) {
        val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
```

```
        val imageResource = R.drawable.beach
        val bgBitmap: Bitmap = BitmapFactory.decodeResource(resources, imageResource)
        var resultBitmap: Bitmap;


        cameraProviderFuture.addListener({
            val cameraProvider: ProcessCameraProvider = cameraProviderFuture.get()


                if (mediaImage != null) {
                    val inputImage =
                        InputImage.fromMediaImage(mediaImage,
imageProxy.imageInfo.rotationDegrees)

                            resultBitmap = overlayForeground(mask, maskWidth,
maskHeight, inputBitmap, backgroundPixels)
                            canvas = surface.lockCanvas(null);
                            canvas.drawBitmap(resultBitmap, 0f, 0f, null)

                            surface.unlockCanvasAndPost(canvas);

                        }
                        .addOnFailureListener { exception ->
                            Log.d("App", exception.message!!)
                        }
                        .addOnCompleteListener {
                            imageProxy.close()
                        }

                }
            };

            val cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA

            try {
                // Unbind use cases before rebinding
                cameraProvider.unbindAll()

                // Bind use cases to camera
                cameraProvider.bindToLifecycle(this, cameraSelector, analysisUseCase)

            } catch(exc: Exception) {
                Log.e(TAG, "Use case binding failed", exc)
            }
```

```
        }, ContextCompat.getMainExecutor(this))
    }
```

在侦听器中，创建 ImageAnalysis.Builder 以访问来自实时相机源的每个帧。将反向压力策略设置为 STRATEGY_KEEP_ONLY_LATEST。这样可以保证一次只能传输一个相机帧进行处理。将每个相机帧转换为位图，这样您就可以提取其像素，以便稍后将其与自定义背景图像合并。

Java

```
val imageAnalyzer = ImageAnalysis.Builder()
analysisUseCase = imageAnalyzer
    .setTargetResolution(Size(360, 640))
    .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
    .build()

analysisUseCase?.setAnalyzer(cameraExecutor) { imageProxy: ImageProxy ->
    val mediaImage = imageProxy.image
    val tempBitmap = imageProxy.toBitmap();
    val inputBitmap = tempBitmap.rotate(imageProxy.imageInfo.rotationDegrees.toFloat())
```

将相机帧传递到 Google 机器学习套件

接下来，创建一个 InputImage 并将其传递给 Segmenter 的实例进行处理。InputImage 可以从 ImageAnalysis 的实例提供的 ImageProxy 创建。向 Segmenter 提供 InputImage 后，它将返回一个遮罩，其置信度分数表示像素出现在前景或背景中的可能性。此遮罩还提供宽度和高度属性，您将使用这些属性来创建一个包含先前加载的自定义背景图像中的背景像素的新数组。

Java

```
if (mediaImage != null) {
        val inputImage =
            InputImage.fromMediaImag


segmenter.process(inputImage)
    .addOnSuccessListener { segmentationMask ->
        val mask = segmentationMask.buffer
        val maskWidth = segmentationMask.width
        val maskHeight = segmentationMask.height
        val backgroundPixels = IntArray(maskWidth * maskHeight)
        bgBitmap.getPixels(backgroundPixels, 0, maskWidth, 0, 0, maskWidth, maskHeight)
```

将相机帧前景叠加到您的自定义背景上

借助包含置信度分数的遮罩、作为位图的相机帧以及自定义背景图像中的彩色像素，您可以拥有将前景叠加到自定义背景所需的一切。然后使用以下参数调用 overlayForeground 函数：

Java

```
resultBitmap = overlayForeground(mask, maskWidth, maskHeight, inputBitmap,
 backgroundPixels)
```

此函数遍历遮罩并检查置信度值，以确定是从背景图像还是从相机帧中获取相应的像素颜色。如果置信度值表明遮罩中的像素很可能位于背景中，则将从背景图像中获得相应的像素颜色；否则，它将从相机帧中获取相应的像素颜色来构建前景。函数完成对遮罩的遍历后，将使用新的彩色像素数组创建一个新的位图并返回。这个新位图包含叠加在自定义背景上的前景。

Java

```
private fun overlayForeground(
        byteBuffer: ByteBuffer,
        maskWidth: Int,
        maskHeight: Int,
        cameraBitmap: Bitmap,
        backgroundPixels: IntArray
    ): Bitmap {
        @ColorInt val colors = IntArray(maskWidth * maskHeight)
        val cameraPixels = IntArray(maskWidth * maskHeight)

        cameraBitmap.getPixels(cameraPixels, 0, maskWidth, 0, 0, maskWidth, maskHeight)

        for (i in 0 until maskWidth * maskHeight) {
            val backgroundLikelihood: Float = 1 - byteBuffer.getFloat()

            // Apply the virtual background to the color if it's not part of the
 foreground
            if (backgroundLikelihood > 0.9) {
                // Get the corresponding pixel color from the background image
                // Set the color in the mask based on the background image pixel color
                colors[i] = backgroundPixels.get(i)
            } else {
                // Get the corresponding pixel color from the camera frame
                // Set the color in the mask based on the camera image pixel color
                colors[i] = cameraPixels.get(i)
            }
```

```
        }

        return Bitmap.createBitmap(
            colors, maskWidth, maskHeight, Bitmap.Config.ARGB_8888
        )
    }
```

将新图像馈送到自定义图像源

然后，您可以将新位图写入到自定义图像源提供的 Surface。这将把它广播到您的舞台。

Java

```
resultBitmap = overlayForeground(mask, inputBitmap, mutableBitmap, bgBitmap)
canvas = surface.lockCanvas(null);
canvas.drawBitmap(resultBitmap, 0f, 0f, null)
```

下面是获取相机帧、将其传递给 Segmenter 并叠加到背景上的完整功能：

Java

```
@androidx.annotation.OptIn(androidx.camera.core.ExperimentalGetImage::class)
    private fun startCamera(surface: Surface) {
        val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
        val imageResource = R.drawable.clouds
        val bgBitmap: Bitmap = BitmapFactory.decodeResource(resources, imageResource)
        var resultBitmap: Bitmap;

        cameraProviderFuture.addListener({
            // Used to bind the lifecycle of cameras to the lifecycle owner
            val cameraProvider: ProcessCameraProvider = cameraProviderFuture.get()

            val imageAnalyzer = ImageAnalysis.Builder()
            analysisUseCase = imageAnalyzer
                .setTargetResolution(Size(720, 1280))
                .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
                .build()

            analysisUseCase!!.setAnalyzer(cameraExecutor) { imageProxy: ImageProxy ->
                val mediaImage = imageProxy.image
                val tempBitmap = imageProxy.toBitmap();
                val inputBitmap =
 tempBitmap.rotate(imageProxy.imageInfo.rotationDegrees.toFloat())
```

```
                if (mediaImage != null) {
                    val inputImage =
                        InputImage.fromMediaImage(mediaImage,
imageProxy.imageInfo.rotationDegrees)

                    segmenter.process(inputImage)
                        .addOnSuccessListener { segmentationMask ->
                            val mask = segmentationMask.buffer
                            val maskWidth = segmentationMask.width
                            val maskHeight = segmentationMask.height
                            val backgroundPixels = IntArray(maskWidth * maskHeight)
                            bgBitmap.getPixels(backgroundPixels, 0, maskWidth, 0, 0,
maskWidth, maskHeight)

                            resultBitmap = overlayForeground(mask, maskWidth,
maskHeight, inputBitmap, backgroundPixels)
                            canvas = surface.lockCanvas(null);
                            canvas.drawBitmap(resultBitmap, 0f, 0f, null)

                            surface.unlockCanvasAndPost(canvas);

                        }
                        .addOnFailureListener { exception ->
                            Log.d("App", exception.message!!)
                        }
                        .addOnCompleteListener {
                            imageProxy.close()
                        }

                }
            };

            val cameraSelector = CameraSelector.DEFAULT_FRONT_CAMERA

            try {
                // Unbind use cases before rebinding
                cameraProvider.unbindAll()

                // Bind use cases to camera
                cameraProvider.bindToLifecycle(this, cameraSelector, analysisUseCase)

            } catch(exc: Exception) {
                Log.e(TAG, "Use case binding failed", exc)
```

```
            }

        }, ContextCompat.getMainExecutor(this))
    }
```

IVS 广播 SDK：移动音频模式（实时直播功能）

音频质量是任何真实团队媒体体验的重要组成部分，而且没有适合所有使用案例的“一刀切”音频配置。为了确保您的用户在收听 IVS 直播时获得最佳体验，我们的移动 SDK 提供了多种预设音频配置，并可根据需要提供更强大的自定义设置。

简介

IVS 移动广播 SDK 提供了一个 `StageAudioManager` 类。该类旨在成为控制两种平台上的底层音频模式的单一接触点。在 Android 上，它控制 AudioManager，包括音频模式、音频源、内容类型、使用情况和通信设备。在 iOS 上，它控制应用程序 AVAudioSession，以及 voiceProcessing 是否已启用。

重要提示：在 IVS 实时广播 SDK 处于活动状态时，请勿与 `AVAudioSession` 或 `AudioManager` 直接交互。那样可能会导致音频丢失，或者在错误的设备上录制或播放音频。

在创建第一个 `DeviceDiscovery` 或 `Stage` 对象之前，必须配置 `StageAudioManager` 类。

Android (Kotlin)

```
StageAudioManager.getInstance(context).setPreset(StageAudioManager.UseCasePreset.VIDEO_CHAT)
 The default value

val deviceDiscovery = DeviceDiscovery(context)
val stage = Stage(context, token, this)

// Other Stage implementation code
```

iOS (Swift)

```
IVSStageAudioManager.sharedInstance().setPreset(.videoChat) // The default value

let deviceDiscovery = IVSDeviceDiscovery()
let stage = try? IVSStage(token: token, strategy: self)
```

```
// Other Stage implementation code
```

如果在初始化 DeviceDiscovery 或 Stage 实例之前未在 StageAudioManager 上设置任何内容，则会自动应用 VideoChat 预设。

音频模式预设

实时广播 SDK 提供了三种预设，各自针对常见使用案例量身定制，如下文所述。对于每种预设，我们涵盖了五个关键类别，用于将预设相互区分开来。

视频聊天

这是默认预设，专为本地设备与其他参与者进行实时对话的应用场景设计。

类别	Android	iOS
回声消除	已启用	已启用
音量摇杆	通话音量	通话音量
麦克风选择	视操作系统进行限制。USB 麦克风可能不可用。	视操作系统进行限制。USB 和蓝牙麦克风可能不可用。 同时处理输入和输出的蓝牙耳机应可正常工作；例如 AirPods。
音频输出	任何输出设备都应正常工作。	视操作系统进行限制。有线耳机可能不可用。
音频质量	中/低。听起来像是打电话，不像播放媒体。	中/低。听起来像是打电话，不像播放媒体。

仅订阅

此预设适合用于您计划订阅其他发布参与者但不打算自己发布的情况。它专注于音频质量并支持所有可用的输出设备。

类别	Android	iOS
回声消除	禁用	禁用
音量摇杆	媒体音量	媒体音量
麦克风选择	不适用，此预设不适合用于发布。	不适用，此预设不适合用于发布。
音频输出	任何输出设备都应正常工作。	任何输出设备都应正常工作。
音频质量	高。任何媒体类型都应清晰，包括音乐。	高。任何媒体类型都应清晰，包括音乐。

Studio

此预设专为高质量订阅设计，同时保持发布能力。它需要录制和播放硬件来消除回声。此处的使用案例是使用 USB 麦克风和有线耳机。该 SDK 将保持最高质量的音频，同时依赖这些设备的物理隔离，以免产生回声。

类别	Android	iOS
回声消除	禁用	禁用
音量摇杆	大多数情况下的媒体音量。连接蓝牙麦克风时的通话音量。	媒体音量
麦克风选择	任何麦克风都应正常工作。	任何麦克风都应正常工作。
音频输出	任何输出设备都应正常工作。	任何输出设备都应正常工作。
音频质量	高。双方都应能够发送音乐，另一方也能清晰地听见。 连接蓝牙耳机后，由于启用了蓝牙 SCO 模式，音频质量将会下降。	高。双方都应能够发送音乐，另一方也能清晰地听见。 连接蓝牙耳机后，由于启用了蓝牙 SCO 模式，音频质量可能会下降，这取决于耳机。

高级使用案例

除了预设之外，iOS 和 Android 实时流式广播 SDK 都允许配置底层平台的音频模式：

- 在 Android 上，设置 AudioSource、Usage 和 ContentType。
- 在 iOS 上，使用 AVAudioSession.Category、AVAudioSession.CategoryOptions、AVAudioSession.Mode，以及在发布时切换是否启用语音处理的功能。

Android (Kotlin)

```
// This would act similar to the Subscribe Only preset, but it uses a different
 ContentType.
StageAudioManager.getInstance(context)
    .setConfiguration(StageAudioManager.Source.GENERIC,
                      StageAudioManager.ContentType.MOVIE,
                      StageAudioManager.Usage.MEDIA);

val stage = Stage(context, token, this)

// Other Stage implementation code
```

iOS (Swift)

```
// This would act similar to the Subscribe Only preset, but it uses a different mode
 and options.
IVSStageAudioManager.sharedInstance()
    .setCategory(.playback,
                 options: [.duckOthers, .mixWithOthers],
                 mode: .default)

let stage = try? IVSStage(token: token, strategy: self)

// Other Stage implementation code
```

在 Android 系统上使用蓝牙发布

满足以下条件时，SDK 将自动恢复为 Android 上的 `VIDEO_CHAT` 预设：

- 分配的配置不使用 `VOICE_COMMUNICATION` 使用值。
- 蓝牙麦克风已连接到设备。
- 本地参与者正在向舞台发布内容。

这是 Android 操作系统在如何使用蓝牙耳机录制音频方面的限制。

与其他 SDK 集成

由于 iOS 和 Android 对每个应用都仅支持一种活动音频模式，因此如果您的应用使用多个需要控制音频模式的 SDK，则经常会遇到冲突。当您遇到这些冲突时，可以尝试一些常见的解决策略，如下所述。

匹配音频模式值

使用 IVS SDK 的高级音频配置选项或其他 SDK 的功能，使这两种 SDK 在底层值上保持一致。

Agora

iOS

在 iOS 上，让 Agora SDK 将 `AVAudioSession` 保持活动状态可防止在 IVS 实时流式广播 SDK 使用它时它被停用。

```
myRtcEngine.SetParameters("{\"che.audio.keep.audiosession\":true}");
```

Android

使用 IVS 实时流式广播 SDK 时，避免对 `RtcEngine` 调用 `setEnableSpeakerphone`，而应调用 `enableLocalAudio(false)`。当 IVS SDK 未发布时，您可以再次调用 `enableLocalAudio(true)`。

将 Amazon EventBridge 与 IVS 实时直播功能结合使用

您可以使用 Amazon EventBridge 来监控您的 Amazon Interactive Video Service (IVS) 流。

Amazon IVS 将有关流状态的更改事件发送到 Amazon EventBridge。传递的所有事件都有效。但是，事件将尽最大努力发出，这意味着并不能保证：

- 传送事件：会发生指定的事件（例如，参与者发布的事件），但 Amazon IVS 可能不会向 EventBridge 发送相应的事件。Amazon IVS 尝试在放弃之前传递几个小时的事件。
- 事件将在指定的时间范围内传递 – 您可能会收到几个小时之前的事件。
- 按顺序传送事件：事件可能无序，尤其是在短时间内相互发送的情况下。例如，您可以在参与者发布之前看到参与者未发布的事件。

尽管事件丢失、延迟或无序的情况很少，但如果您编写了取决于通知事件的顺序或存在的关键业务程序，则应处理这些可能性。

您可以为以下任何事件创建 EventBridge 规则。

事件类型	活动	发送时间：
IVS 合成状态更改	目标故障	尝试输出到目标失败。例如，由于没有流密钥或正在进行其他广播，向某个频道广播失败。
IVS 合成状态更改	目的地开始	输出到目标成功启动。
IVS 合成状态更改	目标结束	输出到目标已完成。
IVS 合成状态更改	目标重新连接	向目标的输出中断，正在尝试重新连接。
IVS 合成状态更改	会话开始	合成会话已创建。合成进程管道初始化成功时触发此事件。此时，合成管道已成功订阅舞台，正在接收媒体并能够合成视频。
IVS 合成状态更改	会话结束	合成会话已完成。

事件类型	活动	发送时间：
IVS 合成状态更改	会话失败	由于舞台资源不可用或任何其他内部错误，合成管道无法初始化。
IVS 舞台更新	参与者已发布	参与者开始发布到舞台。
IVS 舞台更新	参与者已取消发布	参与者已停止发布到舞台。

为 Amazon IVS 创建 Amazon EventBridge 规则

您可以创建针对 Amazon IVS 发出的事件进行触发的规则。请按照 Amazon EventBridge User Guide 中的 Create a rule in Amazon EventBridge 步骤操作。选择服务时，选择 Interactive Video Service（IVS）。

示例：合成状态更改

目标失败：尝试输出到目标失败时发送此事件。例如，由于没有流密钥或正在进行其他广播，向某个频道广播失败。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Failure",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
     "reason": "eg. stream key invalid"
   }
}
```

目标启动：成功启动向目标的输出时发送此事件。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Start",
     "stage_arn": "<stage-arn>",
     "id": "<destination-id>",
   }
}
```

目标结束：向目标输出完成发送此事件。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination End",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
   }
}
```

目标重新连接：向目标的输出中断并且正在尝试重新连接时，发送此事件。

```
{
```

```
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Destination Reconnecting",
     "stage_arn": "<stage-arn>",
     "id": "<Destination-id>",
   }
}
```

会话开始：创建合成会话时发送此事件。合成进程管道初始化成功时触发此事件。此时，合成管道已成功订阅舞台，正在接收媒体并能够合成视频。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session Start",
     "stage_arn": "<stage-arn>"
   }
}
```

会话结束：合成会话完成并且删除了所有资源时，发送此事件。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
```

```
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session End",
     "stage_arn": "<stage-arn>"
   }
}
```

会话失败：由于舞台资源不可用、舞台中没有参与者或任何其他内部错误而导致合成管道无法初始化时，发送此事件。

```
{
   "version": "0",
   "id": "01234567-0123-0123-0123-012345678901",
   "detail-type": "IVS Composition State Change",
   "source": "aws.ivs",
   "account": "aws_account_id",
   "time": "2017-06-12T10:23:43Z",
   "region": "us-east-1",
   "resources": [
     "arn:aws:ivs:us-east-1:aws_account_id:composition/123456789012"
   ],
   "detail": {
     "event_name": "Session Failure",
     "stage_arn": "<stage-arn>",
     "reason": "eg. no participants in the stage"
   }
}
```

示例：舞台更新

舞台更新事件包括事件名称（用于对事件进行分类）和有关该事件的元数据。元数据包括触发事件的参与者 ID、相关舞台和会话 ID 以及用户 ID。

参与者已发布：在参与者开始发布到舞台时发送该事件。

```
{
```

```
    "version": "0",
    "id": "12345678-1a23-4567-a1bc-1a2b34567890",
    "detail-type": "IVS Stage Update",
    "source": "aws.ivs",
    "account": "123456789012",
    "time": "2020-06-23T20:12:36Z",
    "region": "us-west-2",
    "resources": [
        "arn:aws:ivs:us-west-2:123456789012:stage/AbCdef1G2hij"
    ],
    "detail": {
        "session_id": "st-1234567890",
        "event_name": "Participant Published",
        "user_id": "Your User Id",
        "participant_id": "xYz1c2d3e4f"
    }
}
```

参与者已取消发布：在参与者已停止发布到舞台时发送该事件。

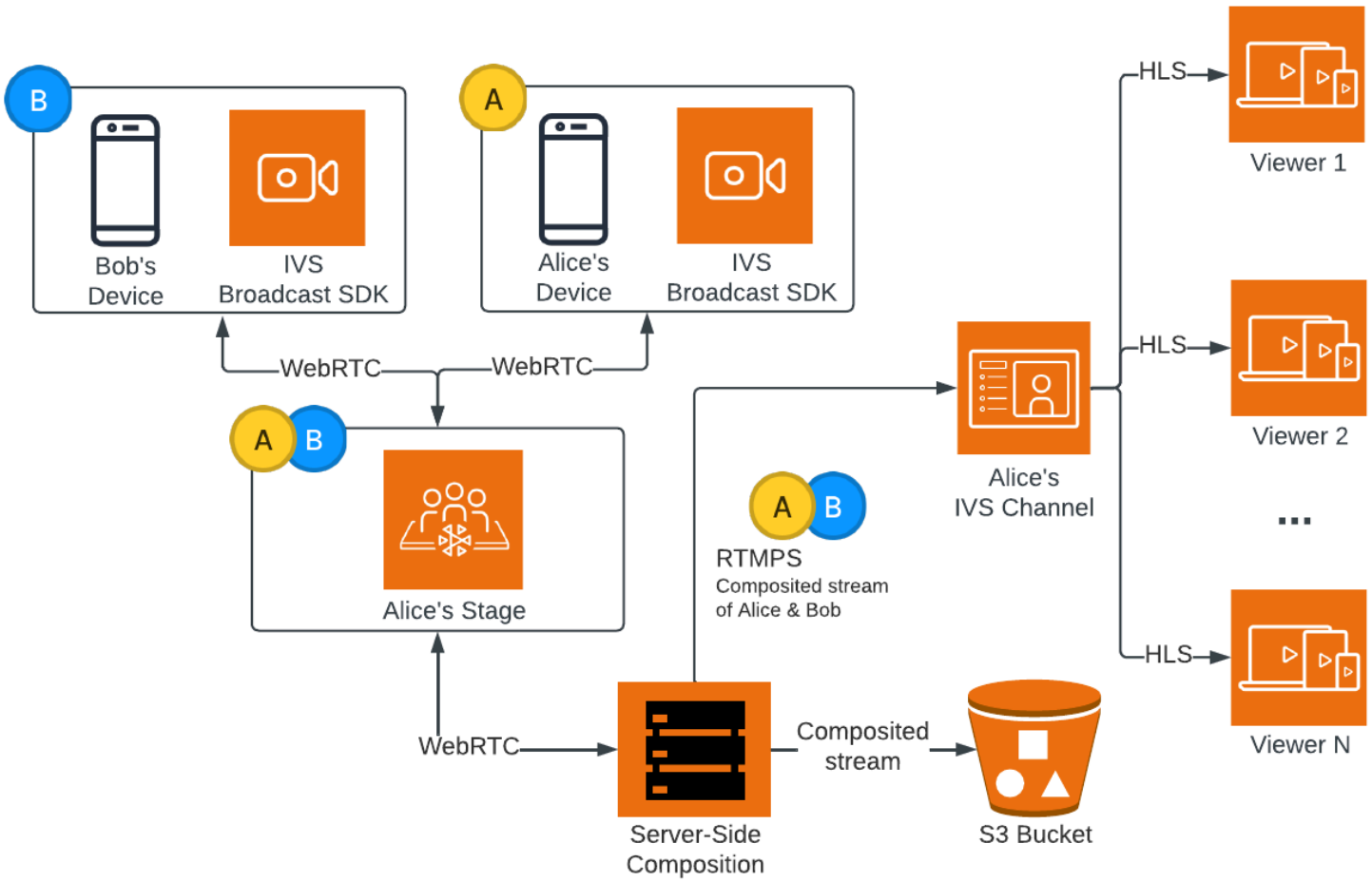
```
{
    "version": "0",
    "id": "12345678-1a23-4567-a1bc-1a2b34567890",
    "detail-type": "IVS Stage Update",
    "source": "aws.ivs",
    "account": "123456789012",
    "time": "2020-06-23T20:12:36Z",
    "region": "us-west-2",
    "resources": [
        "arn:aws:ivs:us-west-2:123456789012:stage/AbCdef1G2hij"
    ],
    "detail": {
        "session_id": "st-1234567890",
        "event_name": "Participant Unpublished",
        "user_id": "Your User Id",
        "participant_id": "xYz1c2d3e4f"
    }
}
```

服务器端合成（实时直播功能）

服务器端合成使用 IVS 服务器混合所有舞台参与者的音频和视频，然后将此混合视频发送到 IVS 频道（例如，为了服务于更多观众）或 S3 存储桶。服务器端合成通过舞台主区域的 IVS 控制面板端点调用。

使用服务器端合成广播或录制舞台有很多好处，对于寻求高效而可靠的云端视频工作流程的用户来说，是一个有吸引力的选择。

下图演示了服务器端合成的工作原理：



优点

与客户端合成相比，服务器端合成具有以下优点：

- 减少客户端负载 — 通过服务器端合成，处理和组合音频和视频源的负担从单个客户端设备转移到服务器本身。服务器端合成使客户端设备无需使用其 CPU 和网络资源来合成视图并将其传输到 IVS。

这意味着观众无需自己的设备处理资源密集型任务即可观看广播，从而延长电池寿命并获得更流畅的观看体验。

- 质量稳定 — 服务器端合成允许精确控制最终流的质量、分辨率和比特率。这样可确保所有观众都能获得一致的观看体验，而与其个人设备的功能无关。
- 弹性 — 通过将合成过程集中到服务器上，广播变得更加强大。即使发布者设备遇到技术限制或波动，服务器也可以进行调整，为所有受众提供更流畅的流。
- 带宽效率 — 由于服务器负责合成，因此舞台发布者不必花费额外的带宽将视频广播到 IVS。

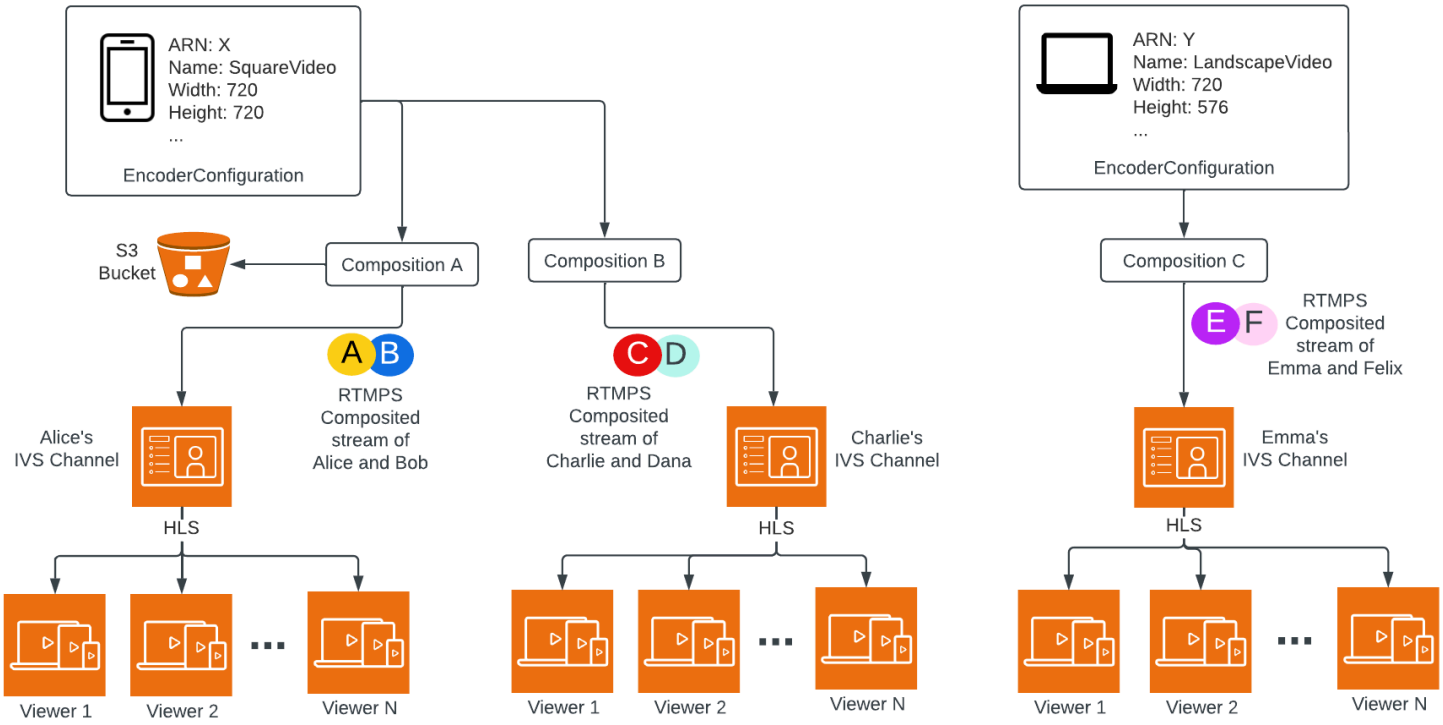
或者，要向 IVS 频道广播舞台，可以在客户端进行合成；请参阅 IVS Low-Latency Streaming User Guide 中的 Enabling Multiple Hosts on an IVS Stream。

IVS API

服务器端合成使用以下关键 API 元素：

- EncoderConfiguration 对象允许您自定义要生成的视频的格式（高度、宽度、比特率和其他流式传输参数）。每次调用 StartComposition 端点时，都可以重复使用 EncoderConfiguration。
- Composition 端点跟踪视频合成并输出到 IVS 频道。
- StorageConfiguration 会跟踪录制合成内容的 S3 存储桶。

要使用服务器端合成，您需要创建一个 EncoderConfiguration 并在调用 StartComposition 端点时将其附上。在此示例中，SquareVideo EncoderConfiguration 用于两个合成：

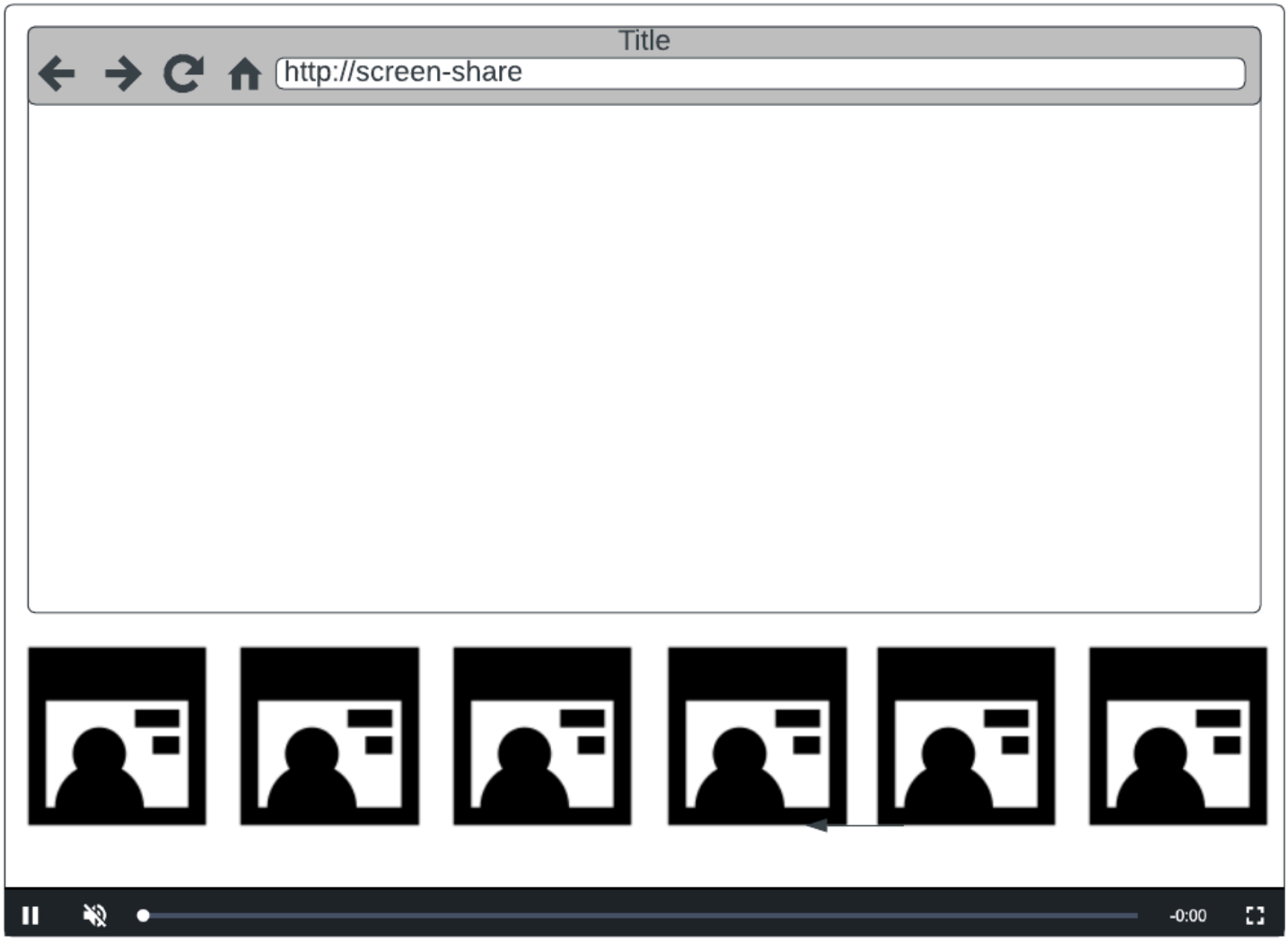


如需完整信息，请参阅 IVS Real-Time Streaming API Reference。

Layouts

默认情况下，服务器端合成功能使用网格布局将舞台参与者排列在大小相等的槽位中：

此布局为客户提供了配置和调用精选槽位的选项。精选槽位显示在主屏幕上，其他参与者显示在大小相等的槽位下方：



注意：舞台发布者在服务器端合成中支持的最大分辨率为 1080p。如果发布者发送的视频高于 1080p，则发布者将呈现为纯音频参与者。

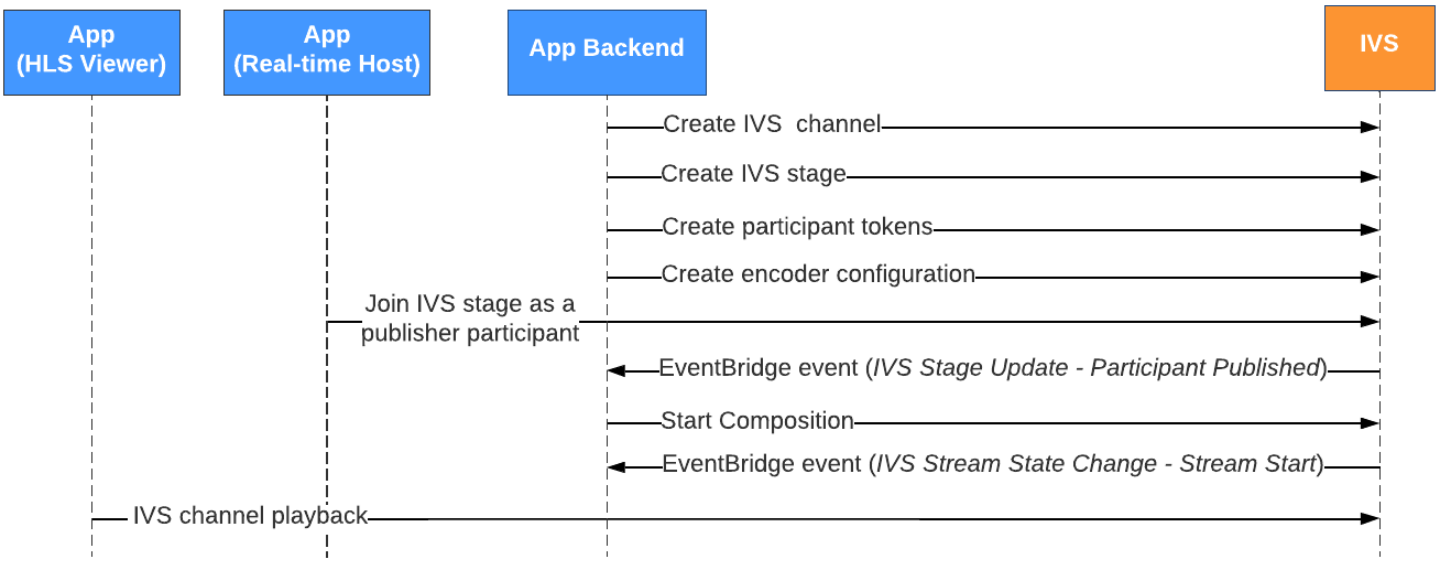
入门

先决条件

要使用服务器端合成，您必须有一个包含活跃发布者的舞台，并使用 IVS 频道和/或 S3 存储桶作为合成目标。下面，我们将介绍一种可能的工作流程，它使用 EventBridge 事件来开始合成，当参与者发布时，该合成将舞台广播到 IVS 频道。或者，您可以根据自己的应用程序逻辑开始和停止合成。请参阅合成录制以获得另一个示例，它展示了如何使用服务器端合成将舞台直接录制到 S3 存储桶。

1. 创建 IVS 频道。请参阅 Amazon IVS 低延迟直播功能入门。
2. 为每个发布者创建 IVS 舞台和参与者令牌。

3. 创建 EncoderConfiguration。
4. 加入舞台并发布到该舞台。（请参阅实时流式广播 SDK 指南的“发布和订阅”部分：Web、Android 和 iOS。）
5. 当您收到参与者发布的 EventBridge 事件时，请调用 StartComposition。
6. 等待几秒钟，然后在频道回放中观看合成视图。



注意：发布者参与者在舞台上处于非活动状态 60 秒后，合成执行自动关闭。此时，合成终止，并转换到 STOPPED 状态。合成处于 STOPPED 状态几分钟后将自动删除。

CLI 说明

使用 Amazon CLI 是一个高级选项，需要先在计算机上下载并配置 CLI。有关详细信息，请参阅 Amazon 命令行界面用户指南。

现在，您可以使用 CLI 创建和管理资源。合成端点位于 `ivs-realtime` 命名空间下。

创建 EncoderConfiguration 资源

EncoderConfiguration 是一个对象，允许您自定义生成的视频的格式（高度、宽度、比特率和其他流式传输参数）。每次调用合成端点时，您都可以重复使用 EncoderConfiguration，如下一步中所述。

下面的命令创建 EncoderConfiguration 资源，用于配置服务器端视频合成参数，例如视频比特率、帧速率和分辨率：

```
aws ivs-realtime create-encoder-configuration --name "MyEncoderConfig" --video
 "bitrate=2500000,height=720,width=1280,framerate=30"
```

响应如下：

```
{
   "encoderConfiguration": {
      "arn": "arn:aws:ivs:us-east-1:927810967299:encoder-configuration/9W59OBY2M8s4",
      "name": "MyEncoderConfig",
      "tags": {},
      "video": {
         "bitrate": 2500000,
         "framerate": 30,
         "height": 720,
         "width": 1280
      }
   }
}
```

开始合成

使用上面响应中提供的 EncoderConfiguration ARN，创建您的合成资源：

```
aws ivs-realtime start-composition --stage-arn "arn:aws:ivs:us-
east-1:927810967299:stage/8faHz1SQp0ik" --destinations  '[{"channel": {"channelArn":
 "arn:aws:ivs:us-east-1:927810967299:channel/DOlMW4dfMR8r", "encoderConfigurationArn":
 "arn:aws:ivs:us-east-1:927810967299:encoder-configuration/9W59OBY2M8s4"}}]'
```

响应将显示该合成使用 STARTING 状态创建。一旦合成开始发布合成，状态就会转换为 ACTIVE。（您可以通过调用 ListCompositions 或 GetComposition 端点来查看状态。）

合成处于 ACTIVE 后，可以使用 ListCompositions 在 IVS 频道上看到 IVS 舞台的合成视图：

```
aws ivs-realtime list-compositions
```

响应如下：

```
{
   "compositions": [
```

```
      {
         "arn": "arn:aws:ivs:us-east-1:927810967299:composition/YVoaXkKdEdRP",
         "destinations": [
            {
               "id": "bD9rRoN91fHU",
               "startTime": "2023-09-21T15:38:39+00:00",
               "state": "ACTIVE"
            }
         ],
         "stageArn": "arn:aws:ivs:us-east-1:927810967299:stage/8faHz1SQp0ik",
         "startTime": "2023-09-21T15:38:37+00:00",
         "state": "ACTIVE",
         "tags": {}
      }
   ]
}
```

注意：您需要让发布者参与者主动发布到舞台上，以使合成保持活动状态。有关更多信息，请参阅实时流式广播 SDK 指南的“发布和订阅”部分：Web、Android 和 iOS。您必须为每位参与者创建不同的舞台令牌。

启用屏幕共享

要使用固定的屏幕共享布局，请执行以下步骤。

创建 EncoderConfiguration 资源

下面的命令创建 EncoderConfiguration 资源，用于配置服务器端合成参数（视频比特率、帧速率和分辨率）。

```
aws ivs-realtime create-encoder-configuration --name "test-ssc-with-screen-share" --
video={bitrate=2000000,framerate=30,height=720,width=1280}
```

使用 `screen-share` 属性创建舞台参与者令牌。由于我们将指定 `screen-share` 作为 `featured` 槽位的名称，因此我们需要创建一个 `screen-share` 属性设置为 `true` 的舞台令牌：

```
aws ivs-realtime create-participant-token --stage-arn "arn:aws:ivs:us-
east-1:123456789012:stage/u9OiE29bT7Xp" --attributes screen-share=true
```

响应如下：

```
{
   "participantToken": {
      "attributes": {
         "screen-share": "true"
      },
      "expirationTime": "2023-08-04T05:26:11+00:00",
      "participantId": "E813MFklPWLF",
      "token":
 "eyJhbGciOiJLTVMiLCJ0eXAiOiJKV1QifQ.eyJleHAiOjE2OTExMjY3NzEsImlhdCI6MTY5MTA4MzU3MSwianRpIjoiRT
   }
}
```

开始合成

要使用屏幕共享功能开始合成，我们使用以下命令：

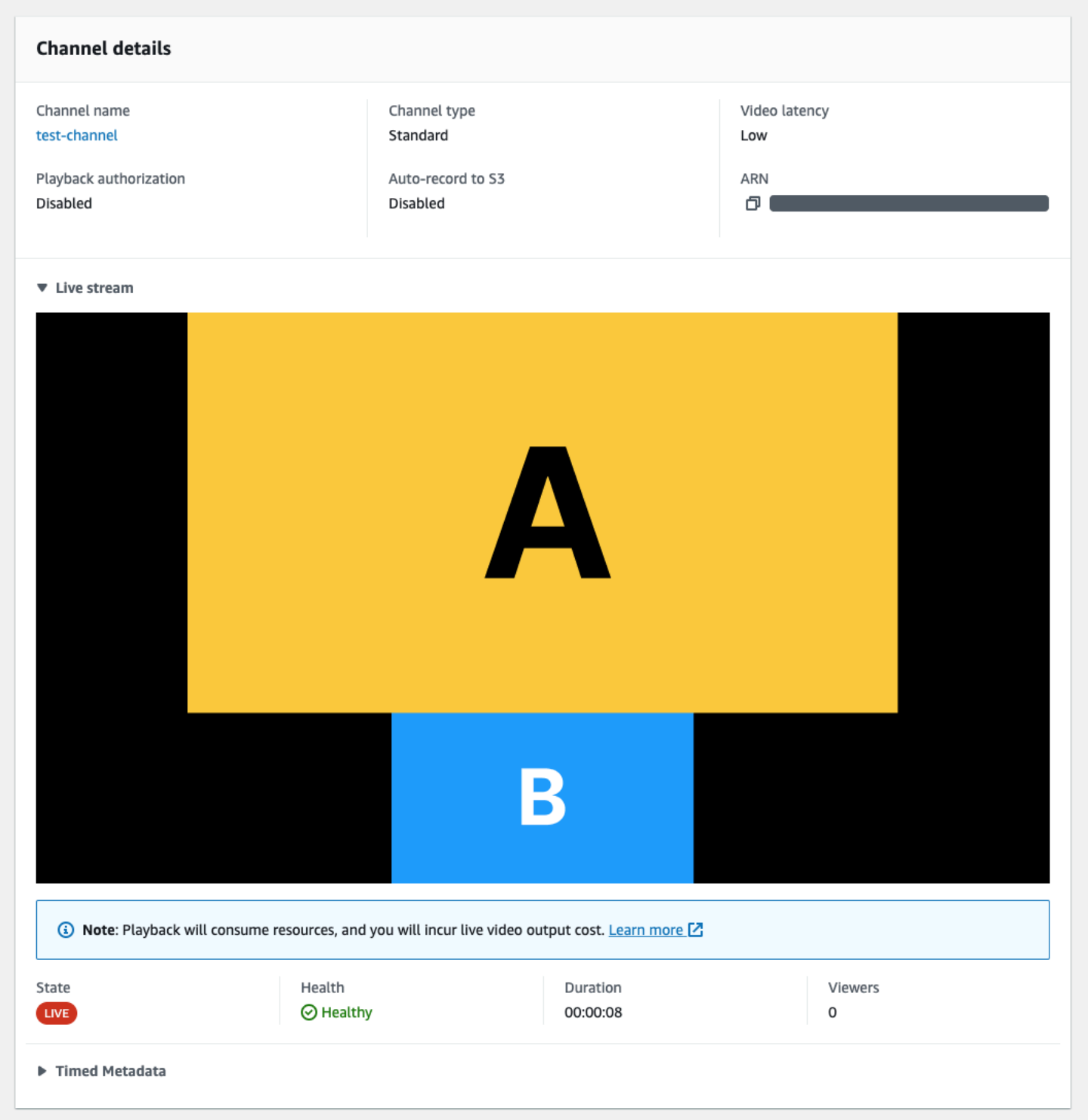
```
aws ivs-realtime start-composition --stage-arn "arn:aws:ivs:us-
east-1:927810967299:stage/8faHz1SQp0ik" --destinations  '[{"channel": {"channelArn":
 "arn:aws:ivs:us-east-1:927810967299:channel/DOlMW4dfMR8r", "encoderConfigurationArn":
 "arn:aws:ivs:us-east-1:927810967299:encoder-configuration/DEkQHWPVaOwO"}}]' --layout
 "grid={featuredParticipantAttribute=screen-share}"
```

响应如下：

```
{
   "composition" : {
      "arn" : "arn:aws:ivs:us-east-1:927810967299:composition/B19tQcXRgtoz",
      "destinations" : [ {
         "configuration" : {
            "channel" : {
               "channelArn" : "arn:aws:ivs:us-east-1:927810967299:channel/
DOlMW4dfMR8r",
               "encoderConfigurationArn" : "arn:aws:ivs:us-east-1:927810967299:encoder-
configuration/DEkQHWPVaOwO"
            },
            "name" : ""
         },
         "id" : "SGmgBXTULuXv",
         "state" : "STARTING"
      } ],
      "layout" : {
         "grid" : {
```

```
            "featuredParticipantAttribute" : "screen-share"
         }
      },
      "stageArn" : "arn:aws:ivs:us-east-1:927810967299:stage/8faHz1SQp0ik",
      "startTime" : "2023-09-27T21:32:38Z",
      "state" : "STARTING",
      "tags" : { }
   }
}
```

当舞台参与者 E813MFklPWLF 加入舞台时，该参与者的视频将显示在精选槽位中，所有其他舞台发布者将在呈现在该槽位下方：



停止合成

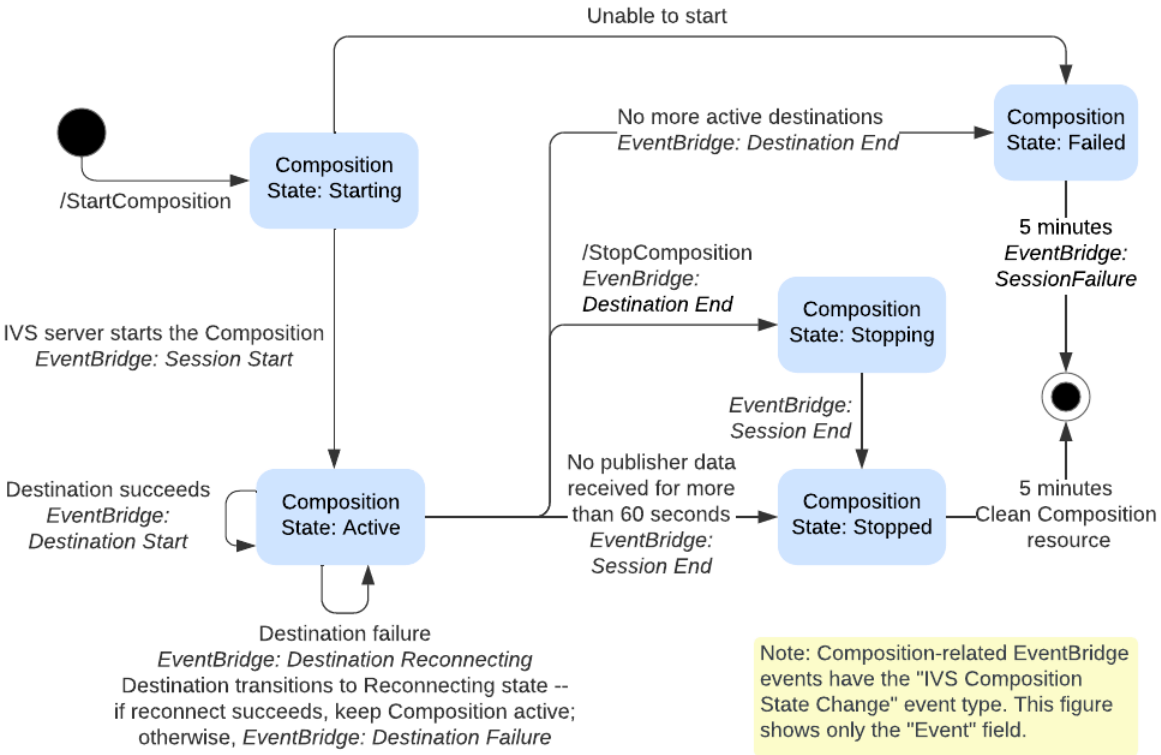
要随时停止合成，请调用 StopComposition 端点：

```
aws ivs-realtime stop-composition --arn arn:aws:ivs:us-east-1:927810967299:composition/
B19tQcXRgtoz
```

合成生命周期

使用下图来了解合成的状态转换。概括来说，合成的生命周期如下所示：

1. 合成资源在用户调用 StartComposition 端点时创建
2. IVS 成功开始合成之后，就会发送“IVS 合成状态更改（会话开始）”EventBridge 事件。有关事件的详细信息，请参阅将 EventBridge 与 IVS 实时直播功能结合使用。
3. 合成处于活动状态后，就会发生以下情况：
   - 用户停止合成 — 如果调用 StopComposition 端点，IVS 会启动合成的正常关闭，发送“目标结束”事件，然后发送“会话结束”事件。
   - 合成执行自动关闭 — 如果没有参与者主动发布到 IVS 舞台，则合成将在 60 秒后自动完成，并发送 EventBridge 事件。
   - 目标故障 — 如果目标意外失败（例如，IVS 频道被删除），则目标将转换为 `RECONNECTING` 状态并发送“目标重新连接”事件。如果无法恢复，IVS 会将目标转换为 `FAILED` 状态并发送“目标故障”事件。如果至少有一个目标处于活动状态，IVS 会使合成保持活动状态。
4. 一旦合成处于 `STOPPED` 或 `FAILED` 状态，就会在五分钟后自动将其清除。（然后，ListCompositions 或 GetComposition 将不再对其进行检索。）


Unable to start
/StartComposition
Composition State: Starting
No more active destinations
EventBridge: Destination End
Composition State: Failed
5 minutes
EventBridge: SessionFailure
/StopComposition
EvenBridge: Destination End
Composition State: Stopping
IVS server starts the Composition
EventBridge: Session Start
EventBridge: Session End
Destination succeeds
EventBridge: Destination Start
Composition State: Active
No publisher data received for more than 60 seconds
EventBridge: Session End
Composition State: Stopped
5 minutes
Clean Composition resource
Destination failure
EventBridge: Destination Reconnecting
Destination transitions to Reconnecting state -- if reconnect succeeds, keep Composition active; otherwise, EventBridge: Destination Failure
Note: Composition-related EventBridge events have the "IVS Composition State Change" event type. This figure shows only the "Event" field.

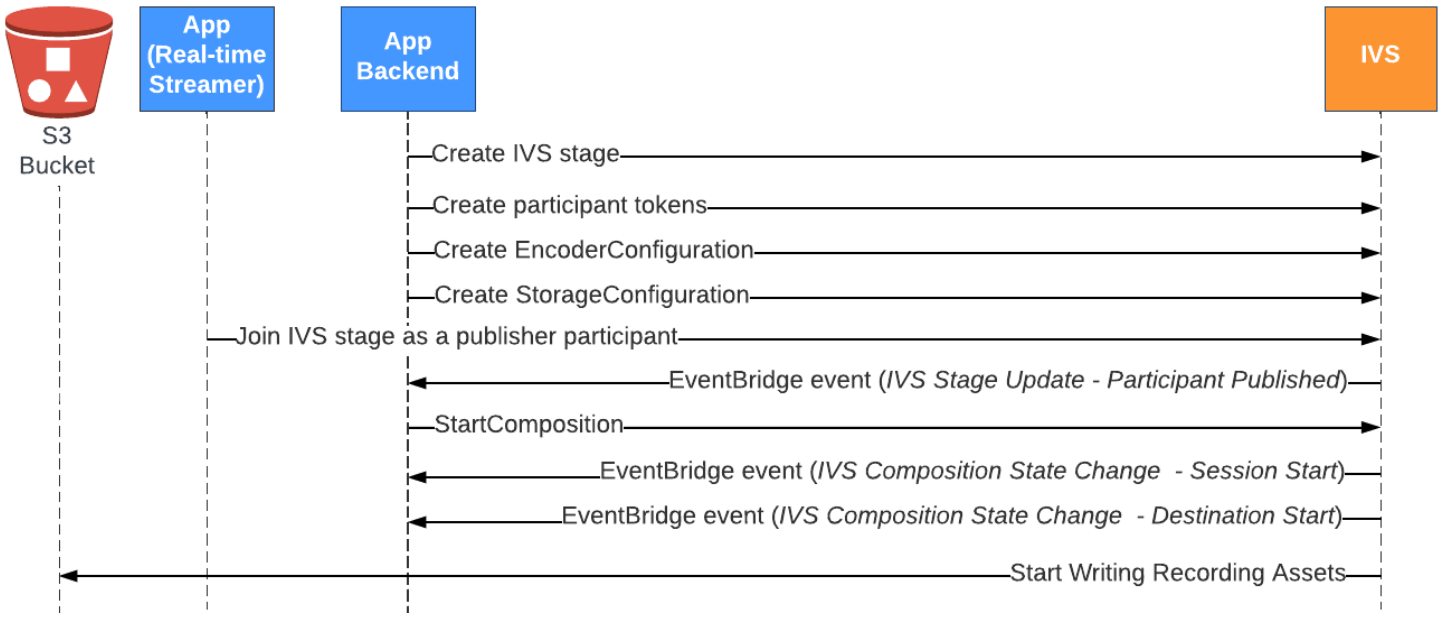
合成录制（实时直播功能）

本文档介绍如何在服务器端合成中使用合成录制功能。合成录制允许您使用 IVS 服务器将所有舞台发布者有效地组合到一个视图中，然后将生成的视频保存到 S3 存储桶中，从而生成 IVS 舞台的 HLS 录制。

先决条件

要使用复合录制，您必须有一个包含活跃发布者的舞台和一个 S3 存储桶作为录制目的地。下面，我们将介绍一种可能的工作流程，该工作流程使用 EventBridge 事件将合成记录到 S3 存储桶中。或者，您可以根据自己的应用程序逻辑开始和停止合成。

1. 为每个发布者创建 IVS 舞台和参与者令牌。
2. 创建 EncoderConfiguration（一个表示应如何渲染录制视频的对象）。
3. 创建 S3 存储桶和 StorageConfiguration（将在其中存储录制内容）。
4. 加入舞台并发布到该舞台。
5. 当您收到 “参与者已发布” EventBridge 事件时，StartComposition以 S3 DestinationConfiguration 对象作为目标进行调用
6. 几秒钟后，您应能看到 HLS 分段已保留到您的 S3 存储桶中。



注意：发布者参与者在舞台上处于非活动状态 60 秒后，合成执行自动关闭。此时，合成终止，并转换到 STOPPED 状态。合成处于 STOPPED 状态几分钟后将自动删除。有关详细信息，请参阅服务器端合成中的合成生命周期。

复合录制示例： StartComposition 使用 S3 存储桶目的地

以下示例显示了对StartComposition端点的典型调用，将 S3 指定为组合的唯一目的地。合成转换到 ACTIVE 状态后，视频片段和元数据将开始写入 storageConfiguration 对象指定的 S3 存储桶。要创建具有不同布局的合成，请参阅服务器端合成中的“布局”以及 IVS Real-Time Streaming API Reference。

Request

```
POST /StartComposition HTTP/1.1
Content-type: application/json

{
   "destinations": [
      {
         "s3": {
            "encoderConfigurationArns": [
              "arn:aws:ivs:ap-northeast-1:927810967299:encoder-configuration/
PAAwglkRtjge"
            ],
            "storageConfigurationArn": "arn:aws:ivs:ap-
northeast-1:927810967299:storage-configuration/ZBcEbgbE24Cq"
         }
      }
   ],
   "idempotencyToken": "db1i782f1g9",
   "stageArn": "arn:aws:ivs:ap-northeast-1:927810967299:stage/WyGkzNFGwiwr"
}
```

响应

```
{
    "composition": {
        "arn": "arn:aws:ivs:ap-northeast-1:927810967299:composition/s2AdaGUbvQgp",
        "destinations": [
            {
                "configuration": {
                    "name": "",
```

```
                    "s3": {
                        "encoderConfigurationArns": [
                            "arn:aws:ivs:ap-northeast-1:927810967299:encoder-
configuration/PAAwglkRtjge"
                        ],
                        "recordingConfiguration": {
                            "format": "HLS"
                        },
                        "storageConfigurationArn": "arn:aws:ivs:ap-
northeast-1:927810967299:storage-configuration/ZBcEbgbE24Cq"
                    }
                },
                "detail": {
                    "s3": {
                        "recordingPrefix": "MNALAcH9j2EJ/s2AdaGUbvQgp/2pBRKrNgX1ff/
composite"
                    }
                },
                "id": "2pBRKrNgX1ff",
                "state": "STARTING"
            }
        ],
        "layout": null,
        "stageArn": "arn:aws:ivs:ap-northeast-1:927810967299:stage/WyGkzNFGwiwr",
        "startTime": "2023-11-01T06:25:37Z",
        "state": "STARTING",
        "tags": {}
    }
}
```


StartComposition 响应中存在的`recordingPrefix`字段可用于确定录音内容的存储位置。

录制内容

当合成转换到`ACTIVE`状态时，您将开始看到 HLS 视频片段和元数据文件正在写入调用 StartComposition时提供的 S3 存储桶。这些内容可用于后处理或作为按需视频播放。

请注意，在合成变成实时后，会发出一个“IVS 合成状态更改”事件，可能需要一点时间写入清单文件和视频段。我们建议仅在收到“IVS 合成状态更改（会话结束）”事件后回放或处理录制的流。有关详细信息，请参阅 EventBridge 与 IVS 实时流媒体配合使用。

以下是 IVS 实时会话录制的示例目录结构和内容：

```
MNALAcH9j2EJ/s2AdaGUbvQgp/2pBRKrNgX1ff/composite
   events
      recording-started.json
      recording-ended.json
   media
      hls
```

events 文件夹包含与录制事件相对应的元数据文件。记录开始、成功结束或以失败结束时会生成 JSON 元数据文件：

- events/recording-started.json
- events/recording-ended.json
- events/recording-failed.json

给定 events 文件夹将包含 recording-started.json 和 recording-ended.json 或 recording-failed.json 之一。

其中包含与录制会话及其输出格式相关的元数据。JSON 详细信息如下。

media 文件夹包含支持的媒体内容。hls 子文件夹包含合成会话期间生成的所有媒体和清单文件，并且可使用 IVS 播放器播放。HLS 清单位于 multivariant.m3u8 文件夹中。

的存储桶政策 StorageConfiguration

创建 StorageConfiguration 对象后，IVS 将有权将内容写入指定的 S3 存储桶。此访问权限通过修改 S3 存储桶的策略来授予。如果通过移除 IVS 的访问权限来更改存储桶的策略，则正在进行的录制和新录制都将失败。

下例显示了允许 IVS 写入到 S3 存储桶的 S3 存储桶策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CompositeWrite-y1d212y",
            "Effect": "Allow",
            "Principal": {
                "Service": "ivs-composite.ap-northeast-1.amazonaws.com"
            },
            "Action": [
```

```
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::my-s3-bucket/*",
            "Condition": {
                "StringEquals": {
                    "s3:x-amz-acl": "bucket-owner-full-control"
                },
                "Bool": {
                    "aws:SecureTransport": "true"
                }
            }
        }
    ]
}
```

JSON 元数据文件

此元数据采用 JSON 格式，它包含以下信息：

字段	类型	必需	描述
stage_arn	字符串	是	用作合成来源的舞台的 ARN。
media	对象	是	包含可用于此录制的媒体内容的枚举对象的对象。有效值："hls"。
hls	对象	是	描述 Apple HLS 格式输出的枚举字段。
duration_ms	integer	条件	所录制 HLS 内容的时长（以毫秒为单位）。此选项仅在 recording_status 为 "RECORDING_ENDED" 或 "RECORDING_ENDED_WITH_FAILURE" 时可用。如果在完成任何录制之前发生故障，则该值为 0。
path	字符串	是	存储 HLS 内容的 S3 前缀的相对路径。

字段	类型	必需	描述
playlist	字符串	是	HLS 主播放列表文件的名称。
renditions	对象	是	元数据对象的呈现数组（HLS 变体）。始终至少有一个呈现。
path	字符串	是	为此呈现存储 HLS 内容的 S3 前缀的相对路径。
playlist	字符串	是	此呈现的媒体播放列表文件的名称。
resolution_height	int	条件	编码视频的像素分辨率高度。仅当呈现包含视频轨道时，此选项才可用。
resolution_width	int	条件	编码视频的像素分辨率宽度。仅当呈现包含视频轨道时，此选项才可用。
recording_ended_at	字符串	条件	录制结束时的 RFC 3339 UTC 时间戳。此选项仅在 recording_status 为 "RECORDING_ENDED" 或 "RECORDING_ENDED_WITH_FAILURE" 时可用。 recording_started_at 和 recording_ended_at 是这些事件生成时的时间戳，可能与 HLS 视频片段的时间戳不完全一致。要准确确定录制的持续时间，请使用 duration_ms 字段。

字段	类型	必需	描述
recording_started_at	字符串	条件	录制开始时的 RFC 3339 UTC 时间戳。这在 recording_status 为 RECORDING_START_FAILED 时不可用。 请参阅上面有关 recording_ended_at 的注释。
recording_status	字符串	是	录制的状态。有效值："RECORDING_STARTED"、"RECORDING_ENDED"、"RECORDING_START_FAILED"、"RECORDING_ENDED_WITH_FAILURE"。
recording_status_message	字符串	条件	状态的描述性信息。此选项仅在 recording_status 为 "RECORDING_ENDED" 或 "RECORDING_ENDED_WITH_FAILURE" 时可用。
version	字符串	是	元数据架构的版本。

示例：recording-started.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-11-01T06:01:36Z",
  "recording_status": "RECORDING_STARTED",
  "media": {
    "hls": {
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
```

```
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

示例：recording-ended.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-10-27T17:00:44Z",
  "recording_ended_at": "2023-10-27T17:08:24Z",
  "recording_status": "RECORDING_ENDED",
  "media": {
    "hls": {
      "duration_ms": 460315,
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

示例：recording-failed.json

```
{
  "version": "v1",
  "stage_arn": "arn:aws:ivs:ap-northeast-1:123456789012:stage/aAbBcCdDeE12",
  "recording_started_at": "2023-10-27T17:00:44Z",
  "recording_ended_at": "2023-10-27T17:08:24Z",
  "recording_status": "RECORDING_ENDED_WITH_FAILURE",
```

```
  "media": {
    "hls": {
      "duration_ms": 460315,
      "path": "media/hls",
      "playlist": "multivariant.m3u8",
      "renditions": [
        {
          "path": "720p30-abcdeABCDE12",
          "playlist": "playlist.m3u8",
          "resolution_width": 1280,
          "resolution_height": 720
        }
      ]
    }
  }
}
```

播放私有存储桶中的录制内容

默认情况下，录制的内容为私有；因此，使用直接 S3 URL 无法访问这些对象。如果您尝试使用 IVS 播放器或其他播放器打开 HLS 多元播放列表（m3u8 文件）进行播放，您将收到错误信息（例如，“您无权访问请求的资源”）。相反，您可以使用 Amazon CloudFront CDN（内容分发网络）播放这些文件。

CloudFront 可以将发行版配置为提供来自私有存储桶的内容。通常，这比拥有可公开访问的存储桶更可取，在这种存储桶中，读取可以绕过提供的 CloudFront控件。您可以通过创建源访问控制 (OAC) 将分配设置为从私有存储桶提供服务，该访问控制是一个对私有源存储桶具有读取权限的特殊 CloudFront 用户。您可以在创建发行版之后通过 CloudFront 控制台或 API 创建 OAC。请参阅《Amazon CloudFront 开发者指南》中的创建新的源站访问控制。

在启用 CORS 的情况下 CloudFront 使用设置播放

此示例介绍开发者如何在启用 CORS 的情况下设置 CloudFront 发行版，从而允许从任何域播放其录音。这在开发阶段特别有用，但是您可以修改下面的示例以满足您的生产需要。

步骤 1：创建 S3 存储桶

创建用于存储录制内容的 S3 存储桶。请注意，存储桶需要处于您用于 IVS 工作流程的同一区域。

向存储桶添加宽松 CORS 策略：

1. 在 AWS 控制台中，转到 S3 存储桶权限选项卡。
2. 复制下面的 CORS 策略并将其粘贴到跨源资源共享（CORS）下。这将在 S3 存储桶上启用 CORS 访问。

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "PUT",
            "POST",
            "DELETE",
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "x-amz-server-side-encryption",
            "x-amz-request-id",
            "x-amz-id-2"
        ]
    }
]
```

步骤 2：创建分 CloudFront 配

请参阅《CloudFront 开发者指南》中的创建 CloudFront 发行版。

使用 AWS 控制台，输入以下内容：

对于此字段.....	选择此.....
源域	在上一步中创建的 S3 桶
源访问	源访问控制设置（推荐），使用默认参数
默认缓存行为：查看器协议策略	将 HTTP 重定向到 HTTPS

对于此字段……	选择此……
默认缓存行为：允许的 HTTP 方法	GET、HEAD 和 OPTIONS
默认缓存行为：缓存密钥和源请求	CachingDisabled 政策
默认缓存行为：源请求策略	CORS-S3Origin
默认缓存行为：响应标头策略	SimpleCORS
Web 应用程序防火墙	启用安全保护

然后保存分 CloudFront 发。

步骤 3：设置 S3 存储桶策略

1. 删除 StorageConfiguration 您为 S3 存储桶设置的所有内容。这将删除在为该存储桶创建策略时自动添加的任何存储桶策略。
2. 转到您的 CloudFront 分发，确保所有分发字段都处于上一步中定义的状态，然后复制存储桶策略（使用复制策略按钮）。
3. 转到您的 S3 桶。在权限选项卡上，选择编辑存储桶策略，然后粘贴您在上一步中复制的存储桶策略。完成此步骤后，存储桶策略应仅使用该 CloudFront 策略。
4. 创建 StorageConfiguration，指定 S3 存储桶。

创建后，您将在 StorageConfiguration S3 存储桶策略中看到两个项目，一个CloudFront 允许读取内容，另一个允许 IVS 写入内容。示例：带有 IVS 访问权限的 S3 存储桶策略中显示了具有 CloudFront 和 IVS 访问权限的最终存储桶策略的 CloudFront 示例。

步骤 4：播放录制内容

成功设置 CloudFront 分发并更新存储桶策略后，您应该能够使用 IVS 播放器播放录音：

1. 成功启动合成，并确保录制内容存储在 S3 存储桶中。
2. 按照本示例中的步骤 1 至步骤 3 进行操作后，视频文件应可通过 CloudFront URL 进行使用。您的 CloudFront URL 是亚马逊 CloudFront 控制台详情选项卡上的分发域名。它应该如下所示：

   ```
   a1b23cdef4ghij.cloudfront.net
   ```

3. 要通过 CloudFront 发行版播放录制的视频，请在 s3 存储桶下找到`multivariant.m3u8`文件的对象密钥。它应该如下所示：

   ```
   FDew6Szq5iTt/9NIpWJHj0wPT/fjFKbylPb3k4/composite/media/hls/
   multivariant.m3u8
   ```

4. 将对象密钥附加到 CloudFront 网址的末尾。您的最终 URL 如下所示：

   ```
   https://a1b23cdef4ghij.cloudfront.net/FDew6Szq5iTt/9NIpWJHj0wPT/
   fjFKbylPb3k4/composite/media/hls/multivariant.m3u8
   ```

5. 现在，您可以将最终 URL 添加到 IVS 播放器的源属性中，以观看完整录制内容。要观看录制的视频，可以使用 IVS 播放器开发工具包：Web 指南的入门中的演示。

示例：带有 CloudFront IVS 访问权限的 S3 存储桶策略

以下代码段说明了一个 S3 存储桶策略，该策略允许 CloudFront 将内容读取到私有存储桶，IVS 允许将内容写入存储桶。注意：请勿将以下代码段复制并粘贴到自己的存储桶中。您的政策应包含与您的 CloudFront 分发相关的ID，以及 StorageConfiguration.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CompositeWrite-7eiKaIGkC9DO",
      "Effect": "Allow",
      "Principal": {
        "Service": "ivs-composite.ap-northeast-1.amazonaws.com"
      },
      "Action": [
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Resource": "arn:aws:s3:::eicheane-test-1026-2-ivs-recordings/*",
      "Condition": {
        "StringEquals": {
          "s3:x-amz-acl": "bucket-owner-full-control"
        },
        "Bool": {
          "aws:SecureTransport": "true"
        }
      }
```

```
    },
    {
      "Sid": "AllowCloudFrontServicePrincipal",
      "Effect": "Allow",
      "Principal": {
        "Service": "cloudfront.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::eicheane-test-1026-2-ivs-recordings/*",
      "Condition": {
        "StringEquals": {
          "AWS:SourceArn": "arn:aws:cloudfront::844311324168:distribution/
E1NG4YMW5MN25A"
        }
      }
    }
  ]
}
```

排查问题

- 组合不会写入 S3 存储桶 — 确保 S3 存储桶和StorageConfiguration 对象是在同一区域创建的。还要通过查看您的存储桶策略来确保 IVS 可以访问存储桶；有关信息，请参阅存储桶策略。StorageConfiguration
- 表演时我找不到构图 ListCompositions—— 构图是短暂的资源。一旦它们转换到最终状态，将会在几分钟后自动删除。
- 我的合成自动停止 — 如果舞台上没有发布者超过 60 秒，合成将自动停止。

已知问题

由合成录制创建的媒体播放列表在合成正在进行时带有 #EXT-X-PLAYLIST-TYPE:EVENT 标签。合成完成后，标签更新为 #EXT-X-PLAYLIST-TYPE:VOD。为了获得流畅的播放体验，我们建议您仅在成功完成合成后使用此播放列表。

OBS 和 WHIP Support（实时直播）

本文档介绍如何使用与 Whip 兼容的编码器（如 OBS）发布到 IVS 实时流媒体。WHIP（Webrtc-HTTP 摄取协议）是一份旨在标准化 WebRTC 摄取的 IETF 草案。

WHIP 支持与 OBS 等软件兼容，为桌面发布提供了替代方案（IVS 广播 SDK）。熟悉 OBS 的更精密的流传输工具可能会更喜欢这种选择，因为其具有高级制作功能，例如场景过渡、音频混音和图形叠加。这为开发者提供了一个多功能的选择：使用IVS网络广播SDK直接进行浏览器发布，或者允许主播在桌面上使用OBS来获得更强大的工具。

此外，在使用 IVS 广播 SDK 不可行或不可取的情况下，WHIP 是有益的。例如，在涉及硬件编码器的设置中，可能无法选择 IVS 广播 SDK。但是，如果编码器支持 WHIP，您仍然可以直接从编码器发布到 IVS。

OBS 指南

从版本 30 开始，OBS 支持 WHIP。首先，请下载 OBS v30 或更高版本：https://obsproject.com/。

要通过 WHIP 使用 OBS 发布到 IVS 阶段，请按照以下步骤操作：

1. 生成具有发布功能的参与者令牌。用WHIP的术语来说，参与者代币是一种不记名代币。默认情况下，参与者令牌将在 12 小时后过期，但您可以将有效期延长至 14 天。
2. 单击设置。在 “设置” 面板的 “直播” 部分，从 “服务” 下拉列表中选择 WHIP。
3. 对于服务器，请输入 https://global.whip.live-video.net/。
4. 对于不记名令牌，请输入您在步骤 2 中生成的参与者令牌。
5. 像往常一样配置视频设置，但有一些限制：
   a. IVS 实时流媒体支持高达 720p 的输入，速度为 8.5 Mbps。如果您超过上述任一限制，您的直播将断开连接。
   b. 我们建议在 “输出” 面板中将 “关键帧间隔” 设置为 1s 或 2s。低关键帧间隔允许观看者更快地开始播放视频。我们还建议将 CPU 使用率预设设置为超快，将 Tune 设置为零延迟，以实现最低延迟。
   c. 由于 OBS 不支持联播，因此我们建议将比特率保持在 2.5 Mbps 以下。这使使用低带宽连接的观众可以观看。
6. 按 “开始流式传输”。

服务限额（实时直播功能）

以下是 Amazon Interactive Video Service（IVS）实时端点、资源和其他操作的服务限额和限制。服务限额（也称为限制）是您的 AWS 账户使用的服务资源或操作的最大数量。也就是说，除非表中另有说明，否则这些限制针对每个 AWS 账户。另请参阅 AWS 服务限额。

要通过编程方式连接到 AWS 服务，您需要使用端点。另请参阅 AWS 服务端点。

所有限额都是按区域强制执行的。

服务限额增加

对于可调配限额，您可以通过 AWS 控制台请求提高速率。也可以使用控制台查看有关服务限额的信息。

API 调用速率限额不可调整。

API 调用速率限额

端点类型	端点	默认
合成	GetComposition	5 TPS
合成	ListCompositions	5 TPS
合成	StartComposition	5 TPS
合成	StopComposition	5 TPS
MediaEncoder	CreateEncoderConfiguration	5 TPS
MediaEncoder	DeleteEncoderConfiguration	5 TPS
MediaEncoder	GetEncoderConfiguration	5 TPS
MediaEncoder	ListEncoderConfigurations	5 TPS
舞台	CreateParticipantToken	50 TPS
舞台	CreateStage	5 TPS

端点类型	端点	默认
舞台	DeleteStage	5 TPS
舞台	DisconnectParticipant	5 TPS
舞台	GetParticipant	5 TPS
舞台	GetStage	5 TPS
舞台	GetStageSession	5 TPS
舞台	ListStages	5 TPS
舞台	UpdateStage	5 TPS
舞台	ListParticipants	5 TPS
舞台	ListParticipantEvents	5 TPS
舞台	ListStageSessions	5 TPS
StorageConfiguration	CreateStorageConfiguration	5 TPS
StorageConfiguration	DeleteStorageConfiguration	5 TPS
StorageConfiguration	GetStorageConfiguration	5 TPS
StorageConfiguration	ListStorageConfigurations	5 TPS
标签	ListTagsForResource	10 TPS
标签	TagResource	10 TPS
标签	UntagResource	10 TPS

其他限额

资源或功能	默认	是否可调整	说明
EncoderConfigurations	20	是	每个账户的编码器配置资源的最大数量。
合成目标	2	不可以	合成资源中的目标对象的最大数量。
合成：最大持续时间	24	不可以	合成可以存在的最大时间，以小时为单位。
合成	5	可以	每个账户的最大并发合成资源。
参与者发布或订阅时长	24	不可以	参与者可以发布或保持舞台订阅的最长时长，以小时为单位。
参与者发布的视频分辨率	720p	不可以	参与者发布的视频最高分辨率。
参与者下载的视频比特率	8.5Mbps	不可以	参与者订阅的所有视频的最大聚合下载比特率。
舞台参与者（发布者）	12	不可以	可以同时发布到舞台的最大参与者数量。
舞台参与者（订阅用户）	10000	可以	可以同时订阅到舞台的最大参与者数量。
阶段	100	是	每个 AWS 区域的最大舞台数量。

实时流式传输优化

为了确保您的用户在使用 IVS 实时流式传输时获得最佳的视频直播和观看体验，您可以使用我们目前提供的所有功能，通过多种方式对部分体验进行改进或优化。

简介

在优化用户的体验质量时，务必考虑用户想要的观看体验，这种体验可能会根据其正在观看的内容和网络状况而变化。

在本指南中，我们重点关注流的发布者或流的订阅用户，并考虑这些用户所需的操作和体验。

自适应流式传输：通过联播分层编码

此功能仅在下列客户端版本中受支持：

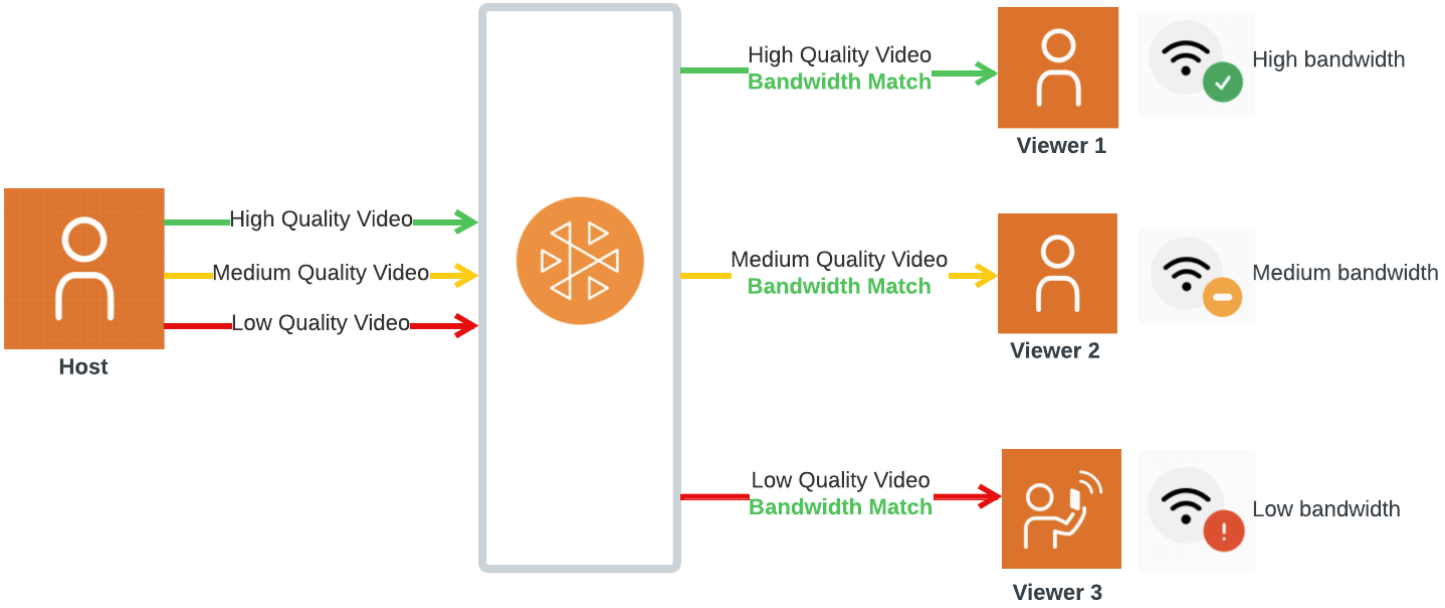
- iOS 和 Android 1.12.0+
- Web 1.5.1+

你必须发送电子邮件至 amazon-ivs-simulcast@amazon .com 才能为你的账户选择使用此功能。除非您选择加入，否则通过 SDK 配置启用联播将不会产生任何效果。

当您选择使用此功能后，使用 IVS 实时广播 SDK 时，发布者会对多层视频进行编码，而订阅用户会自动适应或根据网络状况更改为最佳质量。我们将之称为通过联播分层编码。

Android 和 iOS 以及 Chrome 桌面浏览器（适用于 Windows 和 macOS）均支持通过联播分层编码。但其他浏览器不支持分层编码。

在下图中，主机发送了三种质量（高、中和低）的视频。IVS 根据可用带宽向每位观众发送最高质量的视频；每位观众可以获得最佳视频体验。如果观众 1 的网络连接从良好变为不良，IVS 会自动开始向观众 1 发送较低质量的视频，因而观众 1 可以继续以其所能获得的最佳质量观赏流。



默认分层、质量和帧率

为移动平台和 Web 用户提供默认质量和分层，如下所示：

移动平台（Android、iOS）	Web（Chrome）
高级层（或自定义）： • 最大比特率：900,000 bps • 帧率：15 fps • 分辨率：360x640	高级层（或自定义）： • 最大比特率：1,700,000 bps • 帧率：30 fps • 分辨率：1280x720
中间层：无（不需要，因为移动平台上高级层和低级层的比特率之间的差异很小）	中间层： • 最大比特率：700,000 bps • 帧率：20 fps • 分辨率：640x360
低级层： • 最大比特率：150,000 bps • 帧率：15 fps • 分辨率：180x320	低级层： • 最大比特率：200,000 bps • 帧率：15 fps • 分辨率：320x180

通过联播配置分层编码

要在联播中使用分层编码，您必须选择使用该功能，并在客户端上启用该功能。如果启用它，您将看到传输的总体比特率有所提高，从而减少了视频冻结。

Android

```
// Opt-out of Simulcast
StageVideoConfiguration config = new StageVideoConfiguration();
config.simulcast.setEnabled(true);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
// Opt-out of Simulcast
let config = IVSLocalStageStreamVideoConfiguration()
config.simulcast.enabled = true

let cameraStream = IVSLocalStageStream(device: camera, configuration: config)

// Other Stage implementation code
```

Web

```
// Opt-out of Simulcast
let cameraStream = new LocalStageStream(cameraDevice, {
   simulcast: { enabled: true }
})

// Other Stage implementation code
```

流式传输配置

本节介绍了可对视频和音频流进行的其他配置。

更改视频流比特率

要更改视频流的比特率，请使用以下配置示例。

Android

```
StageVideoConfiguration config = new StageVideoConfiguration();

// Update Max Bitrate to 1.5mbps
config.setMaxBitrate(1500000);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamVideoConfiguration();

// Update Max Bitrate to 1.5mbps
try! config.setMaxBitrate(1500000);

let cameraStream = IVSLocalStageStream(device: camera, configuration: config);

// Other Stage implementation code
```

Web

```
// Note: On web it is also recommended to configure the framerate of your device from
 userMedia
const camera = await navigator.mediaDevices.getUserMedia({
   video: {
      bitrate: {
         ideal: 1500,
         max: 1500,
      },
   },
});

let cameraStream = new LocalStageStream(camera.getVideoTracks()[0], {
   // Update Max Bitrate to 1.5mbps or 1500kbps
   maxBitrate: 1500
})

// Other Stage implementation code
```

更改视频流帧率

要更改视频流的帧率，请使用以下配置示例。

Android

```
StageVideoConfiguration config = new StageVideoConfiguration();

// Update target framerate to 10fps
config.targetFramerate(10);

ImageLocalStageStream cameraStream = new ImageLocalStageStream(frontCamera, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamVideoConfiguration();

// Update target framerate to 10fps
try! config.targetFramerate(10);

let cameraStream = IVSLocalStageStream(device: camera, configuration: config);

// Other Stage implementation code
```

Web

```
// Note: On web it is also recommended to configure the framerate of your device from
 userMedia
const camera = await navigator.mediaDevices.getUserMedia({
   video: {
      frameRate: {
         ideal: 10,
         max: 10,
      },
   },
});

let cameraStream = new LocalStageStream(camera.getVideoTracks()[0], {
   // Update Max Framerate to 10fps
   maxFramerate: 10
```

```
})
// Other Stage implementation code
```

优化音频比特率和立体声支持

要更改音频流的比特率和立体声设置，请使用以下配置示例。

Web

```
// Note: Disable autoGainControl, echoCancellation, and noiseSuppression when enabling
 stereo.
const camera = await navigator.mediaDevices.getUserMedia({
   audio: {
      autoGainControl: false,
      echoCancellation: false,
      noiseSuppression: false
   },
});

let audioStream = new LocalStageStream(camera.getAudioTracks()[0], {
   // Optional: Update Max Audio Bitrate to 96Kbps. Default is 64Kbps
   maxAudioBitrateKbps: 96,

   // Signal stereo support. Note requires dual channel input source.
   stereo: true
})

// Other Stage implementation code
```

Android

```
StageAudioConfiguration config = new StageAudioConfiguration();

// Update Max Bitrate to 96Kbps. Default is 64Kbps.
config.setMaxBitrate(96000);

AudioLocalStageStream microphoneStream = new AudioLocalStageStream(microphone, config);

// Other Stage implementation code
```

iOS

```
let config = IVSLocalStageStreamConfiguration();

// Update Max Bitrate to 96Kbps. Default is 64Kbps.
try! config.audio.setMaxBitrate(96000);

let microphoneStream = IVSLocalStageStream(device: microphone, config: config);

// Other Stage implementation code
```

推荐优化

场景	建议
包含文字或移动缓慢内容的流，例如演示文稿或幻灯片	使用通过联播分层编码或配置帧速率较低的流。
包含动作或大量移动的流	使用通过联播分层编码。
包含对话或少量移动的流	通过联播分层编码或选择纯音频（请参阅《实时流式广播 SDK 指南：Web、Android 和 iOS》中的“订阅参与者”）。
数据有限的用户流式传输	使用通过联播分层编码，或者，如果您想降低所有人的数据使用量，可以配置较低的帧速率并手动降低比特率。

资源和支持（实时流式传输）

资源

https://ivs.rocks/ 是一个专用于浏览已发布内容（演示、代码示例、博客文章）、估算成本并通过现场演示体验 Amazon IVS 的网站。

演示

适用于 iOS 和 Android 的 IVS 实时流式传输演示向开发人员展示了如何使用 Amazon IVS 构建引人入胜的、可由社交用户生成的实时内容应用程序。该应用程序具有用户生成的实时流式传输的可滚动提要。用户可以创建视频流和纯音频聊天室。视频流嘉宾可在嘉宾席位或对战（VS）模式下加入互动。有关如何部署所需后端和构建应用程序的说明，可在以下 GitHub 存储库中找到：

- iOS：https://github.com/aws-samples/amazon-ivs-real-time-for-ios-demo/
- Android： https://github.com/aws-samples/amazon-ivs-real-time-for-android-demo/

- 后端：https://github.com/aws-samples/amazon-ivs-real-time-serverless-demo/

支持

AWS Support Center 提供了一系列计划，您可以通过这些计划获取各种工具和专业知识来为 AWS 解决方案提供支持。所有支持计划均提供全天候客户服务。要获取可规划、部署和改善 AWS 环境的技术支持服务和更多资源，请选择一项最适合 AWS 使用案例的支持计划。

AWS Premium Support 是一对一的快速响应支持通道，可帮助您在 AWS 中构建和运行应用程序。

AWS re:Post 是一个基于社区的问答网站，供开发人员讨论与 Amazon IVS 相关的技术问题。

联系我们包含关于您的账单或账户的非技术性查询的链接。如有技术问题，请使用上述开发论坛或支持连接。

术语表

另请参阅 AWS 术语表。在下表中，LL 代表 IVS 低延迟直播；RT 代表 IVS 实时直播。

租期	描述	LL	RT	聊天
AAC	高级音频编码。AAC 是有损数字音频压缩的音频编码标准。AAC 旨在成为 MP3 格式的继任者，在相同的比特率下，其音质通常比 MP3 更高。ISO 和 IEC 已将 ACC 标准化，作为 MPEG-2 和 MPEG-4 规范的一部分。	✓	✓	
自适应比特率流	自适应比特率（ABR）流允许 IVS 播放器在连接质量下降时切换到较低的比特率，并在连接质量提高时切换回较高的比特率。	✓		
自适应流	请参阅通过联播分层编码。		✓	
管理用户	对 AWS 账户中可用的资源和服务具有管理权限的 AWS 用户。请参阅《AWS 设置用户指南》中的术语。	✓	✓	✓
ARN	Amazon 资源名称，这是 AWS 资源的唯一标识符。具体的 ARN 格式取决于资源类型。有关 IVS 资源使用的 ARN 格式，请参阅《服务授权参考》。	✓	✓	✓
纵横比	描述帧宽度与帧高的比率。例如，16:9 是与全高清或 1080p 分辨率相对应的宽高比。	✓	✓	
音频模式	针对不同类型的移动设备用户及其使用的设备进行优化的预设或自定义音频配置。请参阅 IVS 广播 SDK：移动音频模式（实时直播）。		✓	
AVC、H.264、MPEG-4 第 10 部分	高级视频编码，也称为 H.264 或 MPEG-4 第 10 部分，是有损数字视频压缩的视频压缩标准。	✓	✓	

租期	描述	LL	RT	聊天
背景替换	一种相机滤镜，让实时流创作者能够更改其背景。请参阅 IVS 广播 SDK：第三方相机滤镜（实时直播）中的背景替换。		✓	
比特率	每秒传输或接收的比特数的直播指标。	✓	✓	
广播，广播者	流、直播工具的其他术语。	✓		
缓冲	播放设备在需要播放内容之前无法下载该内容时出现的一种情况。缓冲可以通过多种方式表现出来：内容可能随机停止并开始（也称为卡顿），内容也可能长时间停止（也称为冻结），或者 IVS 播放器可能进入了暂停状态。	✓	✓	
字节范围播放列表	比标准 HLS 播放列表更精细的播放列表。标准 HLS 播放列表由 10 秒的媒体文件组成。对于字节范围播放列表，片段持续时间与为流配置的关键帧间隔相同。 字节范围播放列表仅适用于自动录制到 S3 存储桶的广播。其是在 HLS 播放列表之外创建的。请参阅自动录制到 Amazon S3（低延迟直播）中的字节范围播放列表。	✓		
CBR	恒定比特率，编码器的一种速率控制方法，无论广播期间发生什么情况，都能在视频的整个播放过程中保持一致的比特率。可以填充操作中的间歇以达到所需的比特率，并且可以通过调整编码质量以匹配目标比特率来量化峰值。我们强烈建议使用 CBR 而不是 VBR。	✓	✓	
CDN	内容分发网络，一种地理分布式解决方案，通过让直播视频等内容更靠近用户所在位置来优化内容的交付。	✓		

租期	描述	LL	RT	聊天
频道	存储直播配置的 IVS 资源，包括摄取服务器、流密钥、播放 URL 和录制选项。流传输工具使用与通道关联的流密钥来启动广播。广播期间生成的所有指标和事件都与通道资源相关联。	✓		
通道类型	确定通道允许的分辨率和帧速率。请参阅 IVS Low-Latency Streaming API Reference 中的 Channel Types（通道类型）。	✓		
聊天记录	一个高级选项，可以通过将日志记录配置与某个聊天室相关联来启用。			✓
聊天室	一种 IVS 资源，用于存储聊天会话的配置，包括消息审核处理程序和聊天记录等可选功能。请参阅 Getting Started with IVS Chat 中的 Step 2: Create a Chat Room。			✓
客户端合成	使用主机设备混合舞台参与者的音频和视频流，然后将其作为复合流发送到 IVS 通道。这样可以更好地控制合成的外观，代价是客户端资源的利用率更高，舞台或主机问题影响查看者的风险也更高。 另请参阅服务器端合成。	✓	✓	
CloudFront	Amazon 提供的 CDN 服务。	✓		
CloudTrail	一项 AWS 服务，用于收集、监控、分析和保留来自 AWS 和外部来源的事件和账户活动。请参阅使用 AWS 记录 IVS CloudTrail API 调用。	✓	✓	✓
CloudWatch	一项 AWS 服务，用于监控应用程序、响应性能变化、优化资源使用并提供有关运行状况的见解。您可以使用 CloudWatch 监控 IVS 指标；请参阅监控 IVS 实时流和监控 IVS 低延迟流媒体。	✓	✓	✓
合成	将来自多个来源的音频和视频流合并为单个流的过程。	✓	✓	

租期	描述	LL	RT	聊天
合成管道	合并多个流并对生成的流进行编码所需的一系列处理步骤。	✓	✓	
压缩	使用比原始表示形式更少的比特数对信息进行编码。任何特定的压缩都是无损或有损的。无损压缩通过识别和消除统计冗余来减少比特数。无损压缩不会丢失任何信息。有损压缩通过删除不必要或不太重要的信息来减少比特数。	✓	✓	
控制层面	存储有关 IVS 资源（例如通道、舞台或聊天室）的信息，并提供用于创建和管理这些资源的界面。其具有区域性（基于 AWS 区域）。	✓	✓	✓
CORS	跨源资源共享（CORS），这是一项 AWS 功能，允许在一个域中加载的客户端 Web 应用程序与另一个域中的资源（例如 S3 存储桶）进行交互。可以根据标头、HTTP 方法和源域配置访问权限。请参阅《Amazon Simple Storage Service 用户指南》中的使用跨源资源共享（CORS）–Amazon Simple Storage Service。	✓		
自定义图像源	IVS 广播 SDK 提供的界面，允许应用程序提供自己的图像输入，而不仅限于预设相机。	✓	✓	
数据层面	将数据从摄取传输到出口的基础设施。其根据控制面板中管理的配置运行，不限于 AWS 区域。	✓	✓	✓
编码器、编码	将视频和音频内容转换为适合直播的数字格式的过程。编码可以基于硬件，也可以基于软件。	✓	✓	
事件	IVS 向 AmazonEventBridge监控服务发布的自动通知。事件代表直播资源（例如舞台或合成管道）的状态或运行状况变化。参见使用 Amazon EventBridge 进行 IVS 低延迟流媒体和使用 Amazon 进行 IVS 实时流 EventBridge 式传输。	✓	✓	✓

租期	描述	LL	RT	聊天
FFmpeg	一个免费的开源软件项目，其中包含一套用于处理视频和音频文件以及流的库和程序。FFmpeg 提供了一种跨平台解决方案来录制、转换以及直播音频和视频。	✓		
片段化的流	当广播断开连接，然后在通道录制配置中指定的间隔内重新连接时创建。生成的多个流被视为单个广播，并合并到录制流中。请参阅自动录制到 Amazon S3（低延迟直播）中合并片段化的流。	✓		
帧率	每秒传输或接收的视频帧数的直播指标。	✓	✓	
HLS	HTTP 实时流（HLS），一种基于 HTTP 的自适应比特率流通信协议，用于向查看者传送 IVS 流。	✓		
HLS 播放列表	组成流的媒体片段列表。标准 HLS 播放列表由 10 秒的媒体文件组成。HLS 还支持更精细的字节范围播放列表。	✓		
Host	向舞台发送视频和/或音频的实时事件参与者。		✓	
IAM	Identity and Access Management，这是一项 AWS 服务，允许用户安全地管理身份和访问 AWS 服务和资源，包括 IVS。	✓	✓	✓
提取	用于从主机或广播者接收视频流以进行处理或传送给查看者或其他参与者的 IVS 过程。	✓	✓	
提取服务器	接收视频流并将其传送到转码系统，在该系统中，视频流被转码多路复用或转码为 HLS 以传送给查看者。 摄取服务器是特定的 IVS 组件，用于接收通道流以及摄取协议（RTMP、RTMPS）。有关创建通道的信息，请参阅 IVS 低延迟直播入门。		✓	

租期	描述	LL	RT	聊天
隔行视频	仅传输和显示后续帧的奇数行或偶数行，从而在不消耗额外带宽的情况下实现帧速率的翻倍。出于视频质量方面的考虑，我们不建议使用隔行视频。	✓	✓	
JSON	JavaScript 对象表示法，一种开放标准的文件格式，它使用人类可读的文本来传输由属性值对和数组数据类型或其他可序列化值组成的数据对象。	✓	✓	✓
关键帧、增量帧、关键帧间隔	关键帧（也称为帧内编码或 i 帧）是视频中图像的全帧。后续帧，即增量帧（也称为预测帧或 p 帧），仅包含已更改的信息。关键帧将在一个流中多次出现，具体取决于编码器中定义的关键帧间隔。	✓	✓	
Lambda	一项 AWS 服务，用于运行代码（称为 Lambda 函数），无需预置任何服务器基础设施。Lambda 函数可以响应事件和调用请求运行，也可以根据计划运行。例如，IVS 聊天功能使用 Lambda 函数来启用聊天室的消息审核。	✓	✓	✓
延迟、 glass-to-glass延迟	数据传输中的延迟。IVS 将延迟范围定义如下： • 低延迟：低于 3 秒 • 实时延迟：低于 300 毫秒 G lass-to-glass 延迟是指从摄像机捕捉直播到直播出现在观众屏幕上的延迟。	✓	✓	
通过联播分层编码	支持同时编码并发布具有不同质量级别的多个视频流。请参阅实时直播优化中的自适应直播：通过联播分层编码。		✓	
消息审核处理程序	允许 IVS 聊天功能客户能够在用户聊天消息传送到聊天室之前自动查看/筛选这些消息。将 Lambda 函数与聊天室关联来将其启用。请参阅 Chat Message Review Handler 中的 Creating a Lambda Function。			✓

租期	描述	LL	RT	聊天
混合器	IVS 移动广播 SDK 的一项功能，可接收多个音频和视频源并生成单个输出。其支持对代表源的屏幕视频和音频元素进行管理，例如相机、麦克风、屏幕截图以及应用程序生成的音频和视频。然后可以将输出直播到 IVS。请参阅 IVS 广播 SDK：混合器指南（低延迟直播）中的配置广播会话以进行混合。	✓		
多主机直播	将来自多个主机的流合并为一个流。可通过使用客户端或服务器端合成来实现。 多主机直播支持诸如邀请查看者到舞台进行问答、主机之间的竞赛、视频聊天以及主机当众对话等场景。		✓	
多变体播放列表	可用于广播的所有变体流的索引。	✓		
OAC	Origin Access Control，一种用于限制对 S3 存储桶的访问的机制，因此只能通过 CloudFrontCDN 提供诸如录制流之类的内容。	✓		
OBS	Open Broadcaster Software（OBS），一款用于视频录制和实时直播的免费开源软件。OBS 为桌面发布提供了另一种选择（IVS 广播 SDK）。熟悉 OBS 的更精密的流传输工具可能会更喜欢这种选择，因为其具有高级制作功能，例如场景过渡、音频混音和图形叠加。	✓	✓	
参与者	以主机或查看者身份连接到舞台的实时用户。		✓	
参与者令牌	在实时事件参与者加入舞台时对其进行身份验证。参与者令牌还能控制参与者是否可以向舞台发送视频。		✓	

租期	描述	LL	RT	聊天
播放令牌、播放密钥对	一种授权机制，允许客户限制在私有通道上播放视频。播放令牌由播放密钥对生成。 播放密钥对是用于签名和验证查看者播放授权令牌的公有-私有密钥对。请参阅设置私有通道中的创建或导入播放密钥以及 IVS Low-Latency API Reference 中的播放密钥对端点。	✓		
播放 URL	标识查看者用于开始特定通道播放的地址。此地址可以在全球范围使用。IVS 会自动为每个查看者选择 IVS 全球内容分发网络上的最佳位置，以便向每个查看者传送视频。有关创建通道的信息，请参阅 IVS 低延迟直播入门。	✓		
私有通道	允许客户使用基于播放令牌的授权机制来限制对其流的访问。请参阅设置私有通道中的私有通道的工作流。	✓		
逐行视频	按顺序传输并显示每帧的所有行。我们建议在广播的所有阶段使用渐进式视频。	✓	✓	
配额	AWS 账户使用的 IVS 服务资源或操作的最大数量。也就是说，除非另有说明，否则这些限制针对每个 AWS 账户。所有限额都是按区域强制执行的。请参阅 AWS General Reference Guide中的 Amazon Interactive Video Service endpoints and quotas。	✓	✓	✓

租期	描述	LL	RT	聊天
区域	提供对实际位于特定地理区域的 AWS 服务的访问权限。区域提供容错能力、稳定性和弹性，还可以减少延迟。使用区域，您能够创建保持可用且不受区域中断影响的冗余资源。 大多数 AWS 服务请求都与特定的地理区域相关联。除非您明确使用 AWS 服务提供的复制功能，否则在一个区域中创建的资源在任何其他区域中都不存在。例如，Amazon S3 支持跨区域复制。某些服务（例如 IAM）没有跨区域资源。	✓	✓	✓
解决方案	描述单个视频帧中的像素数，例如，全高清或 1080p 定义了具有 1920x1080 像素的帧。	✓	✓	
根用户	AWS 账户的所有者。根用户对 AWS 账户中的所有 AWS 服务和资源具有完全访问权限。	✓	✓	✓
RTMP、RTMPS	实时消息协议，一种通过网络传输音频、视频和数据的行业标准。RTMPS 是 RTMP 的安全版本，通过传输层安全性协议（TLS/SSL）连接运行。	✓	✓	
S3 存储桶	存储在 Amazon S3 中对象的集合。许多策略（包括访问和复制）都是在存储桶级别定义的，适用于存储桶中的所有对象。例如，IVS 广播作为多个对象存储在 S3 存储桶中。	✓		
SDK	软件开发工具包，供开发人员使用 IVS 构建应用程序的库的集合。	✓	✓	✓
自拍分割	允许使用特定于客户端的解决方案替换实时流中的背景，该解决方案接受相机图像作为输入，并返回一个掩码，该掩码为图像的每个像素提供置信度分数，指示图像是在前景还是背景中。请参阅 IVS 广播 SDK：第三方相机滤镜（实时直播）中的背景替换。		✓	

租期	描述	LL	RT	聊天
Semantic version	Major.Minor.Patch 形式的版本格式。不影响 API 的错误修复会增加补丁版本，向后兼容的 API 添加/更改会增加次要版本，不向后兼容的 API 更改会增加主要版本。	✓	✓	✓
服务器端合成	使用 IVS 服务器混合舞台参与者的音频和视频，然后将此混合视频发送到 IVS 通道，以服务于更多观众或将其存储在 S3 存储桶中。服务器端合成减少了客户端负载，提高了广播的弹性，并可以更有效地使用带宽。 另请参阅客户端合成。		✓	
服务限额	一项 AWS 服务，可帮助您从一个位置管理多个 AWS 服务的限额。除了查找配额值，您也可以从 Service Quotas 控制台请求提高配额。	✓	✓	✓
服务相关角色	与 AWS 服务直接关联的一种独特类型的 IAM 角色。服务相关角色由 IVS 自动创建，并包含该服务代表您调用其他 AWS 服务所需的一切权限，例如访问 S3 存储桶。请参阅 IVS 安全性中的对 IVS 使用服务相关角色。	✓		
舞台	IVS 资源，代表实时事件参与者可以在其中实时交换视频的虚拟空间。请参阅 IVS 实时直播入门中的创建舞台。		✓	
舞台会话	第一个参与者加入舞台时舞台会话开始，最后一个参与者停止向舞台发布几分钟后舞台会话结束。长期存在的舞台在其生命周期内可能有多个会话。		✓	
流	表示从源持续发送到目的地的视频或音频内容的数据。	✓	✓	
流密钥	创建通道时由 IVS 分配的标识符；用于授权直播到该通道。将流密钥视为秘密，因为任何拥有它的人都可以直播到通道。请参阅 IVS 低延迟直播入门。	✓		

租期	描述	LL	RT	聊天
流匮乏	向 IVS 的直播延迟或停止。当 IVS 未收到编码设备宣传的其将在特定时间范围内发送的预期比特量时，就会发生这种情况。出现流匮乏会导致流匮乏事件。 从查看者的角度来看，流匮乏可能表现为视频延迟、缓冲或冻结。流匮乏的持续时间可能较为短暂（小于 5 秒），也可能较长（几分钟），取决于导致流匮乏的具体情况。请参阅问题排查常见问题中的什么是流匮乏。	✓	✓	
流传输工具	向 IVS 发送视频或音频流的个人或设备。	✓	✓	
订阅者	接收主机视频和/或音频的实时事件参与者。请参阅什么是 IVS 实时直播。		✓	
标签	分配给 AWS 资源的元数据标签。标签可帮助您标识和组织 AWS 资源。在 IVS 文档登录页面上，请参阅任何 IVS API 文档中的“标记”（适用于实时直播、低延迟直播或聊天）。	✓	✓	✓
第三方相机滤镜	可以与 IVS 广播 SDK 集成的软件组件，允许应用程序在将图像作为自定义图像源提供给广播 SDK 之前对其进行处理。第三方相机滤镜可以处理来自相机的图像、应用滤镜效果等。	✓	✓	
缩略图	从流中拍摄的缩小尺寸的图像。默认情况下，每 60 秒生成一次缩略图，但可以配置更短的时间间隔。缩略图分辨率取决于通道类型。请参阅自动录制到 Amazon S3（低延迟直播）中的录制内容。	✓		

租期	描述	LL	RT	聊天
定时元数据	与流中特定时间戳相关联的元数据。可以使用 IVS API 以编程方式进行添加，并与特定帧相关联。这样可以确保所有查看者在与视频流相关的同一时间点上接收元数据。 定时元数据可用于触发客户端上的操作，例如在体育赛事期间更新球队统计数据。请参阅将元数据嵌入视频流中。	✓		
转码	将视频和音频从一种格式转换为另一种格式。传入流可以多个比特率和分辨率将其转码为不同格式，以支持一系列播放设备和网络条件。	✓	✓	
转码多路复用	将摄取的流简单地重新打包到 IVS，无需重新编码视频流。“Transmux”是转码多路复用的缩写，这是一个改变音频和/或视频文件格式的过程，同时保留部分或全部原始流。转码多路复用转换为不同的容器格式，而不会更改文件内容。与转码不同。	✓	✓	
变体流	多个不同的质量级别的同一广播的一组编码。每个变体流都编码为单独的 HLS 播放列表。可用变体流的索引称为多变体播放列表。 IVS 播放器从 IVS 接收到多变体播放列表后，可以在播放期间在变体流之间进行选择，随着网络条件的变化，可以无缝地来回转换。	✓		
VBR	可变比特率，编码器的一种速率控制方法，使用动态比特率，根据所需的详情级别在整个播放过程中发生变化。出于视频质量方面的考虑，我们强烈建议不要使用 VBR；改用 CBR。	✓	✓	

租期	描述	LL	RT	聊天
查看	会主动下载或播放视频的独特观看会话。观看次数是并发观看次数限额的基础。 视图会在查看会话开始播放视频时开始。视图会在查看会话停止视频播放时结束。播放是收视率的唯一指标；不考虑音频级别、浏览器选项卡焦点和视频质量等互动启发式算法。在计算观看次数时，IVS 不会考虑个别查看者的合法性，也不会尝试重复计算本地化的收视率，例如单台计算机上的多个视频播放器。请参阅服务限额（低延迟直播）中的其他限额。	✓		
查看者	从接收 IVS 流的人。	✓		
WebRTC	Web 实时通信，一个为 Web 浏览器和移动应用程序提供实时通信的开源项目。。它允许直接通信，无需安装插件或下载本机应用程序，从而允许在网页内部进行音频和视频 peer-to-peer 通信。 WebRTC 背后的技术是作为开放网络标准实现的，可以在所有主流浏览器中作为 JavaScript 常规API使用，也可以作为原生客户端（例如Android和iOS）的库使用。	✓	✓	

租期	描述	LL	RT	聊天
鞭子	Webrtc-HTTP 摄取协议，一种基于 HTTP 的协议，允许将基于 WebRTC 的内容摄取到流媒体服务和/ 或 CD N 中。WHIP 是一份 IETF 草案，旨在对 WebRTC 摄取进行标准化。 WHIP 支持与 OBS 等软件兼容，为桌面发布提供了替代方案（IVS 广播 SDK）。熟悉 OBS 的更老练的主播可能会更喜欢它，因为它具有高级制作功能，例如场景过渡、音频混音和叠加图形 在使用 IVS 广播 SDK 不可行或不可取的情况下，WHIP 也很有用。例如，在涉及硬件编码器的设置中，可能无法选择 IVS 广播 SDK。但是，如果编码器支持 WHIP，您仍然可以直接从编码器发布到 IVS。 参见 OBS 和 WHIP Support。		✓	
WSS	WebSocket 安全，一种通过加密 TLS 连接建立 WebSockets 的协议。用于连接到 IVS 聊天端点。请参阅 Getting Started with IVS Chat中的 Step 4: Send and Receive Your First Message。			✓

文档历史记录（实时直播功能）

Real-Time Streaming User Guide 更改

变更	说明	日期
OBS 和 WHIP Support	添加了一个新页面。本文档介绍如何使用与 Whip 兼容的编码器（如 OBS）发布到 IVS 实时流媒体。WHIP（Webrtc-HTTP 摄取协议）是一份旨在标准化 WebRTC 摄取的 IETF 草案。	2024年2月6日
广播 SDK：安卓 1.14.1、iOS 1.14.1、Web 1.8.0	在 real-time-streaming 广播 SDK 指南中更新了新版本的版本号和工件链接：Android、iOS 和 Web。在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。另请参阅此发行版的 Amazon IVS 发布说明。 在 Android 指南中，我们添加了一个新的已知问题（视频大小小于 176x176）。 在 Web 指南中，我们添加了一个新的已知问题。解决方法是在调用`getUserMedia` 或`getDisplayMedia` 时将视频分辨率限制为 720p。 在“实时直播优化”中，我们更新了使用 Simulcast 配置分层	2024年2月1日

	编码；现在默认情况下，此功能处于禁用状态。	
广播 SDK：安卓 1.13.4、iOS 1.13.4、Web 1.7.0	在 real-time-streaming 广播 SDK 指南中更新了新版本的版本号和工件链接：Android、iOS 和 Web。在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。另请参阅此发行版的 Amazon IVS 发布说明。	2024 年 1 月 3 日
IVS 术语表	扩展了术语表，涵盖 IVS 实时、低延迟和聊天术语。	2023 年 12 月 20 日
Stage Health：新 CloudWatch 指标	将 PacketLoss（阶段）指标重命名为 DownloadPacketLoss（Stage），并发布了 IVS 实时流媒体的其他 CloudWatch 指标： • DownloadPacketLoss（舞台、参与者） • DroppedFrames（舞台、参与者） • SubscribeBitrate（舞台、参与者、MediaType） 请参阅 监控 IVS 实时直播功能。	2023 年 12 月 7 日

IAM 托管式策略	添加了两个托管策略，即 IVS ReadOnlyAccess 和 IV FullAccess S。请参阅： • 安全性页面上关于 Amazon IVS 的托管式策略的新增部分。 • 对IVS 低延迟直播入门中步骤 3：设置 IAM 权限的更改。	2023 年 12 月 5 日
广播 SDK：Android 1.13.2、iOS 1.13.2	在 real-time-streaming 广播 SDK 指南：Andro id 和 iOS 中，更新了新版本的版本号和工件链接。 在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 12 月 4 日
广播 SDK：Android 1.13.1	在 real-time-streaming 广播 SDK 指南：Android 中，更新了新版本的版本号和工件链接。 在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 11 月 21 日
服务限额	将“参与者发布分辨率”从 1080p 更改为 720p。	2023 年 11 月 18 日

广播 SDK：Android 1.13.0、iOS 1.13.0	在 real-time-streaming 广播 SDK 指南：Andro id 和 iOS 中，更新了新版本的版本号和工件链接。 在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。 另请参阅此发行版的 Amazon IVS 发布说明。 我们还对流式传输优化进行了各种更新。除其他外，“自适应流式传输：通过联播分层编码”功能现在需要明确的选择，并且仅支持最新版本的 SDK。	2023 年 11 月 17 日
合成录制	进行了以下更改： • 为这项新功能增加了合成录制页面。 • 更新了 IVS 实时流式传输入门，向“设置 IAM 权限”的策略中添加了 S3 端点。 • 通过新端点的调用速率限额更新 IVS 服务限额。	2023 年 11 月 16 日

服务器端合成（SSC）	IVS 服务器端合成让客户端能够将 IVS 舞台的合成和广播卸载到 IVS 托管的服务。SSC 以及向频道的 RTMP 广播通过舞台主区域的 IVS 控制面板端点调用。请参阅： • 入门 — 我们向“设置 IAM 权限”中的策略添加了 SSC 端点。 • 将 Amazon EventBridge 与 IVS 配合使用 — 我们添加了新的指标。 • 服务器端合成 — 这份新文档包括概述和设置说明。 • 服务限额 – 我们增加了新的调用速率限制和其他限额。 另请参阅： • 下面的 IVS Real-Time Streaming API Reference 更改中列出的更改。 • 文档历史记录（低延迟直播功能）中列出的更改。	2023 年 11 月 16 日
IVS 广播 SDK	在广播 SDK 概述中，我们更新了“平台要求” > “原生平台”，以明确支持哪些 SDK 版本，并增加了“移动浏览器（iOS 和 Android）”。 在广播 Web 指南中，我们增加了“移动 Web 限制”。	2023 年 11 月 9 日

IVS 广播 SDK	我们在第三方相机滤镜中增加了一个新页面。	2023 年 11 月 9 日
IVS 实时流式传输入门	我们更新了 设置 IAM 权限 中的操作过程。	2023 年 10 月 20 日
监控实时直播功能	在CloudWatch 指标：IVS 实时流媒体中，我们添加了维度的样本值。	2023 年 10 月 17 日
广播 SDK：Web 指南	我们对 监控远程参与者媒体静音状态 部分进行了多处更改。	2023 年 10 月 17 日
广播 SDK：Web 1.6.0	在 real-time-streaming 广播 SDK 指南：Web 中，更新了新版本的版本号和工件链接。 Amazon IVS 文档登录页面指向最新版本的广播 SDK 参考。 另请参阅此发行版的 Amazon IVS 发布说明。 在《网络指南》的 “ MediaStream 从设备检索” 中，我们还删除了这两`max`行；最佳做法是仅指定`ideal`。 在“实时直播功能优化”中，我们新增了章节“优化音频比特率和立体声支持”。	2023 年 10 月 16 日
Stage Health：新 CloudWatch 指标	已发布 IVS 实时直播 CloudWatch 指标。请参阅 监控 IVS 实时直播功能。	2023 年 10 月 12 日

广播 SDK：Android 1.12.1	在 real-time-streaming 广播 SDK 指南：Android 中，更新了新版本的版本号和工件链接。此外还增加了新章节 使用蓝牙麦克风。 Amazon IVS 文档登录页面指向最新版本的广播 SDK 参考。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 10 月 12 日
广播 SDK：Web 1.5.2	在 real-time-streaming 广播 SDK 指南：Web 中，更新了新版本的版本号和工件链接。 Amazon IVS 文档登录页面指向最新版本的广播 SDK 参考。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 9 月 14 日
IVS 实时流式传输入门	在 Android > 安装广播 SDK 中，已添加数据绑定。	2023 年 9 月 12 日
广播 SDK 错误处理	在广播 SDK 指南：Web、Android 和 iOS 中，已添加“错误处理”部分。	2023 年 9 月 12 日
IVS 实时流式传输入门	在分发参与者令牌中，添加了一个关于不要基于当前的令牌格式构建功能的重要说明。	2023 年 9 月 1 日
IVS 实时流式传输入门	在设置 IAM 权限中，更新了权限集。	2023 年 8 月 31 日

广播 SDK：Web 1.5.1、Android 1.12.0 和 iOS 1.12.0	在 real-time-streaming 广播 SDK 指南中更新了新版本的版本号和工件链接：Web、Android 和 iOS。 在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 8 月 23 日
实时流式传输发布	此发行版附带主要的文档更改。我们将之前的文档重命名为 IVS Low-Latency Streaming，并发布了新的 IVS Real-Time Streaming 文档。IVS 文档登录页面现在有单独的实时流式传输和低延迟流式传输部分。每个部分都有各自的用户指南和 API 参考。 有关其他文档更改，请参阅 Document History（Low-Latency Streaming）。	2023 年 8 月 7 日
广播 SDK：Web 1.5.0、Android 1.11.0 和 iOS 1.11.0	已在广播 SDK 指南中更新了新版本的版本号和构件链接：Web、Android 和 iOS。 在 Amazon IVS 文档登录页面上，更新了广播 SDK 参考链接以指向新版本。 另请参阅此发行版的 Amazon IVS 发布说明。	2023 年 8 月 7 日

IVS Real-Time Streaming API Reference 更改

API 更改	描述	日期
合成录制	我们添加了 4 个 StorageConfiguration 端点和 7 个对象（DestinationDetail、、S3 RecordingConfiguration DestinationConfiguration、S3Detail、S3 StorageConfiguration、 StorageConfiguration、 StorageConfigurationSummary）。 我们修改了 3 个对象（构图、目的地、 DestinationConfiguration）。这会影响 GetComposition 响应以及 StartComposition 请求和响应。	2023 年 11 月 16 日
服务器端合成	我们添加了 8 个构图和 EncoderConfiguration 端点以及 11 个对象（ChannelDestinationConfiguration CompositionSummary、构图 DestinationConfiguration、 DestinationSummary、 EncoderConfiguration、 EncoderConfigurationSummary、 GridConfiguration、 LayoutConfiguration、、和视频）。	2023 年 11 月 16 日
Stage 运行状况：新的参与者数据	Participant 对象增加了六个字段：`browserName` 、`browserVersion` 、`ispName`、`osName`、`osVersion` 和 `sdkVersion` 。这会影响 GetParticipant 响应。	2023 年 10 月 12 日
参与者令牌	添加了一个关于不要基于当前的令牌格式构建功能的重要说明	2023 年 9 月 1 日
IVS 实时流式传输发布	此发行版附带主要的文档更改。我们将之前的文档重命名为 IVS Low-Latency Streaming，并发布了新的 IVS Real-Time Streaming 文档。IVS 文档登录页面现在有单独的实时流式传输和低延迟流式传输部分。每个部分都有各自的用户指南和 API 参考。 IVS Real-Time Streaming API Reference 是 IVS 实时流式传输文档的一部分。之前的标题为 IVS Stage	2023 年 8 月 7 日

API 更改	描述	日期
	API Reference。其之前的历史记录在 Document History（Low-Latency Streaming）中进行了描述。	

发布说明（实时直播功能）

2024年2月6日

OBS 和 WHIP Support

IVS 可以与 OBS 等兼容 Whip 的编码器一起使用，发布到 IVS 的实时流媒体中。WHIP（Webrtc-HTTP 摄取协议）是一份旨在标准化 WebRTC 摄取的 IETF 草案。参见 OBS 和 WHIP Support 上的新页面。

2024年2月1日

亚马逊 IVS Broadcast SDK：安卓 1.14.1、iOS 1.14.1、Web 1.8.0（实时直播）

平台	下载和更改
网络广播软件开发工具包 1.8.0	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference • 现在，默认情况下，使用联播进行分层编码处于禁用状态。 • 修复了在删除舞台或参与者与服务器断开连接时，舞台实例无法完全断开连接的问题。SDK 现在`STAGE_CONNECTION_STATE_CHANGED` 会发出状态为`DISCONNECTED` （而不是`ERRORED`然后`CONNECTING` ）的事件。 • 修复了使用空音轨或视频轨道更新策略时发布失败的问题。
安卓广播 SDK 1.14.1	参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.14.1/android • 现在，默认情况下，使用联播进行分层编码处于禁用状态。 • `libWebRTC` 从 M108 更新到 M119。

平台	下载和更改
	• 修复了几次崩溃以提高整体稳定性。 • 增加了对立体发布的支持。这可以通过`StageAudioConfiguration` 对象启用。 • 修复了导致参与者在加入会话后收到黑色提要的错误。 • 更新了内部`libWebRTC` 引用，以避免在同一宿主应用程序中包含其他`libWebRTC` 版本时发生符号冲突。
iOS 广播软件开发工具包 1.14.1	下载进行实时直播：https://broadcast.live-video.net/1.14.1/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.14.1/ios • 现在，默认情况下，使用联播进行分层编码处于禁用状态。 • `libWebRTC` 从 M108 更新到 M119。 • 修复了几次崩溃以提高整体稳定性。 • 增加了对立体发布的支持。这可以通过启用`IVSLocalStageStreamAudioConfiguration` 。 • 修复了为其他参与者启用纯音频模式时的崩溃问题。 • 改进了 TTV 并减小了二进制大小。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.223 MB	13.118 MB

架构	压缩大小	未压缩大小
armeabi-v7a	4.524 MB	9.134 MB
x86_64	5.418 MB	13.955 MB
x86	5.61 MB	14.369 MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	3.350 MB	7.790 MB

2024 年 1 月 3 日

亚马逊 IVS Broadcast SDK：安卓 1.13.4、iOS 1.13.4、Web 1.7.0（实时直播）

平台	下载和更改
网络广播软件开发工具包 1.7.0	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference • time-to-video 针对订阅者加入阶段进行了改进。 • 移除了该`minAudioBitrateKbps` 属性（未使用）。 • 改善了互联网中断或变更期间的网络恢复。
安卓广播 SDK 1.13.4	参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.13.4/android • StageAudioConfiguration 现在支持设置是否应启用回声消除。

平台	下载和更改
iOS 广播软件开发工具包 1.13.4	下载进行实时直播：https://broadcast.live-video.net/1.13.4/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.13.4/ios • 在 iOS 上，我们改进了用于录制和播放的音频引擎，重点是稳定性和可恢复性。这增强了在使用时对路线更改的支持，改善了边缘情况下的电池恢复，并减少了主线程阻塞量。 • 修复了即使麦克风脱离舞台，iOS 隐私指示灯仍可能保持活动状态的问题。（SDK 当时没有处理传入的音频。）

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.187 MB	13.025 MB
armeabi-v7a	4.491 MB	9.056 MB
x86_64	5.359 MB	13.829 MB
x86	5.553 MB	14.214 MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	3.45MB	7.84MB

2023 年 12 月 7 日

新 CloudWatch 指标

我们将 PacketLoss （阶段）指标重命名为 DownloadPacketLoss （阶段）。我们还发布了 IVS 实时直播的其他 CloudWatch 指标：

- DownloadPacketLoss （舞台、参与者）
- DroppedFrames （舞台、参与者）
- SubscribeBitrate （舞台、参与者、MediaType）

有关详细信息，请参阅监控 IVS 实时直播功能。

2023 年 12 月 4 日

Amazon IVS 广播 SDK：Android 1.13.2 和 iOS 1.13.2（实时直播）

平台	下载和更改
所有移动设备（Android 和 iOS）	• 开发人员可以启用/禁用噪音抑制配置以进行发布。
Android 广播 SDK 1.13.2	参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.13.2/android • 缩短了加入会话第一个舞台时加载视频（TTV）所需的时间。
iOS 广播 SDK 1.13.2	下载进行实时直播：https://broadcast.live-video.net/1.13.2/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.13.2/ios • 实时 SDK 没有变化。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.177MB	13.01MB
armeabi-v7a	4.485MB	9.045MB
x86_64	5.352MB	13.808MB
x86	5.547MB	14.192MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	3.45MB	7.82MB

2023 年 11 月 21 日

Amazon IVS 广播 SDK：Android 1.13.1（实时直播功能）

平台	下载和更改
Android 广播 SDK 1.13.1	参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.13.1/android • 修复了快速离开、释放和重新加入同一舞台时导致崩溃的问题。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.177MB	13.102MB

架构	压缩大小	未压缩大小
armeabi-v7a	4.485MB	9.046MB
x86_64	5.353MB	13.809MB
x86	5.547MB	14.192MB

2023 年 11 月 17 日

Amazon IVS 广播 SDK：Android 1.13.0 和 iOS 1.13.0（实时直播功能）

平台	下载和更改
所有移动设备（Android 和 iOS）	• 更新了流式传输优化。除其他外，“自适应流式传输：通过联播分层编码”功能现在需要明确的选择，并且仅支持最新版本的 SDK。 • 通过减少罕见崩溃的发生，提高了阶段的稳定性。 • 缩短了加入舞台时加载视频（TTV）所需的时间。 • 改善了使用蓝牙设备时的体验。 • 优化了 SDK 的 CPU 和内存使用率，并降低了库的大小。 • 添加了 `StageAudioManager` 类，该类可用于设置音频采集和播放参数，包括语音通信、媒体播放等的预设。有关详细信息，请参阅新页面“IVS 广播 SDK：移动音频模式”。 • 增加了新 `requestQualityStats` 功能，用于显示来自 WebRTC 统计数据的结构化质量事件。 • 增加了更新音频比特率的新功能。它在 `LocalStageStream` 对象上设置，就像视频配置一样，但是通过新音频配置对象进行。

平台	下载和更改

Android 广播 SDK 1.13.0	参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.13.0/andro id
	• StageRenderer 接口上的所有方法现在均为可选。
	• 增加了对基于 Surfaceview 的预览的支持，以提高性能。Session 和 StageStream 中的现有 getPreview 方法将继续返回 TextureView 的子类，但在未来的 SDK 版本中，这可能会有变化。
	  • 特别地，如果您的应用程序依赖于 TextureView ，则无需更改即可继续。您也可以从 getPreview 切换到 getPreviewTextureView ，为默认 getPreview 返回的内容的最终更改做好准备。
	  • 如果您的应用程序没有特别要求 TextureView ，我们建议切换到 getPreviewSurfaceView 以降低 CPU 和内存使用率。
	• SDK 现在实现了一种名为 ImagePreviewSurfaceTarget 的新型预览，适合与应用程序提供的 Android Surface 对象共用。它不是 Android View 的子类，具有更高的灵活性。
	• 修复了在错误的时间用错误大小调用远程参与者的 onFrame 回调的情况。
	• SurfaceSource # getInputSurface 现在标注为 @Nullable 。您的代码应在使用之前进行检查。
	• 已将 UserId 和 attributes 添加到 ParticipantInfo 。UserId 和 attributes 属性嵌入在令牌中，每当

平台	下载和更改
	有参与者加入时，应用程序都可以通过 `ParticipantInfo` 来检索它们。 • 相机捕捉和预览渲染现在默认为 720 x 1280，或者 15 fps 的发布分辨率（以较高者为准）。您可以使用 `StageVideoConfiguration # setCameraCaptureQuality` 调整分辨率和/或 fps。 • 设置配置属性时引发的 `IllegalArgumentException` 现在包括异常消息中提供的值。

平台	下载和更改
iOS 广播 SDK 1.13.0	实时直播下载：https://broadcast.live-video.net/1.13.0/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.13.0/ ios • 修复了在发布前更新视频配置时 SDK 不会更改视频配置的问题。 • 加入了 Google 针对 LibVPX 安全漏洞的修复程序（CVE-2023-5217）。（请注意，Android SDK 不需要就此问题进行任何更改。） • 使用包含 `libWebRTC` 的其他库的应用程序将不再与 IVS 广播 SDK 产生冲突。 • `IVSStageRenderer` 协议上的所有方法现在都已标记 `@optional` 。 • 如 SDK 本身所述，我们的 SDK 返回的麦克风和摄像机现在可以保证排序顺序。 • 现在，多台摄像机的 `isDefault` 属性值可以为 `true`，各自对应一个由操作系统确定的位置。 • 增加了 `IVSStageAudioManager` ，它允许对底层 `AVAudioSession` 进行精确控制，从而为舞台功能提供更多种类的使用案例。 • 已将 `UserId` 添加到 `ParticipantInfo` 。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.17MB	13.00MB

架构	压缩大小	未压缩大小
armeabi-v7a	4.48MB	9.04MB
x86_64	5.35MB	13.80MB
x86	5.54MB	14.18MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	3.45MB	7.84MB

2023 年 11 月 16 日

合成录制

此新功能可将 IVS 舞台的合成视图录制到 Amazon S3 存储桶。有关更多信息，请参阅：

- 合成录制 — 这是新页面。
- IVS 实时直播功能入门 — 我们向“设置 IAM 权限”中的策略中添加了 S3 端点。
- 服务限额：为新端点添加了调用速率限额。
- IVS 实时流媒体 API 参考 — 我们添加了 4 个 StorageConfiguration 端点和 7 个对象（DestinationDetail、、S3 RecordingConfiguration DestinationConfiguration、S3Detail、S3 StorageConfiguration、、）。 StorageConfiguration StorageConfigurationSummary我们还修改了 3 个对象（构成、目标 DestinationConfiguration）；这会影响 GetComposition 响应以及 StartComposition 请求和响应。

2023 年 11 月 16 日

服务器端合成

IVS 服务器端合成让客户端能够将 IVS 舞台的合成和广播卸载到 IVS 托管的服务。服务器端合成以及向频道的 RTMP 广播通过舞台主区域的 IVS 控制面板端点调用。有关更多信息，请参阅：

- IVS 实时直播功能入门 — 我们向“设置 IAM 权限”中的策略中添加了 SSC 端点。
- 将 Amazon EventBridge 与 IVS 实时流媒体配合使用 — 我们添加了新的指标。
- 服务器端合成 — 这份新文档包括概述和设置说明。
- 服务限额（实时直播功能）– 我们增加了新的调用速率限制和其他限额。
- 实时直播 API 参考 — 我们添加了 8 个合成和 EncoderConfiguration端点以及 11 个对象（ChannelDestinationConfiguration CompositionSummary、构图 DestinationConfiguration、DestinationSummary、、 EncoderConfiguration、 EncoderConfigurationSummary、GridConfiguration、 LayoutConfiguration、、和视频）。

在 IVS Low-Latency Streaming User Guide 中，请参阅：

- 在 IVS 流中启用多台主机 — 我们增加了“广播舞台：客户端与服务器端合成”，并更新了“4. 广播舞台。”

2023 年 10 月 16 日

Amazon IVS 广播 SDK：Web 1.6.0（实时直播功能）

平台	下载和更改
Web 广播 SDK 1.6.0	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference • 改进了视频生成时间（TTV）。 • 增加了 `maxAudioBitrate` 配置，支持高达 128kbps 的单声道或立体声音频通道。

2023 年 10 月 12 日

新的 CloudWatch 指标和参与者数据

我们发布了 IVS 实时直播的 CloudWatch 指标。有关详细信息，请参阅监控 IVS 实时直播功能。

Participant API 对象也增加了六个字段：`browserName`、`browserVersion`、`ispName`、`osName`、`osVersion` 和 `sdkVersion`。这会影响 GetParticipant 响应。请参阅 IVS Real-Time Streaming API Reference。

2023 年 10 月 12 日

Amazon IVS 广播 SDK：Android 1.12.1（实时直播功能）

平台	下载和更改
Android 广播 SDK 1.12.1	参考文档：https://aws.github.io/amazon-ivs-broadcast-docs/1.12.1/android • 修复了调用 `BroadcastSession.setListener` 会导致错误的问题。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.853MB	16.375 MB
armeabi-v7a	4.895MB	10.803MB
x86_64	6.149MB	17.318MB
x86	6.328MB	17.186MB

2023 年 9 月 14 日

Amazon IVS 广播 SDK：Web 1.5.2（实时直播功能）

平台	下载和更改
Web 广播 SDK 1.5.2	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference

平台	下载和更改
	• 修复了在发布状态进入 `refreshStrategy` 状态时无法使用 `ERRORED` 重新发布的错误。

2023 年 8 月 23 日

Amazon IVS 广播 SDK：Web 1.5.1、安卓 1.12.0 和 iOS 1.12.0（实时直播功能）

平台	下载和更改
Web 广播 SDK 1.5.1	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference • 修复了内部 Maybe 类型在 TypeScript 5 上的错误。 • 为联播支持添加了更好的检测功能。 • 修复了尝试发布时 `refreshStrategy` 的两种争用情况。 • 修复了尝试更新要订阅的参与者时 `refreshStrategy` 的一种争用情况。
所有移动设备（Android 和 iOS）	• 修复了发布操作无法完成的罕见问题。 • 通过减少罕见崩溃的发生，提高了阶段的稳定性。 • 通过解决由快速加入/离开导致的争用条件问题，提高了舞台的稳定性。 • 在 `ImageDevice` 上添加了一个新的 `setOnFrameCallback` 方法。这允许在帧穿过设备本身时进行观察，从而深入了解最新图像的纵横比。此方法还可用于检测何时为舞台中的远程参与者渲染第一帧。

平台	下载和更改
Android 广播 SDK 1.12.0	参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.12.0/andro id • 现在支持 Android 9。 • 提高了 CPU 使用率和性能。
iOS 广播 SDK 1.12.0	实时直播下载：https://broadcast.live-video.net/1.12.0/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.12.0/ ios • 更正了 `IVSDeviceDiscovery.createAudioSourceWithName` 的签名以返回 `IVSCustomAudioSource` 而不是 `IVSCustomImageSource` 。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.853MB	16.375 MB
armeabi-v7a	4.895MB	10.803MB
x86_64	6.149MB	17.318MB
x86	6.328MB	17.186MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	5.06MB	10.92MB

2023 年 8 月 7 日

Amazon IVS 广播 SDK：Web 1.5.0、Android 1.11.0 和 iOS 1.11.0

平台	下载和更改
Web 广播 SDK 1.5.0	参考文档：https://aws.github.io/ amazon-ivs-web-broadcast /docs/ sdk-reference • 添加了联播 – 启用此功能后，发布者可以发送高质量和低质量的视频层。订阅用户根据其网络状况自动选择最佳视频质量。请参阅 Optimizing Media。
所有移动设备（Android 和 iOS）	添加了联播 – 启用此功能后，发布者可以发送高质量和低质量的视频层。订阅用户根据其网络状况自动选择最佳视频质量。请参阅 Android 和 iOS Broadcast SDK Guides 中的“Enable/Disable Layered Encoding with Simulcast”。
Android 广播 SDK 1.11.0	参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.11.0/andro id • 修复了创建多个舞台最终导致崩溃的问题。（舞台的确切数量取决于设备。）
iOS 广播 SDK 1.11.0	实时直播下载：https://broadcast.live-video.net/1.11.0/AmazonIVSBroadcast-Stages.xcframework.zip 参考文档：https://aws.github.io/ amazon-ivs-broadcast-docs /1.11.0/ ios • 更正了 `IVSDeviceDiscovery.createAudioSourceWithName` 而不是 `IVSCustomImageSource` 的签名以返回 `IVSCustomAudioSource` 。

广播开发工具包大小：Android

架构	压缩大小	未压缩大小
arm64-v8a	5.811 MB	16.186 MB
armeabi-v7a	4.857 MB	10.646 MB
x86_64	6.108 MB	17.122 MB
x86	6.289 MB	16.994 MB

广播开发工具包大小：iOS

架构	压缩大小	未压缩大小
arm64	5.030 MB	10.810 MB

2023 年 8 月 7 日

实时直播功能

Amazon Interactive Video Service（IVS）实时直播功能让您能够以低于 300 毫秒的延迟在主机和观众之间传送实时流。

此发行版附带主要的文档更改。IVS 文档登录页面现在有单独的实时直播功能和低延迟直播功能部分。每个部分都有各自的用户指南和 API 参考。有关文档的详细信息，请参阅文档历史记录（实时和低延迟文档变更）。对于实时直播功能，请从 IVS Real-Time Streaming User Guide 和 IVS Real-Time Streaming API Reference 开始。

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。