aws

开发人员指南

AWS Device Farm

API 版本 2015-06-23

AWS Device Farm: 开发人员指南

Table of Contents

什么是 AWS Device Farm？

Device Farm 是一项应用程序测试服务，您可以用它在由 Amazon Web Services (AWS) 托管的实际物理手机和平板电脑上测试您的 Android、iOS 和 Web 应用程序并与其交互。

使用 Device Farm 的两种主要方法是：

- 从本地计算机远程访问设备，可以在网络浏览器中以交互方式访问设备，也可以从本地客户端使用 Appium自动对其进行测试。
- 使用 Device Farm 的托管测试执行环境自动执行应用程序测试。

> **Note**
>
> Device Farm 仅在 `us-west-2`（俄勒冈）区域中提供。

远程访问

远程访问允许您通过网络浏览器与设备进行实时交互。远程访问还允许您使用托管 Appium 端点从本地客户端对远程 Device Farm 设备运行 Appium 测试。

在许多场景中，与设备进行实时交互可能很有用，例如手动应用程序测试、在特定设备上重现错误、在不同屏幕类型上检查应用程序的视觉呈现以及应用程序安装和升级顺序。Device Farm 的完全托管的 Appium 端点使您能够开发、测试和调试 Appium 测试，从而快速提供反馈。

Appium 端点支持你选择的任何语言、任何本地 IDE、带断点的实时调试、实时视频和日志以及 Appium Inspector 之类的工具。在远程访问会话期间，您可以在同一台设备上随心所欲地多次执行测试，限制为 150 分钟。

在远程访问会话期间，Device Farm 会记录您与设备交互时发生的操作的详细信息。在会话结束时会生成包含这些详细信息的日志和会话的视频捕获。

自动应用程序测试

Device Farm 允许您通过上传应用和测试在多台设备上并行运行自动测试。测试将在完全托管的环境中自动执行，测试主机可以配置测试规范文件。该环境使用 Device Farm 的测试主机，因此您无需担心

为运行测试配置自己的基础架构。测试主机和设备可以安全地连接到您的 VPC 以访问您的私有终端节点。

测试完成后，将生成一份测试报告，其中包含高级结果、低级日志、屏幕截图和您的测试工件。

Device Farm 支持测试原生和混合安卓和 iOS 应用程序。有关支持的测试类型的更多信息，请参阅AWS Device Farm 中的测试框架和内置测试。

术语

Device Farm 引入了以下定义信息组织方式的术语：

设备池

表示通常具有相似的特征（如平台、制造商或型号）的设备的集合。

作业

在单个设备上测试单个应用程序的 Device Farm 请求。一个任务包含一个或多个套件。

计量

指设备的计费。文档和 API 参考中可能会提及计量设备或非计量设备。有关定价的更多信息，请参阅 AWS Device Farm 定价。

项目

包含运行的逻辑工作区，一次运行用于单个应用程序在一个或多个设备上的每个测试。您可以使用项目以您选择的任何方式组织工作区。例如，可以每个应用程序名称一个项目，也可以每个平台一个项目。您可以根据需要创建任意数量的项目。

报告

包含有关运行的信息，这是在一个或多个设备上测试单个应用程序的 Device Farm 请求。有关更多信息，请参阅 AWS Device Farm 中的报告。

运行

您的应用程序的特定版本，使用一组特定的测试，在一组特定的设备上运行。运行将生成一个结果报告。一次运行包含一个或多个任务。有关更多信息，请参阅 运行。

会话

通过 Web 浏览器与实际、物理设备的实时交互。有关更多信息，请参阅 会话。

套件

测试程序包中的测试的分层组织。一个套件包含一个或多个测试。

测试

测试程序包中的单个测试案例。

有关 Device Farm 的更多信息，请参阅概念。

设置

要使用 Device Farm，请参阅 设置。

设置 AWS Device Farm

首次使用 Device Farm 前，您必须完成以下任务：

主题

第 1 步：注册 AWS

注册 Amazon Web Services (AWS)。

如果您没有 AWS 账户，请完成以下步骤来创建一个。

要注册 AWS 账户

1. 打开https://portal.aws.amazon.com/billing/注册。
2. 按照屏幕上的说明操作。

   在注册时，将接到电话或收到短信，要求使用电话键盘输入一个验证码。

   当您注册时 AWS 账户，就会创建AWS 账户根用户一个。根用户有权访问该账户中的所有 AWS 服务 和资源。作为最佳安全实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

步骤 2：在您的 AWS 账户中创建或使用 IAM 用户

我们建议您不要使用 AWS 根账户来访问 Device Farm。相反，请在您的 AWS 账户中创建一个 AWS Identity and Access Management (IAM) 用户（或使用现有用户），然后使用该 IAM 用户访问 Device Farm。

有关更多信息，请参阅创建 IAM 用户(AWS 管理控制台)。

步骤 3：为 IAM 用户提供访问 Device Farm 的权限

为 IAM 用户提供访问 Device Farm 的权限。要执行此操作，请在 IAM 中创建访问策略，然后将该访问策略分配到 IAM 用户，如下所示。

> **Note**
>
> 用于完成以下步骤的 AWS 根账户或 IAM 用户必须有权创建以下 IAM 策略并将其附加到 IAM 用户。有关更多信息，请参阅使用策略。

1. 使用以下 JSON 正文创建策略。给它起一个描述性的标题，例如。*DeviceFarmAdmin*

   有关如何创建 IAM policy的信息，请参阅《IAM 用户指南》中的创建 IAM policy。
2. 将您创建的 IAM policy 附加到新用户。有关将 IAM policy 附加到用户的更多信息，请参阅《IAM 用户指南》中的添加和删除 IAM policy。

通过附加策略，IAM 用户将具有与该 IAM 用户关联的所有 Device Farm 操作和资源的访问权限。有关如何仅允许 IAM 用户使用一组有限的 Device Farm 操作和资源的信息，请参阅 AWS Device Farm 中的身份识别和访问管理。

后续步骤

现在您已准备就绪，可以开始使用 Device Farm 了。请参阅Device Farm 入门。

Device Farm 入门

本演练展示了如何使用 Device Farm 来测试 Android 或 iOS 原生应用程序。您可以使用 Device Farm 控制台创建项目，上传 .apk 或 .ipa 文件，运行一系列标准测试，然后查看结果。

> **Note**
>
> Device Farm 仅在 `us-west-2`（俄勒冈）AWS 区域中提供。

主题

先决条件

在开始之前，请确保您已满足以下要求：

- 完成设置中的步骤。您需要一个 AWS 账户和一个有权访问 Device Farm 的 AWS Identity and Access Management (IAM) 用户。

- 对于 Android，您可以提交 .apk（Android 应用程序包）文件，或者使用我们提供的示例应用程序。对于 iOS，您需要 .ipa 文件（iOS 应用程序存档）文件。您可在此演练的稍后步骤中将文件上传到 Device Farm。

  > **Note**
  >
  > 确保为 iOS 设备 (而不是模拟器) 构建您的 .ipa 文件。

- （可选）您需要从 Device Farm 支持的某个测试框架中进行测试。您要将此测试包上传到 Device Farm，然后在本演练的稍后步骤中运行测试。(如果没有可用的测试包，则可以指定并运行标准的内置测试套件。) 有关更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

步骤 1：登录到 控制台

您可以使用 Device Farm 控制台创建和管理测试项目及运行。您将在本演练的稍后步骤中了解项目和运行。

- 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。

步骤 2：创建项目

要在 Device Farm 中测试应用程序，您必须首先创建一个项目。

1. 在导航窗格中，选择移动设备测试，然后选择项目。
2. 在移动设备测试项目下，选择创建项目。
3. 在创建项目下，输入项目名称（例如，**MyDemoProject**）。
4. 选择创建。

   控制台将打开您新创建的项目的自动测试页面。

步骤 3：创建和启动运行

现在您已经有了一个项目，下面就可以创建并启动运行。有关更多信息，请参阅 运行。

1. 在自动化测试选项卡上，选择创建运行。或者，您可以通过选择使用教程创建运行来按照控制台中的教程进行操作。
2. （可选）在运行设置下的运行名称部分中，输入您的运行名称。如果未提供名称，则默认情况下，Device Farm 控制台会将您的运行命名为“My Device Farm run”。
3. 在运行设置下的运行类型部分中，选择您的运行类型。如果您没有可供测试的应用程序，或者您测试的是 Android（.apk）应用程序，请选择 Android 应用程序。如果您测试的是 iOS（.ipa）应用程序，请选择 iOS 应用程序。
4. 如果您没有可供测试的应用程序，请在选择应用程序下的应用程序选择选项部分中，选择选择 Device Farm 提供的示例应用程序。如果您自带应用程序，请选择上传自己的应用程序，然后选择

您的应用程序文件。如果您要上传 iOS 应用程序，请确保选择 iOS device (iOS 设备)，而不是模拟器。

5. 在配置测试下的选择测试框架部分中，选择一个测试框架或内置测试套件。有关各选项的信息，请参阅 测试框架和内置测试。
   - 如果您尚未为 Device Farm 打包测试，请选择内置：模糊来运行标准的内置测试套件。您可以保留事件计数、事件限制和随机发生器种子的默认值。有关更多信息，请参阅 the section called “内置：模糊（Android 和 iOS）”。
   - 如果您有来自支持的测试框架之一的测试包，请选择相应的测试框架，然后上传包含您的测试的文件。
6. 在选择设备下，选择使用设备池和主要设备。
7. （可选）要添加其他配置，请打开其他配置下拉列表。在此部分，您可以执行以下任意操作：
   - 要为 Device Farm 提供其他将在运行期间使用的数据，请在添加额外数据旁边选择选择文件，然后浏览到并选择包含这些数据的 .zip 文件。
   - 要安装 Device Farm 将在运行期间使用的其他应用程序，请在安装其他应用程序旁边选择选择文件，然后浏览到并选择包含该应用程序的 .apk 或 .ipa 文件。为您要安装的其他应用程序重复此操作。在上传应用程序之后，您可以拖放应用程序来更改应用程序的安装顺序。
   - 要指定是否将在运行期间启用 Wi-Fi、蓝牙、GPS 或 NFC，请在 Set radio states (设置电台) 旁边选中相应框。
   - 要为运行预设设备纬度和经度，请在 Device location (设备位置) 旁边输入坐标。
   - 要为运行预设设备区域设置，请在设备区域设置中选择区域设置。
   - 选择启用视频记录以在测试期间启用视频记录。
   - 选择启用应用程序性能数据捕获以启用从设备捕获性能数据。

> **Note**
>
> 目前，设置设备无线电状态和区域设置选项仅适用于 Android 本地测试。

> **Note**
>
> 如果您有私有设备，还将显示特定于私有设备的配置。

8. 在页面底部，选择创建运行以安排运行。

Device Farm 将在设备可用后立即启动运行，通常在几分钟内启动。要查看运行状态，请在项目的自动测试页面上，选择运行名称。在运行页面上，在设备下，每台设备都以设备表中的待处理图标

开头，然后在测试开始时切换到运行图标

。

每次测试完成后，控制台会在设备名称旁边显示一个测试结果图标。所有测试完成后，运行旁边的待处理图标将变为测试结果图标。

步骤 4：查看运行结果

要查看运行的测试结果，请在项目的自动测试页面上，选择运行的名称。此时将显示摘要页面：

- 按结果列出的测试总数。
- 具有唯一的警告或故障的测试列表。
- 设备和每个设备的测试结果的列表。
- 在运行期间捕获的任何屏幕截图，按设备分组。
- 用于下载解析结果的部分。

有关更多信息，请参阅 在 Device Farm 中查看测试报告。

后续步骤

有关 Device Farm 的更多信息，请参阅概念。

在 Device Farm 中购买设备槽

你可以使用 Device Farm 控制台、 AWS Command Line Interface (AWS CLI) 或 Device Farm API 来购买设备插槽。

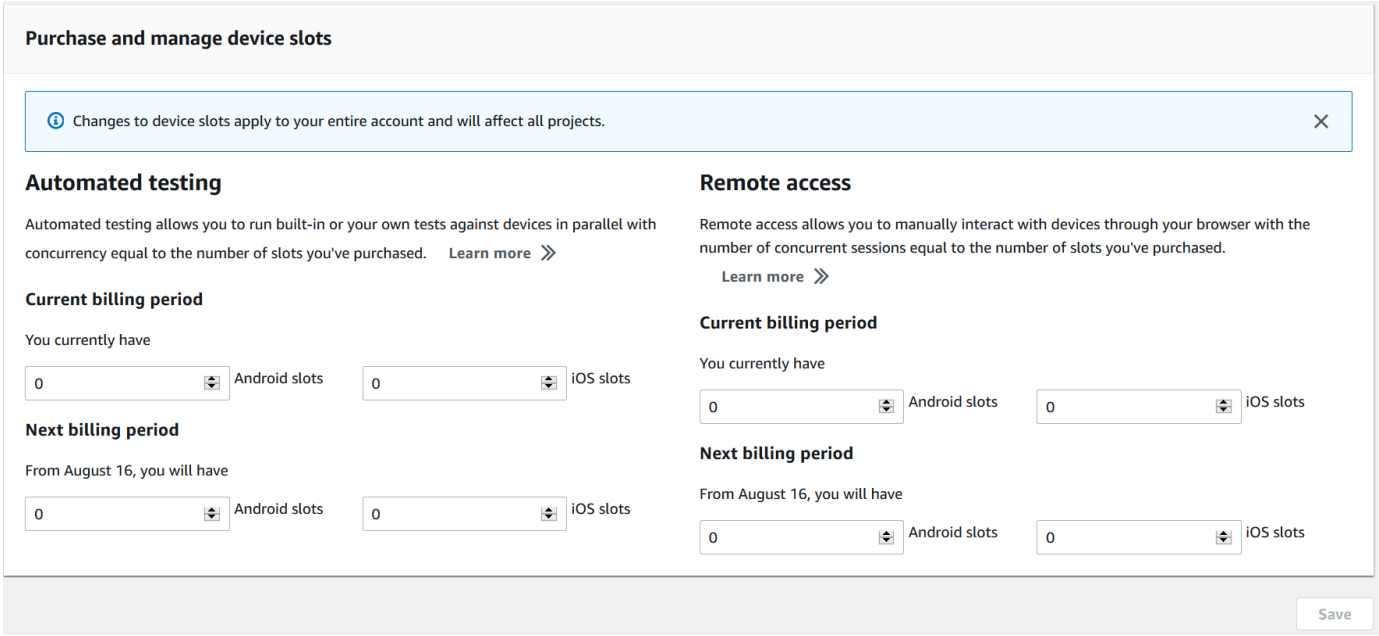
购买设备槽（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在导航窗格中，选择移动设备测试，然后选择设备槽。
3. 在购买和管理设备槽页面上，您可以通过选择要购买的自动测试和远程访问设备的槽数来创建自己的自定义包。指定当前和下一个计费周期的槽数。

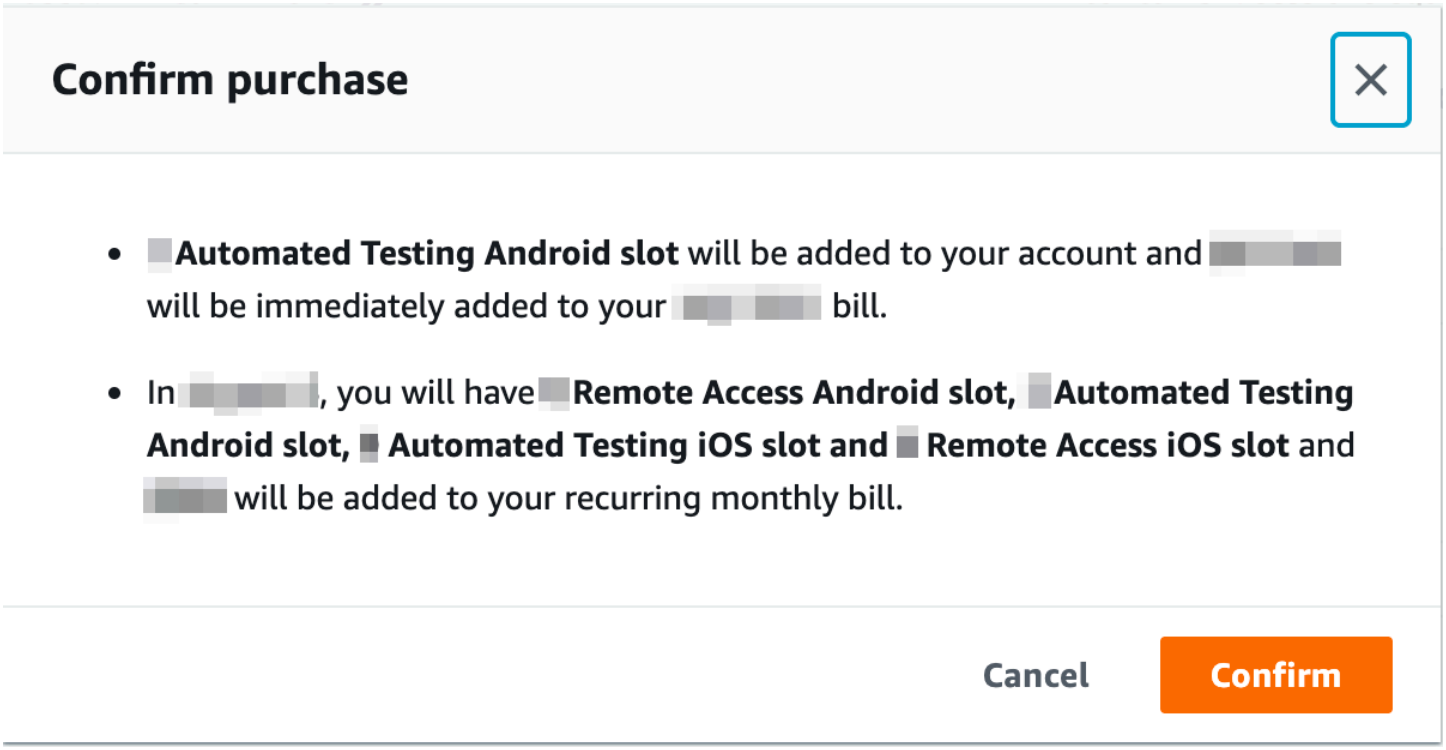
   当您更改槽数时，文本会随着账单金额而动态更新。有关更多信息，请参阅 AWS Device Farm 定价。

   > **Important**
   >
   > 如果您更改了设备插槽的数量，但看到 “联系我们” 或 “联系我们” 购买消息，则说明您的 AWS 账户尚未获得购买您请求数量的设备插槽的批准。
   > 这些选项会提示您向 Device Farm 支持团队发送电子邮件。在电子邮件中，指定您要购买的每种设备类型的数量以及计费周期。

   > **Note**
   >
   > 对设备槽的更改适用于您的整个帐户并会影响所有项目。



4. 选择 Purchase (购买)。系统将显示确认购买窗口。查看信息，然后选择确认以完成交易。



在购买和管理设备槽页面上，您可以看到当前拥有的设备槽数。如果您增加或减少了槽数，则在您做出更改之日后的一个月内会看到您将具有的槽数。

购买设备槽 (AWS CLI)

您可以运行 purchase-offering 命令来购买产品。

要列出您的 Device Farm 账户设置（包括您可以购买的最大设备槽数和您具有的剩余免费试用分钟数），请运行 get-account-settings 命令。将会看到类似下面的输出：

```
{
    "accountSettings": {
        "maxSlots": {
            "GUID": 1,
            "GUID": 1,
            "GUID": 1,
            "GUID": 1
        },
        "unmeteredRemoteAccessDevices": {
            "ANDROID": 0,
            "IOS": 0
        },
        "maxJobTimeoutMinutes": 150,
        "trialMinutes": {
            "total": 1000.0,
            "remaining": 954.1
        },
        "defaultJobTimeoutMinutes": 150,
        "awsAccountNumber": "AWS-ACCOUNT-NUMBER",
        "unmeteredDevices": {
            "ANDROID": 0,
            "IOS": 0
        }
    }
}
```

要列出可供您使用的设备槽产品，请运行 list-offerings 命令。您应该可以看到类似于如下所示的输出内容：

```
{
    "offerings": [
        {
            "recurringCharges": [
                {
                    "cost": {
```

```
                    "amount": 250.0,
                    "currencyCode": "USD"
                },
                "frequency": "MONTHLY"
            }
        ],
        "platform": "IOS",
        "type": "RECURRING",
        "id": "GUID",
        "description": "iOS Unmetered Device Slot"
    },
    {
        "recurringCharges": [
            {
                "cost": {
                    "amount": 250.0,
                    "currencyCode": "USD"
                },
                "frequency": "MONTHLY"
            }
        ],
        "platform": "ANDROID",
        "type": "RECURRING",
        "id": "GUID",
        "description": "Android Unmetered Device Slot"
    },
    {
        "recurringCharges": [
            {
                "cost": {
                    "amount": 250.0,
                    "currencyCode": "USD"
                },
                "frequency": "MONTHLY"
            }
        ],
        "platform": "ANDROID",
        "type": "RECURRING",
        "id": "GUID",
        "description": "Android Remote Access Unmetered Device Slot"
    },
    {
        "recurringCharges": [
            {
```

```
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "IOS",
            "type": "RECURRING",
            "id": "GUID",
            "description": "iOS Remote Access Unmetered Device Slot"
        }
    ]
}
```

要列出可用的产品促销，请运行 list-offering-promotions 命令。

> **Note**
>
> 此命令仅返回您尚未购买的促销产品。在您使用促销购买了任何产品中的一个或多个槽后，该促销产品将不再出现在结果中。

您应该可以看到类似于如下所示的输出内容：

```
{
    "offeringPromotions": [
        {
            "id": "2FREEMONTHS",
            "description": "New device slot customers get 3 months for the price of 1."
        }
    ]
}
```

要获取产品状态，请运行 get-offering-status 命令。您应该可以看到类似于如下所示的输出内容：

```
{
    "current": {
        "GUID": {
            "offering": {
                "platform": "IOS",
```

```
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    },
    "nextPeriod": {
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "IOS",
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    }
}
```

renew-offering 和 list-offering-transactions 命令也可用于此功能。有关更多信息，请参阅AWS CLI 参考文档。

购买设备槽 (API)

1. 调用GetAccountSettings操作列出您的账户设置。
2. 致电ListOfferings运营部门列出可供您使用的设备插槽产品。
3. 致电ListOfferingPromotions运营部门列出可用的优惠促销。

> **Note**
>
> 此命令仅返回您尚未购买的促销产品。在您使用产品促销购买了一个或多个槽后，该促销产品将不再出现在结果中。

4. 致电PurchaseOffering运营部门购买产品。
5. 致电GetOfferingStatus运营部门以获取报价状态。

RenewOffering 和 ListOfferingTransactions 命令也可用于此功能。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

在 Device Farm 中取消设备槽

您可以取消自动测试和远程访问的设备槽数。有关说明，请参阅以下各部分。下一个计费周期对您的账户收取的金额将列在“计费周期”字段下方。

有关设备槽的更多信息，请参阅在 Device Farm 中购买设备槽。

取消设备槽（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在导航窗格中，选择移动设备测试，然后选择设备槽。
3. 在购买和管理设备槽页面上，您可以通过减少下一个计费周期下的值来减少自动测试和远程访问的设备槽数。下一个计费周期对您的账户收取的金额将列在“计费周期”字段下方。
4. 选择保存。系统将显示确认更改窗口。查看信息，然后选择确认以完成交易。

取消设备插槽 (AWS CLI)

您可以运行 renew-offering 命令来更改下一个计费周期的设备数量。

取消设备槽（API）

调用RenewOffering操作以更改您账户中的设备数量。

AWS Device Farm 的概念

Device Farm 是一项应用程序测试服务，您可以用它在由 Amazon Web Services (AWS) 托管的实际物理手机和平板电脑上测试您的 Android、iOS 和 Web 应用程序并与其交互。

本节介绍重要的 Device Farm 概念。

- AWS Device Farm 中的设备支持
- AWS Device Farm 中的测试环境
- 运行
- 应用程序
- AWS Device Farm 中的报告
- 会话

有关 Device Farm 中支持的测试类型的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

AWS Device Farm 中的设备支持

以下几节提供有关 Device Farm 中的设备支持的信息。

主题

- 支持的设备
- 设备池
- 私有设备
- 设备品牌
- 设备槽
- 预安装的设备应用程序
- 设备功能

支持的设备

Device Farm 为数百个独特的常用 Android 和 iOS 设备和操作系统组合提供支持。可用设备的列表随着新设备进入市场而扩大。有关设备的完整列表，请参阅AWS 控制台中的交互式设备列表。

设备池

Device Farm 将其设备组织成设备池，您可用于进行测试。这些设备池包含相关设备，例如只运行在 Android 上或只运行在 iOS 上的设备。Device Farm 提供了精选设备池，例如主要设备的设备池。您还可以创建混合使用公有和私有设备的设备池。

私有设备

私有设备允许您针对测试需要指定精确的硬件和软件配置。某些配置（例如已获得 root 权限的 Android 设备）可以作为私有设备支持。每个私有设备都是 Device Farm 在 Amazon 数据中心代表您部署的物理设备。您的私有设备专门供您用于自动和手动测试。在您选择终止订阅后，将从我们的环境中删除硬件。有关更多信息，请参阅私有设备和AWS Device Farm 中的私有设备。

设备品牌

Device Farm在各种移动设备和平板电脑上运行测试 OEMs.

设备槽

设备槽对应于并发性，您购买的设备槽的数量决定您可以在测试或远程访问会话中运行多少个设备。

设备槽有两种类型：

- 远程访问设备槽是您可以在远程访问会话中并发运行的设备槽。

  如果您有一个远程访问设备槽，则您每次只能运行一个远程访问会话。如果您购买了其他远程测试设备槽，则可以并发运行多个会话。
- 自动测试设备槽是您可以在其上并发运行测试的设备槽。

  如果您有一个自动测试设备槽，则您每次只能在一个设备上运行测试。如果您购买了其他自动测试设备槽，则可以在多个设备上并发运行多个测试以更快地获得测试结果。

您可以根据设备系列购买设备槽 (用于自动测试的 Android 或 iOS 设备，用于远程访问的 Android 或 iOS 设备)。有关更多信息，请参阅 Device Farm 定价。

预安装的设备应用程序

Device Farm 中的设备包括由制造商和运营商预安装的少量应用程序。

设备功能

所有设备都有互联网连接。它们没有运营商连接，无法打电话或发送 SMS 消息。

您可以使用任何支持前置或后置摄像头的设备拍摄照片。由于设备安装方式的不同，照片可能看起来比较暗和模糊。

Google Play 服务和谷歌浏览器安装在安卓设备上。

AWS Device Farm 中的测试环境

AWS Device Farm 提供自定义测试环境和标准测试环境用于运行自动测试。您可以选择自定义测试环境，以完全掌控您的自动测试。或者，您也可以选择 Device Farm 默认的标准测试环境，为自动测试套件中的每个测试提供精细报告。

主题

标准测试环境

在标准环境中运行测试时，Device Farm 会为测试套件中的每个案例提供详细的日志和报告。您可以查看每个测试的性能数据、视频、屏幕截图和日志，以查明并解决应用程序中的问题。

> **Note**
>
> 由于 Device Farm 在标准环境中提供精细报告，因此测试执行时间可能会比您在本地运行测试所用的时间更长。如果您希望缩短执行时间，请在自定义测试环境中运行测试。

自定义测试环境

当您自定义测试环境时，可以指定 Device Farm 应运行以执行测试的命令。这样可确保 Device Farm 上测试的运行方式类似于本地计算机上测试的运行方式。在此模式下运行测试还支持测试的实时日志和视频流。当您在自定义测试环境中运行测试时，将不会获得每个测试案例的精细报告。有关更多信息，请参阅 AWS Device Farm 中的自定义测试环境。

在使用 Device Farm 控制台、 AWS CLI或 Device Farm API 创建测试运行时，可以选择使用自定义测试环境。

有关更多信息，请参阅使用 AWS CLI 和 在 Device Farm 中创建测试运行 上传自定义测试规范。

AWS Device Farm 中的运行

以下几节包含有关 Device Farm 中的运行的信息。

Device Farm 中的运行代表您的应用程序的特定版本，使用一组特定的测试，在一组特定的设备上运行。运行将生成一个报告，其中包含有关运行结果的信息。一次运行包含一个或多个任务。

主题

运行配置

作为运行的一部分，您可以提供 Device Farm 可用于覆盖当前设备设置的设置。其中包括纬度和经度坐标、额外数据（包含在.zip 文件中）和辅助应用程序（应在要测试的应用程序之前安装的应用程序）。在 Android 上，可以更改一些其他设置，例如区域设置和无线电状态（蓝牙、GPS、NFC 和 Wi-Fi）。

运行文件保留

Device Farm 将您的应用程序和文件存储 30 天，然后从其系统中删除它们。不过，您可以随时删除您的文件。

Device Farm 将您的运行结果、日志和屏幕截图存储 400 天，然后从其系统中删除它们。

运行设备状态

Device Farm 总是在使设备可用于执行下一个任务之前先重启设备。

并行运行

Device Farm 在设备变得可用时并行运行测试。

设置执行超时

您可以设置一个值，以指定在让每个设备停止运行测试之前，应执行多长时间的测试运行。例如，如果您的测试需要每个设备花费 20 分钟，应为每个设备选择 30 分钟的超时值。

有关更多信息，请参阅 在 AWS Device Farm 中设置测试运行的执行超时。

运行中的广告

我们建议您从应用程序中删除广告，然后再将它们上传到 Device Farm。我们无法保证广告会在运行期间显示。

运行中的媒体

您可以提供媒体或其他数据来补充您的应用程序。附加的数据必须以 .zip 文件形式提供，并且大小不能超过 4 GB。

运行的常见任务

有关更多信息，请参阅在 Device Farm 中创建测试运行和AWS Device Farm 中的测试运行。

AWS Device Farm 中的应用程序

以下各部分提供了有关 Device Farm 中的应用程序行为的信息。

主题

- 分析应用程序
- 对运行中的应用程序重新签名
- 运行中难以辨认的应用程序

分析应用程序

您无需分析您的应用程序或为 Device Farm 提供您的应用程序的源代码。Android 应用程序无需修改即可提交。必须使用 iOS 设备目标而非模拟器构建 iOS 应用程序。

对运行中的应用程序重新签名

对于 iOS 应用程序，您无需向您的预置配置文件中添加任何 Device Farm UUID。Device Farm 会使用通配符配置文件替换嵌入式预置配置文件，然后重新签署该应用程序。如果您提供了辅助数据，则 Device Farm 会在 Device Farm 安装应用程序之前将辅助数据添加到应用程序的程序包中，以使辅助数据位于您的应用程序的沙盒中。重新签署应用程序会删除权限，例如应用程序组、关联域、游戏中心、HealthKit、HomeKit、无线配件配置、应用程序内购买、应用程序间音频、Apple Pay、推送通知以及 VPN 配置和控制。

对于 Android 应用程序，Device Farm 会对应用程序重新签名。这可能会破坏依赖应用程序签名的任何功能（如 Google Maps Android API），也可能会触发 DexGuard 之类的产品的防盗版或防窜改检测。

运行中难以辨认的应用程序

对于 Android 应用程序，如果应用程序难以辨认，您仍然可以使用 Device Farm 测试它 (如果您使用 ProGuard)。但是，如果您使用包含防盗版措施的 DexGuard，Device Farm 将无法对应用程序重新签名并运行测试。

AWS Device Farm 中的报告

以下部分提供有关 Device Farm 测试报告的信息。

主题

报告保留

Device Farm 将您的报告存储 400 天。这些报告包括元数据、日志、屏幕截图和性能数据。

报告组件

Device Farm 中的报告包含通过和失败信息、崩溃报告、测试和设备日志、屏幕截图以及性能数据。

报告包括详细的每个设备的数据以及概要结果，例如给定问题的发生次数。

报告中的日志

报告包括 Android 测试的完整 logcat 捕获和 iOS 测试的完整设备控制台日志。

报告的常见任务

有关更多信息，请参阅 在 Device Farm 中查看测试报告。

AWS Device Farm 中的会话

你可以使用 Device Farm 通过远程访问会话对安卓和 iOS 应用程序进行交互式测试。这包括在 Web 浏览器中进行手动交互以及从本地客户端对远程设备运行 Appium 测试。开发者可以在特定设备上重现其应用程序或Appium测试中的问题，以隔离和解决问题。

主题

支持远程访问的设备

Device Farm 为许多独特的常用 Android 和 iOS 设备提供支持。可用设备的列表随着新设备进入市场而扩大。Device Farm 控制台中显示了当前可用于远程访问的 Android 和 iOS 设备的列表。有关更多信息，请参阅 AWS Device Farm 中的设备支持。

会话文件保留

Device Farm 将您的应用程序和文件存储 30 天，然后从其系统中删除它们。不过，您可以随时删除您的文件。

Device Farm 会将您的会话日志和捕获的视频存储 400 天，然后从其系统中删除它们。

分析应用程序

您无需分析您的应用程序或为 Device Farm 提供您的应用程序的源代码。无需修改即可提交 Android 和 iOS 应用程序。

对会话中的应用程序重新签名

Device Farm 会对 Android 和 iOS 应用程序重新签名。这可能会破坏依赖应用程序签名的功能。例如，适用于 Android 的 Google Maps API 取决于您的应用程序的签名。应用程序重新签名还可能触发来自安卓设备等 DexGuard 产品的反盗版或防篡改检测。

会话中难以辨认的应用程序

对于 Android 应用程序，如果应用程序经过混淆处理，您仍然可以使用 Device Farm 对其进行测试（如果您使用）。 ProGuard但是，如果您使用 DexGuard 反盗版措施，Device Farm 将无法重新签署应用程序。

AWS Device Farm 中的项目

Device Farm 中的项目表示 Device Farm 中包含运行的逻辑工作区，一次运行用于单个应用程序在一个或多个设备上的每个测试。借助项目，您能够以您选择的任何方式组织工作区。例如，可以每个应用程序名称一个项目，也可以每个平台一个项目。您可以根据需要创建任意数量的项目。

您可以使用 AWS Device Farm 控制台、AWS Command Line Interface (AWS CLI) 或 AWS Device Farm API 来处理项目。

主题

在 AWS Device Farm 中创建项目

您可以使用 AWS Device Farm 控制台或 AWS De AWS CLI vice Farm API 创建项目。

先决条件

- 完成 设置 中的步骤。

创建项目（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 选择 New project (新项目)。
4. 输入项目的名称。（可选）您可以提供以下一个或多个参数，然后选择 “提交”。

   虚拟私有云 (VPC) 设置

   选择要应用于被测设备及其配对测试主机的 VPC、子网和安全组。只有私有设备才支持此功能。请参阅AWS Device Farm 中的 VPC-ENI了解更多信息。

   执行角色 ARN

   在自定义测试环境中由测试运行者担任的 IAM 角色。有关更多信息，请参阅 使用 IAM 执行角色访问 AWS 资源。

环境变量

要插入到测试执行运行器进程环境中的一个或多个变量。以 “DEVICEFARM_” 开头的变量名保留供服务使用。我们建议不要在这些环境变量中存储敏感值，而是建议在测试期间使用 IAM 执行角色从 AWS Secrets Manager 获取此类值。

创建项目 (AWS CLI)

- 运行 create-project 并指定项目名称。

  示例：

  ```
  aws devicefarm create-project --name MyProjectName
  ```

  AWS CLI 响应包括项目的亚马逊资源名称 (ARN)。

  ```
  {
      "project": {
          "name": "MyProjectName",
          "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
  c861-4c0a-b1d5-12345EXAMPLE",
          "created": 1535675814.414
      }
  }
  ```

  有关更多信息，请参阅create-project和AWS CLI 参考文档。

创建项目 (API)

- 调用 CreateProject API。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

在 AWS Device Farm 中查看项目列表

您可以使用 AWS Device Farm 控制台、AWS CLI 或 AWS Device Farm API 查看项目列表。

主题

先决条件

- 在 Device Farm 中至少创建一个项目。按照在 AWS Device Farm 中创建项目中的说明操作，然后返回此页。

查看项目列表（控制台）

1. 通过以下网址登录 Device Farm 控制台：https://console.aws.amazon.com/devicefarm。
2. 要查找可用项目列表，请执行以下操作：
   - 对于移动设备测试项目，在 Device Farm 导航菜单上，选择移动设备测试，然后选择项目。
   - 对于桌面浏览器测试项目，在 Device Farm 导航菜单上，选择桌面浏览器测试，然后选择项目。

查看项目列表 (AWS CLI)

- 要查看项目列表，请运行 list-projects 命令。

  要查看有关单个项目的信息，请运行 get-project 命令。

有关将 AWS CLI 与 Device Farm 结合使用的一般信息，请参阅 AWS CLI 参考文档。

查看项目列表 (API)

- 要查看项目列表，请调用 `ListProjects` API。

  要查看有关单个项目的信息，请调用 `GetProject` API。

有关 AWS Device Farm API 的更多信息，请参阅 自动化 Device Farm。

AWS Device Farm 中的测试运行

Device Farm 中的运行代表您的应用程序的特定版本，使用一组特定的测试，在一组特定的设备上运行。运行将生成一个报告，其中包含有关运行结果的信息。一次运行包含一个或多个任务。有关更多信息，请参阅 运行。

您可以使用 AWS Device Farm 控制台 AWS Command Line Interface (AWS CLI) 或 AWS Device Farm API 进行测试运行。

主题

在 Device Farm 中创建测试运行

你可以使用 Device Farm 控制台或 Device Farm API 来创建测试运行。 AWS CLI您也可以使用支持的插件，如适用于 Device Farm 的 Jenkins 或 Gradle 插件。有关插件的更多信息，请参阅 工具和插件。有关运行的信息，请参阅运行。

主题

先决条件

您在 Device Farm 中必须有一个项目。按照在 AWS Device Farm 中创建项目中的说明操作，然后返回此页。

创建测试运行（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在导航窗格中，选择移动设备测试，然后选择项目。
3. 如果您已具有项目，可以将您的测试上传到现有项目。否则，选择新建项目，输入项目名称，然后选择创建。
4. 打开您的项目，然后选择创建运行。
5. （可选）在运行设置下的运行名称部分中，输入您的运行名称。如果未提供名称，则默认情况下，Device Farm 控制台会将您的运行命名为“My Device Farm run”。
6. （可选）在运行设置下的作业超时部分中，您可以为测试运行指定执行超时。如果您使用的是无限制测试槽位，请确认在计费方法下选择非计量。
7. 在运行设置下的运行类型部分中，选择您的运行类型。如果您没有可供测试的应用程序，或者您测试的是 Android（.apk）应用程序，请选择 Android 应用程序。如果您测试的是 iOS（.ipa）应用程序，请选择 iOS 应用程序。如果要测试 Web 应用程序，请选择 Web 应用程序。
8. 如果您没有可供测试的应用程序，请在选择应用程序下的应用程序选择选项部分中，选择选择 Device Farm 提供的示例应用程序。如果您自带应用程序，请选择上传自己的应用程序，然后选择您的应用程序文件。如果您要上传 iOS 应用程序，请确保选择 iOS device (iOS 设备)，而不是模拟器。
9. 在配置测试下，选择一个可用的测试框架。

   > **Note**
   >
   > 如果您没有任何可用的测试，请选择 Built-in: Fuzz (内置: 模糊) 来运行标准的内置测试套件。如果您选择了内置：模糊并且出现了事件计数、事件节流和随机掩码种子框，则可以更改或保留值。

   有关可用测试套件的信息，请参阅AWS Device Farm 中的测试框架和内置测试。
10. 如果您没有选择内置：模糊，请在选择测试程序包下选择选择文件。浏览到并选择包含您的测试的文件。

11. 对于您的执行环境，请选择在标准环境中运行测试或自定义您的测试环境。有关更多信息，请参阅 AWS Device Farm 中的测试环境。
12. 如果您使用的是自定义测试环境，则可以选择执行以下操作：
    - 如果您要编辑自定义测试环境中的默认测试规范，请选择 Edit (编辑) 以更新默认 YAML 规范。
    - 如果您更改了测试规范，请选择另存为新版本以更新测试规范。
    - 您可以配置环境变量。此处提供的变量将优先于父项目上可能配置的任何变量。
13. 在选定设备下，执行以下操作之一：
    - 要选择内置设备池以对其运行测试，请为 Device pool (设备池) 选择 Top Devices (主要设备)。
    - 要创建您自己的设备池以对其运行测试，请按照创建设备池中的说明操作，然后返回到此页面。
    - 如果您在前面创建了自己的设备池，请为 Device pool (设备池) 选择您的设备池。
    - 选择手动选择设备，然后选择要针对其运行的所需设备。此配置将不会保存。

    有关更多信息，请参阅 AWS Device Farm 中的设备支持。
14. （可选）要添加其他配置，请打开其他配置下拉列表。在此部分，您可以执行以下任意操作：
    - 要提供执行角色 ARN 或覆盖在父项目上配置的执行角色 ARN，请使用执行角色 ARN 字段。
    - 要为 Device Farm 提供其他将在运行期间使用的数据，请在添加额外数据旁边选择选择文件，然后浏览到并选择包含这些数据的 .zip 文件。
    - 要安装 Device Farm 将在运行期间使用的其他应用程序，请在安装其他应用程序旁边选择选择文件，然后浏览到并选择包含该应用程序的 .apk 或 .ipa 文件。为您要安装的其他应用程序重复此操作。在上传应用程序之后，您可以拖放应用程序来更改应用程序的安装顺序。
    - 要指定是否将在运行期间启用 Wi-Fi、蓝牙、GPS 或 NFC，请在 Set radio states (设置电台) 旁边选中相应框。
    - 要为运行预设设备纬度和经度，请在 Device location (设备位置) 旁边输入坐标。
    - 要为运行预设设备区域设置，请在设备区域设置中选择区域设置。
    - 选择启用视频记录以在测试期间启用视频记录。
    - 选择启用应用程序性能数据捕获以启用从设备捕获性能数据。

> **Note**
>
> 目前，设置设备无线电状态和区域设置选项仅适用于 Android 本地测试。

> **Note**
>
> 如果您有私有设备，还将显示特定于私有设备的配置。

15. 在页面底部，选择创建运行以安排运行。

Device Farm 将在设备可用后立即启动运行，通常在几分钟内启动。在测试运行期间，Device Farm 控制台会在运行表中显示一个待处理图标 。运行中的每台设备也将以待处理图标开始，然后在测试开始时切换到正在运行的图标 。每次测试完成后，设备名称旁边都会显示一个测试结果图标。完成所有测试后，运行旁边的待处理图标将变为测试结果图标。

如果您想停止测试运行，请参阅 在 AWS Device Farm 中停止运行。

创建测试运行 (AWS CLI)

您可以使用 AWS CLI 来创建测试运行。

主题

- 步骤 1：选择一个项目
- 步骤 2：选择一个设备池
- 步骤 3：上传您的应用程序文件
- 步骤 4：上传您的测试脚本程序包
- 步骤 5：上传您的自定义测试规范（可选）
- 步骤 6：安排测试运行

步骤 1：选择一个项目

您必须将您的测试运行与一个 Device Farm 项目关联。

1. 要列出您的 Device Farm 项目，请运行 list-projects。如果您没有项目，请参阅在 AWS Device Farm 中创建项目。

示例：

```
aws devicefarm list-projects
```

响应中将包含您的 Device Farm 项目的列表。

```
{
    "projects": [
        {
            "name": "MyProject",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
            "created": 1503612890.057
        }
    ]
}
```

2. 选择一个项目以与您的测试运行关联，请记录它的 Amazon 资源名称（ARN）。

步骤 2：选择一个设备池

您必须选择一个设备池以与您的测试运行关联。

1. 要查看您的设备池，请运行 list-device-pools 并指定您的项目 ARN。

示例：

```
aws devicefarm list-device-pools --arn arn:MyProjectARN
```

响应中将包含内置的 Device Farm 设备池，如 Top Devices，以及之前为此项目创建的任何设备池：

```
{
    "devicePools": [
        {
            "rules": [
                {
                    "attribute": "ARN",
                    "operator": "IN",
```

```
                    "value": "[\"arn:aws:devicefarm:us-west-2::device:example1\",
\"arn:aws:devicefarm:us-west-2::device:example2\",\"arn:aws:devicefarm:us-
west-2::device:example3\"]"
                }
            ],
            "type": "CURATED",
            "name": "Top Devices",
            "arn": "arn:aws:devicefarm:us-west-2::devicepool:example",
            "description": "Top devices"
        },
        {
            "rules": [
                {
                    "attribute": "PLATFORM",
                    "operator": "EQUALS",
                    "value": "\"ANDROID\""
                }
            ],
            "type": "PRIVATE",
            "name": "MyAndroidDevices",
            "arn": "arn:aws:devicefarm:us-west-2:605403973111:devicepool:example2"
        }
    ]
}
```

2. 选择一个设备池，然后记录它的 ARN。

   您也可以创建一个设备池，然后返回到此步骤。有关更多信息，请参阅 创建设备池 (AWS CLI)。

步骤 3：上传您的应用程序文件

要创建上传请求并获取 Amazon Simple Storage Service (Amazon S3) 预签名的上传 URL，您需要：

- 您的项目 ARN。
- 您的应用程序文件的名称。
- 上传的类型。

有关更多信息，请参阅 create-upload。

1. 要上传文件，请使用 `--project-arn`、`--name` 和 `--type` 参数运行 create-upload。

此示例将创建一个用于 Android 应用程序的上传：

```
aws devicefarm create-upload --project-arn arn:MyProjectArn --name MyAndroid.apk --
type ANDROID_APP
```

响应中将包含您的应用程序上传 ARN 和预签名 URL。

```
{
    "upload": {
        "status": "INITIALIZED",
        "name": "MyAndroid.apk",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    }
}
```

2. 记录应用程序上传 ARN 和预签名 URL。
3. 使用 Amazon S3 预签名 URL 上传您的应用程序文件。本示例使用 curl 上传 Android .apk 文件：

```
curl -T MyAndroid.apk "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

有关更多信息，请参阅 Amazon 简单存储服务用户指南 URLs中的使用预签名上传对象。

4. 要检查您的应用程序上传的状态，请运行 get-upload 并指定应用程序上传的 ARN。

```
aws devicefarm get-upload --arn arn:MyAppUploadARN
```

等到响应中的状态为 SUCCEEDED 之后，再上传您的测试脚本程序包。

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyAndroid.apk",
        "created": 1535732625.964,
```

```
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

步骤 4：上传您的测试脚本程序包

接下来，您将上传测试脚本程序包。

1. 要创建您的上传请求并获取 Amazon S3 预签名上传 URL，请使用 `--project-arn`、`--name` 和 `--type` 参数运行 create-upload。

   此示例将创建一个 Appium Java TestNG 测试程序包上传：

   ```
   aws devicefarm create-upload --project-arn arn:MyProjectARN --name MyTests.zip --
   type APPIUM_JAVA_TESTNG_TEST_PACKAGE
   ```

   响应中将包含您的测试程序包上传 ARN 和预签名 URL。

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "name": "MyTests.zip",
           "created": 1535738627.195,
           "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL",
           "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
           "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
   c861-4c0a-b1d5-12345EXAMPLE"
       }
   }
   ```

2. 记录测试程序包上传 ARN 和预签名 URL。

3. 使用 Amazon S3 预签名 URL 上传您的测试脚本程序包文件。此示例使用 curl 上传压缩 Appium TestNG 脚本文件：

```
curl -T MyTests.zip "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

4. 要检查您的测试脚本程序包上传的状态，请运行 get-upload 并指定从步骤 1 获取的测试程序包上传的 ARN。

```
aws devicefarm get-upload --arn arn:MyTestsUploadARN
```

等到响应中的状态为 SUCCEEDED 之后，再继续执行下一个可选步骤。

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTests.zip",
        "created": 1535738627.195,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

步骤 5：上传您的自定义测试规范（可选）

如果您要在标准测试环境中运行测试，请跳过此步骤。

Device Farm 会为每个支持的测试类型维护默认测试规范文件。接下来，您将下载默认测试规范，并使用它来创建自定义测试规范上传，以在自定义测试环境中运行测试。有关更多信息，请参阅 AWS Device Farm 中的测试环境。

1. 要查找您的默认测试规范的上传 ARN，请运行 list-uploads 并指定项目 ARN。

```
aws devicefarm list-uploads --arn arn:MyProjectARN
```

响应中包含每个默认测试规范的条目：

```
{
```

```
    "uploads": [
        {

            {
                "status": "SUCCEEDED",
                "name": "Default TestSpec for Android Appium Java TestNG",
                "created": 1529498177.474,
                "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
                "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
                "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
            }
        }
    ]
}
```

2. 从列表中选择您的默认测试规范。记录它的上传 ARN。
3. 要下载默认测试规范，请运行 get-upload 并指定上传 ARN。

   示例：

   ```
   aws devicefarm get-upload --arn arn:MyDefaultTestSpecARN
   ```

   响应中将包含预签名 URL，您可以从中下载默认测试规范。
4. 此示例使用 curl 下载默认测试规范，并将其另存为 `MyTestSpec.yml`：

   ```
   curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL" >
    MyTestSpec.yml
   ```
5. 您可以根据自己的测试要求编辑默认的测试规范，然后在未来的测试运行中使用修改后的测试规范。跳过此步骤可在自定义测试环境中按原样使用默认的测试规范。
6. 要创建您的自定义测试规范上传，请运行 create-upload 并指定您的测试规范名称、测试规范类型和项目 ARN。

   此示例将创建 Appium Java TestNG 自定义测试程序包的上传：

   ```
   aws devicefarm create-upload --name MyTestSpec.yml --type
    APPIUM_JAVA_TESTNG_TEST_SPEC --project-arn arn:MyProjectARN
   ```

响应中将包含测试规范上传 ARN 和预签名 URL：

```
{
    "upload": {
        "status": "INITIALIZED",
        "category": "PRIVATE",
        "name": "MyTestSpec.yml",
        "created": 1535751101.221,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    }
}
```

7. 记录测试规范上传 ARN 和预签名 URL。
8. 使用 Amazon S3 预签名 URL 上传您的测试规范文件。此示例用于curl上传 Appium JavaTest NG 测试规范：

```
curl -T MyTestSpec.yml "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

9. 要检查您的测试规范上传的状态，请运行 get-upload 并指定上传 ARN。

```
aws devicefarm get-upload --arn arn:MyTestSpecUploadARN
```

等到响应中的状态为 SUCCEEDED 之后，再安排您的测试运行。

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTestSpec.yml",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
```

```
}
```

要更新您的自定义测试规范，请运行 update-upload 并指定测试规范的上传 ARN。有关更多信息，请参阅 update-upload。

步骤 6：安排测试运行

要安排测试运行，请运行 AWS CLIschedule-run，请指定：

- 从步骤 1 获取的项目 ARN。
- 从步骤 2 获取的设备池 ARN。
- 从步骤 3 获取的应用程序上传 ARN。
- 从步骤 4 获取的测试程序包上传 ARN。

如果您要在自定义测试环境中运行测试，则还需要从步骤 5 获取的测试规范 ARN。

安排在标准测试环境中执行运行

- 运行 schedule-run，并指定您的项目 ARN、设备池 ARN、应用程序上传 ARN 和测试程序包信息。

  示例：

```
aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
test type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPackageARN
```

  响应中将包含可用于检查测试运行状态的运行 ARN。

```
{
    "run": {
        "status": "SCHEDULING",
        "appUpload": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345appEXAMPLE",
        "name": "MyTestRun",
        "radios": {
            "gps": true,
            "wifi": true,
            "nfc": true,
```

```
            "bluetooth": true
        },
        "created": 1535756712.946,
        "totalJobs": 179,
        "completedJobs": 0,
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "devicePoolArn": "arn:aws:devicefarm:us-
west-2:123456789101:devicepool:5e01a8c7-c861-4c0a-b1d5-12345devicepoolEXAMPLE",
        "jobTimeoutMinutes": 150,
        "billingMethod": "METERED",
        "type": "APPIUM_JAVA_TESTNG",
        "testSpecArn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345specEXAMPLE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:run:5e01a8c7-c861-4c0a-
b1d5-12345runEXAMPLE",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
            "errored": 0,
            "total": 0
        }
    }
}
```

有关更多信息，请参阅 schedule-run。

安排在自定义测试环境中执行运行

- 所用的步骤几乎与标准测试环境中的步骤相同，只是 `--test` 参数中多了一个 `testSpecArn` 属性。

  示例：

```
aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
test
 testSpecArn=arn:MyTestSpecUploadARN,type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPacka
```

检查您的测试运行的状态

- 使用 get-run 命令并指定运行 ARN：

```
aws devicefarm get-run --arn arn:aws:devicefarm:us-
west-2:111122223333:run:5e01a8c7-c861-4c0a-b1d5-12345runEXAMPLE
```

有关更多信息，请参阅 get-run。有关将 Device Farm 与配合使用的信息 AWS CLI，请参阅AWS CLI 参考文档。

创建测试运行 (API)

这些步骤与本 AWS CLI 节中描述的步骤相同。请参阅创建测试运行 (AWS CLI)。

您调用 ScheduleRun API 时需要以下信息：

- 项目 ARN。请参阅创建项目 (API)和CreateProject。
- 应用程序上传 ARN。请参阅CreateUpload。
- 测试程序包上传 ARN。请参阅CreateUpload。
- 设备池 ARN。请参阅创建设备池和CreateDevicePool。

> **Note**
>
> 如果您要在自定义测试环境中运行测试，则还需要测试规范上传 ARN。有关更多信息，请参阅步骤 5：上传您的自定义测试规范（可选）和CreateUpload。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

后续步骤

在 Device Farm 控制台中，运行完成后，时钟图标

变为结果图标，例如成功

。

测试完成后，将立即显示与运行对应的报告。有关更多信息，请参阅 AWS Device Farm 中的报告。

要使用报告，请按照在 Device Farm 中查看测试报告中的说明操作。

在 AWS Device Farm 中设置测试运行的执行超时

您可以设置一个值，以指定在让每个设备停止运行测试之前，应执行多长时间的测试运行。每个设备的默认执行超时值为 150 分钟，但您可以将该值设置为最短 5 分钟。您可以使用 AWS Device Farm 控制台或 AWS Device Farm API 来设置执行超时。 AWS CLI

> **Important**
>
> 执行超时值选项应设置为测试运行的最大持续时间，另加一些缓冲时间。例如，如果您的测试需要每个设备花费 20 分钟，应为每个设备选择 30 分钟的超时值。

如果执行超出您的超时值，该设备上的执行将被强制停止。如有可能，将提供部分结果。如果您使用的是计量计费选项，将会对该时刻为止的执行进行计费。有关定价的更多信息，请参阅 Device Farm 定价。

如果您知道对每个设备执行测试运行应花费多长时间，建议您使用此功能。如果指定测试运行的执行超时值，则可避免测试运行出于某种原因而堵塞，并避免系统在未执行任何测试时对设备按分钟计费。换句话说，如果测试运行所花费的时间比预期长，您可以使用执行超时值功能停止该运行。

您可以在项目级别和测试运行级别设置执行超时值。

先决条件

1. 完成 设置 中的步骤。
2. 在 Device Farm 中创建项目。按照在 AWS Device Farm 中创建项目中的说明操作，然后返回此页。

设置项目的执行超时值

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 如果您已有项目，请从列表中选择一个项目。否则，请选择新建项目，输入项目的名称，然后选择提交。
4. 选择 Project settings (项目设置)。

5. 在 General (常规) 选项卡上，对于 Execution timeout (执行超时)，请输入值或使用滑块条。
6. 选择保存。

   现在，您项目中的所有测试运行都将使用您指定的执行超时值，除非您在安排运行时覆盖该超时值。

设置测试运行的执行超时值

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 如果您已有项目，请从列表中选择一个项目。否则，请选择新建项目，输入项目的名称，然后选择提交。
4. 选择 Create a new run (创建新运行)。
5. 按照相应步骤选择一个应用程序，配置您的测试，选择您的设备，并指定设备状态。
6. 在检查并启动运行 上，对于设置执行超时，请输入值或使用滑块条。
7. 选择 Confirm and start run (确认并启动运行)。

为您的 AWS Device Farm 运行模拟网络连接和条件

在 Device Farm 中测试 Android、iOS 和网络应用程序时，你可以使用网络整形来模拟网络连接和条件。例如，您可以模拟有损或间歇性的互联网连接。

在您使用默认网络设置创建运行时，每个设备都能通过完整无阻碍的 WiFi 与 Internet 建立连接。使用网络整形时，您可以更改 Wi-Fi 连接以指定网络配置文件，例如 3G 或 Lossy WiFi，用于控制入站和出站流量的吞吐量、延迟、抖动和丢失。

主题

- 在安排测试运行时设置网络塑造
- 创建网络配置文件
- 在测试过程中更改网络条件

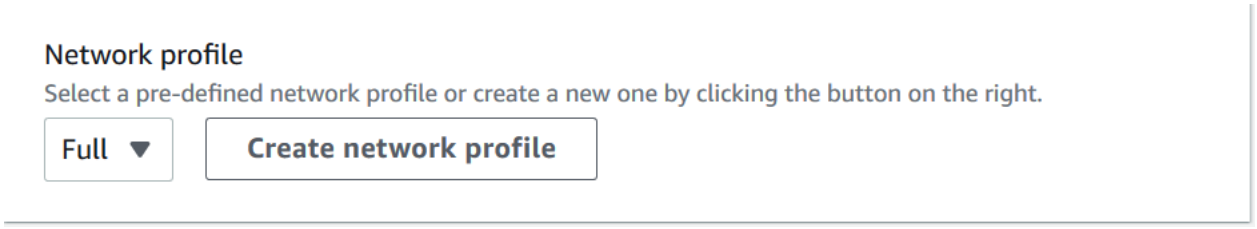
在安排测试运行时设置网络塑造

在安排运行时，您可以选择任何 Device Farm 精选配置文件，也可以创建并管理自己的配置文件。

1. 从任何 Device Farm 项目中，选择创建新运行。

   如果您还没有项目，请参阅在 AWS Device Farm 中创建项目。

2. 选择您的应用程序，然后选择下一步。
3. 配置您的测试，然后选择下一步。
4. 选择您的设备，然后选择下一步。
5. 在位置和网络设置部分，选择网络配置文件或选择创建网络配置文件来创建自己的网络配置文件。

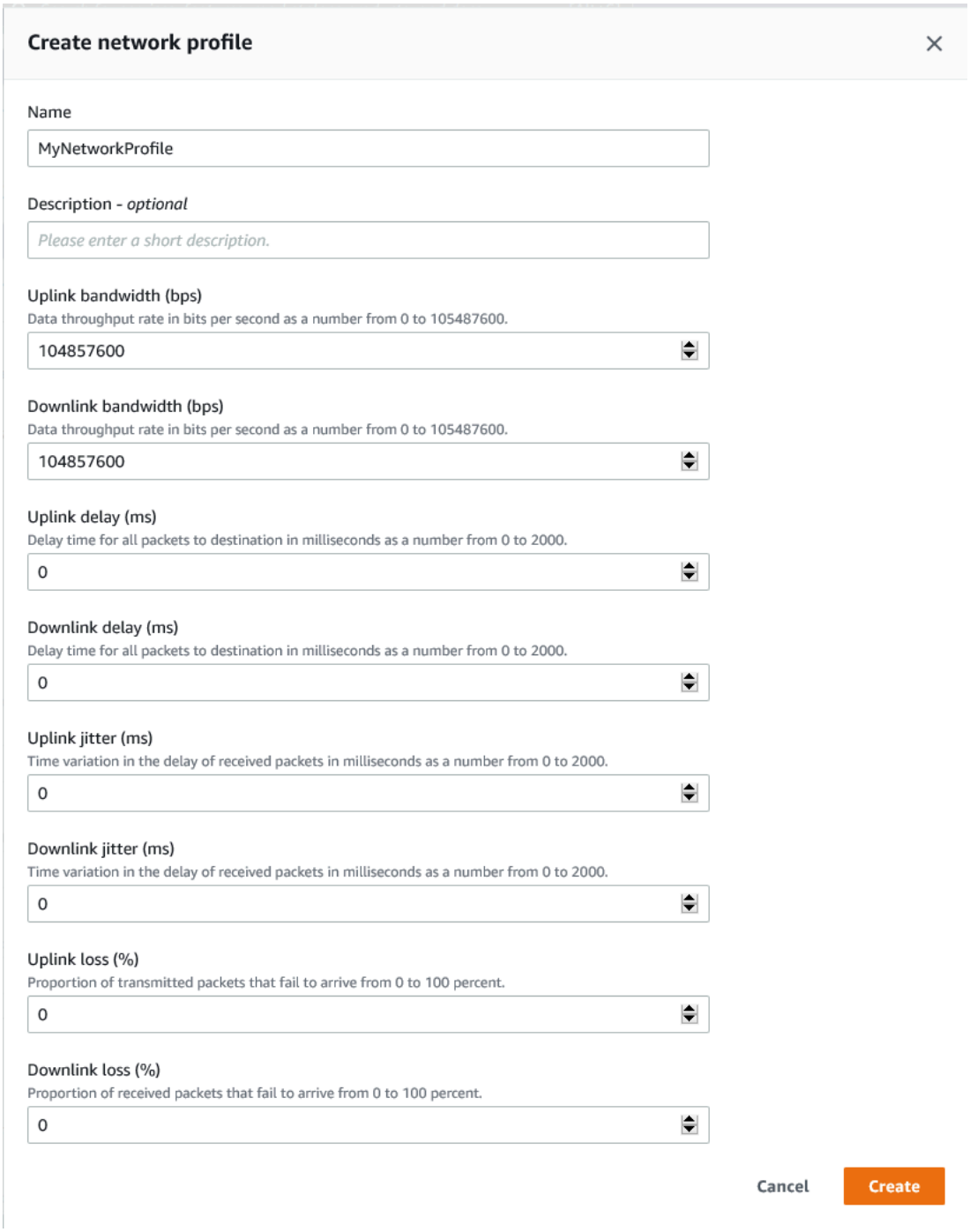


6. 选择下一步。
7. 检查并启动您的测试运行。

创建网络配置文件

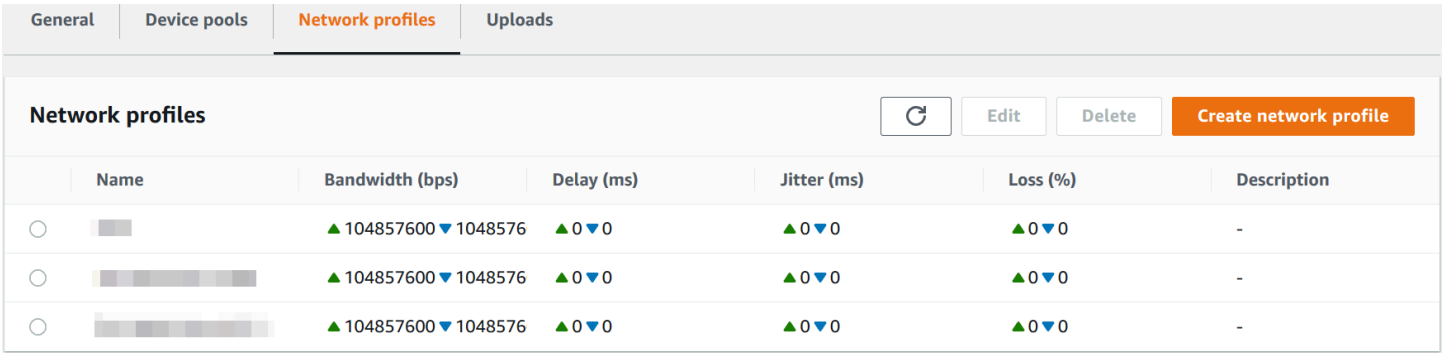
在您创建测试运行时，可以创建网络配置文件。

1. 选择创建新的网络配置文件。



2. 为您的网络配置文件输入名称和设置。
3. 选择创建。
4. 完成测试运行的创建工作并启动运行。

创建网络配置文件后，可在 Project settings (项目设置) 页面上查看和管理它。



在测试过程中更改网络条件

您可以使用 Appium 等框架从您的设备主机调用 API，以模拟动态网络条件，例如在测试运行期间减少带宽。有关更多信息，请参阅 CreateNetworkProfile。

在 AWS Device Farm 中停止运行

您可能希望在启动运行后将其停止。例如，您可能在测试正在运行时注意到一个问题，并希望使用更新的测试脚本重启运行。

您可以使用 Device Farm 控制台或 API 来停止运行。 AWS CLI

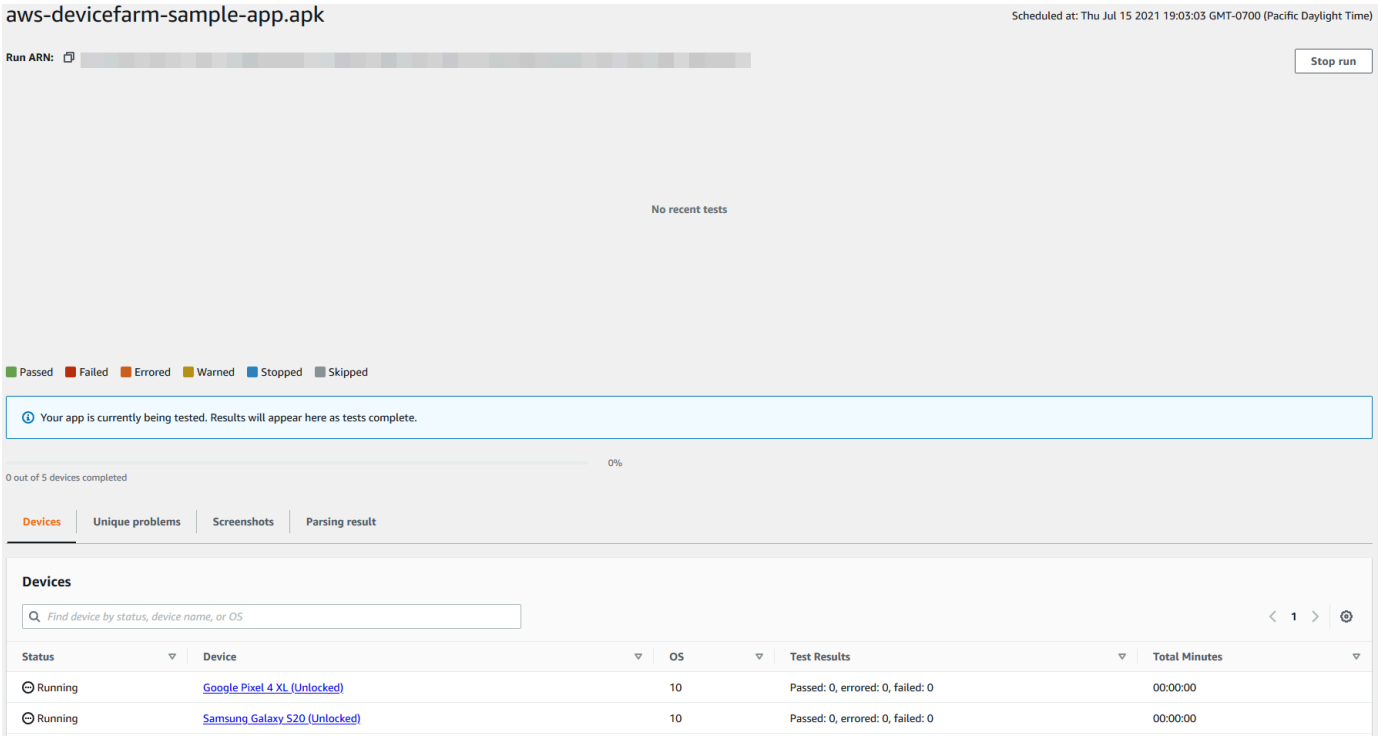
主题

- 停止运行（控制台）
- 停止运行 (AWS CLI)
- 停止运行 (API)

停止运行（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 选择其中具有主动测试运行的项目。
4. 在自动化测试页面上，选择测试运行。

   设备名称的左侧应当显示待处理或正在运行的图标。

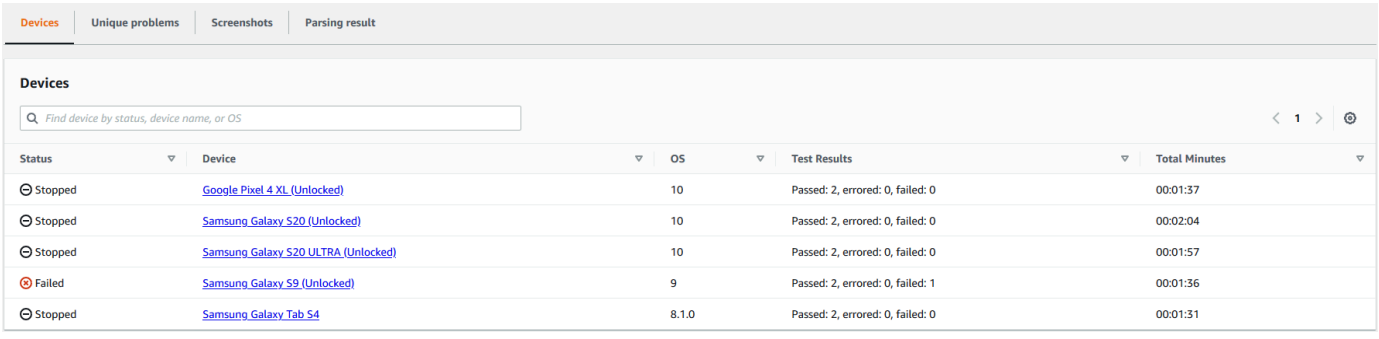


5. 选择 Stop run (停止运行)。

很快，设备名称旁边会出现一个带有红色圆圈（内有减号）的图标。运行停止后，图标颜色会从红色变为黑色。

> **Important**
>
> 如果测试已完成，则 Device Farm 无法停止测试。如果测试正在进行，Device Farm 会停止测试。总分钟数（您会为此付费）显示在 设备 部分。此外，您还需要为 Device Farm 运行安装套件和停用套件所花的总分钟数付费。有关更多信息，请参阅 Device Farm 定价。

下图显示了成功停止测试运行后的示例 Devices (设备) 部分。



停止运行 (AWS CLI)

您可以运行以下命令来停止指定的测试运行，其中*myARN*是测试运行的 Amazon 资源名称 (ARN)。

```
$ aws devicefarm stop-run --arn myARN
```

您应该可以看到类似于如下所示的输出内容：

```
{
    "run": {
        "status": "STOPPING",
        "name": "Name of your run",
        "created": 1458329687.951,
        "totalJobs": 7,
        "completedJobs": 5,
        "deviceMinutes": {
            "unmetered": 0.0,
            "total": 0.0,
            "metered": 0.0
        },
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "billingMethod": "METERED",
        "type": "BUILTIN_EXPLORER",
        "arn": "myARN",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
            "errored": 0,
            "total": 0
        }
    }
}
```

要获得您的运行的 ARN，请使用 `list-runs` 命令。此输出应当类似于如下所示：

```
{
    "runs": [
        {
```

```
            "status": "RUNNING",
            "name": "Name of your run",
            "created": 1458329687.951,
            "totalJobs": 7,
            "completedJobs": 5,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 0.0,
                "metered": 0.0
            },
            "platform": "ANDROID_APP",
            "result": "PENDING",
            "billingMethod": "METERED",
            "type": "BUILTIN_EXPLORER",
            "arn": "Your ARN will be here",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 0,
                "errored": 0,
                "total": 0
            }
        }
    ]
}
```


有关将 Device Farm 与配合使用的信息 AWS CLI，请参阅AWS CLI 参考文档。

停止运行 (API)

- 将StopRun操作调用到测试运行。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

查看 AWS Device Farm 中的运行列表

您可以使用 Device Farm 控制台或 API 来查看项目的运行列表。 AWS CLI

主题

查看运行列表（控制台）

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 在项目列表中，选择与要查看的列表相对应的项目。

> **Tip**
> 您可以使用搜索栏按名称筛选项目列表。

查看运行列表 (AWS CLI)

- 运行 list-runs 命令。

  要查看有关单次运行的信息，请运行 get-run 命令。

有关将 Device Farm 与配合使用的信息 AWS CLI，请参阅AWS CLI 参考文档。

查看运行列表 (API)

- 调用 `ListRuns` API。

  要查看有关单次运行的信息，请调用 `GetRun` API。

有关 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

在 AWS Device Farm 中创建设备池

您可以使用 Device Farm 控制台或 API 来创建设备池。 AWS CLI

主题

先决条件

- 在 Device Farm 控制台中创建一个运行。按照在 Device Farm 中创建测试运行中的说明进行操作。在到达 Select devices (选择设备) 页面时，继续按照本节中的说明操作。

创建设备池（控制台）

1. 在项目页面，选择您的项目。在项目详细信息页面，选择项目设置。在设备池选项卡中，选择创建设备池。
2. 对于 Name (名称)，输入一个可轻松识别此设备池的名称。
3. 对于 Description (说明)，输入可轻松识别此设备池的说明。
4. 如果您希望对此设备池中的设备使用一个或多个选择标准，请执行以下操作：
   a. 选择创建动态设备池。
   b. 选择添加规则。
   c. 对于字段（第一个下拉列表），选择以下选项之一：
      - 要按制造商名称包含设备，请选择设备制造商。
      - 要按外形规格（平板电脑或手机）包含设备，请选择外形规格。
      - 要根据负载按可用性状态包含设备，请选择可用性。
      - 要仅包含公有或私有设备，请选择实例集类型。
      - 要按操作系统包含设备，请选择平台。
      - 有些设备有关于该设备的附加标签或描述。您可以通过选择实例标签来根据设备的标签内容查找设备。
      - 要按操作系统版本包含设备，请选择操作系统版本。
      - 要按型号包含设备，请选择型号。

d. 对于运算符（第二个下拉列表），请根据查询选择逻辑运算（EQUALS、CONTAINS 等）以包含设备。例如，您可以选择*Availability EQUALS AVAILABLE*包括当前处于该Available状态的设备。

e. 对于值（第三个下拉列表），输入或选择要为字段和运算符值指定的值。根据您的字段选择，值会受到限制。例如，如果您为字段选择了平台，则可用的选项只有 ANDROID 和 IOS。类似地，如果您为字段选择了外形规格，则可用的选项只有 PHONE 和 TABLET。

f. 要添加其他规则，请选择添加规则。

在创建第一条规则后，设备列表中与该规则匹配的每个设备旁边的框将会被选中。在您创建或更改规则后，设备列表中与这些组合规则匹配的每个设备旁边的框将会被选中。具有已选中框的设备将包括在设备池中。具有已清除框的设备被排除在设备池外。

g. 在最大设备数量下，输入要在设备池中使用的设备数量。如果您未输入设备的最大数量，Device Farm 将选择实例集中与您创建的规则相匹配的所有设备。为避免产生额外收费，请将此数字设置为与您的实际并行执行和设备种类要求相匹配的数量。

h. 要删除规则，请选择删除规则。

5. 如果要手动包含或排除单个设备，请执行以下操作：

a. 选择创建静态设备池。

b. 选中或清除每台设备旁边的复选框。仅当您没有指定任何规则时，才可以选中或清除框。

6. 如果您要包括或排除所有显示的设备，请选中或清除列表的列标题行中的框。如果您只想查看私有设备实例，请选择仅查看私有设备实例。

> **Important**
>
> 虽然您可以使用列标题行中的框来更改显示的设备列表，但这并不表示剩余的显示设备只是包括或排除的设备。要确认将包括或排除哪些设备，请确保清除列标题行中的所有框的内容，然后浏览各个框。

7. 选择创建。

创建设备池 (AWS CLI)

> **Tip**
>
> 如果您未输入设备的最大数量，Device Farm 将选择实例集中与您创建的规则相匹配的所有设备。为避免产生额外收费，请将此数字设置为与您的实际并行执行和设备种类要求相匹配的数量。

- 运行 create-device-pool 命令。

有关将 Device Farm 与配合使用的信息 AWS CLI，请参阅AWS CLI 参考文档。

创建设备池 (API)

> **Tip**
>
> 如果您未输入设备的最大数量，Device Farm 将选择实例集中与您创建的规则相匹配的所有设备。为避免产生额外收费，请将此数字设置为与您的实际并行执行和设备种类要求相匹配的数量。

- 调用 `CreateDevicePool` API。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

在 AWS Device Farm 中分析测试结果

在标准测试环境中，您可以使用 Device Farm 控制台查看测试运行中每个测试的报告。查看报告可帮助您了解哪些测试通过或失败，并为您提供有关应用程序在不同设备配置下的性能和行为的详细信息。

Device Farm 还会收集在您完成测试运行时可以下载的其他构件，例如文件、日志和图像。这些信息可以帮助您分析您的应用程序在真实设备上的表现，识别问题或错误，并诊断问题。

主题

- 在 Device Farm 中查看测试报告

- 在 Device Farm 中下载构件

在 Device Farm 中查看测试报告

使用 Device Farm 控制台查看测试报告。有关更多信息，请参阅 AWS Device Farm 中的报告。

主题

- 先决条件
- 查看报告
- Device Farm 测试结果状态

先决条件

设置测试运行并验证它是否已完成。

1. 要创建运行，请按照在 Device Farm 中创建测试运行中的说明操作，然后返回到此页面。
2. 验证运行是否已完成。在测试运行期间，Device Farm 控制台会显示正在进行的运行的待处理图标

   。

   运行中的每台设备也将以待处理图标开始，然后在测试开始时切换到正在运行的

   图标。每次测试完成后，设备名称旁边都会显示一个测试结果图标。完成所有测试后，运行旁边的待处理图标将变为测试结果图标。有关更多信息，请参阅 Device Farm 测试结果状态。

查看报告

您可以在 Device Farm 控制台中查看测试结果。

主题

- 查看测试运行摘要页面
- 查看唯一问题报告
- 查看设备报告。
- 查看测试套件报告
- 查看测试报告
- 查看报告中的问题、设备、套件或测试的日志信息

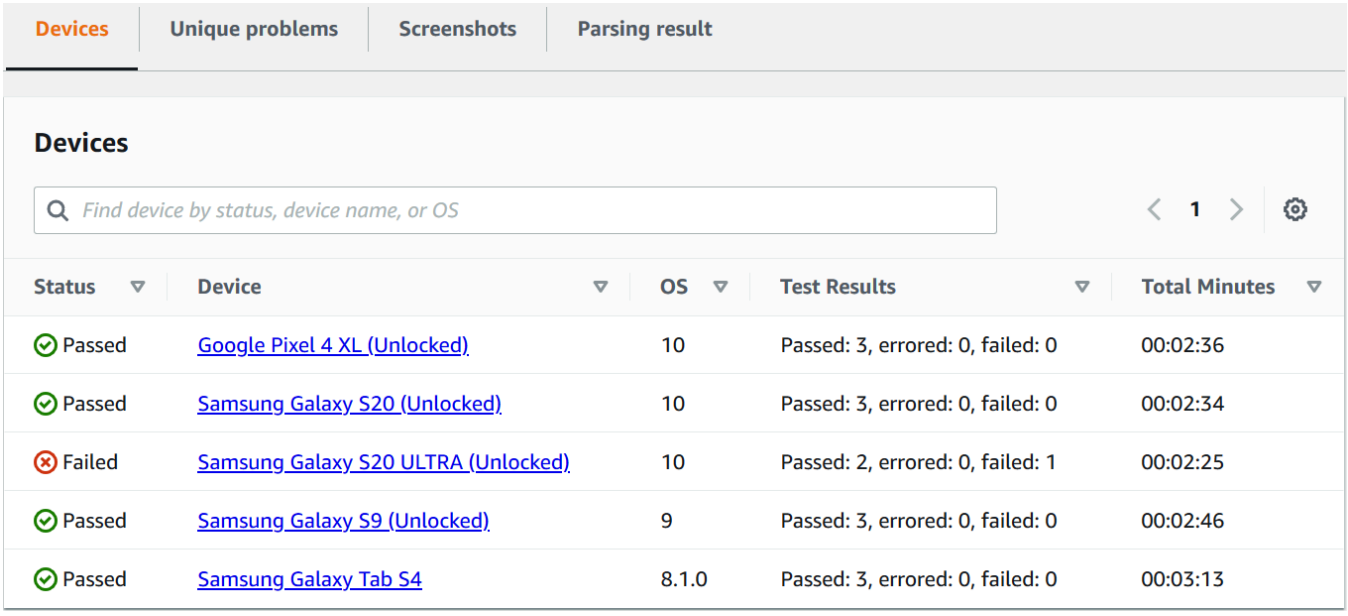
查看测试运行摘要页面

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在导航窗格中，选择移动设备测试，然后选择项目。
3. 从项目列表中选择用于运行的项目。

> **Tip**
>
> 使用搜索栏按名称筛选项目列表。

4. 选择已完成的运行以查看其摘要报告页面。
5. 测试运行摘要页面会显示测试结果的概览。

   - Unique problems (唯一问题) 部分列出唯一的警告和故障。要查看具有唯一性的问题，请按照查看唯一问题报告中的说明操作。
   - Devices (设备) 部分按结果显示每个设备的测试总数。



   在此示例中，有多个设备。在第一个表格条目中，运行 Android 版本 10 的 Google Pixel 4 XL 设备报告了三次成功的测试，运行时间为 02:36 分钟。

   要按设备查看结果，请按照查看设备报告。中的说明操作。

   - 屏幕截图部分显示 Device Farm 在运行期间捕获的任何屏幕截图的列表（按设备分组）。
   - 在解析结果部分，您可以下载解析结果。

查看唯一问题报告

1. 在 Unique problems (唯一问题) 中，选择要查看的问题。
2. 选择设备。报告显示有关该问题的信息。

   Video (视频) 部分显示该测试的可下载视频记录。

   结果部分显示测试结果。状态以结果图标表示。有关更多信息，请参阅 单个测试的状态。

   日志部分显示 Device Farm 在测试期间记录的任何信息。要查看此信息，请按照查看报告中的问题、设备、套件或测试的日志信息中的说明操作。

   文件选项卡显示该测试的任何可下载关联文件（如日志文件）的列表。要下载文件，请在列表中选择该文件的链接。

   屏幕截图选项卡显示 Device Farm 在测试期间捕获的任何屏幕截图的列表。

查看设备报告。

- 在 Devices (设备) 部分，选择设备。

  Video (视频) 部分显示该测试的可下载视频记录。

  套件部分显示一个表，其中包含有关设备套件的信息。

  在此表中，测试结果列按设备上运行的每个测试套件的结果汇总了测试数。这些数据还有一个图形组件。有关更多信息，请参阅 多个测试的状态。

  要按套件查看完整结果，请按照 查看测试套件报告 中的说明操作。

  日志部分显示 Device Farm 在运行期间为该设备记录的任何信息。要查看此信息，请按照查看报告中的问题、设备、套件或测试的日志信息中的说明操作。

  文件部分显示该设备的套件以及任何可下载关联文件（如日志文件）的列表。要下载文件，请在列表中选择该文件的链接。

  截图部分显示 Device Farm 在运行期间为该设备捕获的任何屏幕截图的列表（按套件分组）。

查看测试套件报告

1. 在 Devices (设备) 部分，选择设备。

2. 在套件部分，从表中选择套件。

   Video (视频) 部分显示该测试的可下载视频记录。

   测试部分显示一个表格，其中包含有关套件中测试的信息。

   在表中，测试结果列显示结果。这些数据还有一个图形组件。有关更多信息，请参阅 多个测试的状态。

   要按测试查看完整结果，请按照 查看测试报告 中的说明操作。

   日志部分显示 Device Farm 在运行期间为该套件记录的任何信息。要查看此信息，请按照查看报告中的问题、设备、套件或测试的日志信息中的说明操作。

   文件部分显示该套件的测试以及任何可下载关联文件（如日志文件）的列表。要下载文件，请在列表中选择该文件的链接。

   屏幕截图部分显示 Device Farm 在运行期间为该套件捕获的任何屏幕截图的列表（按测试分组）。

查看测试报告

1. 在 Devices (设备) 部分，选择设备。
2. 在 Suites (套件) 部分中选择套件。
3. 在测试部分中，选择测试。
4. Video (视频) 部分显示该测试的可下载视频记录。

   结果部分显示测试结果。状态以结果图标表示。有关更多信息，请参阅 单个测试的状态。

   日志部分显示 Device Farm 在测试期间记录的任何信息。要查看此信息，请按照查看报告中的问题、设备、套件或测试的日志信息中的说明操作。

   文件选项卡显示该测试的任何可下载关联文件（如日志文件）的列表。要下载文件，请在列表中选择该文件的链接。

   屏幕截图选项卡显示 Device Farm 在测试期间捕获的任何屏幕截图的列表。

查看报告中的问题、设备、套件或测试的日志信息

日志部分显示以下信息：

- Source (来源) 表示日志条目的来源。可能的值包括：
  - Harness 表示 Device Farm 创建的日志条目。这些日志条目通常在启动和停止事件期间创建。
  - 设备表示设备创建的日志条目。对于 Android，这些日志条目与 Logcat 兼容。对于 iOS，这些日志条目与 syslog 兼容。
  - Test (测试) 表示某测试或其测试框架创建的一个日志条目。
- Time (时间) 表示第一个日志条目与此日志条目之间相隔的时间。时间以*MM:SS.SSS*格式表示，其中*M*表示分钟，*S*代表秒。
- PID 表示创建了日志条目的进程标识符 (PID)。设备上的应用程序创建的所有日志条目具有相同的 PID。
- Level (级别) 表示日志条目的日志记录级别。例如，`Logger.debug("This is a message!")` 会记录 Debug 的级别。可能的值有：
  - 提醒
  - 重大
  - Debug
  - Emergency
  - 错误
  - Errored
  - 已失败
  - 信息
  - Internal
  - Notice
  - Passed
  - Skipped
  - Stopped
  - 详细
  - Warned
  - 警告
- Tag (标签) 表示日志条目的任意元数据。例如，Android Logcat 可用它来描述系统的哪个部分创建了该日志条目 (例如，`ActivityManager`)。
- Message (消息) 表示日志条目的消息或数据。例如，`Logger.debug("Hello, World!")` 会记录 `"Hello, World!"` 的消息。

仅显示信息的一部分：

- 要显示与特定列中的某个值匹配的所有日志条目，请在搜索栏中输入该值。例如，要显示源值为 `Harness` 的所有日志条目，请在搜索栏中输入 **`Harness`**。
- 要从列标题框中删除所有字符，请选择该列标题框中的 X。从列标题框中删除所有字符与在该列标题框中键入 **`*`** 的作用相同。

要下载设备的所有日志信息，包括曾运行的所有套件和测试，请选择下载日志。

Device Farm 测试结果状态

Device Farm 控制台中显示的图标可帮助您快速评估已完成的测试运行的状态。有关 Device Farm 的更多信息，请参阅 AWS Device Farm 中的报告。

主题

- 单个测试的状态
- 多个测试的状态

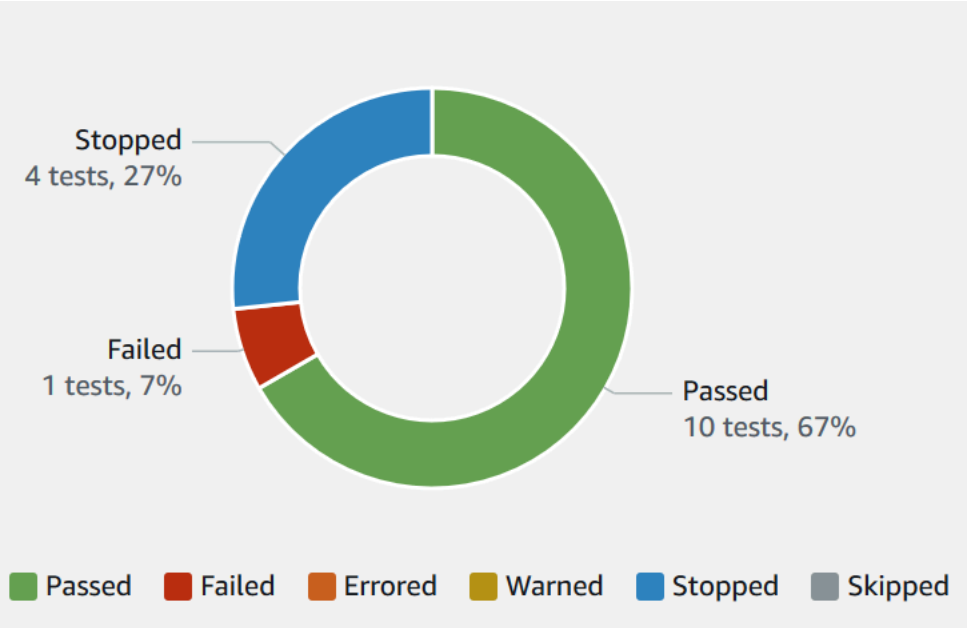
单个测试的状态

对于描述单个测试的报告，Device Farm 会显示表示测试结果状态的图标：

说明	图标
测试成功。	
测试失败。	
Device Farm 跳过了测试。	
已停止测试。	
Device Farm 返回了警告。	
Device Farm 返回了错误。	

多个测试的状态

如果您选择已完成的运行，Device Farm 会显示一个汇总图表，其中显示不同状态下的测试所占百分比。



例如，此测试运行结果栏显示运行中有 4 个测试已停止、有 1 个测试失败以及有 10 个测试成功。

图表始终采用颜色编码和标记。

在 Device Farm 中下载构件

Device Farm 会收集运行中的每个测试的构件，如报告、日志文件和图像。

您可以下载测试运行期间创建的项目：

文件

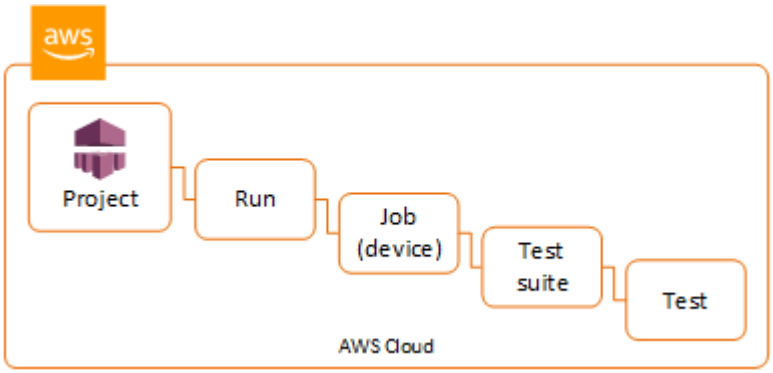
测试运行期间生成的包括 Device Farm 报告的文件。有关更多信息，请参阅 在 Device Farm 中查看测试报告。

日志

测试运行中的每个测试的输出。

屏幕截图

针对测试运行中的每个测试记录的屏幕图像。



下载构件（控制台）

1. 在测试运行报告页面上，从 Devices (设备) 中选择一个移动设备。
2. 要下载文件，请从 Files (文件) 中选择一个文件。
3. 要下载测试运行的日志，请从 Logs (日志) 中选择 Download logs (下载日志)。
4. 要下载屏幕截图，请从 Screenshots (屏幕截图) 中选择一个屏幕截图。

有关在自定义测试环境中下载项目的更多信息，请参阅在自定义测试环境中下载构件。

下载构件（AWS CLI）

您可以使用列 AWS CLI 出您的测试运行工件。

主题

- 步骤 1：获取 Amazon 资源名称（ARN）
- 步骤 2：列出您的构件
- 步骤 3：下载您的构件

步骤 1：获取 Amazon 资源名称（ARN）

您可以按运行、作业、测试套件或者测试列出项目。您需要相应的 ARN。下表显示了每个 AWS CLI 列表命令的输入 ARN：

AWS CLI 列表命令	所需的 ARN
list-projects	此命令返回所有项目，并且不需要 ARN。

AWS CLI 列表命令	所需的 ARN
list-runs	`project`
list-jobs	`run`
list-suites	`job`
list-tests	`suite`

例如，要查找测试 ARN，请使用测试套件 ARN 作为输入参数来运行 list-tests。

示例：

```
aws devicefarm list-tests --arn arn:MyTestSuiteARN
```

响应中包含测试套件中每个测试的测试 ARN。

```
{
    "tests": [
        {
            "status": "COMPLETED",
            "name": "Tests.FixturesTest.testExample",
            "created": 1537563725.116,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 1.89,
                "metered": 1.89
            },
            "result": "PASSED",
            "message": "testExample passed",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:test:5e01a8c7-c861-4c0a-
b1d5-12345EXAMPLE",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 1,
                "errored": 0,
                "total": 1
```

```
            }
        }
    ]
}
```

步骤 2：列出您的构件

AWS CLI list-ar tifacts 命令返回构件列表，例如文件、屏幕截图和日志。每个项目都有一个 URL，方便您下载该文件。

- 调用指定了运行、作业、测试套件或测试 ARN 的 list-artifacts。指定 FILE、LOG 或 SCREENSHOT 的类型。

  此示例会返回单个测试可用的每个项目的下载 URL：

  ```
  aws devicefarm list-artifacts --arn arn:MyTestARN --type "FILE"
  ```

  响应中包含每个项目的下载 URL。

  ```
  {
      "artifacts": [
          {
              "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
  ExampleURL",
              "extension": "txt",
              "type": "APPIUM_JAVA_OUTPUT",
              "name": "Appium Java Output",
              "arn": "arn:aws:devicefarm:us-west-2:123456789101:artifact:5e01a8c7-
  c861-4c0a-b1d5-12345EXAMPLE",
          }
      ]
  }
  ```

步骤 3：下载您的构件

- 使用上一步骤获得的 URL 下载您的项目。此示例使用 curl 下载 Android Appium Java 输出文件：

  ```
  curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL"
   > MyArtifactName.txt
  ```

下载构件（API）

Device Farm API ListArtifacts方法会返回构件列表，例如文件、屏幕截图和日志。每个项目都有一个URL，方便您下载该文件。

在自定义测试环境中下载构件

在自定义测试环境中，Device Farm 会收集自定义报告、日志文件和图像等构件。这些项目可用于测试运行中的每台设备。

您可以下载这些在测试运行期间创建的项目：

测试规范输出

运行测试规范 YAML 文件中的命令获得的输出。

客户项目

一个压缩文件，其中包含测试运行的项目。可在测试规范 YAML 文件中的项目：部分配置客户项目。

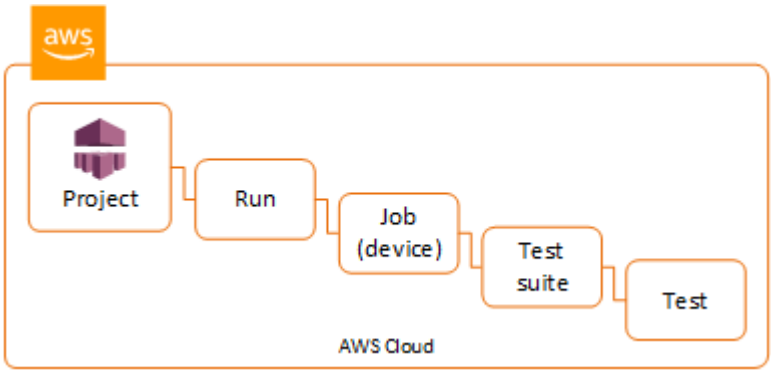
测试规范 shell 脚本

从您的 YAML 文件创建的中间 Shell 脚本文件。由于该脚本在测试运行中使用，因此可以使用 shell 脚本文件调试 YAML 文件。

测试规范文件

测试运行中使用的 YAML 文件。

有关更多信息，请参阅 在 Device Farm 中下载构件。



标记 AWS Device Farm 资源

AWS Device Farm 与 Res AWS ource Groups 标记 API 配合使用。此 API 允许您使用标签管理 AWS 账户中的资源。您可以为资源添加标签，例如项目和测试运行。

您可以使用标签执行以下操作：

- 组织 AWS 账单来反映您自身的成本结构。要执行此操作，请注册以获取包含标签密钥值的 AWS 账户账单。然后，如需查看组合资源的成本，请按有同样标签键值的资源组织您的账单信息。例如，您可以将某个应用程序名称用作几个资源的标签，然后组织账单信息，以便查看多个服务中使用该应用程序的总成本。有关更多信息，请参阅《关于 AWS 账单与成本管理》中的成本分配和标签。
- 通过 IAM 策略控制访问。为此，您需要使用标签值条件创建允许访问一个资源或一组资源的策略。
- 标识和管理具有标签形式的特定属性的运行，例如指定用于测试的分支的属性。

有关标记资源的更多信息，请参阅标记最佳实践白皮书。

主题

标注资源

借助 AWS Resource Group Tagging API，您可以添加、删除或修改资源上的标签。有关更多信息，请参阅 AWS Resource Group Tagging API Reference。

要标记资源，请使用 `resourcegroupstaggingapi` 终端节点中的 `TagResources` 操作。此操作 ARNs 从支持的服务中获取列表和键值对列表。值是可选的。空字符串表示该标签不应有值。例如，以下 Python 示例 ARNs 使用带有值的标签标记`build-config`了一系列项目`release`：

```
import boto3

client = boto3.client('resourcegroupstaggingapi')

client.tag_resources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000",
```

```
                                        "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655441111",
                                        "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655442222"]
                      Tags={"build-config":"release", "git-commit":"8fe28cb"})
```

标签值不是必需的。要设置不带值的标签，请在指定值时使用空字符串 ("")。一个标签只能有一个值。标签对某个资源使用的任何先前值都将被新值覆盖。

按标签查找资源

要按标签查找资源，请使用 `resourcegrouptaggingapi` 终端节点中的 `GetResources` 操作。此操作会采用一系列筛选条件（也可以不采用筛选条件），并返回与给定条件匹配的资源。如果不采用筛选条件，则会返回所有带标签的资源。`GetResources` 操作允许您根据以下条件筛选资源

- 标签值
- 资源类型（例如 `devicefarm:run`）

有关更多信息，请参阅 AWS Resource Group Tagging API Reference。

以下示例使用具有 `production` 值的 `stack` 标签查找 Device Farm 桌面浏览器测试会话（`devicefarm:testgrid-session` 资源）：

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
sessions = client.get_resources(ResourceTypeFilters=['devicefarm:testgrid-session'],
                                TagFilters=[
                                  {"Key":"stack","Values":["production"]}
                                ])
```

从资源中删除标签

要删除标签，请使用 `UntagResources` 操作，并指定资源列表和要删除的标签：

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
client.UntagResources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000"], TagKeys=["RunCI"])
```

AWS Device Farm 中的测试框架和内置测试

本节介绍 Device Farm 对测试框架和内置测试类型的支持。

Device Farm 通过让您将应用程序和测试上传到由该服务管理的安全的 Amazon S3 存储桶来运行自动测试。上传后，它会启动底层基础架构，包括服务管理的测试主机，并在多台设备上并行执行测试。测试结果存储在服务托管 S3 存储桶中。这种架构称为服务端执行，是一种在物理上靠近设备的主机上运行测试的快速高效方法，无需自己管理测试主机基础架构。这种方法可以很好地扩展到在许多设备上独立进行测试，也可以在 CI/CD 管道环境中进行测试。

有关 Device Farm 如何运行测试的更多信息，请参阅 AWS Device Farm 中的测试环境。

> **Note**
>
> 对于 Appium 测试人员，你可能更喜欢在本地环境中运行 Appium 测试。通过远程访问会话，您可以运行客户端 Appium 测试。如需了解更多信息，请参阅客户端 Appi um 测试。

测试框架

Device Farm 支持以下移动自动化测试框架：

Android 应用程序测试框架

- Appium 自动测试
- Instrumentation

iOS 应用程序测试框架

- Appium 自动测试
- XCTest
- XCTest 用户界面

Web 应用程序测试框架

使用 Appium 支持 Web 应用程序。有关将测试引入 Appium 的更多信息，请参阅在 Device Farm 中自动运行 Appium 测试。

自定义测试环境中的框架

Device Farm 不支持为 XCTest 框架自定义测试环境。有关更多信息，请参阅 AWS Device Farm 中的自定义测试环境。

Appium 版本支持

对于在自定义环境中运行的测试，Device Farm 支持 Appium 版本 1。有关更多信息，请参阅 AWS Device Farm 中的测试环境。

内置测试类型

借助内置测试，您可以在多个设备上测试您的应用程序，而不必编写和维护测试自动化脚本。Device Farm 提供了一种内置测试类型：

- 内置：模糊（Android 和 iOS）

在 Device Farm 中自动运行 Appium 测试

> **Note**
>
> 本页介绍了在 Device Farm 的托管服务器端执行环境中运行 Appium 测试。要在远程访问会话期间从本地客户端环境运行 Appium 测试，请参阅客户端 Appium 测试。

本节介绍如何配置、打包和上传您的 Appium 测试，以便在 Device Farm 的托管服务器端环境中运行。Appium 是一种开源工具，用于自动执行本机和移动合应用程序。有关更多信息，请参阅 Appium 网站上的 Appium 简介。

有关示例应用程序和工作测试链接，请参阅适用于 Android 的 Device Farm 示例应用程序和适用于 iOS 的 Device Farm 示例应用程序 GitHub。

有关在 Device Farm 中进行测试以及服务器端工作原理的更多信息，请参阅AWS Device Farm 中的测试框架和内置测试。

选择 Appium 版本

> **Note**
>
> 对特定 Appium 版本、Appium 驱动程序或编程的支持 SDKs 将取决于为测试运行选择的设备和测试主机。

Device Farm 测试主机预装了 Appium，以便在更简单的用例中更快地设置测试。但是，如果需要，使用测试规范文件可以安装不同版本的 Appium。

场景 1：预配置的 Appium 版本

根据测试主机，Device Farm 预先配置了不同的 Appium 服务器版本。主机附带的工具可使用设备平台的默认驱动程序启用预配置版本（ UiAutomator2 个适用于 Android，iOS 有 2 个）。 XCUITest

```
phases:
  install:
    commands:
      - export APPIUM_VERSION=2
      - devicefarm-cli use appium $APPIUM_VERSION
```

要查看支持的软件列表，请参阅中的主题自定义测试环境中支持的软件。

场景 2：自定义 Appium 版本

要选择自定义版本的 Appium，请使用npm命令进行安装。以下示例显示如何安装最新版本的 Appium 2。

```
phases:
  install:
    commands:
      - export APPIUM_VERSION=2
      - npm install -g appium@$APPIUM_VERSION
```

场景 3：旧版 iOS 主机上的 Appium

在上旧版 iOS 测试主机，你可以使用选择特定的 Appium 版本。avm例如，要使用avm命令将 Appium 服务器版本设置为2.1.2，请将这些命令添加到您的测试规范 YAML 文件中。

```
phases:
```

```
  install:
    commands:
      - export APPIUM_VERSION=2.1.2
      - avm $APPIUM_VERSION
```

为 iOS 测试选择 WebDriverAgent 版本

要在 iOS 设备上运行 Appium 测试，需要使用。 WebDriverAgent 此应用程序必须经过签名才能安装在 iOS 设备上。Device Farm 提供了在自定义测试环境运行期间可用的预签名版本。 WebDriverAgent

以下代码片段可用于在测试规范文件中的 De WebDriverAgent vice Farm 上选择与您的 XCTest UI 驱动程序版本兼容的版本。

```
phases:
  pre_test:
    commands:
      - |-
        APPIUM_DRIVER_VERSION=$(appium driver list --installed --json | jq -r
 ".xcuitest.version" | cut -d "." -f 1);
        CORRESPONDING_APPIUM_WDA=$(env | grep
 "DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V${APPIUM_DRIVER_VERSION}")
        if [[ ! -z "$APPIUM_DRIVER_VERSION" ]] && [[ ! -z
 "$CORRESPONDING_APPIUM_WDA" ]]; then
          echo "Using Device Farm's prebuilt WDA version ${APPIUM_DRIVER_VERSION}.x,
 which corresponds with your driver";
          DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo $CORRESPONDING_APPIUM_WDA |
 cut -d "=" -f2)
        else
          LATEST_SUPPORTED_WDA_VERSION=$(env | grep
 "DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V" | sort -V -r | head -n 1)
          echo "Unknown driver version $APPIUM_DRIVER_VERSION; falling back to the
 Device Farm default version of $LATEST_SUPPORTED_WDA_VERSION";
          DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo $LATEST_SUPPORTED_WDA_VERSION
 | cut -d "=" -f2)
        fi;
```

有关更多信息 WebDriverAgent，请参阅 Appium 的文档。

将 Appium 测试与 Device Farm 集成

按照以下说明将 Appium 测试与 AWS Device Farm 集成。有关在 Device Farm 中使用 Appium 测试的更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

配置您的 Appium 测试程序包

使用下面的说明来配置您的测试程序包。

Java (JUnit)

1. 修改 pom.xml 以将包设置为 JAR 文件：

```
<groupId>com.acme</groupId>
<artifactId>acme-myApp-appium</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

2. 修改 pom.xml 以使用 maven-jar-plugin 将测试构建到 JAR 文件中。

   以下插件将您的测试源代码 (src/test 目录中的任何内容) 构建到 JAR 文件中：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. 修改 pom.xml 以使用 maven-dependency-plugin 将依赖项构建为 JAR 文件。

   以下插件会将您的依赖项复制到 dependency-jars 目录：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
```

```
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4. 将以下 XML 组件保存到 `src/main/assembly/zip.xml`。

   以下 XML 是一个组件定义，配置后可指示 Maven 构建一个 .zip 文件，其中包含构建输出目录和 `dependency-jars` 目录的根中的所有内容：

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. 修改 pom.xml 以使用 maven-assembly-plugin 将测试和所有依赖项打包到一个 .zip 文件。

   每当 mvn package 运行时，以下插件便使用前面的组件在构建输出目录中创建一个名为 zip-with-dependencies 的 .zip 文件：

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

> **Note**
>
> 如果您收到表明 1.3 不支持注释的错误消息，请将以下内容添加到 pom.xml：
>
> ```
> <plugin>
>   <artifactId>maven-compiler-plugin</artifactId>
>   <configuration>
>     <source>1.7</source>
>     <target>1.7</target>
>   </configuration>
> </plugin>
> ```

Java (TestNG)

1. 修改 pom.xml 以将包设置为 JAR 文件：

```
<groupId>com.acme</groupId>
<artifactId>acme-myApp-appium</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

2. 修改 pom.xml 以使用 maven-jar-plugin 将测试构建到 JAR 文件中。

   以下插件将您的测试源代码 (src/test 目录中的任何内容) 构建到 JAR 文件中：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. 修改 pom.xml 以使用 maven-dependency-plugin 将依赖项构建为 JAR 文件。

   以下插件会将您的依赖项复制到 dependency-jars 目录：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
```

```
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4. 将以下 XML 组件保存到 `src/main/assembly/zip.xml`。

   以下 XML 是一个组件定义，配置后可指示 Maven 构建一个 .zip 文件，其中包含构建输出目录和 `dependency-jars` 目录的根中的所有内容：

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. 修改 `pom.xml` 以使用 `maven-assembly-plugin` 将测试和所有依赖项打包到一个 .zip 文件。

每当 mvn package 运行时，以下插件便使用前面的组件在构建输出目录中创建一个名为 zip-with-dependencies 的 .zip 文件：

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

> **Note**
>
> 如果您收到表明 1.3 不支持注释的错误消息，请将以下内容添加到 pom.xml：
>
> ```
> <plugin>
>   <artifactId>maven-compiler-plugin</artifactId>
>   <configuration>
>     <source>1.7</source>
>     <target>1.7</target>
>   </configuration>
> </plugin>
> ```

Node.JS

要打包 Appium Node.js 测试并将其上传到 Device Farm，您必须在本地计算机上安装以下内容：

- 节点版本管理器 (nvm)

  在开发和打包测试时使用此工具，以便测试程序包中不包含不必要的依赖项。
- Node.js
- npm-bundle（全局安装）

1. 验证 nvm 是否存在

   ```
   command -v nvm
   ```

   您应在输出中看到 nvm。

   有关更多信息，请参阅 nvm on。 GitHub
2. 运行此命令以安装 Node.js：

   ```
   nvm install node
   ```

   您可以指定 Node.js 的特定版本：

   ```
   nvm install 11.4.0
   ```
3. 验证是否正在使用正确版本的 Node：

   ```
   node -v
   ```
4. 全局安装 npm-bundle：

   ```
   npm install -g npm-bundle
   ```

Python

1. 我们强烈建议您设置 Python virtualenv 用于开发和打包测试，以便不必要的依赖项不包含在您的应用程序包中。

   ```
   $ virtualenv workspace
   $ cd workspace
   $ source bin/activate
   ```

> **Tip**
>
> - 请勿使用 `--system-site-packages` 选项创建 Python virtualenv，因为它会从全局 site-packages 目录中继承程序包。这可能导致您将测试不需要的依赖项包含在您的虚拟环境中。
> - 您还应验证您的测试不使用依赖本机库的依赖项，因为这些本机库可能不位于运行这些测试的实例上。

2. 在您的虚拟环境中安装 py.test。

```
$ pip install pytest
```

3. 在您的虚拟环境中安装 Appium Python 客户端。

```
$ pip install Appium-Python-Client
```

4. 除非您在自定义模式下指定其他路径，否则 Device Farm 会认为您的测试存储在 `tests/` 中。您可以使用 `find` 显示文件夹中的所有文件：

```
$ find tests/
```

确认这些文件包含您要在 Device Farm 上运行的测试套件

```
tests/
tests/my-first-tests.py
tests/my-second-tests/py
```

5. 从虚拟环境工作空间文件夹运行此命令，以显示测试列表而不运行它们。

```
$ py.test --collect-only tests/
```

确认输出显示您要在 Device Farm 上运行的测试。

6. 清除 tests/ 文件夹下的所有缓存文件：

```
$ find . -name '__pycache__' -type d -exec rm -r {} +
$ find . -name '*.pyc' -exec rm -f {} +
$ find . -name '*.pyo' -exec rm -f {} +
```

```
$ find . -name '*~' -exec rm -f {} +
```

7. 在您的工作空间中运行以下命令以生成 requirements.txt 文件：

```
$ pip freeze > requirements.txt
```

Ruby

要打包 Appium Ruby 测试并将其上传到 Device Farm，您必须在本地计算机上安装以下内容：

- Ruby 版本管理器 (RVM)

  在开发和打包测试时使用此命令行工具，以便测试程序包中不包含不必要的依赖项。
- Ruby
- Bundler（此 gem 通常与 Ruby 一起安装。）

1. 安装所需的密钥、RVM 和 Ruby。有关说明，请参阅 RVM 网站上的安装 RVM。

   安装完成后，通过注销然后再次登录来重新加载终端。

   > **Note**
   >
   > RVM 仅作为 bash shell 的函数加载。

2. 验证 rvm 是否已正确安装。

```
command -v rvm
```

   您应在输出中看到 `rvm`。

3. 如果要安装特定版本的 Ruby，例如*2.5.3*，请运行以下命令：

```
rvm install ruby 2.5.3 --autolibs=0
```

   验证您是否在使用所请求的 Ruby 版本：

```
ruby -v
```

4. 配置捆绑器以编译适用于所需测试平台的软件包：

```
bundle config specific_platform true
```

5. 更新您的 .lock 文件以添加运行测试所需的平台。

   - 如果您正在编译要在 Android 设备上运行的测试，请运行以下命令将 Gemfile 配置为使用 Android 测试主机的依赖项：

     ```
     bundle lock --add-platform x86_64-linux
     ```

   - 如果您正在编译要在 iOS 设备上运行的测试，请运行以下命令将 Gemfile 配置为使用 iOS 测试主机的依赖项：

     ```
     bundle lock --add-platform x86_64-darwin
     ```

6. 默认情况下，通常会安装 bundler gem。如果未安装，请安装它：

   ```
   gem install bundler -v 2.3.26
   ```

创建压缩程序包文件

> **Warning**
>
> 在 Device Farm 中，压缩后的测试包中文件的文件夹结构很重要，一些归档工具会隐式更改 ZIP 文件的结构。我们建议您使用下面指定的命令行实用程序，而不是使用本地桌面文件管理器中内置的归档实用程序（例如 Finder 或 Windows Explorer）。

现在，将测试打包以用于 Device Farm。

Java (JUnit)

构建和打包测试：

```
$ mvn clean package -DskipTests=true
```

最后将创建文件 `zip-with-dependencies.zip`。这是您的测试程序包。

Java (TestNG)

构建和打包测试：

```
$ mvn clean package -DskipTests=true
```

最后将创建文件 `zip-with-dependencies.zip`。这是您的测试程序包。

Node.JS

1. 查看您的项目。

   确保您位于项目的根目录中。您可以在根目录中看到 `package.json`。

2. 运行此命令来安装您的本地依赖项。

   ```
   npm install
   ```

   此命令还会在当前目录中创建一个 `node_modules` 文件夹。

   > **Note**
   >
   > 此时，您应该能够在本地运行您的测试。

3. 运行此命令以将当前文件夹中的文件打包为 *.tgz 文件。使用 `package.json` 文件中的 `name` 属性命名此文件。

   ```
   npm-bundle
   ```

   此 tarball (.tgz) 文件包含您的所有代码和依赖项。

4. 运行此命令将上一步中生成的 tarball（*.tgz 文件）捆绑到单个压缩存档中：

   ```
   zip -r MyTests.zip *.tgz
   ```

   这是您在以下过程中上传到 Device Farm 的 `MyTests.zip` 文件。

Python

Python 2

使用 pip 生成所需的 Python 程序包的存档（称为“wheelhouse”）：

```
$ pip wheel --wheel-dir wheelhouse -r requirements.txt
```

将 wheelhouse、测试和 pip 要求打包到 zip 存档中，以用于 Device Farm：

```
$ zip -r test_bundle.zip tests/ wheelhouse/ requirements.txt
```

Python 3

将测试和 pip 要求打包到 zip 文件中：

```
$ zip -r test_bundle.zip tests/ requirements.txt
```

Ruby

1. 运行此命令以创建虚拟 Ruby 环境：

   ```
   # myGemset is the name of your virtual Ruby environment
   rvm gemset create myGemset
   ```

2. 运行此命令以使用您刚刚创建的环境：

   ```
   rvm gemset use myGemset
   ```

3. 请查看您的源代码。

   确保您位于项目的根目录中。您可以在根目录中看到 `Gemfile`。

4. 运行此命令以从 `Gemfile` 安装您的本地依赖项和所有 Gem。

   ```
   bundle install
   ```

   > **Note**
   >
   > 此时，您应该能够在本地运行您的测试。使用此命令在本地运行测试：

```
bundle exec $test_command
```

5. 将您的 Gem 打包到 vendor/cache 文件夹中。

```
# This will copy all the .gem files needed to run your tests into the vendor/
cache directory
bundle package --all-platforms
```

6. 运行以下命令将源代码以及所有依赖项捆绑到单个压缩存档中：

```
zip -r MyTests.zip Gemfile vendor/ $(any other source code directory files)
```

这是您在以下过程中上传到 Device Farm 的 MyTests.zip 文件。

将您的测试程序包上传到 Device Farm

您可以使用 Device Farm 控制台上传测试。

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 如果您是新用户，请选择新建项目，输入项目的名称，然后选择提交。

   如果您已有项目，可以选择该项目以将您的测试上传到该项目。
4. 打开您的项目，然后选择创建运行。
5. 在运行设置下，为您的测试指定一个适当的名称。这可能包含空格或标点符号的任意组合。
6. 适用于本机 Android 和 iOS 测试

   如果您测试的是 Android（.apk）应用程序，请在运行设置下选择 Android 应用程序；如果您测试的是 iOS（.ipa）应用程序，请选择 iOS 应用程序。然后，在选择应用程序下，选择上传自己的应用程序以上传您应用程序的可分发程序包。

> **Note**
>
> 该文件必须是 Android .apk 或 iOS .ipa。iOS 应用程序必须是针对真实设备而不是模拟器构建的。

适用于移动 Web 应用程序测试

在运行设置下，选择 Web 应用程序。

7. 在配置测试下的选择测试框架部分中，选择测试使用的 Appium 框架，然后上传自己的测试程序包。
8. 浏览到并选择包含您的测试的 .zip 文件。该 .zip 文件必须遵循配置您的 Appium 测试程序包中所述的格式。
9. 按照说明来选择设备并开始运行。有关更多信息，请参阅 在 Device Farm 中创建测试运行。

> **Note**
>
> Device Farm 不会修改 Appium 测试。

拍摄测试的屏幕截图（可选）

您可以拍摄屏幕截图作为测试的一部分。

Device Farm 会将 `DEVICEFARM_SCREENSHOT_PATH` 属性设置为本地文件系统中的一个完全限定路径（这是 Device Farm 希望 Appium 屏幕截图保存到的路径)。存储屏幕截图的特定于测试的目录在运行时定义。系统会自动将这些屏幕截图提取到您的 Device Farm 报告中。要查看屏幕截图，请在 Device Farm 控制台中选择 Screenshots (屏幕截图) 部分。

有关在 Appium 测试中拍摄屏幕截图的更多信息，请参阅 Appium API 文档中的拍摄屏幕截图。

AWS Device Farm 中的 Android 测试

Device Farm 为适用于 Android 设备的多种自动化测试类型提供支持，并提供两种内置测试。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

Android 应用程序测试框架

以下测试适用于 Android 设备。

- Appium 自动测试
- Instrumentation

Android 的内置测试类型

有一种内置测试类型适用于 Android 设备：

- 内置：模糊（Android 和 iOS）

适用于 Android 的 Instrumentation 与 AWS Device Farm

Device Farm 支持安卓版仪器（JUnit、Espresso、Robotium 或任何基于仪器的测试）。

Device Farm 还提供了示例 Android 应用程序，以及指向三个 Android 自动化框架（包括 Instrumentation (Espresso)）中的工作测试的链接。安卓版 Device Farm 示例应用程序可在上下载 GitHub。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

主题

- 什么是 Instrumentation？
- Android Instrumentation 测试的注意事项
- 标准模式测试解析
- 将 Android Instrumentation 与 Device Farm 集成

什么是 Instrumentation？

使用 Android Instrumentation，您可在测试代码中调用回调方法，从而能够逐步运行组件的生命周期，就像在调试组件一样。有关更多信息，请参阅《Android Developer Tool》https://developer.android.com/studio/test/test-in-android-studio#test_types_and_locations 文档的“Test types and locations” 一节中的 Instrumented tests。

Android Instrumentation 测试的注意事项

使用 Android Instrumentation 时应注意以下建议和注意事项。

查看 Android 操作系统的兼容性

请查看 Android 文档，确保 Instrumentation 与您的 Android 操作系统版本兼容。

从命令行运行

要通过命令行运行 Instrumentation 测试，请按照 Android 文档进行操作。

系统动画

根据有关 Espresso 测试的 Android 文档，建议在真实设备上测试时关闭系统动画。Device Farm 使用 an droid.support.test.runner. JUnit Android Runner 插件测试运行器执行时，它会自动禁用窗口动画比例、过渡动画比例和动画师持续时间缩放设置。

测试记录器

Device Farm 支持具有 record-and-playback脚本工具的框架，例如 Robotium。

标准模式测试解析

在标准运行模式下，Device Farm 会解析您的测试套件并识别它将运行的唯一测试类和方法。此启用方式是通过名为 Dex Test Parser 的工具实现的。

当给定一个 Android instrumenting .apk 文件作为输入时，解析器会返回符合 JUnit 3 和 JUnit 4 惯例的测试的完全限定方法名称。

要在本地环境中对此进行测试，请执行以下操作：

1. 下载 `dex-test-parser` 二进制文件。
2. 运行以下命令以获取将在 Device Farm 上运行的测试方法列表：

```
java -jar parser.jar path/to/apk path/for/output
```

将 Android Instrumentation 与 Device Farm 集成

> **Note**
>
> 按照以下说明将 Android Instrumentation 测试与 AWS Device Farm 集成。有关在 Device Farm 中使用 Instrumentation 测试的更多信息，请参阅适用于 Android 的 Instrumentation 与 AWS Device Farm。

上传 Android Instrumentation 测试

使用 Device Farm 控制台上传您的测试。

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。

2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 在项目列表中，选择要将测试上传到的项目。

> **Tip**
>
> 您可以使用搜索栏按名称筛选项目列表。
> 要创建项目，请按照在 AWS Device Farm 中创建项目中的说明操作。

4. 选择创建规则。
5. 在选择应用程序下的应用程序选择选项部分中，选择上传自己的应用程序。
6. 浏览到并选择您的 Android 应用程序文件。该文件必须是 .apk 文件。
7. 在配置测试的设置测试框架部分中，选择 Instrumentation，然后选择选择文件。
8. 浏览到并选择包含您的测试的 .apk 文件。
9. 按照剩余说明进行操作，以选择设备并开始运行。

（可选）在 Android Instrumentation 测试中截图

您可以拍摄屏幕截图作为您的 Android Instrumentation 测试的一部分。

要拍摄屏幕截图，请调用以下方法之一：

- 对于 Robotium，调用 `takeScreenShot` 方法 (例如，`solo.takeScreenShot();`)。
- 对于 Spoon，调用 `screenshot` 方法，例如：

```
Spoon.screenshot(activity, "initial_state");
/* Normal test code... */
Spoon.screenshot(activity, "after_login");
```

在测试运行期间，Device Farm 会从设备上的以下位置获得屏幕截图（如果存在），然后将它们添加到测试报告中：

- `/sdcard/robotium-screenshots`
- `/sdcard/test-screenshots`
- `/sdcard/Download/spoon-screenshots/`*`test-class-name`*`/`*`test-method-name`*
- `/data/data/`*`application-package-name`*`/app_spoon-screenshots/`*`test-class-name`*`/`*`test-method-name`*

AWS Device Farm 中的 iOS 测试

Device Farm 为适用于 iOS 设备的多种自动化测试类型提供支持，并提供一种内置测试。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

iOS 应用程序测试框架

以下测试适用于 iOS 设备。

- Appium 自动测试
- XCTest
- XCTest 用户界面

iOS 的内置测试类型

目前有一种内置测试类型可用于 iOS 设备。

- 内置：模糊（Android 和 iOS）

将 Device Farm 与 XCTest 适用于 iOS 的集成

借助 Device Farm，您可以使用该 XCTest 框架在真实设备上测试您的应用程序。有关更多信息 XCTest，请参阅 Xcode 测试中的测试基础知识。

要运行测试，请为测试运行创建程序包，然后将这些程序包上传到 Device Farm。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

主题

- 为你的 XCTest 跑步创建软件包
- 将你的 XCTest 跑步包上传到 Device Farm

为你的 XCTest 跑步创建软件包

要使用该 XCTest 框架测试您的应用程序，Device Farm 需要满足以下条件：

- 您的应用程序包采用 `.ipa` 文件形式。

- 您的 XCTest 包裹作为`.zip`文件。

您可以使用 Xcode 生成的构建输出来创建这些程序包。完成以下步骤以创建程序包，以便您可以将它们上传到 Device Farm。

要为您的应用程序生成构建输出

1. 在 Xcode 中打开应用程序项目。
2. 在 Xcode 工具栏的方案下拉菜单中，选择 Generic iOS Device (常规 iOS 设备) 作为目的地。
3. 在 Product (产品) 菜单中，选择 Build For (构建属于)，然后选择 Testing (测试)。

创建应用程序包

1. 在 Xcode 中的项目导航器的 Products (产品) 下，打开针对名为 *`app-project-name`*`.app` 的文件的上下文菜单。然后，选择 Show in Finder (在 Finder 中显示)。Finder 打开一个名为 `Debug-iphoneos` 的文件夹，其中包含 Xcode 为您的测试构建生成的输出。此文件夹包含您的 `.app` 文件。
2. 在 Finder 中，创建一个新文件夹，并将其命名为 `Payload`。
3. 复制 *`app-project-name`*`.app` 文件，然后将它粘贴到 `Payload` 文件夹中。
4. 打开 `Payload` 文件夹的上下文菜单，然后选择 Compress "Payload" (压缩“负载”)。此时会创建名为 `Payload.zip` 的文件。
5. 将文件夹名和扩展名 `Payload.zip` 更改为 *`app-project-name`*`.ipa`。

   在稍后的步骤中，您将此文件提供给 Device Farm。要使文件更易于查找，您可能需要将其移动到其他位置，例如桌面。
6. 或者，您可以删除 `Payload` 文件夹以及其中的 `.app` 文件。

创建 XCTest 软件包

1. 在 Finder 的 `Debug-iphoneos` 目录中，打开 *`app-project-name`*`.app` 文件的上下文菜单。然后，选择 Show Package Contents (显示程序包内容)。
2. 在程序包内容中，打开 `Plugins` 文件夹。此文件夹包含名为 *`app-project-name`*`.xctest` 的文件。
3. 打开此文件的上下文菜单，然后选择 Compress "***`app-project-name`*`.xctest`**" (压缩“app-project-name.xctest”)。此时会创建名为 *`app-project-name`*`.xctest.zip` 的文件。

在稍后的步骤中，您将此文件提供给 Device Farm。要使文件更易于查找，您可能需要将其移动到其他位置，例如桌面。

将你的 XCTest 跑步包上传到 Device Farm

使用 Device Farm 控制台上传用于您的测试的程序包。

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 如果您还没有项目，请创建一个项目。有关创建项目的步骤，请参阅在 AWS Device Farm 中创建项目。

   否则，在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 选择要用于运行测试的项目。
4. 选择创建运行。
5. 在运行设置下的运行类型部分中，选择 iOS 应用程序。
6. 在选择应用程序下的应用程序选择选项部分中，选择上传自己的应用程序。然后，在上传应用程序下，选择选择文件。
7. 浏览到用于您的应用程序的 `.ipa` 文件并上传它。

   > **Note**
   >
   > 必须构建 `.ipa` 程序包以进行测试。

8. 在 “配置测试” 下的 “选择测试框架” 部分，选择XCTest。然后，在上传应用程序下，选择选择文件。
9. 浏览到包含您的应用程序 XCTest 包`.zip`的文件并将其上传。
10. 完成项目创建过程中的其余步骤。您将选择要在其上进行测试的设备并指定设备状态。
11. 选择创建运行。Device Farm 运行测试并在控制台中显示结果。

将 iOS XCTest 用户界面与 Device Farm 集成

Device Farm 为 XCTest 用户界面测试框架提供支持。具体而言，Device Farm 支持同时使用 Objective-C 和 Swift 编写的 XCTest 用户界面测试。

XCTest 用户界面框架支持在 iOS 开发中进行用户界面测试，其基础是 XCTest。有关更多信息，请参阅 iOS Developer Library 中的 User Interface Testing。

有关 Device Farm 中测试的一般信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

按照以下说明将 Device Farm 与 iOS XCTest 用户界面测试框架集成。

主题

准备好你的 iOS XCTest 用户界面测试

您可以为 XCTEST_UI 测试程序包上传 `.ipa` 文件或 `.zip` 文件。

`.ipa` 文件是包含捆绑包格式的 iOS Runner 应用程序的应用程序存档。`.ipa` 文件中不能包含其他文件。

如果您上传一个 `.zip` 文件，它可以直接包含 iOS Runner 应用程序，也可以包含一个 `.ipa` 文件。如果您想在测试期间使用其他文件，您也可以在 `.zip` 文件中包含它们。例如，您可以在 `.zip` 文件中包含诸如 `.xctestrun`、`.xcworkspace` 或 `.xcodeproj` 之类的文件，以便在 Device Farm 上运行 XCUI 测试计划。XCUI 测试类型的默认测试规范文件中提供了关于如何运行测试计划的详细说明。

选项 1：创建 XCTest UI .ipa 包

yourAppNameUITest-runner.app 捆绑包是由 Xcode 在构建项目进行测试时生成的。您可在项目的 Products 目录中找到该捆绑包。

创建 .ipa 文件：

1. 创建名为 *`Payload`* 的目录。
2. 将您的应用程序目录添加到 Payload 目录。
3. 将 Payload 目录存档成一个 `.zip` 文件，然后将文件扩展名更改为 `.ipa`。

以下文件夹结构显示了如何*`my-project-nameUITest-Runner.app`*将名为的示例应用程序打包为`.ipa`文件：

```
.
### my-project-nameUITest.ipa
    ### Payload (directory)
        ### my-project-nameUITest-Runner.app
```

选项 2：创建 XCTest 用户界面.zip 包

Device Farm 会自动为您生成一个.xctestrun文件，用于运行完整的 XCTest 用户界面测试套件。如果您想在 Device Farm 上使用自己的 .xctestrun 文件，您可以将 .xctestrun 文件和应用程序目录压缩成一个 .zip 文件。如果您已经有测试包的.ipa文件，则可以将其包含在此处，而不是*-Runner.app。

```
.
### swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### SampleTestPlan_2.xctestrun
   ### SampleTestPlan_1.xctestrun
   ### (any other files)
```

如果你想在 Device Farm 上为 XCUI 测试运行 Xcode 测试计划，你可以创建一个 zip 文件，其中包含你的 my-project-nameUITest-runner.app 或 my-project-nameUITest. ip a 文件以及运行带有测试计划的 XCTEST_UI 所需的 xcode 源代码文件，包括或文件。.xcworkspace .xcodeproj

以下是使用 .xcodeproj 文件的 zip 示例：

```
.
### swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### (any directory)
   ### SampleXcodeProject.xcodeproj
        ### Testplan_1.xctestplan
        ### Testplan_2.xctestplan
        ### (any other source code files created by xcode with .xcodeproj)
```

以下是使用 .xcworkspace 文件的 zip 示例：

```
.
###swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### (any directory)
   #   ### SampleXcodeProject.xcodeproj
   #   ### Testplan_1.xctestplan
   #   ### Testplan_2.xctestplan
   |   ### (any other source code files created by xcode with .xcodeproj)
   ### SampleWorkspace.xcworkspace
       ### contents.xcworkspacedata
```

> **Note**
>
> 请确保您的 XCTest UI .zip 包中没有名为 “Payload” 的目录。

上传你的 iOS XCTest 用户界面测试

使用 Device Farm 控制台上传您的测试。

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 在项目列表中，选择要将测试上传到的项目。

   > **Tip**
   >
   > 您可以使用搜索栏按名称筛选项目列表。
   > 要创建项目，请按照 在 AWS Device Farm 中创建项目 中的说明操作。

4. 选择创建运行。
5. 在运行设置下的运行类型部分中，选择 iOS 应用程序。
6. 在选择应用程序下的应用程序选择选项部分中，选择上传自己的应用程序。然后，在上传应用程序下，选择选择文件。
7. 浏览到并选择您的 iOS 应用程序文件。该文件必须是 .ipa 文件。

> **Note**
>
> 确保为 iOS 设备 (而不是模拟器) 构建您的 .ipa 文件。

8. 在 “配置测试” 下的 “选择测试框架” 部分中，选择 XCTest UI。然后，在上传应用程序下，选择选择文件。
9. 浏览并选择包含您的 iOS XCTest 用户界面测试运行器的.ipa 或.zip 文件。
10. 完成运行创建过程中的其余步骤。您将选择要从中进行测试的设备，并可选择指定其他配置。
11. 选择创建运行。Device Farm 运行测试并在控制台中显示结果。

AWS Device Farm 中的 Web 应用程序测试

Device Farm 使用 Appium 为网络应用程序提供测试。有关在 Device Farm 上设置 Appium 测试的更多信息，请参阅 the section called “Appium 自动测试”。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

计量和非计量设备的规则

计量是指设备的计费。默认情况下，当免费试用时间用完后，系统便会对 Device Farm 设备进行计量并按分钟向您收取费用。您也可以选择购买非计量设备，这样便可采用包月收费方式无限制地测试。有关定价的更多信息，请参阅 AWS Device Farm 定价。

如果您选择使用包含 iOS 和 Android 设备的设备池启动运行，则存在适用于计量和非计量设备的规则。例如，如果您有 5 个非计量 Android 设备和 5 个非计量 iOS 设备，则您的 Web 测试运行将使用非计量设备。

下面是另一个示例：假设您有 5 个非计量 Android 设备和 0 个非计量 iOS 设备。如果您只为 Web 运行选择 Android 设备，将使用非计量设备。如果您为 Web 运行选择 Android 和 iOS 设备，将对计费方法进行计量，并且不会使用非计量设备。

AWS Device Farm 中的内置测试

Device Farm 为适用于 Android 和 iOS 设备的内置测试类型提供支持。

借助内置测试，您可以在多个设备上测试您的应用程序，而不必编写和维护测试自动化脚本。这样可以节省您的时间和精力，尤其是在您刚开始使用 Device Farm 时。Device Farm 提供了以下内置测试类型：

- 内置：模糊（Android 和 iOS） – 模糊测试将用户界面事件随机发送至设备，然后报告结果。

有关 Device Farm 中测试和测试框架的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

运行 Device Farm 的内置模糊测试（Android 和 iOS）

Device Farm 内置模糊测试将用户界面事件随机发送至设备，然后报告结果。

有关 Device Farm 中测试的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

运行内置模糊测试

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 在项目列表中，选择要在其中运行内置模糊测试的项目。

   > **Tip**
   >
   > 您可以使用搜索栏按名称筛选项目列表。
   > 要创建项目，请按照在 AWS Device Farm 中创建项目中的说明操作。

4. 选择创建运行。
5. 在运行设置下的运行类型部分中，选择您的运行类型。如果您没有可供测试的应用程序，或者您测试的是 Android（.apk）应用程序，请选择 Android 应用程序。如果您测试的是 iOS（.ipa）应用程序，请选择 iOS 应用程序。
6. 如果您没有可供测试的应用程序，请在选择应用程序下，选择选择 Device Farm 提供的示例应用程序。如果您自带应用程序，请选择上传自己的应用程序，然后选择您的应用程序文件。
7. 在配置测试下的设置测试框架部分中，选择内置：模糊。
8. 如果出现以下任一设置，您可以接受默认值或自己指定一个值：

   - Event count (事件计数)：指定一个介于 1 和 10000 之间的数字，表示要执行的模糊测试的用户界面事件数。
   - 事件限制：指定一个介于 0 和 1,000 之间的数字，表示执行下一个用户界面事件之前模糊测试等待的毫秒数。
   - Randomizer seed (随机程序种子)：为模糊测试指定一个数字，以便用于随机化用户界面事件。为后续模糊测试指定相同的数字可确保相同的事件序列。

9. 按照剩余说明进行操作，以选择设备并开始运行。

AWS Device Farm 中的自定义测试环境

AWS Device Farm 支持配置用于自动测试的自定义环境（自定义模式），这是针对所有 Device Farm 用户的推荐方法。要详细了解 Device Farm 中的环境，请参阅测试环境。

与标准模式相比，自定义模式的优势包括：

- 更快地执行 end-to-end测试：不会对测试包进行解析以检测套件中的每个测试，从而避免了 preprocessing/postprocessing 开销。
- 实时日志和视频流：使用自定义模式时，您的客户端测试日志和视频将进行实时流式传输。此功能在标准模式中不可用。
- 捕获所有构件：在主机和设备上，自定义模式允许您捕获所有测试构件。在标准模式下可能无法做到这一点。
- 更一致且可复制的本地环境：在标准模式下，将为每个单独的测试单独提供构件，这在某些情况下可能很有用。但是，您的本地测试环境可能会与原始配置有所不同，因为 Device Farm 对每个已执行的测试的处理方式不同。

  相比之下，借助自定义模式，您的 Device Farm 测试执行环境能够与本地测试环境保持一致。

自定义环境是使用 YAML 格式的测试规范（测试规范）文件配置的。Device Farm 为每种支持的测试类型提供了一个默认的测试规范文件，可以按原样使用或自定义；测试筛选条件或配置文件等自定义项可以添加到测试规范中。可以保存编辑后的测试规格，以备将来的测试运行使用。

有关更多信息，请参阅使用 AWS CLI 和 在 Device Farm 中创建测试运行 上传自定义测试规范。

主题

测试规范参考和语法

测试规范（测试规范）是用于在 Device Farm 中定义自定义测试环境的文件。

测试规范工作流程

Device Farm 测试规范按预先确定的顺序执行各阶段及其命令，允许您自定义环境的准备和执行方式。执行每个阶段时，其命令将按照测试规范文件中列出的顺序运行。阶段按以下顺序执行

1. install-应在此处定义下载、安装和设置工具等操作。
2. pre_test-应在此处定义诸如启动后台进程之类的测试前操作。
3. test-应在此处定义调用测试的命令。
4. post_test-应在此处定义测试结束后需要运行的任何最终任务，例如测试报告生成和工件文件聚合。

测试规范语法

以下是测试规范文件的 YAML 架构

```
version: 0.1

android_test_host: "string"
ios_test_host: "string"

phases:
  install:
    commands:
      - "string"
      - "string"
  pre_test:
    commands:
      - "string"
      - "string"
  test:
    commands:
      - "string"
      - "string"
  post_test:
    commands:
      - "string"
```

```
      - "string"

artifacts:
    - "string"
    - "string"
```

version

（必填，数字）

反映 Device Farm 支持的测试规范版本。当前版本号是 `0.1`。

android_test_host

（可选，字符串）

将为在 Android 设备上执行的测试运行选择的测试主机。要在 Android 设备上运行测试，此字段是必填字段。有关更多信息，请参阅 自定义测试环境的可用测试主机。

ios_test_host

（可选，字符串）

将为在 iOS 设备上执行的测试运行选择的测试主机。在主版本大于 26 的 iOS 设备上进行测试时，此字段为必填字段。有关更多信息，请参阅 自定义测试环境的可用测试主机。

phases

本节包含在测试运行期间执行的命令组，其中每个阶段都是可选的。允许的测试阶段名称为：`install`、`pre_test`、`test`、和`post_test`。

- `install`-已经安装了 Device Farm 支持的测试框架的默认依赖项。此阶段包含 Device Farm 在安装期间运行的其他命令（如果有）。
- `pre_test`-在自动测试之前执行的命令（如果有）。
- `test`-在自动测试运行期间执行的命令。如果测试阶段的任何命令失败（这意味着它返回非零的退出代码），则测试将被标记为失败
- `post_test`-在自动测试运行后执行的命令（如果有）。无论您在该`test`阶段的测试成功还是失败，都将执行此操作。

commands

（可选，列表 [字符串]）

该阶段要作为 shell 命令执行的字符串列表。

artifacts

（可选，列表 [字符串]）

Device Farm 会从此处指定的位置收集自定义报告、日志文件和图像等构件。不支持在项目位置中使用通配符，因此，您必须为每个位置指定有效的路径。

这些测试项目可用于测试运行中的每台设备。有关检索测试项目的更多信息，请参阅在自定义测试环境中下载构件。

> **Important**
>
> 测试规范必须格式化为有效的 YAML 文件。如果测试规范中的缩进或间距无效，测试运行可能会失败。YAML 文件中不允许使用制表符。您可以使用 YAML 验证程序测试您的测试规范是否为效的 YAML 文件。有关更多信息，请参阅 YAML 网站。

测试规范示例

以下示例显示了可以在 Device Farm 上执行的测试规范。

Simple Demo

以下是一个测试规范文件示例，该文件仅`Hello world!`作为测试运行工件进行记录。

```
version: 0.1

android_test_host: amazon_linux_2
ios_test_host: macos_sequoia

phases:
  install:
    commands:
      # Setup your environment by installing and/or validating software
      - devicefarm-cli use python 3.11
      - python --version

  pre_test:
    commands:
      # Setup your tests by starting background tasks or setting up
      # additional environment variables.
      - OUTPUT_FILE="/tmp/hello.log"
```

```
  test:
    commands:
      # Run your tests within this phase.
      - python -c 'print("Hello world!")' &> $OUTPUT_FILE

  post_test:
    commands:
      # Perform any remaining tasks within this phase, such as copying
      # artifacts to the DEVICEFARM_LOG_DIR for upload
      - cp $OUTPUT_FILE $DEVICEFARM_LOG_DIR

artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Appium 安卓

以下是配置在 Android 上运行的 Appium Java Testng 测试的测试规范文件示例...

```
version: 0.1

# The following fields(s) allow you to select which Device Farm test host is used
 for your test run.
android_test_host: amazon_linux_2

phases:

  # The install phase contains commands for installing dependencies to run your
 tests.
  # Certain frequently used dependencies are preinstalled on the test host to
 accelerate and
  # simplify your test setup. To find these dependencies, versions supported and
 additional
  # software installation please see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environments-hosts-software.html
  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
```

```
      - devicefarm-cli use node 20
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium.
      - devicefarm-cli use appium 2
      - appium --version

      # The Device Farm service periodically updates the preinstalled Appium
versions over time to
      # incorporate the latest minor and patch versions for each major version. If
you wish to
      # select a specific version of Appium, you can use NPM to install it.
      # - npm install -g appium@2.19.0

      # When running Android tests with Appium version 2, the uiautomator2 driver is
preinstalled using driver
      # version 2.44.1 for Appium 2.5.1  If you want to install a different version
of the driver,
      # you can use the Appium extension CLI to uninstall the existing uiautomator2
driver
      # and install your desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      #   then
      #     appium driver uninstall uiautomator2;
      #     appium driver install uiautomator2@2.34.0;
      #   fi;

      # Based on Appium framework's recommendation, we recommend setting the Appium
server's
      # base path explicitly for accepting commands. If you prefer the legacy base
path of /wd/hub,
      # please set it here.
      - export APPIUM_BASE_PATH=

      # Use the devicefarm-cli to setup a Java environment, with which you can run
your test suite.
      - devicefarm-cli use java 17
      - java -version

  # The pre-test phase contains commands for setting up your test environment.
  pre_test:
    commands:
      # Setup the CLASSPATH so that Java knows where to find your test classes.
```

```
      - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/*
      - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/dependency-jars/*

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how
 they're set, please see
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
          --log-no-colors --relaxed-security --default-capabilities \
          "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
          \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
          \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
          \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
          \"appium:chromedriverExecutableDir\":
 \"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
          \"appium:automationName\": \"UiAutomator2\"}" \
          >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &;

      # This code snippet is to wait until the Appium server starts.
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status";
 do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
 $DEVICEFARM_TEST_PACKAGE_PATH directory.
```

```
      - echo "Navigate to test package directory"
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      - echo "Starting the Appium TestNG test"

      # The following command runs your Appium Java TestNG test.
      # For more information, please see TestNG's documentation here:
      # https://testng.org/#_running_testng
      - |-
        java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG
 -testjar *-tests.jar \
          -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

      # To run your tests with a testng.xml file that is a part of your test
 package,
      # use the following commands instead:

      # - echo "Unzipping the tests JAR file"
      # - unzip *-tests.jar
      # - |-
      #   java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH
 org.testng.TestNG -testjar *-tests.jar \
      #     testng.xml -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

  # The post-test phase contains commands that are run after your tests have
 completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output
 and reports.
# All files in these paths will be collected by Device Farm, with certain limits
 (see limit details
# here: https://docs.aws.amazon.com/devicefarm/latest/developerguide/
limits.html#file-limits).
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Appium iOS

以下是配置在 iOS 上运行的 Appium Java Testng 测试的测试规范文件示例。

```
version: 0.1

# The following fields(s) allow you to select which Device Farm test host is used
 for your test run.
ios_test_host: macos_sequoia

phases:

  # The install phase contains commands for installing dependencies to run your
 tests.
  # Certain frequently used dependencies are preinstalled on the test host to
 accelerate and
  # simplify your test setup. To find these dependencies, versions supported and
 additional
  # software installation please see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environments-hosts-software.html
  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
      - devicefarm-cli use node 20
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium.
      - devicefarm-cli use appium 2
      - appium --version

      # The Device Farm service periodically updates the preinstalled Appium
 versions over time to
      # incorporate the latest minor and patch versions for each major version. If
 you wish to
      # select a specific version of Appium, you can use NPM to install it.
      # - npm install -g appium@2.19.0

      # When running iOS tests with Appium version 2, the XCUITest driver is
 preinstalled using driver
```

```
    # version 9.10.5 for Appium 2.5.4. If you want to install a different version
of the driver,
    # you can use the Appium extension CLI to uninstall the existing XCUITest
driver
    # and install your desired version:
    # - |-
    #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
    #   then
    #     appium driver uninstall xcuitest;
    #     appium driver install xcuitest@10.0.0;
    #   fi;

    # Based on Appium framework's recommendation, we recommend setting the Appium
server's
    # base path explicitly for accepting commands. If you prefer the legacy base
path of /wd/hub,
    # please set it here.
    - export APPIUM_BASE_PATH=

    # Use the devicefarm-cli to setup a Java environment, with which you can run
your test suite.
    - devicefarm-cli use java 17
    - java -version

 # The pre-test phase contains commands for setting up your test environment.
 pre_test:
   commands:
    # Setup the CLASSPATH so that Java knows where to find your test classes.
    - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/*
    - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/dependency-jars/*

    # Device Farm provides multiple pre-built versions of WebDriverAgent (WDA), a
required
    # Appium dependency for iOS, where each version corresponds to the XCUITest
driver version selected.
    # If Device Farm cannot find a corresponding version of WDA for your XCUITest
driver,
    # the latest available version is selected by default.
    - |-
      APPIUM_DRIVER_VERSION=$(appium driver list --installed --json | jq -r
".xcuitest.version" | cut -d "." -f 1);
      CORRESPONDING_APPIUM_WDA=$(env | grep
"DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V${APPIUM_DRIVER_VERSION}")
```

```
       if [[ ! -z "$APPIUM_DRIVER_VERSION" ]] && [[ ! -z
 "$CORRESPONDING_APPIUM_WDA" ]]; then
         echo "Using Device Farm's prebuilt WDA version ${APPIUM_DRIVER_VERSION}.x,
 which corresponds with your driver";
         DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo $CORRESPONDING_APPIUM_WDA |
 cut -d "=" -f2)
       else
         LATEST_SUPPORTED_WDA_VERSION=$(env | grep
 "DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V" | sort -V -r | head -n 1)
         echo "Unknown driver version $APPIUM_DRIVER_VERSION; falling back to the
 Device Farm default version of $LATEST_SUPPORTED_WDA_VERSION";
         DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo
 $LATEST_SUPPORTED_WDA_VERSION | cut -d "=" -f2)
       fi;

     # For iOS versions 16 and below only, the device unique identifier (UDID)
 needs to modified for Appium tests
     # on Device Farm to remove the hypens.
     - |-
       if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ]; then
         DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$DEVICEFARM_DEVICE_UDID;
         if [ $(echo $DEVICEFARM_DEVICE_OS_VERSION | cut -d "." -f 1) -le 16 ];
 then
           DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$(echo $DEVICEFARM_DEVICE_UDID | tr -d
 "-");
         fi;
       fi;

     # We recommend starting the Appium server process in the background using the
 command below.
     # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
     # The environment variables passed as capabilities to the server will be
 automatically assigned
     # during your test run based on your test's specific device.
     # For more information about which environment variables are set and how
 they're set, please see
     # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
     - |-
       appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
         --log-no-colors --relaxed-security --default-capabilities \
         "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
         \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
         \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
```

```
          \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\", \
          \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
          \"appium:derivedDataPath\": \"$DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH\",
\
          \"appium:usePrebuiltWDA\": true, \
          \"appium:automationName\": \"XCUITest\"}" \
          >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &

     # This code snippet is to wait until the Appium server starts.
     - |-
       appium_initialization_time=0;
       until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status";
do
         if [[ $appium_initialization_time -gt 30 ]]; then
           echo "Appium did not start within 30 seconds. Exiting...";
           exit 1;
         fi;
         appium_initialization_time=$((appium_initialization_time + 1));
         echo "Waiting for Appium to start on port 4723...";
         sleep 1;
       done;

 # The test phase contains commands for running your tests.
 test:
   commands:
     # Your test package is downloaded and unpackaged into the
$DEVICEFARM_TEST_PACKAGE_PATH directory.
     - echo "Navigate to test package directory"
     - cd $DEVICEFARM_TEST_PACKAGE_PATH
     - echo "Starting the Appium TestNG test"

     # The following command runs your Appium Java TestNG test.
     # For more information, please see TestNG's documentation here:
     # https://testng.org/#_running_testng
     - |-
       java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG
-testjar *-tests.jar \
         -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

     # To run your tests with a testng.xml file that is a part of your test
package,
     # use the following commands instead:

     # - echo "Unzipping the tests JAR file"
```

```
      # - unzip *-tests.jar
      # - |-
      #   java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH
 org.testng.TestNG -testjar *-tests.jar \
      #     testng.xml -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

  # The post-test phase contains commands that are run after your tests have
 completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output
 and reports.
# All files in these paths will be collected by Device Farm, with certain limits
 (see limit details
# here: https://docs.aws.amazon.com/devicefarm/latest/developerguide/
limits.html#file-limits).
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Appium (Both Platforms)

以下是一个测试规范文件示例，用于配置在安卓和 iOS 上运行的 Appium Java Testng 测试。

```
version: 0.1

# The following fields(s) allow you to select which Device Farm test host is used
 for your test run.
android_test_host: amazon_linux_2
ios_test_host: macos_sequoia

phases:

  # The install phase contains commands for installing dependencies to run your
 tests.
  # Certain frequently used dependencies are preinstalled on the test host to
 accelerate and
```

```
  # simplify your test setup. To find these dependencies, versions supported and
 additional
  # software installation please see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environments-hosts-software.html
  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
      - devicefarm-cli use node 20
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium.
      - devicefarm-cli use appium 2
      - appium --version

      # The Device Farm service periodically updates the preinstalled Appium
 versions over time to
      # incorporate the latest minor and patch versions for each major version. If
 you wish to
      # select a specific version of Appium, you can use NPM to install it.
      # - npm install -g appium@2.19.0

      # When running Android tests with Appium version 2, the uiautomator2 driver is
 preinstalled using driver
      # version 2.44.1 for Appium 2.5.1  If you want to install a different version
 of the driver,
      # you can use the Appium extension CLI to uninstall the existing uiautomator2
 driver
      # and install your desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      #   then
      #     appium driver uninstall uiautomator2;
      #     appium driver install uiautomator2@2.34.0;
      #   fi;

      # When running iOS tests with Appium version 2, the XCUITest driver is
 preinstalled using driver
      # version 9.10.5 for Appium 2.5.4. If you want to install a different version
 of the driver,
```

```
    # you can use the Appium extension CLI to uninstall the existing XCUITest
driver
    # and install your desired version:
    # - |-
    #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
    #   then
    #     appium driver uninstall xcuitest;
    #     appium driver install xcuitest@10.0.0;
    #   fi;

    # Based on Appium framework's recommendation, we recommend setting the Appium
server's
    # base path explicitly for accepting commands. If you prefer the legacy base
path of /wd/hub,
    # please set it here.
    - export APPIUM_BASE_PATH=

    # Use the devicefarm-cli to setup a Java environment, with which you can run
your test suite.
    - devicefarm-cli use java 17
    - java -version

 # The pre-test phase contains commands for setting up your test environment.
 pre_test:
   commands:
    # Setup the CLASSPATH so that Java knows where to find your test classes.
    - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/*
    - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/dependency-jars/*

    # Device Farm provides multiple pre-built versions of WebDriverAgent (WDA), a
required
    # Appium dependency for iOS, where each version corresponds to the XCUITest
driver version selected.
    # If Device Farm cannot find a corresponding version of WDA for your XCUITest
driver,
    # the latest available version is selected by default.
    - |-
      if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ]; then
        APPIUM_DRIVER_VERSION=$(appium driver list --installed --json | jq -r
".xcuitest.version" | cut -d "." -f 1);
        CORRESPONDING_APPIUM_WDA=$(env | grep
"DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V${APPIUM_DRIVER_VERSION}")
        if [[ ! -z "$APPIUM_DRIVER_VERSION" ]] && [[ ! -z
"$CORRESPONDING_APPIUM_WDA" ]]; then
```

```
            echo "Using Device Farm's prebuilt WDA version
 ${APPIUM_DRIVER_VERSION}.x, which corresponds with your driver";
            DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo $CORRESPONDING_APPIUM_WDA
 | cut -d "=" -f2)
          else
            LATEST_SUPPORTED_WDA_VERSION=$(env | grep
 "DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V" | sort -V -r | head -n 1)
            echo "Unknown driver version $APPIUM_DRIVER_VERSION; falling back to the
 Device Farm default version of $LATEST_SUPPORTED_WDA_VERSION";
            DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH=$(echo
 $LATEST_SUPPORTED_WDA_VERSION | cut -d "=" -f2)
          fi;
        fi;

      # For iOS versions 16 and below only, the device unique identifier (UDID)
 needs to modified for Appium tests
      # on Device Farm to remove the hypens.
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ]; then
          DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$DEVICEFARM_DEVICE_UDID;
          if [ $(echo $DEVICEFARM_DEVICE_OS_VERSION | cut -d "." -f 1) -le 16 ];
 then
            DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$(echo $DEVICEFARM_DEVICE_UDID | tr -d
 "-");
          fi;
        fi;

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how
 they're set, please see
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ]; then
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
```

```
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:chromedriverExecutableDir\":
\"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
            \"appium:automationName\": \"UiAutomator2\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        else
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\", \
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:derivedDataPath\": \"$DEVICEFARM_WDA_DERIVED_DATA_PATH\", \
            \"appium:usePrebuiltWDA\": true, \
            \"appium:automationName\": \"XCUITest\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        fi;

      # This code snippet is to wait until the Appium server starts.
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status";
do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
$DEVICEFARM_TEST_PACKAGE_PATH directory.
      - echo "Navigate to test package directory"
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      - echo "Starting the Appium TestNG test"

      # The following command runs your Appium Java TestNG test.
      # For more information, please see TestNG's documentation here:
      # https://testng.org/#_running_testng
```

```
      - |-
        java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG
 -testjar *-tests.jar \
          -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

      # To run your tests with a testng.xml file that is a part of your test
 package,
      # use the following commands instead:

      # - echo "Unzipping the tests JAR file"
      # - unzip *-tests.jar
      # - |-
      #   java -Dappium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH
 org.testng.TestNG -testjar *-tests.jar \
      #     testng.xml -d $DEVICEFARM_LOG_DIR/test-output -verbose 10

  # The post-test phase contains commands that are run after your tests have
 completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output
 and reports.
# All files in these paths will be collected by Device Farm, with certain limits
 (see limit details
# here: https://docs.aws.amazon.com/devicefarm/latest/developerguide/
limits.html#file-limits).
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

用于自定义测试环境的主机

Device Farm 通过使用测试主机环境支持一组带有预配置软件的操作系统。在测试执行期间，Device Farm 使用亚马逊管理的实例（主机），这些实例（主机）动态连接到所选的被测设备。此实例已完全清理，不会在两次运行之间重复使用，并在测试运行结束后以其生成的工件终止。

主题

自定义测试环境的可用测试主机

测试主机完全由 Device Farm 管理。下表列出了用于自定义测试环境的当前可用和支持的 Device Farm 测试主机。

设备平台	测试主机	操作系统	架构	支持的设备
Android	amazon_linux_2	Amazon Linux 2	x86_64	Android6 及以上
iOS	macos_sequoia	macOS Sequoia（第 15 版）	arm64	iOS15 到 26

> **Note**
>
> Device Farm 会定期为设备平台添加新的测试主机，以支持较新的设备操作系统版本及其依赖项。发生这种情况时，相应设备平台的较旧测试主机将终止支持。

操作系统版本

每台可用的测试主机都使用当时 Device Farm 支持的特定版本的操作系统。尽管我们尝试使用最新的操作系统版本，但这可能不是最新的公开发行版本。Device Farm 将使用次要版本更新和安全补丁定期更新操作系统。

要了解测试运行期间使用的操作系统的特定版本（包括次要版本），可以将以下代码片段添加到测试规范文件的任何阶段。

Example

```
phases:
  install:
    commands:
      # The following example prints the instance's operating system version details
      - |-
        if [[ "Darwin" == "$(uname)" ]]; then
          echo "$(sw_vers --productName) $(sw_vers --productVersion) ($(sw_vers --
buildVersion))";
        else
          echo "$(. /etc/os-release && echo $PRETTY_NAME) ($(uname -r))";
        fi
```

为自定义测试环境选择测试主机

您可以在测试规范文件的相应ios_test_host变量中指定 Android android_test_host 和 iOS 测试主机。

如果您没有为给定设备平台指定测试主机选择，则测试将在Device Farm设置为指定设备和测试配置的默认设置的测试主机上运行。

> **Important**
>
> 在 iOS 18 及更低版本上进行测试时，如果未选择主机，则将使用旧版测试主机。有关更多信息，请参阅上的主题旧版 iOS 测试主机。

例如，请查看以下代码片段：

Example

```
version: 0.1
android_test_host: amazon_linux_2
ios_test_host: macos_sequoia

phases:
  # ...
```

自定义测试环境中支持的软件

Device Farm 使用预先安装了许多必需软件库的主机来运行我们服务支持的测试框架，从而在启动时提供现成的测试环境。Device Farm 通过使用我们的软件选择机制支持多种语言，并将定期更新环境中包含的语言版本。

对于任何其他必需的软件，您可以修改测试规范文件以从测试包中安装、从 Internet 下载或访问 VPC 内的私有来源（有关更多信息，请参阅 VPC ENI）。有关更多信息，请参阅 测试规范示例。

预先配置的软件

为了便于在每个平台上进行设备测试，测试主机上提供了以下工具：

工具	设备平台
Android SDK Build-Tools	Android
Android SDK Platform-Tools（包括adb）	Android
Xcode	iOS

可选软件

除了主机上预先配置的软件外，Device Farm 还提供了一种通过`devicefarm-cli`工具选择特定版本的支持软件的方法。

下表包含可选软件和包含这些软件的测试主机。

软件/工具	支持该软件的主机	要在测试规范中使用的命令
Java 17	amazon_linux_2 macos_sequoia	`devicefarm-cli use java 17`
Java 11	amazon_linux_2 macos_sequoia	`devicefarm-cli use java 11`
Java 8	amazon_linux_2	`devicefarm-cli use java 8`

软件/工具	支持该软件的主机	要在测试规范中使用的命令
	macos_sequoia	
Node.js 20	amazon_linux_2 macos_sequoia	`devicefarm-cli use node 20`
Node.js 18	amazon_linux_2 macos_sequoia	`devicefarm-cli use node 18`
Node.js 16	amazon_linux_2	`devicefarm-cli use node 16`
Python 3.11	amazon_linux_2 macos_sequoia	`devicefarm-cli use python 3.11`
Python 3.10	amazon_linux_2 macos_sequoia	`devicefarm-cli use python 3.10`
Python 3.9	amazon_linux_2 macos_sequoia	`devicefarm-cli use python 3.9`
Python 3.8	amazon_linux_2	`devicefarm-cli use python 3.8`
Ruby 3.2	amazon_linux_2 macos_sequoia	`devicefarm-cli use ruby 3.2`
Ruby 2.7	amazon_linux_2	`devicefarm-cli use ruby 2.7`
Appium 3	amazon_linux_2	`devicefarm-cli use appium 3`

软件/工具	支持该软件的主机	要在测试规范中使用的命令
Appium 2	amazon_linux_2 macos_sequoia	`devicefarm-cli use appium 2`
Appium 1	amazon_linux_2	`devicefarm-cli use appium 1`
Xcode 26	macos_sequoia	`devicefarm-cli use xcode 26`
Xcode 16	macos_sequoia	`devicefarm-cli use xcode 16`

测试主机还包括每个软件版本的常用支持工具，例如`pip`和`npm`包管理器（分别包含在 Python 和 Node.js 中）以及 Appium 等工具的依赖项（例如 Appium UIAutomator2 驱动程序）。这可确保您拥有使用支持的测试框架所需的工具。

在自定义测试环境中使用 devicefarm-cli 工具

测试主机使用名为的标准化版本管理工具 `devicefarm-cli`来选择软件版本。此工具与 Device Farm 分开， AWS CLI 且仅在 Device Farm 测试主机上可用。使用 `devicefarm-cli`，您可以切换到测试主机上预安装的任何软件版本。这提供了一种随时间推移维护 Device Farm 测试规范文件的简单方法，并为您提供了将来升级软件版本的可预测机制。

> **Important**
>
> 此命令行工具在旧版 iOS 主机上不可用。有关更多信息，请参阅上的主题旧版 iOS 测试主机。

以下片段显示了 `devicefarm-cli` 的 `help` 页面：

```
$ devicefarm-cli help
 Usage: devicefarm-cli COMMAND [ARGS]

     Commands:
         help                          Prints this usage message.
         list                          Lists all versions of software configurable
                                       via this CLI.
```

```
        use <software> <version>     Configures the software for usage within the
                                     current shell's environment.
```

让我们来看几个使用 devicefarm-cli 的示例。要使用该工具将测试规范文件*3.9*中的 Python 版本从*3.10*更改为，请运行以下命令：

```
$ python --version
Python 3.10.12
$ devicefarm-cli use python 3.9
$ python --version
Python 3.9.17
```

要将 Appium 版本从*1*更改为，请执行以下操作：*2*

```
$ appium --version
1.22.3
$ devicefarm-cli use appium 2
$ appium --version
2.1.2
```

> **Tip**
>
> 请注意，在选择软件版本时，devicefarm-cli 还会切换这些语言的支持工具，例如适用于 Python 的 pip 和适用于 NodeJS 的 npm。

有关测试主机上预安装的软件的更多信息，请参见自定义测试环境中支持的软件。

安卓设备的测试环境

AWS Device Farm 利用运行 Amazon Linux 2 的 Amazon Elastic Compute Cloud (EC2) 主机来执行 Android 测试。当您安排测试运行时，Device Farm 会为每台设备分配一台专属主机，以独立运行测试。在测试运行后，主机将与生成的构件一起终止运行。

Amazon Linux 2 主机具有以下优点：

- 更快、更可靠的测试：与传统主机相比，新的测试主机显著提高了测试速度，尤其是缩短了测试开始时间。Amazon Linux 2 主机在测试期间还表现出更高的稳定性和可靠性。
- 增强了用于手动测试的远程访问功能：升级到最新的测试主机并进行了改进，从而降低了延迟，提高了 Android 手动测试的视频性能。

- 标准软件版本选择：Device Farm 现在标准化了测试主机上的主要编程语言支持以及 Appium 框架版本。对于支持的语言（目前是 Java、Python、Node.js 和 Ruby）和 Appium，新的测试主机在发布后不久就会提供长期稳定的版本。通过 `devicefarm-cli` 工具进行集中式版本管理可实现跨框架一致的测试规范文件开发。

主题

Device Farm 中 Amazon Linux 2 测试环境支持的 IP 范围

客户通常需要知道 Device Farm 流量来源的 IP 范围，尤其是在配置防火墙和安全设置时。对于 Amazon EC2 测试主机，IP 范围包括整个 `us-west-2` 区域。对于 Amazon Linux 2 测试主机（新 Android 运行的默认选项），IP 范围已受到限制。现在，流量来自一组特定的 NAT 网关，将 IP 范围限制为以下地址：

IP 范围
44.236.137.143
52.13.151.244
52.35.189.191
54.201.250.26

有关 Device Farm 中 Android 测试环境的更多信息，请参阅 安卓设备的测试环境。

iOS 设备的测试环境

Device Farm 使用亚马逊管理的 macOS 实例（主机），这些实例（主机）在测试运行期间动态连接到 iOS 设备。每台主机都预先配置了软件，可以在各种流行的测试平台（例如 XCTest用户界面和 Appium）上进行设备测试。

与之前的版本相比，iOS 测试主机的当前版本在测试体验的基础上有所改进，包括：

- iOS 15 到 iOS 26 的一致主机操作系统和工具体验以前，测试主机是由使用的设备决定的，因此在多个 iOS 版本上执行时会导致软件环境分散。当前的体验允许简单的主机选择，从而实现跨设备一致的环境。这将使每台 iOS 设备都可以使用相同的 macOS 版本和工具（例如 Xcode）。

- iOS 15 和 16 测试的性能改进使用更新的基础架构，iOS 15 和 16 测试的设置时间已大大缩短。
- 支持依赖项的标准化可选软件版本我们现在在 iOS 和 Android 测试主机上都`devicefarm-cli`安装了软件选择系统，使您能够选择我们支持的依赖项的首选版本。对于支持的依赖项（例如 Java、Python、Node.js、Ruby 和 Appium），可以通过测试规范选择版本。要了解此功能的工作原理，请参阅上的主题自定义测试环境中支持的软件。

> **Important**
>
> 如果在 iOS 18 及更低版本上执行，则默认情况下，您的测试将在旧版测试主机上执行。有关如何迁移出旧版主机的信息，请参阅以下主题。

旧版 iOS 测试主机

对于 iOS 18 及更低版本上的现有测试，默认情况下会为自定义测试环境选择传统测试主机。下表包含由 iOS 设备版本执行的测试主机版本。

操作系统	架构	设备的默认值
macOS Sonoma（第 14 版）	arm64	iOS 18
macOS Ventura（第 13 版）	arm64	iOS 17
macOS Monterey（第 12 版）	x86_64	iOS 16及以下

要选择较新的测试主机，请参阅相关主题将您的自定义测试环境迁移到新的 iOS 测试主机。

iOS 设备支持的软件

为了支持 iOS 设备测试，适用于 iOS 设备的 Device Farm 测试主机预先配置了 Xcode 及其相关的命令行工具。有关其他可用软件，请查看相关主题自定义测试环境中支持的软件。

将您的自定义测试环境迁移到新的 iOS 测试主机

要将现有测试从旧版主机迁移到新的 macOS 测试主机，您需要根据先前存在的测试规范文件开发新的测试规范文件。

推荐的方法是从所需测试类型的示例测试规范文件开始，然后将相关命令从旧的测试规范文件迁移到新的测试规范文件中。这使您可以利用新主机的示例测试规范的新功能和优化，同时重复使用现有代码片段。

主题

教程：使用控制台迁移 iOS 测试规范文件

在此示例中，Device Farm 控制台将用于载入现有 iOS 设备测试规范，以使用新的测试主机。

步骤 1：使用控制台创建新的测试规范文件

1. 登录 AWS Device Farm 控制台。
2. 导航到包含您的自动化测试的 Device Farm 项目。
3. 下载一份您希望加入的现有测试规范的副本。
   a. 单击 “项目设置” 选项，然后导航到 “上传” 选项卡。
   b. 导航到您要使用的测试规范文件。
   c. 单击 “下载” 按钮制作此文件的本地副本。
4. 返回到 “项目” 页面，然后单击 “创建”、“运行”。
5. 填写向导上的选项，就像开始新的运行一样，但在 “选择测试规范” 选项处停下来。
6. 使用默认选择的 iOS 测试规范，单击 “创建测试规范” 按钮。
7. 修改文本编辑器中默认选择的测试规范。
   a. 如果尚不存在，请使用以下命令修改测试规范文件以选择新主机：

      ```
      ios_test_host: macos_sequoia
      ```

   b. 从上一步中下载的测试规范副本中，查看每份规范 `phase`。
   c. 将旧测试规范阶段的命令复制到新测试规范的每个相应阶段，忽略与安装或选择 Java、Python、Node.js、Ruby、Appium 或 Xcode 相关的命令。
8. 在另存为文本框中输入新的文件名。
9. 单击 “另存为新内容” 按钮以保存您的更改。

有关可用作参考的测试规范文件的示例，请参阅中提供的示例测试规范示例。

第 2 步：选择软件预安装的软件

在新的测试主机中，使用名devicefarm-cli为的新标准化版本管理工具选择预安装的软件版本。现在，推荐使用此工具来使用我们在测试主机上提供的各种软件。

例如，您可以添加以下行来使用不同的 JDK 17 测试环境：

```
- devicefarm-cli use java 17
```

有关可用软件支持的更多信息，请查看：自定义测试环境中支持的软件。

第 3 步：通过软件选择工具使用 Appium 及其依赖项

新的测试主机仅支持 Appium 2.x 及更高版本。请使用明确选择 Appium 版本devicefarm-cli，同时移除传统工具，例如。 avm例如：

```
# This line using 'avm' should be removed
# - avm 2.3.1

# And the following lines should be added
- devicefarm-cli use appium 2 # Selects the version
- appium --version            # Prints the version
```

所选的 Appium 版本预devicefarm-cli装了适用于 iOS 的驱动程序 XCUITest 的兼容版本。

此外，你还需要更新测试规范才能使用， DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V9而不是 DEVICEFARM_WDA_DERIVED_DATA_PATH。新的环境变量指向 WebDriverAgent 9.x 的预构建版本，这是 Appium 2 测试支持的最新版本。

欲了解更多信息，请查看为 iOS 测试选择 WebDriverAgent 版本和Appium 测试的环境变量。

新测试主机和旧版测试主机之间的差异

在编辑测试规范文件以使用新的 iOS 测试主机并从旧版测试主机过渡测试时，请注意以下主要环境差异：

- Xcode 版本：在旧版测试主机环境中，可用的 Xcode 版本基于用于测试的设备的 iOS 版本。例如，在 iOS 18 设备上进行的测试在旧版主机上使用 Xcode 16，而在 iOS 17 上进行的测试则使用 Xcode

15。在新的主机环境中，所有设备都可以访问相同版本的 Xcode，从而为不同版本的设备上的测试提供了一致的环境。有关当前可用的 Xcode 版本的列表，请参阅支持的软件。

- 选择软件版本：在许多情况下，默认软件版本已更改，因此，如果您之前没有在旧版测试主机中明确选择软件版本，则可能需要使用立即在新测试主机中指定该版本`devicefarm-cli`。在绝大多数用例中，我们建议客户明确选择他们使用的软件版本。通过选择软件版本，`devicefarm-cli`您将获得可预测且一致的使用体验，并且如果 Device Farm 计划从测试主机中删除该版本，则会收到大量警告。

  此外，诸如 `nvm`、`pyenv`、 `avm` 和 `rvm` 之类的软件选择工具已被删除，取而代之的是新的 `devicefarm-cli` 软件选择系统。

- 可用的软件版本：以前预安装的软件的许多版本已被删除，并添加了许多新版本。因此，请确保在使用 `devicefarm-cli` 选择软件版本时，选择支持的版本列表中的版本。

- 该工具**libimobiledevice**套件已被删除，转而使用较新的/第一方工具来跟踪当前的iOS设备测试和行业标准。对于 iOS 17 及更高版本，你可以迁移大部分命令以使用类似的 Xcode 工具，名为`devicectl`。有关信息`devicectl`，可以在安装了 Xcode 的计算机上运行`xcrun devicectl help`。

- 在旧版主机测试规范文件中硬编码为绝对路径的文件路径很可能无法在新测试主机中按预期工作，并且通常不建议将其用于测试规范文件。我们建议您对所有测试规范文件代码使用相对路径和环境变量。有关更多信息，请查看上的主题执行自定义测试环境的最佳实践。

- 操作系统版本和架构：传统测试主机根据分配的设备使用各种 macOS 版本和 CPU 架构。因此，用户可能会注意到环境中可用的系统库存在一些差异。有关先前主机操作系统版本的更多信息，请查看旧版 iOS 测试主机。

- 对于 Appium 用户，选择的方式 WebDriverAgent 已更改为使用环境变量前缀， `DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V`而不是旧`DEVICEFARM_WDA_DERIVED_DATA_PATH_V`的前缀。有关更新后的变量的更多信息，请查看Appium 测试的环境变量。

- 对于 Appium Java 用户，新的测试主机的类路径中不包含任何预安装的 JAR 文件，而之前的主机包含一个 TestNG 框架的 JAR 文件（通过环境变量）。`$DEVICEFARM_TESTNG_JAR`我们建议客户将测试框架所必需的 JAR 文件打包到测试包中，并从测试规范文件中删除 `$DEVICEFARM_TESTNG_JAR` 变量的实例。

如果您从软件角度对测试主机之间的差异有任何反馈或疑问，我们建议您通过支持案例与服务团队联系。

使用 IAM 执行角色访问 AWS 资源

Device Farm 支持指定将在测试执行期间由自定义测试运行时环境担任的 IAM 角色。此功能允许您的测试安全地访问您账户中的 AWS 资源，例如 Amazon S3 存储桶、DynamoDB 表或您的应用程序所依赖的其他 AWS 服务。

主题

概述

当您指定 IAM 执行角色时，Device Farm 将在测试执行期间担任此角色，从而允许您的测试使用角色中定义的权限与 AWS 服务进行交互。

IAM 执行角色的常见用例包括：

- 访问存储在 Amazon S3 存储桶中的测试数据
- 将测试项目推送到 Amazon S3 存储桶
- 从 AWS 检索应用程序配置 AppConfig
- 将测试日志和指标写入 Amazon CloudWatch
- 向 Amazon SQS 队列发送测试结果或状态消息
- 在测试工作流程中调用 AWS Lambda 函数

IAM 角色要求

要在 Device Farm 中使用 IAM 执行角色，您的角色必须满足以下要求：

- 信任关系：必须信任 Device Farm 服务主体才能担任该角色。信任策略必须包含`devicefarm.amazonaws.com`为可信实体。
- 权限：该角色必须具有访问测试所需的 AWS 资源所需的必要权限。

- 会话时长：角色的最大会话持续时间必须至少等于您的 Device Farm 项目的作业超时设置。默认情况下，Device Farm 项目的作业超时时间为 150 分钟，因此您的角色必须支持至少 150 分钟的会话持续时间。
- 相同的账户要求：IAM 角色必须与用于调用 Device Farm 的角色位于相同的 AWS 账户中。不支持跨账户角色假设。
- PassRole 权限：必须通过允许对指定执行角色执行`iam:PassRole`操作的策略授权调用方传递 IAM 角色。

示例信任策略

以下示例显示了允许 Device Farm 担任您的执行角色的信任策略。此信任策略应仅附加到您打算在 Device Farm 中使用的特定 IAM 角色，而不应附加到您账户中的其他角色：

Example

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "devicefarm.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

权限策略示例

以下示例显示了一个权限策略，该策略授予对测试中使用的常见 AWS 服务的访问权限：

Example

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::my-test-bucket",
        "arn:aws:s3:::my-test-bucket/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "appconfig:GetConfiguration",
        "appconfig:StartConfigurationSession"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:log-group:/devicefarm/test-*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage",
        "sqs:GetQueueUrl"
      ],
      "Resource": "arn:aws:sqs:*:*:test-results-*"
    }
  ]
}
```

配置 IAM 执行角色

您可以在项目级别或为单个测试运行指定 IAM 执行角色。在项目级别进行配置后，该项目中的所有运行都将继承执行角色。在运行时配置的执行角色将取代其父项目上配置的任何执行角色。

有关配置执行角色的详细说明，请参阅：

- 在 AWS Device Farm 中创建项目-用于在项目级别配置执行角色
- 在 Device Farm 中创建测试运行-用于为单个运行配置执行角色

您也可以使用 Device Farm API 配置执行角色。如需了解更多信息，请参阅 Dev ice Farm API 参考。

最佳实践

在为 Device Farm 测试配置 IAM 执行角色时，请遵循以下最佳实践：

- 最低权限原则：仅授予测试运行所需的最低权限。避免使用过于宽泛的权限，例如*操作或资源。
- 使用特定资源的权限：如果可能，请将权限限制为特定资源（例如特定的 S3 存储桶或 DynamoDB 表），而不是某一类型的所有资源。
- 将测试资源和生产资源分开：使用专用的测试资源和角色，以避免在测试期间意外影响生产系统。
- 定期审查角色：定期审查和更新您的执行角色，以确保他们仍然满足您的测试需求并遵循安全最佳实践。
- 使用条件键：考虑使用 IAM 条件键来进一步限制角色的使用时间和方式。

问题排查

如果您遇到与 IAM 执行角色有关的问题，请检查以下内容：

- 信任关系：验证角色的信任策略是否包含`devicefarm.amazonaws.com`为可信服务。
- 权限：检查该角色是否具有您的测试尝试访问的 AWS 服务的必要权限。
- 测试日志：查看测试执行日志，了解与 AWS API 调用或权限拒绝相关的特定错误消息。

自定义测试环境的环境变量

Device Farm 动态配置多个环境变量，以便在自定义测试环境运行中使用。

主题

自定义环境变量

Device Farm 支持配置在测试主机上作为环境变量应用的键值对。这些变量可以在 Device Farm 项目上配置，也可以在运行创建过程中进行配置；运行时配置的任何变量都将取代其父项目上可能配置的任何变量。以下限制适用：

- 旧版 iOS 测试主机不支持自定义环境变量。有关更多信息，请参阅 旧版 iOS 测试主机。
- 以开头的$DEVICEFARM_变量名保留供内部服务使用。
- 自定义环境变量不得用于在测试规范中配置测试主机计算选择。

常用环境变量

本节介绍了 Device Farm 中所有测试共有的环境变量。

$DEVICEFARM_DEVICE_NAME

运行测试的设备。它代表设备的唯一设备标识符 (UDID)。

$DEVICEFARM_DEVICE_UDID

设备的唯一标识符。

$DEVICEFARM_DEVICE_PLATFORM_NAME

设备的平台名称。要么是要Android么iOS。

$DEVICEFARM_DEVICE_OS_VERSION

设备的操作系统版本。

$DEVICEFARM_APP_PATH

（移动应用程序测试）

在其上执行测试的主机上移动应用程序的路径。此变量在 Web 测试期间不可用。

$DEVICEFARM_LOG_DIR

默认目录的路径，存储客户日志、工件和其他所需文件以供日后检索。使用示例测试规范，此目录中的文件存档在 ZIP 文件中，并在测试运行后作为构件提供。

$DEVICEFARM_SCREENSHOT_PATH

测试运行期间捕获的屏幕截图（如果有）的路径。

$DEVICEFARM_PROJECT_ARN

作业的上级项目的 ARN。

$DEVICEFARM_RUN_ARN

作业的父项运行的 ARN。

$DEVICEFARM_DEVICE_ARN

被测设备的 ARN。

$DEVICEFARM_TOTAL_JOBS

与其父 Device Farm 运行相关的任务总数。

$DEVICEFARM_JOB_NUMBER

这份工作的编号在里面$DEVICEFARM_TOTAL_JOBS。例如，一次运行可能包含 5 个作业，每个任务都有一个$DEVICEFARM_JOB_NUMBER从 0 到 4 的唯一任务。

$AWS_REGION

AWS 区域。该服务会将其设置为与被测设备所在的区域相匹配。如果需要，它可以被自定义环境变量覆盖。

$ANDROID_HOME

（仅限 Android 设备）

Android 软件开发工具包安装目录的路径。

Appium 测试的环境变量

本节介绍任何 Appium 测试在 Device Farm 的自定义测试环境中使用的环境变量。

$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR

（仅限 Android 设备）

包含在 Appium Web 和 ChromeDriver 混合测试中使用的必要可执行文件的目录的位置。

$DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V<N>

（仅限 iOS）

为在 Device Farm 上运行 WebDriverAgent 而构建的版本的派生数据路径。变量上的编号将对应于的主要版本。 WebDriverAgent举个例子，DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V9将指向 9.x WebDriverAgent 的 a 版本。有关更多信息，请参阅 为 iOS 测试选择 WebDriverAgent 版本。

> **Note**
>
> $DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V<N>环境变量仅存在于非旧版 iOS 主机上。有关更多信息，请参阅 旧版 iOS 测试主机。

$DEVICEFARM_WDA_DERIVED_DATA_PATH_V9

（仅限 iOS，已弃用）

为在 Device Farm 上运行 WebDriverAgent 而构建的版本的派生数据路径。有关替换命名方案，请参阅。$DEVICEFARM_APPIUM_WDA_DERIVED_DATA_PATH_V<N>

XCUITest 测试的环境变量

本节介绍 XCUITest 测试在 Device Farm 的自定义测试环境中使用的环境变量。

$DEVICEFARM_XCUITESTRUN_FILE

Device Farm `.xctestun` 文件的路径。它是从您的应用程序和测试程序包生成的。

$DEVICEFARM_DERIVED_DATA_PATH

Device Farm xcodebuild 输出的预期路径。

$DEVICEFARM_XCTEST_BUILD_DIRECTORY

测试程序包文件的解压缩内容的路径。

执行自定义测试环境的最佳实践

以下主题涵盖了在 Device Farm 中使用自定义测试执行的推荐最佳实践。

运行配置

- 尽可能@@ 依靠 Device Farm 托管软件和 API 功能进行运行配置，而不是通过测试规范文件中的 shell 命令应用类似的配置。这包括测试主机和设备的配置，因为这将在测试主机和设备之间更具可持续性和一致性。

  虽然 Device Farm 鼓励您根据需要自定义测试规范文件以运行测试，但随着时间的推移，随着更多自定义命令的添加，测试规范文件可能会变得难以维护。使用 Device Farm 托管软件（通过类似工具 `devicefarm-cli`和中的默认可用工具`$PATH`），并使用托管功能（如`deviceProxy`请求参数），通过将维护责任转移给 Device Farm 本身来简化测试规范文件。

测试规范和测试包代码

- 请勿在测试规范文件或测试包代码中使用绝对路径或依赖特定的次要版本。Device Farm 会将例行更新应用于选定的测试主机及其随附的软件版本。使用特定或绝对路径（例如 `/usr/local/bin/python`代替`python`）或要求特定的次要版本（例如`Node.js 20.3.1`而不仅仅是 `20`）可能会导致您的测试无法找到所需的可执行文件/文件。

  作为自定义测试执行的一部分，Device Farm 会设置各种环境`$PATH`变量和变量，以确保测试在我们的动态环境中获得一致的体验。有关更多信息，请参阅 自定义测试环境的环境变量 和 自定义测试环境中支持的软件。
- 在测试运行期间，将生成或复制的文件保存在临时目录中。今天，我们确保用户在测试执行期间可以访问临时目录 (`/tmp`)（除了托管目录，例如`$DEVICEFARM_LOG_DIR`）。用户有权访问的其他目录可能会随着时间的推移而发生变化，这取决于所使用的服务或操作系统的需要。
- 将您的测试执行日志保存到**`$DEVICEFARM_LOG_DIR`**。这是为您的执行提供的默认构件目录，用于向其中添加执行日志/工件。默认情况下，我们提供的每个示例测试规范都使用此目录存放工件。
- 在测试规范`test`阶段，确保命令在失败时返回非零代码。我们通过检查该`test`阶段调用的每个 shell 命令的非零退出代码来确定您的执行是否失败。您应确保您的逻辑或测试框架将为所有所需场景返回非零的退出代码，这可能需要额外的配置。

  例如，某些测试框架（例如 JUnit5）不认为零测试运行失败，这会导致即使未执行任何测试也能检测到您的测试已成功运行。以此 JUnit5 为例，您需要指定命令行选项，以确保此场景`--fail-if-no-tests`以非零的退出代码退出。

- 查看软件与将用于测试运行的设备操作系统版本和测试主机版本的兼容性。例如，测试软件框架（例如：Appium）中的某些功能可能无法在被测试设备的所有操作系统版本上按预期运行。

安全性

- 避免在测试规范文件中存储或记录敏感变量（如 AWS 密钥）。测试规范文件、测试规范生成的脚本和测试规范脚本的日志都作为可下载的工件在测试执行结束时提供。这可能会导致您账户中对您的测试运行具有读取权限的其他用户意外泄露机密。

将测试从标准测试环境迁移到自定义测试环境

在 AWS Device Farm 中，您可以从标准测试执行模式切换到自定义执行模式。迁移主要涉及两种不同的执行模式：

1. 标准模式：此测试执行模式主要用于为客户提供精细的报告和完全托管的环境。
2. 自定义模式：此测试执行模式专为不同的用例而构建，这些用例需要更快的测试运行、提升和转换能力、并能实现与本地环境的平衡，以及实时视频流。

有关 Device Farm 中标准模式和自定义模式的更多信息，请参阅 AWS Device Farm 中的测试环境和 AWS Device Farm 中的自定义测试环境。

迁移时的注意事项

本节列出了迁移到自定义模式时需要考虑的一些重要用例：

1. 速度：在标准执行模式下，Device Farm 使用特定框架的打包说明解析您已打包并上传的测试的元数据。解析会检测软件包中的测试数量。之后，Device Farm 将分别运行每项测试，并分别显示每项测试的日志、视频和其他结果构件。但是，这会稳步增加总 end-to-end测试执行时间，因为服务端有测试和结果工件的预处理和后处理。

   相比之下，自定义执行模式不会解析您的测试包；这意味着无需对测试或结果构件进行预处理和最少的后期处理。这会使总 end-to-end执行时间接近您的本地设置。测试的执行格式与在本地计算机上运行时的格式相同。测试结果与您在本地获得的结果相同，可在任务执行结束时下载。
2. 自定义或灵活性：标准执行模式解析您的测试包以检测测试数量，然后分别运行每个测试。请注意，不能保证测试会按照您指定的顺序运行。因此，需要特定执行顺序的测试可能无法按预期运行。此外，无法自定义主机环境或传递以某种方式运行测试所需的配置文件。

相比之下，自定义模式允许您配置主机环境，包括安装其他软件、将筛选条件传递给测试、传递配置文件以及控制测试执行设置。它通过一个 yaml 文件（也称为 testspec 文件）来实现这一点，您可以通过向其中添加 shell 命令来修改该文件。此 yaml 文件被转换为在测试主机上执行的 shell 脚本。您可以保存多个 yaml 文件，并在安排运行时根据需要动态选择一个。

3. 直播视频和日志：标准和自定义执行模式均可为您提供测试所需的视频和日志。但是，在标准模式下，只有在测试完成后才能获得测试的视频和预定义日志。

   相比之下，自定义模式为您提供测试视频和客户端日志的实时流。此外，您还可以在测试结束时下载视频和其他构件。

> **Tip**
>
> 如果您的用例至少涉及上述因素之一，我们强烈建议您切换到自定义执行模式。

迁移步骤

要从标准模式迁移到自定义模式，请执行以下操作：

1. 登录 AWS 管理控制台 并打开 Device Farm 控制台，网址为https://console.aws.amazon.com/devicefarm/。
2. 选择您的项目，然后启动新的自动化运行。
3. 上传您的应用程序（或选择 `web app`），选择您的测试框架类型，上传您的测试包，然后在“`Choose your execution environment`”参数下选择选项以`Run your test in a custom environment`。
4. 默认情况下，将显示 Device Farm 的示例测试规范文件以供您查看和编辑。此示例文件可用作在自定义环境模式下试用测试的起点。然后，在控制台确认测试运行正常后，您可以更改与 Device Farm 的任何 API、CLI 和管道集成，以便在安排测试运行时使用此测试规范文件作为参数。有关如何添加测试规范文件作为运行参数的信息，请参阅我们的《API 指南》中有关 `ScheduleRun` API 的 `testSpecArn` 参数部分。https://docs.aws.amazon.com/devicefarm/latest/APIReference/API_ScheduleRun.html

Appium 框架

在自定义测试环境中，Device Farm 不会在 Appium 框架测试中插入或覆盖任何 Appium 功能。您必须在测试规范 YAML 文件或测试代码中指定测试的 Appium 功能。

Android Instrumentation

您不需要执行任何更改，即可将 Android Instrumentation 测试迁移到自定义测试环境。

iOS XCUITest

您无需进行更改即可将 iOS XCUITest 测试移至自定义测试环境。

在 Device Farm 中扩展自定义测试环境

AWS Device Farm 支持配置用于自动测试的自定义环境（自定义模式），这是针对所有 Device Farm 用户的推荐方法。Device Farm 自定义模式支持运行的不仅限于测试套件。在本节中，您将学习如何扩展测试套件和优化测试。

有关 Device Farm 中自定义测试环境的更多信息，请参阅 AWS Device Farm 中的自定义测试环境。

主题

在 Device Farm 中运行测试时设置设备 PIN

某些应用程序要求您在设备上设置 PIN。Device Farm 不支持在设备上原生设置 PIN。但是，这是可能的，但需要注意以下几点：

- 设备必须运行 Android 8 或更高版本。
- 测试完成后必须删除 PIN。

要在测试中设置 PIN，请使用 `pre_test` 和 `post_test` 阶段来设置和删除 PIN，如下所示：

```
phases:
    pre_test:
      - # ... among your pre_test commands
      - DEVICE_PIN_CODE="1234"
      - adb shell locksettings set-pin "$DEVICE_PIN_CODE"
    post_test:
      - # ... Among your post_test commands
      - adb shell locksettings clear --old "$DEVICE_PIN_CODE"
```

开始您的测试套件时，将设置 PIN 1234。退出测试套件后，PIN 将被删除。

> **Warning**
>
> 如果您在测试完成后没有从设备上删除 PIN 码，则设备和您的账户将被隔离。

有关扩展测试套件和优化测试的更多方法，请参阅 在 Device Farm 中扩展自定义测试环境。

通过所需的功能加快 Device Farm 中基于 Appium 的测试

使用 Appium 时，您可能会发现标准模式测试套件非常慢。这是因为 Device Farm 会应用默认设置，并且不会对您希望如何使用 Appium 环境做出任何假设。虽然这些默认值是围绕行业最佳实践建立的，但它们可能不适用于您的情况。要微调 Appium 服务器的参数，可以在测试规范中调整默认的 Appium 功能。例如，以下内容在 iOS 测试套件中将 `usePrebuildWDA` 功能设置为 `true`，以加快初始启动时间：

```
phases:
  pre_test:
    - # ... Start up Appium
    - >-
    appium --log-timestamp
    --default-capabilities "{\"usePrebuiltWDA\": true, \"derivedDataPath\":
\"$DEVICEFARM_WDA_DERIVED_DATA_PATH\",
    \"deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \"platformName\":
\"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \"app\":\"$DEVICEFARM_APP_PATH\",
    \"automationName\":\"XCUITest\", \"udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\",
 \"platformVersion\":\"$DEVICEFARM_DEVICE_OS_VERSION\"}"
    >> $DEVICEFARM_LOG_DIR/appiumlog.txt 2>&1 &
```

Appium 功能必须是经过 shell 转义的、带引号的 JSON 结构。

以下 Appium 功能是性能改进的常见来源：

`noReset` 和 `fullReset`

这两种功能是相互排斥的，它们描述了 Appium 在每个会话完成后的行为。当 `noReset` 设置为 `true` 时，Appium 服务器不会在 Appium 会话结束时从应用程序中删除数据，实际上不会进行任何清理。`fullReset` 会在会话关闭后从设备中卸载并清除所有应用程序数据。有关更多信息，请参阅 Appium 文档中的 Reset Strategies。

`ignoreUnimportantViews`（仅限 Android）

指示 Appium 将 Android 界面层次结构仅压缩为测试的相关视图，从而加快某些元素的查找速度。但是，这可能会破坏一些 XPath基于测试套件，因为界面布局的层次结构已更改。

`skipUnlock`（仅限 Android）

通知 Appium 当前没有设置 PIN 码，这样可以在屏幕关闭事件或其他锁定事件发生后加快测试速度。

`webDriverAgentUrl`（仅限 iOS）

指示 Appium 假设一个基本的 iOS 依赖项 `webDriverAgent` 已经在运行，并且可以在指定 URL 上接受 HTTP 请求。如果 `webDriverAgent` 尚未启动并运行，Appium 在测试套件开始时可能需要一些时间才能启动 `webDriverAgent`。如果您自己启动 `webDriverAgent` 并在启动 Appium 时将 `webDriverAgentUrl` 设置为 `http://localhost:8100`，则可以更快地启动测试套件。请注意，切勿将此功能与 `useNewWDA` 功能一起使用。

您可以使用以下代码通过设备本地端口 `8100` 上的测试规范文件启动 `webDriverAgent`，然后将其转发到测试主机的本地端口`8100`（这允许您将 `webDriverAgentUrl` 的值设置为 `http://localhost:8100`）。在定义了任何用于设置 Appium 和 `webDriverAgent` 环境变量的代码之后，应在安装阶段运行此代码：

```
      # Start WebDriverAgent and iProxy
      - >-
        xcodebuild test-without-building -project /usr/local/avm/versions/
$APPIUM_VERSION/node_modules/appium/node_modules/appium-webdriveragent/
WebDriverAgent.xcodeproj
```

```
        -scheme WebDriverAgentRunner -derivedDataPath
 $DEVICEFARM_WDA_DERIVED_DATA_PATH
        -destination id=$DEVICEFARM_DEVICE_UDID_FOR_APPIUM
 IPHONEOS_DEPLOYMENT_TARGET=$DEVICEFARM_DEVICE_OS_VERSION
        GCC_TREAT_WARNINGS_AS_ERRORS=0 COMPILER_INDEX_STORE_ENABLE=NO >>
 $DEVICEFARM_LOG_DIR/webdriveragent_log.txt 2>&1 &

        iproxy 8100 8100 >> $DEVICEFARM_LOG_DIR/iproxy_log.txt 2>&1 &
```

然后，您可以将以下代码添加到测试规范文件中，以确保 webDriverAgent 成功启动。在确保 Appium 成功启动后，应在预测试阶段结束时运行此代码：

```
      # Wait for WebDriverAgent to start
      - >-
        start_wda_timeout=0;
        while [ true ];
        do
          if [ $start_wda_timeout -gt 60 ];
          then
              echo "WebDriverAgent server never started in 60 seconds.";
              exit 1;
          fi;
          grep -i "ServerURLHere" $DEVICEFARM_LOG_DIR/webdriveragent_log.txt >> /
dev/null 2>&1;
          if [ $? -eq 0 ];
          then
              echo "WebDriverAgent REST http interface listener started";
              break;
          else
              echo "Waiting for WebDriverAgent server to start. Sleeping for 1
 seconds";
              sleep 1;
              start_wda_timeout=$((start_wda_timeout+1));
          fi;
        done;
```

有关 Appium 支持的功能的更多信息，请参阅 Appium 文档中的 Appium Desired Capabilities。

有关扩展测试套件和优化测试的更多方法，请参阅 在 Device Farm 中扩展自定义测试环境。

在 Device Farm 中运行测试 APIs 后使用 Webhook 和其他内容

在每个测试套件使用 curl 完毕后，您可以让 Device Farm 调用 webhook。执行此操作的过程因目的地和格式而异。对于您的特定 webhook，请参阅该 webhook 的文档。以下示例会在每次测试套件完成时向 Slack webhook 发布一条消息：

```
phases:
  post_test:
    - curl -X POST -H 'Content-type: application/json' --data '{"text":"Tests on
 '$DEVICEFARM_DEVICE_NAME' have finished!"}' https://hooks.slack.com/services/
T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXX
```

有关将 webhook 与 Slack 结合使用的更多信息，请参阅 Slack API 参考中的 Sending your first Slack message using Webhook。

有关扩展测试套件和优化测试的更多方法，请参阅 在 Device Farm 中扩展自定义测试环境。

您不仅局限于使用 curl 来调用 webhook。测试包可以包含额外的脚本和工具，前提是它们与 Device Farm 执行环境兼容。例如，您的测试包可能包含向其他人发出请求的辅助脚本 APIs。确保将所有必需的软件包与测试套件的要求一起安装。要添加在测试套件完成后运行的脚本，请将该脚本包含在测试包中，并将以下内容添加到测试规范中：

```
phases:
  post_test:
    - python post_test.py
```

> **Note**
>
> 您有责任维护测试包中使用的任何 API 密钥或其他身份验证令牌。我们建议您将任何形式的安全凭证排除在源代码控制之外，使用权限尽可能少的证书，并尽可能使用可撤销的短期令牌。要验证安全要求，请参阅您 APIs 使用的第三方的文档。

如果您计划将 AWS 服务用作测试执行套件的一部分，则应使用在测试套件之外生成并包含在测试包中的 IAM 临时证书。这些证书应具有最少的授予权限和最短的生命周期。有关创建临时安全凭证的更多信息，请参阅《IAM 用户指南》中的使用临时安全凭证。

有关扩展测试套件和优化测试的更多方法，请参阅 在 Device Farm 中扩展自定义测试环境。

在 Device Farm 中向测试程序包中添加额外文件

您可能希望在测试中使用其他文件作为额外配置文件或其他测试数据。您可以在将这些额外文件上传到AWS Device Farm之前将其添加到测试包中，然后通过自定义环境模式访问这些文件。从根本上讲，所有测试包上传格式（ZIP、IPA、APK、JAR 等）都是支持标准 ZIP 操作的包存档格式。

在将文件上传到测试存档之前，您可以使用以下命令将其添加到 AWS Device Farm 测试存档：

```
$ zip zip-with-dependencies.zip extra_file
```

要获取额外文件的目录，请执行以下操作：

```
$ zip -r zip-with-dependencies.zip extra_files/
```

除了 IPA 文件外，这些命令适用于所有测试包上传格式。对于 IPA 文件，尤其是与一起使用时 XCUITests，由于 iOS 测试包的 AWS Device Farm 重新设计方式，我们建议您将任何多余的文件放在略有不同的位置。构建 iOS 测试时，测试应用程序目录将位于另一个名为的目录中*Payload*。

例如，此类 iOS 测试目录可能是这样的：

```
$ tree
.
### Payload
    ### ADFiOSReferenceAppUITests-Runner.app
        ### ADFiOSReferenceAppUITests-Runner
        ### Frameworks
        #   ### XCTAutomationSupport.framework
        #   #   ### Info.plist
        #   #   ### XCTAutomationSupport
        #   #   ### _CodeSignature
        #   #   #   ### CodeResources
        #   #   ### version.plist
        #   ### XCTest.framework
        #       ### Info.plist
        #       ### XCTest
        #       ### _CodeSignature
        #       #   ### CodeResources
        #       ### en.lproj
        #       #   ### InfoPlist.strings
        #       ### version.plist
        ### Info.plist
        ### PkgInfo
```

```
        ### PlugIns
        #   ### ADFiOSReferenceAppUITests.xctest
        #   #   ### ADFiOSReferenceAppUITests
        #   #   ### Info.plist
        #   #   ### _CodeSignature
        #   #       ### CodeResources
        #   ### ADFiOSReferenceAppUITests.xctest.dSYM
        #       ### Contents
        #           ### Info.plist
        #           ### Resources
        #               ### DWARF
        #                   ### ADFiOSReferenceAppUITests
        ### _CodeSignature
        #   ### CodeResources
        ### embedded.mobileprovision
```

对于这些 XCUITest 软件包，请将任何额外的文件添加到目录*.app*内结尾的*Payload*目录中。例如，以下命令显示如何向此测试包中添加文件：

```
$ mv extra_file Payload/*.app/
$ zip -r my_xcui_tests.ipa Payload/
```

将文件添加到测试包时，根据其上传格式， AWS Device Farm 中的交互行为可能会略有不同。如果上传文件使用 ZIP 文件扩展名，则 AWS Device Farm 将在测试之前自动解压缩上传文件，并将解压缩的文件留在环境变量所在的位置。*$DEVICEFARM_TEST_PACKAGE_PATH*（这意味着，如果您像第一个示例那样在档案的根目录中添加了一个名*extra_file*为的文件，则该文件将在测试*$DEVICEFARM_TEST_PACKAGE_PATH/extra_file*期间位于该文件所在的位置）。

举一个更实际的例子，如果你是 Appium Testng 用户，想要在测试中加入一个*testng.xml*文件，你可以使用以下命令将其包含在存档中：

```
$ zip zip-with-dependencies.zip testng.xml
```

然后，您可以在自定义环境模式下将测试命令更改为以下内容：

```
java -D appium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG -testjar
 *-tests.jar -d $DEVICEFARM_LOG_DIR/test-output $DEVICEFARM_TEST_PACKAGE_PATH/
testng.xml
```

如果您的测试包上传扩展名不是 ZIP（例如 APK、IPA 或 JAR 文件），则可以在上找到上传的包文件本身*$DEVICEFARM_TEST_PACKAGE_PATH*。由于这些文件仍然是存档格式的文件，因此您可以解压

缩文件，以便从中访问其他文件。例如，以下命令会将测试包的内容（用于 APK、IPA 或 JAR 文件）解压缩到该*/tmp*目录：

```
unzip $DEVICEFARM_TEST_PACKAGE_PATH -d /tmp
```

如果是 APK 或 JAR 文件，您会发现多余的文件已解压缩到该*/tmp*目录（例如）。*/tmp/extra_file*如前所述，对于 IPA 文件，多余文件将位于以结尾的文件夹内稍有不同的位置*.app*，也就是*Payload*目录内。例如，根据上面的 IPA 示例，可以在该位置找到该文件*/tmp/Payload/ADFiOSReferenceAppUITests-Runner.app/extra_file*（可引用为*/tmp/Payload/*.app/extra_file*）。

有关扩展测试套件和优化测试的更多方法，请参阅 在 Device Farm 中扩展自定义测试环境。

AWS Device Farm 中的远程访问

利用远程访问，您可以通过 Web 浏览器使用轻扫和手势功能，并实现与设备实时交互，以测试功能和重现客户问题。您可以通过与设备建立远程访问会话，与该特定设备进行交互。

Device Farm 中的会话是与托管在网络浏览器中的实际物理设备的实时互动。会话显示您在启动会话时选择的单台设备。用户一次可以启动多个会话，同时运行的设备总数受您拥有的设备槽数限制。您可以根据设备系列（例如，Android 或 iOS 设备）来购买设备槽。有关更多信息，请参阅 Device Farm 定价。

Device Farm 目前提供一部分设备用于远程访问测试。我们一直继续向设备池添加新设备。

Device Farm 捕获每个远程访问会话的视频，并生成会话期间的活动的日志。这些结果包含您在会话期间提供的任何信息。

> **Note**
>
> 出于安全考虑，建议您避免在远程访问会话期间提供或输入账号、个人登录信息和其他详细信息等敏感信息。如果可能，请使用专门为测试开发的替代方案，例如测试账户。

主题

在 AWS Device Farm 中创建远程访问会话

有关远程访问会话的更多信息，请参阅会话。

先决条件

- 在 Device Farm 中创建项目。按照在 AWS Device Farm 中创建项目中的说明操作，然后返回此页。

创建远程会话

Console

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 如果您已有项目，请从列表中选择一个项目。否则，请按照 在 AWS Device Farm 中创建项目中的说明创建项目。
4. 在 “远程访问” 选项卡上，选择 “创建远程访问会话”。
5. 为您的会话选择一个设备。您可以从可用设备列表中进行选择，或使用列表顶部的搜索栏搜索设备。
6. （可选）在会话中加入应用程序和辅助应用程序。这些可以是新上传的应用程序，也可以是过去 30 天内在此项目中上传的应用程序（30 天后，应用程序上传将过期）。
7. 在 Session name (会话名称) 中输入会话名称。
8. 选择 Confirm and start session (确认并启动会话)。

AWS CLI

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

首先， up-to-date通过下载并安装最新版本来验证您的 AWS CLI 版本是否为最新版本。

> **Important**
>
> 本文档中提到的某些命令在旧版本的 AWS CLI 中不可用。

然后，您可以确定要在哪台设备上进行测试：

```
$ aws devicefarm list-devices
```

这将显示如下输出：

```
{
    "devices":
    [
        {
            "arn": "arn:aws:devicefarm:us-
west-2::device:DE5BD47FF3BD42C3A14BF7A6EFB1BFE7",
            "name": "Google Pixel 8",
            "remoteAccessEnabled": true,
            "availability": "HIGHLY_AVAILABLE"
            ...
        },
        ...
    ]
}
```

然后，您可以使用您选择的设备 ARN 创建远程访问会话：

```
$ aws devicefarm create-remote-access-session \
  --project-arn arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef \
  --device-arn arn:aws:devicefarm:us-west-2::device:DE5BD47FF3BD42C3A14BF7A6EFB1BFE7
 \
  --app-arn arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
 \
  --configuration '{
      "auxiliaryApps": [
          "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
          "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
      ]
  }'
```

这将显示如下输出：

```
{
    "remoteAccessSession": {
        "arn": "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000",
        "name": "Google Pixel 8",
```

```
        "status": "PENDING",
        ...
}
```

现在，我们可以选择投票并等待会话准备就绪：

```
$ POLL_INTERVAL=3
TIMEOUT=600
DEADLINE=$(( $(date +%s) + TIMEOUT ))

while [[ "$(date +%s)" -lt "$DEADLINE" ]]; do

  STATUS=$(aws devicefarm get-remote-access-session \
    --arn "$DEVICE_FARM_SESSION_ARN" \
    --query 'remoteAccessSession.status' \
    --output text)

  case "$STATUS" in
    RUNNING)
      echo "Session is ready with status: $STATUS"
      break
      ;;
    STOPPING|COMPLETED)
      echo "Session ended early with status: $STATUS"
      exit 1
      ;;
  esac

done
```

Python

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

此示例首先在 Device Farm 上找到所有可用的 Google Pixel 设备，然后使用该设备创建远程访问会话并等待会话运行。

```
import random
import time
import boto3

client = boto3.client("devicefarm", region_name="us-west-2")
```

```
# 1) Gather all matching devices via paginated ListDevices with filters
filters = [
    {"attribute": "MODEL",        "operator": "CONTAINS", "values": ["Pixel"]},
    {"attribute": "AVAILABILITY", "operator": "EQUALS",   "values": ["AVAILABLE"]},
]

matching_arns = []
next_token = None
while True:
    args = {"filters": filters}
    if next_token:
        args["nextToken"] = next_token
    page = client.list_devices(**args)
    for d in page.get("devices", []):
        matching_arns.append(d["arn"])
    next_token = page.get("nextToken")
    if not next_token:
        break

if not matching_arns:
    raise RuntimeError("No available Google Pixel device found.")

# Randomly select one device from the full matching set
device_arn = random.choice(matching_arns)
print("Selected device ARN:", device_arn)

# 2) Create remote access session and wait until RUNNING
resp = client.create_remote_access_session(
    projectArn="arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef",
    deviceArn=device_arn,
    appArn="arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
  # optional
    configuration={
        "auxiliaryApps": [  # optional
            "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
            "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
        ]
    },
)
```

```
session_arn = resp["remoteAccessSession"]["arn"]
print(f"Created Remote Access Session: {session_arn}")

poll_interval = 3
timeout = 600
deadline = time.time() + timeout
terminal_states = ["STOPPING", "COMPLETED"]

while True:
    out = client.get_remote_access_session(arn=session_arn)
    status = out["remoteAccessSession"]["status"]
    print(f"Current status: {status}")

    if status == "RUNNING":
        print(f"Session is ready with status: {status}")
        break
    if status in terminal_states:
        raise RuntimeError(f"Session ended early with status: {status}")
    if time.time() >= deadline:
        raise RuntimeError("Timed out waiting for session to be ready.")
    time.sleep(poll_interval)
```

Java

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

注意：此示例使用适用于 Java 的 AWS 开发工具包 v2，并且与 JDK 版本 11 及更高版本兼容。

此示例首先在 Device Farm 上找到所有可用的 Google Pixel 设备，然后使用该设备创建远程访问会话并等待会话运行。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import
 software.amazon.awssdk.services.devicefarm.model.CreateRemoteAccessSessionConfiguration;
import
 software.amazon.awssdk.services.devicefarm.model.CreateRemoteAccessSessionRequest;
```

```
import
 software.amazon.awssdk.services.devicefarm.model.CreateRemoteAccessSessionResponse;
import software.amazon.awssdk.services.devicefarm.model.Device;
import software.amazon.awssdk.services.devicefarm.model.DeviceFilter;
import software.amazon.awssdk.services.devicefarm.model.DeviceFilterAttribute;
import
 software.amazon.awssdk.services.devicefarm.model.GetRemoteAccessSessionRequest;
import
 software.amazon.awssdk.services.devicefarm.model.GetRemoteAccessSessionResponse;
import software.amazon.awssdk.services.devicefarm.model.ListDevicesRequest;
import software.amazon.awssdk.services.devicefarm.model.ListDevicesResponse;
import software.amazon.awssdk.services.devicefarm.model.RuleOperator;

public class CreateRemoteAccessSession {
  public static void main(String[] args) throws Exception {
    DeviceFarmClient client = DeviceFarmClient.builder()
        .region(Region.US_WEST_2)
        .build();

    String projectArn = "arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef";
    String appArn     = "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
    String aux1       = "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
    String aux2       = "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789

    // 1) Gather all matching devices via paginated ListDevices with filters
    List<DeviceFilter> filters = Arrays.asList(
        DeviceFilter.builder()
            .attribute(DeviceFilterAttribute.MODEL)
            .operator(RuleOperator.CONTAINS)
            .values("Pixel")
            .build(),
        DeviceFilter.builder()
            .attribute(DeviceFilterAttribute.AVAILABILITY)
            .operator(RuleOperator.EQUALS)
            .values("AVAILABLE")
            .build()
    );

    List<String> matchingDeviceArns = new ArrayList<>();
    String next = null;
```

```
    do {
      ListDevicesResponse page = client.listDevices(
          ListDevicesRequest.builder().filters(filters).nextToken(next).build());
      for (Device d : page.devices()) {
        matchingDeviceArns.add(d.arn());
      }
      next = page.nextToken();
    } while (next != null);

    if (matchingDeviceArns.isEmpty()) {
      throw new RuntimeException("No available Google Pixel device found.");
    }

    // Randomly select one device from the full matching set
    String deviceArn = matchingDeviceArns.get(
        ThreadLocalRandom.current().nextInt(matchingDeviceArns.size()));
    System.out.println("Selected device ARN: " + deviceArn);

    // 2) Create Remote Access session and wait until it is RUNNING
    CreateRemoteAccessSessionConfiguration cfg =
CreateRemoteAccessSessionConfiguration.builder()
        .auxiliaryApps(Arrays.asList(aux1, aux2))
        .build();

    CreateRemoteAccessSessionResponse res = client.createRemoteAccessSession(
        CreateRemoteAccessSessionRequest.builder()
            .projectArn(projectArn)
            .deviceArn(deviceArn)
            .appArn(appArn)       // optional
            .configuration(cfg)   // optional
            .build());

    String sessionArn = res.remoteAccessSession().arn();
    System.out.println("Created Remote Access Session: " + sessionArn);

    int pollIntervalMs = 3000;
    long timeoutMs = 600_000L;
    long deadline = System.currentTimeMillis() + timeoutMs;

    while (true) {
      GetRemoteAccessSessionResponse get = client.getRemoteAccessSession(
          GetRemoteAccessSessionRequest.builder().arn(sessionArn).build());
      String status = get.remoteAccessSession().statusAsString();
      System.out.println("Current status: " + status);
```

```
      if ("RUNNING".equals(status)) {
        System.out.println("Session is ready with status: " + status);
        break;
      }
      if ("STOPPING".equals(status) || "COMPLETED".equals(status)) {
        throw new RuntimeException("Session ended early with status: " + status);
      }
      if (System.currentTimeMillis() >= deadline) {
        throw new RuntimeException("Timed out waiting for session to be ready.");
      }
      Thread.sleep(pollIntervalMs);
    }
  }
}
```

JavaScript

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

注意：此示例使用适用于 JavaScript v3 的 AWS 开发工具包。

此示例首先在 Device Farm 上找到所有可用的 Google Pixel 设备，然后使用该设备创建远程访问会话并等待会话运行。

```
import {
  DeviceFarmClient,
  ListDevicesCommand,
  CreateRemoteAccessSessionCommand,
  GetRemoteAccessSessionCommand,
} from "@aws-sdk/client-device-farm";

const client = new DeviceFarmClient({ region: "us-west-2" });

// 1) Gather all matching devices via paginated ListDevices with filters
const filters = [
  { attribute: "MODEL",        operator: "CONTAINS", values: ["Pixel"] },
  { attribute: "AVAILABILITY", operator: "EQUALS",   values: ["AVAILABLE"] },
];

let nextToken;
const matching = [];
```

```
while (true) {
  const page = await client.send(new ListDevicesCommand({ filters, nextToken }));
  for (const d of page.devices ?? []) {
    matching.push(d.arn);
  }
  nextToken = page.nextToken;
  if (!nextToken) break;
}

if (matching.length === 0) {
  throw new Error("No available Google Pixel device found.");
}

// Randomly select one device from the full matching set
const deviceArn = matching[Math.floor(Math.random() * matching.length)];
console.log("Selected device ARN:", deviceArn);

// 2) Create remote access session and wait until RUNNING
const out = await client.send(new CreateRemoteAccessSessionCommand({
  projectArn: "arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef",
  deviceArn,
  appArn: "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
 optional
  configuration: {
    auxiliaryApps: [ // optional
      "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
      "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
    ],
  },
}));

const sessionArn = out.remoteAccessSession?.arn;
console.log("Created Remote Access Session:", sessionArn);

const pollIntervalMs = 3000;
const timeoutMs = 600000;
const deadline = Date.now() + timeoutMs;

while (true) {
```

```
  const get = await client.send(new GetRemoteAccessSessionCommand({ arn:
 sessionArn }));
  const status = get.remoteAccessSession?.status;
  console.log("Current status:", status);

  if (status === "RUNNING") {
    console.log("Session is ready with status:", status);
    break;
  }
  if (status === "STOPPING" || status === "COMPLETED") {
    throw new Error(`Session ended early with status: ${status}`);
  }
  if (Date.now() >= deadline) {
    throw new Error("Timed out waiting for session to be ready.");
  }
  await new Promise((r) => setTimeout(r, pollIntervalMs));
}
```

C#

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

此示例首先在 Device Farm 上找到所有可用的 Google Pixel 设备，然后使用该设备创建远程访问会话并等待会话运行。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Amazon;
using Amazon.DeviceFarm;
using Amazon.DeviceFarm.Model;

class Program
{
    static async Task Main()
    {
        var client = new AmazonDeviceFarmClient(RegionEndpoint.USWest2);

        // 1) Gather all matching devices via paginated ListDevices with filters
        var filters = new List<DeviceFilter>
        {
```

```
            new DeviceFilter { Attribute = DeviceAttribute.MODEL,        Operator =
 RuleOperator.CONTAINS, Values = new List<string>{ "Pixel" } },
            new DeviceFilter { Attribute = DeviceAttribute.AVAILABILITY, Operator =
 RuleOperator.EQUALS,   Values = new List<string>{ "AVAILABLE" } },
        };

        var matchingArns = new List<string>();
        string nextToken = null;

        do
        {
            var list = await client.ListDevicesAsync(new ListDevicesRequest
            {
                Filters = filters,
                NextToken = nextToken
            });

            foreach (var d in list.Devices)
                matchingArns.Add(d.Arn);

            nextToken = list.NextToken;
        }
        while (nextToken != null);

        if (matchingArns.Count == 0)
            throw new Exception("No available Google Pixel device found.");

        // Randomly select one device from the full matching set
        var rnd = new Random();
        var deviceArn = matchingArns[rnd.Next(matchingArns.Count)];
        Console.WriteLine($"Selected device ARN: {deviceArn}");

        // 2) Create remote access session and wait until RUNNING
        var request = new CreateRemoteAccessSessionRequest
        {
            ProjectArn = "arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef",
            DeviceArn  = deviceArn,
            AppArn     = "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
 optional
            Configuration = new CreateRemoteAccessSessionConfiguration
            {
                AuxiliaryApps = new List<string>
```

```
                {
                    "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
                    "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
                }
            }
        };

        request.Configuration.AuxiliaryApps.RemoveAll(string.IsNullOrWhiteSpace);

        var response = await client.CreateRemoteAccessSessionAsync(request);
        var sessionArn = response.RemoteAccessSession.Arn;
        Console.WriteLine($"Created Remote Access Session: {sessionArn}");

        var pollIntervalMs = 3000;
        var timeoutMs = 600000;
        var deadline = DateTime.UtcNow.AddMilliseconds(timeoutMs);

        while (true)
        {
            var get = await client.GetRemoteAccessSessionAsync(new
 GetRemoteAccessSessionRequest { Arn = sessionArn });
            var status = get.RemoteAccessSession.Status.Value;
            Console.WriteLine($"Current status: {status}");

            if (status == "RUNNING")
            {
                Console.WriteLine($"Session is ready with status: {status}");
                break;
            }
            if (status == "STOPPING" || status == "COMPLETED")
            {
                throw new Exception($"Session ended early with status: {status}");
            }
            if (DateTime.UtcNow >= deadline)
            {
                throw new TimeoutException("Timed out waiting for session to be
 ready.");
            }

            await Task.Delay(pollIntervalMs);
        }
    }
```

```
}
```

Ruby

注意：这些说明仅侧重于创建远程访问会话。有关如何上传应用程序以便在会话期间使用的说明，请参阅自动上传应用程序。

此示例首先在 Device Farm 上找到所有可用的 Google Pixel 设备，然后使用该设备创建远程访问会话并等待会话运行。

```
require 'aws-sdk-devicefarm'

client = Aws::DeviceFarm::Client.new(region: 'us-west-2')

# 1) Gather all matching devices via paginated ListDevices with filters
filters = [
  { attribute: 'MODEL',        operator: 'CONTAINS', values: ['Pixel'] },
  { attribute: 'AVAILABILITY', operator: 'EQUALS',   values: ['AVAILABLE'] },
]

matching_arns = []
next_token = nil
loop do
  resp = client.list_devices(filters: filters, next_token: next_token)
  resp.devices&.each { |d| matching_arns << d.arn }
  next_token = resp.next_token
  break unless next_token
end

abort "No available Google Pixel device found." if matching_arns.empty?

# Randomly select one device from the full matching set
device_arn = matching_arns.sample
puts "Selected device ARN: #{device_arn}"

# 2) Create remote access session and wait until RUNNING
resp = client.create_remote_access_session(
  project_arn: "arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef",
  device_arn:  device_arn,
  app_arn:     "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
 # optional
```

```
  configuration: {
    auxiliary_apps: [ # optional
      "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
      "arn:aws:devicefarm:us-
west-2:111122223333:upload:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-456789
    ].compact
  }
)

session_arn = resp.remote_access_session.arn
puts "Created Remote Access Session: #{session_arn}"

poll_interval = 3
timeout = 600
deadline = Time.now + timeout
terminal = %w[STOPPING COMPLETED]

loop do
  get = client.get_remote_access_session(arn: session_arn)
  status = get.remote_access_session.status
  puts "Current status: #{status}"

  if status == 'RUNNING'
    puts "Session is ready with status: #{status}"
    break
  end

  abort "Session ended early with status: #{status}" if terminal.include?(status)
  abort "Timed out waiting for session to be ready." if Time.now >= deadline
  sleep poll_interval
end
```

后续步骤

一旦请求的设备和基础设施可用，Device Farm 就会立即启动会话，通常在几分钟之内。将会显示请求的设备对话框，直到会话开始。要取消会话请求，请选择 Cancel request (取消请求)。

如果所选设备不可用或忙碌，则会话状态将显示为 “待处理设备”，这表示您可能需要等待一段时间才能使用该设备进行测试。

如果您的账户已达到公共按流量计费或不按流量计费的设备的并发限制，则会话状态将显示为 “待定并发”。对于不按流量计费的设备插槽，您可以通过购买更多设备插槽来增加并发度。对于按流量计费的 pay-as-you-go设备，请通过支持请求与 AWS 联系，申请增加服务配额。

当会话设置开始时，当您的本地 Web 浏览器尝试打开与设备的远程连接时，会话设置首先显示为 “进行中” 状态，然后显示为 “正在连接” 状态。

会话开始后，如果您关闭浏览器或浏览器选项卡而没有停止会话，或者浏览器和 Internet 之间的连接中断，那么在五分钟内会话将保持活动状态。之后，Device Farm 结束会话。但是，我们仍将针对空闲时间向您的账户收费。

会话开始后，您可以在 Web 浏览器中与设备进行交互，或者使用 Appi um 测试设备。

在 AWS Device Farm 中使用远程访问会话

有关通过远程访问会话对 Android 和 iOS 应用程序执行交互式测试的信息，请参阅 会话。

先决条件

- 创建会话。按照创建会话中的说明操作，然后返回此页。

在 Device Farm 控制台中使用会话

一旦您为远程访问会话请求的设备变得可用，控制台将会显示该设备屏幕。会话的最大长度为 150 分钟。会话的剩余时间显示在设备名称旁边的剩余时间字段中。

安装应用程序

要在会话设备上安装应用程序，请在安装应用程序中选择选择文件，然后选择您要安装的 .apk 文件 (Android) 或 .ipa 文件 (iOS)。您在远程访问会话中运行的应用程序不需要任何测试设备或配置。

> **Note**
>
> 在您上传应用程序时，有时可能需要等一段时间，应用程序才可用。将出现一条确认消息，告知您该应用程序是否已成功安装。

控制设备

您可以使用您的触摸式鼠标或可比设备和设备屏幕上的键盘与控制台中显示的设备交互，就像您与实际的物理设备交互一样。对于 Android 设备，View controls (查看控件) 中有一些按钮的功能与 Android 设备上的 Home (主页) 和 Back (返回) 按钮一样。对于 iOS 设备，Home (主页) 按钮的功能与 iOS 设备上的主页按钮一样。您还可以通过选择近期使用的应用程序 在设备上运行的应用程序之间切换。

在纵向和横向模式之间切换

您还可以在所使用的设备的纵向（垂直）和横向（水平）模式之间切换。

后续步骤

Device Farm 会继续会话，直到您手动停止它，或直到达到 150 分钟的时间限制。要结束会话，请选择停止会话按钮。会话停止后，您可以访问捕获的视频和生成的日志。有关更多信息，请参阅 检索会话结果。

提示与诀窍

在某些 AWS 区域，您可能会遇到远程访问会话的性能问题。这部分是由于一些区域中存在延迟。如果您遇到性能问题，请在再次与应用程序交互之前留出一些时间，以便远程访问会话跟上您的步伐。

在 AWS Device Farm 中检索远程访问会话的结果

有关会话的更多信息，请参阅会话。

- 先决条件
- 查看会话详细信息
- 下载会话视频或日志

先决条件

- 完成会话。按照在 AWS Device Farm 中使用远程访问会话中的说明操作，然后返回此页。

查看会话详细信息

当远程访问会话结束时，Device Farm 控制台会显示一个表，其中包含关于会话期间的活动的详细信息。有关更多信息，请参阅分析日志信息。

要想稍后返回到会话的详细信息：

1. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
2. 选择包含会话的项目。
3. 选择远程访问，然后从列表中选择要查看的会话。

下载会话视频或日志

当远程访问会话结束时，您可以通过 Device Farm 控制台访问会话的视频捕获和活动日志。在会话结果中，选择 Files (文件) 选项卡，以查看会话视频和日志的链接列表。您可以在浏览器中查看这些文件，或将其保存在本地。

AWS Device Farm 中的 Appium 测试

在远程访问会话期间，您可以在本地环境中运行 Appium 测试，使用托管 Appium 端点瞄准会话的设备。借助 Appium 端点，您可以通过快速反馈和快速迭代来开发、测试和执行 Appium 代码。这种客户端测试方法使您可以灵活地从您选择的任何 Appium 客户端环境连接到 Device Farm 设备。

为了补充客户端测试，Device Farm 还支持在该服务管理的基础架构上运行测试，称为服务器端执行。通过这种方法，您可以将应用程序和测试上传到服务，然后使用服务管理的测试主机在多台设备上并行执行测试。这种方法可以很好地扩展到在许多设备上独立进行测试，也可以在 CI/CD 管道环境中进行测试。

要了解有关服务器端执行的更多信息，请参阅测试框架和内置测试。

主题

什么是 Appium 终端节点？

Appium 是一个流行的开源软件测试框架，用于在 iOS 和 Android 上测试不同设备（包括手机和平板电脑）上的原生、混合和移动网络应用程序。它允许开发人员和 QA（质量保证）工程师编写脚本，这些脚本可以远程控制设备、模拟用户交互并验证被测应用程序是否按预期运行。Appium 从最终用户的角度与应用程序交互，使测试人员能够开发测试来模拟真实用户将如何使用该应用程序进行测试。

Appium 建立在客户端-服务器模型之上，即本地客户端请求（本地或远程）Appium 服务器代表他们命令设备。Appium 服务器管理用于与设备通信的驱动程序，例如适用于 Android 的UIAutomator2 驱动程序或 iOS 的XCUITest 驱动程序。所有命令都遵循关于如何控制设备的 W3C WebDriver 标准。

Device Farm 的 Appium 端点会在您的远程访问会话中显示该设备的 Appium 服务器网址。Appium 端点 URL 将特定于该会话中的该设备，并在会话期间保持有效，这样您就可以在同一台设备上进行迭代，而无需额外的设置时间。有关远程访问的更多信息，请参阅远程访问。

Appium 测试入门

对于大多数 Appium 用户来说，使用 Device Farm 进行 Appium 测试只需要对现有测试配置进行细微的更改即可。

简而言之，使用 Device Farm 进行客户端 Appium 测试需要三个步骤：

1. 首先，您需要创建远程访问会话来测试 Device Farm 设备。您可以将应用程序包含在远程访问请求中，也可以在会话开始后安装应用程序。
2. 会话运行后，你可以复制 Appium 端点 URL，然后通过独立工具（比如 Appium Ins pect or）或从 IDE 中的 Appium 测试代码中使用它。该 URL 将在远程访问会话期间有效。
3. 最后，Appium 测试开始后，您可以在测试执行期间查看您的 Appium 服务器日志以及设备的视频流。

使用 Appium 与设备互动

创建远程访问会话后，该设备将可用于 Appium 测试。在整个远程访问会话期间，你可以在设备上随心所欲地运行任意数量的 Appium 会话，对使用的客户端没有限制。例如，你可以先使用 IDE 中的本地 Appium 代码运行测试，然后切换到使用 Appium Inspector 来解决遇到的任何问题。会话最长可持续 150 分钟，但是，如果超过 5 分钟没有活动（通过交互式控制台或 Appium 端点），则会话将超时。

在 Appium 会话中使用应用程序进行测试

Device Farm 允许您将应用程序用作远程访问会话创建请求的一部分，或者在远程访问会话本身期间安装应用程序。这些应用程序会自动安装到被测设备上，并作为任何 Appium 会话请求的默认功能注入。创建远程访问会话时，您可以选择传入应用程序 ARN（默认情况下，该应用程序将用作所有后续的 Appium 会话的`appium:app`功能）以及辅助应用程序 ARNs（将用作该功能）。`appium:otherApps`

例如，如果您使用应用程序`com.aws.devicefarm.sample`作为应用程序和`com.aws.devicefarm.other.sample`辅助应用程序之一来创建远程访问会话，那么当您开始创建 Appium 会话时，它将具有类似于以下内容的功能：

```
{
    "value":
    {
        "sessionId": "abcdef123456-1234-5678-abcd-abcdef123456",
        "capabilities":
        {
```

```
            "app": "/tmp/com.aws.devicefarm.sample.apk",
            "otherApps": "[\"/tmp/com.aws.devicefarm.other.sample.apk\"]",
            ...
        }
    }
}
```

在会话期间，您可以安装其他应用程序（在控制台中或使用[InstallToRemoteAccessSession](#)API）。它们将覆盖以前用作该appium:app功能的任何现有应用程序。如果以前使用的应用程序具有不同的软件包名称，则它们将保留在设备上并用作appium:otherApps功能的一部分。

例如，如果您最初在创建远程访问会话com.aws.devicefarm.sample时使用应用程序，但随后在会话com.aws.devicefarm.other.sample期间安装了一个名为的新应用程序，则您的 Appium 会话将具有与以下内容类似的功能：

```
{
    "value":
    {
        "sessionId": "abcdef123456-1234-5678-abcd-abcdef123456",
        "capabilities":
        {
            "app": "/tmp/com.aws.devicefarm.other.sample.apk",
            "otherApps": "[\"/tmp/com.aws.devicefarm.sample.apk\"]",
            ...
        }
    }
}
```

如果你愿意，你可以使用应用程序名称明确指定应用程序的功能（分别使用适用于 Android 和 iOS 的appium:appPackage或appium:bundleId功能）。

如果您正在测试 Web 应用程序，请为 Appium 会话创建请求指定browserName功能。该Chrome浏览器可在所有 Android 设备上使用，该Safari浏览器可在所有 iOS 设备上使用。

Device Farm 不支持在远程访问会话appium:app期间传入远程 URL 或本地文件系统路径。将应用程序上传到 Device Farm，改为将其包含在会话中。

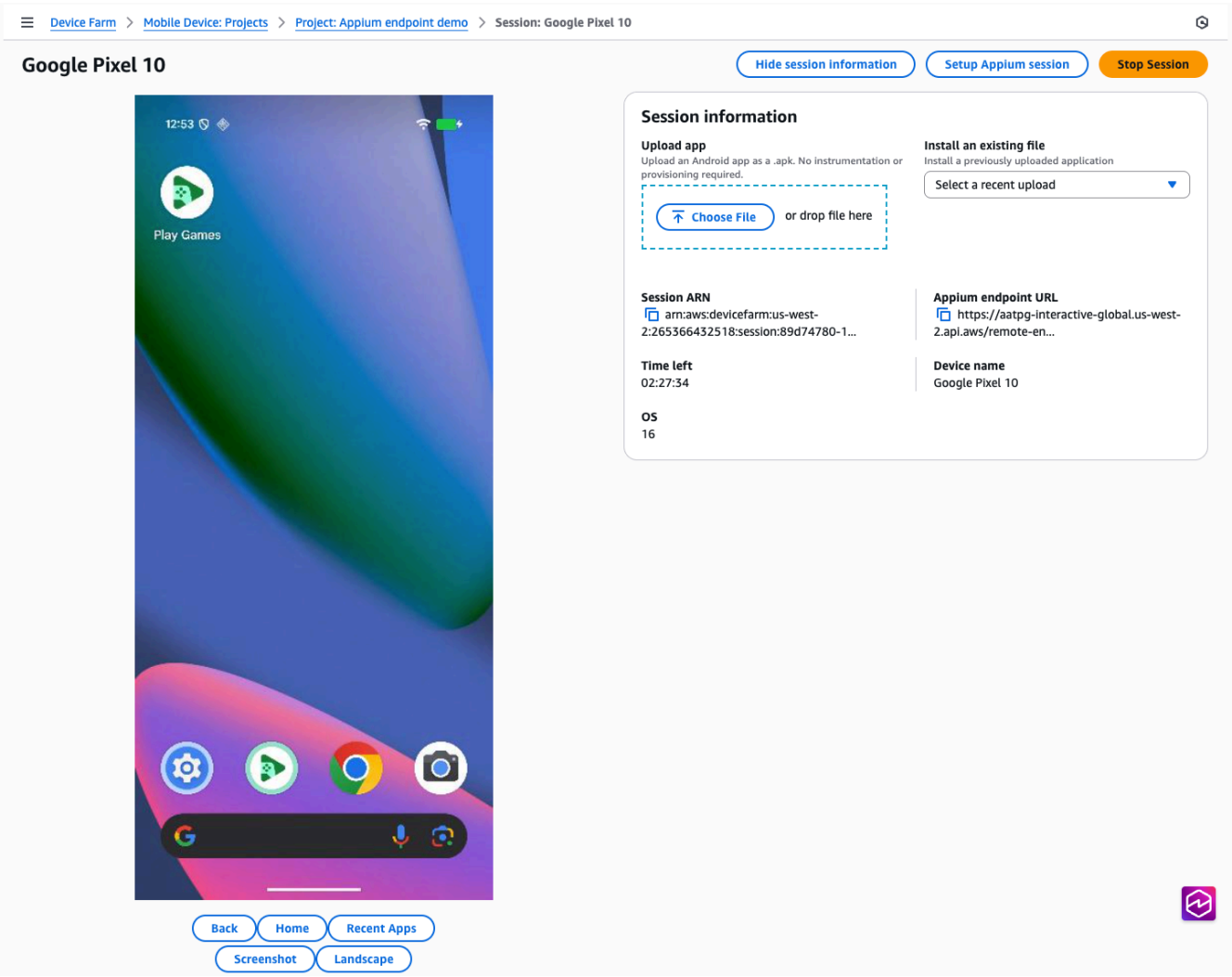
> **Note**
>
> 有关在远程访问会话中自动上传应用程序的更多信息，请参阅[自动上传应用程序](#)。

如何使用 Appium 终端节点

以下是从控制台、和访问会话的 Appium 端点的 AWS CLI步骤。 AWS SDKs这些步骤包括如何开始使用各种 Appium 客户端测试框架运行测试：

Console

1. 在 Web 浏览器中打开远程访问会话页面：



2. 要使用 Appium Inspector 运行会话，请执行以下操作：

   a. 单击 “设置 Appi um 会话” 按钮

   b. 按照页面上的说明进行操作，了解如何使用 Appium Inspector 启动会话。

3. 要从本地 IDE 运行 Appium 测试，请执行以下操作：

   a. 点击文本 Appium 端点网址旁边的 “复制” 图标

b. 将此 URL 粘贴到您当前指定远程地址或命令执行器的本地 Appium 代码中。要查看特定语言的示例，请单击此示例窗口中您选择的语言的选项卡。

AWS CLI

首先， up-to-date通过下载并安装最新版本来验证您的 AWS CLI 版本是否为最新版本。

> **Important**
>
> Appium 终端节点字段在旧版本的 AWS CLI 中不可用。

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：`GetRemoteAccessSession`

```
$ aws devicefarm get-remote-access-session \
    --arn "arn:aws:devicefarm:us-west-2:123456789876:session:abcdef123456-1234-5678-abcd-abcdef123456/abcdef123456-1234-5678-abcd-abcdef123456/00000"
```

这将显示如下输出：

```
{
    "remoteAccessSession": {
        "arn": "arn:aws:devicefarm:us-west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/abcdef123456-1234-5678-abcd-abcdef123456/00000",
        "name": "Google Pixel 8",
        "status": "RUNNING",
        "endpoints": {
            "remoteDriverEndpoint": "https://devicefarm-interactive-global.us-west-2.api.aws/remote-endpoint/ABCD1234...",
        ...
}
```

无论您当前指定远程地址或命令执行器，都可以在本地 Appium 代码中使用此 URL。要查看特定语言的示例，请单击此示例窗口中您选择的语言的选项卡。

有关如何直接从命令行与端点交互的示例，您可以使用命令行工具 curl 直接调用端点： WebDriver

```
$ curl "https://devicefarm-interactive-global.us-west-2.api.aws/remote-endpoint/
ABCD1234.../status"
```

这将显示如下输出：

```
{
    "value":
    {
        "ready": true,
        "message": "The server is ready to accept new connections",
        "build":
        {
            "version": "2.5.1"
        }
    }
}
```

Python

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：GetRemoteAccessSession

```
# To get the URL
import sys
import boto3
from botocore.exceptions import ClientError

def get_appium_endpoint() -> str:
    session_arn = "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000"
    device_farm_client = boto3.client("devicefarm", region_name="us-west-2")

    try:
        resp = device_farm_client.get_remote_access_session(arn=session_arn)
    except ClientError as exc:
        sys.exit(f"Failed to call Device Farm: {exc}")

    remote_access_session = resp.get("remoteAccessSession", {})
    endpoints = remote_access_session.get("endpoints", {})
    endpoint = endpoints.get("remoteDriverEndpoint")
```

```
    if not endpoint:
        sys.exit("Device Farm response did not include
 endpoints.remoteDriverEndpoint")

    return endpoint

# To use the URL
from appium import webdriver
from appium.options.android import UiAutomator2Options

opts = UiAutomator2Options()
driver = webdriver.Remote(get_appium_endpoint(), options=opts)
# ...
driver.quit()
```

Java

注意：此示例使用 AWS 适用于 Java v2 的 SDK，并且与 JDK 版本 11 及更高版本兼容。

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：[GetRemoteAccessSession]

```
// To get the URL
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import
 software.amazon.awssdk.services.devicefarm.model.GetRemoteAccessSessionRequest;
import
 software.amazon.awssdk.services.devicefarm.model.GetRemoteAccessSessionResponse;

public class AppiumEndpointBuilder {
    public static String getAppiumEndpoint() throws Exception {
        String session_arn = "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000";

        try (DeviceFarmClient client = DeviceFarmClient.builder()
                .region(Region.US_WEST_2)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build()) {

            GetRemoteAccessSessionResponse resp = client.getRemoteAccessSession(
                    GetRemoteAccessSessionRequest.builder().arn(session_arn).build()
```

```
            );

            String endpoint =
 resp.remoteAccessSession().endpoints().remoteDriverEndpoint();
            if (endpoint == null || endpoint.isEmpty()) {
                throw new IllegalStateException("remoteDriverEndpoint missing from
 response");
            }
            return endpoint;
        }
    }
}

// To use the URL
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.android.options.UiAutomator2Options;

import java.net.URL;

public class ExampleTest {
    public static void main(String[] args) throws Exception {
        String endpoint = AppiumEndpointBuilder.getAppiumEndpoint();
        UiAutomator2Options options = new UiAutomator2Options();
        AndroidDriver driver = new AndroidDriver(new URL(endpoint), options);

        try {
            // ... your test ...
        } finally {
            driver.quit();
        }
    }
}
```

JavaScript

注意：此示例使用 AWS 适用于 v JavaScript 3 的 SDK，使用 Node 18+ 的 WebDriverIO v8+。

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：[GetRemoteAccessSession]

```
// To get the URL
import { DeviceFarmClient, GetRemoteAccessSessionCommand } from "@aws-sdk/client-
device-farm";
```

```
export async function getAppiumEndpoint() {
  const sessionArn = "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000";

  const client = new DeviceFarmClient({ region: "us-west-2" });
  const resp = await client.send(new GetRemoteAccessSessionCommand({ arn:
 sessionArn }));

  const endpoint = resp?.remoteAccessSession?.endpoints?.remoteDriverEndpoint;
  if (!endpoint) throw new Error("remoteDriverEndpoint missing from response");
  return endpoint;
}

// To use the URL with WebdriverIO
import { remote } from "webdriverio";

(async () => {
  const endpoint = await getAppiumEndpoint();
  const u = new URL(endpoint);

  const driver = await remote({
    protocol: u.protocol.replace(":", ""),
    hostname: u.hostname,
    port: u.port ? Number(u.port) : (u.protocol === "https:" ? 443 : 80),
    path: u.pathname + u.search,
    capabilities: {
      platformName: "Android",
      "appium:automationName": "UiAutomator2",
      // ...other caps...
    },
  });

  try {
    // ... your test ...
  } finally {
    await driver.deleteSession();
  }
})();
```

C#

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：[GetRemoteAccessSession]

```
// To get the URL
using System;
using System.Threading.Tasks;
using Amazon;
using Amazon.DeviceFarm;
using Amazon.DeviceFarm.Model;

public static class AppiumEndpointBuilder
{
    public static async Task<string> GetAppiumEndpointAsync()
    {
        var sessionArn = "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000";

        var config = new AmazonDeviceFarmConfig
        {
            RegionEndpoint = RegionEndpoint.USWest2
        };
        using var client = new AmazonDeviceFarmClient(config);

        var resp = await client.GetRemoteAccessSessionAsync(new
 GetRemoteAccessSessionRequest { Arn = sessionArn });
        var endpoint = resp?.RemoteAccessSession?.Endpoints?.RemoteDriverEndpoint;

        if (string.IsNullOrWhiteSpace(endpoint))
            throw new InvalidOperationException("RemoteDriverEndpoint missing from
 response");

        return endpoint;
    }
}

// To use the URL
using OpenQA.Selenium.Appium;
using OpenQA.Selenium.Appium.Android;

class Example
{
    static async Task Main()
    {
        var endpoint = await AppiumEndpointBuilder.GetAppiumEndpointAsync();
```

```
        var options = new AppiumOptions();
        options.PlatformName = "Android";
        options.AutomationName = "UiAutomator2";

        using var driver = new AndroidDriver(new Uri(endpoint), options);
        try
        {
            // ... your test ...
        }
        finally
        {
            driver.Quit();
        }
    }
}
```

Ruby

会话启动并运行后，Appium 端点网址将通过`remoteDriverEndpoint`在响应 API 调用时命名的字段提供：[GetRemoteAccessSession]

```
# To get the URL
require 'aws-sdk-devicefarm'

def get_appium_endpoint
  session_arn = "arn:aws:devicefarm:us-
west-2:111122223333:session:abcdef123456-1234-5678-abcd-abcdef123456/
abcdef123456-1234-5678-abcd-abcdef123456/00000"

  client = Aws::DeviceFarm::Client.new(region: 'us-west-2')
  resp = client.get_remote_access_session(arn: session_arn)
  endpoint = resp.remote_access_session.endpoints.remote_driver_endpoint
  raise "remote_driver_endpoint missing from response" if endpoint.nil? ||
 endpoint.empty?
  endpoint
end

# To use the URL
require 'appium_lib_core'

endpoint = get_appium_endpoint
opts = {
  server_url: endpoint,
```

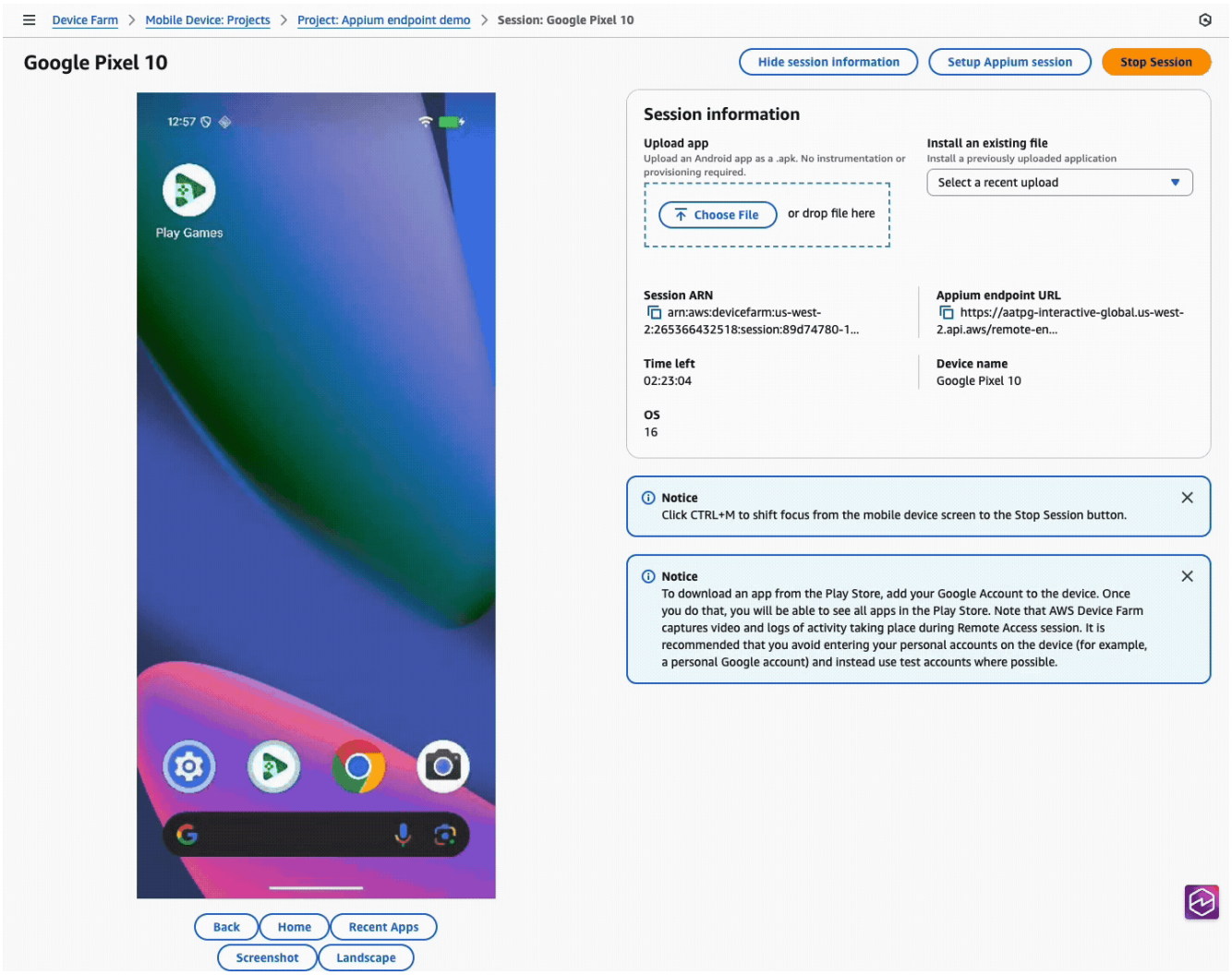
```
  capabilities: {
    'platformName' => 'Android',
    'appium:automationName' => 'UiAutomator2'
  }
}

driver = Appium::Core.for(opts).start_driver
begin
  # ... your test ...
ensure
  driver.quit
end
```

查看你的 Appium 服务器日志

启动 Appium 会话后，您可以在 Device Farm 控制台中实时查看 Appium 服务器日志，也可以在远程访问会话结束后下载这些日志。以下是执行此操作的说明：

Console

1. 在 Device Farm 控制台中，打开设备的远程访问会话。
2. 通过本地 IDE 或 Appium Inspector 与设备启动 Appium 端点会话
3. 然后，Appium 服务器日志将出现在远程访问会话页面的设备旁边，“会话信息” 位于设备下方的页面底部：



AWS CLI

注意：此示例使用命令行工具*curl*从 D evice Farm 中提取日志。

在会话期间或会话之后，您可以使用设备群的 `ListArtifacts`API 下载 Appium 服务器日志。

```
$ aws devicefarm list-artifacts \
  --type FILE \
  --arn arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678
```

这将显示会话期间的输出，如下所示：

```
{
    "artifacts": [
```

```
        {
            "arn": "arn:aws:devicefarm:us-
west-2:111122223333:artifact:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-4567
            "name": "AppiumServerLogOutput",
            "type": "APPIUM_SERVER_LOG_OUTPUT",
            "extension": "",
            "url": "https://prod-us-west-2-results.s3.dualstack.us-
west-2.amazonaws.com/111122223333/12345678..."
        }
    ]
}
```

会话结束后还有以下几点：

```
{
    "artifacts": [
        {
            "arn": "arn:aws:devicefarm:us-
west-2:111122223333:artifact:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-4567
            "name": "Appium Server Output",
            "type": "APPIUM_SERVER_OUTPUT",
            "extension": "log",
            "url": "https://prod-us-west-2-results.s3.dualstack.us-
west-2.amazonaws.com/111122223333/12345678..."
        }
    ]
}
```

```
$ curl "https://prod-us-west-2-results.s3.dualstack.us-
west-2.amazonaws.com/111122223333/12345678..."
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1',
info Appium   allowInsecure:
info Appium    [ 'execute_driver_script',
info Appium      'session_discovery',
info Appium      'perf_record',
info Appium      'adb_shell',
info Appium      'chromedriver_autodownload',
```

```
info Appium      'get_server_logs' ],
info Appium   keepAliveTimeout: 0,
info Appium   logNoColors: true,
info Appium   logTimestamp: true,
info Appium   longStacktrace: true,
info Appium   sessionOverride: true,
info Appium   strictCaps: true,
info Appium   useDrivers: [ 'uiautomator' ] }
```

Python

注意：此示例使用第三方*requests*软件包下载日志，并使用适用于 Python 的 AWS SDK *boto3*。

在会话期间或之后，您可以使用设备群的 ListArtifactsAPI 来检索 Appium 服务器日志 URL，然后将其下载。

```
import pathlib
import requests
import boto3

def download_appium_log():
    session_arn = "arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678
    client = boto3.client("devicefarm", region_name="us-west-2")

    # 1) List artifacts for the session (FILE artifacts), handling pagination
    artifacts = []
    token = None
    while True:
        kwargs = {"arn": session_arn, "type": "FILE"}
        if token:
            kwargs["nextToken"] = token
        resp = client.list_artifacts(**kwargs)
        artifacts.extend(resp.get("artifacts", []))
        token = resp.get("nextToken")
        if not token:
            break

    if not artifacts:
        raise RuntimeError("No artifacts found in this session")

    # Filter strictly to Appium server logs
```

```
    allowed = {"APPIUM_SERVER_OUTPUT", "APPIUM_SERVER_LOG_OUTPUT"}
    filtered = [a for a in artifacts if a.get("type") in allowed]
    if not filtered:
        raise RuntimeError("No Appium server log artifacts found (expected
 APPIUM_SERVER_OUTPUT or APPIUM_SERVER_LOG_OUTPUT)")

    # Prefer the final 'OUTPUT' log, else the live 'LOG_OUTPUT'
    chosen = (next((a for a in filtered if a.get("type") == "APPIUM_SERVER_OUTPUT"),
 None)
              or next((a for a in filtered if a.get("type") ==
 "APPIUM_SERVER_LOG_OUTPUT"), None))

    url = chosen["url"]
    ext = chosen.get("extension") or "log"
    out = pathlib.Path(f"./appium_server_log.{ext}")

    # 2) Download the artifact
    with requests.get(url, stream=True) as r:
        r.raise_for_status()
        with open(out, "wb") as fh:
            for chunk in r.iter_content(chunk_size=1024 * 1024):
                if chunk:
                    fh.write(chunk)

    print(f"Saved Appium server log to: {out.resolve()}")

download_appium_log()
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1', allowInsecure: [ 'execute_driver_script', ... ],
 useDrivers: [ 'uiautomator' ] }
```

Java

注意：此示例使用 AWS 适用于 Java v2 的 SDK 下载日志，并且与 JDK 版本 11 及更高版本兼容。*HttpClient*

在会话期间或之后，您可以使用设备群的 ListArtifactsAPI 来检索 Appium 服务器日志 URL，然后将其下载。

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Path;
import java.time.Duration;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import software.amazon.awssdk.services.devicefarm.model.Artifact;
import software.amazon.awssdk.services.devicefarm.model.ArtifactCategory;
import software.amazon.awssdk.services.devicefarm.model.ListArtifactsRequest;
import software.amazon.awssdk.services.devicefarm.model.ListArtifactsResponse;

public class AppiumLogDownloader {

    public static void main(String[] args) throws Exception {
        String sessionArn = "arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678

        try (DeviceFarmClient client = DeviceFarmClient.builder()
                .region(Region.US_WEST_2)
                .build()) {

            // 1) List artifacts for the session (FILE artifacts) with pagination
            List<Artifact> all = new ArrayList<>();
            String token = null;
            do {
                ListArtifactsRequest.Builder b = ListArtifactsRequest.builder()
                        .arn(sessionArn)
                        .type(ArtifactCategory.FILE);
                if (token != null) b.nextToken(token);
                ListArtifactsResponse page = client.listArtifacts(b.build());
                all.addAll(page.artifacts());
                token = page.nextToken();
            } while (token != null && !token.isBlank());

            // Filter strictly to Appium logs
            List<Artifact> filtered = all.stream()
```

```
                    .filter(a -> {
                        String t = a.typeAsString();
                        return "APPIUM_SERVER_OUTPUT".equals(t) ||
"APPIUM_SERVER_LOG_OUTPUT".equals(t);
                    })
                    .toList();

            if (filtered.isEmpty()) {
                throw new RuntimeException("No Appium server log artifacts found
(expected APPIUM_SERVER_OUTPUT or APPIUM_SERVER_LOG_OUTPUT).");
            }

            // Prefer OUTPUT; else LOG_OUTPUT
            Artifact chosen = filtered.stream()
                    .filter(a -> "APPIUM_SERVER_OUTPUT".equals(a.typeAsString()))
                    .findFirst()
                    .orElseGet(() -> filtered.stream()
                            .filter(a ->
"APPIUM_SERVER_LOG_OUTPUT".equals(a.typeAsString()))
                            .findFirst()
                            .get());

            String url = chosen.url();
            String ext = (chosen.extension() == null ||
chosen.extension().isBlank()) ? "log" : chosen.extension();
            Path out = Path.of("appium_server_log." + ext);

            // 2) Download the artifact with HttpClient
            HttpClient http = HttpClient.newBuilder()
                    .connectTimeout(Duration.ofSeconds(10))
                    .build();

            HttpRequest get = HttpRequest.newBuilder(URI.create(url))
                    .timeout(Duration.ofMinutes(5))
                    .GET()
                    .build();

            HttpResponse<Path> resp = http.send(get,
HttpResponse.BodyHandlers.ofFile(out));
            if (resp.statusCode() / 100 != 2) {
                throw new IOException("Failed to download log, HTTP " +
resp.statusCode());
            }
```

```
            System.out.println("Saved Appium server log to: " +
 out.toAbsolutePath());
        }
    }
}
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1', ..., useDrivers: [ 'uiautomator' ] }
```

JavaScript

注意：此示例使用 AWS 适用于 JavaScript (v3) 和节点 18+ *fetch* 的 SDK 下载日志。

在会话期间或之后，您可以使用设备群的 ListArtifactsAPI 来检索 Appium 服务器日志 URL，然后将其下载。

```
import { DeviceFarmClient, ListArtifactsCommand } from "@aws-sdk/client-device-
farm";
import { createWriteStream } from "fs";
import { pipeline } from "stream";
import { promisify } from "util";

const pipe = promisify(pipeline);
const client = new DeviceFarmClient({ region: "us-west-2" });

const sessionArn = "arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678

// 1) List artifacts for the session (FILE artifacts), handling pagination
const artifacts = [];
let nextToken;
do {
  const page = await client.send(new ListArtifactsCommand({
    arn: sessionArn,
    type: "FILE",
    nextToken
  }));
  artifacts.push(...(page.artifacts ?? []));
  nextToken = page.nextToken;
} while (nextToken);
```

```
if (!artifacts.length) throw new Error("No artifacts found");

// Strict filter to Appium logs
const filtered = (artifacts ?? []).filter(a =>
  a.type === "APPIUM_SERVER_OUTPUT" || a.type === "APPIUM_SERVER_LOG_OUTPUT"
);
if (!filtered.length) {
  throw new Error("No Appium server log artifacts found (expected
 APPIUM_SERVER_OUTPUT or APPIUM_SERVER_LOG_OUTPUT).");
}

// Prefer OUTPUT; else LOG_OUTPUT
const chosen =
  filtered.find(a => a.type === "APPIUM_SERVER_OUTPUT") ??
  filtered.find(a => a.type === "APPIUM_SERVER_LOG_OUTPUT");

const url = chosen.url;
const ext = chosen.extension || "log";
const outPath = `./appium_server_log.${ext}`;

// 2) Download the artifact
const resp = await fetch(url);
if (!resp.ok) {
  throw new Error(`Failed to download log: ${resp.status} ${await
 resp.text().catch(()=>"")}`);
}
await pipe(resp.body, createWriteStream(outPath));
console.log("Saved Appium server log to:", outPath);
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1', allowInsecure: [ 'execute_driver_script', ... ],
 useDrivers: [ 'uiautomator' ] }
```

C#

注意：此示例使用适用于.NET 的 AWS SDK 下载日志。*HttpClient*

在会话期间或之后，您可以使用设备群的 ListArtifactsAPI 来检索 Appium 服务器日志 URL，然后将其下载。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using System.Threading.Tasks;
using System.Linq;
using Amazon;
using Amazon.DeviceFarm;
using Amazon.DeviceFarm.Model;

class AppiumLogDownloader
{
    static async Task Main()
    {
        var sessionArn = "arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678

        using var client = new AmazonDeviceFarmClient(RegionEndpoint.USWest2);

        // 1) List artifacts for the session (FILE artifacts), handling pagination
        var all = new List<Artifact>();
        string? token = null;
        do
        {
            var page = await client.ListArtifactsAsync(new ListArtifactsRequest
            {
                Arn = sessionArn,
                Type = ArtifactCategory.FILE,
                NextToken = token
            });
            if (page.Artifacts != null) all.AddRange(page.Artifacts);
            token = page.NextToken;
        } while (!string.IsNullOrEmpty(token));

        if (all.Count == 0)
            throw new Exception("No artifacts found");

        // Strict filter to Appium logs
        var filtered = all.Where(a =>
            a.Type == "APPIUM_SERVER_OUTPUT" || a.Type ==
 "APPIUM_SERVER_LOG_OUTPUT").ToList();

        if (filtered.Count == 0)
```

```
            throw new Exception("No Appium server log artifacts found (expected
 APPIUM_SERVER_OUTPUT or APPIUM_SERVER_LOG_OUTPUT).");

        // Prefer OUTPUT; else LOG_OUTPUT
        var chosen = filtered.FirstOrDefault(a => a.Type == "APPIUM_SERVER_OUTPUT")
                    ?? filtered.First(a => a.Type == "APPIUM_SERVER_LOG_OUTPUT");

        var url = chosen.Url;
        var ext = string.IsNullOrWhiteSpace(chosen.Extension) ? "log" :
 chosen.Extension;
        var outPath = $"./appium_server_log.{ext}";

        // 2) Download the artifact
        using var http = new HttpClient();
        using var resp = await http.GetAsync(url,
 HttpCompletionOption.ResponseHeadersRead);
        resp.EnsureSuccessStatusCode();
        await using (var fs = File.Create(outPath))
        {
            await resp.Content.CopyToAsync(fs);
        }
        Console.WriteLine($"Saved Appium server log to:
 {Path.GetFullPath(outPath)}");
    }
}
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1', ..., useDrivers: [ 'uiautomator' ] }
```

Ruby

注意：此示例使用适用于 Ruby 的 AWS SDK 和*Net::HTTP*来下载日志。

在会话期间或之后，您可以使用设备群的 ListArtifactsAPI 来检索 Appium 服务器日志 URL，然后将其下载。

```
require "aws-sdk-devicefarm"
require "net/http"
require "uri"
```

```
client = Aws::DeviceFarm::Client.new(region: "us-west-2")
session_arn = "arn:aws:devicefarm:us-
west-2:111122223333:session:12345678-1111-2222-333-456789abcdef/12345678-1111-2222-333-45678

# 1) List artifacts for the session (FILE artifacts), handling pagination
artifacts = []
token = nil
loop do
  page = client.list_artifacts(arn: session_arn, type: "FILE", next_token: token)
  artifacts.concat(page.artifacts || [])
  token = page.next_token
  break if token.nil? || token.empty?
end

raise "No artifacts found" if artifacts.empty?

# Strict filter to Appium logs
filtered = (artifacts || []).select { |a| ["APPIUM_SERVER_OUTPUT",
 "APPIUM_SERVER_LOG_OUTPUT"].include?(a.type) }
raise "No Appium server log artifacts found (expected APPIUM_SERVER_OUTPUT or
 APPIUM_SERVER_LOG_OUTPUT)." if filtered.empty?

# Prefer OUTPUT; else LOG_OUTPUT
chosen = filtered.find { |a| a.type == "APPIUM_SERVER_OUTPUT" } ||
         filtered.find { |a| a.type == "APPIUM_SERVER_LOG_OUTPUT" }

url = chosen.url
ext = (chosen.extension && !chosen.extension.empty?) ? chosen.extension : "log"
out_path = "./appium_server_log.#{ext}"

# 2) Download the artifact
uri = URI.parse(url)
Net::HTTP.start(uri.host, uri.port, use_ssl: (uri.scheme == "https")) do |http|
  req = Net::HTTP::Get.new(uri)
  http.request(req) do |resp|
    raise "Failed GET: #{resp.code} #{resp.body}" unless resp.code.to_i / 100 == 2
    File.open(out_path, "wb") { |f| resp.read_body { |chunk| f.write(chunk) } }
  end
end
puts "Saved Appium server log to: #{File.expand_path(out_path)}"
```

这将显示如下输出：

```
info Appium Welcome to Appium v2.5.4
info Appium Non-default server args:
info Appium { address: '127.0.0.1', allowInsecure: [ 'execute_driver_script', ... ],
 useDrivers: [ 'uiautomator' ] }
```

支持的 Appium 功能和命令

Device Farm 的 Appium 端点支持您在本地设备上使用的大多数命令和所需功能，但少数例外情况除外。以下列表显示了当前不支持哪些功能和命令。如果由于功能受限，您的测试无法按预期运行，请提交支持案例以获取更多指导。

支持的功能

在 Device Farm 上创建 Appium 会话时，我们建议使用一组不同的功能，其中不包括本地设备特有的任何功能。在 Device Farm 上，如果设置了某些不支持的功能，则会话创建可能会失败。这包括特定于设备的功能，例如`udid`和。`platformVersion`此外，不支持 Android 和 iOS WebDriverAgent 上与之相关的某些功能，以及仅在模拟器和模拟器上支持的功能。 ChromeDriver

支持的 命令

大多数在真实安卓和 iOS 设备上正常运行的 Appium 命令都将按预期在 Device Farm 上运行，但以下情况除外：

Appium 设备命令 () **`/appium/device`**

- `install_app`
- `finger_print`
- `send_sms`
- `gsm_call`
- `gsm_signal`
- `gsm_voice`
- `power_ac`
- `power_capacity`
- `network_speed`
- `shake`

Appium 执行方法和脚本 () **/execute**

- installApp
- execEmuConsoleCommand
- fingerprint
- gsmCall
- gsmSignal
- sendSms
- gsmVoice
- powerAC
- powerCapacity
- networkSpeed
- sensorSet
- injectEmulatorCameraImage
- isGpsEnabled
- shake
- clearApp
- clearKeychains
- configureLocalization
- enrollBiometric
- getPasteboard
- installXCTestBundle
- listXCTestBundles
- listXCTestsInTestBundle
- runXCTest
- sendBiometricMatch
- setPasteboard
- setPermission
- startAudioRecording
- startLogsBroadcast
- startRecordingScreen

- startScreenStreaming
- startXCTestScreenRecording
- stopAudioRecording
- stopLogsBroadcast
- stopRecordingScreen
- stopScreenStreaming
- stopXCTestScreenRecording
- updateSafariPreferences

AWS Device Farm 中的私有设备

私有设备是 AWS Device Farm 在 Amazon 数据中心代表您部署的物理移动设备。此设备是您的 AWS 账户专有的。

> **Note**
>
> 目前，私有设备仅 AWS 在美国西部（俄勒冈）区域 (us-west-2) 可用。

如果您拥有一组私有设备，则可以使用私有设备创建远程访问会话和安排测试运行。有关更多信息，请参阅 在 AWS Device Farm 中创建测试运行或启动远程访问会话。您还可以创建实例配置文件来在远程访问会话或测试运行期间控制私有设备的行为。有关更多信息，请参阅 在 AWS Device Farm 中创建实例配置文件。或者，您可以请求将某些 Android 私有设备部署为 root 设备。

您还可以创建 Amazon Virtual Private Cloud 终端节点服务，以测试您的公司可以访问、但无法通过 Internet 访问的私有应用程序。例如，您可能有一个在 VPC 中运行的 Web 应用程序，您希望在移动设备上对其进行测试。有关更多信息，请参阅 将 Amazon VPC 端点服务与 Device Farm 结合使用 – 传统做法（不建议）。

如果您有兴趣使用私有设备队组，请联系我们。Device Farm 团队必须与您合作，为您的 AWS 账户设置和部署一组私有设备。

主题

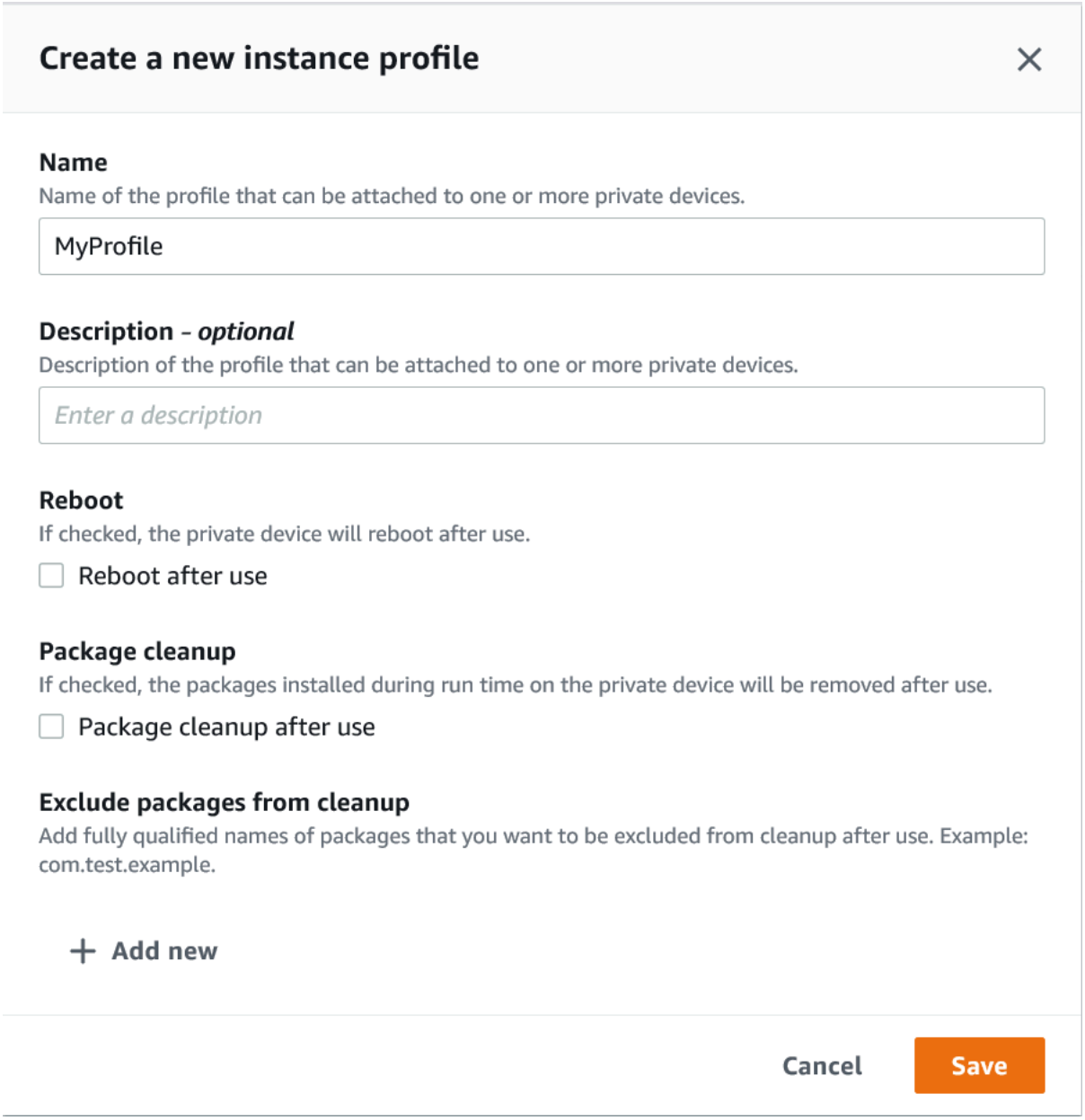
在 AWS Device Farm 中创建实例配置文件

您可以设置一个包含一个或多个私有设备的队组。这些设备专用于您的 AWS 账户。设置设备后，您可以选择为其创建一个或多个实例配置文件。实例配置文件可帮助您自动化测试运行并一致地将相同设置

应用于设备实例。实例配置文件还可以帮助您控制远程访问会话的行为。有关 Device Farm 中私有设备的更多信息，请参阅 AWS Device Farm 中的私有设备。

创建实例

1. 打开 Device Farm 控制台，网址为https://console.aws.amazon.com/devicefarm/。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择私有设备。
3. 选择 Instance profiles (实例配置文件)。
4. 选择新建实例配置文件。
5. 输入实例配置文件的名称。



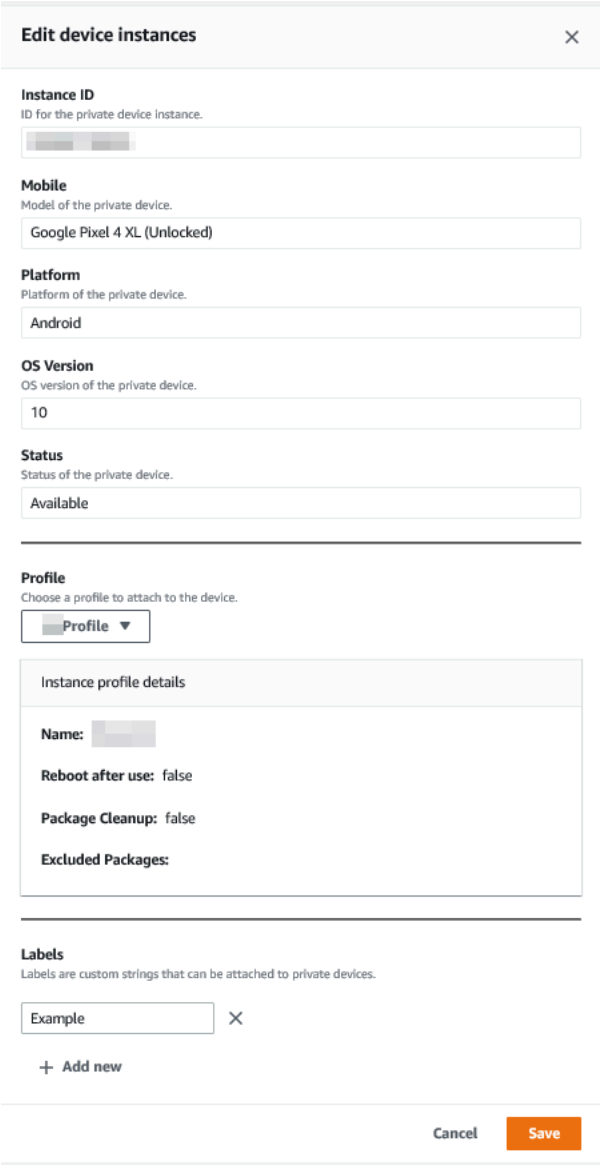
6. （可选）输入实例配置文件的描述。
7. （可选）更改以下任一设置以指定您希望 Device Farm 在每次测试运行或会话结束后要在设备上采取的操作：
   - 使用后重启 – 要重启设备，请选中此复选框。默认情况下，此复选框处于未选中状态 (`false`)。
   - 程序包清理 – 要删除已安装在设备上的所有应用程序包，请选中此复选框。默认情况下，此复选框处于未选中状态 (`false`)。要保留已安装在设备上的所有应用程序包，请取消选中此复选框。
   - 清理时需排除的程序包 – 要在设备上保留仅选中的应用程序包，请选中程序包清理复选框，然后选择新增。对于程序包名称，输入要在设备上保留的应用程序包的完全限定名称（例如，`com.test.example`）。要在设备上保留更多应用程序包，请选择 Add new (新增)，然后输入每个程序包的完全限定名称。
8. 选择保存。

在 AWS Device Farm 中请求额外的私有设备

在 AWS Device Farm 中，您可以请求将额外的私有设备实例添加到您的实例集中。您还可以查看和更改实例集中现有私有设备实例的设置。有关私有设备的更多信息，请参阅 AWS Device Farm 中的私有设备。

请求额外的私有设备或更改其设置

1. 打开 Device Farm 控制台，网址为https://console.aws.amazon.com/devicefarm/。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择私有设备。
3. 选择 Device instances (设备实例)。Device instances (设备实例) 选项卡显示位于您的队组中的私有设备表。要快速搜索或筛选表，请在列上方的搜索栏中输入搜索词。
4. 要请求新的私有设备实例，请选择请求设备实例或联系我们。私有设备需要在 Device Farm 团队的帮助下进行额外的设置。
5. 选择要查看或管理相关信息的实例旁边的切换选项，然后选择编辑。



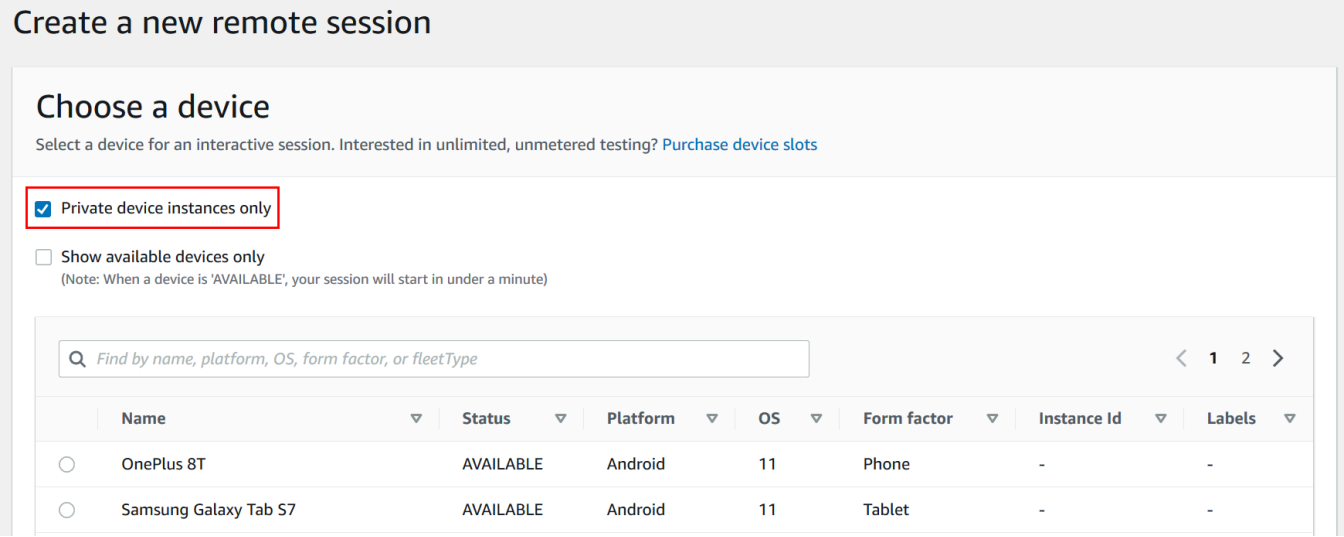
6. 要将某个实例配置文件附加到设备实例，请从配置文件下拉列表中选择该实例配置文件。例如，如果您要从清理任务中始终排除某特定应用程序包，那么附加实例配置文件会很有用。有关将实例配置文件与设备结合使用的更多信息，请参阅在 AWS Device Farm 中创建实例配置文件。
7. （可选）在 Labels (标签) 下，选择 Add new (新增) 以将标签添加到设备实例。标签可以帮助您更轻松地对设备进行分类并查找特定设备。
8. 选择保存。

在 AWS Device Farm 中创建测试运行或启动远程访问会话

在 AWS Device Farm 中，设置私有设备实例集后，您可以使用该实例集中的一个或多个私有设备创建测试运行或启动远程访问会话。有关私有设备的更多信息，请参阅 AWS Device Farm 中的私有设备。

创建测试运行或启动远程访问会话

1. 打开 Device Farm 控制台，网址为https://console.aws.amazon.com/devicefarm/。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 从列表中选择一个现有项目或创建一个新项目。要创建新项目，请选择新建项目，输入项目的名称，然后选择提交。
4. 请执行以下操作之一：
   - 要创建测试运行，请选择 Automated tests (自动测试)，然后选择 Create a new run (创建新运行)。此向导将指导您完成创建运行的步骤。在 “选择设备” 步骤中，您可以编辑现有设备池或创建仅包含 Device Farm 团队设置并与您的 AWS 帐户关联的私有设备的新设备池。有关更多信息，请参阅 the section called “创建私有设备池”。
   - 要启动远程访问会话，请选择 Remote access (远程访问)，然后选择 Start a new session (启动新会话)。在 “选择设备” 页面上，选择 “仅限私有设备实例”，将列表限制为 Device Farm 团队设置并与您的 AWS 账户关联的私有设备。然后，选择要访问的设备，输入远程访问会话的名称，然后选择 Confirm and start session (确认并启动会话)。



在 AWS Device Farm 的设备池中选择私有设备

要在测试运行中使用私有设备，您可以创建一个设备池来选择您的私有设备。设备池允许您主要通过三种类型的设备池规则来选择私有设备：

1. 基于设备 ARN 的规则

2. 基于设备实例标签的规则
3. 基于设备实例 ARN 的规则

在以下各节中，将深入介绍每种规则类型及其用例。您可以使用 Device Farm 控制台、 AWS 命令行界面 (AWS CLI) 或 Device Farm API 使用这些规则创建或修改带有私有设备的设备池。

主题

设备 ARN

设备 ARN 是一个标识符，代表一种设备类型，而不是任何特定的物理设备实例。设备类型由以下属性定义：

- 设备的实例集 ID
- 设备的 OEM
- 设备的型号
- 设备的操作系统的版本
- 指示设备是否已取得 root 权限的设备状态

许多物理设备实例可以由单个设备类型表示，其中该类型的每个实例对这些属性具有相同的值。例如，如果您的私有队列*16.1.0*中有三台 iOS 版本的*Apple iPhone 13*设备，则每台设备将共享相同的设备 ARN。如果在您的设备队组中添加或移除任何具有相同属性的设备，则设备 ARN 将继续代表您的设备队组中该设备类型的所有可用设备。

设备 ARN 是为设备池选择私有设备的最可靠方法，因为它允许设备池继续选择设备，无论您在任何给定时间部署了哪些特定设备实例。单个私有设备实例可能会遇到硬件故障，这会提示 Device Farm 自动将其替换为相同设备类型的新工作实例。在这些情况下，设备 ARN 规则可确保您的设备池在出现硬件故障时可以继续选择设备。

当您对设备池中的私有设备使用设备 ARN 规则并安排使用该池的测试运行时，Device Farm 将自动检查该设备 ARN 代表哪些私有设备实例。在当前可用的实例中，系统将分配其中一个实例来来运行您的测试。如果当前没有可用的实例，Device Farm 将等待该设备 ARN 的第一个可用实例变为可用，然后分配该实例以运行您的测试。

设备实例标注

设备实例标签是您可以作为设备实例的元数据附加的文本标识符。您可以将多个标签附加到每个设备实例，将同一个标签附加到多个设备实例。有关在设备实例中添加、修改或移除设备标签的更多信息，请参阅管理私有设备。

设备实例标签可能是为设备池选择私有设备的有效方法，因为如果您有多个具有相同标签的设备实例，则它允许设备池从其中任何一个实例中进行选择以进行测试。如果设备 ARN 不适合您的用例（例如，如果您想从多种设备类型的设备中进行选择，或者想要从某一设备类型的所有设备的子集中进行选择），那么设备实例标签可以让您在设备池的多个设备中进行更精细的选择。单个私有设备实例可能会遇到硬件故障，这会提示 Device Farm 自动将其替换为相同设备类型的新工作实例。在这些情况下，替换设备实例将不会保留被替换设备的任何实例标签元数据。因此，如果您将相同的设备实例标签应用于多个设备实例，则设备实例标签规则可确保您的设备池在出现硬件故障时可以继续选择设备实例。

当您对设备池中的私有设备使用设备实例标签规则并安排使用该池的测试运行时，Device Farm 将自动检查哪些私有设备实例由该设备实例标签表示，并从这些实例中随机选择一个可用于运行测试的实例。如果没有可用的设备，Device Farm 将随机选择任何带有设备实例标签的设备实例来运行您的测试，并在设备可用时将测试排队在设备上运行。

实例 ARN

设备实例 ARN 是一个标识符，代表部署在私有设备队组中的物理裸机设备实例。例如，如果您的私有队列*15.0.0*中的操作系统上有三*iPhone 13*台设备，而每台设备共享相同的设备 ARN，则每台设备也将有自己的实例 ARN，仅代表该实例。

设备实例 ARN 是为设备池选择私有设备的最不稳健的方法，仅当设备 ARNs 和设备实例标签不符合您的用例时，才建议使用该方法。当以独特和特定的方式配置特定设备实例作为测试的先决条件，并且在对其进行测试之前需要知道和验证该配置时，设备实例通常被用作设备池的规则。 ARNs 单个私有设备实例可能会遇到硬件故障，这会提示 Device Farm 自动将其替换为相同设备类型的新工作实例。在这些情况下，替换设备实例与被替换设备的设备实例 ARN不同。因此，如果您的设备池依赖设备实例 ARNs ，则需要手动将设备池的规则定义从使用旧 ARN 更改为使用新 ARN。如果您确实需要手动预配置设备以进行测试，那么这可能是一个有效的工作流程（与设备相比 ARNs）。要进行大规模测试，建议尝试调整这些用例以使用设备实例标签，并在可能的情况下预先配置多个设备实例进行测试。

当您对设备池中的私有设备使用设备实例 ARN 规则并计划使用该池进行测试时，Device Farm 会自动将测试分配给该设备实例。如果该设备实例不可用，Device Farm 将对在其可用后在设备上进行的测试进行排队。

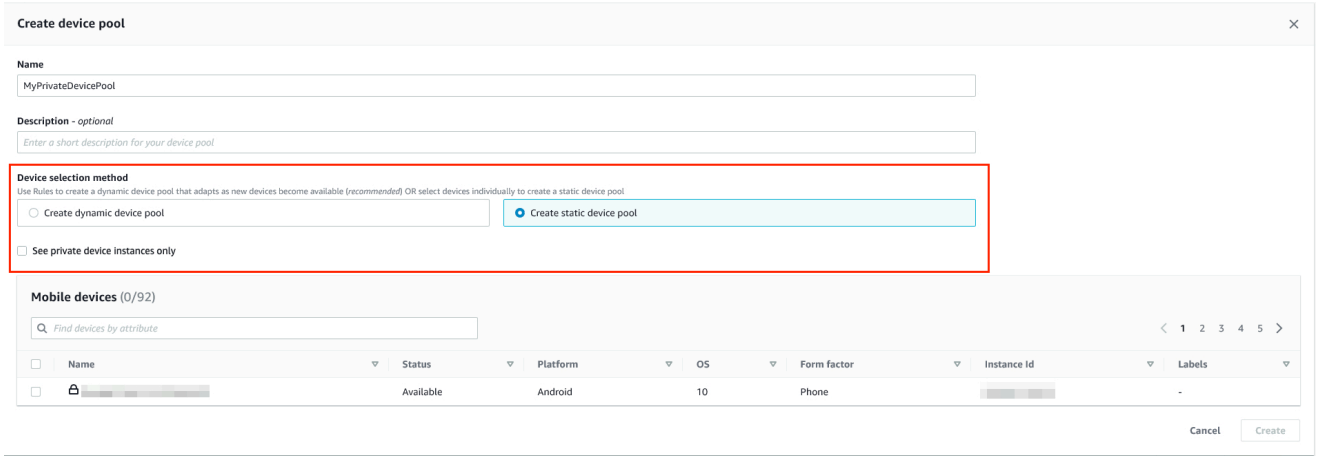
使用私有设备创建私有设备池（控制台）

当您创建测试运行测试，可以创建一个用于测试运行的设备池，并确保该池仅包含您的私有设备。

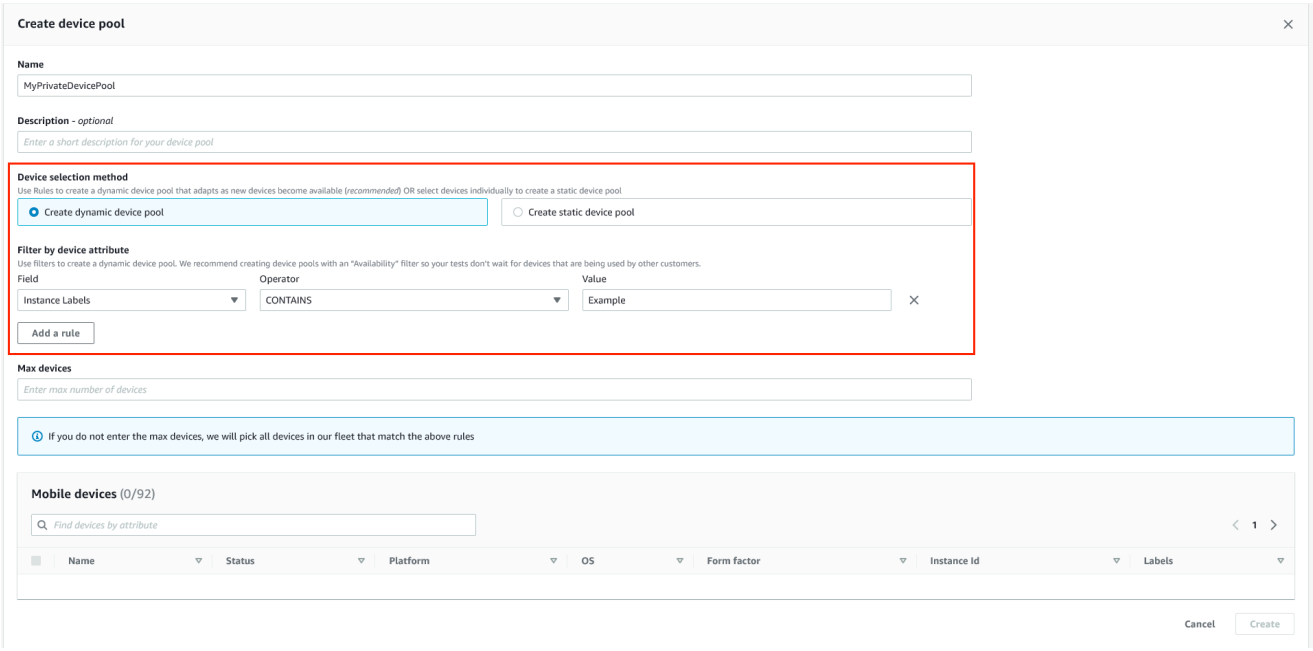
> **Note**
>
> 在控制台中创建包含私有设备的设备池时，只能使用三个可用规则中的任何一个来选择私有设备。如果要创建包含多种私有设备规则的设备池（例如，包含设备 ARNs 和设备实例规则的设备池 ARNs），则需要通过 CLI 或 API 创建该池。

1. 打开 Device Farm 控制台，网址为https://console.aws.amazon.com/devicefarm/。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 从列表中选择一个现有项目或创建一个新项目。要创建新项目，请选择新建项目，输入项目的名称，然后选择提交。
4. 选择项目设置，然后导航到设备池选项卡。
5. 选择创建设备池，然后输入设备池的名称和可选描述。

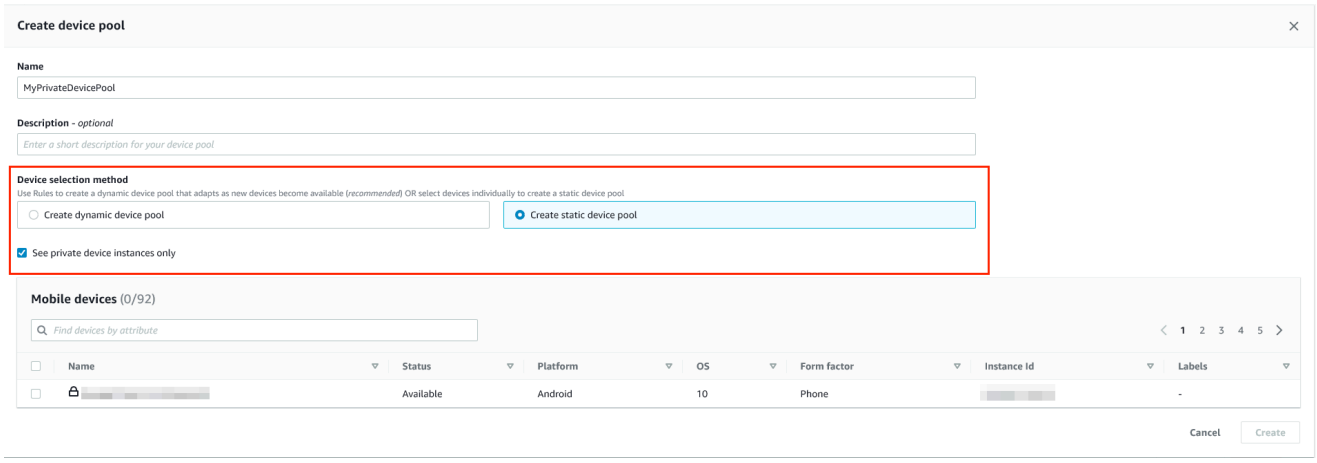
   a. 要对设备池使用设备 ARN 规则，请选择创建静态设备池，然后从列表中选择要在设备池中使用的特定设备类型。不要选择私仅有设备实例，因为此选项会导致使用设备实例 ARN 规则（而不是设备 ARN 规则）创建设备池。



b. 要对设备池使用设备实例标签规则，请选择创建动态设备池。然后，对于要在设备池中使用的每个标签，选择添加规则。对于每条规则，选择实例标签作为 `Field`，选择包含作为 `Operator`，然后将所需的设备实例标签指定为`Value`。



c. 要对设备池使用设备实例 ARN 规则，请选择创建静态设备池，然后选择仅私有设备实例，将设备列表限制为仅显示 Device Farm 已与您的 AWS 账户关联的私有设备实例。



6. 选择创建。

使用私有设备 (AWS CLI) 创建私有设备池

- 运行 create-device-pool 命令。

有关将 Device Farm 与配合使用的信息 AWS CLI，请参阅AWS CLI 参考文档。

使用私有设备 (API) 创建私有设备池

- 调用 `CreateDevicePool` API。

有关如何使用 Device Farm API 的更多信息，请参阅 自动化 Device Farm。

在 AWS Device Farm 中跳过私有设备上的应用程序重新签名

应用程序签名是通过私钥对应用程序包（如 APK、IPA）进行数字签名的过程，完成后才能将此应用程序包安装在设备上或发布到 Google Play Store 或 Apple App Store 等应用商店。为通过减少所需签名和配置文件的数量来优化测试流程，并提升远程设备上的数据安全性，AWS Device Farm 将在您的应用程序上传到服务后对其进行重新签名。

将应用程序上传到 AWS Device Farm 后，该服务将使用自己的签名证书和预置配置文件为该应用程序生成新的签名。此过程将原始应用程序签名替换为 AWS Device Farm 的签名。然后，重新签名的应用程序将安装在 AWS Device Farm 提供的测试设备上。获得新签名后，应用程序无需原始开发者的证书即可在这些设备上安装和运行。

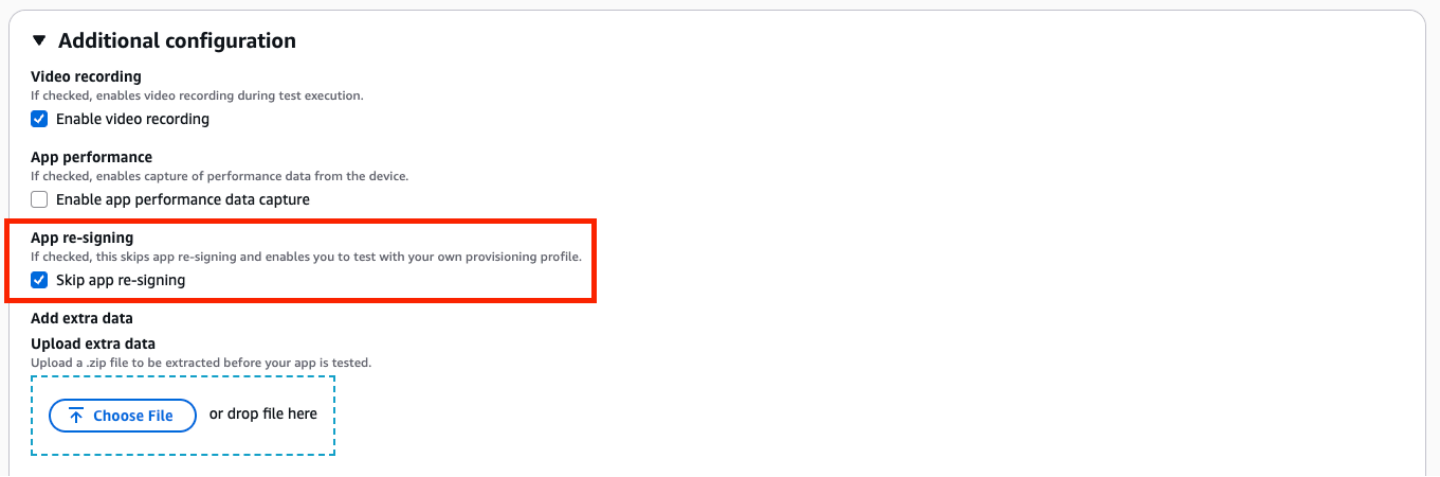
在 iOS 上，我们将嵌入式配置文件替换为通配符配置文件并重新签署应用程序。如果您提供辅助数据，我们将在安装前将其添加到应用程序包中，这样这些数据就会出现在您的应用程序的沙盒中。重新签名 iOS 应用程序会导致所有权利被删除。

在安卓系统上，我们会重新签署应用程序。这可能会破坏依赖于应用程序签名的功能，例如 Google Maps Android API。它还可能触发产品提供的反盗版和防篡改检测，例如。 DexGuard对于内置测试，我们可能会修改清单以包含捕获和保存屏幕截图所需的权限。

如果您使用私有设备，则可以跳过 AWS Device Farm 对您的应用程序重新签名的步骤。这一点与公有设备不同，对于公有设备，Device Farm 始终会对 Android 和 iOS 平台上的应用程序重新签名。

您可以在创建远程访问会话或测试运行时跳过应用程序重新签名。如果您的应用程序功能在 Device Farm 对您的应用程序重新签名时发生崩溃，这会很有用。例如，重新签名后推送通知可能无法正常工作。有关 Device Farm 在测试您的应用程序时所做的更改的更多信息，请参阅 AWS Device Farm FAQs 或应用程序页面。

要跳过测试运行中的应用程序重新签名，请在其他配置下选择跳过应用程序重新签名。此选项仅在私有设备上可用。



> **Note**
>
> 如果您使用的是 XCTest 框架，则跳过应用程序重新签名选项不可用。有关更多信息，请参阅将 Device Farm 与 XCTest 适用于 iOS 的集成。

根据您使用的是私有 Android 设备还是 iOS 设备，配置应用程序签名设置的其他步骤有所不同。

在 Android 设备上跳过应用程序重新签名

如果您在私有 Android 设备上测试应用程序，请在创建测试运行或远程访问会话时选择 Skip app re-signing (跳过应用程序重新签名)。无需其他配置。

在 iOS 设备上跳过应用程序重新签名

Apple 要求您在将用于测试的应用程序加载到设备上之前对应用程序签名。对于 iOS 设备，您有两种选择来对应用程序签名。

- 如果您使用的是内部（企业）开发人员配置文件，则可以跳到下一部分the section called “创建远程访问会话，以信任您的应用程序”。

- 如果您使用的是临时 iOS 应用程序开发配置文件，则必须先将设备注册到您的 Apple 开发人员账户，然后更新您的预配置配置文件，使其包含私有设备。然后，您必须使用更新的预配置配置文件对应用程序重新签名。然后可以在 Device Farm 中运行重新签名的应用程序。

将设备注册到临时 iOS 应用程序开发预配置配置文件

1. 登录您的 Apple 开发人员账户。
2. 导航到控制台的 “证书 IDs、和配置文件” 部分。
3. 转到 Devices (设备)。
4. 在您的 Apple 开发人员账户中注册该设备。要获取设备的名称和 UDID，请使用 Device Farm API 的 `ListDeviceInstances` 操作。
5. 转到您的预配置配置文件，然后选择 Edit (编辑)。
6. 从列表中选择设备。
7. 在 XCode 中获取经过更新的预配置配置文件，然后对应用程序重新签名。

无需其他配置。现在，您可以创建远程访问会话或测试运行，然后选择 Skip app re-signing (跳过应用程序重新签名)。

创建远程访问会话，以信任您的 iOS 应用程序

如果您使用的是内部（企业）开发人员预配置配置文件，则必须在每台私有设备上执行一次性过程，以信任内部应用程序开发人员证书。

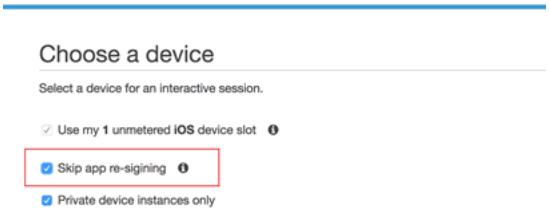
为此，您必须安装占位符应用程序，该应用程序使用与您要测试的应用程序相同的证书进行签名。在设备信任配置文件或企业应用程序开发者之后，该开发者的所有应用程序都将在私有设备上受到信任，直到您将其删除。因此，当您安装要测试的应用程序的新版本时，您不必每次都再次信任应用程序开发者。如果您运行测试自动化，不希望在每次测试您的应用程序时都创建远程访问会话，这种方法特别有用。

许多客户常用的程序是重新签名 iOS 版 Device Farm 示例应用程序，然后将其作为占位符应用安装到他们的设备上。

在启动远程访问会话之前，请按照 在 AWS Device Farm 中创建实例配置文件 中的步骤在 Device Farm 中创建或修改实例配置文件。在实例配置文件中，将占位符应用程序的捆绑包 ID 添加到 “从清理中排除软件包” 设置中。然后，将实例配置文件附加到私有设备实例，以确保 Device Farm 在启动新的测试运行之前不会从设备中删除该应用程序。这样可确保您的开发人员证书仍是可信的。

您可以使用远程访问会话将占位符应用程序上传到设备，这样您就可以启动应用程序并信任开发者。

1. 按照创建会话中的说明操作，创建使用您创建的私有设备实例配置文件的远程访问会话。在创建会话时，请务必选择 Skip app re-signing (跳过应用程序重新签名)。



> **Important**
>
> 要将设备列表筛选为仅包含私有设备，请选择 Private device instances only (仅限私有设备实例)，以确保您使用的是具有正确实例配置文件的私有设备。

此外，请务必将占位符应用程序或要测试的应用程序添加到附加到此实例的实例配置文件的 “从清理中排除软件包” 设置中。

2. 您的远程会话启动时，请选择选择文件来安装使用内部预置配置文件的应用程序。
3. 启动您刚刚上传的应用程序。
4. 确认出现 iOS 对话框，指示企业应用程序开发者不受信任。
5. 然后，如果 iOS 设备使用的是 iOS 版本 18 或更高版本，请向 Device Farm 团队提交支持请求，让我们的团队信任您的应用程序，因为这些设备需要手动信任该应用程序。 AWS 否则，如果iOS版本为17或更低，则可以进入设置应用程序，然后在常规设置下，从 VPN和配置文件菜单中自己信任该应用程序。

现在，该配置文件或企业应用程序开发人员的所有应用程序在此私有设备上都是可信的，直到您删除这些应用程序。

AWS Device Farm 中跨 AWS 区域的 Amazon VPC

Device Farm 服务仅位于美国西部（俄勒冈州）(`us-west-2`) 区域。您可以使用亚马逊虚拟私有云（亚马逊 VPC）通过 Device Farm 访问其他 AWS 区域的亚马逊虚拟私有云中的服务。如果 Device Farm 和您的服务位于同一区域，请参阅 将 Amazon VPC 端点服务与 Device Farm 结合使用 – 传统做法（不建议）。

有两种方法可以访问您位于其他区域的私有服务。如果您的服务位于另一个非 `us-west-2` 的区域，则您可以使用 VPC 对等连接将该区域的 VPC 对等连接到与 `us-west-2` 中的 Device Farm 连接的另一个 VPC。但是，如果您在多个区域提供服务，Transit Gateway 将允许您通过更简单的网络配置访问这些服务。

有关更多信息，请参阅《Amazon VPC 对等连接指南》中的 VPC 对等方案。

AWS Device Farm VPCs 中不同区域的 VPC 对等互连概述

只要不同区域 VPCs 中的任意两个区域具有不同的、不重叠的 CIDR 块，您就可以对其进行对等。这可确保所有私有 IP 地址都是唯一的，并且允许中的所有资源互相寻址，而无需进行任何形式的网络地址转换 (NAT)。 VPCs有关 CIDR 表示法的更多信息，请参阅 RFC 4632。

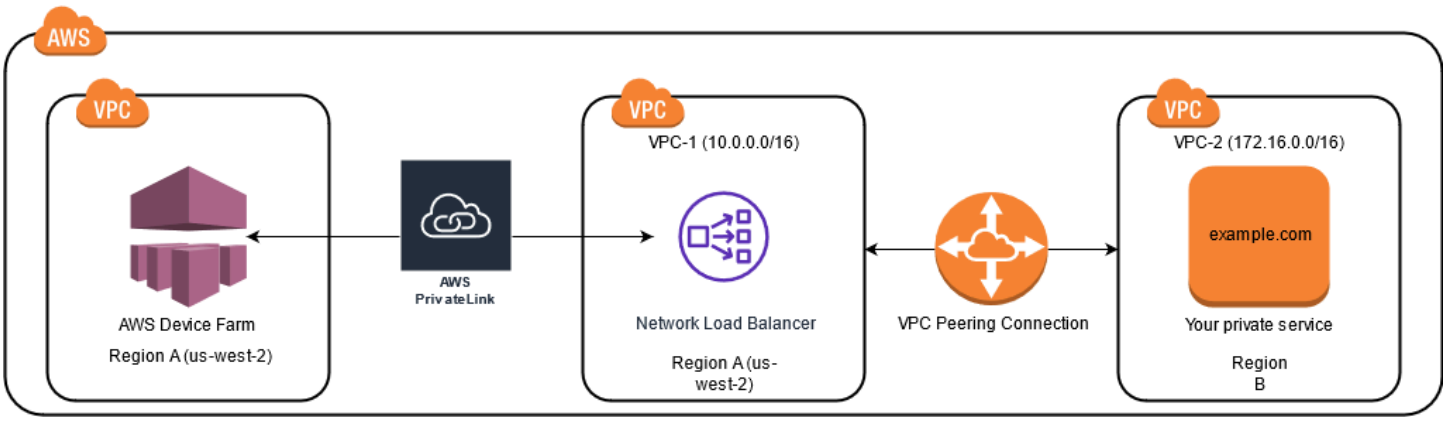
本主题包含一个跨区域示例场景，其中，Device Farm（称作 VPC-1）位于美国西部（俄勒冈州）(`us-west-2`) 区域。本示例中的第二个 VPC（称作 VPC-2）位于另一个区域。

Device Farm VPC 跨区域示例

VPC 组件	VPC-1	VPC-2
CIDR	10.0.0.0/16	172.16.0.0/16

> **Important**
>
> 在两者之间建立对等连接 VPCs 可以改变的安全状态。 VPCs此外，向其路由表中添加新条目可能会更改路由表中资源的安全状况 VPCs。您有责任以满足组织安全要求的方式实施这些配置。有关更多信息，请参阅责任共担模式。

下图显示了示例中的组件以及这些组件之间的交互。



主题

- 在 AWS Device Farm 中使用 Amazon VPC 的先决条件
- 步骤 1：在 VPC-1 和 VPC-2 之间建立对等连接

- 步骤 2：更新 VPC-1 和 VPC-2 中的路由表
- 步骤 3：创建目标组
- 步骤 4：创建 Network Load Balancer
- 步骤 5：创建 VPC 端点服务以将您的 VPC 连接到 Device Farm
- 步骤 6：在 VPC 与 Device Farm 之间创建 VPC 端点配置
- 步骤 7：创建测试运行以使用 VPC 端点配置
- 使用 Transit Gateway 创建可扩展的网络

在 AWS Device Farm 中使用 Amazon VPC 的先决条件

此示例要求以下内容：

- 两个配置 VPCs 的子网包含不重叠的 CIDR 块。
- VPC-1 必须位于 us-west-2 区域并且包含可用区 us-west-2a、us-west-2b 和 us-west-2c 的子网。

有关创建 VPCs 和配置子网的更多信息，请参阅 Amazon VPC 对等互连指南中的使用 VPCs 和子网。

步骤 1：在 VPC-1 和 VPC-2 之间建立对等连接

在 VPCs 包含不重叠的 CIDR 块的两者之间建立对等连接。为此，请参阅《Amazon VPC 对等连接指南》中的创建和接受 VPC 对等连接。使用本主题的跨区域场景和《Amazon VPC 对等连接指南》，创建了以下对等连接配置示例：

名称

```
Device-Farm-Peering-Connection-1
```

VPC ID（请求者）

```
vpc-0987654321gfedcba (VPC-2)
```

Account

```
My account
```

区域

```
US West (Oregon) (us-west-2)
```

VPC ID（接受者）

```
vpc-1234567890abcdefg (VPC-1)
```

> **Note**
>
> 在建立任何新的对等连接时，请务必查阅您的 VPC 对等连接配额。有关更多信息，请参阅《Amazon VPC 对等连接指南》中的 Amazon VPC 配额。

步骤 2：更新 VPC-1 和 VPC-2 中的路由表

设置对等连接后，您必须在两者之间建立目的地路由， VPCs 以便在两者之间传输数据。要建立此路由，您可以手动更新 VPC-1 的路由表以指向 VPC-2 的子网，反之亦然。为此，请参阅《Amazon VPC 对等连接指南》中的为 VPC 对等连接更新路由表。使用本主题的跨区域场景和《Amazon VPC 对等连接指南》，创建了以下示例路由表配置：

Device Farm VPC 路由表示例

VPC 组件	VPC-1	VPC-2
路由表 ID	rtb-1234567890abcdefg	rtb-0987654321gfedcba
本地地址范围	10.0.0.0/16	172.16.0.0/16
目标地址范围	172.16.0.0/16	10.0.0.0/16

步骤 3：创建目标组

设置目标路由后，您可以在 VPC-1 中配置网络负载均衡器，将请求路由到 VPC-2。

网络负载均衡器必须首先包含一个目标组，该目标组包含请求发送到的 IP 地址。

要创建目标组

1. 确定要在 VPC-2 中定位的服务的 IP 地址。
   - 这些 IP 地址必须是对等连接中使用的子网的成员。
   - 目标 IP 地址必须是不可变的静态地址。如果您的服务具有动态 IP 地址，请考虑将静态资源（例如网络负载均衡器）作为目标，然后让该静态资源将请求路由到您的真实目标。

> **Note**
>
> - 如果您的目标是一个或多个独立的亚马逊弹性计算云 (Amazon EC2) 实例，请打开https://console.aws.amazon.com/ec2/位于的亚马逊 EC2 控制台，然后选择实例。
> - 如果您的目标是 Amazon EC2 实例的 Amazon EC2 Auto Scaling 组，则必须将 Amazon EC2 Auto Scaling 组与网络负载均衡器关联起来。有关更多信息，请参阅《Amazon EC2 Auto Scaling 用户指南》中的将负载均衡器挂接到自动扩缩组。
>
> 然后，您可以在上打开 Amazon EC2 控制台 https://console.aws.amazon.com/ec2/，然后选择网络接口。从那里您可以查看每个可用区中网络负载均衡器的每个网络接口的 IP 地址。

2. 在 VPC-1 中创建目标组。有关更多信息，请参阅《Network Load Balancer 用户指南》中的为 Network Load Balancer 创建目标组。

   不同 VPC 中服务的目标组需要以下配置：

   - 在 选择目标类型中，选择 IP 地址。
   - 对于 VPC，请选择将托管负载均衡器的 VPC。对于主题示例，这将是 VPC-1。
   - 在注册目标页面上，为 VPC-2 中的每个 IP 地址注册一个目标。

     对于网络，选择其他私有 IP 地址。

     对于可用区，请在 VPC-1 中选择所需的区域。

     对于IPv4 地址，请选择 VPC-2 IP 地址。

     对于端口，请选择您的端口。
   - 选择在下面以待注册的形式添加。指定完地址后，选择注册待注册目标。

使用本主题的跨区域场景和网络负载均衡器用户指南，在目标组配置中使用了以下值：

Target type

```
IP addresses
```

目标组名称

```
my-target-group
```

协议/端口

```
TCP : 80
```

VPC

```
vpc-1234567890abcdefg (VPC-1)
```

Network

```
Other private IP address
```

可用区

```
all
```

IPv4 address

```
172.16.100.60
```

端口

```
80
```

步骤 4：创建 Network Load Balancer

使用步骤 3 中描述的目标组创建网络负载均衡器。为此，请参阅创建 Network Load Balancer。

使用本主题的跨区域场景，在网络负载均衡器配置示例中使用了以下值：

负载均衡器名称

```
my-nlb
```

Scheme

```
Internal
```

VPC

```
vpc-1234567890abcdefg (VPC-1)
```

映射

```
us-west-2a - subnet-4i23iuufkdiufsloi
```

```
us-west-2b - subnet-7x989pkjj78nmn23j

us-west-2c - subnet-0231ndmas12bnnsds
```

协议/端口

```
TCP : 80
```

目标组

```
my-target-group
```

步骤 5：创建 VPC 端点服务以将您的 VPC 连接到 Device Farm

您可以使用网络负载均衡器创建 VPC 终端节点服务。通过此 VPC 终端节点服务，Device Farm 无需任何其他基础设施（例如互联网网关、NAT 实例或 VPN 连接）即可连接到 VPC-2 中的服务。

为此，请参阅在 Device Farm 中创建 VPC 终端节点配置。

步骤 6：在 VPC 与 Device Farm 之间创建 VPC 端点配置

您现在可以在 VPC 和 Device Farm 之间建立私有连接。您可以使用 Device Farm 来测试私有服务，而无需通过公共互联网公开这些服务。为此，请参阅在 Device Farm 中创建 VPC 终端节点配置。

使用本主题的跨区域场景，在 VPC 终端节点配置示例中使用了以下值：

名称

```
My VPCE Configuration
```

VPCE 服务名称

```
com.amazonaws.vpce.us-west-2.vpce-svc-1234567890abcdefg
```

服务 DNS 名称

```
devicefarm.com
```

步骤 7：创建测试运行以使用 VPC 端点配置

您可以创建使用步骤 6 中所述的 VPC 终端节点配置的测试运行。有关更多信息，请参阅 在 Device Farm 中创建测试运行或 创建会话。

使用 Transit Gateway 创建可扩展的网络

要使用两个以上的网络创建可扩展网络 VPCs，您可以使用 Transit Gateway 充当网络传输中心，将您的网络 VPCs 和本地网络互连。要将与 Device Farm 位于同一区域的 VPC 配置为使用 Transit Gateway，您可以按照 Amazon VPC endpoint services with Device Farm 指南，根据其私有 IP 地址将资源定位到其他区域。

有关中转网关的更多信息，请参阅《Amazon VPC Transit Gateway 指南》中的什么是中转网关？。

在 Device Farm 中终止私有设备

要在最初的约定期限之后终止私人设备，您必须通过我们的电子邮件地址 <aws-devicefarm-support@amazon> .com 提供 30 天不续订通知。有关私有设备的更多信息，请参阅 AWS Device Farm 中的私有设备。

> **Important**
>
> 这些说明仅适用于终止私有设备协议。有关所有其他 AWS 服务和计费问题，请参阅这些产品的相应文档或联系 AWS 支持人员。

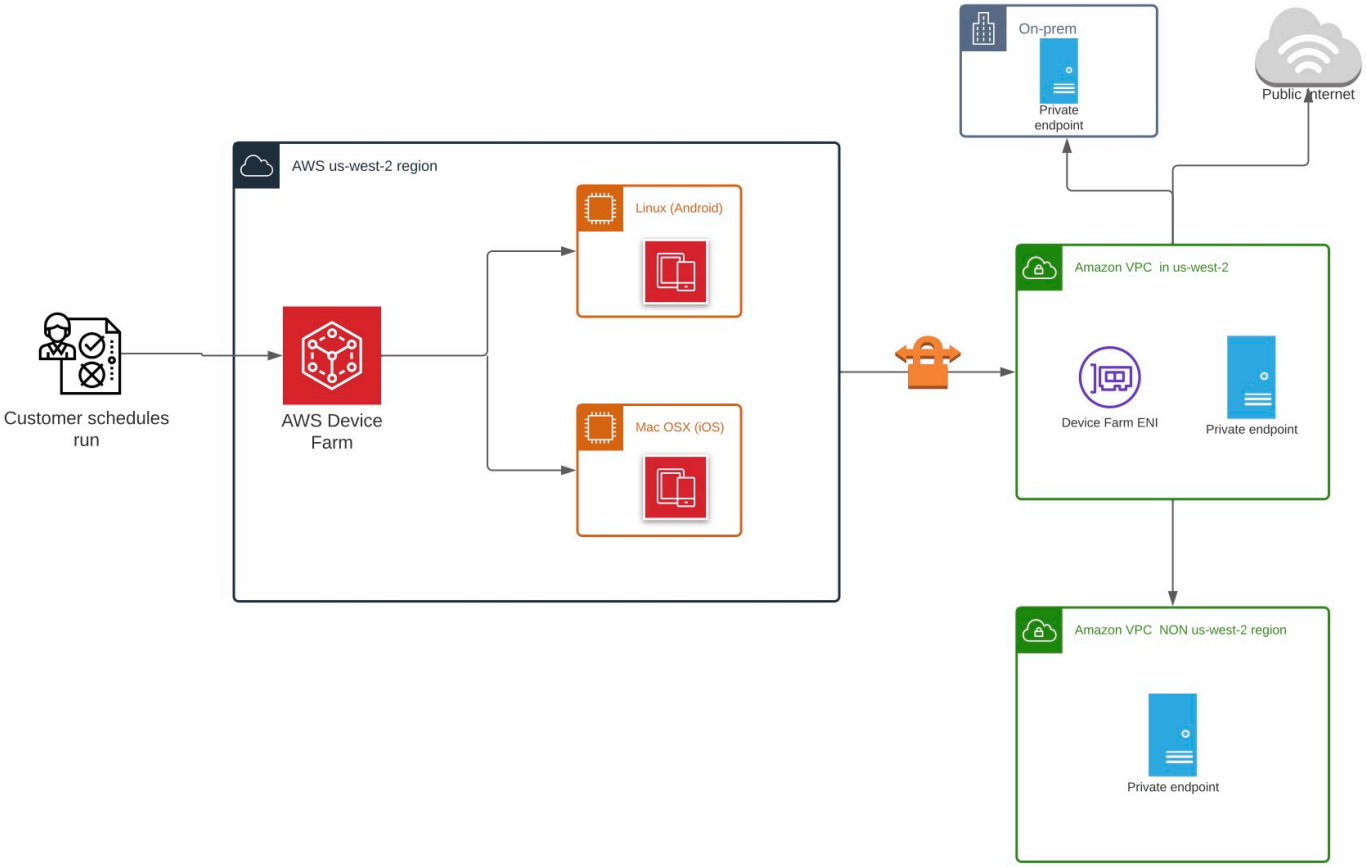
AWS Device Farm 中的 VPC-ENI

> **Warning**
>
> 此功能仅在私有设备上可用。要在您的 AWS 账户中申请使用私人设备，请联系我们。如果您的 AWS 账户中已经添加了私有设备，我们强烈建议您使用这种 VPC 连接方法。

AWS Device Farm 的 VPC-ENI 连接功能可帮助客户安全地连接到托管在本地软件或其他云提供 AWS 商上的私有终端节点。

您可以将 Device Farm 移动设备及其主机连接到us-west-2该地区的亚马逊虚拟私有云 (Amazon VPC) 环境，从而允许通过弹性网络接口访问隔离的 non-internet-facing服务和应用程序。有关更多信息 VPCs，请参阅 Amazon VPC 用户指南。

如果您的私有终端节点或 VPC 不在 us-west-2 区域内，则可以使用 Transit Gateway 或 VPC 对等连接等解决方案将其与 us-west-2 区域的 VPC 关联起来。在这种情况下，Device Farm 将在您为 us-west-2 区域 VPC 提供的子网中创建 ENI，并且您将负责确保可以在 us-west-2 区域 VPC 与其他区域的 VPC 之间建立连接。



有关使用 AWS CloudFormation 自动创建和对等操作的信息 VPCs，请参阅上VPCPeering 模板存储库中的 AWS CloudFormation 模板 GitHub。

> **Note**
>
> Device Farm 不会为在客户的 VPC ENIs 中创建而收取任何费用us-west-2。此功能不包括跨区域或外部 VPC 间连接的费用。

配置了 VPC 访问权限后，除非在 VPC 中指定了 NAT 网关，否则用于测试的设备和主机将无法连接到 VPC 以外的资源（例如公用 CDNs）。有关更多信息，请参阅《Amazon VPC 用户指南》中的 NAT 网关。

主题

AWS 访问控制和 IAM

AWS Device Farm 允许您使用 AWS Identity and Access Management (IAM) 创建策略，以授予或限制对 Device Farm 功能的访问权限。要在 AWS Device Farm 中使用 VPC 连接功能，您用于访问 AWS Device Farm 的用户账户或角色需要以下 IAM policy：

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [{
      "Effect": "Allow",
      "Action": [
        "devicefarm:*",
        "ec2:DescribeVpcs",
        "ec2:DescribeSubnets",
        "ec2:DescribeSecurityGroups",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iam:CreateServiceLinkedRole",
      "Resource": "arn:aws:iam::*:role/aws-service-role/devicefarm.amazonaws.com/
AWSServiceRoleForDeviceFarm",
      "Condition": {
        "StringLike": {
```

```
          "iam:AWSServiceName": "devicefarm.amazonaws.com"
        }
      }
    }
  ]
}
```

要创建或更新具有 VPC 配置的 Device Farm 项目，您的 IAM policy 必须允许您对 VPC 配置中列出的资源调用以下操作：

```
"ec2:DescribeVpcs"
"ec2:DescribeSubnets"
"ec2:DescribeSecurityGroups"
"ec2:CreateNetworkInterface"
```

此外，您的 IAM policy 还必须允许创建服务相关角色：

```
"iam:CreateServiceLinkedRole"
```

> **Note**
>
> 在项目中不使用 VPC 配置的用户不需要这些权限。

服务关联角色

AWS Device Farm 使用 AWS Identity and Access Management (IAM) 服务相关角色。服务相关角色是一种独特类型的 IAM 角色，它与 Device Farm 直接相关。服务相关角色由 Device Farm 预定义，包括该服务代表您调用其他 AWS 服务所需的所有权限。

服务相关角色可让您更轻松地设置 Device Farm，因为您不必手动添加必要的权限。Device Farm 定义其服务相关角色的权限，除非另外定义，否则只有 Device Farm 可以代入该角色。定义的权限包括信任策略和权限策略，以及不能附加到任何其他 IAM 实体的权限策略。

只有在首先删除相关资源后，您才能删除服务关联角色。这将保护您的 Device Farm 资源，因为您不会无意中删除对资源的访问权限。

有关支持服务关联的角色的其他服务的信息，请参阅与 IAM 配合使用的亚马逊云科技服务，并查找 服务相关角色（Service-Linked Role）列设为 Yes（是）的服务。选择是，可转到查看该服务的服务相关角色文档的链接。

Device Farm 的服务相关角色权限

Device Farm 使用名为 AWSServiceRoleForDeviceFarm— 允许 Device Farm 代表您访问 AWS 资源的服务相关角色。

AWSServiceRoleForDeviceFarm 服务相关角色信任以下服务来代入该角色：

- `devicefarm.amazonaws.com`

角色权限策略允许 Device Farm 完成以下操作：

- 对于您的账户
  - 创建网络接口
  - 描述网络接口
  - 描述 VPCs
  - 描述子网
  - 描述安全组
  - 删除接口
  - 修改网络接口
- 对于网络接口
  - 创建标签
- 对于由 Device Farm 管理的 EC2 网络接口
  - 创建网络接口权限

完整的 IAM policy 内容如下：

JSON

```
{
"Version":"2012-10-17",
"Statement": [
 {
```

```
  "Effect": "Allow",
  "Action": [
   "ec2:DescribeNetworkInterfaces",
   "ec2:DescribeVpcs",
   "ec2:DescribeSubnets",
   "ec2:DescribeSecurityGroups"
  ],
  "Resource": "*"
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateNetworkInterface"
  ],
  "Resource": [
   "arn:aws:ec2:*:*:subnet/*",
   "arn:aws:ec2:*:*:security-group/*"
  ]
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateNetworkInterface"
  ],
  "Resource": [
   "arn:aws:ec2:*:*:network-interface/*"
  ],
  "Condition": {
   "StringEquals": {
    "aws:RequestTag/AWSDeviceFarmManaged": "true"
   }
  }
 },
 {
  "Effect": "Allow",
  "Action": [
   "ec2:CreateTags"
  ],
  "Resource": "arn:aws:ec2:*:*:network-interface/*",
  "Condition": {
   "StringEquals": {
    "ec2:CreateAction": "CreateNetworkInterface"
   }
  }
```

```
  },
  {
   "Effect": "Allow",
   "Action": [
    "ec2:CreateNetworkInterfacePermission",
    "ec2:DeleteNetworkInterface"
   ],
   "Resource": "arn:aws:ec2:*:*:network-interface/*",
   "Condition": {
    "StringEquals": {
     "aws:ResourceTag/AWSDeviceFarmManaged": "true"
    }
   }
  },
  {
   "Effect": "Allow",
   "Action": [
    "ec2:ModifyNetworkInterfaceAttribute"
   ],
   "Resource": [
    "arn:aws:ec2:*:*:security-group/*",
    "arn:aws:ec2:*:*:instance/*"
   ]
  },
  {
   "Effect": "Allow",
   "Action": [
    "ec2:ModifyNetworkInterfaceAttribute"
   ],
   "Resource": "arn:aws:ec2:*:*:network-interface/*",
   "Condition": {
    "StringEquals": {
     "aws:ResourceTag/AWSDeviceFarmManaged": "true"
    }
   }
  }
 ]
}
```

您必须配置权限，允许 IAM 实体（如用户、组或角色）创建、编辑或删除服务关联角色。有关更多信息，请参阅《IAM 用户指南》中的服务关联角色权限。

创建 Device Farm 的服务相关角色

当您为移动测试项目提供 VPC 配置时，无需手动创建服务相关角色。当你在 AWS 管理控制台、或 AWS API 中创建第一个 Device Farm 资源时，Device Farm 会为你创建服务相关角色。 AWS CLI

如果您删除该服务关联角色，然后需要再次创建，您可以使用相同流程在账户中重新创建此角色。在创建第一个 Device Farm 资源时，Device Farm 将再次为您创建服务相关角色。

您也可以使用 IAM 控制台为 Device Farm 使用案例创建服务相关角色。在 AWS CLI 或 AWS API 中，使用服务名称创建服务相关角色。`devicefarm.amazonaws.com`有关更多信息，请参阅《IAM 用户指南》中的创建服务相关角色。如果您删除了此服务相关角色，可以使用同样的过程再次创建角色。

编辑 Device Farm 的服务相关角色

Device Farm 不允许您编辑 AWSServiceRoleForDeviceFarm 服务相关角色。创建服务关联角色后，您将无法更改角色的名称，因为可能有多种实体引用该角色。但是可以使用 IAM 编辑角色描述。有关更多信息，请参阅《IAM 用户指南》中的编辑服务关联角色。

删除 Device Farm 的服务相关角色

如果不再需要使用某个需要服务关联角色的功能或服务，我们建议您删除该角色。这样就没有未被主动监控或维护的未使用实体。但是，必须先清除服务相关角色的资源，然后才能手动删除它。

> **Note**
>
> 如果在您试图删除资源时 Device Farm 服务正在使用该角色，则删除操作可能会失败。如果发生这种情况，请等待几分钟后重试。

使用 IAM 手动删除服务关联角色

使用 IAM 控制台 AWS CLI、或 AWS API 删除 AWSServiceRoleForDeviceFarm 服务相关角色。有关更多信息，请参见《IAM 用户指南》中的删除服务相关角色。

Device Farm 服务相关角色的受支持区域

Device Farm 支持在服务可用的所有区域中使用服务相关角色。有关更多信息，请参阅AWS 区域和端点。

Device Farm 不支持在服务可用的每个区域中使用服务相关角色。您可以在以下区域使用该 AWSServiceRoleForDeviceFarm 角色。

区域名称	区域标识	在 Device Farm 中受支持
美国东部（弗吉尼亚州北部）	us-east-1	否
美国东部（俄亥俄州）	us-east-2	否
美国西部（北加利福尼亚）	us-west-1	否
美国西部（俄勒冈州）	us-west-2	是
亚太地区（孟买）	ap-south-1	否
亚太地区（大阪）	ap-northeast-3	否
亚太地区（首尔）	ap-northeast-2	否
亚太地区（新加坡）	ap-southeast-1	否
亚太地区（悉尼）	ap-southeast-2	否
亚太地区（东京)	ap-northeast-1	否
加拿大（中部）	ca-central-1	否
欧洲地区（法兰克福）	eu-central-1	否
欧洲地区（爱尔兰）	eu-west-1	否
欧洲地区（伦敦）	eu-west-2	否
欧洲地区（巴黎）	eu-west-3	否
南美洲（圣保罗）	sa-east-1	否
AWS GovCloud (US)	us-gov-west-1	否

先决条件

以下列表描述了创建 VPC-ENI 配置时需要查看的一些要求和建议：

- 必须将私人设备分配给您的 AWS 账户。

- 您必须拥有有权创建服务相关角色的 AWS 账户用户或角色。使用带有 Device Farm 移动测试功能的 Amazon VPC 终端节点时，Device Farm 会创建一个 AWS Identity and Access Management (IAM) 服务相关角色。
- Device Farm VPCs 只能连接到该us-west-2区域内。如果您在 us-west-2 区域没有 VPC，则需要创建一个。然后，要访问另一个区域的 VPC 中的资源，您必须在 us-west-2 区域的 VPC 与其他区域的 VPC 之间建立对等连接。有关对等互连的信息 VPCs，请参阅 Amazon VPC 对等互连指南。

  在配置连接时，您应验证是否有权访问自己指定的 VPC。您必须为 Device Farm 配置特定的 Amazon Elastic Compute Cloud (Amazon EC2) 权限。
- 您使用的 VPC 中需要 DNS 解析。
- 创建 VPC 后，您将需要有关 us-west-2 区域 VPC 的以下信息：
  - - VPC ID
  - 子网 IDs（仅限私有子网）
  - 安全组 IDs
- 您必须根据每个项目配置 Amazon VPC 连接。目前，每个项目只能配置一个 VPC 配置。当您配置 VPC 时，Amazon VPC 会在您的 VPC 中创建一个接口并将其分配给指定的子网和安全组。所有与该项目关联的未来会话都将使用已配置的 VPC 连接。
- 您不能将 VPC-ENI 配置与传统 VPCE 功能一起使用。
- 我们强烈建议不要使用 VPC-ENI 配置更新现有项目，因为现有项目可能有在运行级别上持久存在的 VPCE 设置。相反，如果您已经在使用现有的 VPCE 功能，请将 VPC-ENI 用于所有新项目。

连接到 Amazon VPC

您可以配置和更新您的项目以使用 Amazon VPC 终端节点。VPC-ENI 配置根据每个项目进行配置。一个项目在任何给定时间都只能有一个 VPC-ENI 端点。要配置项目的 VPC 访问权限，您必须了解以下详细信息：

- us-west-2 中的 VPC ID（如果您的应用程序托管在那里），或 us-west-2 VPC ID（用于连接到其他区域中的其他 VPC）。
- 要应用到连接的适用安全组。
- 将与连接关联的子网。会话启动时，将使用最大的可用子网。我们建议将多个子网与不同的可用区域相关联，以改善您的 VPC 连接的可用性。

- 使用 VPC-ENI 时，Device Farm 测试主机和设备使用的 DNS 解析器将是客户子网中的 DHCP 服务提供的服务器。在默认配置中，这将是 VPC 的默认解析器。想要指定自定义 DNS 解析器的客户可以在其 VPC 中配置 DHCP 选项集。

创建 VPC-ENI 配置后，您可以按照以下步骤使用控制台或 CLI 更新其详细信息。

Console

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在 Device Farm 导航面板上，选择移动设备测试，然后选择项目。
3. 在移动测试项目下，从列表中选择您的项目名称。
4. 选择 Project settings (项目设置)。
5. 在虚拟私有云（VPC）设置部分，您可以更改 VPC、Subnets（仅限私有子网）和 Security Groups。
6. 选择保存。

CLI

使用以下 AWS CLI 命令更新 Amazon VPC：

```
$  aws devicefarm update-project \
--arn arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

您还可以在创建项目时配置 Amazon VPC：

```
$  aws devicefarm create-project \
--name VPCDemo \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

限制

以下限制适用于 VPC-ENI 功能：

- 在 Device Farm 项目的 VPC 配置中，您最多可以提供五个安全组。
- 在 Device Farm 项目的 VPC 配置中，您最多可以提供八个子网。
- 将 Device Farm 项目配置为与您的 VPC 配合使用时，您可以提供的最小子网必须至少有五个可用 IPv4 地址。
- 目前不支持公有 IP 地址。相反，我们建议您在 Device Farm 项目中使用私有子网。如果您在测试期间需要公共互联网访问权限，请使用网络地址转换 (NAT) 网关。使用公有子网配置 Device Farm 项目不会为您的测试提供互联网访问权限或公有 IP 地址。
- VPC-ENI 集成仅支持您 VPC 中的私有子网。
- 仅支持来自服务托管 ENI 的传出流量。这意味着 ENI 无法收到来自 VPC 的未经请求的入站请求。

将 Amazon VPC 端点服务与 Device Farm 结合使用 – 传统做法（不建议）

> **Warning**
>
> 我们强烈建议使用本页面描述的 VPC-ENI 连接进行私有端点连接，因为 VPCE 现在被视为一项传统功能。与 VPCE 连接方法相比，VPC-ENI 提供了更大的灵活性、更简单的配置、更具成本效益，并且所需的维护开销大幅降低。

> **Note**
>
> 只有配置了私有设备的客户才支持将 Amazon VPC 终端节点服务与 Device Farm 结合使用。要启用您的 AWS 账户以将此功能与私有设备结合使用，请联系我们。

Amazon Virtual Private Cloud（亚马逊 VPC）是一项 AWS 服务，可用于在您定义的虚拟网络中启动 AWS 资源。借助 VPC，您可以控制您的网络设置，如 IP 地址范围、子网、路由表和网络网关。

如果您使用 Amazon VPC 在美国西部（俄勒冈）(`us-west-2`) AWS 地区托管私有应用程序，则可以在您的 VPC 和 Device Farm 之间建立私有连接。借助此连接，您可以使用 Device Farm 测试私有应

用程序，而无需通过公共 Internet 公开它们。要使您的 AWS 账户能够在私人设备上使用此功能，请联系我们。

要将您的 VPC 中的资源连接到 Device Farm，您可以使用 Amazon VPC 控制台创建 VPC 终端节点服务。利用此终端节点服务，您可以通过 VPC 终端节点将您的 VPC 中的资源提供给 Device Farm。该终端节点服务提供了到 Device Farm 的可靠、可扩展的连接，无需 Internet 网关、网络地址转换 (NAT) 实例或 VPN 连接。有关更多信息，请参阅AWS PrivateLink 指南中的 VPC 终端节点服务 (AWS PrivateLink)。

> **Important**
>
> Device Farm VPC 终端节点功能可帮助您使用连接将 VPC 中的私有内部服务安全地 AWS PrivateLink 连接到 Device Farm 公有 VPC。虽然连接是安全且私有的，但这种安全性取决于您保护对 AWS 凭证的保护。如果您的 AWS 凭据遭到泄露，攻击者可以访问您的服务数据或将其暴露给外界。

在 Amazon VPC 中创建 VPC 终端节点服务后，您可以使用 Device Farm 控制台在 Device Farm 中创建 VPC 终端节点配置。本主题介绍如何在 Device Farm 中创建 Amazon VPC 连接和 VPC 终端节点配置。

开始前的准备工作

以下信息适用于美国西部（俄勒冈州）(`us-west-2`) 区域的 Amazon VPC 用户，其子网位于以下每个可用区：us-west-2a、us-west-2b 和 us-west-2c。

Device Farm 对于其可结合使用的 VPC 终端节点服务具有其他要求。当您创建和配置 VPC 终端节点服务以使用 Device Farm 时，请确保选择满足以下要求的选项：

- 服务的可用区必须包括 us-west-2a、us-west-2b 和 us-west-2c。VPC 终端节点服务的可用区由与终端节点服务关联的网络负载均衡器确定。如果您的 VPC 终端节点服务不显示所有这三个可用区，您必须重新创建网络负载均衡器来启用这三个区域，然后将网络负载均衡器与您的终端节点服务重新关联。
- 终端节点服务的白名单委托人必须包含 Device Farm VPC 终端节点的 Amazon 资源名称（ARN）。在创建终端节点服务后，将 Device Farm VPC 终端节点服务 ARN 添加到您的白名单，以向 Device Farm 提供访问您的 VPC 终端节点服务的权限。要获取 Device Farm VPC 终端节点服务 ARN，请联系我们。

此外，如果您在创建 VPC 终端节点服务时将需要接受设置保持开启状态，则必须手动接受 Device Farm 向终端节点服务发送的每个连接请求。要更改此设备，请在 Amazon VPC 控制台上选择终端节点服务，选择操作，然后选择修改终端节点接受设置。有关更多信息，请参阅《AWS PrivateLink 指南》中的更改负载均衡器和接受设置。

下一部分将说明如何创建满足这些要求的 Amazon VPC 终端节点服务。

步骤 1：创建网络负载均衡器

在您的 VPC 和 Device Farm 之间建立私有连接的第一步是创建网络负载均衡器，以便将请求路由到目标组。

New console

要使用新控制台创建网络负载均衡器

1. 打开亚马逊弹性计算云 (Amazon EC2) 控制台，网址为。https://console.aws.amazon.com/ec2/
2. 在导航窗格中的负载均衡下，选择负载均衡器。
3. 选择创建负载均衡器。
4. 在网络负载均衡器下，选择创建。
5. 在创建网络负载均衡器页面的基本配置下，执行以下操作：
   a. 输入负载均衡器名称。
   b. 对于方案，选择内部。
6. 在 网络映射下，请执行以下操作：
   a. 为您的目标组选择 VPC。
   b. 选择以下映射：
      - us-west-2a
      - us-west-2b
      - us-west-2c
7. 在侦听器和路由下，使用协议和端口选项来选择目标组。

> **Note**
>
> 默认情况下，跨可用性区域负载均衡处于禁用状态。

由于负载均衡器使用可用区 us-west-2a、us-west-2b 和 us-west-2c，它要求在每个可用区中注册目标，或者，如果您在少于所有三个区域中注册目标，则要求您启用跨区域负载均衡。否则，负载均衡器可能无法按预期运行。

8. 选择创建负载均衡器。

Old console

要使用旧控制台创建网络负载均衡器

1. 打开亚马逊弹性计算云 (Amazon EC2) 控制台，网址为。https://console.aws.amazon.com/ec2/
2. 在导航窗格中的负载均衡下，选择负载均衡器。
3. 选择创建负载均衡器。
4. 在网络负载均衡器下，选择创建。
5. 在配置负载均衡器页面的基本配置下，执行以下操作：
   a. 输入负载均衡器名称。
   b. 对于方案，选择内部。
6. 在侦听器下，选择目标组正在使用的协议和端口。
7. 在可用区下，请执行以下操作：
   a. 为您的目标组选择 VPC。
   b. 选择以下可用区：
      - us-west-2a
      - us-west-2b
      - us-west-2c
   c. 选择下一步：配置安全设置。
8. （可选）配置您的安全设置，然后选择下一步：配置路由。
9. 在配置路由页面上，执行以下操作：
   a. 对于目标组，选择现有目标组。
   b. 对于名称，选择您的目标组。
   c. 选择下一步：注册目标。

10. 在注册目标页面上，查看您的目标，然后选择下一步：查看。

> **Note**
>
> 默认情况下，跨可用性区域负载均衡处于禁用状态。
> 由于负载均衡器使用可用区 us-west-2a、us-west-2b 和 us-west-2c，它要求在每个可用区中注册目标，或者，如果您在少于所有三个区域中注册目标，则要求您启用跨区域负载均衡。否则，负载均衡器可能无法按预期运行。

11. 检查您的负载均衡器配置，然后选择创建。

步骤 2：创建 Amazon VPC 终端节点服务

创建网络负载均衡器后，使用 Amazon VPC 控制台在您的 VPC 中创建终端节点服务。

1. 打开位于 https://console.aws.amazon.com/vpc/ 的 Amazon VPC 控制台。
2. 在按区域列出的资源 下，选择 终端节点服务。
3. 选择创建终端节点服务。
4. 请执行以下操作之一：
   - 如果您已有一个希望终端节点服务使用的网络负载均衡器，请在可用的负载均衡器下选择它，然后继续执行步骤 5。
   - 如果您尚未创建网络负载均衡器，请选择创建新的负载均衡器。打开 Amazon EC2 控制台。按照创建网络负载均衡器中的步骤进行操作（从步骤 3 开始），然后在 Amazon VPC 控制台中继续执行这些步骤。
5. 对于包括的可用区，验证 us-west-2a、us-west-2b 和 us-west-2c 是否显示在列表中。
6. 如果您不想手动接受或拒绝发送到终端节点服务的每个连接请求，请在其他设置下，清除需要接受。如果清除此复选框，则终端节点服务将自动接受其收到的每个连接请求。
7. 选择创建。
8. 在新的终端节点服务中，选择允许主体。
9. 联系我们以获取要添加到终端节点服务允许列表的 Device Farm VPC 终端节点的 ARN，然后将该服务 ARN 添加到服务的允许列表中。
10. 在终端节点服务的详细信息选项卡上，记下服务的名称（服务名称）。您将在下一步中创建 VPC 终端节点配置时需要此名称。

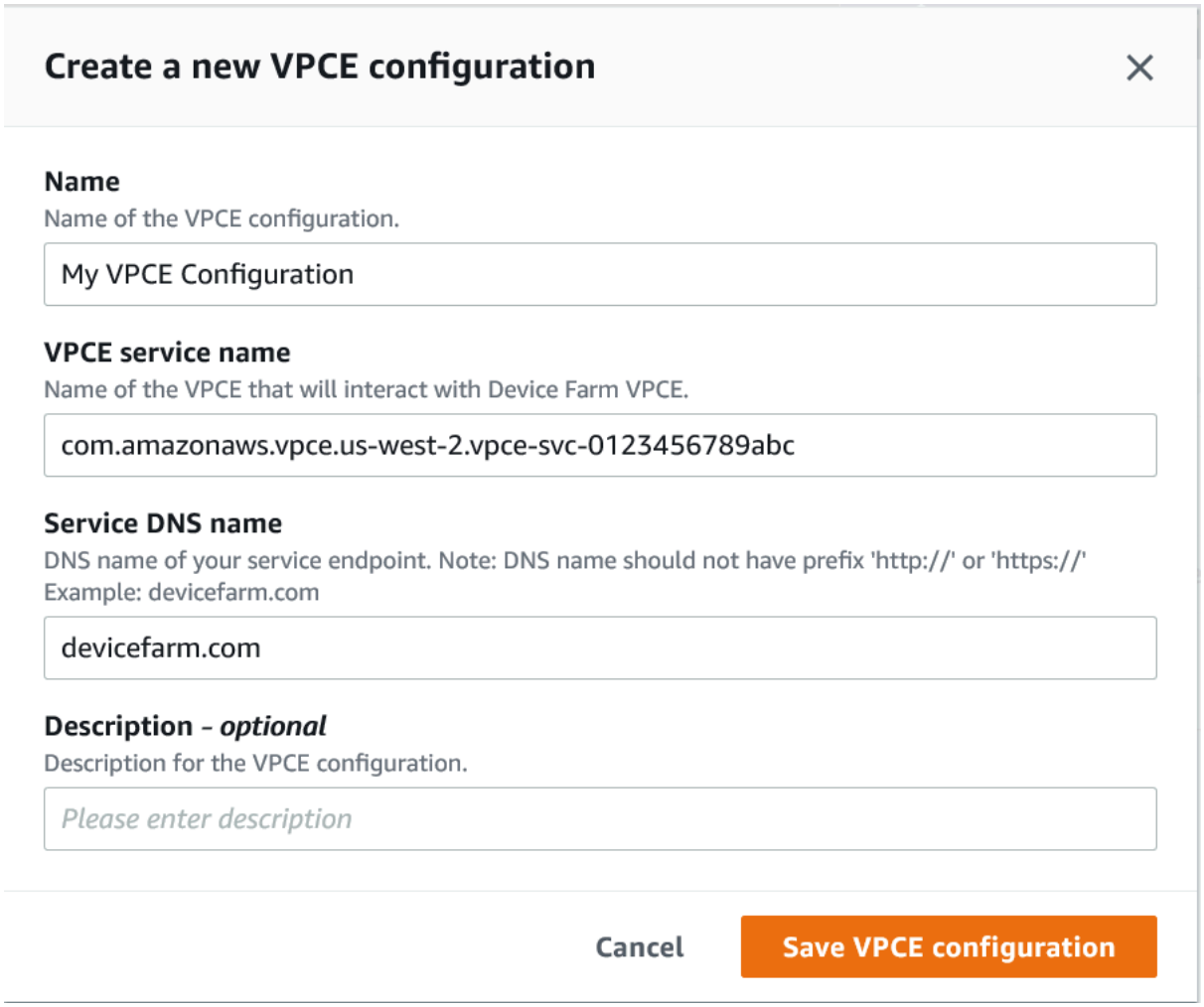
您的 VPC 终端节点服务现已准备好用于 Device Farm。

步骤 3：在 Device Farm 中创建 VPC 终端节点配置

在 Amazon VPC 中创建终端节点服务后，您可以在 Device Farm 中创建 Amazon VPC 终端节点配置。

1. 登录 DeviceFarm 控制台，网址为 https://console.aws.amazon.com/devicef arm。
2. 在导航窗格中，选择移动设备测试，然后选择私有设备。
3. 选择 VPCE 配置。
4. 选择创建 VPCE 配置。
5. 在创建新的 VPCE 配置下，输入 VPC 终端节点配置的名称。
6. 在 VPCE 服务名称中，输入您在 Amazon VPC 控制台中记下的 Amazon VPC 终端节点服务的名称（服务名称）。名称的形式类似于 `com.amazonaws.vpce.us-west-2.vpce-svc-id`。
7. 输入要测试的应用程序的服务 DNS 名称（例如，`devicefarm.com`）。请不要在服务 DNS 名称之前指定 `http` 或 `https`。

   域名不可通过公共 Internet 访问。此外，与您的 VPC 终端节点服务相对应的此新域名由 Amazon Route 53 生成且仅在您的 Device Farm 会话中供您使用。
8. 选择保存。



步骤 4：创建测试运行

保存 VPC 终端节点配置后，您可以使用该配置来创建测试运行或远程访问会话。有关更多信息，请参阅 在 Device Farm 中创建测试运行或 创建会话。

使用 AWS CloudTrail 记录 AWS Device Farm API 调用

AWS Device Farm 与 AWS CloudTrail 集成，后者是一项服务，提供用户、角色或 AWS 服务在 AWS Device Farm 中所采取操作的记录。CloudTrail 将 AWS Device Farm 的所有 API 调用作为事件捕获。捕获的调用包含来自 AWS Device Farm 控制台的调用和对 AWS Device Farm API 操作的代码调用。如果您创建跟踪记录，则可以使 CloudTrail 事件持续传送到 Amazon S3 存储桶（包括 AWS Device Farm 的事件）。如果您不配置跟踪，则仍可在 CloudTrail 控制台中的事件历史记录中查看最新事件。使用 CloudTrail 收集的信息，您可以确定向 AWS Device Farm 发出了什么请求、发出请求的 IP 地址、请求方、请求时间以及其他详细信息。

要了解有关 CloudTrail 的更多信息，请参阅《AWS CloudTrail 用户指南》https://docs.aws.amazon.com/awscloudtrail/latest/userguide/。

CloudTrail 中的 AWS Device Farm 信息

在您创建 AWS 账户时，将在该账户上启用 CloudTrail。当 AWS Device Farm 中发生活动时，该活动将记录在 CloudTrail 事件中，并与其它 AWS 服务事件一同保存在事件历史记录中。您可以在 AWS 账户中查看、搜索和下载最新事件。有关更多信息，请参阅使用 CloudTrail 事件历史记录查看事件。

要持续记录 AWS 账户中的事件（包括 AWS Device Farm 的事件），请创建跟踪。通过跟踪，CloudTrail 可将日志文件传送至 Amazon S3 存储桶。默认情况下，在控制台中创建跟踪记录时，此跟踪记录应用于所有 AWS 区域。此跟踪在 AWS 分区中记录所有区域中的事件，并将日志文件传送至您指定的 Amazon S3 存储桶。此外，您可以配置其他 AWS 服务，进一步分析在 CloudTrail 日志中收集的事件数据并采取行动。有关更多信息，请参阅下列内容：

- 创建跟踪概述
- CloudTrail 支持的服务和集成
- 为 CloudTrail 配置 Amazon SNS 通知
- 从多个区域接收 CloudTrail 日志文件和从多个账户接收 CloudTrail 日志文件

在您的 AWS 账户中启用 CloudTrail 日志记录后，将在日志文件中跟踪对 Device Farm 操作执行的 API 调用。Device Farm 记录会与其他 AWS 服务记录一起写入日志文件。CloudTrail 基于时间段和文件大小来确定何时创建并写入新文件。

所有 Device Farm 操作都会被记录，并正式记载到 AWS CLI 参考文档 和 自动化 Device Farm 中。例如，对在 Device Farm 中创建新的项目或运行进行的调用会在 CloudTrail 日志文件中生成条目。

每个事件或日志条目都包含有关生成请求的人员信息。身份信息可帮助您确定以下内容：

- 请求是使用根用户凭证还是 AWS Identity and Access Management（IAM）用户凭证发出的。
- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 AWS 服务发出。

有关更多信息，请参阅 CloudTrail userIdentity 元素。

了解 AWS Device Farm 日志文件条目

跟踪是一种配置，可用于将事件作为日志文件传送到您指定的 Amazon S3 存储桶。CloudTrail 日志文件包含一个或多个日记账条目。一个事件表示来自任何源的一个请求，包括有关所请求的操作、操作的日期和时间、请求参数等方面的信息。CloudTrail 日志文件不是公用 API 调用的有序堆栈跟踪，因此它们不会按任何特定顺序显示。

下面的示例显示了一个 CloudTrail 日志条目，该条目说明了 Device Farm `ListRuns` 操作：

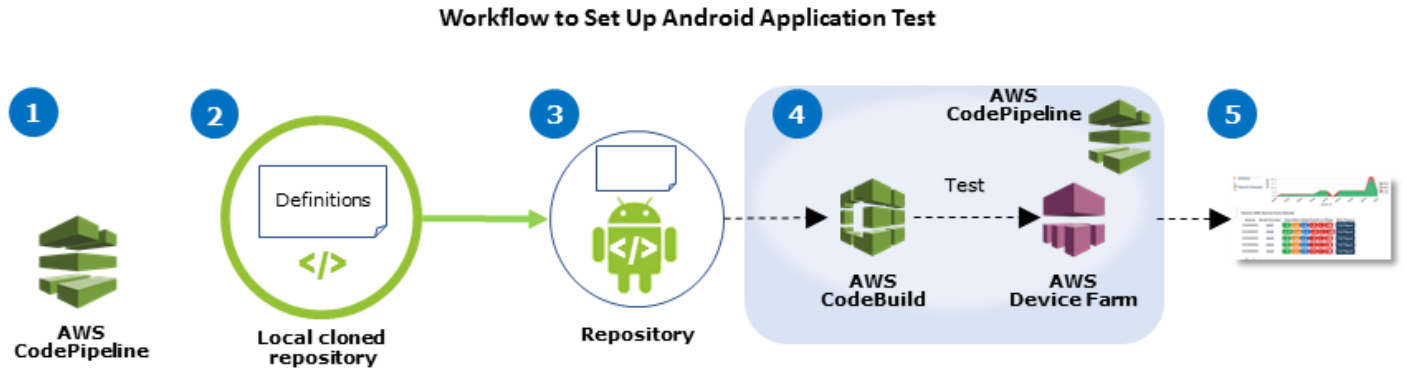
```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "Root",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:root",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2015-07-08T21:13:35Z"
          }
        }
      },
      "eventTime":"2015-07-09T00:51:22Z",
      "eventSource": "devicefarm.amazonaws.com",
      "eventName":"ListRuns",
      "awsRegion":"us-west-2",
      "sourceIPAddress":"203.0.113.11",
      "userAgent":"example-user-agent-string",
      "requestParameters": {
```

```
        "arn":"arn:aws:devicefarm:us-west-2:123456789012:project:a9129b8c-
df6b-4cdd-8009-40a25EXAMPLE"},
        "responseElements": {
          "runs": [
            {
              "created": "Jul 8, 2015 11:26:12 PM",
              "name": "example.apk",
              "completedJobs": 2,
              "arn": "arn:aws:devicefarm:us-west-2:123456789012:run:a9129b8c-
df6b-4cdd-8009-40a256aEXAMPLE/1452d105-e354-4e53-99d8-6c993EXAMPLE",
              "counters": {
                "stopped": 0,
                "warned": 0,
                "failed": 0,
                "passed": 4,
                "skipped": 0,
                "total": 4,
                "errored": 0
              },
              "type": "BUILTIN_FUZZ",
              "status": "RUNNING",
              "totalJobs": 3,
              "platform": "ANDROID_APP",
              "result": "PENDING"
            },
            ... additional entries ...
          ]
        }
      }
    }
  ]
}
```

在 CodePipeline 测试阶段集成 AWS Device Farm

您可以使用 AWS CodePipeline 将在 Device Farm 中配置的移动应用程序测试合并到 AWS 托管的自动化发布管道中。您可以将您的管道配置为按需、按计划或作为持续集成流程的一部分运行测试。

下图显示了在每次向存储库中提交推送时，在其中构建和测试 Android 应用程序的持续集成流程。要创建此工作流配置，请参阅教程：推送到 Android 应用程序时构建和测试 GitHub。



1。配置	2。添加定义	3。Push	4。构建和测试	5。报告
配置管道资源	将构建和测试定义添加到程序包	将程序包推送到存储库	构建输出项目的应用程序构建和测试自动启动	查看测试结果

要了解如何配置管道以便将已编译的应用程序（如 iOS `.ipa` 或 Android `.apk` 文件）作为其源进行持续测试，请参阅教程：每次将 .ipa 文件上传到 Amazon S3 存储桶时测试 iOS 应用程序。

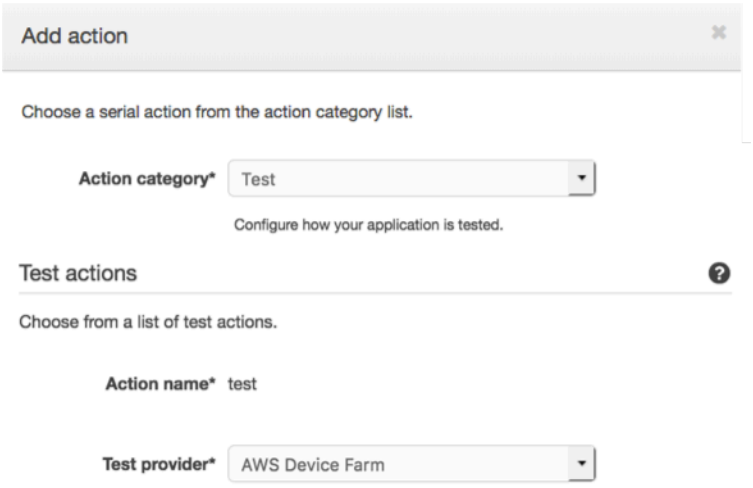
配置 CodePipeline 为使用您的 Device Farm 测试

在这些步骤中，我们假定您已配置了一个 Device Farm 项目并创建了一个管道。该管道应配置了测试阶段，用于接收输入项目，此输入项目包含您的测试定义和编译的应用程序包文件。测试阶段输入项目可以是源代码或在管道中配置的构建阶段的输出项目。

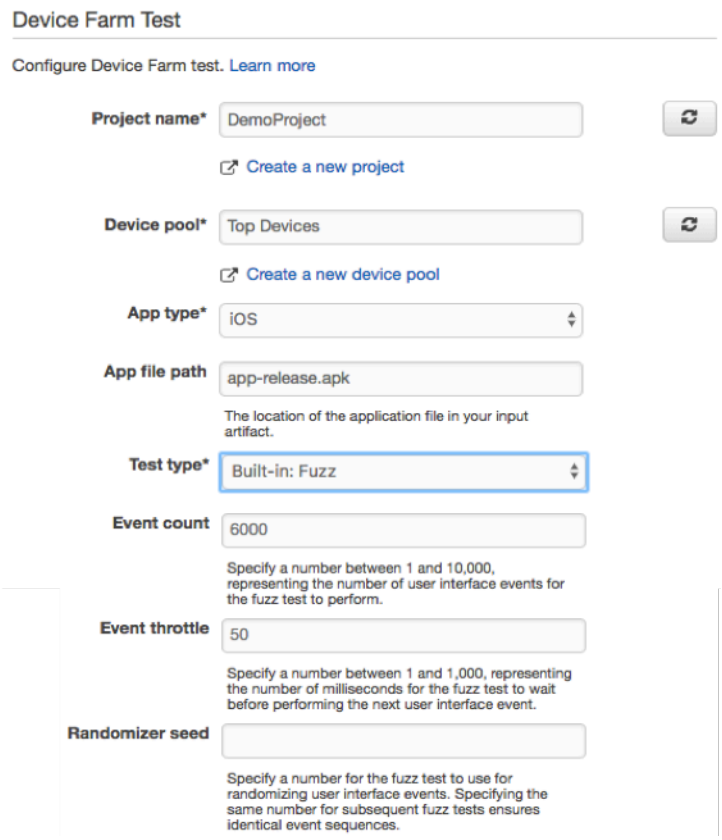
将 Device Farm 测试运行配置为 CodePipeline 测试操作

1. 登录 AWS 管理控制台 并打开 CodePipeline 控制台，网址为https://console.aws.amazon.com/codepipeline/。

2. 为您的应用程序版本选择管道。
3. 在测试阶段面板中，选择铅笔图标，然后选择 Action (操作)。
4. 在 Add action (添加操作)窗格中，对于 Action category (操作类别)，选择 Test (测试)。
5. 在操作名称中输入名称。
6. 在测试提供商中，选择 AWS Device Farm。

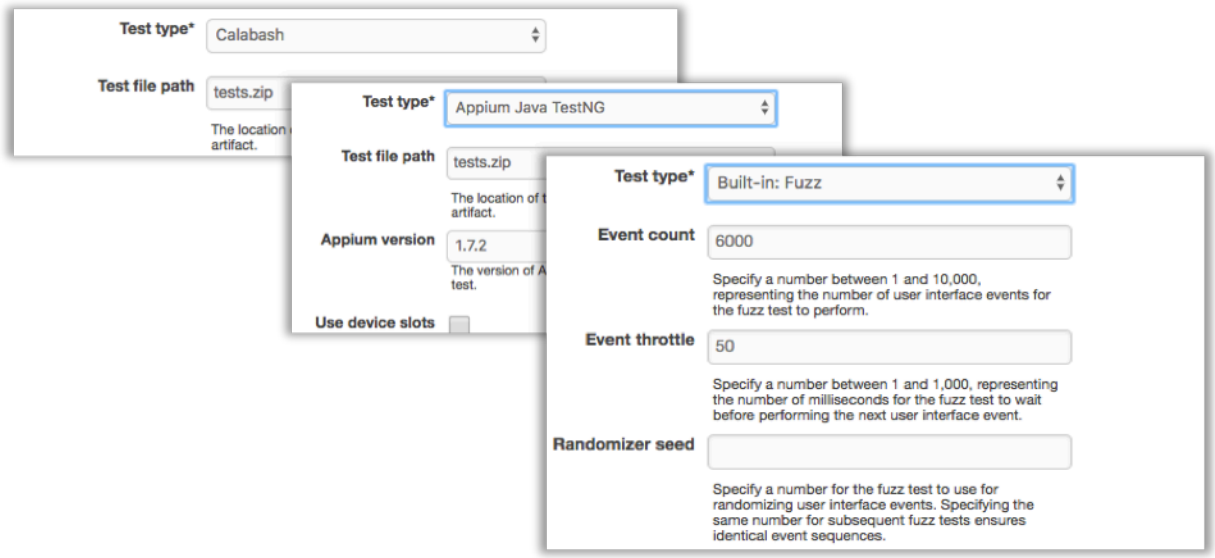


7. 在项目名称中，选择现有的 Device Farm 项目或选择创建新项目。
8. 在 Device pool (设备池) 中，选择现有设备池或选择 Create a new device pool (创建新的设备池)。如果创建设备池，则需要选择一组测试设备。
9. 在 App type (应用程序类型)中，为您的应用程序选择平台。

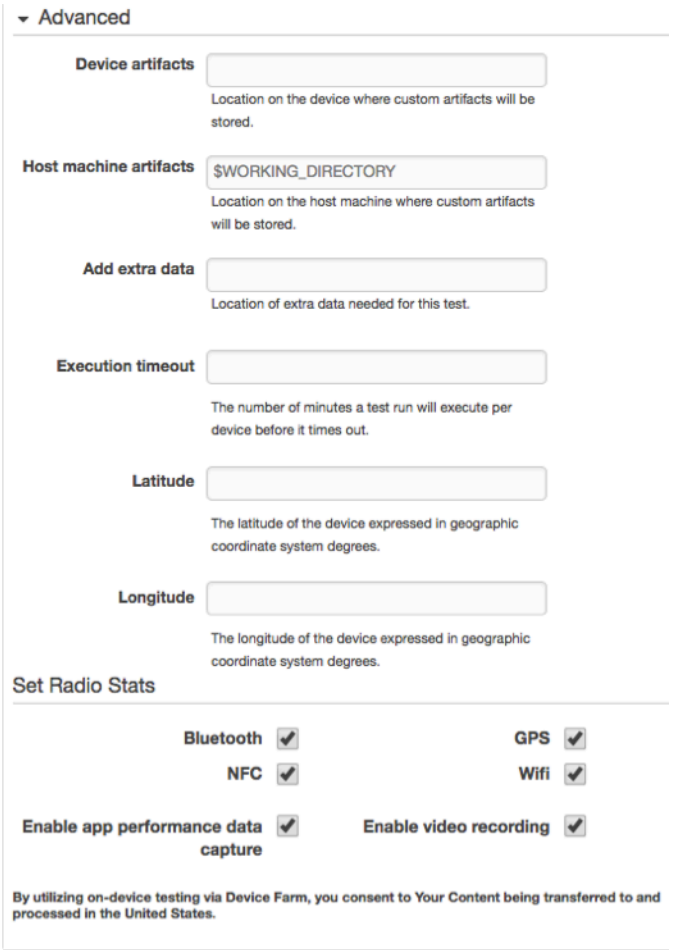


10. 在 App file path (应用程序文件路径) 中，输入已编译的应用程序包的路径。路径相对于测试的输入项目的根。
11. 在 Test type (测试类型) 中，执行下列操作之一：

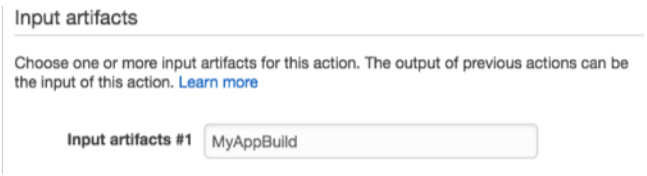
    - 如果使用其中一个内置 Device Farm 测试，请选择在 Device Farm 项目中配置的测试类型。
    - 如果不使用其中一个 Device Farm 内置测试，请在测试文件路径中输入测试定义文件的路径。路径相对于测试的输入项目的根。



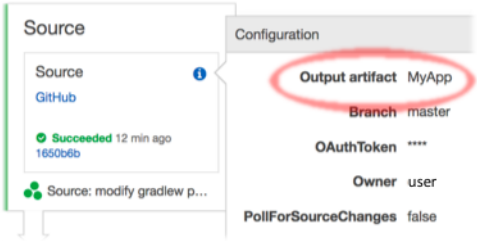
12. 在其余字段中，提供适合测试和应用程序类型的配置。
13. （可选）在 Advanced (高级) 中，为您的测试运行提供详细的配置。



14. 在 Input artifacts (输入项目) 中，选择与管道测试阶段之前的那个阶段的输出项目相匹配的输入项目。



在 CodePipeline 控制台中，将鼠标悬停在管道图中的信息图标上，可以找到每个阶段的输出工件的名称。如果您的管道直接从 S ourc e 阶段测试您的应用程序，请选择MyApp。如果您的管道包含构建阶段，请选择MyAppBuild。



15. 在面板底部，选择 Add Action (添加操作)。
16. 在 CodePipeline 窗格中，选择 “保存管道更改”，然后选择 “保存更改”。
17. 要提交所做的更改并开始管道构建，请选择发布更改，然后选择发布。

AWS Device Farm 的 AWS CLI 参考文档

要使用 AWS Command Line Interface (AWS CLI) 运行 Device Farm 命令，请参阅 AWS Device Farm 的 AWS CLI 参考文档。

有关 AWS CLI 的一般信息，请参阅 AWS Command Line Interface 用户指南和 AWS CLI 命令参考。

AWS Device Farm 的 Windows PowerShell 参考

要使用 Windows PowerShell 运行 Device Farm 命令，请参阅 AWS Tools for Windows PowerShell Cmdlet 参考中的 Device Farm Cmdlet 参考。有关更多信息，请参阅《AWS Tools for PowerShell 用户指南》中的设置适用于 Windows PowerShell 的 AWS 工具。

自动化 AWS Device Farm

对 Device Farm 的编程访问是一种强大的方式，可以自动执行您需要完成的常见任务，例如安排运行或下载运行、套件或测试的构件。S AWS DK 和 AWS CLI 提供实现此目的的方法。

该 AWS 软件开发工具包提供对所有 AWS 服务的访问权限，包括 Device Farm、Amazon S3 等。有关更多信息，请参阅

- AWS 工具和 SDKs
- AWS Device Farm API 参考

示例：使用 AWS CLI 或 SDK 将应用程序或测试上传到 Device Farm

以下示例展示了如何使用 AWS CLI 或使用各种语言的 AWS SDK 在 Device Farm 上创建上传。上传是在 Device Farm 上安排测试运行的核心组成部分，包括以下内容：

- 你的应用程序
- 你的测试
- 您的测试规范文件

上传是使用 `CreateUpload`API 创建的。此 API 返回一个 S3 预签名 URL，您可以使用 HTTP PUT 请求将上传内容推送到该网址。网址将在 24 小时后过期。

AWS CLI

注意：此示例使用命令行工具将应用程序推送*curl*到 D evice Farm。

首先，如果你还没有创建一个项目。

```
$ aws devicefarm create-project --name MyProjectName
```

这将显示如下输出：

```
{
```

```
    "project": {
        "name": "MyProjectName",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "created": 1535675814.414
    }
}
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
$ export APP_PATH="/local/path/to/my_sample_app.apk"
$ export APP_TYPE="ANDROID_APP"
```

首先，我们在 Device Farm 中创建上传文件：

```
$ aws devicefarm create-upload \
  --project-arn "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE" \
  --name "$(basename "$APP_PATH")" \
  --type "$APP_TYPE"
```

这将显示如下输出：

```
{
    "upload": {
        "arn": "arn:aws:devicefarm:us-
west-2:385076942068:upload:490a6350-0ba3-43e5-83f5-d2896b069a34/a120e848-c57b-4e8d-
a720-d750a0c4d936",
        "name": "my_sample_app.apk",
        "created": 1760747318.266,
        "type": "ANDROID_APP",
        "status": "INITIALIZED",
        "url": "https://prod-us-west-2-uploads.s3.dualstack.us-west-2.amazonaws.com/
arn%3Aaws%3Adevicefarm%3Aus-west-2...",
        "category": "PRIVATE"
    }
}
```

然后，使用 curl 进行 PUT 调用，将应用程序推送到 Device Farm 的 S3 存储桶：

```
$ curl -T "$APP_PATH" "https://prod-us-west-2-uploads.s3.dualstack.us-
west-2.amazonaws.com/arn%3Aaws%3Adevicefarm%3Aus-west-2..."
```

最后，等待应用程序处于“成功”状态：

```
$ aws devicefarm get-upload --arn "arn:aws:devicefarm:us-
west-2:385076942068:upload:490a6350-0ba3-43e5-83f5-d2896b069a34/a120e848-c57b-4e8d-
a720-d750a0c4d936"
```

这将显示如下输出：

```
{
    "upload": {
        "arn": "arn:aws:devicefarm:us-
west-2:385076942068:upload:490a6350-0ba3-43e5-83f5-d2896b069a34/a120e848-c57b-4e8d-
a720-d750a0c4d936",
        "name": "my_sample_app.apk",
        "created": 1760747318.266,
        "type": "ANDROID_APP",
        "status": "SUCCEEDED",
        "url": "https://prod-us-west-2-uploads.s3.dualstack.us-west-2.amazonaws.com/
arn%3Aaws%3Adevicefarm%3Aus-west-2...",
        "metadata": "{\"activity_name\":
\"com.amazonaws.devicefarm.android.referenceapp.Activities.MainActivity\",
\"package_name\":\"com.amazonaws.devicefarm.android.referenceapp\",...}",
        "category": "PRIVATE"
    }
}
```

Python

注意：此示例使用第三方*requests*软件包将应用程序推送到 Device Farm，并使用适用于 Python 的 AWS SDK *boto3*。

首先，如果你还没有创建一个项目。

```
import boto3

client = boto3.client("devicefarm", region_name="us-west-2")
resp = client.create_project(name="MyProjectName")

print(resp)
```

```
# Response will be something like:
# {
#     "project": {
#         "name": "MyProjectName",
#         "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
#         "created": 1535675814.414
#     }
# }
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
import os
import time
import datetime
import requests
from pathlib import Path
import boto3


def upload_device_farm_file():
    project_arn = "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    app_path = Path("/local/path/to/my_sample_app.apk")
    file_type = "ANDROID_APP"

    if not app_path.is_file():
        raise RuntimeError(f"{app_path} is not a valid app file path")

    client = boto3.client("devicefarm", region_name="us-west-2")

    # 1) Create the upload in Device Farm
    create = client.create_upload(
        projectArn=project_arn,
        name=app_path.name,
        type=file_type,
        contentType="application/octet-stream",
    )
    upload = create["upload"]
    upload_arn = upload["arn"]
    upload_url = upload["url"]
```

```
    # This will show output such as the following:
    # { "upload": { "arn": "...", "name": "my_sample_app.apk", "type":
 "ANDROID_APP", "status": "INITIALIZED", "url": "https://..." } }

    # 2) Do an HTTP PUT command to push the file to the pre-signed S3 URL
    with app_path.open("rb") as fh:
        print(f"Uploading {app_path.name} to Device Farm...")
        put_resp = requests.put(upload_url, data=fh, headers={"Content-Type":
 "application/octet-stream"})
        put_resp.raise_for_status()

    # 3) Wait for the app to be in "SUCCEEDED" status (or fail/timeout)
    timeout_seconds = 30
    start = time.time()
    while True:
        get_resp = client.get_upload(arn=upload_arn)
        status = get_resp["upload"]["status"]
        msg = get_resp["upload"].get("message") or
 get_resp["upload"].get("metadata") or ""
        elapsed = datetime.timedelta(seconds=int(time.time() - start))
        print(f"[{elapsed}] status={status}{' - ' + msg if msg else ''}")

        if status == "SUCCEEDED":
            print(f"Upload complete: {upload_arn}")
            return upload_arn
        if status == "FAILED":
            raise RuntimeError(f"Device Farm failed to process upload: {msg}")

        if (time.time() - start) > timeout_seconds:
            raise RuntimeError(f"Timed out after {timeout_seconds}s waiting for
 upload to process (last status={status}).")

        time.sleep(1)

upload_device_farm_file()
```

Java

注意：此示例使用 AWS 适用于 Java v2 的 SDK 将应用程序推送*HttpClient*到 Device Farm，并且与 JDK 版本 11 及更高版本兼容。

首先，如果你还没有创建一个项目。

```
import software.amazon.awssdk.regions.Region;
```

```
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import software.amazon.awssdk.services.devicefarm.model.CreateProjectRequest;
import software.amazon.awssdk.services.devicefarm.model.CreateProjectResponse;

try (DeviceFarmClient client = DeviceFarmClient.builder()
        .region(Region.US_WEST_2)
        .build()) {
    CreateProjectResponse resp = client.createProject(
        CreateProjectRequest.builder().name("MyProjectName").build());
    System.out.println(resp.project());
    // Response will be something like:
    // Project{name=MyProjectName, arn=arn:aws:devicefarm:us-
west-2:123456789101:project:5e01a8c7-..., created=...}
}
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.time.Duration;
import java.time.Instant;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import software.amazon.awssdk.services.devicefarm.model.CreateUploadRequest;
import software.amazon.awssdk.services.devicefarm.model.CreateUploadResponse;
import software.amazon.awssdk.services.devicefarm.model.GetUploadRequest;
import software.amazon.awssdk.services.devicefarm.model.GetUploadResponse;
import software.amazon.awssdk.services.devicefarm.model.Upload;
import software.amazon.awssdk.services.devicefarm.model.UploadType;

public class DeviceFarmUploader {

    public static String upload(String projectArn, Path appPath) throws Exception {
        if (projectArn == null || projectArn.isEmpty()) {
```

```
            throw new IllegalArgumentException("Missing projectArn");
        }
        if (!Files.isRegularFile(appPath)) {
            throw new IllegalArgumentException("Invalid app path: " + appPath);
        }

        String fileName = appPath.getFileName().toString().trim();
        UploadType type = UploadType.ANDROID_APP;

        // Build a reusable HttpClient
        HttpClient http = HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_1_1)
                .connectTimeout(Duration.ofSeconds(10))
                .build();

        try (DeviceFarmClient client = DeviceFarmClient.builder()
                .region(Region.US_WEST_2)
                .build()) {

            // 1) Create the upload in Device Farm
            CreateUploadResponse create =
client.createUpload(CreateUploadRequest.builder()
                    .projectArn(projectArn)
                    .name(fileName)
                    .type(type)
                    .contentType("application/octet-stream")
                    .build());

            Upload upload = create.upload();
            String uploadArn = upload.arn();
            String url = upload.url();
            // This will show output such as the following:
            // { "upload": { "arn": "...", "name": "my_sample_app.apk", "type":
"ANDROID_APP", "status": "INITIALIZED", "url": "https://..." } }

            // 2) PUT file to pre-signed URL using HttpClient
            HttpRequest put = HttpRequest.newBuilder(URI.create(url))
                    .timeout(Duration.ofMinutes(15))
                    .header("Content-Type", "application/octet-stream")
                    .PUT(HttpRequest.BodyPublishers.ofFile(appPath))
                    .build();

            HttpResponse<Void> resp = http.send(put,
HttpResponse.BodyHandlers.discarding());
```

```
            int code = resp.statusCode();
            if (code / 100 != 2) {
                throw new IOException("Failed PUT to S3 pre-signed URL, HTTP " +
 code);
            }

            // 3) Wait for the app to be in "SUCCEEDED" status (or fail/timeout)
            Instant deadline = Instant.now().plusSeconds(30); // 30-second timeout
            while (true) {
                GetUploadResponse got = client.getUpload(GetUploadRequest.builder()
                        .arn(uploadArn)
                        .build());

                String status = got.upload().statusAsString();
                String msg = got.upload().metadata();
                System.out.println("status=" + status + (msg != null ? " - " + msg :
 ""));

                if ("SUCCEEDED".equals(status)) return uploadArn;
                if ("FAILED".equals(status)) throw new RuntimeException("Upload
 failed: " + msg);
                if (Instant.now().isAfter(deadline)) {
                    throw new RuntimeException("Timeout waiting for processing, last
 status=" + status);
                }
                Thread.sleep(2000);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        String projectArn = "arn:aws:devicefarm:us-
west-2:123456789101:project:5e01a8c7-c861-4c0a-b1d5-12345EXAMPLE";
        Path appPath = Paths.get("/local/path/to/my_sample_app.apk");
        String result = upload(projectArn, appPath);
        System.out.println("Upload ARN: " + result);
    }
}
```

JavaScript

注意：此示例使用 AWS 适用于 JavaScript (v3) 和 Node 18+ 的 SDK 将应用程序推送*fetch*到 Device Farm。

首先，如果你还没有创建一个项目。

```
import { DeviceFarmClient, CreateProjectCommand } from "@aws-sdk/client-device-
farm";

const df = new DeviceFarmClient({ region: "us-west-2" });
const resp = await df.send(new CreateProjectCommand({ name: "MyProjectName" }));
console.log(resp);
// Response will be something like:
// { project: { name: 'MyProjectName', arn: 'arn:aws:devicefarm:us-
west-2:123456789101:project:5e01a8c7-...', created: 1535675814.414 } }
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
import { DeviceFarmClient, CreateUploadCommand, GetUploadCommand } from "@aws-sdk/
client-device-farm";
import { createReadStream } from "fs";
import { basename } from "path";

const projectArn = "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE";
const appPath = "/local/path/to/my_sample_app.apk";
const name = basename(appPath).trim();
const type = "ANDROID_APP";

const client = new DeviceFarmClient({ region: "us-west-2" });

// 1) Create the upload in Device Farm
const create = await client.send(new CreateUploadCommand({
  projectArn,
  name,
  type,
  contentType: "application/octet-stream",
}));

const uploadArn = create.upload.arn;
const url = create.upload.url;
// This will show output such as the following:
// { upload: { arn: '...', name: 'my_sample_app.apk', type: 'ANDROID_APP', status:
 'INITIALIZED', url: 'https://...' } }
```

```
// 2) PUT to pre-signed URL
const putResp = await fetch(url, {
  method: "PUT",
  headers: { "Content-Type": "application/octet-stream" },
  body: createReadStream(appPath),
});
if (!putResp.ok) {
  throw new Error(`Failed PUT to pre-signed URL: ${putResp.status} ${await
 putResp.text().catch(()=>"")}`);
}

// 3) Wait for the app to be in "SUCCEEDED" status (or fail/timeout)
const deadline = Date.now() + (30 * 1000); // 30-second timeout
while (true) {
  const response = await client.send(new GetUploadCommand({ arn: uploadArn }));
  const { status, message, metadata } = response.upload;
  console.log(`status=${status}${message ? " - " + message : metadata ? " - " +
 metadata : ""}`);
  if (status === "SUCCEEDED") {
    console.log("Upload complete:", uploadArn);
    break;
  }
  if (status === "FAILED") {
    throw new Error(`Upload failed: ${message || metadata || "unknown"}`);
  }
  if (Date.now() > deadline) throw new Error(`Timeout waiting for processing (last
 status=${status})`);
  await new Promise(r => setTimeout(r, 2000));
}
```

C#

注意：此示例使用适用于.NET 的 AWS SDK*HttpClient*，并将应用程序推送到 Device Farm。

首先，如果你还没有创建一个项目。

```
using System;
using Amazon;
using Amazon.DeviceFarm;
using Amazon.DeviceFarm.Model;

using var client = new AmazonDeviceFarmClient(RegionEndpoint.USWest2);
var resp = await client.CreateProjectAsync(new CreateProjectRequest { Name =
 "MyProjectName" });
```

```
Console.WriteLine(resp.Project);
// Response will be something like:
// { Name = MyProjectName, Arn = arn:aws:devicefarm:us-
west-2:123456789101:project:5e01a8c7-..., Created = ... }
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
using System;
using System.IO;
using System.Net.Http;
using System.Threading.Tasks;
using System.Net.Http.Headers;
using Amazon;
using Amazon.DeviceFarm;
using Amazon.DeviceFarm.Model;

class DeviceFarmUploader
{
    public static async Task<string> UploadAsync(string projectArn, string appPath)
    {
        if (string.IsNullOrWhiteSpace(projectArn)) throw new
 ArgumentException("Missing projectArn");
        if (!File.Exists(appPath)) throw new ArgumentException($"Invalid app path:
 {appPath}");
        var type = UploadType.ANDROID_APP;

        using var client = new AmazonDeviceFarmClient(RegionEndpoint.USWest2);
        // 1) Create the upload in Device Farm
        var create = await client.CreateUploadAsync(new CreateUploadRequest
        {
            ProjectArn = projectArn,
            Name = Path.GetFileName(appPath),
            Type = type,
            ContentType = "application/octet-stream"
        });

        var uploadArn = create.Upload.Arn;
        var url = create.Upload.Url;
        // This will show output such as the following:
        // { Upload: { Arn = ..., Name = my_sample_app.apk, Type = ANDROID_APP,
 Status = INITIALIZED, Url = https://... } }
```

```
        // 2) PUT file to pre-signed URL
        using (var http = new HttpClient())
        using (var fs = File.OpenRead(appPath))
        using (var content = new StreamContent(fs))
        {
            content.Headers.Add("Content-Type", "application/octet-stream");
            var resp = await http.PutAsync(url, content);
            if (!resp.IsSuccessStatusCode)
                throw new Exception($"Failed PUT to pre-signed URL:
 {(int)resp.StatusCode} {await resp.Content.ReadAsStringAsync()}");
        }

        // 3) Wait for the app to be in "SUCCEEDED" status (or fail/timeout)
        var deadline = DateTime.UtcNow.AddSeconds(30); // 30-second timeout
        while (true)
        {
            var got = await client.GetUploadAsync(new GetUploadRequest { Arn =
 uploadArn });
            var status = got.Upload.Status.Value;
            var msg = got.Upload.Message ?? got.Upload.Metadata;
            Console.WriteLine($"status={status}{(string.IsNullOrEmpty(msg) ? "" : "
 - " + msg)}");

            if (status == UploadStatus.SUCCEEDED.Value) return uploadArn;
            if (status == UploadStatus.FAILED.Value) throw new Exception($"Upload
 failed: {msg}");
            if (DateTime.UtcNow > deadline) throw new TimeoutException($"Timeout
 waiting for processing (last status={status})");
            await Task.Delay(2000);
        }
    }

    static async Task Main()
    {
        var projectArn = "arn:aws:devicefarm:us-
west-2:123456789101:project:5e01a8c7-c861-4c0a-b1d5-12345EXAMPLE";
        var appPath = "/local/path/to/my_sample_app.apk";
        var result = await UploadAsync(projectArn!, appPath!);
        Console.WriteLine("Upload ARN: " + result);
    }
}
```

Ruby

注意：此示例使用 AWS 适用于 Ruby 的 SDK*Net::HTTP*，并将应用程序推送到 Device Farm。

首先，如果你还没有创建一个项目。

```
require "aws-sdk-devicefarm"

client = Aws::DeviceFarm::Client.new(region: "us-west-2")
resp = client.create_project(name: "MyProjectName")
puts resp.project.inspect
# Response will be something like:
# #<struct Aws::DeviceFarm::Types::Project name="MyProjectName",
 arn="arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-...",
 created=1535675814.414>
```

然后，执行以下操作来创建您的上传内容并将其推送到 Device Farm。在此示例中，我们将使用本地 APK 文件创建 Android 应用程序上传。如需更多上传类型信息，包括有关 iOS 应用上传类型的详细信息，请参阅我们用于创建的 API 文档Upload。

```
require "aws-sdk-devicefarm"
require "net/http"
require "uri"

project_arn = "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-c861-4c0a-
b1d5-12345EXAMPLE"
app_path    = "/local/path/to/my_sample_app.apk"
raise "Invalid APP_PATH: #{app_path}" unless File.file?(app_path)
type = "ANDROID_APP"

client = Aws::DeviceFarm::Client.new(region: "us-west-2")

# 1) Create the upload in Device Farm
create = client.create_upload(
  project_arn: project_arn,
  name: File.basename(app_path),
  type: type,
  content_type: "application/octet-stream"
)

upload_arn = create.upload.arn
url = create.upload.url
# This will show output such as the following:
```

```
# #<Upload arn="...", name="my_sample_app.apk", type="ANDROID_APP",
 status="INITIALIZED", url="https://...">

# 2) PUT the file to the pre-signed URL
uri = URI.parse(url)
Net::HTTP.start(uri.host, uri.port, use_ssl: (uri.scheme == "https")) do |http|
  req = Net::HTTP::Put.new(uri)
  req["Content-Type"] = "application/octet-stream"
  req.body_stream = File.open(app_path, "rb")
  req.content_length = File.size(app_path)
  resp = http.request(req)
  raise "Failed PUT: #{resp.code} #{resp.body}" unless resp.code.to_i / 100 == 2
end

# 3) Wait for the app to be in "SUCCEEDED" status (or fail/timeout)
deadline = Time.now + 30 # 30-second timeout
loop do
  got = client.get_upload(arn: upload_arn)
  status = got.upload.status
  msg = got.upload.message || got.upload.metadata
  puts "status=#{status}#{msg ? " - #{msg}" : ""}"

  case status
  when "SUCCEEDED" then puts "Upload complete: #{upload_arn}"; break
  when "FAILED"    then raise "Upload failed: #{msg}"
  end
  raise "Timeout waiting for processing (last status=#{status})" if Time.now >
 deadline
  sleep 2
end
```

示例：使用 AWS SDK 启动 Device Farm 运行并收集工件

以下示例 beginning-to-end演示了如何使用软件开发工具包与 AWS Device Farm 配合使用。本示例执行以下操作：

- 将测试和应用程序包上传到 Device Farm
- 启动测试运行并等待测试完成（或失败）
- 下载测试套件生成的所有构件

本示例利用第三方 requests 程序包与 HTTP 交互。

```
import boto3
import os
import requests
import string
import random
import time
import datetime
import time
import json

# The following script runs a test through Device Farm
#
# Things you have to change:
config = {
    # This is our app under test.
    "appFilePath":"app-debug.apk",
    "projectArn": "arn:aws:devicefarm:us-
west-2:111122223333:project:1b99bcff-1111-2222-ab2f-8c3c733c55ed",
    # Since we care about the most popular devices, we'll use a curated pool.
    "testSpecArn":"arn:aws:devicefarm:us-west-2::upload:101e31e8-12ac-11e9-ab14-
d663bd873e83",
    "poolArn":"arn:aws:devicefarm:us-west-2::devicepool:082d10e5-d7d7-48a5-ba5c-
b33d66efa1f5",
    "namePrefix":"MyAppTest",
    # This is our test package. This tutorial won't go into how to make these.
    "testPackage":"tests.zip"
}

client = boto3.client('devicefarm')

unique =
 config['namePrefix']+"-"+(datetime.date.today().isoformat())+(''.join(random.sample(string.asc

print(f"The unique identifier for this run is going to be {unique} -- all uploads will
 be prefixed with this.")

def upload_df_file(filename, type_, mime='application/octet-stream'):
    response = client.create_upload(projectArn=config['projectArn'],
        name = (unique)+"_"+os.path.basename(filename),
        type=type_,
        contentType=mime
        )
```

```
    # Get the upload ARN, which we'll return later.
    upload_arn = response['upload']['arn']
    # We're going to extract the URL of the upload and use Requests to upload it
    upload_url = response['upload']['url']
    with open(filename, 'rb') as file_stream:
        print(f"Uploading {filename} to Device Farm as {response['upload']['name']}...
 ",end='')
        put_req = requests.put(upload_url, data=file_stream, headers={"content-
type":mime})
        print(' done')
        if not put_req.ok:
            raise Exception("Couldn't upload, requests said we're not ok. Requests
 says: "+put_req.reason)
    started = datetime.datetime.now()
    while True:
        print(f"Upload of {filename} in state {response['upload']['status']} after
 "+str(datetime.datetime.now() - started))
        if response['upload']['status'] == 'FAILED':
            raise Exception("The upload failed processing. DeviceFarm says reason
 is: \n"+(response['upload']['message'] if 'message' in response['upload'] else
 response['upload']['metadata']))
        if response['upload']['status'] == 'SUCCEEDED':
            break
        time.sleep(5)
        response = client.get_upload(arn=upload_arn)
    print("")
    return upload_arn

our_upload_arn = upload_df_file(config['appFilePath'], "ANDROID_APP")
our_test_package_arn = upload_df_file(config['testPackage'],
 'APPIUM_PYTHON_TEST_PACKAGE')
print(our_upload_arn, our_test_package_arn)
# Now that we have those out of the way, we can start the test run...
response = client.schedule_run(
    projectArn = config["projectArn"],
    appArn = our_upload_arn,
    devicePoolArn = config["poolArn"],
    name=unique,
    test = {
        "type":"APPIUM_PYTHON",
        "testSpecArn": config["testSpecArn"],
        "testPackageArn": our_test_package_arn
        }
    )
```

```
run_arn = response['run']['arn']
start_time = datetime.datetime.now()
print(f"Run {unique} is scheduled as arn {run_arn} ")

try:

    while True:
        response = client.get_run(arn=run_arn)
        state = response['run']['status']
        if state == 'COMPLETED' or state == 'ERRORED':
            break
        else:
            print(f" Run {unique} in state {state}, total time
 "+str(datetime.datetime.now()-start_time))
            time.sleep(10)
except:
    # If something goes wrong in this process, we stop the run and exit.

    client.stop_run(arn=run_arn)
    exit(1)
print(f"Tests finished in state {state} after "+str(datetime.datetime.now() -
 start_time))
# now, we pull all the logs.
jobs_response = client.list_jobs(arn=run_arn)
# Save the output somewhere. We're using the unique value, but you could use something
 else
save_path = os.path.join(os.getcwd(), unique)
os.mkdir(save_path)
# Save the last run information
for job in jobs_response['jobs'] :
    # Make a directory for our information
    job_name = job['name']
    os.makedirs(os.path.join(save_path, job_name), exist_ok=True)
    # Get each suite within the job
    suites = client.list_suites(arn=job['arn'])['suites']
    for suite in suites:
        for test in client.list_tests(arn=suite['arn'])['tests']:
            # Get the artifacts
            for artifact_type in ['FILE','SCREENSHOT','LOG']:
                artifacts = client.list_artifacts(
                    type=artifact_type,
                    arn = test['arn']
                )['artifacts']
                for artifact in artifacts:
```

```
                    # We replace : because it has a special meaning in Windows & macos
                    path_to = os.path.join(save_path, job_name, suite['name'],
 test['name'].replace(':','_') )
                    os.makedirs(path_to, exist_ok=True)
                    filename =
 artifact['type']+"_"+artifact['name']+"."+artifact['extension']
                    artifact_save_path = os.path.join(path_to, filename)
                    print("Downloading "+artifact_save_path)
                    with open(artifact_save_path, 'wb') as fn,
 requests.get(artifact['url'],allow_redirects=True) as request:
                        fn.write(request.content)
                    #/for artifact in artifacts
                #/for artifact type in []
            #/ for test in ()[]
        #/ for suite in suites
    #/ for job in _[]
# done
print("Finished")
```

Device Farm 错误故障排除

在本节中，您将找到可帮助修复 Device Farm 常见问题的错误消息和程序。

> **Note**
>
> 要对在 Device Farm 上意外失败的 Appium 测试进行故障排除，请参阅我们的客户端 Appium 测试指南

主题

在 AWS Device Farm 中对 Android 应用程序测试进行故障排除

以下主题列出了在上传 Android 应用程序测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 以下说明基于 Linux x86_64 和 Mac。

ANDROID_APP_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的应用程序。请验证文件是否有效，然后重试。

确保您可以解压应用程序包，而不会出现错误。在以下示例中，程序包的名称为 app-debug.apk。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip app-debug.apk
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   有效的 Android 应用程序包应生成类似以下内容的输出：

   ```
   .
   |-- AndroidManifest.xml
   |-- classes.dex
   |-- resources.arsc
   |-- assets (directory)
   |-- res (directory)
   `-- META-INF (directory)
   ```

   有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

ANDROID_APP_AAPT_DEBUG_BADGING_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法提取有关您的应用程序的信息。请通过运行命令 `aapt debug badging` *`<path to your test package>`* 来验证应用程序是否有效，并在命令不输出任何错误后重试。

在上传验证过程中，AWS Device Farm 解析 `aapt debug badging` *`<path to your package>`* 命令输出中的信息。

确保您可以在 Android 应用程序上成功运行此命令。在以下示例中，程序包的名称为 app-debug.apk。

- 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ aapt debug badging app-debug.apk
```

有效的 Android 应用程序包应生成类似以下内容的输出：

```
package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
 versionName='1.0' platformBuildVersionName='5.1.1-1819727'
sdkVersion:'9'
application-label:'ReferenceApp'
application: label='ReferenceApp' icon='res/mipmap-mdpi-v4/ic_launcher.png'
application-debuggable
launchable-activity:
 name='com.amazon.aws.adf.android.referenceapp.Activities.MainActivity'
 label='ReferenceApp' icon=''
uses-feature: name='android.hardware.bluetooth'
uses-implied-feature: name='android.hardware.bluetooth' reason='requested
 android.permission.BLUETOOTH permission, and targetSdkVersion > 4'
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '160' '213' '240' '320' '480' '640'
```

有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

ANDROID_APP_PACKAGE_NAME_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的应用程序中找到程序包名称值。请通过运行命令 `aapt debug badging` *`<path to your test package>`* 来验证应用程序是否有效，并在关键字“package: name”后面找到程序包名称值后重试。

在上传验证过程中，AWS Device Farm 解析 `aapt debug badging` *<path to your package>* 命令输出中的程序包名称值。

确保您可以在 Android 应用程序上运行此命令并成功找到程序包名称值。在以下示例中，程序包的名称为 app-debug.apk。

- 将您的应用程序包复制到工作目录，然后运行以下命令：

  ```
  $ aapt debug badging app-debug.apk | grep "package: name="
  ```

  有效的 Android 应用程序包应生成类似以下内容的输出：

  ```
  package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
   versionName='1.0' platformBuildVersionName='5.1.1-1819727'
  ```

  有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

ANDROID_APP_SDK_VERSION_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的应用程序中找到开发工具包版本值。请通过运行命令 `aapt debug badging` *<path to your test package>* 来验证应用程序是否有效，并在关键字 `sdkVersion` 后面找到开发工具包版本值后重试。

在上传验证过程中，AWS Device Farm 解析 `aapt debug badging` *<path to your package>* 命令输出中的开发工具包版本值。

确保您可以在 Android 应用程序上运行此命令并成功找到程序包名称值。在以下示例中，程序包的名称为 app-debug.apk。

- 将您的应用程序包复制到工作目录，然后运行以下命令：

  ```
  $ aapt debug badging app-debug.apk | grep "sdkVersion"
  ```

  有效的 Android 应用程序包应生成类似以下内容的输出：

```
sdkVersion:'9'
```

有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

ANDROID_APP_AAPT_DUMP_XMLTREE_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们在您的应用程序中找不到有效的 AndroidManifest .xml。请通过运行命令 `aapt dump xmltree <path to your test package> AndroidManifest.xml` 验证测试程序包是否有效，然后在该命令未输出任何错误后重试。

在上传验证过程中，AWS Device Farm 使用命令 `aapt dump xmltree <path to your package> AndroidManifest.xml` 解析程序包中包含的 XML 文件的 XML 解析树中的信息。

确保您可以在 Android 应用程序上成功运行此命令。在以下示例中，程序包的名称为 app-debug.apk。

- 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ aapt dump xmltree app-debug.apk. AndroidManifest.xml
```

有效的 Android 应用程序包应生成类似以下内容的输出：

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
    A: package="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: uses-permission (line=11)
```

```
      A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
 "android.permission.INTERNET")
    E: uses-permission (line=12)
      A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
 "android.permission.CAMERA")
```

有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

ANDROID_APP_DEVICE_ADMIN_PERMISSIONS

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现，您的应用程序需要设备管理员权限。请通过运行命令 `aapt dump xmltree <path to your test package> AndroidManifest.xml` 确认不需要此权限，并在确保输出不包含关键字 `android.permission.BIND_DEVICE_ADMIN` 后重试。

在上传验证过程中，AWS Device Farm 使用命令 `aapt dump xmltree <path to your package> AndroidManifest.xml` 解析程序包中包含的 XML 文件的 XML 解析树中的权限信息。

请确保您的应用程序不需要设备管理员权限。在以下示例中，程序包的名称为 app-debug.apk。

- 将您的应用程序包复制到工作目录，然后运行以下命令：

  ```
  $ aapt dump xmltree app-debug.apk AndroidManifest.xml
  ```

  您应找到类似以下内容的输出：

  ```
  N: android=http://schemas.android.com/apk/res/android
    E: manifest (line=2)
      A: android:versionCode(0x0101021b)=(type 0x10)0x1
      A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
      A: package="com.amazonaws.devicefarm.android.referenceapp" (Raw:
   "com.amazonaws.devicefarm.android.referenceapp")
      A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
      A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
      E: uses-sdk (line=7)
        A: android:minSdkVersion(0x0101020c)=(type 0x10)0xa
  ```

```
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: uses-permission (line=11)
      A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
"android.permission.INTERNET")
    E: uses-permission (line=12)
      A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
"android.permission.CAMERA")
        ……
```

如果 Android 应用程序有效，则输出不应包含以下内容：`A: android:name(0x01010003)="android.permission.BIND_DEVICE_ADMIN" (Raw: "android.permission.BIND_DEVICE_ADMIN")`。

有关更多信息，请参阅 AWS Device Farm 中的 Android 测试。

我的 Android 应用程序中的某些窗口显示空白或黑屏

如果您正在测试 Android 应用程序，并且注意到该应用程序中的某些窗口在 Device Farm 的测试视频录制中出现黑屏，则您的应用程序可能正在使用 Android 的 FLAG_SECURE 功能。此标志（如 Android 官方文档中所述）用于防止屏幕录制工具录制应用程序的某些窗口。因此，如果窗口使用此标志，Device Farm 的屏幕录制功能（用于自动化和远程访问测试）可能会在应用程序窗口的位置显示黑屏。

开发人员经常将此标志用于其应用程序中包含敏感信息（例如登录页面）的页面。如果您在某些页面（例如登录页面）上看到应用程序屏幕出现黑屏，请与您的开发人员合作，获取不使用此标志进行测试的应用程序版本。

此外，请注意，Device Farm 仍然可以与带有此标志的应用程序窗口进行交互。因此，如果您的应用程序的登录页面显示为黑屏，您仍然可以输入凭据以登录应用程序（从而查看未被 FLAG_SECURE 标志屏蔽的页面）。

在 AWS Device Farm 中对 Appium Java JUnit 测试进行故障排除

以下主题列出了在上传 Appium Java JUnit 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 以下说明基于 Linux x86_64 和 Mac。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   有效的 Appium Java JUnit 程序包应生成类似以下内容的输出：

   ```
   .
   |– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |– zip-with-dependencies.zip (this .zip file contains all of the items)
   `– dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
         |– com.some-dependency.bar-4.1.jar
         |– com.another-dependency.thing-1.0.jar
         |– joda-time-2.7.jar
         `– log4j-1.2.14.jar
   ```

   有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 dependency-jars 目录。请解压缩您的测试程序包，验证 dependency-jars 目录位于该程序包中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将在工作目录中找到 *dependency-jars* 目录：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DI

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 dependency-jars 目录树中找到 JAR 文件。请解压缩您的测试程序包，打开 dependency-jars 目录，并验证至少有一个 JAR 文件在该目录中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将在 *dependency-jars* 目录中至少找到一个 *jar* 文件：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 *-tests.jar 文件。请解压缩您的测试程序包，验证至少有一个 *-tests.jar 文件位于该程序包中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将至少找到一个 *jar* 文件，例如我们的示例中的 *acme-android-appium-1.0-SNAPSHOT-tests.jar*。文件的名称可能不同，但它应以 *-tests.jar* 结尾。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JA

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在测试 JAR 文件中找到类文件。请解压缩您的测试程序包，解压测试 JAR 文件，并验证至少有一个类文件在该 JAR 文件中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应至少找到一个 jar 文件，例如我们的示例中的 *acme-android-appium-1.0-SNAPSHOT-tests.jar*。文件的名称可能不同，但它应以 *-tests.jar* 结尾。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. 成功提取文件后，您应通过运行以下命令在工作目录树中至少找到一个类：

```
$ tree .
```

您应看到类似如下的输出：

```
.
```

```
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `-another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法找到 JUnit 版本值。请解压缩您的测试程序包，打开 dependency-jars 目录，并验证 JUnit JAR 文件在该目录中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
tree .
```

输出应该如下所示：

```
.
```

```
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
    |— junit-4.10.jar
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

如果 Appium Java JUnit 程序包有效，您将找到类似于我们的示例中的 jar 文件 *junit-4.10.jar* 的 JUnit 依赖项文件。名称应该包括关键字 *junit* 和其版本号，在此示例中为 4.10。

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现 JUnit 版本低于我们支持的最低版本 4.10。请更改 JUnit 版本，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   您应找到 JUnit 依赖项文件（例如我们的示例中的 *junit-4.10.jar*）及其版本号（在我们的示例中为 4.10）：

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– junit-4.10.jar
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

> **Note**
>
> 如果测试程序包中指定的 JUnit 版本低于我们支持的最低版本 4.10，您的测试可能无法正确执行。

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

对 AWS Device Farm 中的 Appium Java JUnit Web 应用程序测试进行故障排除

以下主题列出了在上传 Appium Java JUnit Web 应用程序测试期间出现的错误消息，并推荐了解决每个错误的解决方法。有关将 Appium 与 Device Farm 配合使用的更多信息，请参阅 the section called “Appium 自动测试”。

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 Appium Java JUnit 包应生成如下所示的输出：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
> 我们无法在您的测试程序包中找到 dependency-jars 目录。请解压缩您的测试程序包，验证 dependency-jars 目录位于该程序包中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，则可以在工作*dependency-jars*目录中找到该目录：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDEN

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 dependency-jars 目录树中找到 JAR 文件。请解压缩您的测试程序包，打开 dependency-jars 目录，并验证至少有一个 JAR 文件在该目录中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，您将在目录中找到至少一个*jar*文件：*dependency-jars*

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 *-tests.jar 文件。请解压缩您的测试程序包，验证至少有一个 *-tests.jar 文件位于该程序包中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，您会发现至少一个*jar*文件，如我们的示例*acme-android-appium-1.0-SNAPSHOT-tests.jar*所示。文件名可能不同，但应以结尾*-tests.jar*。

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TES

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在测试 JAR 文件中找到类文件。请解压缩您的测试程序包，解压测试 JAR 文件，并验证至少有一个类文件在该 JAR 文件中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   你应该找到至少一个 jar 文件，就像*acme-android-appium-1.0-SNAPSHOT-tests.jar*我们的例子一样。文件名可能不同，但应以结尾*-tests.jar*。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. 成功提取文件后，您应通过运行以下命令在工作目录树中至少找到一个类：

```
$ tree .
```

您应看到类似如下的输出：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `-another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNK

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们找不到 JUnit 版本值。请解压缩您的测试包并打开 dependency-jars 目录，验证 J JUnit AR 文件是否在该目录中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
tree .
```

输出应该如下所示：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

如果 Appium Java JUnit 包有效，您将在我们的示例*junit-4.10.jar*中找到类似于 jar 文件的 JUnit 依赖文件。名称应由关键字*junit*及其版本号组成，在本例中为 4.10。

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现该 JUnit 版本低于我们支持的最低版本 4.10。请更改 JUnit版本并重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

你应该找到一个像我们的例子一样*junit-4.10.jar*的 JUnit 依赖文件及其版本号，在我们的例子中是 4.10：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

> **Note**
>
> 如果您的测试包中指定的 JUnit 版本低于我们支持的最低版本 4.10，则您的测试可能无法正确执行。

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

在 AWS Device Farm 中对 Appium Java TestNG 测试进行故障排除

以下主题列出了在上传 Appium Java TestNG 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 以下说明基于 Linux x86_64 和 Mac。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip zip-with-dependencies.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   有效的 Appium Java JUnit 程序包应生成类似以下内容的输出：

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
    built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
    everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars  (this is the directory that contains all of your dependencies,
    built as JAR files)
   ```

```
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 dependency-jars 目录。请解压缩您的测试程序包，验证 dependency-jars 目录位于该程序包中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将在工作目录中找到 *dependency-jars* 目录。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
```

```
      `– log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 dependency-jars 目录树中找到 JAR 文件。请解压缩您的测试程序包，打开 dependency-jars 目录，并验证至少有一个 JAR 文件在该目录中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将在 *dependency-jars* 目录中至少找到一个 *jar* 文件。

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 *-tests.jar 文件。请解压缩您的测试程序包，验证至少有一个 *-tests.jar 文件位于该程序包中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 程序包有效，您将至少找到一个 *jar* 文件，例如我们的示例中的 *acme-android-appium-1.0-SNAPSHOT-tests.jar*。文件的名称可能不同，但它应以 *-tests.jar* 结尾。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在测试 JAR 文件中找到类文件。请解压缩您的测试程序包，解压测试 JAR 文件，并验证至少有一个类文件在该 JAR 文件中，然后重试。

在以下示例中，程序包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应至少找到一个 jar 文件，例如我们的示例中的 *acme-android-appium-1.0-SNAPSHOT-tests.jar*。文件的名称可能不同，但它应以 *-tests.jar* 结尾。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. 要从 jar 文件中提取文件，可以运行以下命令：

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. 在您成功提取文件后，运行以下命令：

```
$ tree .
```

您应在工作目录树中至少找到一个类：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `— another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

在 AWS Device Farm 中对 Appium Java TestNG Web 应用程序进行故障排除

以下主题列出了在上传 Appium Java TestNG Web 应用程序测试期间出现的错误消息并推荐了解决方法来解决每个错误。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 Appium Java JUnit 包应生成如下所示的输出：

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSIN

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 dependency-jars 目录。请解压缩您的测试程序包，验证 dependency-jars 目录位于该程序包中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，则可以在工作*dependency-jars*目录中找到该目录。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPEND

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 dependency-jars 目录树中找到 JAR 文件。请解压缩您的测试程序包，打开 dependency-jars 目录，并验证至少有一个 JAR 文件在该目录中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，您将在该目录中*dependency-jars*找到至少一个*jar*文件。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 *-tests.jar 文件。请解压缩您的测试程序包，验证至少有一个 *-tests.jar 文件位于该程序包中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Java JUnit 软件包有效，您会发现至少一个*jar*文件，如我们的示例*acme-android-appium-1.0-SNAPSHOT-tests.jar*所示。文件名可能不同，但应以结尾*-tests.jar*。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_T

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在测试 JAR 文件中找到类文件。请解压缩您的测试程序包，解压测试 JAR 文件，并验证至少有一个类文件在该 JAR 文件中，然后重试。

在以下示例中，软件包的名称为 zip-with-dependencies.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip zip-with-dependencies.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

你应该找到至少一个 jar 文件，就像*acme-android-appium-1.0-SNAPSHOT-tests.jar*我们的例子一样。文件名可能不同，但应以结尾*-tests.jar*。

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. 要从 jar 文件中提取文件，可以运行以下命令：

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. 在您成功提取文件后，运行以下命令：

```
$ tree .
```

您应在工作目录树中至少找到一个类：

```
.
|– acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|– acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `– another-class-file.class
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

在 AWS Device Farm 中对 Appium Python 测试进行故障排除

以下主题列出了在上传 Appium Python 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

APPIUM_PYTHON_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的 Appium 测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 Appium Python 程序包应生成类似以下内容的输出：

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 wheelhouse 目录树中找到依赖项 wheel 文件。请解压缩您的测试程序包，打开 wheelhouse 目录，并验证至少有一个 wheel 文件在该目录中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 程序包有效，您将在 *wheelhouse* 目录中找到至少一个类似于突出显示的文件的 *.whl* 依赖项文件。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现至少有一个 wheel 文件指定了我们不支持的平台。请解压缩您的测试程序包，打开 wheelhouse 目录，并验证 wheel 文件的名称以 -any.whl 或 -linux_x86_64.whl 结尾，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 程序包有效，您将在 *wheelhouse* 目录中找到至少一个类似于突出显示的文件的 *.whl* 依赖项文件。文件的名称可能不同，但它应以指定该平台的 *-any.whl* 或 *-linux_x86_64.whl* 结尾。其他任何平台，如 windows，均不受支持。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 tests 目录。请解压缩您的测试程序包，验证 tests 目录位于该程序包中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 程序包有效，您将在工作目录中找到 *tests* 目录。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 tests 目录树中找到有效的测试文件。请解压缩您的程序包，打开 tests 目录，并验证至少有一个文件的名称以关键字“test”开头或结尾，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip test_bundle.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   如果 Appium Python 程序包有效，您将在工作目录中找到 *tests* 目录。文件的名称可能不同，但它应以 *test_* 开头或以 *_test.py* 结尾。

   ```
   .
   |-- requirements.txt
   ```

```
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 requirements.txt 文件。请解压缩您的测试程序包，验证 requirements.txt 文件位于该程序包中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip test_bundle.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   如果 Appium Python 程序包有效，您将在工作目录中找到 *requirements.txt* 文件。

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `-- test_unittest.py
   `-- wheelhouse (directory)
   ```

```
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现 pytest 版本低于我们支持的最低版本 2.8.0。请更改 requirements.txt 文件内的 pytest 版本，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应在工作目录中找到 *requirements.txt* 文件。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `--test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
```

```
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

3. 要获取 pytest 版本，您可以运行以下命令：

```
$ grep "pytest" requirements.txt
```

您应找到类似以下内容的输出：

```
pytest==2.9.0
```

它显示了 pytest 版本，在本例中为 2.9.0。如果 Appium Python 程序包有效，pytest 版本应当高于或等于 2.8.0。

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAIL

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们未能安装依赖项 wheel。请解压缩您的测试程序包，打开 requirements.txt 文件和 wheelhouse 目录，并验证 requirements.txt 文件中指定的依赖项 wheel 与 wheelhouse 目录中的依赖项 wheel 完全匹配，然后重试。

我们强烈建议您为包装测试设置 Python virtualenv。以下是使用 Python virtualenv 创建虚拟环境然后将其激活的示例流程：

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 要测试安装 wheel 文件，您可以运行以下命令：

```
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
requirements.txt
```

有效的 Appium Python 程序包应生成类似以下内容的输出：

```
Ignoring indexes: https://pypi.python.org/simple
Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
 selenium-2.52.0 wheel-0.26.0
```

3. 要停用虚拟环境，您可以运行以下命令：

```
$ deactivate
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们未能在 tests 目录中收集测试。请解压缩您的测试程序包，通过运行命令 `py.test --collect-only <path to your tests directory>` 验证测试程序包是否有效，然后在该命令未输出任何错误后重试。

我们强烈建议您为包装测试设置 Python virtualenv。以下是使用 Python virtualenv 创建虚拟环境然后将其激活的示例流程：

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip test_bundle.zip
   ```

2. 要安装 wheel 文件，您可以运行以下命令：

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

3. 要收集测试，您可以运行以下命令：

   ```
   $ py.test --collect-only tests
   ```

   有效的 Appium Python 程序包应生成类似以下内容的输出：

   ```
   ==================== test session starts ====================
   platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
   rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
   collected 1 items
   <Module 'test_unittest.py'>
     <UnitTestCase 'DeviceFarmAppiumWebTests'>
       <TestCaseFunction 'test_devicefarm'>

   ==================== no tests ran in 0.11 seconds ====================
   ```

4. 要停用虚拟环境，您可以运行以下命令：

   ```
   $ deactivate
   ```

   有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEELS_INSUFFICIENT

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 wheelhouse 目录中找到足够的 wheel 依赖项。请解压缩您的测试包，然后打开 wheelhouse 目录。确认您已在 requirements.txt 文件中指定了所有 wheel 依赖项。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip test_bundle.zip
   ```

2. 检查 *requirements.txt* 文件的长度以及 wheelhouse 目录中 *.whl* 依赖文件的数量：

   ```
   $ cat requirements.txt | egrep "." |wc -l
       12
   $ ls wheelhouse/ | egrep ".+\.whl" | wc -l
       11
   ```

   如果 *.whl* 依赖文件的数量小于 *requirements.txt* 文件中的非空行数，则需要确保以下几点：

   - *requirements.txt* 文件中的每一行都有一个与 *.whl* 相关的文件。
   - *requirements.txt* 文件中没有其他行包含依赖项包名称以外的信息。
   - *requirements.txt* 文件中没有在多行中重复依赖项名称，因此该文件中的两行可能对应于一个 *.whl* 依赖文件。

   AWS Device Farm 不支持 *requirements.txt* 文件中与依赖项包不直接对应的行，例如为`pip install` 命令指定全局选项的行。有关全局选项的列表，请参阅要求文件格式。

   有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

在 AWS Device Farm 中对 Appium Python Web 应用程序测试进行故障排除

以下主题列出了在上传 Appium Python Web 应用程序测试期间出现的错误消息并推荐了解决方法来解决每个错误。

APPIUM_WEB_PYTHON_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的 Appium 测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 Appium Python 程序包应生成类似以下内容的输出：

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 wheelhouse 目录树中找到依赖项 wheel 文件。请解压缩您的测试程序包，打开 wheelhouse 目录，并验证至少有一个 wheel 文件在该目录中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 包有效，您将在目录中找到至少一个*.whl*依赖文件，例如突出显示的*wheelhouse*文件。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现至少有一个 wheel 文件指定了我们不支持的平台。请解压缩您的测试程序包，打开 wheelhouse 目录，并验证 wheel 文件的名称以 -any.whl 或 -linux_x86_64.whl 结尾，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 包有效，您将在目录中找到至少一个*.whl*依赖文件，例如突出显示的*wheelhouse*文件。文件名可能不同，但应以*-any.whl*或结尾*-linux_x86_64.whl*，后者指定平台。其他任何平台，如 windows，均不受支持。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 tests 目录。请解压缩您的测试程序包，验证 tests 目录位于该程序包中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 包有效，你将在工作*tests*目录中找到该目录。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 tests 目录树中找到有效的测试文件。请解压缩您的程序包，打开 tests 目录，并验证至少有一个文件的名称以关键字“test”开头或结尾，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 包有效，你将在工作*tests*目录中找到该目录。文件名可能不同，但应以开头*test_*或结尾为*_test.py*。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISS

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 requirements.txt 文件。请解压缩您的测试程序包，验证 requirements.txt 文件位于该程序包中，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 Appium Python 包有效，你将在工作目录中找到该*requirements.txt*文件。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现 pytest 版本低于我们支持的最低版本 2.8.0。请更改 requirements.txt 文件内的 pytest 版本，然后重试。

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在工作目录中找到该*requirement.txt*文件。

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `--test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

3. 要获取 pytest 版本，您可以运行以下命令：

```
$ grep "pytest" requirements.txt
```

您应找到类似以下内容的输出：

```
pytest==2.9.0
```

它显示了 pytest 版本，在本例中为 2.9.0。如果 Appium Python 程序包有效，pytest 版本应当高于或等于 2.8.0。

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们未能安装依赖项 wheel。请解压缩您的测试程序包，打开 requirements.txt 文件和 wheelhouse 目录，并验证 requirements.txt 文件中指定的依赖项 wheel 与 wheelhouse 目录中的依赖项 wheel 完全匹配，然后重试。

我们强烈建议您为包装测试设置 Python virtualenv。以下是使用 Python virtualenv 创建虚拟环境然后将其激活的示例流程：

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip test_bundle.zip
   ```

2. 要测试安装 wheel 文件，您可以运行以下命令：

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

   有效的 Appium Python 程序包应生成类似以下内容的输出：

   ```
   Ignoring indexes: https://pypi.python.org/simple
   Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
   Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
   Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
   Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
   ```

```
Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
 selenium-2.52.0 wheel-0.26.0
```

3. 要停用虚拟环境，您可以运行以下命令：

```
$ deactivate
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

APPIUM_WEB_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们未能在 tests 目录中收集测试。请解压缩您的测试程序包，通过运行命令“py.test --collect-only <path to your tests directory>”验证测试程序包是否有效，然后在该命令未输出任何错误后重试。

我们强烈建议您为包装测试设置 Python virtualenv。以下是使用 Python virtualenv 创建虚拟环境然后将其激活的示例流程：

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 test_bundle.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip test_bundle.zip
```

2. 要安装 wheel 文件，您可以运行以下命令：

```
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
requirements.txt
```

3. 要收集测试，您可以运行以下命令：

```
$ py.test --collect-only tests
```

有效的 Appium Python 程序包应生成类似以下内容的输出：

```
==================== test session starts ====================
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
collected 1 items
<Module 'test_unittest.py'>
  <UnitTestCase 'DeviceFarmAppiumWebTests'>
    <TestCaseFunction 'test_devicefarm'>

==================== no tests ran in 0.11 seconds ====================
```

4. 要停用虚拟环境，您可以运行以下命令：

```
$ deactivate
```

有关更多信息，请参阅 在 Device Farm 中自动运行 Appium 测试。

在 AWS Device Farm 中对 Instrumentation 测试进行故障排除

以下主题列出了在上传 Instrumentation 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 有关在 AWS Device Farm 中使用 Instrumentation 测试时的重要注意事项，请参阅适用于 Android 的 Instrumentation 与 AWS Device Farm。

INSTRUMENTATION_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

```
Warning: We could not open your test APK file. Please verify that the file is valid and
    try again.
```

确保您可以解压测试程序包，而不会出现错误。在以下示例中，程序包的名称为 app-debug-androidTest-unaligned.apk。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip app-debug-androidTest-unaligned.apk
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   有效的 Instrumentation 测试程序包将生成类似以下内容的输出：

   ```
   .
   |-- AndroidManifest.xml
   |--  classes.dex
   |-- resources.arsc
   |-- LICENSE-junit.txt
   |-- junit (directory)
   `-- META-INF (directory)
   ```

   有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

INSTRUMENTATION_TEST_PACKAGE_AAPT_DEBUG_BADGING_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not extract information about your test package. Please verify that the
      test package is valid by running the command "aapt debug badging <path to your
 test
      package>", and try again after the command does not print any error.
```

在上传验证过程中，Device Farm 解析 `aapt debug badging <path to your package>` 命令输出中的信息。

确保您可以对 Instrumentation 测试程序包成功运行此命令。

在以下示例中，程序包的名称为 app-debug-androidTest-unaligned.apk。

- 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ aapt debug badging app-debug-androidTest-unaligned.apk
```

有效的 Instrumentation 测试程序包将生成类似以下内容的输出：

```
package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
 versionName='' platformBuildVersionName='5.1.1-1819727'
sdkVersion:'9'
targetSdkVersion:'22'
application-label:'Test-api'
application: label='Test-api' icon=''
application-debuggable
uses-library:'android.test.runner'
feature-group: label=''
uses-feature: name='android.hardware.touchscreen'
uses-implied-feature: name='android.hardware.touchscreen' reason='default feature
 for all apps'
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--_--'
densities: '160'
```

有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_RUNNER_VALU

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the instrumentation runner value in the AndroidManifest.xml.
      Please verify the test package is valid by running the command "aapt dump xmltree
 <path to
      your test package> AndroidManifest.xml", and try again after finding the
 instrumentation
      runner value behind the keyword "instrumentation."
```

在上传验证过程中，Device Farm 解析程序包中包含的 XML 文件的 XML 解析树中的 Instrumentation 运行程序值。您可使用以下命令：`aapt dump xmltree <path to your package> AndroidManifest.xml`。

确保您可以对 Instrumentation 测试程序包运行此命令并成功找到 Instrumentation 值。

在以下示例中，程序包的名称为 app-debug-androidTest-unaligned.apk。

- 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml | grep
 -A5 "instrumentation"
```

有效的 Instrumentation 测试程序包将生成类似以下内容的输出：

```
E: instrumentation (line=9)
      A: android:label(0x01010001)="Tests for
 com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
 com.amazon.aws.adf.android.referenceapp")
      A:
 android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
 "android.support.test.runner.AndroidJUnitRunner")
      A:
 android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
      A: android:handleProfiling(0x01010022)=(type 0x12)0x0
      A: android:functionalTest(0x01010023)=(type 0x12)0x0
```

有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the valid AndroidManifest.xml in your test package. Please
      verify that the test package is valid by running the command "aapt dump xmltree
 <path to
      your test package> AndroidManifest.xml", and try again after the command does not
 print any
      error.
```

在上传验证过程中，Device Farm 使用以下命令解析程序包中包含的 XML 文件的 XML 解析树中的信息：`aapt dump xmltree <path to your package> AndroidManifest.xml`。

确保您可以对 Instrumentation 测试程序包成功运行此命令。

在以下示例中，程序包的名称为 app-debug-androidTest-unaligned.apk。

- 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml
```

有效的 Instrumentation 测试程序包将生成类似以下内容的输出：

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: package="com.amazon.aws.adf.android.referenceapp.test" (Raw:
 "com.amazon.aws.adf.android.referenceapp.test")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=5)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: instrumentation (line=9)
      A: android:label(0x01010001)="Tests for
 com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
 com.amazon.aws.adf.android.referenceapp")
      A:
 android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
 "android.support.test.runner.AndroidJUnitRunner")
      A:
 android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
      A: android:handleProfiling(0x01010022)=(type 0x12)0x0
      A: android:functionalTest(0x01010023)=(type 0x12)0x0
    E: application (line=16)
      A: android:label(0x01010001)=@0x7f020000
      A: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
      E: uses-library (line=17)
        A: android:name(0x01010003)="android.test.runner" (Raw:
 "android.test.runner")
```

有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

INSTRUMENTATION_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISS

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the package name in your test package. Please verify that the
      test package is valid by running the command "aapt debug badging <path to your
 test
      package>", and try again after finding the package name value behind the keyword
 "package:
      name."
```

在上传验证过程中，Device Farm 解析以下命令输出中的程序包名称值：`aapt debug badging <path to your package>`。

确保您可以对 Instrumentation 测试程序包运行此命令并成功找到程序包名称值。

在以下示例中，程序包的名称为 app-debug-androidTest-unaligned.apk。

- 将您的测试程序包复制到工作目录，然后运行以下命令：

  ```
  $ aapt debug badging app-debug-androidTest-unaligned.apk | grep "package: name="
  ```

  有效的 Instrumentation 测试程序包将生成类似以下内容的输出：

  ```
  package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
   versionName='' platformBuildVersionName='5.1.1-1819727'
  ```

  有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

在 AWS Device Farm 中对 iOS 应用程序测试进行故障排除

以下主题列出了在上传 iOS 应用程序测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 以下说明基于 Linux x86_64 和 Mac。

IOS_APP_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的应用程序。请验证文件是否有效，然后重试。

确保您可以解压应用程序包，而不会出现错误。在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   有效的 iOS 应用程序包应生成类似以下内容的输出：

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                       |-- Info.plist
                       `-- (any other files)
   ```

   有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_PAYLOAD_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的应用程序中找到 .Payload 目录。请解压缩您的应用程序，验证 Payload 目录位于该程序包中，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 iOS 应用程序包有效，您将在工作目录中找到 *Payload* 目录。

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_APP_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Payload 目录中找到 .app 目录。请解压缩您的应用程序，打开 Payload 目录，并验证 .app 目录在该目录中，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 iOS 应用程序包有效，您将在 *Payload* 目录中找到 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*)。

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_PLIST_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 .app 目录中找到 Info.plist 文件。请解压缩您的应用程序，打开 .app 目录，并验证 Info.plist 文件在该目录中，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 iOS 应用程序包有效，您将在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件。

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
```

```
                    `-- (any other files)
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_CPU_ARCHITECTURE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到 CPU 架构值。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证已指定“UIRequiredDeviceCapabilities”键，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                    |-- Info.plist
                    `-- (any other files)
```

3. 要查找 CPU 架构值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

有效的 iOS 应用程序包应生成类似以下内容的输出：

```
['armv7']
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_PLATFORM_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到平台值。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证已指定“CFBundleSupportedPlatforms”键，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
```

```
                        `-- (any other files)
```

3. 要查找平台值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   有效的 iOS 应用程序包应生成类似以下内容的输出：

   ```
   ['iPhoneOS']
   ```

   有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_WRONG_PLATFORM_DEVICE_VALUE

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们发现 Info.plist 文件中的平台设备值错误。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证“CFBundleSupportedPlatforms”键值不包含关键字“simulator”，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                        |-- Info.plist
                        `-- (any other files)
```

3. 要查找平台值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   有效的 iOS 应用程序包应生成类似以下内容的输出：

   ```
   ['iPhoneOS']
   ```

   如果 iOS 应用程序有效，则该值不应包含关键字 `simulator`。

   有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_FORM_FACTOR_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到外形规格值。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证已指定“UIDeviceFamily”键，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

   您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                    |-- Info.plist
                    `-- (any other files)
```

3. 要查找外形规格值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIDeviceFamily']
```

   有效的 iOS 应用程序包应生成类似以下内容的输出：

```
[1, 2]
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_PACKAGE_NAME_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到程序包名称值。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证已指定“CFBundleIdentifier”键，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                    |-- Info.plist
                    `-- (any other files)
```

3. 要查找程序包名称值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleIdentifier']
```

有效的 iOS 应用程序包应生成类似以下内容的输出：

```
Amazon.AWSDeviceFarmiOSReferenceApp
```

有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

IOS_APP_EXECUTABLE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到可执行文件值。请解压缩您的应用程序，打开 .app 目录中的 Info.plist 文件，并验证已指定“CFBundleExecutable”键，然后重试。

在以下示例中，程序包的名称为 AWSDeviceFarmiOSReferenceApp.ipa。

1. 将您的应用程序包复制到工作目录，然后运行以下命令：

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *AWSDeviceFarmiOSReferenceApp.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
```

```
                        |-- Info.plist
                        `-- (any other files)
```

3. 要查找可执行文件值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleExecutable']
   ```

   有效的 iOS 应用程序包应生成类似以下内容的输出：

   ```
   AWSDeviceFarmiOSReferenceApp
   ```

   有关更多信息，请参阅 AWS Device Farm 中的 iOS 测试。

在 AWS Device Farm 中对 XCTest 测试进行故障排除

以下主题列出了在上传 XCTest 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 下面的说明假定您使用的是 MacOS。

XCTEST_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法打开您的测试 ZIP 文件。请验证文件是否有效，然后重试。

确保您可以解压应用程序包，而不会出现错误。在以下示例中，程序包的名称为 swiftExampleTests.xctest-1.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 XCTest 程序包应生成类似以下内容的输出：

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

有关更多信息，请参阅 [将 Device Farm 与 XCTest 适用于 iOS 的集成]。

XCTEST_TEST_PACKAGE_XCTEST_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在您的测试程序包中找到 .xctest 目录。请解压缩您的测试程序包，验证 .xctest 目录位于该程序包中，然后重试。

在以下示例中，程序包的名称为 swiftExampleTests.xctest-1.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest 程序包有效，您将在工作目录中找到一个其名称类似于 *swiftExampleTests.xctest* 的目录。该名称应以 *.xctest* 结尾。

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

有关更多信息，请参阅 将 Device Farm 与 XCTest 适用于 iOS 的集成。

XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 .xctest 目录中找到 Info.plist 文件。请解压缩您的测试程序包，打开 .xctest 目录，并验证 Info.plist 文件在该目录中，然后重试。

在以下示例中，程序包的名称为 swiftExampleTests.xctest-1.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest 程序包有效，您将在 *.xctest* 目录中找到 *Info.plist* 文件。在下面的示例中，该目录名为 *swiftExampleTests.xctest*。

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

有关更多信息，请参阅 将 Device Farm 与 XCTest 适用于 iOS 的集成。

XCTEST_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到程序包名称值。请解压缩您的测试程序包，然后打开 Info.plist 文件，确认指定了“CFBundleIdentifier”键，然后重试。

在以下示例中，程序包的名称为 swiftExampleTests.xctest-1.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.xctest* 目录 (例如我们示例中的 *swiftExampleTests.xctest*) 中找到 *Info.plist* 文件：

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

3. 要查找程序包名称值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
```

```
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

有效的 XCtest 应用程序包应生成类似以下内容的输出：

```
com.amazon.kanapka.swiftExampleTests
```

有关更多信息，请参阅 将 Device Farm 与 XCTest 适用于 iOS 的集成。

XCTEST_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

> **Warning**
>
> 我们无法在 Info.plist 文件中找到可执行文件值。请解压缩您的测试程序包，然后打开 Info.plist 文件，确认指定了“CFBundleExecutable”键，然后重试。

在以下示例中，程序包的名称为 swiftExampleTests.xctest-1.zip。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.xctest* 目录 (例如我们示例中的 *swiftExampleTests.xctest*) 中找到 *Info.plist* 文件：

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

3. 要查找程序包名称值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleExecutable']
```

有效的 XCtest 应用程序包应生成类似以下内容的输出：

```
swiftExampleTests
```

有关更多信息，请参阅 将 Device Farm 与 XCTest 适用于 iOS 的集成。

在 AWS Device Farm 中对 XCTest UI 测试进行故障排除

以下主题列出了在上传 XCTest UI 测试期间出现的错误消息并推荐了解决方法来解决每个错误。

> **Note**
>
> 以下说明基于 Linux x86_64 和 Mac。

XCTEST_UI_TEST_PACKAGE_UNZIP_FAILED

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not open your test IPA file. Please verify that the file is valid and try again.
```

确保您可以解压应用程序包，而不会出现错误。在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

有效的 iOS 应用程序包应生成类似以下内容的输出：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the Payload directory inside your test package. Please unzip your test package, verify that the Payload directory is inside the package, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在工作目录中找到 *Payload* 目录。

```
.
`-- Payload (directory)
```

```
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                       |-- Info.plist
                        |                       `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_APP_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the .app directory inside the Payload directory. Please unzip your test package and then open the Payload directory, verify that the .app directory is inside the directory, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在 *Payload* 目录中找到 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*)。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                       |-- Info.plist
                        |                       `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the Plugins directory inside the .app directory. Please
unzip your test package and then open the .app directory, verify that the
Plugins directory is inside the directory, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在 *.app* 目录中找到 *Plugins* 目录。在我们的示例中，该目录名为 *swift-sampleUITests-Runner.app*。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUGINS_DIR

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the .xctest directory inside the plugins directory.
Please unzip your test package and then open the plugins directory, verify
that the .xctest directory is inside the directory, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在 *Plugins* 目录中找到 *.xctest* 目录。在我们的示例中，该目录名为 *swift-sampleUITests.xctest*。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 [将 iOS XCTest 用户界面与 Device Farm 集成]。

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the Info.plist file inside the .app directory. Please
unzip your test package and then open the .app directory, verify that the
Info.plist file is inside the directory, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在 *.app* 目录中找到 *Info.plist* 文件。在下面的示例中，该目录名为 *swift-sampleUITests-Runner.app*。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTEST_DIR

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the Info.plist file inside the .xctest directory. Please
unzip your test package and then open the .xctest directory, verify that
the Info.plist file is inside the directory, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您将在 *.xctest* 目录中找到 *Info.plist* 文件。在下面的示例中，该目录名为 *swift-sampleUITests.xctest*。

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not the CPU architecture value in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the key "UIRequiredDeviceCapabilities" is specified,
and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

   ```
   .
   `-- Payload (directory)
   ```

```
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

3. 要查找 CPU 架构值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/
Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

   有效的 XCtest UI 程序包应生成类似以下内容的输出：

```
['armv7']
```

   有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PLATFORM_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the platform value in the Info.plist. Please unzip your
test package and then open the Info.plist file inside the .app directory,
verify that the key "CFBundleSupportedPlatforms" is specified, and try
again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. 要查找平台值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

有效的 XCtest UI 程序包应生成类似以下内容的输出：

```
['iPhoneOS']
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_WRONG_PLATFORM_DEVICE_VALUE

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We found the platform device value was wrong in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the value of the key "CFBundleSupportedPlatforms"
does not contain the keyword "simulator", and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                       |-- Info.plist
                       |-- Plugins (directory)
                       |       `swift-sampleUITests.xctest (directory)
                       |                         |-- Info.plist
                       |                         `-- (any other files)
                       `-- (any other files)
   ```

3. 要查找平台值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   ```

```
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

有效的 XCtest UI 程序包应生成类似以下内容的输出：

```
['iPhoneOS']
```

如果 XCTest UI 程序包有效，则该值不应包含关键字 `simulator`。

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_FORM_FACTOR_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not the form factor value in the Info.plist. Please unzip your
test package and then open the Info.plist file inside the .app directory,
verify that the key "UIDeviceFamily" is specified, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                       |-- Info.plist
                        |                       `-- (any other files)
```

```
                        `-- (any other files)
```

3. 要查找外形规格值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['UIDeviceFamily']
   ```

   有效的 XCtest UI 程序包应生成类似以下内容的输出：

   ```
   [1, 2]
   ```

   有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the package name value in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the key "CFBundleIdentifier" is specified, and try
again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. 要查找程序包名称值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['CFBundleIdentifier']
   ```

   有效的 XCtest UI 程序包应生成类似以下内容的输出：

   ```
   com.apple.test.swift-sampleUITests-Runner
   ```

   有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the executable value in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the key "CFBundleExecutable" is specified, and try
again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                        |-- Info.plist
                        |-- Plugins (directory)
                        |       `swift-sampleUITests.xctest (directory)
                        |                         |-- Info.plist
                        |                         `-- (any other files)
                        `-- (any other files)
```

3. 要查找可执行文件值，您可以使用 Xcode 或 Python 打开 Info.plist。

对于 Python，您可以通过运行以下命令来安装 biplist 模块：

```
$ pip install biplist
```

4. 接下来，打开 Python 并运行以下命令：

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleExecutable']
```

有效的 XCtest UI 程序包应生成类似以下内容的输出：

```
XCTRunner
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the package name value in the Info.plist file inside
the .xctest directory. Please unzip your test package and then open
the Info.plist file inside the .xctest directory, verify that the key
"CFBundleIdentifier" is specified, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                       |-- Info.plist
                       |-- Plugins (directory)
                       |       `swift-sampleUITests.xctest (directory)
                       |                       |-- Info.plist
                       |                       `-- (any other files)
                       `-- (any other files)
   ```

3. 要查找程序包名称值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   ```

```
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
swift-sampleUITests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

有效的 XCtest UI 程序包应生成类似以下内容的输出：

```
com.amazon.swift-sampleUITests
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_TEST_EXECUTABLE_VALUE_MISSING

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We could not find the executable value in the Info.plist file inside
the .xctest directory. Please unzip your test package and then open
the Info.plist file inside the .xctest directory, verify that the key
"CFBundleExecutable" is specified, and try again.
```

在以下示例中，程序包的名称为 swift-sample-UI.ipa。

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.ipa
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

您应该在 *.app* 目录 (例如我们示例中的 *swift-sampleUITests-Runner.app*) 中找到 *Info.plist* 文件：

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
```

```
                        `-- (any other files)
```

3. 要查找可执行文件值，您可以使用 Xcode 或 Python 打开 Info.plist。

   对于 Python，您可以通过运行以下命令来安装 biplist 模块：

   ```
   $ pip install biplist
   ```

4. 接下来，打开 Python 并运行以下命令：

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
   swift-sampleUITests.xctest/Info.plist')
   print info_plist['CFBundleExecutable']
   ```

   有效的 XCtest UI 程序包应生成类似以下内容的输出：

   ```
   swift-sampleUITests
   ```

   有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_MULTIPLE_APP_DIRS

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We found multiple .app directories inside your test package. Please unzip
your test package, verify that only a single .app directory is present
inside the package, then try again.
```

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

   ```
   $ unzip swift-sample-UI.zip
   ```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

   ```
   $ tree .
   ```

   如果 XCTest UI 程序包有效，您应在 .zip 测试程序包中找到类似于示例中的 `swift-sampleUITests-Runner.app` 的单个 `.app` 目录。

```
.
`--swift-sample-UI.zip--(directory)
    `-- swift-sampleUITests-Runner.app (directory)
            |-- Info.plist
            |-- Plugins (directory)
            |       `swift-sampleUITests.xctest (directory)
           |            |-- Info.plist
           |            `-- (any other files)
            `-- (any other files)
    `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_MULTIPLE_IPA_DIRS

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We found multiple .ipa directories inside your test package. Please unzip your test package, verify that only a single .ipa directory is present inside the package, then try again.
```

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您应在 .zip 测试程序包中找到类似于示例中的 `sampleUITests.ipa` 的单个 `.ipa` 目录。

```
.
`--swift-sample-UI.zip--(directory)
    `-- sampleUITests.ipa (directory)
            `-- Payload (directory)
                `-- swift-sampleUITests-Runner.app (directory)
    `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_BOTH_APP_AND_IPA_DIR_PRESENT

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We found both .app and .ipa files inside your test package. Please unzip
your test package, verify that only a single .app or .ipa file is present
inside the package, then try again.
```

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您应在 .zip 测试程序包中找到类似于示例中的 `sampleUITests.ipa` 的 `.ipa` 目录或类似于 `swift-sampleUITests-Runner.app` 的 `.app` 目录。您可以参考 将 iOS XCTest 用户界面与 Device Farm 集成 相关文档中的有效 XCTEST_UI 测试程序包示例。

```
.
`--swift-sample-UI.zip--(directory)
    `-- sampleUITests.ipa (directory)
            `-- Payload (directory)
                `-- swift-sampleUITests-Runner.app (directory)
   `-- (any other files)
```

或

```
.
`--swift-sample-UI.zip--(directory)
    `-- swift-sampleUITests-Runner.app (directory)
            |-- Info.plist
            |-- Plugins (directory)
            `-- (any other files)
```

```
    `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_PRESENT_IN_ZIP

如果您看到以下消息，请执行以下步骤来修复此问题。

```
We found a Payload directory inside your .zip test package. Please unzip your test package, ensure that a Payload directory is not present in the package, then try again.
```

1. 将您的测试程序包复制到工作目录，然后运行以下命令：

```
$ unzip swift-sample-UI.zip
```

2. 成功解压缩程序包后，您可以通过运行以下命令找到工作目录树结构：

```
$ tree .
```

如果 XCTest UI 程序包有效，您不应在测试程序包中找到 Payload 目录。

```
.
`--swift-sample-UI.zip--(directory)
    `-- swift-sampleUITests-Runner.app (directory)
            |-- Info.plist
            |-- Plugins (directory)
            `-- (any other files)
   `-- Payload (directory) [This directory should not be present]
            |-- (any other files)
   `-- (any other files)
```

有关更多信息，请参阅 将 iOS XCTest 用户界面与 Device Farm 集成。

安全性 AWS Device Farm

云安全 AWS 是重中之重。作为 AWS 客户，您可以受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。

安全是双方共同承担 AWS 的责任。责任共担模式将其描述为云的安全性和云中的安全性：

- 云安全 — AWS 负责保护在 AWS 云中运行 AWS 服务的基础架构。 AWS 还为您提供可以安全使用的服务。作为AWS 合规计划合规计划合规计划合的一部分，第三方审计师定期测试和验证我们安全的有效性。要了解适用于的合规计划 AWS Device Farm，请参阅按合规计划划分的 AWS 范围内服务 AWS 按合规计划。
- 云中的安全性：您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 Device Farm 时应用责任共担模式。以下主题说明如何配置 Device Farm 以实现您的安全性和合规性目标。您还将了解如何使用其他 AWS 服务来帮助您监控和保护 Device Farm 资源。

主题

AWS Device Farm 中的身份识别和访问管理

受众

您的使用方式 AWS Identity and Access Management (IAM) 因您的角色而异：

- 服务用户：如果您无法访问功能，请从管理员处请求权限（请参阅对 AWS Device Farm 身份验证和访问进行故障排除）
- 服务管理员：确定用户访问权限并提交权限请求（请参阅AWS Device Farm 如何与 IAM 配合使用）
- IAM 管理员：编写用于管理访问权限的策略（请参阅AWS Device Farm 基于身份的策略示例）

使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份进行身份验证 AWS 账户根用户，或者通过担任 IAM 角色进行身份验证。

您可以使用来自身份源的证书 AWS IAM Identity Center（例如（IAM Identity Center）、单点登录身份验证或 Google/Facebook 证书，以联合身份登录。有关登录的更多信息，请参阅《AWS 登录 用户指南》中的如何登录您的 AWS 账户。

对于编程访问， AWS 提供 SDK 和 CLI 来对请求进行加密签名。有关更多信息，请参阅《IAM 用户指南》中的适用于 API 请求的AWS 签名版本 4。

AWS 账户 root 用户

创建时 AWS 账户，首先会有一个名为 AWS 账户 root 用户的登录身份，该身份可以完全访问所有资源 AWS 服务 和资源。我们强烈建议不要使用根用户进行日常任务。有关要求根用户凭证的任务，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

IAM 用户和组

IAM 用户是对某个人员或应用程序具有特定权限的一个身份。建议使用临时凭证，而非具有长期凭证的 IAM 用户。有关更多信息，请参阅 IAM 用户指南中的要求人类用户使用身份提供商的联合身份验证才能 AWS 使用临时证书进行访问。

IAM 组指定一组 IAM 用户，便于更轻松地对大量用户进行权限管理。有关更多信息，请参阅《IAM 用户指南》中的 IAM 用户使用案例。

IAM 角色

IAM 角色是具有特定权限的身份，可提供临时凭证。您可以通过从用户切换到 IAM 角色（控制台）或调用 AWS CLI 或 AWS API 操作来代入角色。有关更多信息，请参阅《IAM 用户指南》中的担任角色的方法。

IAM 角色对于联合用户访问、临时 IAM 用户权限、跨账户访问、跨服务访问以及在 Amazon EC2 上运行的应用程序非常有用。有关更多信息，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

AWS Device Farm 如何与 IAM 配合使用

在使用 IAM 管理对 Device Farm 的访问之前，您应了解哪些 IAM 功能可与 Device Farm 结合使用。要全面了解 Device Farm 和其他 AWS 服务如何与 IAM 配合使用，请参阅 IAM 用户指南中的与 IAM 配合使用的AWS 服务。

主题

Device Farm 基于身份的策略

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。Device Farm 支持特定的操作、资源和条件键。要了解在 JSON 策略中使用的所有元素，请参阅《IAM 用户指南》 中的 IAM JSON 策略元素参考。

操作

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

JSON 策略的 `Action` 元素描述可用于在策略中允许或拒绝访问的操作。在策略中包含操作以授予执行关联操作的权限。

Device Farm 中的策略操作在操作前使用以下前缀：`devicefarm:`。例如，要授予某人使用 Device Farm 桌面浏览器测试 `CreateTestGridUrl` API 操作启动 Selenium 会话的权限，您应将 `devicefarm:CreateTestGridUrl` 操作添加到这些会话的策略中。策略语句必须包含 `Action` 或 `NotAction` 元素。Device Farm 定义了一组自己的操作，以描述您可以使用该服务执行的任务。

要在单个语句中指定多项操作，请使用逗号将它们隔开，如下所示：

```
"Action": [
      "devicefarm:action1",
      "devicefarm:action2"
```

您也可以使用通配符（*）指定多个操作。例如，要指定以单词 List 开头的所有操作，包括以下操作：

```
"Action": "devicefarm:List*"
```

要查看 Device Farm 操作的列表，请参阅《IAM 服务授权参考》中的 AWS Device Farm定义的操作。

资源

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

Resource JSON 策略元素指定要向其应用操作的一个或多个对象。作为最佳实践，请使用其 Amazon 资源名称（ARN）指定资源。对于不支持资源级权限的操作，请使用通配符 (*) 指示语句应用于所有资源。

```
"Resource": "*"
```

Amazon EC2 实例资源具有以下 ARN：

```
arn:${Partition}:ec2:${Region}:${Account}:instance/${InstanceId}
```

有关格式的更多信息 ARNs，请参阅 Amazon 资源名称 (ARNs) 和 AWS 服务命名空间。

例如，要在语句中指定 i-1234567890abcdef0 实例，请使用以下 ARN：

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/i-1234567890abcdef0"
```

要指定属于某个账户的所有实例，请使用通配符 (*)：

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/*"
```

无法对某个资源执行某些 Device Farm 操作，例如用于创建资源的操作。在这些情况下，您必须使用通配符（*）。

```
"Resource": "*"
```

许多 Amazon EC2 API 操作涉及多种资源。例如，`AttachVolume` 将一个 Amazon EBS 卷附加到一个实例，从而使 IAM 用户必须获得相应权限才能使用该卷和该实例。要在单个语句中指定多个资源，请 ARNs 用逗号分隔。

```
"Resource": [
      "resource1",
      "resource2"
```

要查看 Device Farm 资源类型及其列表 ARNs，请参阅《IAM 服务授权参考》 AWS Device Farm中定义的资源类型。要了解您可以在哪些操作中指定每个资源的 ARN，请参阅《IAM 服务授权参考》中的AWS Device Farm定义的操作。

条件键

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

`Condition` 元素根据定义的条件指定语句何时执行。您可以创建使用条件运算符（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。要查看所有 AWS 全局条件键，请参阅 IAM 用户指南中的AWS 全局条件上下文密钥。

Device Farm 定义了自己的一组条件键，还支持使用一些全局条件键。要查看所有 AWS 全局条件键，请参阅 IAM 用户指南中的AWS 全局条件上下文密钥。

有关 Device Farm 条件密钥的列表，请参阅《IAM 服务授权参考》中的 AWS Device Farm的条件密钥。要了解您可以对哪些操作和资源使用条件键，请参阅《IAM 服务授权参考》中的 AWS Device Farm定义的操作。

示例

要查看 Device Farm 基于身份的策略的示例，请参阅 AWS Device Farm 基于身份的策略示例。

Device Farm 基于资源的策略

Device Farm 不支持基于资源的策略。

访问控制列表

Device Farm 不支持访问控制列表 (ACLs)。

基于 Device Farm 标签的授权

您可以将标签附加到 Device Farm 资源或将请求中的标签传递到 Device Farm。要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name`aws:RequestTag/*key-name* 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。有关标记 Device Farm 资源的更多信息，请参阅 在 Device Farm 中标记。

要查看基于身份的策略（用于根据资源上的标签来限制对该资源的访问）的示例，请参阅 根据标签查看 Device Farm 桌面浏览器测试项目。

Device Farm IAM 角色

I AM 角色是您的 AWS 账户中具有特定权限的实体。

将临时凭证用于 Device Farm

Device Farm 支持使用临时凭证。

您可以使用临时凭证通过联合身份验证登录，以代入 IAM 角色或跨账户角色。您可以通过调用AssumeRole或之类的 AWS STS API 操作来获取临时安全证书GetFederationToken。

服务关联角色

服务相关角色允许 AWS 服务访问其他服务中的资源以代表您完成操作。服务关联角色显示在 IAM 账户中，并归该服务所有。IAM 管理员可以查看，但不能编辑服务相关角色的权限。

Device Farm 在 Device Farm 桌面浏览器测试功能中使用与服务相关的角色。有关这些角色的信息，请参阅开发者指南中的 Using Service-Linked Roles in Device Farm desktop browser testing。

服务角色

Device Farm 不支持服务角色。

此功能允许服务代表您担任服务角色。此角色允许服务访问其他服务中的资源以代表您完成操作。服务角色显示在 IAM 账户中，并归该账户所有。这意味着，IAM 管理员可以更改该角色的权限。但是，这样做可能会中断服务的功能。

使用策略管理访问

您可以 AWS 通过创建策略并将其附加到 AWS 身份或资源来控制中的访问权限。策略定义了与身份或资源关联时的权限。 AWS 在委托人提出请求时评估这些政策。大多数策略都以 JSON 文档的 AWS 形式存储在中。有关 JSON 策略文档的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概述。

管理员使用策略，通过定义哪个主体可以在什么条件下对哪些资源执行哪些操作来指定谁有权访问什么。

默认情况下，用户和角色没有权限。IAM 管理员创建 IAM 策略并将其添加到角色中，然后用户可以担任这些角色。IAM 策略定义权限，与执行操作所用的方法无关。

基于身份的策略

基于身份的策略是您附加到身份（用户、组或角色）的 JSON 权限策略文档。这些策略控制身份可以执行什么操作、对哪些资源执行以及在什么条件下执行。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的使用客户管理型策略定义自定义 IAM 权限。

基于身份的策略可以是内联策略（直接嵌入到单个身份中）或托管策略（附加到多个身份的独立策略）。要了解如何在托管策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管策略与内联策略之间进行选择。

下表概述了 Device Farm AWS 托管的策略。

更改	描述	日期
AWSDeviceFarmFullAccess	提供对所有 AWS Device Farm 操作的完全访问权限。	2015 年 7 月 15 日
AWSServiceRoleForDeviceFarmTestGrid	允许 Device Farm 代表您访问 AWS 资源。	2021 年 5 月 20 日

其他策略类型

AWS 支持其他策略类型，这些策略类型可以设置更常见的策略类型授予的最大权限：

- 权限边界 – 设置基于身份的策略可以授予 IAM 实体的最大权限。有关更多信息，请参阅《 IAM 用户指南》中的 IAM 实体的权限边界。
- 服务控制策略 (SCPs)-在中指定组织或组织单位的最大权限 AWS Organizations。有关更多信息，请参阅《AWS Organizations 用户指南》中的服务控制策略。
- 资源控制策略 (RCPs)-设置账户中资源的最大可用权限。有关更多信息，请参阅《AWS Organizations 用户指南》中的资源控制策略 (RCPs)。
- 会话策略 – 在为角色或联合用户创建临时会话时，作为参数传递的高级策略。有关更多信息，请参阅《IAM 用户指南》中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解在涉及多种策略类型时如何 AWS 确定是否允许请求，请参阅 IAM 用户指南中的策略评估逻辑。

AWS Device Farm 基于身份的策略示例

默认情况下，IAM 用户和角色没有创建或修改 Device Farm 资源的权限。他们也无法使用 AWS 管理控制台 AWS CLI、或 AWS API 执行任务。IAM 管理员必须创建 IAM 策略，以便为用户和角色授予权限以对所需的指定资源执行特定的 API 操作。然后，管理员必须将这些策略附加到需要这些权限的 IAM 用户或组。

要了解如何使用这些示例 JSON 策略文档创建 IAM 基于身份的策略，请参阅《IAM 用户指南》中的 在 JSON 选项卡上创建策略。

主题

- 策略最佳实践
- 允许用户查看他们自己的权限
- 访问一个 Device Farm 桌面浏览器测试项目
- 根据标签查看 Device Farm 桌面浏览器测试项目

策略最佳实践

基于身份的策略确定某个人是否可以创建、访问或删除您账户中的 Device Farm 资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下指南和建议：

- 开始使用 AWS 托管策略并转向最低权限权限 — 要开始向用户和工作负载授予权限，请使用为许多常见用例授予权限的AWS 托管策略。它们在你的版本中可用 AWS 账户。我们建议您通过定义针对您的用例的 AWS 客户托管策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管策略或工作职能的AWS 托管策略。
- 应用最低权限：在使用 IAM 策略设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的 IAM 中的策略和权限。
- 使用 IAM 策略中的条件进一步限制访问权限：您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果服务操作是通过特定的方式使用的，则也可以使用条件来授予对服务操作的访问权限 AWS 服务，例如 CloudFormation。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。

- 使用 IAM Access Analyzer 验证您的 IAM 策略，以确保权限的安全性和功能性：IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM 策略语言（JSON）和 IAM 最佳实践。IAM Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的使用 IAM Access Analyzer 验证策略。
- 需要多重身份验证 (MFA)-如果 AWS 账户您的场景需要 IAM 用户或根用户，请启用 MFA 以提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的使用 MFA 保护 API 访问。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实践。

允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管式策略。此策略包括在控制台上或使用 AWS CLI 或 AWS API 以编程方式完成此操作的权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
```

```
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

访问一个 Device Farm 桌面浏览器测试项目

在此示例中，您想向 AWS 账户中的 IAM 用户授予访问您的 Device Farm 桌面版浏览器测试项目的权限。`arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655441111`您希望该账户能够查看与该项目相关的内容。

除了 `devicefarm:GetTestGridProject` 终端节点之外，该账户还必须具有 `devicefarm:ListTestGridSessions`、`devicefarm:GetTestGridSession`、`devicefarm:ListTe` 和 `devicefarm:ListTestGridSessionArtifacts` 终端节点。

如果您使用 CI 系统，则应为各个 CI 运行者提供不同的访问凭证。例如，CI 系统不太可能需要 `devicefarm:ScheduleRun` 或 `devicefarm:CreateUpload` 之外的权限。以下 IAM policy 概述了允许 CI 运行者执行以下操作的最小策略：创建上传并使用该上传来计划测试运行，从而启动新的 Device Farm 本机应用程序测试：

根据标签查看 Device Farm 桌面浏览器测试项目

您可以在基于身份的策略中使用条件，以便基于标签控制对 Device Farm 资源的访问。此示例说明了如何创建允许查看项目及其会话的策略。如果所请求资源的 `Owner` 标签与请求账户的用户名匹配，则授予权限。

您可以将该策略附加到您账户中的 IAM 用户。如果名为 `richard-roe` 的用户尝试查看 Device Farm 项目或会话，则该项目必须具有 `Owner=richard-roe` 或 `owner=richard-roe` 标签。否则，该用户将被拒绝访问。条件标签键 `Owner` 匹配 `Owner` 和 `owner`，因为条件键名称不区分大小写。有关更多信息，请参阅 IAM 用户指南 中的 IAM JSON 策略元素：条件。

对 AWS Device Farm 身份验证和访问进行故障排除

使用以下信息可帮助您诊断和修复在使用 Device Farm 和 IAM 时可能遇到的常见问题。

我无权在 Device Farm 中执行操作

如果您在中收到错误消息 AWS 管理控制台 ，提示您无权执行某项操作，则必须联系管理员寻求帮助。您的管理员是指为您提供用户名和密码的那个人。

当不具有 devicefarm:GetRun 权限的 IAM 用户 mateojackson 尝试使用控制台查看有关运行的详细信息时，就会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 devicefarm:GetRun on resource: arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-
e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111
```

在这种情况下，Mateo 请求管理员更新其策略，以允许他使用 devicefarm:GetRun 操作访问 arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111 资源上的 devicefarm:GetRun。

我无权执行 iam：PassRole

如果您收到一个错误，表明您无权执行 iam:PassRole 操作，则必须更新策略以允许您将角色传递给 Device Farm。

有些 AWS 服务 允许您将现有角色传递给该服务，而不是创建新的服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为 marymajor 的 IAM 用户尝试使用控制台在 Device Farm 中执行操作时，会发生以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 iam:PassRole 操作。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

我想要查看我的访问密钥

在创建 IAM 用户访问密钥后，您可以随时查看您的访问密钥 ID。但是，您无法再查看您的秘密访问密钥。如果您丢失了私有密钥，则必须创建一个新的访问密钥对。

访问密钥包含两部分：访问密钥 ID（例如 AKIAIOSFODNN7EXAMPLE）和秘密访问密钥（例如 wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY）。与用户名和密码一样，您必须同时使用访问密钥 ID 和秘密访问密钥对请求执行身份验证。像对用户名和密码一样，安全地管理访问密钥。

> **Important**
>
> 请不要向第三方提供访问密钥，即便是为了帮助找到您的规范用户 ID 也不行。通过这样做，您可以授予他人永久访问您的权限 AWS 账户。

当您创建访问密钥对时，系统会提示您将访问密钥 ID 和秘密访问密钥保存在一个安全位置。秘密访问密钥仅在您创建它时可用。如果丢失了您的秘密访问密钥，您必须为 IAM 用户添加新的访问密钥。您最多可拥有两个访问密钥。如果您已有两个密钥，则必须删除一个密钥对，然后再创建新的密钥。要查看说明，请参阅 IAM 用户指南中的管理访问密钥。

我是管理员并希望允许其他人访问 Device Farm

要让其他人访问 Device Farm，您必须向需要访问的人员或应用程序授予权限。如果使用 AWS IAM Identity Center 管理人员和应用程序，则可以向用户或组分配权限集来定义其访问权限级别。权限集会自动创建 IAM 策略并将其分配给与人员或应用程序关联的 IAM 角色。有关更多信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。

如果未使用 IAM Identity Center，则必须为需要访问的人员或应用程序创建 IAM 实体（用户或角色）。然后，您必须将策略附加到实体，以便在 Device Farm 中向其授予正确的权限。授予权限后，向用户或应用程序开发人员提供凭证。他们将使用这些凭证访问 AWS。要了解有关创建 IAM 用户、组、策略和权限的更多信息，请参阅《IAM 用户指南》中的 IAM 身份和 IAM 中的策略和权限。

我想允许 AWS 账户以外的用户访问我的 Device Farm 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以代入角色。对于支持基于资源的策略或访问控制列表 (ACLs) 的服务，您可以使用这些策略向人们授予访问您的资源的权限。

要了解更多信息，请参阅以下内容：

- 要了解 Device Farm 是否支持这些功能，请参阅 AWS Device Farm 如何与 IAM 配合使用。
- 要了解如何提供对您拥有的资源的访问权限 AWS 账户 ，请参阅 IAM 用户指南中的向您拥有 AWS 账户 的另一个 IAM 用户提供访问权限。
- 要了解如何向第三方提供对您的资源的访问权限 AWS 账户，请参阅 IAM 用户指南中的向第三方提供访问权限。 AWS 账户
- 要了解如何通过身份联合验证提供访问权限，请参阅《IAM 用户指南》中的为经过外部身份验证的用户（身份联合验证）提供访问权限。

- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的 IAM 中的跨账户资源访问。

的合规性验证 AWS Device Farm

作为多个 AWS Device Farm 合规性计划的一部分，第三方审计员将评估 AWS 的安全性和合规性。其中包括 SOC、PCI、FedRAMP、HIPAA 及其他合规性计划。AWS Device Farm 不在任何 AWS 合规性计划的范围内。

有关特定合规性计划范围内的 AWS 服务列表，请参阅按合规性计划提供的范围内 AWS 服务。有关常规信息，请参阅 AWS 合规性计划。

您可以使用AWS Artifact下载第三方审计报告。有关更多信息，请参阅在 AWS Artifact 中下载报告。

您使用 Device Farm 的合规性责任取决于您数据的敏感度、贵公司的合规性目标以及适用的法律法规。AWS 提供以下资源来帮助满足合规性：

- 安全性与合规性快速入门指南 - 这些部署指南讨论了架构注意事项，并提供了在AWS上部署基于安全性和合规性的基准环境的步骤。
- AWS 合规性资源：此业务手册和指南集合可能适用于您的行业和位置。
- 《AWS Config 开发人员指南》中的 Evaluating Resources with Rules – AWS Config 评估您的资源配置对内部实践、行业指南和法规的遵循情况。
- AWS Security Hub CSPM – 此AWS服务提供了AWS中安全状态的全面视图，可帮助您检查是否符合安全行业标准和最佳实践。

中的数据保护 AWS Device Farm

AWS 分担责任模型分适用于 AWS Device Farm （ Device Farm ）中的数据保护。如本模型所述 AWS ，负责保护运行所有内容的全球基础架构 AWS Cloud。您负责维护对托管在此基础结构上的内容的控制。您还负责您所使用的 AWS 服务 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS Security Blog 上的 AWS Shared Responsibility Model and GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 设置个人用户。这样，每个用户只获得履行其工作职责所需的权限。还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证（MFA）。
- 用于 SSL/TLS 与 AWS 资源通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用设置 API 和用户活动日志 AWS CloudTrail。有关使用 CloudTrail 跟踪捕获 AWS 活动的信息，请参阅《AWS CloudTrail 用户指南》中的使用跟 CloudTrail 踪。
- 使用 AWS 加密解决方案以及其中的所有默认安全控件 AWS 服务。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果您在 AWS 通过命令行界面或 API 进行访问时需要经过 FIPS 140-3 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅《美国联邦信息处理标准（FIPS）第 140-3 版》https://aws.amazon.com/compliance/fips/。

强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括您 AWS 服务 使用控制台、API 或与 Device Farm 或其他人合作时 AWS SDKs。 AWS CLI在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供 URL，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

传输中加密

Device Farm 端点仅支持签名的 HTTPS (SSL/TLS) requests except where otherwise noted. All content retrieved from or placed in Amazon S3 through upload URLs is encrypted using SSL/TLS. 有关 HTTPS 请求如何登录的更多信息 AWS，请参阅 AWS 通用参考中的签署 AWS API 请求。

对经过测试的应用程序进行的任何通信，以及在运行设备端测试的过程中安装的任何额外应用程序进行的任何通信，您需要负责加密和保护。

静态加密

Device Farm 的桌面浏览器测试功能支持对测试期间生成的构件进行静态加密。

Device Farm 的物理移动设备测试数据没有静态加密。

数据留存

Device Farm 中的数据将在有限的时间内保留。在保留期到期后，将从 Device Farm 的备份存储中删除数据。

内容类型	保留周期	元数据保留期（天）
上传的应用程序	30	30
上传的测试程序包	30	30
日志	400	400
视频录制和其他构件	400	400

对于任何您想要保留更长时间的内容，您需要负责对其进行存档。

数据管理

根据使用的功能，Device Farm 中的数据管理有所不同。本部分介绍在您使用 Device Farm 期间以及之后如何管理数据。

桌面浏览器测试

Selenium 会话期间使用的实例不会保存。在会话结束时，由浏览器交互生成的所有数据都会被丢弃。

此功能目前支持对测试期间生成的构件进行静态加密。

物理设备测试

以下各节提供有关使用 Device Farm 后清理或销毁设备所 AWS 采取的步骤的信息。

Device Farm 的物理移动设备测试数据没有静态加密。

公有设备队组

测试执行完成后，Device Farm 会对公有设备队组中的每台设备执行一系列清理任务，包括卸载您的应用程序。如果我们无法验证您的应用程序已卸载或任何其他清理步骤已成功完成，设备再次使用之前会恢复出厂设置。

> **Note**
>
> 在某些情况下，特别是如果在您的应用程序环境之外使用了 设备系统，则可能在不同会话之间保留数据。出于此原因，并且由于 Device Farm 会针对您使用每台设备期间发生的活动录制视

频并记录日志，因此建议您在自动化测试和远程访问会话期间不要输入敏感信息（如 Google 账户或 Apple ID）、个人信息和其他与安全相关的详细敏感信息。

私有设备

您的私有设备合同到期或终止后，设备将不再使用，并根据 AWS 销毁政策安全销毁。有关更多信息，请参阅 AWS Device Farm 中的私有设备。

密钥管理

目前，Device Farm 不针对数据加密（静态或传输中）提供任何外部密钥管理。

互联网络流量隐私

Device Farm 可以配置为仅供私有设备使用，这些设备使用 VPC 终端节点连接到 AWS中您的资源。访问与您的账户关联的任何非公有 AWS 基础设施（例如，没有公有 IP 地址的 Amazon EC2 实例）都必须使用 Amazon VPC 终端节点。无论 VPC 终端节点采用何种配置，Device Farm 都会将您的流量与 Device Farm 网络中其他用户的流量分隔开。

我们不能保证您在 AWS 网络之外的连接是安全的，因此您有责任保护您的应用程序建立的任何互联网连接。

AWS Device Farm 中的故障恢复能力

AWS全球基础架构围绕AWS区域和可用区构建。AWS区域提供多个在物理上独立且隔离的可用区，这些可用区通过延迟低、吞吐量高且冗余性高的网络连接在一起。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错能力和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅 AWS全球基础设施。

由于 Device Farm 仅在 `us-west-2` 区域中可用，因此我们强烈建议您执行备份和恢复过程。Device Farm 不应是任何已上传内容的唯一来源。

Device Farm 不保证公有设备的可用性。系统会根据各种因素（例如故障率和隔离状态）将这些设备加入和移出公有设备池。我们不建议您依赖于公有设备池中任何一台设备的可用性。

基础设施安全 AWS Device Farm

作为一项托管服务 AWS Device Farm ，受 AWS 全球网络安全的保护。有关 AWS 安全服务以及如何 AWS 保护基础设施的信息，请参阅AWS 云安全。要使用基础设施安全的最佳实践来设计您的 AWS 环境，请参阅 S AWS ecurity Pillar Well-Architected Fram ework 中的基础设施保护。

您可以使用 AWS 已发布的 API 调用通过网络访问 Device Farm。客户端必须支持以下内容：

- 传输层安全性协议（TLS）。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密（PFS）的密码套件，例如 DHE（临时 Diffie-Hellman）或 ECDHE（临时椭圆曲线 Diffie-Hellman）。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 主体关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

物理设备测试的基础设施安全性

在物理设备测试期间，设备会在物理上被分隔开。网络隔离可防止通过无线网络进行跨设备通信。

公有设备是共享的，Device Farm 会尽最大努力确保设备在整个过程中的安全。某些操作会导致公有设备被隔离，例如尝试获取设备上的完整管理员权限（这种做法被称为获取 Root 权限或越狱）。它们会自动从公共池中删除，并进行手动审查。

只有明确授权的 AWS 账户才能访问私有设备。Device Farm 将这些设备与其他设备进行物理隔离，并将它们保存在单独的网络上。

在私有管理的设备上，可以将测试配置为使用 Amazon VPC 终端节点来保护进出您 AWS 账户的连接。

桌面浏览器测试的基础设施安全性

当您使用桌面浏览器测试功能时，所有测试会话会彼此分隔开。如果没有外部的中间第三方，Selenium 实例就无法交叉通信。 AWS

所有到 Selenium WebDriver 控制器的流量都必须通过使用生成的 HTTPS 端点进行。`createTestGridUrl`

您需负责确保每个 Device Farm 测试实例都可以安全地访问它所测试的资源。默认情况下，Device Farm 的桌面浏览器测试实例可以访问公有互联网。当您将实例连接到 VPC 时，它的行为与任何其他 EC2 实例一样，对资源的访问权限由 VPC 的配置及其关联的联网组件决定。AWS 提供安全组和网络

访问控制列表 (ACLs)，以提高您的 VPC 的安全性。安全组控制资源的入站和出站流量，网络 ACLs 控制子网的入站和出站流量。安全组为大多数子网提供足够的访问控制。 ACLs 如果您想为自己的 VPC 增加一层安全保护，则可以使用网络。有关使用亚马逊时安全最佳实践的一般指南 VPCs，请参阅《亚马逊虚拟私有云用户指南》中的 VPC 安全最佳实践。

Device Farm 中的配置漏洞分析和管理

Device Farm 允许您运行未由供应商（例如操作系统供应商、硬件供应商或电话运营商）积极维护或修补的软件。Device Farm 尽最大努力维护最新的软件，但不保证物理设备上的任何特定版本的软件都是最新的，因为其设计允许将可能存在漏洞的软件投入使用。

例如，如果在搭载 Android 4.4.2 的设备上进行了测试，Device Farm 不保证该设备已针对安卓系统中名为的漏洞进行了修补。 StageFright设备的供应商（有时是运营商）负责为设备提供安全更新。我们不保证利用此漏洞的恶意应用程序一定会被我们的自动隔离措施捕获。

私人设备将根据您与之达成的协议进行维护 AWS。

Device Farm 尽最大努力防止客户应用程序执行诸如 root 或越狱等操作。Device Farm 会从公有池中删除隔离的设备，直到手动检查这些设备为止。

您需负责及时更新在测试中使用的任何库或软件版本（例如 Python wheel 和 Ruby gem）。Device Farm 建议您更新您的测试库。

这些资源有助于您将测试依赖关系保持最新：

- 有关如何保护 Ruby gems 的信息，请参阅 RubyGems 网站上的安全实践。
- 有关 Pipenv 使用并经 Python 打包机构认可的用于扫描依赖关系图中是否存在已知漏洞的安全包的信息，请参阅上的 “安全漏洞检测”。 GitHub
- 有关开放 Web 应用程序安全项目 (OWASP) Maven 依赖项检查器的信息，请参阅 OWASP 网站上的 OWAS DependencyCheck P。

切记，即使自动化系统认为不存在任何已知的安全问题，也并不意味着就真的不存在任何安全问题。在使用来自第三方的库或工具时务必要谨慎，并在可能或合理的情况下验证加密签名。

Device Farm 中的事件响应

Device Farm 会持续监控设备是否存在各种可能表明安全问题的行为。 AWS 如果得知其他客户可以访问客户数据（例如测试结果或写入公共设备的文件），则根据 AWS 服务中使用的标准事件警报和报告政策， AWS 联系受影响的客户。

Device Farm 中的日志记录和监控

该服务支持 AWS CloudTrail，这是一项记录您的 AWS 呼叫 AWS 账户 并将日志文件传输到 Amazon S3 存储桶的服务。通过使用收集的信息 CloudTrail，您可以确定成功向哪些请求发出 AWS 服务、请求是谁发来的、何时发出的，等等。要了解更多信息 CloudTrail，包括如何将其开启和查找日志文件，请参阅AWS CloudTrail 用户指南。

有关与 Device Farm CloudTrail 配合使用的信息，请参阅使用 AWS CloudTrail 记录 AWS Device Farm API 调用。

Device Farm 的安全最佳实践

Device Farm 提供了在您开发和实施自己的安全策略时需要考虑的大量安全特征。以下最佳实践是一般指导原则，并不代表完整安全解决方案。这些最佳实践可能不适合环境或不满足环境要求，请将其视为有用的考虑因素而不是惯例。

- 为您使用的任何持续集成（CI）系统授予 IAM 中的最少权限。考虑为每个 CI 系统测试使用临时凭证，这样即使 CI 系统遭到破坏，它也无法发出虚假请求。有关临时凭证的更多信息，请参阅《IAM 用户指南》。https://docs.aws.amazon.com//IAM/latest/UserGuide/id_credentials_temp_request.html#api_assumerole
- 在自定义测试环境中使用 adb 命令来清除应用程序创建的任何内容。有关自定义测试环境的更多信息，请参阅自定义测试环境。

AWS Device Farm 中的限制

主题

服务限制

- 您可以包括在测试运行中的设备数量没有限制。但是，在测试运行期间 Device Farm 将同时测试的设备数上限为 5。该数字可以根据要求增加，并由服务团队根据具体情况进行评估。
- 您可以安排的运行次数没有限制。请注意，这些运行最多只能排队 24 小时。
- 远程访问会话的持续时间有 150 分钟的硬性限制。
- 自动测试运行的持续时间有 150 分钟的硬性限制。
- 包括您账户中待处理的排队任务在内的最大数量为 250。这是一个软性限制。
- 您在测试运行中可以包含的设备数量没有限制。在任何给定时间，您可以在其中并行运行测试的设备（作业）数量等于您的账户级并发度。Device Farm 中用于计量的默认账户级别并发度为 5。
- 根据使用案例，计量并发限制可按需提升至特定阈值。非计量用途的默认账户级别并发度等于您为该平台订阅的插槽数量。

  有关默认计量并发限制或一般配额的更多信息，请参阅配额页面。
- 不使用自定义测试环境的自动化运行中最多只能有 250 个单独的测试用例。否则，可能会跳过运行。

文件限制

- 您可以上传的应用程序的最大文件大小为 4 GB。请注意，我们目前不接受适用于 Android 的 .aab 格式文件。

- 在测试运行期间，Device Farm 自动生成的视频的最大大小为 1 GB。任何超过此大小的视频，超出此部分的所有剩余视频内容都将被截断。客户仍可以使用自己的视频录制解决方案（如果有），并将其存储在 Device Farm 的托管存储外部。
- 在测试运行期间，Device Farm 自动生成的设备日志（Android 上的 logcat 或 iOS 上的 syslog）的最大大小为 1 GB。任何超过此大小的日志，超出此部分的所有剩余日志都将被截断。如果日志超过 1 GB，客户可以将这些日志存储在 Device Farm 的托管存储外部。
- Device Farm 的自定义环境模式客户构件的最大累积大小为 1 GB。如果您的构件超过此大小，则所有构件都将不可用。
- 如果测试运行期间生成的所有构件的累积大小超过 4 GB，则系统可能会丢弃某些构件（包括视频、设备日志和客户构件）。

API 限制

- Device Farm 按照令牌存储桶算法对 API 调用速率进行节流。例如，假设创建一个存放令牌的存储桶。一个令牌代表一个事务，一次 API 调用会用掉一个令牌。令牌以固定速率（例如每秒 10 个令牌）添加到存储桶中，存储桶的容量有上限（例如 100 个令牌）。请求或程序包到达时，必须从存储桶中获取令牌才能被处理。如果有足够的令牌，则系统会通过请求并移除令牌。如果没有足够的令牌，则请求要么延迟，要么被丢弃，具体取决于实现情况。

  在 Device Farm 中，算法的实现方式如下：

  - Burst API 请求数是服务在指定客户账户 ID 中针对指定 API 能够响应的最大请求数。换句话说，这就是存储桶的容量。只要存储桶中还有令牌，您就可以多次调用 API，并且每个请求会消耗一个令牌。
  - Transactions-per-second (TPS) 速率是可以执行您的 API 请求的最低速率。换句话说，这就是存储桶每秒重填令牌的速率。例如，如果某个 API 的突发数为 10，但 TPS 为 1，则可以立即调用它 10 次。但是，存储桶只能以每秒 1 个令牌的速度重新获得令牌，除非您停止调用 API 让存储桶重填令牌，否则系统将节流为每秒 1 个调用。

以下是 Device Farm 的费率 APIs：

- 对于 List an APIs d Get，Burst API 请求容量为50，Transactions-per-second (TPS) 速率为10。
- 对于所有其他请求 APIs，Burst API 请求容量为10，Transactions-per-second (TPS) 速率为1。

Appium 端点限制

以下限制适用于所有 Appium 端点会话。有关如何最好地处理限制的问题和指导，请提交支持案例。

- 每个 Appium 命令的执行时长限制为 4 分钟，之后该命令就会超时。
- 该端点接受的输入有效载荷大小不超过 20MB，并允许输出有效载荷大小不超过 20MB。任何输入或输出大小大于此值的请求都将收到 WebDriver 错误消息`'unsupported operation'`。
- 请求按照收到的顺序在设备上按顺序执行。因此，我们强烈建议按顺序发送命令，并在发送新命令之前等待每个命令的响应。也就是说，某些 Appium 服务器命令可以并行发送，特别是：
  - 获取状态
  - 获取会话
- 该端点目前不支持该WebDriver BiDi 协议。
- 该端点不支持 Appium 插件或除 XCUITest 和 UIAutomator2 驱动程序之外的驱动程序。
- 通过远程访问会话创建请求，最多可以将 3 个应用程序用作辅助应用程序。也就是说，在会话期间使用 `InstallToRemoteAccessSession`API 可以安装的应用程序数量没有限制。

自定义环境变量限制

以下限制适用于所有自定义环境变量。有关如何最好地处理限制的问题和指导，请提交支持案例。

- 在给定的 Device Farm 项目或运行中，最多可以配置 32 个变量。
- 变量名称的长度不能超过 256 个字符。
- 变量名受所施加的限制的约束`bash`。也就是说，它们必须仅包含字母数字和下划线字符，并且不能以数字开头。
- 以开头的`$DEVICEFARM_`变量名保留供内部服务使用。
- 变量值的长度不能超过 256 个字符。
- 环境变量不能用于在测试规范文件中配置测试主机计算选择。

AWS Device Farm 的工具和插件

本节包含有关使用 AWS Device Farm 工具和插件的链接和信息。您可以在 GitHub 上的 AWS 实验室中找到 Device Farm 插件。

如果您是 Android 开发人员，我们在 GitHub 上还提供一个适用于 Android 的 AWS Device Farm 应用程序示例。您可以使用应用程序和示例测试作为您自己的 Device Farm 测试脚本的参考。

主题

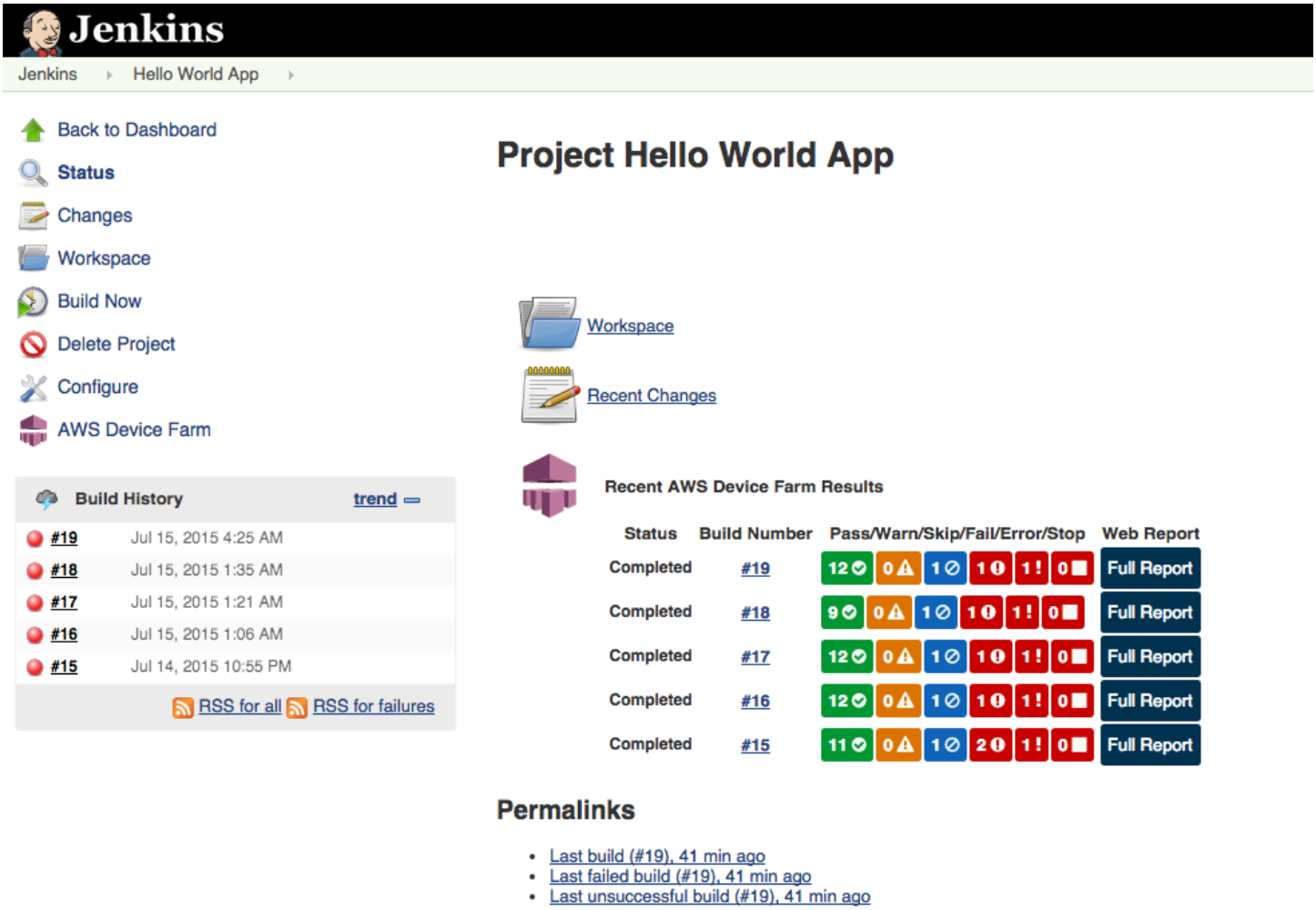
将 Device Farm 与 Jenkins CI 服务器集成

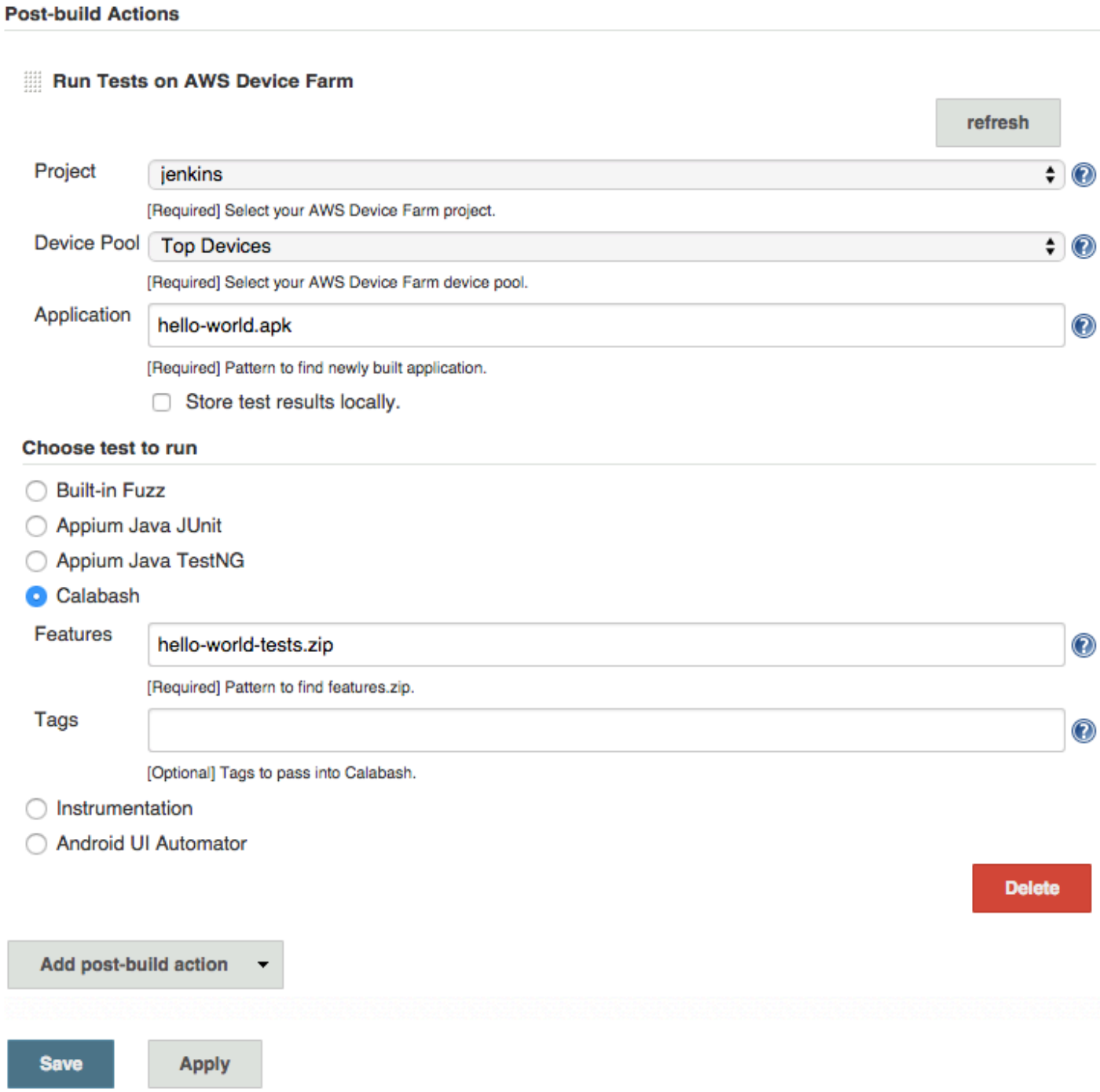
Jenkins CI 插件从您自己的 Jenkins 持续集成（CI）服务器提供 AWS Device Farm 功能。有关更多信息，请参阅 Jenkins (软件)。

> **Note**
>
> 要下载 Jenkins 插件，请转至GitHub并按照中的说明进行操作。步骤 1：为 AWS Device Farm 安装 Jenkins CI 插件

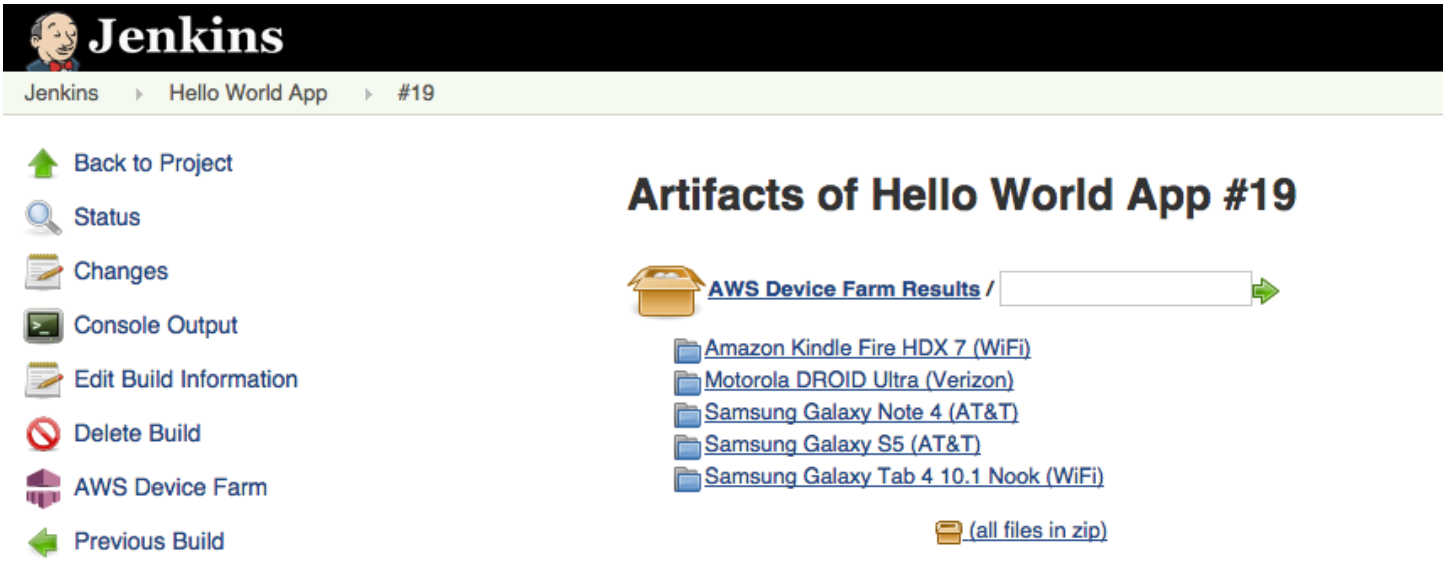
本节包含设置 Jenkins CI 插件并与 AWS Device Farm 一起使用的一系列过程。

下图显示 Jenkins CI 插件的功能。


Jenkins
Jenkins ▸ Hello World App ▸
Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
AWS Device Farm
Build History trend
#19 Jul 15, 2015 4:25 AM
#18 Jul 15, 2015 1:35 AM
#17 Jul 15, 2015 1:21 AM
#16 Jul 15, 2015 1:06 AM
#15 Jul 14, 2015 10:55 PM
RSS for all RSS for failures
Project Hello World App
Workspace
Recent Changes
Recent AWS Device Farm Results
Status Build Number Pass/Warn/Skip/Fail/Error/Stop Web Report
Completed #19 12 0 1 1 1 0 Full Report
Completed #18 9 0 1 1 1 0 Full Report
Completed #17 12 0 1 1 1 0 Full Report
Completed #16 12 0 1 1 1 0 Full Report
Completed #15 11 0 1 2 1 0 Full Report
Permalinks
Last build (#19), 41 min ago
Last failed build (#19), 41 min ago
Last unsuccessful build (#19), 41 min ago



此插件还可在本地下拉所有测试项目（日志、屏幕截图等）：



主题

依赖项

Jenkins CI 插件需要 AWS 移动 SDK 1.10.5 或更高版本。有关更多信息以及若要安装开发工具包，请参阅 AWS 移动开发工具包。

步骤 1：为 AWS Device Farm 安装 Jenkins CI 插件

有两个选项可用于为 AWS Device Farm 安装 Jenkins 持续集成（CI）插件。您可以从 Jenkins Web UI 的 Available Plugins (可用插件) 对话框中搜索插件，也可以下载 `hpi` 文件并从 Jenkins 中安装。

从 Jenkins UI 中安装

1. 通过依次选择 Manage Jenkins (管理 Jenkins)、Manage Plugins (管理插件) 和 Available (可用)，可在 Jenkins UI 中查找插件。

2. 搜索 aws-device-farm。
3. 安装 AWS Device Farm 插件。
4. 确保该插件归 `Jenkins` 用户所有。
5. 重启 Jenkins。

下载插件

1. 直接从 http://updates.jenkins-ci 下载`hpi`文件。 org/latest/aws-device-farm.hpi。
2. 确保该插件归 `Jenkins` 用户所有。
3. 使用以下选项之一安装插件：
   - 通过依次选择 Manage Jenkins (管理 Jenkins)、Manage Plugins (管理插件)、Advanced (高级) 和 Upload plugin (上传插件) 来上传插件。
   - 将 `hpi` 文件置于 Jenkins 插件目录 (通常为 `/var/lib/jenkins/plugins`) 中。
4. 重启 Jenkins。

第 2 步：为适用于 AWS De AWS Identity and Access Management vice Farm 的 Jenkins CI 插件创建用户

我们建议您不要使用 AWS 根账户来访问 Device Farm。相反，请在您的 AWS 账户中创建一个新 AWS Identity and Access Management (IAM) 用户（或使用现有的 IAM 用户），然后使用该 IAM 用户访问 Device Farm。

要创建新的 IAM 用户，请参阅创建 IAM 用户（AWS 管理控制台）。确保为每个用户生成访问密钥并下载或保存用户安全凭证。您稍后将需要凭证。

为 IAM 用户提供访问 Device Farm 的权限。

要为 IAM 用户提供访问 Device Farm 的权限，请在 IAM 中创建新的访问策略，然后将此访问策略分配给 IAM 用户，如下所示。

> **Note**
>
> 用于完成以下步骤的 AWS 根账户或 IAM 用户必须有权创建以下 IAM 策略并将其附加到 IAM 用户。有关更多信息，请参阅使用策略

在 IAM 中创建访问策略

1. 使用 https://console.aws.amazon.com/iam/ 打开 IAM 控制台。
2. 选择策略。
3. 选择创建策略。（如果 Get Started 按钮出现，选择此按钮，然后选择 Create Policy。）
4. 在 Create Your Own Policy 旁，选择 Select。
5. 对于策略名称，键入策略的名称（例如 **AWSDeviceFarmAccessPolicy**）。
6. 对于说明，键入说明以帮助您将此 IAM 用户与您的 Jenkins 项目相关联。
7. 为策略文档键入以下声明：

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DeviceFarmAll",
            "Effect": "Allow",
            "Action": [ "devicefarm:*" ],
            "Resource": [ "*" ]
        }
    ]
}
```

8. 选择创建策略。

将访问策略分配给 IAM 用户

1. 使用 https://console.aws.amazon.com/iam/ 打开 IAM 控制台。
2. 选择用户。
3. 选择将要向其分配访问策略的 IAM 用户。
4. 在权限区域中，为管理的策略选择附加策略。
5. 选中刚创建的策略（例如 AWSDeviceFarmAccessPolicy）。
6. 选择附加策略。

步骤 3：在 AWS Device Farm 中首次配置 Jenkins CI 插件

首次运行 Jenkins 服务器时，您将需要按如下所示配置系统。

> **Note**
>
> 如果您要使用设备槽，请将其启用，设备槽功能默认情况下处于禁用状态。

1. 登录到您的 Jenkins Web 用户界面。
2. 在屏幕左侧，选择 Manage Jenkins (管理 Jenkins)。
3. 选择 Configure System (配置系统)。
4. 向下滚动到 AWS Device Farm 标题。
5. 从 为您的 Jenkins CI 插件创建 IAM 用户 复制您的安全凭证，并将您的访问密钥 ID 和私有访问密钥粘贴到其各自的框中。
6. 选择保存。

步骤 4：在 Jenkins 任务中使用插件

安装 Jenkins 插件后，请按照以下说明在 Jenkins 任务中使用该插件。

1. 登录到您的 Jenkins Web UI。
2. 单击要编辑的任务。
3. 在屏幕左侧，选择 Configure (配置)。
4. 向下滚动至 Post-build Actions (构建后操作) 标题。
5. 单击添加构建后操作并选择在 AWS Device Farm 上运行测试。
6. 选择要使用的项目。
7. 选择要使用的设备池。
8. 选择是否想将测试项目 (如日志和屏幕截图) 存档在本地。
9. 在 Application (应用程序) 中，填写已编译的应用程序的路径。
10. 选择您想运行的测试并填写所有必填字段。
11. 选择保存。

将 Device Farm 与 Gradle 构建系统集成

利用 Device Farm Gradle 插件，AWS Device Farm 可与 Android Studio 中的 Gradle 构建系统集成。有关更多信息，请参阅 Gradle。

> **Note**
>
> 要下载 Gradle 插件，请转至GitHub并按照中的说明进行操作。构建 Device Farm Gradle 插件

Device Farm Gradle 插件可在您的 Android Studio 环境中提供 Device Farm 功能。您可以在由 Device Farm 托管的真实 Android 手机和平板电脑上开始测试。

本节包含设置和使用 Device Farm Gradle 插件的一系列过程。

主题

依赖项

运行时

- Device Farm Gradle 插件需要 AWS 移动 SDK 1.10.15 或更高版本。有关更多信息以及若要安装开发工具包，请参阅 AWS 移动开发工具包。
- Android tools builder test api 0.5.2
- Apache Commons Lang3 3.3.4

对于单元测试

- Testng 6.8.8
- Jmockit 1.19

- Android gradle tools 1.3.0

步骤 1：构建 AWS Device Farm Gradle 插件

利用此插件，AWS Device Farm 可与 Android Studio 中的 Gradle 构建系统集成。有关更多信息，请参阅 Gradle。

> **Note**
>
> 构建此插件是可选的。通过 Maven Central 发布了此插件。如果您希望允许 Gradle 直接下载此插件，请跳过此步骤并跳转到 步骤 2：设置 AWS Device Farm Gradle 插件。

构建此插件

1. 前往GitHub并克隆存储库。
2. 使用 `gradle install` 构建此插件。

   此插件将安装到您的本地 maven 存储库。

下一步: 步骤 2：设置 AWS Device Farm Gradle 插件

步骤 2：设置 AWS Device Farm Gradle 插件

请使用此处的过程克隆存储库并安装插件：构建 Device Farm Gradle 插件 (如果您尚未执行此操作)。

配置 AWS Device Farm Gradle 插件

1. 向 `build.gradle` 中的依赖项列表添加插件项目。

```
buildscript {

    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.0'
        classpath 'com.amazonaws:aws-devicefarm-gradle-plugin:1.0'
```

```
        }
    }
```

2. 在 build.gradle 文件中配置插件。以下测试特定的配置应作为您的指南：

```
apply plugin: 'devicefarm'

devicefarm {

    // Required. The project must already exist. You can create a project in the
 AWS Device Farm console.
    projectName "My Project" // required: Must already exist.

    // Optional. Defaults to "Top Devices"
    // devicePool "My Device Pool Name"

    // Optional. Default is 150 minutes
    // executionTimeoutMinutes 150

    // Optional. Set to "off" if you want to disable device video recording during
 a run. Default is "on"
    // videoRecording "on"

    // Optional. Set to "off" if you want to disable device performance monitoring
 during a run. Default is "on"
    // performanceMonitoring "on"

    // Optional. Add this if you have a subscription and want to use your unmetered
 slots
    // useUnmeteredDevices()

    // Required. You must specify either accessKey and secretKey OR roleArn.
 roleArn takes precedence.
    authentication {
        accessKey "AKIAIOSFODNN7EXAMPLE"
        secretKey "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"

        // OR

        roleArn "arn:aws:iam::111122223333:role/DeviceFarmRole"
    }

    // Optionally, you can
    // - enable or disable Wi-Fi, Bluetooth, GPS, NFC radios
```

```
    // - set the GPS coordinates
    // - specify files and applications that must be on the device when your test
 runs
    devicestate {
        // Extra files to include on the device.
        // extraDataZipFile file("path/to/zip")

        // Other applications that must be installed in addition to yours.
        // auxiliaryApps files(file("path/to/app"), file("path/to/app2"))

        // By default, Wi-Fi, Bluetooth, GPS, and NFC are turned on.
        // wifi "off"
        // bluetooth "off"
        // gps "off"
        // nfc "off"

        // You can specify GPS location. By default, this location is 47.6204,
 -122.3491
        // latitude 44.97005
        // longitude -93.28872
    }

    // By default, the Instrumentation test is used.
    // If you want to use a different test type, configure it here.
    // You can set only one test type (for example, Calabash, Fuzz, and so on)

    // Fuzz
    // fuzz { }

    // Calabash
    // calabash { tests file("path-to-features.zip") }

}
```

3. 使用以下任务运行 Device Farm 测试：`gradle devicefarmUpload`。

   构建输出将输出一个指向 Device Farm 控制台的链接，您可在其中监控测试执行情况。

下一步: 在 Device Farm Gradle 插件中生成 IAM 用户

步骤 3：在 Device Farm Gradle 插件中生成 IAM 用户

AWS Identity and Access Management (IAM) 可帮助您管理使用 AWS 资源的权限和策略。本主题演示如何生成具有 AWS Device Farm 资源访问权限的 IAM 用户。

如果您还没有完成步骤 1 和 2，请先完成这两个步骤，然后再生成 IAM 用户。

我们建议您不要使用 AWS 根账户来访问 Device Farm。而应在您的 AWS 账户中创建一个新 IAM 用户（或使用现有的 IAM 用户），然后使用该 IAM 用户访问 Device Farm。

> **Note**
>
> 用于完成以下步骤的 AWS 根账户或 IAM 用户必须有权创建以下 IAM 策略并将其附加到 IAM 用户。有关更多信息，请参阅使用策略。

使用适当访问策略在 IAM 中创建新用户

1. 使用 https://console.aws.amazon.com/iam/ 打开 IAM 控制台。
2. 选择用户。
3. 选择创建新用户。
4. 请输入您选择的用户名称。

   例如 **`GradleUser`**。
5. 选择创建。
6. 选择下载凭证，并将这些凭证保存在您之后可以轻松检索它们的位置。
7. 选择关闭。
8. 在列表中选择用户名称。
9. 在权限下，通过单击右侧的向下箭头来展开内联策略标题。
10. 选择单击此处，其中显示没有任何内联策略可显示。要创建一个，请单击此处。
11. 在设置权限屏幕上，选择自定义策略。
12. 选定选择。
13. 为您的策略提供名称，例如 **`AWSDeviceFarmGradlePolicy`**。
14. 将以下策略粘贴到策略文档中。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DeviceFarmAll",
            "Effect": "Allow",
            "Action": [ "devicefarm:*" ],
            "Resource": [ "*" ]
        }
    ]
}
```

15. 选择应用策略。

下一步：配置测试类型。

有关更多信息，请参阅创建 IAM 用户 (AWS 管理控制台) 或 设置。

步骤 4：配置测试类型

默认情况下，AWS Device Farm Gradle 插件运行 适用于 Android 的 Instrumentation 与 AWS Device Farm 测试。如果要运行自己的测试或指定其他参数，可以选择配置测试类型。本主题提供有关每个可用测试类型的信息，以及您需要在 Android Studio 中执行哪些操作才能将其配置为可供使用。有关 Device Farm 中可用测试类型的更多信息，请参阅 AWS Device Farm 中的测试框架和内置测试。

如果您还没有完成步骤 1 到 3，请先完成这些步骤，然后再配置测试类型。

> **Note**
>
> 如果您要使用设备槽，请将其启用，设备槽功能默认情况下处于禁用状态。

Appium

Device Farm 支持 Appium Java 和 Android JUnit 版 TestNG。

- Appium（在 Java () JUnit 下）

- Appium [在 Java（TestNG）下]

您可以选择 `useTestNG()` 或 `useJUnit()`。`JUnit` 是默认值，不需要显式指定。

```
    appium {
        tests file("path to zip file") // required
        useTestNG() // or useJUnit()
    }
```

内置：模糊

Device Farm 提供内置模糊测试类型，该测试类型将用户界面事件随机发送至设备，然后报告结果。

```
    fuzz {

       eventThrottle 50 // optional default
       eventCount 6000  // optional default
       randomizerSeed 1234 // optional default blank

     }
```

有关更多信息，请参阅 运行 Device Farm 的内置模糊测试（Android 和 iOS）。

Instrumentation

Device Farm 支持安卓版仪器（JUnit、Espresso、Robotium 或任何基于仪器的测试）。有关更多信息，请参阅 适用于 Android 的 Instrumentation 与 AWS Device Farm。

在 Gradle 中运行 Instrumentation 时， 使用从 androidTest 目录生成的 `.apk` 文件作为测试源。

```
    instrumentation {

        filter "test filter per developer docs" // optional

    }
```

AWS Device Farm 文档历史记录

下表描述了自本指南上一次发布以来对文档所做的重要改动。

更改	描述	更改日期
Appium 端点支持	Device Farm 现在为远程设备测试提供了一个完全托管的 Appium 端点，可实现快速的测试开发和调试。这补充了现有的服务器端执行方法，即直接在 Device Farm 上上传和运行测试。虽然服务器端执行非常适合 CI/CD 管道和大规模测试，但新的本地 Appium 端点允许在真实设备上更快地迭代和开发测试。	2025年11月17日
iOS 测试主机改进	Device Farm 现在支持 iOS 测试环境的更新体验，从而使安卓和 iOS 测试之间的设置保持一致。请参阅 用于自定义测试环境的主机，了解更多信息。 此外，与已停用的 Android 测试主机相关的信息已被删除。我们鼓励安卓用户使用亚马逊 Linux 2 测试主机。	2025 年 10 月 31 日
AL2 支持	Device Farm 现在支持安卓版的 AL2 测试环境。了解有关 AL2 的更多信息。	2023 年 11 月 6 日
从标准测试环境迁移到自定义测试环境	更新了迁移指南，以记录在 2023 年 12 月弃用标准模式测试的情况。	2023 年 9 月 3 日
VPC ENI 支持	现在，借助 Device Farm，私有设备可以使用 VPC-ENI 连接特征，以帮助客户安全地连接到托管在 AWS、本地软件或其他云提供商上的私有端点。了解有关 VPC-ENI 的更多信息。	2023 年 5 月 15 日
Polaris UI 更新	Device Farm 控制台现在支持 Polaris 框架。	2021 年 7 月 28 日
Python 3 支持	现在，Device Farm 在自定义模式测试中支持 Python 3。了解有关在测试程序包中使用 Python 3 的更多信息： • Appium (Python)	2020 年 4 月 20 日

更改	描述	更改日期
	• Appium (Python)	
新的安全信息和有关标记 AWS 资源的信息。	为了更简单、更全面地保护 AWS 服务，新增了有关安全的章节。要了解更多信息，请参阅安全性 AWS Device Farm 新增了一个介绍 Device Farm 中的添加标签功能的小节。要了解有关标记的更多信息，请参阅在 Device Farm 中标记。	2020 年 3 月 27 日
删除直接设备访问权限。	直接设备访问（专用设备上的远程调试）不再适用于一般使用情况。如需了解直接设备访问将来的可用性，请联系我们。	2019 年 9 月 9 日
更新 Gradle 插件配置	Gradle 插件配置经过了修订，现在包含一个可自定义的 Gradle 配置版本，其中注释掉了可选参数。了解有关 设置 Device Farm Gradle 插件 的更多信息。	2019 年 8 月 16 日
对测试运行的新要求 XCTest	对于使用该 XCTest 框架的测试运行，Device Farm 现在需要一个专为测试而构建的应用包。了解有关 the section called “XCTest” 的更多信息。	2019 年 2 月 4 日
在自定义环境中支持 Appium Node.js 和 Appium Ruby 测试类型	您现在可以在 Appium Node.js 和 Appium Ruby 自定义测试环境中运行测试。了解有关 AWS Device Farm 中的测试框架和内置测试 的更多信息。	2019 年 1 月 10 日
标准环境和自定义环境对 Appium 服务器版本 1.7.2 的支持。自定义测试环境对使用自定义测试规范 YAML 文件的版本 1.8.1 的支持。	现在，您可以在标准测试环境和自定义测试环境中使用 Appium 服务器版本 1.72、1.71 和 1.6.5 运行测试。在自定义测试环境中，您还可以通过使用自定义测试规范 YAML 文件的版本 1.8.1 和 1.8.0 运行测试。了解有关 AWS Device Farm 中的测试框架和内置测试 的更多信息。	2018 年 10 月 2 日

更改	描述	更改日期
自定义测试环境	使用自定义测试环境，您可以确保测试像在本地环境中一样运行。Device Farm 现在支持实时日志和视频流，因此您可以获得有关在自定义测试环境中运行的测试的即时反馈。了解有关 AWS Device Farm 中的自定义测试环境 的更多信息。	2018 年 8 月 16 日
支持使用 Device Farm 作为 AWS CodePipeline 测试提供商	现在，您可以在中配置管道， AWS CodePipeline 以便在发布过程中使用 AWS Device Farm 运行作为测试操作。CodePipeline 使您能够快速将存储库链接到构建和测试阶段，从而实现根据您的需求定制的持续集成系统。了解有关 在 CodePipeline 测试阶段集成 AWS Device Farm 的更多信息。	2018 年 7 月 19 日
支持私有设备	现在，您可以使用私有设备安排测试运行，并启动远程访问会话。您可以管理这些设备的配置文件和设置，创建 Amazon VPC 终端节点来测试专用应用程序，还能创建远程调试会话。了解有关 AWS Device Farm 中的私有设备 的更多信息。	2018 年 5 月 2 日
对 Appium 1.6.3 的支持	您现在可以为 Appium 自定义测试设置 Appium 版本。	2017 年 3 月 21 日
设置测试运行的执行超时值	您可以为测试运行或为项目中的所有测试设置执行超时值。了解有关 在 AWS Device Farm 中设置测试运行的执行超时 的更多信息。	2017 年 2 月 9 日
网络塑造	您现在可以模拟测试运行的网络连接和条件。了解有关 为您的 AWS Device Farm 运行模拟网络连接和条件 的更多信息。	2016 年 12 月 8 日
新故障排除部分	您现在可以使用一套旨在解决 Device Farm 控制台中可能遇到的错误消息的程序，对测试程序包上传进行故障排除。了解有关 Device Farm 错误故障排除 的更多信息。	2016 年 8 月 10 日
远程访问会话	您现在可以远程访问控制台中的单台设备并与之交互。了解有关 远程访问 的更多信息。	2016 年 4 月 19 日

更改	描述	更改日期
自助式设备槽	现在，您可以使用 AWS 管理控制台 AWS Command Line Interface、或 API 购买设备插槽。了解有关如何在 Device Farm 中购买设备槽的更多信息。	2016 年 3 月 22 日
如何停止测试运行	现在，您可以使用 AWS 管理控制台 AWS Command Line Interface、或 API 停止测试运行。了解有关如何在 AWS Device Farm 中停止运行的更多信息。	2016 年 3 月 22 日
新的 XCTest UI 测试类型	现在，您可以在 iOS 应用程序上运行 XCTest UI 自定义测试。了解有关 将 iOS XCTest 用户界面与 Device Farm 集成 测试类型的更多信息。	2016 年 3 月 8 日
新 Appium Python 测试类型	您现在可以在 Android、iOS 和 Web 应用程序上运行 Appium Python 自定义测试。了解有关 AWS Device Farm 中的测试框架和内置测试 的更多信息。	2016 年 1 月 19 日
Web 应用程序测试类型	现在，你可以在 Web 应用程序上运行 Appium Java JUnit 和 Testng 自定义测试。了解有关 AWS Device Farm 中的 Web 应用程序测试 的更多信息。	2015 年 11 月 19 日
AWS Device Farm Gradle 插件	了解有关如何安装和使用 Device Farm Gradle 插件 的更多信息。	2015 年 9 月 28 日
新 Android 内置测试：浏览器	管理器测试通过分析每个屏幕 (就像它是最终用户一样) 对您的应用程序进行爬网，并在它浏览时拍摄屏幕截图。	2015 年 9 月 16 日
增加了 iOS 支持	要详细了解如何测试 iOS 设备和运行 iOS 测试（包括 XCTest），请参阅AWS Device Farm 中的测试框架和内置测试。	2015 年 8 月 4 日
第一个公开发布版	这是《AWS Device Farm 开发人员指南》的第一个公开发布版。	2015 年 7 月 13 日

AWS 术语表

有关最新的 AWS 术语，请参阅 AWS 词汇表 参考中的 AWS 词汇表。

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。