开发工具包 v2 开发人员指南

适用于 JavaScript 的 Amazon SDK

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

适用于 JavaScript 的 Amazon SDK: 开发工具包 v2 开发人员指南

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Amazon Web Services 文档中描述的 Amazon Web Services 服务或功能可能因区域而异。要查看适用于中国区域的差异，请参阅 中国的 Amazon Web Services 服务入门 (PDF)。

Table of Contents

- ix
- 适用于 JavaScript 的 Amazon SDK 是什么？ 1
  - SDK 主要版本的维护和支持 1
  - 将 SDK 与 Node.js 配合使用 2
  - 将 SDK 与 Amazon Amplify 配合使用 2
  - 将开发工具包与 Web 浏览器结合使用 2
    - 常见使用案例 2
    - 关于示例 3
- 开始使用 4
  - 浏览器脚本入门 4
    - 情景 4
    - 步骤 1：创建一个 Amazon Cognito 身份池 5
    - 步骤 2：将策略添加到创建的 IAM 角色 6
    - 步骤 3：创建 HTML 页面 6
    - 步骤 4：写入浏览器脚本 7
    - 步骤 5：运行示例 9
    - 完整示例 9
    - 可能的增强功能 10
  - Node.js 入门 11
    - 情景 11
    - 先决条件任务 11
    - 步骤 1：安装 SDK 和依赖项 12
    - 步骤 2：配置凭证 12
    - 步骤 3：为项目创建包 JSON 13
    - 步骤 4：编写 Node.js 代码 14
    - 步骤 5：运行示例 15
- 设置 SDK for JavaScript 16
  - 先决条件 16
    - 设置 Amazon Node.js 环境 16
    - 支持的 Web 浏览器 17
  - 安装 SDK 18
    - 使用 Bower 安装 19
  - 加载 SDK 19
  - 从版本 1 升级 20

自动转换输入/输出中的 Base64 和时间戳类型 20
将 response.data.RequestId 移动到 response.requestId 21
公开的包装元素 21
删除的客户端属性 26
配置 SDK for JavaScript 27
使用全局配置对象 27
设置全局配置 28
设置每个服务的配置 29
不可变的配置数据 30
设置 Amazon 区域 30
在客户端类构造函数中 30
使用全局配置对象 30
使用环境变量 31
使用共享配置文件 31
区域的优先设置顺序 31
指定自定义终端节点 32
终端节点字符串格式 32
ap-northeast-3 区域的终端节点 32
MediaConvert 的端点 32
使用 Amazon 进行 SDK 身份验证 33
开始 Amazon 访问门户会话 34
更多身份验证信息 34
设置凭证 35
凭证的最佳实践 35
在 Node.js 中设置凭证 36
在 Web 浏览器中设置凭证 40
锁定 API 版本 49
获取 API 版本 50
Node.js 注意事项 50
使用内置 Node.js 模块 50
使用 NPM 程序包 51
在 Node.js 中配置 maxSockets 51
在 Node.js 中重复使用具有保持连接功能的连接 52
配置 Node.js 的代理 53
在 Node.js 中注册证书包 54
浏览器脚本注意事项 54

为浏览器构建 SDK 54
跨源资源共享 (CORS) 57
使用 Webpack 捆绑 61
安装 Webpack 61
配置 Webpack 61
运行 Webpack 62
使用 Webpack 捆绑 63
导入单独的服务 64
适用于 Node.js 的捆绑 64
使用服务 66
创建和调用服务对象 67
要求单个服务 67
创建服务对象 68
锁定服务对象的 API 版本 69
指定服务对象参数 69
记录适用于 JavaScript 的 Amazon SDK调用 70
使用第三方日志记录程序 70
异步调用服务 71
管理异步调用 71
使用回调函数 72
使用请求对象事件侦听器 73
使用异步/等待 78
使用 Promise 79
使用响应对象 81
访问在响应对象中返回的数据 81
分页查看返回的数据 82
访问来自响应对象的错误信息 83
访问原始请求对象 83
使用 JSON 83
将 JSON 作为服务对象参数 84
以 JSON 格式返回数据 85
重试 85
基于指数回退的重试行为 86
SDK for JavaScript 代码示例 89
Amazon CloudWatch 示例 89
在 Amazon CloudWatch 中创建警报 90

在 Amazon CloudWatch 中使用警报操作 ... 94
从 Amazon CloudWatch 获取指标 ... 98
将事件发送到 Amazon CloudWatch Events ... 101
在 Amazon CloudWatch Logs 中使用订阅筛选条件 ... 106
Amazon DynamoDB 示例 ... 110
在 DynamoDB 中创建和使用表 ... 111
在 DynamoDB 中读取和写入单个项目 ... 115
在 DynamoDB 中批量读取和写入项目 ... 119
查询和扫描 DynamoDB 表 ... 122
使用 DynamoDB 文档客户端 ... 125
Amazon EC2 示例 ... 131
创建 Amazon EC2 实例 ... 132
管理 Amazon EC2 实例 ... 134
使用 &EC2; 密钥对 ... 140
将区域和可用区用于 Amazon EC2 ... 143
使用 Amazon EC2 中的安全组 ... 145
在 Amazon EC2 中使用弹性 IP 地址 ... 150
MediaConvert 示例 ... 154
创建和管理任务 ... 154
使用作业模板 ... 161
Amazon IAM 示例 ... 169
管理 IAM 用户 ... 170
使用 IAM 策略 ... 175
管理 IAM 访问密钥 ... 180
使用 IAM 服务器证书 ... 185
管理 IAM 账户别名 ... 189
Amazon Kinesis 示例 ... 192
使用 Amazon Kinesis 捕获网页滚动进度 ... 193
Amazon S3 示例 ... 199
Amazon S3 浏览器示例 ... 200
Amazon S3 Node.js 示例 ... 227
Amazon SES 示例 ... 247
管理身份 ... 247
使用电子邮件模板 ... 252
使用 Amazon SES 发送电子邮件 ... 258
使用 IP 地址筛选条件 ... 263

- 使用接收规则 267
- Amazon SNS 示例 272
  - 管理主题 273
  - 将消息发布到主题 278
  - 管理订阅 280
  - 发送短信 286
- Amazon SQS 示例 292
  - 在 &SQS; 中使用队列 293
  - 在 Amazon SQS 中发送和接收消息 297
  - 在 &SQS; 中管理可见性超时 300
  - 在 Amazon SQS 中启用长轮询 302
  - 在 &SQS; 中使用死信队列 306

教程 309
- 教程：在 Amazon EC2 实例上设置 Node.js 309
  - 先决条件 309
  - 过程 309
  - 创建 Amazon 系统映像 311
  - 相关资源 311

API 参考和更改日志 312
- GitHub 上的开发工具包更改日志 312

迁移到 v3 313

安全性 314
- 数据保护 314
- 身份和访问管理 315
  - 受众 315
  - 使用身份进行身份验证 316
  - 使用策略管理访问 317
  - Amazon Web Services 服务如何与 IAM 协同工作 318
  - 对 Amazon 身份和访问进行问题排查 319
- 合规性验证 320
- 韧性 321
- 基础设施安全性 321
- 强制实施最低版本 TLS 322
  - 在 Node.js 中验证并强制执行 TLS 322
  - 在浏览器脚本中验证并强制执行 TLS 325

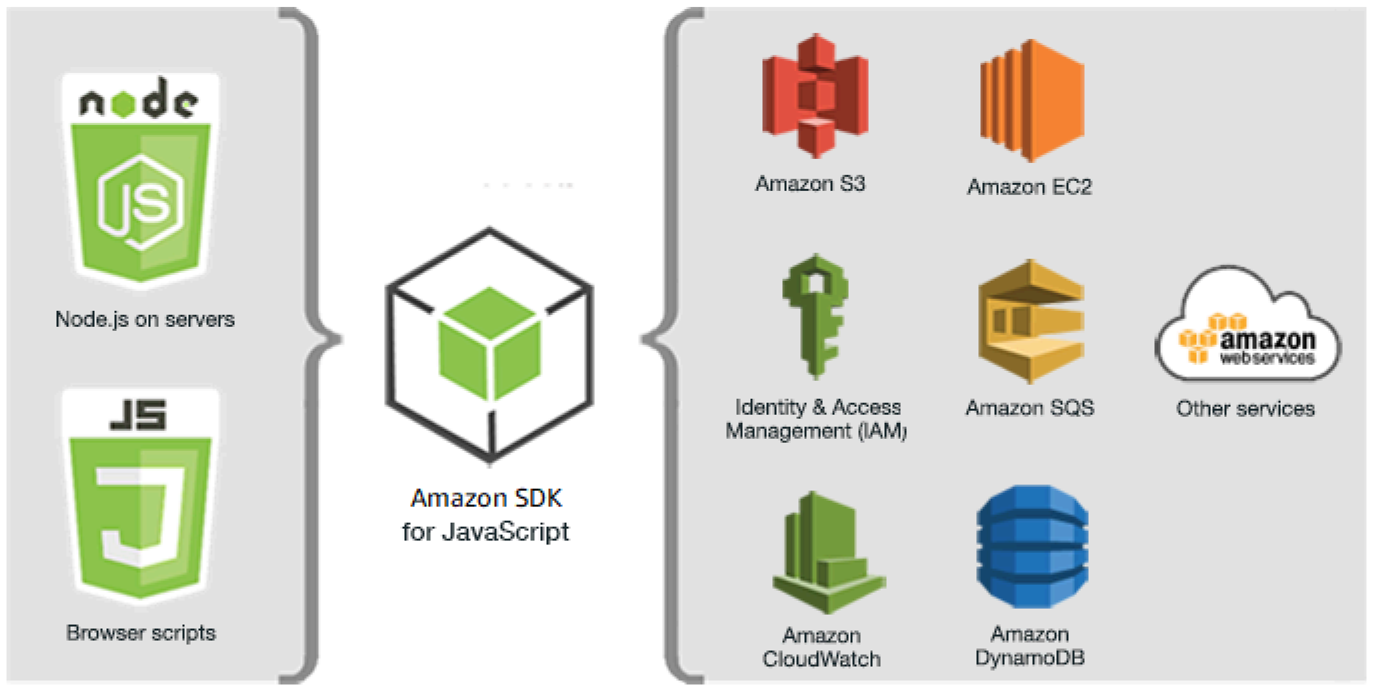
其他资源 327

Amazon SDKs and Tools Reference Guide ... 327
JavaScript 开发工具包论坛 ... 327
GitHub 上的 JavaScript 开发工具包和开发人员指南 ... 327
Gitter 上的 JavaScript 开发工具包 ... 327
文档历史记录 ... 328
文档历史记录 ... 328
早期更新 ... 329

适用于 JavaScript 的 Amazon SDK v2 已终止支持。建议您迁移到 适用于 JavaScript 的 Amazon SDK v3。有关更多详情和如何迁移的信息，请参阅本公告。

适用于 JavaScript 的 Amazon SDK 是什么？

适用于 JavaScript 的 Amazon SDK 为 Amazon 服务提供了 JavaScript API。您可以使用 JavaScript API 构建适用于 Node.js 或浏览器的库或应用程序。



开发工具包中的服务并不是全部直接可用的。要确定 适用于 JavaScript 的 Amazon SDK 当前支持哪些服务，请参阅 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md。有关 GitHub 上的 SDK for JavaScript 的信息，请参阅 其他资源。

SDK 主要版本的维护和支持

有关维护和支持 SDK 主要版本及其基础依赖关系的信息，请参阅 Amazon SDK 和工具参考指南中的以下内容：

- Amazon SDK 和工具维护策略
- Amazon SDK 和工具版本支持矩阵

将 SDK 与 Node.js 配合使用

Node.js 是一个用于运行服务器端 JavaScript 应用程序的跨平台运行时系统。您可以在 Amazon EC2 实例上设置 Node.js 以在服务器上运行。您还可以使用 Node.js 来编写按需 Amazon Lambda 函数。

使用 SDK for Node.js 与在 Web 浏览器中将其用于 JavaScript 的方式不同。区别在于您加载 SDK 以及获取访问特定 Web 服务所需凭证的方法。如果在 Node.js 与浏览器之间使用特定 API 存在差别时，将说明这些差别。

将 SDK 与 Amazon Amplify 配合使用

对于基于浏览器的 Web、移动和混合应用程序，您还可以使用 GitHub 上的 Amazon Amplify 库，该库对 SDK for JavaScript 进行了扩展，提供声明性接口。

> **Note**
>
> Amazon Amplify 等框架可能无法提供与 SDK for JavaScript 相同的浏览器支持。有关详细信息，请查看框架的相应文档。

将开发工具包与 Web 浏览器结合使用

所有主流 Web 浏览器支持 JavaScript 的执行。在 Web 浏览器中运行的 JavaScript 代码通常称为客户端 JavaScript。

在网络浏览器中使用 SDK for JavaScript 的方式，与在 Node.js 中使用它的方式不同。区别在于您加载 SDK 以及获取访问特定 Web 服务所需凭证的方法。如果在 Node.js 与浏览器之间使用特定 API 存在差别时，将说明这些差别。

有关 适用于 JavaScript 的 Amazon SDK 支持的浏览器列表，请参阅支持的 Web 浏览器。

常见使用案例

在浏览器脚本中使用 SDK for JavaScript 实现了多种颇具吸引力的使用案例。通过使用 SDK for JavaScript 访问各种 Web 服务，您可以在浏览器应用程序中构建一些东西，此处介绍了几个相关想法。

- 构建 Amazon 服务的自定义控制台，在其中您可以跨区域和服务访问并组合功能，从而最好地满足您的组织或项目需求。

- 使用 Amazon Cognito 以启用对您的浏览器应用程序和网站的经身份验证用户的访问，包括使用来自 Facebook 和其他提供商的第三方身份验证。
- 使用 Amazon Kinesis 实时处理点击流或其它营销数据。
- 使用 Amazon DynamoDB 实现无服务器数据持久性，例如针对网站访客或应用程序用户的单独用户首选项。
- 使用 Amazon Lambda 封装专有逻辑，您可以从浏览器脚本调用逻辑而无需下载和向用户泄露您的知识产权。

关于示例

您可以在 Amazon Code Example Library 中浏览 SDK for JavaScript 示例。

适用于 JavaScript 的 Amazon SDK 入门

适用于 JavaScript 的 Amazon SDK提供了在浏览器脚本或 Node.js 中对 Web 服务的访问。本部分有两个入门练习，向您演示如何在这两个 JavaScript 环境中使用 SDK for JavaScript。

主题

- 浏览器脚本入门
- Node.js 入门

浏览器脚本入门

此浏览器脚本示例向您演示：

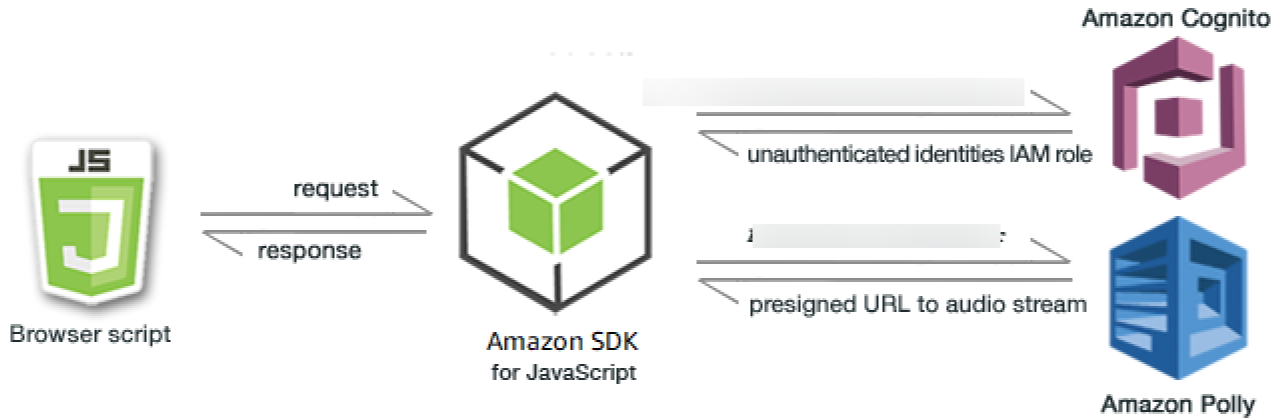
- 如何使用 Amazon Cognito Identity 通过浏览器脚本访问 Amazon 服务。
- 如何使用 Amazon Polly 将文本转换为合成语音。
- 如何使用预签名程序对象创建预签名 URL。

情景

Amazon Polly 云服务可以将文本转化为逼真的语音。可以使用 Amazon Polly 开发能提高参与度和可用性的应用程序。Amazon Polly 支持多种语言，并包含各种逼真的语音。有关 Amazon Polly 的更多信息，请参阅《Amazon Polly Developer Guide》。

本示例演示如何设置和运行简单的浏览器脚本，该脚本获取您输入的文本、将文本发送到 Amazon Polly，然后返回文本的合成音频 URL 供您播放。浏览器脚本使用 Amazon Cognito Identity 提供访问 Amazon 服务所需的凭证。您将看到在浏览器脚本中加载和使用 SDK for JavaScript 的基本模式。

> **Note**
>
> 在此示例中，合成语音的播放需要在支持 HTML 5 音频的浏览器中运行。



浏览器脚本使用 SDK for JavaScript，通过以下 API 来对文本执行合成：

- `AWS.CognitoIdentityCredentials` 构造函数
- `AWS.Polly.Presigner` 构造函数
- `getSynthesizeSpeechUrl`

步骤 1：创建一个 Amazon Cognito 身份池

在本练习中，您将创建并使用一个 Amazon Cognito 身份池，为浏览器脚本提供对 Amazon Polly 服务的无需验证身份的访问。创建身份池还会创建两个 IAM 角色，一个用于支持由身份提供商进行了身份验证的用户，另一个用于支持未经身份验证的来宾用户。

在本练习中，我们仅使用未经身份验证的用户角色，将重点放在任务上。您可在以后集成对身份提供商和通过身份验证的用户的支持。有关添加 Amazon Cognito 身份池的更多信息，请参阅《Amazon Cognito 开发人员指南》中的教程：创建身份池。

创建 Amazon Cognito 身份池

1. 登录 Amazon Web Services 管理控制台 并打开 Amazon Cognito 控制台，网址为 https://console.aws.amazon.com/cognito/。
2. 在左侧导航窗格中，选择身份池。
3. 选择创建身份池。
4. 在配置身份池信任中，选择来宾访问权限进行用户身份验证。
5. 在配置权限中，选择创建新的 IAM 角色并在 IAM 角色名称中输入一个名称（例如 getStartedRole）。
6. 在配置属性中，在身份池名称中输入一个名称（例如 getStartedPool）。

7. 在查看并创建中，确认您为新身份池所做的选择。选择编辑以返回向导并更改任何设置。完成后，选择创建身份池。
8. 记下新创建的 Amazon Cognito 身份池的身份池 ID 和区域。您需要这些值以便替换 步骤 4：写入浏览器脚本 中的*IDENTITY_POOL_ID* 和 *REGION*。

在创建 Amazon Cognito 身份池之后，您已准备好添加浏览器脚本使用 Amazon Polly 所需的权限。

步骤 2：将策略添加到创建的 IAM 角色

要启用浏览器脚本对 Amazon Polly 的访问以进行语音合成，请使用为 Amazon Cognito 身份池创建的未经身份验证的 IAM 角色。这需要您将 IAM policy 添加到角色。有关修改 IAM 角色的更多信息，请参阅《IAM 用户指南》中的修改角色权限策略。

将 Amazon Polly 策略添加到与未验证身份用户关联的 IAM 角色

1. 登录 Amazon Web Services 管理控制台，然后通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 在左侧导航窗格中，选择角色。
3. 选择要修改的角色的名称（例如，getStartedRole），然后选择权限选项卡。
4. 选择添加权限，然后选择附加策略。
5. 在此角色的添加权限页面中，找到并选中 AmazonPollyReadOnly 的复选框。

   > Note
   >
   > 您可以使用此流程来启用对任何 Amazon 服务的访问权限。

6. 选择添加权限。

创建 Amazon Cognito 身份池并将 Amazon Polly 的权限添加到未验证身份用户的 IAM 角色之后，您已准备好生成网页和浏览器脚本。

步骤 3：创建 HTML 页面

示例应用程序只有一个 HTML 页面，其中包含用户界面和浏览器脚本。要开始，请创建一个 HTML 文档并将以下内容复制到其中。该页面包括输入字段和按钮、`<audio>` 元素（用于播放合成语音）以及 `<p>` 元素（用于显示消息）。（请注意，完整示例显示在此页面的底部。）

有关 `<audio>` 元素的更多信息，请参阅音频。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Amazon SDK for JavaScript - Browser Getting Started Application</title>
  </head>

  <body>
    <div id="textToSynth">
      <input autofocus size="23" type="text" id="textEntry" value="It's very good to
 meet you."/>
      <button class="btn default" onClick="speakText()">Synthesize</button>
      <p id="result">Enter text above then click Synthesize</p>
    </div>
    <audio id="audioPlayback" controls>
      <source id="audioSource" type="audio/mp3" src="">
    </audio>
    <!-- (script elements go here) -->
 </body>
</html>
```

保存 HTML 文件，并将它命名为 polly.html。为应用程序创建用户界面之后，您已准备好添加运行应用程序的浏览器脚本代码。

步骤 4：写入浏览器脚本

创建浏览器脚本时首先要做的是，通过在页面的 <audio> 元素之后添加 <script> 元素来包括 SDK for JavaScript：要查找当前的 SDK_VERSION_NUMBER，请参阅 适用于 JavaScript 的 Amazon SDK API Reference Guide 中适用于 SDK for JavaScript 的 API 参考。

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></script>
```

然后，在开发工具包条目之后添加新的 <script type="text/javascript"> 元素。您将浏览器脚本添加到此元素。为 SDK 设置 Amazon 区域和凭证。接下来，创建名为 speakText() 的函数，该函数由按钮作为事件处理程序调用。

要通过 Amazon Polly 合成语音，您必须提供多种参数，包括输出的声音格式、采样率、所用语音的 ID 以及要播放的文本。在您最初创建参数时，请将 Text: 参数设置为空字符串；Text: 参数将设置为您从网页的 <input> 元素检索的值。将以下代码中的 *IDENTITY_POOL_ID* 和 *REGION* 替换为步骤 1：创建一个 Amazon Cognito 身份池中记下的值。

```
    <script type="text/javascript">

        // Initialize the Amazon Cognito credentials provider
        AWS.config.region = 'REGION';
        AWS.config.credentials = new AWS.CognitoIdentityCredentials({IdentityPoolId:
 'IDENTITY_POOL_ID'});

        // Function invoked by button click
        function speakText() {
            // Create the JSON parameters for getSynthesizeSpeechUrl
            var speechParams = {
                OutputFormat: "mp3",
                SampleRate: "16000",
                Text: "",
                TextType: "text",
                VoiceId: "Matthew"
            };
            speechParams.Text = document.getElementById("textEntry").value;
```

Amazon Polly 将合成语音作为音频流返回。在浏览器中播放音频的最简单方法是让 Amazon Polly 通过预签名 URL 来提供音频，然后您可以在网页的 <audio> 元素中设置 `src` 属性。

创建新的 `AWS.Polly` 服务对象。然后创建 `AWS.Polly.Presigner` 对象，您将用它来创建可从中检索合成语音音频的预签名 URL。您必须传递所定义的语音参数，以及您为 `AWS.Polly` 构造函数创建的 `AWS.Polly.Presigner` 服务对象。

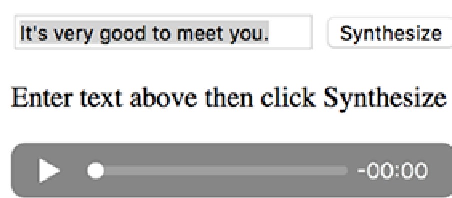
创建预签名程序对象之后，调用该对象的 `getSynthesizeSpeechUrl` 方法并传递语音参数。如果成功，此方法返回合成语音的 URL，然后您将其分配到 <audio> 元素进行播放。

```
            // Create the Polly service object and presigner object
            var polly = new AWS.Polly({apiVersion: '2016-06-10'});
            var signer = new AWS.Polly.Presigner(speechParams, polly)

            // Create presigned URL of synthesized speech file
            signer.getSynthesizeSpeechUrl(speechParams, function(error, url) {
            if (error) {
                document.getElementById('result').innerHTML = error;
            } else {
                document.getElementById('audioSource').src = url;
                document.getElementById('audioPlayback').load();
                document.getElementById('result').innerHTML = "Speech ready to play.";
            }
```

```
            });
        }
    </script>
```

步骤 5：运行示例

要运行示例应用程序，请将 `polly.html` 加载到 Web 浏览器中。浏览器的演示应该类似于此。



在输入框中输入您希望转换为语音的短语，然后选择 Synthesize (合成)。准备好播放音频时，将显示一条消息。使用音频播放器控件收听合成语音。

完整示例

下面是带有浏览器脚本的完整 HTML 页面。GitHub 上的此处也提供了该示例。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>AWS SDK for JavaScript - Browser Getting Started Application</title>
  </head>

  <body>
    <div id="textToSynth">
      <input autofocus size="23" type="text" id="textEntry" value="It's very good to
 meet you."/>
      <button class="btn default" onClick="speakText()">Synthesize</button>
      <p id="result">Enter text above then click Synthesize</p>
    </div>
    <audio id="audioPlayback" controls>
      <source id="audioSource" type="audio/mp3" src="">
    </audio>
    <script src="https://sdk.amazonaws.com/js/aws-sdk-2.410.0.min.js"></script>
    <script type="text/javascript">

        // Initialize the Amazon Cognito credentials provider
```

```
        AWS.config.region = 'REGION';
        AWS.config.credentials = new AWS.CognitoIdentityCredentials({IdentityPoolId:
 'IDENTITY_POOL_ID'});

        // Function invoked by button click
        function speakText() {
            // Create the JSON parameters for getSynthesizeSpeechUrl
            var speechParams = {
                OutputFormat: "mp3",
                SampleRate: "16000",
                Text: "",
                TextType: "text",
                VoiceId: "Matthew"
            };
            speechParams.Text = document.getElementById("textEntry").value;

            // Create the Polly service object and presigner object
            var polly = new AWS.Polly({apiVersion: '2016-06-10'});
            var signer = new AWS.Polly.Presigner(speechParams, polly)

            // Create presigned URL of synthesized speech file
            signer.getSynthesizeSpeechUrl(speechParams, function(error, url) {
            if (error) {
                document.getElementById('result').innerHTML = error;
            } else {
                document.getElementById('audioSource').src = url;
                document.getElementById('audioPlayback').load();
                document.getElementById('result').innerHTML = "Speech ready to play.";
            }
          });
        }
    </script>
  </body>
</html>
```

可能的增强功能

此处为该应用程序的变体，可用于进一步探索如何在浏览器脚本中使用 SDK for JavaScript。

- 使用其他声音输出格式进行试验。
- 添加选项，用于从 Amazon Polly 提供的多种语音中任选其一。
- 集成 Facebook 或 Amazon 等身份提供商以使用经过身份验证的 IAM 角色。

Node.js 入门

此 Node.js 代码示例演示：

- 如何为您的项目创建 `package.json` 清单。
- 如何安装和包括项目所用的模块。
- 如何通过 `AWS.S3` 客户端类创建 Amazon Simple Storage Service（Amazon S3）服务对象。
- 如何创建 Amazon S3 桶并将对象上传到该桶。

情景

示例演示如何设置和运行简单 Node.js 模块，该模块创建 Amazon S3 桶，然后将文本对象添加到该桶。

由于 Amazon S3 中的桶名称必须全局唯一，此示例包括了第三方 Node.js 模块，用于生成您可纳入桶名称的唯一 ID 值。此额外的模块名为 `uuid`。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 创建用于开发 Node.js 模块的工作目录。将此目录命名为 `awsnodesample`。请注意，必须在应用程序可更新的位置中创建该目录。例如，在 Windows 中，请勿在“C:\Program Files”下创建该目录。
- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。您可在 https://nodejs.org/en/download/current/ 中查找和下载适用于各种操作系统的 Node.js 最新版本和 LTS 版本。

目录

- 步骤 1：安装 SDK 和依赖项
- 步骤 2：配置凭证
- 步骤 3：为项目创建包 JSON
- 步骤 4：编写 Node.js 代码

- 步骤 5：运行示例

步骤 1：安装 SDK 和依赖项

使用 npm（Node.js 包管理器）安装 SDK for JavaScript 包：

从程序包中的 awsnodesample 目录，在命令行中键入以下内容。

```
npm install aws-sdk
```

此命令在项目中安装 SDK for JavaScript，并更新 package.json 以将该 SDK 作为项目依赖项列出。您可以在 npm 网站上通过搜索“aws-sdk”找到有关此程序包的信息。

接下来，通过在命令行中键入以下内容，在项目中安装 uuid 模块，这会安装模块并更新 package.json。有关 uuid 的更多信息，请参阅位于 https://www.npmjs.com/package/uuid 的模块页面。

```
npm install uuid
```

这些程序包及其关联的代码将安装在项目的 node_modules 子目录中。

有关安装 Node.js 包的更多信息，请参阅 npm（Node.js 包管理器）网站上的 Downloading and installing packages locally 和 Creating Node.js Modules。有关下载和安装适用于 JavaScript 的 Amazon SDK的信息，请参阅安装 SDK for JavaScript。

步骤 2：配置凭证

您需要向 Amazon 提供凭证，这样只有您的账户及其资源可由 SDK 访问。有关获取您的账户凭证的更多信息，请参阅使用 Amazon 进行 SDK 身份验证。

要保存此信息，我们建议您创建一个共享凭证文件。要了解如何操作，请参阅从共享凭证文件加载 Node.js 中的凭证。您的凭证文件应该类似于下例所示。

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

您可以通过在 Node.js 中执行以下代码来确定是否已正确设置您的凭证：

```
var AWS = require("aws-sdk");
```

```
AWS.config.getCredentials(function(err) {
  if (err) console.log(err.stack);
  // credentials not loaded
  else {
    console.log("Access key:", AWS.config.credentials.accessKeyId);
  }
});
```

同样，如果您已在 `config` 文件中正确设置了区域，您可以通过将 `AWS_SDK_LOAD_CONFIG` 环境变量设置为任何值并使用以下代码来显示该值：

```
var AWS = require("aws-sdk");

console.log("Region: ", AWS.config.region);
```

步骤 3：为项目创建包 JSON

创建 `awsnodesample` 项目目录之后，您可以创建和添加 `package.json` 文件用于保存 Node.js 项目的元数据。有关在 Node.js 项目中使用 `package.json` 的详细信息，请参阅 [Creating a package.json file]。

在项目目录中，创建名为 `package.json` 的新文件。然后将此 JSON 添加到文件。

```
{
  "dependencies": {},
  "name": "aws-nodejs-sample",
  "description": "A simple Node.js application illustrating usage of the SDK for
 JavaScript.",
  "version": "1.0.1",
  "main": "sample.js",
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "NAME",
  "license": "ISC"
}
```

保存该文件。在您安装所需的模块时，文件的 `dependencies` 部分将完成。您可以在 GitHub 上的此处找到显示这些依赖项的示例的 JSON 文件。

步骤 4：编写 Node.js 代码

创建名为 `sample.js` 的新文件以包含示例代码。首先，添加 `require` 函数调用以包括 SDK for JavaScript 和 `uuid` 模块，使其可供您使用。

通过将唯一 ID 值附加到可识别的前缀（在本例中为 `'node-sdk-sample-'`），生成用于创建 Amazon S3 桶的唯一桶名称。您可以通过调用 `uuid` 模块来生成唯一 ID。然后为用于将对象上传到存储桶的 `Key` 参数创建名称。

创建 `promise` 对象以调用 `createBucket` 服务对象的 `AWS.S3` 方法。收到成功响应时，创建将文本上传到新创建存储桶所需的参数。使用另一个 promise，调用 `putObject` 方法来将文本对象上传到存储桶。

```
// Load the SDK and UUID
var AWS = require("aws-sdk");
var uuid = require("uuid");

// Create unique bucket name
var bucketName = "node-sdk-sample-" + uuid.v4();
// Create name for uploaded object key
var keyName = "hello_world.txt";

// Create a promise on S3 service object
var bucketPromise = new AWS.S3({ apiVersion: "2006-03-01" })
  .createBucket({ Bucket: bucketName })
  .promise();

// Handle promise fulfilled/rejected states
bucketPromise
  .then(function (data) {
    // Create params for putObject call
    var objectParams = {
      Bucket: bucketName,
      Key: keyName,
      Body: "Hello World!",
    };
    // Create object upload promise
    var uploadPromise = new AWS.S3({ apiVersion: "2006-03-01" })
      .putObject(objectParams)
      .promise();
    uploadPromise.then(function (data) {
      console.log(
```

```
        "Successfully uploaded data to " + bucketName + "/" + keyName
      );
    });
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

此示例代码可在 GitHub 上的此处找到。

步骤 5：运行示例

键入以下命令以运行示例。

```
node sample.js
```

如果上传成功，您将在命令行中看到一条确认消息。您还可在 Amazon S3 控制台中找到存储桶和上传的文本对象。

设置 SDK for JavaScript

本部分中的主题介绍了如何安装 SDK for JavaScript 以在 Web 浏览器和 Node.js 中使用。其中还演示了如何加载开发工具包，这样您可以访问开发工具包支持的 Web 服务。

> **Note**
>
> React Native 开发人员应使用 Amazon Amplify 在 Amazon 上创建新项目。有关详细信息，请参阅 aws-sdk-react-native 存档。

主题

- 先决条件
- 安装 SDK for JavaScript
- 加载 SDK for JavaScript
- 从版本 1 升级 SDK for JavaScript

先决条件

在您使用适用于 JavaScript 的 Amazon SDK之前，确定您的代码需要在 Node.js 还是 Web 浏览器中运行。在此之后，请执行以下操作：

- 对于 Node.js，在服务器上安装 Node.js（如果尚未安装）。
- 对于 Web 浏览器，确定您需要支持的浏览器版本。

主题

- 设置 Amazon Node.js 环境
- 支持的 Web 浏览器

设置 Amazon Node.js 环境

要设置可在其中运行应用程序的 Amazon Node.js 环境，请使用以下任意方法：

- 选择预安装 Node.js 的亚马逊机器映像（AMI），并使用该 AMI 创建 Amazon EC2 实例。创建 Amazon EC2 实例时，从 Amazon Web Services Marketplace 中选择您的 AMI。在 Amazon Web

Services Marketplace 中搜索 Node.js，然后选择包含预安装 Node.js（32 位或 64 位）版本的 AMI 选项。

- 创建 Amazon EC2 实例并在该实例上安装 Node.js。有关如何在 Amazon Linux 实例上安装 Node.js 的更多信息，请参阅教程：在 Amazon EC2 实例上设置 Node.js。
- 使用 Amazon Lambda 创建无服务器环境，将 Node.js 作为 Lambda 函数运行。有关在 Lambda 函数中使用 Node.js 的更多信息，请参阅《Amazon Lambda 开发人员指南》中的编程模型（Node.js）。
- 将 Node.js 应用程序部署到 Amazon Elastic Beanstalk。有关将 Node.js 与 Elastic Beanstalk 配合使用的更多信息，请参阅《Amazon Elastic Beanstalk 开发人员指南》中的将 Node.js 应用程序部署到 Amazon Elastic Beanstalk。

支持的 Web 浏览器

SDK for JavaScript 支持所有现代 Web 浏览器，包括以下最低版本：

浏览器	版本
Google Chrome	28.0+
Mozilla Firefox	26.0+
Opera	17.0+
Microsoft Edge	25.10+
Windows Internet Explorer	不适用
Apple Safari	5+
Android 浏览器	4.3+

> **Note**
>
> Amazon Amplify 等框架可能无法提供与 SDK for JavaScript 相同的浏览器支持。有关详细信息，请查看框架的相应文档。

安装 SDK for JavaScript

您是否安装以及如何安装适用于 JavaScript 的 Amazon SDK取决于代码在 Node.js 模块还是浏览器脚本中执行。

开发工具包中的服务并不是全部直接可用的。要确定适用于 JavaScript 的 Amazon SDK当前支持哪些服务，请参阅 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md

Node

为 Node.js 安装适用于 JavaScript 的 Amazon SDK的首选方法是使用 Node.js 程序包管理器 npm。要执行此操作，请在命令行中键入此内容。

```
npm install aws-sdk
```

如果您看到此错误消息：

```
npm WARN deprecated node-uuid@1.4.8: Use uuid module instead
```

请在命令行上键入这些命令：

```
npm uninstall --save node-uuid
npm install --save uuid
```

Browser

您无需安装开发工具包以在浏览器脚本中使用它。您可在 HTML 页面中，使用脚本直接从 Amazon Web Services 加载托管的 SDK 包。托管的开发工具包支持实施跨源资源共享 (CORS) 的 Amazon 服务子集。有关更多信息，请参阅 加载 SDK for JavaScript。

您可以创建自定义工作版本的开发工具包，在其中可以选择要使用的特定 Web 服务和版本。然后，您可以下载自定义开发工具包用于本地部署，并可托管它供应用程序使用。有关创建自定义版本的开发工具包的更多信息，请参阅为浏览器构建 SDK。

您可以从 GitHub 下载当前适用于 JavaScript 的 Amazon SDK的简化和非简化可分发版本，地址为：

https://github.com/aws/aws-sdk-js/tree/master/dist

使用 Bower 安装

Bower 是适用于 Web 的程序包管理器。安装 Bower 之后，您可以用它来安装开发工具包。要使用 Bower 安装开发工具包，请在终端窗口中键入以下命令：

```
bower install aws-sdk-js
```

加载 SDK for JavaScript

加载 SDK for JavaScript 的方式取决于您加载它是为了在 Web 浏览器还是 Node.js 中运行。

开发工具包中的服务并不是全部直接可用的。要确定适用于 JavaScript 的 Amazon SDK当前支持哪些服务，请参阅 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md

Node.js

安装 SDK 之后，您可以使用 `require`，将 Amazon 包加载到节点应用程序中。

```
var AWS = require('aws-sdk');
```

React Native

要在 React Native 项目中使用开发工具包，请首先使用 npm 安装开发工具包：

```
npm install aws-sdk
```

在您的应用程序中，使用以下代码引用开发工具包的 React Native 兼容版本：

```
var AWS = require('aws-sdk/dist/aws-sdk-react-native');
```

Browser

开始使用 SDK 的最快捷方法是直接从 Amazon Web Services 加载托管的 SDK 包。为此，请采用以下格式将 `<script>` 元素添加到您的 HTML 页面：

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></
script>
```

要查找当前的 SDK_VERSION_NUMBER，请参阅 适用于 JavaScript 的 Amazon SDK API Reference Guide 中适用于 SDK for JavaScript 的 API 参考。

页面中加载开发工具包之后，可从全局变量 `AWS`（或 `window.AWS`）使用开发工具包。

如果您使用 browserify 打包代码和模块依赖项，则使用 `require` 加载开发工具包，就像在 Node.js 中的操作一样。

从版本 1 升级 SDK for JavaScript

以下说明可帮助您将 SDK for JavaScript 从版本 1 升级到版本 2。

自动转换输入/输出中的 Base64 和时间戳类型

该开发工具包现在可代表用户自动编码和解码 base64 编码值以及时间戳值。对于请求会发送 base64 或时间戳值的任意操作，或者在返回响应中允许 base64 编码值的任意操作，此更改会有影响。

以前需要转换为 base64 的代码，现在不再需要转换。编码为 base64 的值现在作为来自服务器响应的缓冲区对象返回，也可以作为缓冲区输入传递。例如，以下版本 1 `SQS.sendMessage` 参数：

```
var params = {
   MessageBody: 'Some Message',
   MessageAttributes: {
      attrName: {
         DataType: 'Binary',
         BinaryValue: new Buffer('example text').toString('base64')
      }
   }
};
```

可以按以下方式重新编写。

```
var params = {
   MessageBody: 'Some Message',
   MessageAttributes: {
      attrName: {
         DataType: 'Binary',
         BinaryValue: 'example text'
      }
   }
};
```

以下是读取消息的方式。

```
sqs.receiveMessage(params, function(err, data) {
  // buf is <Buffer 65 78 61 6d 70 6c 65 20 74 65 78 74>
  var buf = data.Messages[0].MessageAttributes.attrName.BinaryValue;
  console.log(buf.toString()); // "example text"
});
```

将 response.data.RequestId 移动到 response.requestId

现在，该开发工具包将所有服务的请求 ID 存储到 `response` 对象上一致的位置，而不是 `response.data` 属性内部。这可以改进以不同方式公开请求 ID 的跨服务一致性。这也是一个重大更改，将 `response.data.RequestId` 属性重命名为 `response.requestId`（回调函数内部的 `this.requestId`）。

在您的代码中，更改以下内容：

```
svc.operation(params, function (err, data) {
  console.log('Request ID:', data.RequestId);
});
```

更改为以下来源：

```
svc.operation(params, function () {
  console.log('Request ID:', this.requestId);
});
```

公开的包装元素

如果您使用 `AWS.ElastiCache`、`AWS.RDS` 或 `AWS.Redshift`，则对于一些操作，您必须通过响应中的顶级输出属性访问响应。

例如，用于返回以下内容的 `RDS.describeEngineDefaultParameters` 方法。

```
{ Parameters: [ ... ] }
```

现在它返回以下内容。

```
{ EngineDefaults: { Parameters: [ ... ] } }
```

各个服务受影响操作的列表如下表中所示。

客户端类	操作
AWS.ElastiCache	authorizeCacheSecurityGroupIngress createCacheCluster createCacheParameterGroup createCacheSecurityGroup createCacheSubnetGroup createReplicationGroup deleteCacheCluster deleteReplicationGroup describeEngineDefaultParameters modifyCacheCluster modifyCacheSubnetGroup modifyReplicationGroup purchaseReservedCacheNodesOffering rebootCacheCluster revokeCacheSecurityGroupIngress
AWS.RDS	addSourceIdentifierToSubscription authorizeDBSecurityGroupIngress copyDBSnapshot createDBInstance createDBInstanceReadReplica

客户端类	操作
	createDBParameterGroup createDBSecurityGroup createDBSnapshot createDBSubnetGroup createEventSubscription createOptionGroup deleteDBInstance deleteDBSnapshot deleteEventSubscription describeEngineDefaultParameters modifyDBInstance modifyDBSubnetGroup modifyEventSubscription modifyOptionGroup promoteReadReplica purchaseReservedDBInstancesOffering rebootDBInstance removeSourceIdentifierFromSubscription restoreDBInstanceFromDBSnapshot restoreDBInstanceToPointInTime

客户端类	操作
	revokeDBSecurityGroupIngress

客户端类	操作
AWS.Redshift	authorizeClusterSecurityGroupIngress authorizeSnapshotAccess copyClusterSnapshot createCluster createClusterParameterGroup createClusterSecurityGroup createClusterSnapshot createClusterSubnetGroup createEventSubscription createHsmClientCertificate createHsmConfiguration deleteCluster deleteClusterSnapshot describeDefaultClusterParameters disableSnapshotCopy enableSnapshotCopy modifyCluster modifyClusterSubnetGroup modifyEventSubscription modifySnapshotCopyRetentionPeriod

客户端类	操作
	purchaseReservedNodeOffering rebootCluster restoreFromClusterSnapshot revokeClusterSecurityGroupIngress revokeSnapshotAccess rotateEncryptionKey

删除的客户端属性

从服务对象中删除了 `.Client` 和 `.client` 属性。如果您在服务类上使用 `.Client` 属性，或在服务对象实例上使用 `.client` 属性，请从代码中删除这些属性。

以下代码用于 SDK for JavaScript 的版本 1：

```
var sts = new AWS.STS.Client();
// or
var sts = new AWS.STS();

sts.client.operation(...);
```

应更改为以下代码。

```
var sts = new AWS.STS();
sts.operation(...)
```

配置 SDK for JavaScript

使用 SDK for JavaScript 通过 API 调用 Web 服务之前，您必须配置 SDK。至少，您必须配置以下设置：

- 您将在其中请求服务的区域。
- 授权您访问开发工具包资源的 凭证。

除了这些设置，您可能还必须配置您的 Amazon 资源的权限。例如，您可以限制对某个 Amazon S3 桶的访问或者限制对某个 Amazon DynamoDB 表只能进行只读访问。

Amazon SDKs and Tools Reference Guide 还包含许多 Amazon SDK 中常见的设置、功能和其它基础概念。

本部分中的主题描述了为 Web 浏览器中运行的 Node.js 和 JavaScript 配置 SDK for JavaScript 的各种方法。

主题

- 使用全局配置对象
- 设置 Amazon 区域
- 指定自定义终端节点
- 使用 Amazon 进行 SDK 身份验证
- 设置凭证
- 锁定 API 版本
- Node.js 注意事项
- 浏览器脚本注意事项
- 使用 Webpack 捆绑应用程序

使用全局配置对象

可通过两种方式配置开发工具包：

- 使用 `AWS.Config` 设置全局配置。

- 将额外的配置信息传递给服务对象。

使用 AWS.Config 设置全局配置通常更容易上手，但服务级别的配置可以提供对单个服务的更多控制。由 AWS.Config 指定的全局配置为您随后创建的服务对象提供默认设置，从而简化其配置。但是，当您的需求与全局配置不同时，您可以更新各个服务对象的配置。

设置全局配置

在代码中加载 aws-sdk 程序包后，您可以使用 AWS 全局变量来访问开发工具包的类并与各个服务进行交互。开发工具包中包含一个全局配置对象 AWS.Config，您可以利用它来指定您的应用程序所需的开发工具包配置设置。

根据您的应用程序的需求设置 AWS.Config 属性，对开发工具包进行配置。下表总结了常用于设置开发工具包配置的 AWS.Config 属性。

配置选项	说明
credentials	必填？ 指定用于确定服务和资源访问权限的凭证。
region	：必填项。指定执行服务请求的区域。
maxRetries	可选。指定给定请求的最大重试次数。
logger	可选。指定要将调试信息写入的记录器对象。
update	可选。使用新值更新当前配置。

有关配置对象的更多信息，请参阅 API 参考中的 [Class: AWS.Config]。

全局配置示例

您必须在 AWS.Config 中设置区域和凭证。您可以将这些属性设置为 AWS.Config 构造函数的一部分，如以下浏览器脚本示例所示：

```
var myCredentials = new
 AWS.CognitoIdentityCredentials({IdentityPoolId:'IDENTITY_POOL_ID'});
var myConfig = new AWS.Config({
  credentials: myCredentials, region: 'us-west-2'
```

```
});
```

您还可以在使用 AWS.Config 方法创建 update 后设置这些属性，如用于更新区域的以下示例所示：

```
myConfig = new AWS.Config();
myConfig.update({region: 'us-east-1'});
```

您可以通过调用 getCredentials 的静态 AWS.config 方法来获取默认的凭证：

```
var AWS = require("aws-sdk");

AWS.config.getCredentials(function(err) {
  if (err) console.log(err.stack);
  // credentials not loaded
  else {
    console.log("Access key:", AWS.config.credentials.accessKeyId);
  }
});
```

同样，如果您已在 config 文件中正确设置了区域，您可以通过将 AWS_SDK_LOAD_CONFIG 环境变量设置为任何值并调用 AWS.config 的静态 region 属性来获取该值：

```
var AWS = require("aws-sdk");

console.log("Region: ", AWS.config.region);
```

设置每个服务的配置

您在 SDK for JavaScript 中使用的每个服务都是通过服务对象访问的，该服务对象是该服务的 API 的一部分。例如，要访问 Amazon S3 服务，您需要创建 Amazon S3 服务对象。您可以将特定于某项服务的配置设置指定为该服务对象的构造函数的一部分。在服务对象上设置配置值时，构造函数将采用 AWS.Config 使用的所有配置值，包括凭证。

例如，如果需要访问多个区域中的 Amazon EC2 对象，请为每个区域创建一个 Amazon EC2 服务对象，然后相应地设置每个服务对象的区域配置。

```
var ec2_regionA = new AWS.EC2({region: 'ap-southeast-2', maxRetries: 15, apiVersion:
 '2014-10-01'});
var ec2_regionB = new AWS.EC2({region: 'us-east-1', maxRetries: 15, apiVersion:
 '2014-10-01'});
```

在使用 `AWS.Config` 配置开发工具包时，您还可以设置特定于服务的配置值。全局配置对象支持许多特定于服务的配置选项。有关特定于服务的配置的更多信息，请参阅 适用于 JavaScript 的 Amazon SDK API 参考中的 `Class: AWS.Config`。

不可变的配置数据

全局配置更改适用于所有新创建的服务对象的请求。新创建的服务对象首先使用当前的全局配置数据进行配置，然后使用任何本地配置选项进行配置。您对全局 `AWS.config` 对象所做的更新不适用于以前创建的服务对象。

必须使用新配置数据手动更新现有服务对象，或者必须创建和使用具有新配置数据的新服务对象。以下示例使用新配置数据创建一个新的 Amazon S3 服务对象：

```
s3 = new AWS.S3(s3.config);
```

设置 Amazon 区域

区域是同一地理区域中的一组指定的 Amazon 资源。区域的一个例子是 `us-east-1`，即美国东部（弗吉尼亚州北部）区域。在配置 SDK for JavaScript 时指定区域，以便 SDK 访问该区域中的资源。有些服务仅在特定区域中提供。

默认情况下，SDK for JavaScript 不选择区域。但是，您可以使用环境变量、共享的 `config` 文件或全局配置对象来设置区域。

在客户端类构造函数中

实例化服务对象时，可以将该资源的 区域指定为客户端类构造函数的一部分，如此处所示。

```
var s3 = new AWS.S3({apiVersion: '2006-03-01', region: 'us-east-1'});
```

使用全局配置对象

要在 JavaScript 代码中设置区域，请更新 `AWS.Config` 全局配置对象，如此处所示。

```
AWS.config.update({region: 'us-east-1'});
```

有关当前区域和每个区域中可用服务的更多信息，请参阅《Amazon Web Services 一般参考》中的 Amazon 区域和端点。

使用环境变量

您可以使用 AWS_REGION 环境变量设置区域。如果您定义此变量，则 SDK for JavaScript 会读取并使用它。

使用共享配置文件

您可以存储共享凭证文件供开发工具包使用的凭证，共享配置文件与之非常相似，您可以将区域和其他配置设置保存在开发工具包使用的名为 config 的共享文件中。如果 AWS_SDK_LOAD_CONFIG 环境变量已设置为任何值，则 SDK for JavaScript 在加载时会自动搜索 config 文件。保存 config 文件的位置取决于您的操作系统：

- Linux、macOS 或 Unix 用户：~/.aws/config
- Windows 用户：C:\Users\USER_NAME\.aws\config

如果您还没有共享 config 文件，您可以在指定的目录中创建一个。在以下示例中，config 文件设置区域和输出格式。

```
[default]
   region=us-east-1
   output=json
```

有关使用共享配置文件和共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证或《Amazon Command Line Interface 用户指南》中的配置和凭证文件。

区域的优先设置顺序

区域设置的优先顺序如下：

- 如果将某个区域传递给客户端类构造函数，则使用该区域。如果没有传递，则...
- 如果在全局配置对象上设置了区域，则使用该区域。如果没有传递，则...
- 如果 AWS_REGION 环境变量是真值，则使用该区域。如果没有传递，则...
- 如果 AMAZON_REGION 环境变量是真值，则使用该区域。如果没有传递，则...
- 如果 AWS_SDK_LOAD_CONFIG 环境变量设置为任何值，并且共享凭证文件（~/.aws/credentials 或由 AWS_SHARED_CREDENTIALS_FILE 指示的路径）对于配置的配置文件包含某个区域，则使用该区域。如果没有传递，则...

- 如果 AWS_SDK_LOAD_CONFIG 环境变量设置为任何值，并且配置文件（~/.aws/config 或由 AWS_CONFIG_FILE 指示的路径）对于配置的配置文件包含某个区域，则使用该区域。

指定自定义终端节点

在 SDK for JavaScript 中调用 API 方法针对的是服务端点 URI。默认情况下，这些终端节点是根据您为代码配置的区域构建的。但是，在某些情况下，您需要为 API 调用指定自定义终端节点。

终端节点字符串格式

终端节点值应为以下格式的字符串：

https://*{service}.{region}*.amazonaws.com

ap-northeast-3 区域的终端节点

日本的 ap-northeast-3 区域不会被区域枚举 API（例如 EC2.describeRegions）返回。要定义此区域的终端节点，请遵循前面描述的格式。因此，此区域的 Amazon EC2 端点将是

ec2.ap-northeast-3.amazonaws.com

MediaConvert 的端点

您需要创建一个与 MediaConvert 配合使用的自定义端点。每个客户账户都分配了自己的终端节点，您必须使用该终端节点。以下示例演示如何将自定义端点与 MediaConvert 配合使用。

```
// Create MediaConvert service object using custom endpoint
var mcClient = new AWS.MediaConvert({endpoint: 'https://abcd1234.mediaconvert.us-
west-1.amazonaws.com'});

var getJobParams = {Id: 'job_ID'};

mcClient.getJob(getJobParams, function(err, data)) {
   if (err) console.log(err, err.stack); // an error occurred
   else console.log(data); // successful response
};
```

要获取您的账户 API 终端节点，请参阅 API 参考中的MediaConvert.describeEndpoints。

确保在代码中指定与自定义终端节点 URI 中的区域相同的区域。区域设置和自定义终端节点 URI 之间的不匹配可能导致 API 调用失败。

有关 MediaConvert 的更多信息，请参阅 API 参考中的 AWS.MediaConvert 类或《AWS Elemental MediaConvert User Guide》。

使用 Amazon 进行 SDK 身份验证

使用 Amazon Web Services 服务进行开发时，您必须确定您的代码如何向 Amazon 进行身份验证。您可以通过不同的方式来配置对 Amazon 资源的编程访问权限，具体取决于环境和可用的 Amazon 访问权限。

要选择您的身份验证方法并针对 SDK 进行配置，请参阅 Amazon SDK 和工具参考指南中的身份验证和访问。

我们建议：在本地开发且雇主未向其提供身份验证方法的新用户应设置 Amazon IAM Identity Center。此方法包括安装 Amazon CLI 以便于配置和定期登录 Amazon 访问门户。如果选择此方法，则在完成 Amazon SDK 和工具参考指南中的 IAM Identity Center 身份验证程序后，您的环境应包含以下元素：

- Amazon CLI，您可用于在运行应用程序之前启动 Amazon 访问门户会话。
- 共享 Amazonconfig 文件，其 [default] 配置文件包含一组可从 SDK 中引用的配置值。要查找此文件的位置，请参阅《Amazon SDK 和工具参考指南》中的共享文件的位置。
- 共享 config 文件设置了 region 设置。这将设置 SDK 用于 Amazon 请求的默认值 Amazon Web Services 区域。此区域用于未指定使用区域的 SDK 服务请求。
- 在向 Amazon 发送请求之前，SDK 使用配置文件的 SSO 令牌提供程序配置来获取凭证。sso_role_name 值是与 IAM Identity Center 权限集关联的 IAM 角色，允许访问应用程序中使用的 Amazon Web Services 服务。

  以下示例 config 文件展示了使用 SSO 令牌提供程序配置来设置的默认配置文件。配置文件的 sso_session 设置是指所指定的 sso-session 节。sso-session 节包含启动 Amazon 访问门户会话的设置。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
```

```
sso_registration_scopes = sso:account:access
```

SDK for JavaScript 无需向应用程序添加其它包（例如 SSO 和 SSOOIDC）即可使用 IAM Identity Center 身份验证。

开始 Amazon 访问门户会话

在运行访问 Amazon Web Services 服务 的应用程序之前，需要有活动的 Amazon 访问门户会话，以便开发工具包使用 IAM Identity Center 身份验证来解析凭证。根据配置的会话时长，访问权限最终将过期，并且开发工具包将遇到身份验证错误。要登录 Amazon 访问门户，请在 Amazon CLI 中运行以下命令。

```
aws sso login
```

如果遵循引导并具有默认的配置文件设置，则无需使用 `--profile` 选项来调用该命令。如果您的 SSO 令牌提供程序配置在使用指定的配置文件，则命令为 `aws sso login --profile named-profile`。

要选择性测试是否已有活动会话，请运行以下 Amazon CLI 命令。

```
aws sts get-caller-identity
```

如果会话是活动的，则对此命令的响应会报告共享 `config` 文件中配置的 IAM Identity Center 账户和权限集。

> **Note**
>
> 如果您已经有一个有效的 Amazon 访问门户会话并且运行了 `aws sso login`，则无需提供凭证。
>
> 登录过程可能会提示您允许 Amazon CLI 访问您的数据。由于 Amazon CLI 基于适用于 Python 的 SDK 而构建，因此权限消息可能包含 `botocore` 名称的变体。

更多身份验证信息

人类用户，也称为人类身份，是应用程序的人员、管理员、开发人员、操作员和使用者。他们必须有身份才能访问您的 Amazon 环境和应用程序。作为组织成员的人类用户（即您、开发人员）也称为工作人员身份。

访问 Amazon 时使用临时凭证。您可以使用身份提供商来以担任角色的形式提供为人类用户对 Amazon 账户的联合访问权限，这将提供临时证书。对于集中式访问权限管理，我们建议使用 Amazon IAM Identity Center (IAM Identity Center) 来管理对您账户的访问权限以及这些账户中的其他权限。有关更多替代方案，请参阅以下内容：

- 有关最佳实践的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实践。
- 要创建短期 Amazon 凭证，请参阅《IAM 用户指南》中的临时安全凭证。
- 要了解其它 SDK for JavaScript 凭证提供程序，请参阅《Amazon SDKs and Tools Reference Guide》中的 Standardized credential providers。

设置凭证

Amazon 使用凭证来识别谁正在调用服务以及是否允许访问所请求的资源。

无论是在 Web 浏览器中运行还是在 Node.js 服务器中运行，您的 JavaScript 代码都必须先获取有效凭证，然后才能通过 API 访问服务。可以使用 `AWS.Config` 或针对每个服务在配置对象上全局设置凭证，方法是将凭证直接传递给服务对象。

有几种方法可以设置在 Node.js 和 Web 浏览器中的 JavaScript 之间不同的凭证。本部分中的主题介绍如何在 Node.js 或 Web 浏览器中设置凭证。在每种情况下，选项以推荐顺序显示。

凭证的最佳实践

正确设置凭证可确保您的应用程序或浏览器脚本可以访问所需的服务和资源，同时最大限度地减少可能影响关键任务型应用程序或危及敏感数据的安全问题。

设置凭证时应用的一个重要原则是始终授予您的任务所需的最小权限。提供对资源的最小权限并根据需要添加更多权限更安全，而不是提供超过最小权限的权限，因此需要修复以后可能发现的安全问题。例如，除非您需要读取和写入单独的资源（例如 Amazon S3 桶或 DynamoDB 表中的对象），否则请将这些权限设置为只读。

有关授予最低权限的更多信息，请参阅《IAM 用户指南》最佳实践主题中的授予最低权限部分。

Warning

虽然可以这样做，但我们建议您不要在应用程序或浏览器脚本中对凭证进行硬编码。硬编码凭证存在泄露敏感信息的风险。

有关如何管理访问密钥的更多信息，请参阅《Amazon Web Services 一般参考》中的管理 Amazon 访问密钥的最佳实践。

主题

- 在 Node.js 中设置凭证
- 在 Web 浏览器中设置凭证

在 Node.js 中设置凭证

Node.js 有几种方法可以为 SDK 提供凭证。其中一些方法更安全，而另一些方法则在开发应用程序时可以提供更大的便利。在 Node.js 中获取凭证时，请注意依赖多个源，例如环境变量和您加载的 JSON 文件。您可以更改运行代码的权限，而不会意识到已发生更改。

下面是按推荐顺序提供凭证的方法：

1. 从适用于 Amazon EC2 的 Amazon Identity and Access Management（IAM）角色加载
2. 从共享凭证文件 (`~/.aws/credentials`) 加载
3. 从环境变量加载
4. 从磁盘上的 JSON 文件加载
5. JavaScript 开发工具包提供的其他 credential-provider 类

如果多个凭证源适用于该开发工具包，则选择的默认优先顺序如下所示：

1. 通过服务-客户端构造函数显式设置的凭证
2. 环境变量
3. 共享凭证文件
4. 从 ECS 凭证提供程序加载的凭证（如果适用）
5. 通过使用共享的 Amazon config 文件或共享凭证文件中指定的凭据过程获取的凭证。有关更多信息，请参阅the section called “使用已配置凭证过程的凭证”。
6. 使用 Amazon EC2 实例的凭证提供程序从 Amazon IAM 加载的凭证（如果在实例元数据中配置）

有关更多信息，请参阅 API 参考中的 `Class: AWS.Credentials` 和 `Class: AWS.CredentialProviderChain`。

> ⚠ Warning
>
> 虽然可以这样做，但我们不建议在您的应用程序中对您的 Amazon 凭证进行硬编码。硬编码凭证存在暴露您的访问密钥 ID 和秘密访问密钥的风险。

本部分中的主题介绍如何将凭证加载到 Node.js 中。

主题

- 从适用于 Amazon EC2 的 IAM 角色加载 Node.js 中的凭证
- 加载 Node.js Lambda 函数的凭证
- 从共享凭证文件加载 Node.js 中的凭证
- 从环境变量加载 Node.js 中的凭证
- 从 JSON 文件加载 Node.js 中的凭证
- 使用已配置过程在 Node.js 中加载凭证

从适用于 Amazon EC2 的 IAM 角色加载 Node.js 中的凭证

如果在 Amazon EC2 实例上运行 Node.js 应用程序，则可以利用 Amazon EC2 的 IAM 角色自动为实例提供凭证。如果将实例配置为使用 IAM 角色，则 SDK 会自动为您的应用程序选择 IAM 凭证，从而无需手动提供凭证。

有关向 Amazon EC2 实例添加 IAM 角色的更多信息，请参阅《Amazon SDK 和工具参考指南》中的使用适用于 Amazon EC2 实例的 IAM 角色。

加载 Node.js Lambda 函数的凭证

在创建 Amazon Lambda 函数时，您必须创建一个具有该函数的执行权限的特殊 IAM 角色。此角色称为执行角色。当您设置 Lambda 函数时，您必须指定您创建的 IAM 角色作为相应的执行角色。

执行角色为 Lambda 函数提供运行和调用其它 Web 服务所需的凭证。因此，您不需要为在 Lambda 函数中编写的 Node.js 代码提供凭证。

有关创建 Lambda 执行角色的更多信息，请参阅《Amazon Lambda 开发人员指南》中的管理权限：使用 IAM 角色（执行角色）。

从共享凭证文件加载 Node.js 中的凭证

您可以将 Amazon 凭证数据保存在 SDK 和命令行界面使用的共享文件中。当 SDK for JavaScript 加载时，它会自动搜索共享凭证文件（名为“credentials”）。保存共享凭证文件的位置取决于您的操作系统：

- Linux、Unix 和 macOS 上的共享凭证文件：`~/.aws/credentials`
- Windows 上的共享凭证文件：`C:\Users\USER_NAME\.aws\credentials`

如果您还没有共享凭证文件，请参阅 使用 Amazon 进行 SDK 身份验证。按照这些说明操作后，您应该在凭证文件中看到与以下内容类似的文本，其中 *<YOUR_ACCESS_KEY_ID>* 是您的访问密钥 ID，*<YOUR_SECRET_ACCESS_KEY>* 是您的秘密访问密钥：

```
[default]
aws_access_key_id = <YOUR_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_SECRET_ACCESS_KEY>
```

有关显示正在使用的此文件的示例，请参阅 Node.js 入门。

`[default]` 部分标题指定默认配置文件和凭证的相关值。您可以在同一共享配置文件中创建其他配置文件，每个配置文件都有自己的凭证信息。以下示例显示了具有默认配置文件和两个其他配置文件的配置文件：

```
[default] ; default profile
aws_access_key_id = <DEFAULT_ACCESS_KEY_ID>
aws_secret_access_key = <DEFAULT_SECRET_ACCESS_KEY>

[personal-account] ; personal account profile
aws_access_key_id = <PERSONAL_ACCESS_KEY_ID>
aws_secret_access_key = <PERSONAL_SECRET_ACCESS_KEY>

[work-account] ; work account profile
aws_access_key_id = <WORK_ACCESS_KEY_ID>
aws_secret_access_key = <WORK_SECRET_ACCESS_KEY>
```

默认情况下，开发工具包会检查 `AWS_PROFILE` 环境变量以确定使用哪些配置文件。如果在您的环境中未设置 `AWS_PROFILE` 变量，开发工具包将对 `[default]` 配置文件使用凭证。要使用其中一个替代配置文件，请设置或更改 `AWS_PROFILE` 环境变量的值。例如，假定上述配置文件，要从工作账户使用凭证，请将 `AWS_PROFILE` 环境变量设置为 `work-account`（相应适用于您的操作系统）。

Note

设置环境变量时，请务必随后采取适当操作（根据您的操作系统的需求）以使这些变量在 shell 或命令环境中可用。

设置环境变量（如果需要）后，您可以运行一个使用 SDK 的 JavaScript 文件，例如，一个名为 `script.js` 的文件。

```
$ node script.js
```

您也可以明确选择开发工具包使用的配置文件，方法为在加载开发工具包之前设置 `process.env.AWS_PROFILE`，或者选择以下示例所示的凭证提供程序：

```
var credentials = new AWS.SharedIniFileCredentials({profile: 'work-account'});
AWS.config.credentials = credentials;
```

从环境变量加载 Node.js 中的凭证

SDK 会自动检测环境中设置为变量的 Amazon 凭证，并将其用于 SDK 请求，从而无需管理应用程序中的凭证。您设置为提供凭证的环境变量是：

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_SESSION_TOKEN`

有关设置环境变量的更多详细信息，请参阅《Amazon SDKs and Tools Reference Guide》中的 Environment variables support。

从 JSON 文件加载 Node.js 中的凭证

您可以使用 `AWS.config.loadFromPath` 从磁盘上的 JSON 文档加载配置和凭证。指定的路径相对于进程的当前工作目录。例如，使用以下内容从 `'config.json'` 文件加载凭证：

```
{ "accessKeyId": <YOUR_ACCESS_KEY_ID>, "secretAccessKey": <YOUR_SECRET_ACCESS_KEY>,
 "region": "us-east-1" }
```

然后使用以下代码：

```
var AWS = require("aws-sdk");
AWS.config.loadFromPath('./config.json');
```

> **Note**
>
> 从 JSON 文档加载配置数据会重置所有现有的配置数据。使用此技术后添加其他配置数据。浏览器脚本不支持从 JSON 文档加载凭证。

使用已配置过程在 Node.js 中加载凭证

您可以使用未内置到开发工具包中的方法来获取凭证。要执行此操作，请在共享 Amazon 配置文件或共享凭证文件中指定凭证过程。如果 `AWS_SDK_LOAD_CONFIG` 环境变量设置为任何值，则 SDK 会认为在配置文件中指定的过程优先于在凭证文件中指定的过程（如果有）。

有关在共享 Amazon 配置文件或共享凭证文件中指定凭证过程的详细信息，请参阅《Amazon CLI 命令参考》，尤其是有关从外部过程获取凭证的信息。

有关使用 `AWS_SDK_LOAD_CONFIG` 环境变量的信息，请参阅本文档中的the section called “使用共享配置文件”。

在 Web 浏览器中设置凭证

有几种方法可以从浏览器脚本为 SDK 提供凭证。其中一些方法更安全，而另一些方法则在开发脚本时可以提供更大的便利。下面是按推荐顺序提供凭证的方法：

1. 使用 Amazon Cognito Identity 验证用户身份和提供凭证
2. 使用 Web 联合身份验证
3. 在脚本中硬编码

> **Warning**
>
> 我们不建议在脚本中对您的 Amazon 凭证进行硬编码。硬编码凭证存在暴露您的访问密钥 ID 和秘密访问密钥的风险。

主题

- 使用 Amazon Cognito Identity 验证用户身份
- 使用 Web 联合身份验证来验证用户身份
- Web 联合身份验证示例

使用 Amazon Cognito Identity 验证用户身份

获取浏览器脚本 Amazon 凭证的推荐方法是使用 Amazon Cognito Identity 凭证对象 `AWS.CognitoIdentityCredentials`。Amazon Cognito 支持通过第三方身份提供商对用户进行身份验证。

要使用 Amazon Cognito Identity，您必须先在 Amazon Cognito 控制台中创建一个身份池。身份池表示应用程序为用户提供的身份组。为用户提供的身份唯一地标识每个用户账户。Amazon Cognito 身份并不是凭证。可以在 Amazon Security Token Service (Amazon STS) 中使用 Web 联合身份验证支持为凭证交换这些身份。

Amazon Cognito 使用 `AWS.CognitoIdentityCredentials` 对象帮助您管理跨多个身份提供商的身份抽象。然后，在 Amazon STS 中为凭证交换加载的身份。

配置 Amazon Cognito Identity 凭证对象

如果您尚未创建身份池，则在配置 `AWS.CognitoIdentityCredentials` 之前，请先创建一个，以与 Amazon Cognito 控制台中的浏览器脚本一起使用。为身份池创建并关联经过身份验证和未经身份验证的 IAM 角色。

未经身份验证的用户的身份未经过验证，因此，该角色很适合您的应用程序的来宾用户或用户身份验证与否无关紧要的情形。经过身份验证的用户可以通过证实其身份的第三方身份提供商登录到您的应用程序。确保您的资源的权限范围适当，让未经身份验证的用户无权访问这些资源。

通过关联的身份提供商配置身份池后，您可以使用 `AWS.CognitoIdentityCredentials` 验证用户身份。要将应用程序凭证配置为使用 `AWS.CognitoIdentityCredentials`，则为 `credentials` 或基于每个服务配置设置 `AWS.Config` 属性。以下示例使用 `AWS.Config`：

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',
  Logins: { // optional tokens, used for authenticated login
    'graph.facebook.com': 'FBTOKEN',
    'www.amazon.com': 'AMAZONTOKEN',
    'accounts.google.com': 'GOOGLETOKEN'
  }
```

```
});
```

可选的 Logins 属性是身份提供商名称到这些提供商身份令牌的映射。您如何从身份提供商获得令牌的方式取决于您使用的提供商。例如，如果 Facebook 是您的身份提供商之一，则您可以使用来自 Facebook 软件开发工具包的 FB.login 函数获取身份提供商令牌：

```
FB.login(function (response) {
  if (response.authResponse) { // logged in
    AWS.config.credentials = new AWS.CognitoIdentityCredentials({
      IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',
      Logins: {
        'graph.facebook.com': response.authResponse.accessToken
      }
    });

    s3 = new AWS.S3; // we can now create our service object

    console.log('You are now logged in.');
  } else {
    console.log('There was a problem logging you in.');
  }
});
```

将未经身份验证的用户切换为经过身份验证的用户

Amazon Cognito 同时支持经过身份验证的用户和未经身份验证的用户。即使未经身份验证的用户不通过任何身份提供商登录，这些用户也有权访问您的资源。此级别的访问可用于向尚未登录的用户显示内容。即使每个未经身份验证的用户尚未单独登录和经过身份验证，这些用户在 Amazon Cognito 中也都具有唯一的身份。

最初未经身份验证的用户

用户通常从未经身份验证的角色开始，为此需要设置配置对象的凭证属性而不是 Logins 属性。在这种情况下，您的默认配置可能如下所示：

```
// set the default config object
var creds = new AWS.CognitoIdentityCredentials({
 IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030'
});
AWS.config.credentials = creds;
```

切换为经过身份验证的用户

当未经身份验证的用户登录身份提供商并且您拥有令牌时，您可以通过调用可更新凭证对象和添加 Logins 令牌的自定义函数，来将用户从未经身份验证的用户切换为经过身份验证的用户：

```
// Called when an identity provider has a token for a logged in user
function userLoggedIn(providerName, token) {
  creds.params.Logins = creds.params.Logins || {};
  creds.params.Logins[providerName] = token;

  // Expire credentials to refresh them on the next request
  creds.expired = true;
}
```

您还可以创建 CognitoIdentityCredentials 对象。如果这样做，则必须重置您创建的现有服务对象的凭证属性。仅在对象初始化时从全局配置读取服务对象。

有关 CognitoIdentityCredentials 对象的更多信息，请参阅 适用于 JavaScript 的 Amazon SDK API 参考中的 AWS.CognitoIdentityCredentials。

使用 Web 联合身份验证来验证用户身份

您可以使用 Web 身份联合验证直接配置各个身份提供商以访问 Amazon 资源。Amazon 目前支持通过多个身份提供商使用 Web 身份联合验证来验证用户身份：

- Login with Amazon
- Facebook 登录
- Google 登录

您必须首先向您的应用程序支持的提供商注册您的应用程序。接下来，创建 IAM 角色并为其设置权限。然后，您创建的 IAM 角色将通过相应的身份提供商授予您为其配置的权限。例如，您可以设置一个角色，允许通过 Facebook 登录的用户对您控制的特定 Amazon S3 桶具有读取权限。

有了配置了权限的 IAM 角色以及向所选身份提供商注册的应用程序之后，您可以将 SDK 设置为使用帮助程序代码获取 IAM 角色的凭证，如下所示：

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
   RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>/:role/<WEB_IDENTITY_ROLE_NAME>',
   ProviderId: 'graph.facebook.com|www.amazon.com', // this is null for Google
```

```
    WebIdentityToken: ACCESS_TOKEN
});
```

ProviderId 参数中的值取决于指定的身份提供商。WebIdentityToken 参数中的值是从使用身份提供商成功登录时检索的访问令牌。有关如何为每个身份提供商配置和检索访问令牌的更多信息，请参阅身份提供商的相关文档。

步骤 1：向身份提供商注册

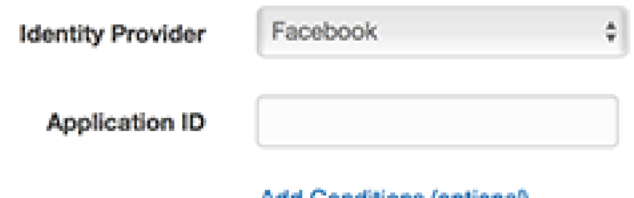
首先，向您选择支持的身份提供商注册应用程序。系统会要求您提供信息来标识您的应用程序以及作者（在可能的情况下）。这可以确保身份提供商知道谁正在接收他们的用户信息。在每种情况下，身份提供商将发出用于配置用户角色的应用程序 ID。

步骤 2：为身份提供商创建 IAM 角色

从身份提供商获取应用程序 ID 后，转到 IAM 控制台（地址：https://console.aws.amazon.com/iam/）以创建新的 IAM 角色。

为身份提供商创建 IAM 角色

1. 转到控制台的角色部分，然后选择创建新角色。
2. 键入新角色的名称，以帮助您跟踪其使用情况，例如 **facebookIdentity**，然后选择下一步。
3. 在选择角色类型中，选择用于身份提供商访问的角色。
4. 对于授予 Web 身份提供商访问权限，选择选择。
5. 在身份提供商列表中，选择要为此 IAM 角色使用的身份提供商。



6. 在应用程序 ID 中键入身份提供商提供的应用程序 ID，然后选择下一步。
7. 配置要公开的资源的权限，以允许访问特定资源上的特定操作。有关 IAM 权限的更多信息，请参阅《IAM 用户指南》中的 Amazon IAM 权限概述。查看并根据需要自定义角色的信任关系，然后选择下一步。
8. 附加您需要的其他政策，然后选择下一步。有关 IAM policy 的更多信息，请参阅《IAM 用户指南》中的 IAM policy 概述。

9. 检查新角色，然后选择创建角色。

您可以为角色提供其他约束，例如将其限定为特定用户 ID。如果角色授予对资源的写入权限，请确保正确地将角色限定为具有正确权限的用户，否则具有 Amazon、Facebook 或 Google 身份的任何用户都将能够修改应用程序中的资源。

有关在 IAM 中使用 Web 身份联合验证的更多信息，请参阅《IAM 用户指南》中的关于 Web 身份联合验证。

步骤 3：登录后获取提供商访问令牌

使用身份提供商的开发工具包为您的应用程序设置登录操作。您可以使用 OAuth 或 OpenID 从身份提供商下载并安装允许用户登录的 JavaScript 开发工具包。有关如何在应用程序中下载和设置开发工具包代码的信息，请参阅身份提供商的开发工具包文档：

- Login with Amazon
- Facebook 登录
- Google 登录

步骤 4：获取临时凭证

在配置应用程序、角色和资源权限后，将代码添加到应用程序以获取临时凭证。通过 Amazon Security Token Service 使用 Web 联合身份验证提供这些凭证。用户登录到身份提供商，该提供商返回访问令牌。针对您为此身份提供者创建的 IAM 角色，使用 ARN 设置 `AWS.WebIdentityCredentials` 对象：

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
    RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>',
    ProviderId: 'graph.facebook.com|www.amazon.com', // Omit this for Google
    WebIdentityToken: ACCESS_TOKEN // Access token from identity provider
});
```

随后创建的服务对象将具有适当的凭证。在设置 `AWS.config.credentials` 属性之前创建的对象将不具有当前凭证。

您还可以在检索访问令牌之前创建 `AWS.WebIdentityCredentials`。这允许您在加载访问令牌之前创建依赖于凭证的服务对象。为此，请在不使用 `WebIdentityToken` 参数的情况下创建凭证对象：

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
```

```
  RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>',
  ProviderId: 'graph.facebook.com|www.amazon.com' // Omit this for Google
});

// Create a service object
var s3 = new AWS.S3;
```

然后，在包含访问令牌的身份提供商开发工具包的回调中设置 `WebIdentityToken`：

```
AWS.config.credentials.params.WebIdentityToken = accessToken;
```

Web 联合身份验证示例

以下是使用 Web 联合身份验证在浏览器 JavaScript 中获取凭证的几个示例。必须从 http:// 或 https:// 主机模式运行这些示例，以确保身份提供商可以重定向到您的应用程序。

Login with Amazon 示例

以下代码显示如何将 Login as Amazon 用作身份提供商。

```
<a href="#" id="login">
  <img border="0" alt="Login with Amazon"
    src="https://images-na.ssl-images-amazon.com/images/G/01/lwa/
btnLWA_gold_156x32.png"
    width="156" height="32" />
</a>
<div id="amazon-root"></div>
<script type="text/javascript">
  var s3 = null;
  var clientId = 'amzn1.application-oa2-client.1234567890abcdef'; // client ID
  var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

  window.onAmazonLoginReady = function() {
    amazon.Login.setClientId(clientId); // set client ID

    document.getElementById('login').onclick = function() {
      amazon.Login.authorize({scope: 'profile'}, function(response) {
        if (!response.error) { // logged in
          AWS.config.credentials = new AWS.WebIdentityCredentials({
            RoleArn: roleArn,
            ProviderId: 'www.amazon.com',
            WebIdentityToken: response.access_token
```

```
          });

          s3 = new AWS.S3();

          console.log('You are now logged in.');
        } else {
          console.log('There was a problem logging you in.');
        }
      });
    };
  };

  (function(d) {
    var a = d.createElement('script'); a.type = 'text/javascript';
    a.async = true; a.id = 'amazon-login-sdk';
    a.src = 'https://api-cdn.amazon.com/sdk/login1.js';
    d.getElementById('amazon-root').appendChild(a);
  })(document);
</script>
```

Facebook Login 示例

以下代码显示如何将 Facebook Login 用作身份提供商：

```
<button id="login">Login</button>
<div id="fb-root"></div>
<script type="text/javascript">
var s3 = null;
var appId = '1234567890'; // Facebook app ID
var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

window.fbAsyncInit = function() {
  // init the FB JS SDK
  FB.init({appId: appId});

  document.getElementById('login').onclick = function() {
    FB.login(function (response) {
      if (response.authResponse) { // logged in
        AWS.config.credentials = new AWS.WebIdentityCredentials({
          RoleArn: roleArn,
          ProviderId: 'graph.facebook.com',
          WebIdentityToken: response.authResponse.accessToken
        });
```

```
        s3 = new AWS.S3;

        console.log('You are now logged in.');
      } else {
        console.log('There was a problem logging you in.');
      }
    });
  };
};

// Load the FB JS SDK asynchronously
(function(d, s, id){
   var js, fjs = d.getElementsByTagName(s)[0];
   if (d.getElementById(id)) {return;}
   js = d.createElement(s); js.id = id;
   js.src = "//connect.facebook.net/en_US/all.js";
   fjs.parentNode.insertBefore(js, fjs);
 }(document, 'script', 'facebook-jssdk'));
</script>
```

Google+ Sign-in 示例

以下代码显示如何将 Google+ Sign-in 用作身份提供商。用于来自 Google 的 Web 联合身份验证的访问令牌存储在 `response.id_token` 中，而不是像其他身份提供商一样存储在 `access_token` 中。

```
<span
  id="login"
  class="g-signin"
  data-height="short"
  data-callback="loginToGoogle"
  data-cookiepolicy="single_host_origin"
  data-requestvisibleactions="http://schemas.google.com/AddActivity"
  data-scope="https://www.googleapis.com/auth/plus.login">
</span>
<script type="text/javascript">
  var s3 = null;
  var clientID = '1234567890.apps.googleusercontent.com'; // Google client ID
  var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

  document.getElementById('login').setAttribute('data-clientid', clientID);
  function loginToGoogle(response) {
    if (!response.error) {
      AWS.config.credentials = new AWS.WebIdentityCredentials({
```

```
        RoleArn: roleArn, WebIdentityToken: response.id_token
      });

      s3 = new AWS.S3();

      console.log('You are now logged in.');
    } else {
      console.log('There was a problem logging you in.');
    }
  }

  (function() {
    var po = document.createElement('script'); po.type = 'text/javascript'; po.async =
 true;
    po.src = 'https://apis.google.com/js/client:plusone.js';
    var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(po,
 s);
  })();
 </script>
```

锁定 API 版本

Amazon 服务具有 API 版本号，可用于跟踪 API 兼容性。Amazon 服务中的 API 版本由 YYYY-mm-dd 格式的日期字符串标识。例如，Amazon S3 的当前 API 版本为 2006-03-01。

如果您在生产代码中依赖某项服务的 API 版本，我们建议您锁定该版本。这可以使您的应用程序与开发工具包更新导致的服务更改隔离开来。如果在创建服务对象时未指定 API 版本，则开发工具包默认使用最新的 API 版本。这可能会导致您的应用程序使用对自身产生负面影响的更改来引用更新的 API。

要锁定用于某项服务的 API 版本，请在构造服务对象时传递 `apiVersion` 参数。在以下示例中，新创建的 `AWS.DynamoDB` 服务对象锁定到 `2011-12-05` API 版本：

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2011-12-05'});
```

您可以通过在 `apiVersions` 中指定 `AWS.Config` 参数来全局配置一组服务 API 版本。例如，要设置特定版本的 DynamoDB 和 Amazon EC2 API 以及当前 Amazon Redshift API，请按如下所示设置 `apiVersions`：

```
AWS.config.apiVersions = {
```

```
  dynamodb: '2011-12-05',
  ec2: '2013-02-01',
  redshift: 'latest'
};
```

获取 API 版本

要获取某项服务的 API 版本，请参阅该服务参考页面上的“Locking the API Version”部分（例如 Amazon S3 的 https://docs.amazonaws.cn/AWSJavaScriptSDK/latest/AWS/S3.html）。

Node.js 注意事项

虽然 Node.js 代码是 JavaScript，但在 Node.js 中使用 适用于 JavaScript 的 Amazon SDK 与在浏览器脚本中使用 SDK 有所不同。一些 API 方法在 Node.js 中有效，但在浏览器脚本以及其他方法中不起作用。成功使用某些 API 取决于您对常见 Node.js 代码编写模式的熟悉程度，例如导入和使用其他 Node.js 模块，如 `File System (fs)` 模块。

使用内置 Node.js 模块

Node.js 提供了一组内置模块，无需安装即可使用它们。要使用这些模块，请使用 `require` 方法创建一个对象以指定模块名称。例如，要包含内置的 HTTP 模块，请使用以下方法。

```
var http = require('http');
```

调用模块的方法，就好像它们是该对象的方法一样。例如，下面的代码读取您的 HTML 文件。

```
// include File System module
var fs = require('fs');
// Invoke readFile method
fs.readFile('index.html', function(err, data) {
  if (err) {
    throw err;
  } else {
    // Successful file read
  }
});
```

有关 Node.js 提供的所有内置模块的完整列表，请参阅 Node.js 网站上的 Node.js v6.11.1 文档。

使用 NPM 程序包

除了内置模块，您还可以包含并合并来自 npm（即 Node.js 程序包管理器）的第三方代码。这是一个开源 Node.js 程序包的存储库和一个用于安装这些程序包的命令行界面。有关 npm 和当前可用程序包列表的更多信息，请参阅 https://www.npmjs.com。您还可以在此处了解可在 GitHub 上使用的其他 Node.js 程序包。

可与适用于 JavaScript 的 Amazon SDK结合使用的 npm 包的一个示例是 `browserify`。有关更多信息，请参阅 使用 Browserify 构建开发工具包作为依赖关系。另一个示例是 `webpack`。有关更多信息，请参阅 使用 Webpack 捆绑应用程序。

主题

- 在 Node.js 中配置 maxSockets
- 在 Node.js 中重复使用具有保持连接功能的连接
- 配置 Node.js 的代理
- 在 Node.js 中注册证书包

在 Node.js 中配置 maxSockets

在 Node.js 中，您可以设置每个源的最大连接数。如果设置了 `maxSockets`，则低级 HTTP 客户端会将请求排队，并在它们可用时将它们分配给套接字。

这使您可以设置在某个时间对给定源的并发请求数的上限。降低此值可以减少收到的限制或超时错误的数量。但是，它还会增加内存使用量，因为请求进行排队，直到套接字变为可用状态。

以下示例说明如何为您创建的所有服务对象设置 `maxSockets`。此示例针对每个服务终端节点最多允许 25 个并发连接。

```
var AWS = require('aws-sdk');
var https = require('https');
var agent = new https.Agent({
   maxSockets: 25
});

AWS.config.update({
   httpOptions:{
      agent: agent
   }
});
```

每项服务都可以这样做。

```
var AWS = require('aws-sdk');
var https = require('https');
var agent = new https.Agent({
   maxSockets: 25
});

var dynamodb = new AWS.DynamoDB({
   apiVersion: '2012-08-10'
   httpOptions:{
      agent: agent
   }
});
```

当使用默认值 `https` 时，SDK 从 `globalAgent` 获取 `maxSockets` 值。如果未定义 `maxSockets` 值，或者该值是 `Infinity`，则开发工具包采用 `maxSockets` 值 50。

有关在 Node.js 中设置 `maxSockets` 的更多信息，请参阅 Node.js 在线文档。

在 Node.js 中重复使用具有保持连接功能的连接

默认情况下，默认的 Node.js HTTP/HTTPS 代理会为每个新请求创建一个新的 TCP 连接。为了节省建立新连接的成本，您可以重复使用现有的连接。

对于短期操作（如 DynamoDB 查询），设置 TCP 连接的延迟开销可能大于操作本身。此外，由于 DynamoDB 静态加密已与 Amazon KMS 集成，您可能会遇到延迟，因为数据库必须为每个操作重新建立新的 Amazon KMS 缓存条目。

配置 SDK for JavaScript 以重复使用 TCP 连接的最简单方法是将 `AWS_NODEJS_CONNECTION_REUSE_ENABLED` 环境变量设置为 `1`。2.463.0 版本中已添加此功能。

或者，您可以将 HTTP 或 HTTPS 代理的 `keepAlive` 属性设置为 `true`，如以下示例所示。

```
const AWS = require('aws-sdk');
// http or https
const http = require('http');
const agent = new http.Agent({
  keepAlive: true,
// Infinity is read as 50 sockets
  maxSockets: Infinity
});
```

```
AWS.config.update({
  httpOptions: {
    agent
  }
});
```

以下示例演示了如何仅为 DynamoDB 客户端设置 `keepAlive`：

```
const AWS = require('aws-sdk')
// http or https
const https = require('https');
const agent = new https.Agent({
  keepAlive: true
});

const dynamodb = new AWS.DynamoDB({
  httpOptions: {
    agent
  }
});
```

如果已启用 `keepAlive`，您还可以使用 `keepAliveMsecs` 设置 TCP Keep-Alive 数据包的初始延迟，默认值为 1000ms。有关详细信息，请参阅 Node.js 文档。

配置 Node.js 的代理

如果您无法直接连接到 Internet，则 SDK for JavaScript 支持通过第三方 HTTP 代理（例如 proxy-agent）使用 HTTP 或 HTTPS 代理。要安装 proxy-agent，请在命令行中键入以下内容。

```
npm install proxy-agent --save
```

如果您决定使用其他代理，请首先按照该代理的安装和配置说明进行操作。要在应用程序中使用此代理或其他第三方代理，必须设置 `httpOptions` 的 `AWS.Config` 属性以指定您选择的代理。此示例显示了 `proxy-agent`。

```
var AWS = require("aws-sdk");
var ProxyAgent = require('proxy-agent').ProxyAgent;
AWS.config.update({
  httpOptions: { agent: new ProxyAgent('http://internal.proxy.com') }
});
```

有关其他代理库的更多信息，请参阅 Node.js 程序包管理器 (npm)。

在 Node.js 中注册证书包

Node.js 的默认信任存储包含访问 Amazon 服务所需的证书。在某些情况下，最好只包括一组特定的证书。

在本示例中，使用磁盘上的特定证书创建 `https.Agent`，除非提供指定的证书，否则它会拒绝连接。然后，系统会使用新创建的 `https.Agent` 更新开发工具包配置。

```
var fs = require('fs');
var https = require('https');
var certs = [
  fs.readFileSync('/path/to/cert.pem')
];

AWS.config.update({
  httpOptions: {
    agent: new https.Agent({
      rejectUnauthorized: true,
      ca: certs
    })
  }
});
```

浏览器脚本注意事项

以下主题介绍了在浏览器脚本中使用 适用于 JavaScript 的 Amazon SDK 的特殊注意事项。

主题

- 为浏览器构建 SDK
- 跨源资源共享 (CORS)

为浏览器构建 SDK

SDK for JavaScript 作为 JavaScript 文件提供，其中包含对一组默认服务的支持。此文件通常使用引用托管开发工具包的 `<script>` 标记加载到浏览器脚本中。但是，您可能需要支持默认服务集以外的服务，或者需要自定义开发工具包。

如果您在强制浏览器使用 CORS 的环境之外使用 SDK，并且您希望访问 SDK for JavaScript 提供的所有服务，则可以通过克隆存储库并运行相同的构建工具（这些工具构建了 SDK 的默认托管版本）在本地构建 SDK 的自定义副本。下面几部分介绍使用额外服务和 API 版本构建 SDK 的步骤。

主题

- 使用 SDK 生成器构建 SDK for JavaScript
- 使用 CLI 构建 SDK for JavaScript
- 构建特定服务和 API 版本
- 使用 Browserify 构建开发工具包作为依赖关系

使用 SDK 生成器构建 SDK for JavaScript

创建您自己的 适用于 JavaScript 的 Amazon SDK 构建的最简单方法是在 https://sdk.amazonaws.com/builder/js 上使用 SDK 生成器 Web 应用程序。使用开发工具包构建器指定要包含在构建中的服务及其 API 版本。

选择选择所有服务或选择默认服务作为您可以添加或删除服务的起点。选择开发以获得更易读的代码，或选择简化以创建要部署的简化构建。选择要包含的服务和版本后，选择构建以构建和下载您的自定义开发工具包。

使用 CLI 构建 SDK for JavaScript

要使用 Amazon CLI 构建 SDK for JavaScript，需要首先克隆包含 SDK 源代码的 Git 存储库。您必须在计算机上安装 Git 和 Node.js。

首先，从 GitHub 克隆存储库并将目录更改为以下目录：

```
git clone https://github.com/aws/aws-sdk-js.git
cd aws-sdk-js
```

克隆存储库后，下载开发工具包和构建工具的依赖模块：

```
npm install
```

您现在可以构建开发工具包的打包版本。

从命令行构建

生成器工具位于 `dist-tools/browser-builder.js` 中。通过键入以下内容运行此脚本：

```
node dist-tools/browser-builder.js > aws-sdk.js
```

此命令会构建 aws-sdk.js 文件。此文件未压缩。默认情况下，此软件包仅包含一组标准服务。

简化构建输出

为了减少通过网络传输的数据量，可以通过称为简化 的过程压缩 JavaScript 文件。简化过程会去掉注释、不必要的空格以及便于阅读但不影响代码执行的其他字符。生成器工具可以生成未压缩的输出或简化的输出。要简化构建输出，请设置 `MINIFY` 环境变量：

```
MINIFY=1 node dist-tools/browser-builder.js > aws-sdk.js
```

构建特定服务和 API 版本

您可以选择要在开发工具包中构建的服务。要选择服务，请将逗号分隔的服务名称指定为参数。例如，要仅构建 Amazon S3 和 Amazon EC2，请使用以下命令：

```
node dist-tools/browser-builder.js s3,ec2 > aws-sdk-s3-ec2.js
```

您还可以通过在服务名称后添加版本名称来选择服务构建的特定 API 版本。例如，要构建 Amazon DynamoDB 的这两个 API 版本，请使用以下命令：

```
node dist-tools/browser-builder.js dynamodb-2011-12-05,dynamodb-2012-08-10
```

服务标识符和 API 版本在 https://github.com/aws/aws-sdk-js/tree/master/apis 上的特定于服务的配置文件中提供。

构建所有服务

您可以通过包括 `all` 参数来构建所有服务和 API 版本：

```
node dist-tools/browser-builder.js all > aws-sdk-full.js
```

构建特定服务

要自定义构建中包含的选定服务集，请将 `AWS_SERVICES` 环境变量传递给包含所需服务列表的 Browserify 命令。以下示例将构建 Amazon EC2、Amazon S3 和 DynamoDB 服务。

```
$ AWS_SERVICES=ec2,s3,dynamodb browserify index.js > browser-app.js
```

使用 Browserify 构建开发工具包作为依赖关系

Node.js 有一个基于模块的机制，可用于包含来自第三方开发人员的代码和功能。Web 浏览器中运行的 JavaScript 本身不支持此模块化方法。不过，利用名为 Browserify 的工具，您可以使用 Node.js 模块方法并在浏览器中使用为 Node.js 编写的模块。Browserify 将浏览器脚本的模块依赖项构建为可在浏览器中使用的单个自包含 JavaScript 文件。

您可以使用 Browserify 将开发工具包构建为任何浏览器脚本的库依赖项。例如，以下 Node.js 代码需要开发工具包：

```
var AWS = require('aws-sdk');
var s3 = new AWS.S3();
s3.listBuckets(function(err, data) { console.log(err, data); });
```

可以使用 Browserify 将此示例代码编译为与浏览器兼容的版本：

```
$ browserify index.js > browser-app.js
```

然后，应用程序（包括其开发工具包依赖项）可通过 `browser-app.js` 在浏览器中使用。

有关 Browserify 的更多信息，请参阅 Browserify 网站。

跨源资源共享 (CORS)

跨源资源共享（即 CORS）是一项现代 Web 浏览器的安全功能。它使得 Web 浏览器可以协商哪些域能够发出对外部网站或服务的请求。在使用 适用于 JavaScript 的 Amazon SDK 开发浏览器应用程序时，CORS 是一个重要的考虑因素，因为对资源的大部分请求发送到外部域，例如 Web 服务的端点。如果您的 JavaScript 环境实施 CORS 安全性，则必须对该服务配置 CORS。

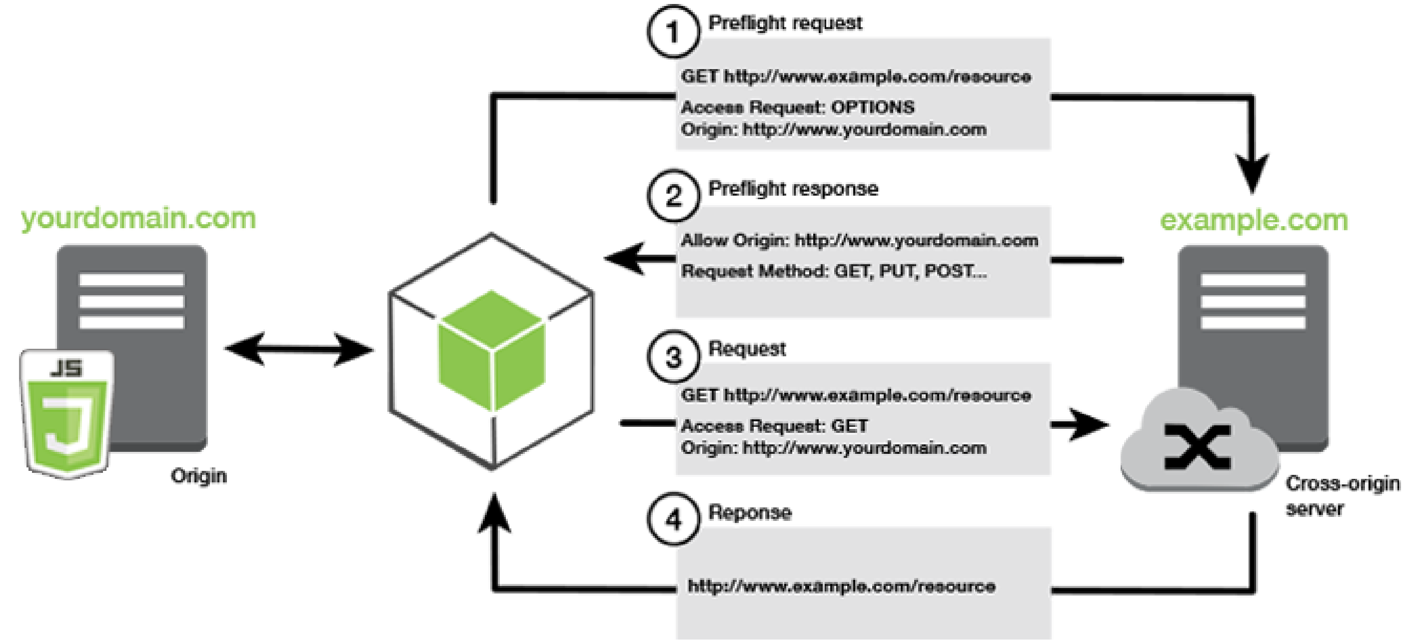
CORS 根据以下条件，确定是否允许跨源请求中的资源共享：

- 发出请求的特定域
- 发出的 HTTP 请求的类型（GET、PUT、POST、DELETE 等等）

CORS 工作原理

在最简单的情况下，浏览器脚本从其他域中的服务器发出对某个资源的 GET 请求。根据该服务器的 CORS 配置，如果请求来自已授权提交 GET 请求的域，则跨来源服务器通过返回请求的资源做出响应。

如果请求域或者 HTTP 请求的类型未获得授权，则将拒绝请求。但是，CORS 实现了在实际提交请求之前进行预检。在这种情况下将提交预检请求，在其中发送 OPTIONS 访问请求操作。如果跨来源服务器的 CORS 配置授予对请求域的访问权限，则服务器发送回预检响应，其中列出请求域可以对所请求资源发出的所有 HTTP 请求类型。



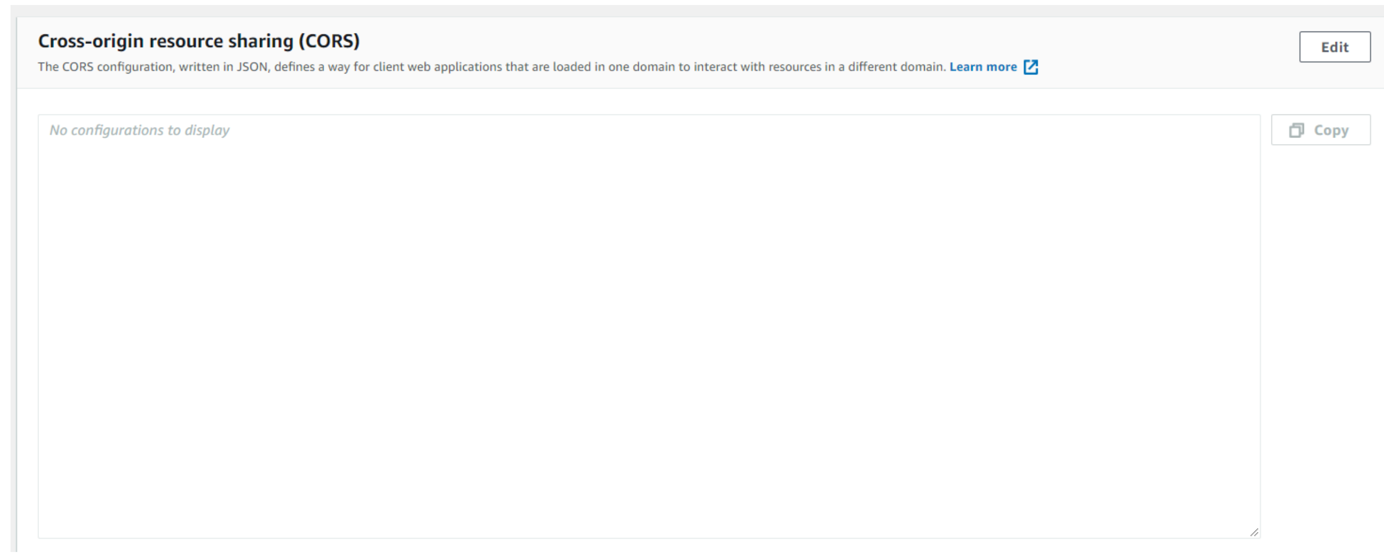
是否需要 CORS 配置

Amazon S3 桶需要 CORS 配置，然后才能在桶上执行操作。在某些 JavaScript 环境中，CORS 可能未实施，因此不需要配置 CORS。例如，如果您在 Amazon S3 桶中托管应用程序并访问 `*.s3.amazonaws.com` 或某个其它特定端点的资源，您的请求不会访问外部域。因此，此配置不需要 CORS。在这种情况下，Amazon S3 之外的服务仍使用 CORS。

为 Amazon S3 桶配置 CORS

您可以在 Amazon S3 控制台中配置 Amazon S3 桶，以使用 CORS。

1. 在 Amazon S3 控制台中，选择您要编辑的桶。
2. 选择权限选项卡，然后向下滚动到跨源资源共享（CORS）面板。



3. 在 CORS 配置编辑器中，选择编辑，键入您的 CORS 配置，然后选择保存。

CORS 配置是一个 XML 文件，在 `<CORSRule>` 中包含了一系列规则。一个配置最多可以有 100 个规则。规则由以下标签之一定义：

- `<AllowedOrigin>`，指定您允许发出跨域请求的域源。
- `<AllowedMethod>`，指定您允许在跨域请求中使用的请求类型（GET、PUT、POST、DELETE、HEAD）。
- `<AllowedHeader>`，指定预检请求中允许的标头。

有关配置示例，请参阅《Amazon Simple Storage Service 用户指南》中的[如何在我的桶上配置 CORS？](#)

CORS 配置示例

以下 CORS 配置示例允许用户从域 `example.org` 查看、添加、删除或更新存储桶中的对象，不过建议您将 `<AllowedOrigin>` 的范围限定为您网站的域。您可以指定 `"*"` 以允许任意源。

Important

在新的 S3 控制台中，CORS 配置必须是 JSON。

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>https://example.org</AllowedOrigin>
    <AllowedMethod>HEAD</AllowedMethod>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
    <ExposeHeader>ETag</ExposeHeader>
    <ExposeHeader>x-amz-meta-custom-header</ExposeHeader>
  </CORSRule>
</CORSConfiguration>
```

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
        "AllowedOrigins": [
            "https://www.example.org"
        ],
        "ExposeHeaders": [
             "ETag",
             "x-amz-meta-custom-header"]
    }
]
```

此配置不授权用户在存储桶上执行操作。它使浏览器的安全模型允许对 Amazon S3 的请求。必须通过存储桶权限或 IAM 角色权限来配置权限。

您可以使用 ExposeHeader，让 SDK 读取从 Amazon S3 返回的响应标头。例如，如果您希望从 PUT 或分段上传读取 ETag 标头，则需要在配置中包括 ExposeHeader 标签，如上例中所示。SDK 只能访问通过 CORS 配置公开的标头。如果您在对象上设置元数据，则将值作为标头返回并带有 x-amz-meta- 前缀，例如 x-amz-meta-my-custom-header，并且也必须通过相同的方式公开。

使用 Webpack 捆绑应用程序

浏览器脚本或 Node.js 中使用代码模块的 Web 应用程序会创建依赖关系。这些代码模块可能会具有自身的依赖关系，导致您的应用程序需要一组互连的模块才能正常工作。要管理依赖关系，您可以使用 webpack 等模块捆绑程序。

Webpack 模块捆绑程序解析您的应用程序代码，搜索 import 或 require 语句，创建包含您应用程序所需的全部资产的捆绑，这样可以轻松地通过网页提供资产服务。SDK for JavaScript 可以作为包括在输出捆绑中的依赖项之一包括在 webpack 中。

有关 webpack 的更多信息，请参阅 GitHub 上的 webpack 模块捆绑程序。

安装 Webpack

要安装 webpack 模块捆绑程序，您必须已经安装了 npm（Node.js 程序包管理器）。键入以下命令以安装 webpack CLI 和 JavaScript 模块。

```
npm install webpack
```

您可能还需要安装允许它加载 JSON 文件的 webpack 插件。键入以下命令，安装 JSON 加载程序插件。

```
npm install json-loader
```

配置 Webpack

默认情况下，webpack 在项目的根目录中搜索名为 webpack.config.js 的 JavaScript 文件。此文件指定您的配置选项。下面是 webpack.config.js 配置文件的示例。

```
// Import path for resolving file paths
var path = require('path');
module.exports = {
  // Specify the entry point for our app.
```

```
  entry: [
    path.join(__dirname, 'browser.js')
  ],
  // Specify the output file containing our bundled code
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    /**
      * Tell webpack how to load 'json' files.
      * When webpack encounters a 'require()' statement
      * where a 'json' file is being imported, it will use
      * the json-loader.
      */
    loaders: [
      {
        test: /\.json$/,
        loaders: ['json']
      }
    ]
  }
}
```

在本示例中，指定 `browser.js` 为入口点。入口点 是 webpack 开始搜索导入的模块所用的文件。输出的文件名指定为 `bundle.js`。此输出文件包含应用程序运行所需的全部 JavaScript。如果入口点中指定的代码导入或需要其它模块（例如 SDK for JavaScript），则将捆绑该代码而无需在配置中指定它。

之前安装的 `json-loader` 插件中的配置指定 webpack 如何导入 JSON 文件。默认情况下，Webpack 仅支持 JavaScript，不过可使用加载程序来添加对导入其他文件类型的支持。由于 SDK for JavaScript 广泛使用 JSON 文件，如果未包括 `json-loader`，webpack 在生成捆绑时会引发错误。

运行 Webpack

要生成应用程序以使用 webpack，请将以下内容添加到您 `package.json` 文件的 `scripts` 对象。

```
"build": "webpack"
```

此处是演示添加 webpack 的示例 `package.json`。

```
{
  "name": "aws-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "aws-sdk": "^2.6.1"
  },
  "devDependencies": {
    "json-loader": "^0.5.4",
    "webpack": "^1.13.2"
  }
}
```

要生成应用程序，请键入以下命令。

```
npm run build
```

随后，webpack 模块捆绑程序生成您在项目的根目录中指定的 JavaScript 文件。

使用 Webpack 捆绑

要在浏览器脚本中使用捆绑，您可以使用 `<script>` 标签整合捆绑，如下例中所示。

```
<!DOCTYPE html>
<html>
    <head>
        <title>AWS SDK with webpack</title>
    </head>
    <body>
        <div id="list"></div>
        <script src="bundle.js"></script>
    </body>
</html>
```

导入单独的服务

webpack 的优势之一是，它解析您代码中的依赖关系并仅捆绑应用程序所需的代码。如果您使用 SDK for JavaScript，则仅捆绑您应用程序实际需要的部分 SDK 可以显著减少 webpack 输出的大小。

请考虑以下用于创建 Amazon S3 服务对象的代码示例。

```
// Import the AWS SDK
var AWS = require('aws-sdk');

// Set credentials and Region
// This can also be done directly on the service client
AWS.config.update({region: 'us-west-1', credentials: {YOUR_CREDENTIALS}});

var s3 = new AWS.S3({apiVersion: '2006-03-01'});
```

`require()` 函数指定整个开发工具包。使用此代码生成的 webpack 捆绑将包括完整 SDK，但如果仅使用 Amazon S3 客户端类，则不需要完整 SDK。如果只包括 Amazon S3 服务所需的部分 SDK，捆绑的大小将大幅减小。甚至设置配置也不需要完整 SDK，因为您可以在 Amazon S3 服务对象上设置配置数据。

以下是在仅包括 SDK 的 Amazon S3 部分时，相同代码的情况。

```
// Import the Amazon S3 service client
var S3 = require('aws-sdk/clients/s3');

// Set credentials and Region
var s3 = new S3({
    apiVersion: '2006-03-01',
    region: 'us-west-1',
    credentials: {YOUR_CREDENTIALS}
  });
```

适用于 Node.js 的捆绑

您可以通过在配置中将其指定为目标，使用 webpack 生成在 Node.js 中运行的捆绑。

```
target: "node"
```

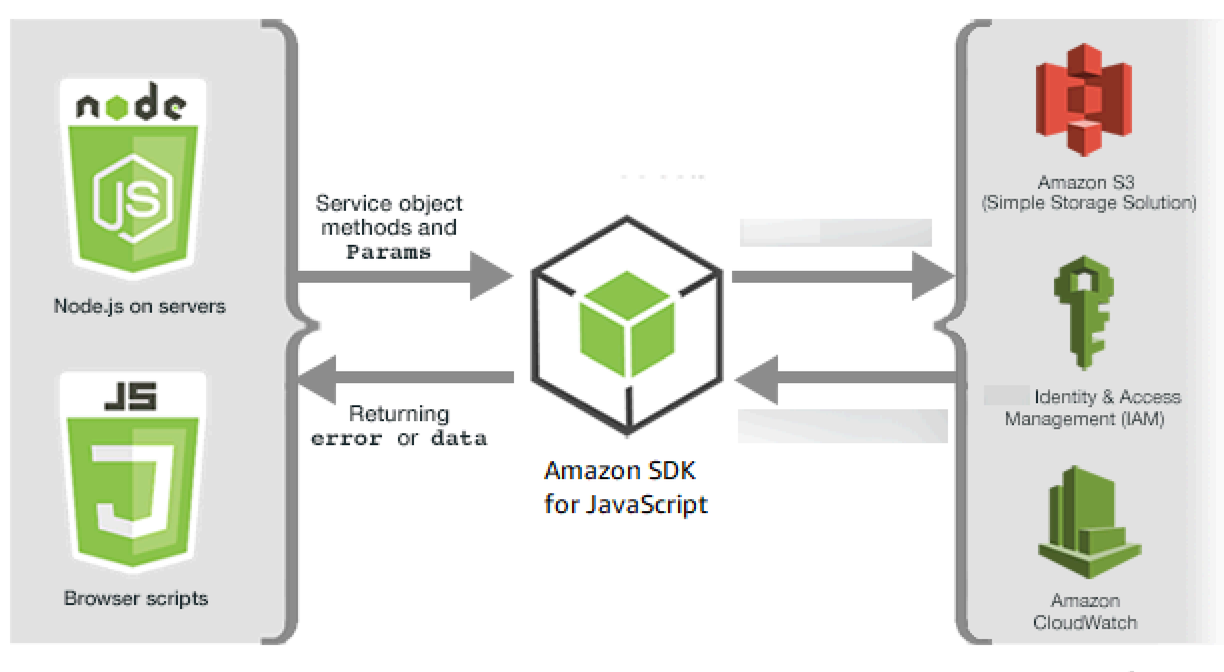
在磁盘空间有限的环境中运行 Node.js 应用程序时，这非常有用。此处是将 Node.js 指定为输出目标的示例 `webpack.config.js` 配置。

```
// Import path for resolving file paths
var path = require('path');
module.exports = {
  // Specify the entry point for our app
  entry: [
    path.join(__dirname, 'node.js')
  ],
  // Specify the output file containing our bundled code
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  // Let webpack know to generate a Node.js bundle
  target: "node",
  module: {
    /**
      * Tell webpack how to load JSON files.
      * When webpack encounters a 'require()' statement
      * where a JSON file is being imported, it will use
      * the json-loader
      */
    loaders: [
      {
        test: /\.json$/,
        loaders: ['json']
      }
    ]
  }
}
```

使用 SDK for JavaScript 中的服务

适用于 JavaScript 的 Amazon SDK通过一组客户端类提供对所支持服务的访问。从这些客户端类，您可以创建服务接口对象，这些对象通常称为服务对象。所支持的每个 Amazon 服务有一个或多个客户端类，提供低级别 API 来使用服务功能和资源。例如，Amazon DynamoDB API 通过 `AWS.DynamoDB` 类提供。

通过 SDK for JavaScript 公开的服务采用请求/响应模式与调用应用程序交换消息。在此模式中，调用服务的代码向服务的端点提交 HTTP/HTTPS 请求。请求中包含成功调用特定功能所需的参数。调用的服务将生成发送回请求方的响应。如果操作成功，则响应包含数据，如果操作不成功，则包含错误消息。



调用 Amazon 服务包括服务对象操作的完整请求和响应生命周期，包含所执行的任何重试。请求由 `AWS.Request` 对象封装在开发工具包中。响应由 `AWS.Response` 对象封装在开发工具包中，通过多种技术之一提供给请求方，例如回调函数或 JavaScript promise。

主题

- 创建和调用服务对象
- 记录适用于 JavaScript 的 Amazon SDK调用
- 异步调用服务
- 使用响应对象

- 使用 JSON
- 在适用于 JavaScript 的 Amazon SDK v2 中重试策略

创建和调用服务对象

JavaScript API 支持大多数可用的 Amazon 服务。JavaScript API 中的各个服务类提供了对其服务中各个 API 调用的访问。有关 JavaScript API 中服务类、操作和参数的更多信息，请参阅 API 参考。

在 Node.js 中使用开发工具包时，您使用 `require` 将开发工具包添加到应用程序，这为所有当前服务提供支持。

```
var AWS = require('aws-sdk');
```

在将 SDK 与浏览器 JavaScript 结合使用时，您使用 AWS 托管的 SDK 包，将 SDK 包加载到浏览器脚本。要加载开发工具包，请添加以下 `<script>` 元素：

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></script>
```

要查找当前的 SDK_VERSION_NUMBER，请参阅 适用于 JavaScript 的 Amazon SDK API Reference Guide 中适用于 SDK for JavaScript 的 API 参考。

默认的托管 SDK 包提供对可用 Amazon 服务中部分服务的支持。有关浏览器的托管开发工具包中默认服务的列表，请参阅 API 参考中的支持的服务。如果已经禁用了 CORS 安全检查，则可以将开发工具包用于其他服务。在这种情况下，您可以生成自定义版本的开发工具包以包含所需的其他服务。有关生成自定义版本的开发工具包的更多信息，请参阅为浏览器构建 SDK。

要求单个服务

要求 SDK for JavaScript 如前所示将整个 SDK 包括到您的代码中。或者，您可以选择仅要求代码所使用的单个服务。请考虑用于创建 Amazon S3 服务对象的下列代码。

```
// Import the AWS SDK
var AWS = require('aws-sdk');

// Set credentials and Region
// This can also be done directly on the service client
AWS.config.update({region: 'us-west-1', credentials: {YOUR_CREDENTIALS}});

var s3 = new AWS.S3({apiVersion: '2006-03-01'});
```

在上例中，require 函数指定整个开发工具包。如果只包含 Amazon S3 服务所需的 SDK 部分，则通过网络传输的代码量以及代码的内存开销都会显著减少。如果需要单独的服务，请调用 require 函数如下所示，包括全小写的服务构造函数。

```
require('aws-sdk/clients/SERVICE');
```

此处所示是当只包含 SDK 的 Amazon S3 部分时，创建前述 Amazon S3 服务对象的代码。

```
// Import the Amazon S3 service client
var S3 = require('aws-sdk/clients/s3');

// Set credentials and Region
var s3 = new S3({
    apiVersion: '2006-03-01',
    region: 'us-west-1',
    credentials: {YOUR_CREDENTIALS}
  });
```

您仍可以访问全局 Amazon 命名空间而无需将每个服务附加到其上。

```
require('aws-sdk/global');
```

在跨多个单独的服务应用相同配置时，例如为所有服务提供相同的凭证，这是一种非常有用的技术。要求单个服务应减少 Node.js 中的加载时间和内存消耗。在通过 Browserify 或 webpack 等捆绑工具完成后，要求单个服务会生成只有完整大小一部分的开发工具包。对于内存或磁盘空间有限的环境，例如 IoT 设备或在 Lambda 函数中，这会很有帮助。

创建服务对象

要通过 JavaScript API 访问服务功能，您需要先创建服务对象，通过该服务对象来访问由底层客户端类提供的一组功能。通常而言，为每个服务提供有一个客户端类；但是，一些服务会在多个客户端类中划分对其功能的访问。

要使用某个功能，您必须创建提供对该功能访问的类的实例。以下示例演示了从 AWS.DynamoDB 客户端类为 DynamoDB 创建服务对象。

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
```

默认情况下，服务对象配置了同样用于配置开发工具包的全局设置。但是，您可以使用运行时配置数据来配置服务对象，使其特定于该服务对象。服务特定的配置数据在应用了全局配置设置之后应用。

以下示例中的 Amazon EC2 服务对象是使用针对特定区域的配置创建的，但却使用全局配置。

```
var ec2 = new AWS.EC2({region: 'us-west-2', apiVersion: '2014-10-01'});
```

除了支持应用到单个服务对象的服务特定配置之外，您还可以应用服务特定配置到指定类的所有新创建服务对象。例如，要配置从 Amazon EC2 类创建的所有服务对象使用美国西部（俄勒冈州）（us-west-2）区域，请将以下内容添加到 AWS.config 全局配置对象中。

```
AWS.config.ec2 = {region: 'us-west-2', apiVersion: '2016-04-01'};
```

锁定服务对象的 API 版本

通过在创建对象时指定 apiVersion 选项，您可以将某个服务对象锁定为服务的特定 API 版本。以下示例将创建锁定到特定 API 版本的 DynamoDB 服务对象。

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2011-12-05'});
```

有关锁定服务对象的 API 版本的更多信息，请参阅锁定 API 版本。

指定服务对象参数

调用服务对象的方法时，根据 API 的需要在 JSON 中传递参数。例如，在 Amazon S3 中，要获取指定桶和键的对象，请将以下参数传递到 getObject 方法。有关传递 JSON 参数的更多信息，请参阅使用 JSON。

```
s3.getObject({Bucket: 'bucketName', Key: 'keyName'});
```

有关 Amazon S3 参数的更多信息，请参阅 API 参考中的 Class: AWS.S3。

此外，使用 params 参数创建服务对象时，您可以将值绑定到单独的参数。服务对象的 params 参数的值是一个映射，指定由服务对象定义的一个或多个参数值。以下示例显示 Amazon S3 服务对象的 Bucket 参数，该服务对象绑定到名为 amzn-s3-demo-bucket 的桶。

```
var s3bucket = new AWS.S3({params: {Bucket: 'amzn-s3-demo-bucket'}, apiVersion:
 '2006-03-01' });
```

通过将服务对象绑定到存储桶，s3bucket 服务对象将 amzn-s3-demo-bucket 参数值视为在后续操作中不再需要指定的默认值。在为操作使用对象时，如果参数值不适用，则将忽略任意绑定参数值。在对服务对象进行调用时，您可以通过指定新值来覆盖此绑定参数。

```
var s3bucket = new AWS.S3({ params: {Bucket: 'amzn-s3-demo-bucket'}, apiVersion:
 '2006-03-01' });
s3bucket.getObject({Key: 'keyName'});
// ...
s3bucket.getObject({Bucket: 'amzn-s3-demo-bucket3', Key: 'keyOtherName'});
```

在 API 参考中可以找到各个方法的可用参数的详细信息。

记录适用于 JavaScript 的 Amazon SDK调用

适用于 JavaScript 的 Amazon SDK 具备内置的日志记录程序，因此您可以记录使用 SDK for JavaScript 发出的 API 调用。

要打开日志记录程序并在控制台中输出日志条目，请添加以下语句到代码中。

```
AWS.config.logger = console;
```

以下是日志输出的示例。

```
[AWS s3 200 0.185s 0 retries] createMultipartUpload({ Bucket: 'amzn-s3-demo-logging-
bucket', Key: 'issues_1704' })
```

使用第三方日志记录程序

如果第三方日志记录程序具有 `log()` 或 `write()` 操作以写入日志文件或服务器，则可以使用该日志记录程序。您必须先按照说明安装和设置自定义日志记录程序，然后才能将其与 SDK for JavaScript 结合使用。

logplease 是您可在浏览器脚本或 Node.js 中使用的此类日志记录程序之一。在 Node.js 中，您可以配置 logplease 将日志条目写入日志文件。您还可以将其与 webpack 结合使用。

使用第三方日志记录程序时，请先设置所有选项，然后将日志记录程序分配到 `AWS.Config.logger`。例如，以下内容指定外部日志文件并为 logplease 设置日志记录级别

```
// Require AWS Node.js SDK
const AWS = require('aws-sdk')
// Require logplease
const logplease = require('logplease');
// Set external log file option
logplease.setLogfile('debug.log');
```

```
// Set log level
logplease.setLogLevel('DEBUG');
// Create logger
const logger = logplease.create('logger name');
// Assign logger to SDK
AWS.config.logger = logger;
```

有关 logplease 的更多信息，请参阅 GitHub 上的 logplease 简单 JavaScript 日志记录程序。

异步调用服务

通过 SDK 发出的所有请求均为异步。在编写浏览器脚本时，务必记住这一点。在 Web 浏览器中运行的 JavaScript 通常只有一个执行线程。在对 Amazon 服务进行异步调用之后，浏览器脚本继续运行，并可在该过程中结果返回之前，尝试执行依赖于该异步结果的代码。

对 Amazon 服务进行异步调用包括管理这些调用，使得您的代码不会在数据可用之前尝试使用这些数据。本部分中的主题说明管理异步调用的需求，以及在管理它们时可以使用的具体不同技术。

主题

- 管理异步调用
- 使用匿名回调函数
- 使用请求对象事件侦听器
- 使用异步/等待
- 使用 JavaScript Promise

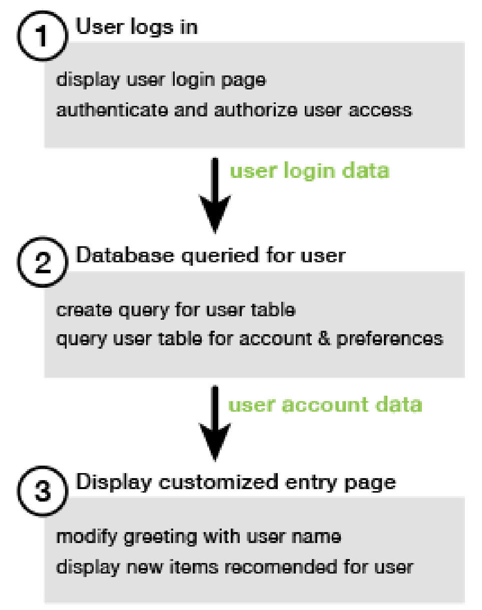
管理异步调用

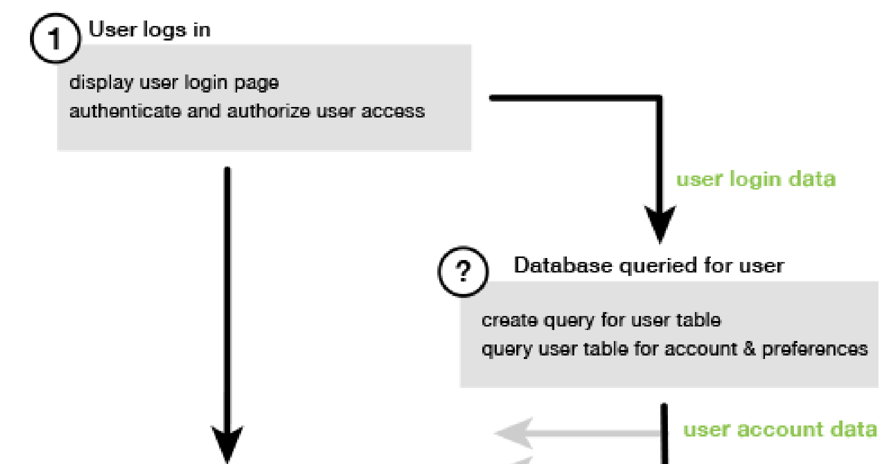
例如，电子商务网站的主页会让返回的客户登录。客户登录可以获得的一部分好处在于，登录之后网站可以根据其特定首选项进行自定义。要做到这一点：

1. 客户必须登录并使用其登录凭证进行验证。
2. 从客户数据库中请求客户的首选项。
3. 数据库提供客户的首选项，这些首选项用于在页面加载之前自定义网站。

如果这些任务同步执行，则必须在每个任务完成之后才能执行下一个任务。在数据库返回客户首选项之前，网页无法完成加载。但是，在数据库查询发送到服务器之后，由于网络瓶颈、极高的数据库流量或者糟糕的移动设备连接，客户数据的接收可能会延迟甚至失败。

要避免网站在这些情况下停滞不前，可以异步调用数据库。数据库调用执行之后，发送您的异步请求，您的代码继续按预期方式执行。如果您未能正确地管理异步调用的响应，代码会在数据尚不可用时，尝试使用预期从数据库返回的信息。





使用匿名回调函数

创建 AWS.Request 对象的各个服务对象方法可以接受匿名回调函数作为最后一个参数。此回调函数的签名为：

```
function(error, data) {
    // callback handling code
}
```

此回调函数在返回成功响应或错误数据时执行。如果方法调用成功，则响应的内容在 data 参数中供回调函数使用。如果调用不成功，则在 error 参数中提供有关失败的详细信息。

通常，回调函数内部的代码经过了错误测试，在返回错误时会进行处理。如果未返回错误，则代码从 data 参数检索响应中的数据。回调函数的基本格式如此例中所示。

```
function(error, data) {
    if (error) {
        // error handling code
```

```
        console.log(error);
    } else {
        // data handling code
        console.log(data);
    }
}
```

在以上示例中，错误的详细信息或者返回的数据记录到控制台中。此处的示例演示了作为对服务对象调用方法的一部分传递的回调函数。

```
new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances(function(error, data) {
  if (error) {
    console.log(error); // an error occurred
  } else {
    console.log(data); // request succeeded
  }
});
```

访问请求和响应对象

在回调函数内部，对于大多数服务，JavaScript 关键字 this 是指底层 AWS.Response 对象。在以下示例中，httpResponse 对象的 AWS.Response 属性在回调函数中用于记录原始响应数据和标头，以帮助调试。

```
new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances(function(error, data) {
  if (error) {
    console.log(error); // an error occurred
    // Using this keyword to access AWS.Response object and properties
    console.log("Response data and headers: " + JSON.stringify(this.httpResponse));
  } else {
    console.log(data); // request succeeded
  }
});
```

此外，由于 AWS.Response 对象具有一个 Request 属性，其中包含由原始方法调用发送的 AWS.Request，您还可以访问所发出请求的详细信息。

使用请求对象事件侦听器

在调用服务对象方法时，如果您未创建并将匿名回调函数作为参数传递，则方法调用生成 AWS.Request 对象，该对象必须使用其 send 方法手动设置。

要处理响应，您必须为 AWS.Request 对象创建一个事件侦听器，以便为方法调用注册回调函数。以下示例演示如何创建 AWS.Request 对象用于调用服务对象方法，以及创建针对成功返回的事件侦听器。

```
// create the AWS.Request object
var request = new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances();

// register a callback event handler
request.on('success', function(response) {
  // log the successful data response
  console.log(response.data);
});

// send the request
request.send();
```

在调用了 send 上的 AWS.Request 方法之后，当服务对象收到 AWS.Response 对象时，事件处理程序执行。

有关 AWS.Request 对象的更多信息，请参阅 API 参考中的 Class: AWS.Request。有关 AWS.Response 对象的更多信息，请参阅使用响应对象或 API 参考中的 Class: AWS.Response。

链接多个回调

您可在任何请求对象上注册多个回调。可以为不同事件或相同事件注册多个回调。此外，您可以按下例中所示链接多个回调。

```
request.
  on('success', function(response) {
    console.log("Success!");
  }).
  on('error', function(response) {
    console.log("Error!");
  }).
  on('complete', function() {
    console.log("Always!");
  }).
  send();
```

请求对象完成事件

AWS.Request 对象根据各个服务操作方法的响应引发这些完成事件：

- `success`
- `error`
- `complete`

您可以注册回调函数以响应任意这些事件。有关所有请求对象事件的完整列表，请参阅 API 参考中的 [Class: AWS.Request]。

success 事件

从服务对象收到成功响应时引发 `success` 事件。下面说明您如何为此事件注册回调函数。

```
request.on('success', function(response) {
  // event handler code
});
```

该响应提供 `data` 属性，其中包含来自服务的序列化响应数据。例如，以下内容调用 Amazon S3 服务对象的 `listBuckets` 方法

```
s3.listBuckets.on('success', function(response) {
  console.log(response.data);
}).send();
```

返回响应，然后将以下 `data` 属性内容输出到控制台。

```
{ Owner: { ID: '...', DisplayName: '...' },
  Buckets:
   [ { Name: 'someBucketName', CreationDate: someCreationDate },
     { Name: 'otherBucketName', CreationDate: otherCreationDate } ],
  RequestId: '...' }
```

error 事件

从服务对象收到出错响应时引发 `error` 事件。下面说明您如何为此事件注册回调函数。

```
request.on('error', function(error, response) {
  // event handling code
});
```

引发 `error` 事件时，响应的 `data` 属性的值为 `null`，`error` 属性包含错误数据。关联的 `error` 对象作为第一个参数传递到注册的回调函数。例如，以下代码：

```
s3.config.credentials.accessKeyId = 'invalid';
s3.listBuckets().on('error', function(error, response) {
  console.log(error);
}).send();
```

返回错误，然后将以下错误数据输出到控制台。

```
{ code: 'Forbidden', message: null }
```

complete 事件

服务对象调用完成时将引发 `complete` 事件，而不论调用结果为成功还是失败。下面说明您如何为此事件注册回调函数。

```
request.on('complete', function(response) {
  // event handler code
});
```

使用 `complete` 事件回调来处理不论成功还是出错均必须执行的任何请求清理。如果您在 `complete` 事件的回调中使用响应数据，请先检查 `response.data` 或 `response.error` 属性，然后尝试访问任意一个属性，如下例中所示。

```
request.on('complete', function(response) {
  if (response.error) {
    // an error occurred, handle it
  } else {
    // we can use response.data here
  }
}).send();
```

请求对象 HTTP 事件

`AWS.Request` 对象根据各个服务操作方法的响应引发这些 HTTP 事件：

- `httpHeaders`
- `httpData`
- `httpUploadProgress`
- `httpDownloadProgress`

- `httpError`
- `httpDone`

您可以注册回调函数以响应任意这些事件。有关所有请求对象事件的完整列表，请参阅 API 参考中的 [Class: AWS.Request]。

httpHeaders 事件

当远程服务器发送标头时，将引发 `httpHeaders` 事件。下面说明您如何为此事件注册回调函数。

```
request.on('httpHeaders', function(statusCode, headers, response) {
  // event handling code
});
```

传递到回调函数的 `statusCode` 参数是 HTTP 状态代码。`headers` 参数包含响应标头。

httpData 事件

引发 `httpData` 事件用于流式传输来自服务的响应数据包。下面说明您如何为此事件注册回调函数。

```
request.on('httpData', function(chunk, response) {
  // event handling code
});
```

此事件通常在不适合将整个响应加载到内存中时，用于分块接收较大的响应。此事件具有额外的 `chunk` 参数，其中包含来自服务器的实际数据的一部分。

如果您为 `httpData` 事件注册回调，则响应的 `data` 属性包含请求的整个序列化输出。如果您没有用于内置处理程序的额外解析和内存开销，则必须删除默认 `httpData` 侦听器。

httpUploadProgress 和 httpDownloadProgress 事件

HTTP 请求上传了更多数据时将引发 `httpUploadProgress` 事件。与此类似，HTTP 请求下载了更多数据时将引发 `httpDownloadProgress` 事件。下面说明您如何为这些事件注册回调函数。

```
request.on('httpUploadProgress', function(progress, response) {
  // event handling code
})
.on('httpDownloadProgress', function(progress, response) {
  // event handling code
```

```
});
```

传递到回调函数的 `progress` 参数包含一个对象，该对象保存请求的已加载字节数和总字节数。

httpError 事件

HTTP 请求失败时将引发 `httpError` 事件。下面说明您如何为此事件注册回调函数。

```
request.on('httpError', function(error, response) {
  // event handling code
});
```

传递到回调函数的 `error` 参数包含所引发的错误。

httpDone 事件

服务器完成数据发送时将引发 `httpDone` 事件。下面说明您如何为此事件注册回调函数。

```
request.on('httpDone', function(response) {
  // event handling code
});
```

使用异步/等待

您可以在调用 适用于 JavaScript 的 Amazon SDK 时使用 `async/await` 模式。大多数接受回调的函数都不会返回 promise。由于您只使用返回 promise 的 `await` 函数，因此要使用 `async/await` 模式，您需要将 `.promise()` 方法链接到调用的末尾并移除回调。

以下示例使用 async/await 来列出您在 `us-west-2` 中的所有 Amazon DynamoDB 表。

```
var AWS = require("aws-sdk");
//Create an Amazon DynamoDB client service object.
dbClient = new AWS.DynamoDB({ region: "us-west-2" });
// Call DynamoDB to list existing tables
const run = async () => {
  try {
    const results = await dbClient.listTables({}).promise();
    console.log(results.TableNames.join("\n"));
  } catch (err) {
    console.error(err);
  }
```

```
};
run();
```

> **Note**
>
> 并非所有浏览器都支持 async/await。有关支持异步/等待的浏览器列表，请参阅异步函数。

使用 JavaScript Promise

`AWS.Request.promise` 方法提供了一种方式来调用服务操作和管理异步流，而不使用回调。在 Node.js 和浏览器脚本中，未使用回调函数调用了服务操作时，将返回 `AWS.Request` 对象。您可以调用请求的 `send` 方法来发出服务调用。

但是，`AWS.Request.promise` 立即启动服务调用并返回 promise，其中通过响应 `data` 属性来执行，或者通过响应 `error` 属性来拒绝。

```
var request = new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances();

// create the promise object
var promise = request.promise();

// handle promise's fulfilled/rejected states
promise.then(
  function(data) {
    /* process the data */
  },
  function(error) {
    /* handle the error */
  }
);
```

接下来的示例返回 promise，其中通过 `data` 对象来执行，或者通过 `error` 对象来拒绝。使用 promise，单个回调不负责检测错误。相反，根据请求的成功或失败来调用正确的回调。

```
var s3 = new AWS.S3({apiVersion: '2006-03-01', region: 'us-west-2'});
var params = {
  Bucket: 'bucket',
  Key: 'example2.txt',
  Body: 'Uploaded text using the promise-based method!'
};
```

```
var putObjectPromise = s3.putObject(params).promise();
putObjectPromise.then(function(data) {
  console.log('Success');
}).catch(function(err) {
  console.log(err);
});
```

协调多个 Promise

在某些情况下，您的代码必须进行多个异步调用，这些调用只有在它们都成功返回时才需要执行操作。如果您使用 promise 管理这些单独的异步方法调用，则可以创建使用 `all` 方法的额外 promise。此方法只有在执行了您传递到方法中的 promise 数组时，才会执行此伞形 promise。回调函数将 promise 的值数组传递到 `all` 方法。

在以下示例中，Amazon Lambda 函数必须对 Amazon DynamoDB 进行三个异步调用，但只有在执行了各个调用的 promise 时才能完成。

```
Promise.all([firstPromise, secondPromise, thirdPromise]).then(function(values) {

  console.log("Value 0 is " + values[0].toString);
  console.log("Value 1 is " + values[1].toString);
  console.log("Value 2 is " + values[2].toString);

  // return the result to the caller of the Lambda function
  callback(null, values);
});
```

Promise 的浏览器和 Node.js 支持

对原生 JavaScript promise (ECMAScript 2015) 的支持取决于执行代码的 JavaScript 引擎和版本。为帮助确定您要运行代码的各个环境中的 JavaScript promise 支持情况，请参阅 GitHub 上的 ECMAScript 兼容性表。

使用其他 Promise 实施

除了 ECMAScript 2015 中的原生 promise 实施之外，您还可以使用第三方 promise 库，包括：

- bluebird
- RSVP
- Q

如果您的环境不支持 ECMAScript 5 和 ECMAScript 2015 中的原生 promise 实施，而您需要在环境中运行代码，这些可选的 promise 库会非常有用。

要使用第三方 promise 库，请通过在全局配置对象上调用 `setPromisesDependency` 方法，在开发工具包上设置 promise 依赖关系。在浏览器脚本中，请确保先加载第三方 promise 库，然后加载开发工具包。在以下示例中，开发工具包配置为使用 bluebird promise 库中的实施。

```
AWS.config.setPromisesDependency(require('bluebird'));
```

要返回使用 JavaScript 引擎的原生 promise 实施，请再次调用 `setPromisesDependency`，传递 `null` 而不是库名。

使用响应对象

调用服务对象方法之后，它通过将 `AWS.Response` 对象传递到回调函数来返回对象。您通过 `AWS.Response` 对象的属性访问响应的内容。`AWS.Response` 对象有两个您可用于访问响应内容的属性：

- `data` 属性
- `error` 属性

使用标准回调机制时，这两个属性作为参数在匿名回调函数上提供，如下例中所示。

```
function(error, data) {
    if (error) {
        // error handling code
        console.log(error);
    } else {
        // data handling code
        console.log(data);
    }
}
```

访问在响应对象中返回的数据

`data` 对象的 `AWS.Response` 属性包含由服务请求返回的序列化数据。请求成功时，`data` 属性包含的对象中包含了对所返回数据的映射。如果出错，`data` 属性可以为 null。

以下示例调用 DynamoDB 表的 `getItem` 方法，用于检索在游戏中使用的图像文件的文件名。

```
// Initialize parameters needed to call DynamoDB
var slotParams = {
    Key : {'slotPosition' : {N: '0'}},
    TableName : 'slotWheels',
    ProjectionExpression: 'imageFile'
};

// prepare request object for call to DynamoDB
var request = new AWS.DynamoDB({region: 'us-west-2', apiVersion:
 '2012-08-10'}).getItem(slotParams);
// log the name of the image file to load in the slot machine
request.on('success', function(response) {
    // logs a value like "cherries.jpg" returned from DynamoDB
    console.log(response.data.Item.imageFile.S);
});
// submit DynamoDB request
request.send();
```

对于此示例，DynamoDB 表是对显示老虎机拉杆结果的图像进行的查找，这由 `slotParams` 中的参数指定。

成功调用 `getItem` 方法时，`AWS.Response` 对象的 `data` 属性包含 DynamoDB 返回的 `Item` 对象。返回的数据根据请求的 `ProjectionExpression` 参数进行访问，在此例中意味着 `imageFile` 对象的 `Item` 成员。由于 `imageFile` 成员保存字符串值，您可以通过 `S` 的 `imageFile` 子成员的值，访问图像本身的文件名。

分页查看返回的数据

有时候，服务请求的 `data` 属性返回的数据可能会跨多个页面。您可以通过调用 `response.nextPage` 方法访问数据的下一页。此方法发送新请求。来自请求的响应可以通过回调捕获，或者通过成功和错误侦听器捕获。

您可以通过调用 `response.hasNextPage` 方法，检查以确定服务请求返回的数据是否有额外的数据页面。此方法返回一个布尔值，指示调用 `response.nextPage` 是否返回了额外的数据。

```
s3.listObjects({Bucket: 'bucket'}).on('success', function handlePage(response) {
    // do something with response.data
    if (response.hasNextPage()) {
        response.nextPage().on('success', handlePage).send();
    }
}).send();
```

访问来自响应对象的错误信息

在出现服务错误或传输错误事件时，error 对象的 AWS.Response 属性包含可用的错误数据。返回的错误采用以下形式。

```
{ code: 'SHORT_UNIQUE_ERROR_CODE', message: 'a descriptive error message' }
```

在出错时，data 属性的值为 null。如果您处理可以处于故障状态的事件，请始终首先检查是否设置了 error 属性，然后尝试访问 data 属性的值。

访问原始请求对象

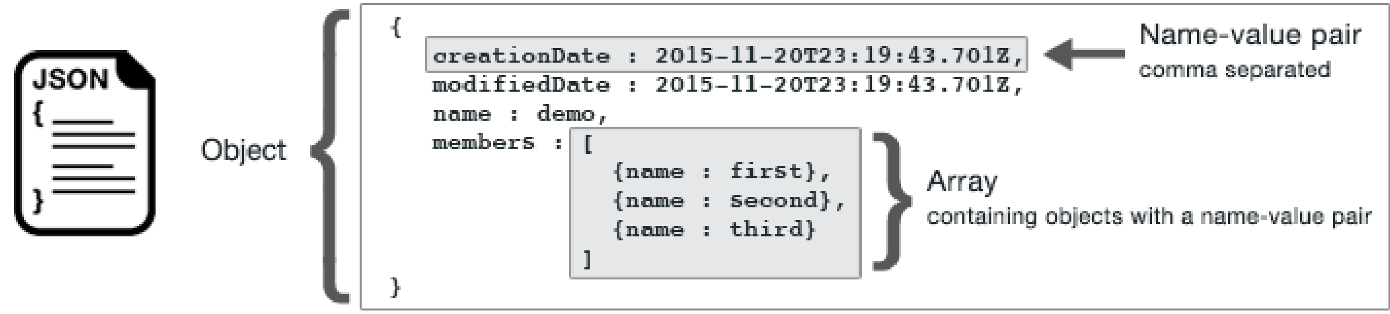
request 属性提供对原始 AWS.Request 对象的访问。引用原始 AWS.Request 对象以访问它发送的原始参数会非常有用。在以下示例中，使用 request 属性访问原始服务请求的 Key 参数。

```
s3.getObject({Bucket: 'bucket', Key: 'key'}).on('success', function(response) {
   console.log("Key was", response.request.params.Key);
}).send();
```

使用 JSON

JSON 是一种数据交换格式，便于人类阅读，并且是机器可读的。虽然名称 JSON 是 JavaScript 对象表示法 (JavaScript Object Notation) 的缩写，但 JSON 的格式独立于任何编程语言。

SDK for JavaScript 在发出请求时使用 JSON 将数据发送到服务对象，并以 JSON 格式从服务对象接收数据。有关 JSON 的更多信息，请参阅 json.org。



JSON 通过两种方式表示数据：

- 对象，其是无序名称-值对集合。对象在左大括号 ({) 和右大括号 (}) 内定义。每个名称-值对以名称开头，后接一个冒号，再接值。名称/值对以逗号分隔。

- 数组，其是有序值集合。数组在左方括号 ([) 和右方括号 (]) 内定义。数组中的项目以逗号分隔。

下面是 JSON 对象示例，其中包含一个对象数组，这些对象表示扑克游戏中的扑克。每张扑克都由两个名称/值对定义，一个指定用于表示扑克的唯一值，另一个指定指向对应扑克图像的 URL。

```
var cards = [{"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"}];
```

将 JSON 作为服务对象参数

以下是一个简单 JSON 示例，用于定义对 Lambda 服务对象的调用的参数。

```
var pullParams = {
   FunctionName : 'slotPull',
   InvocationType : 'RequestResponse',
   LogType : 'None'
};
```

`pullParams` 对象由三个名称/值对定义，在左右大括号中以逗号分隔。向服务对象方法调用提供参数时，名称由您计划调用的服务对象方法的参数名称确定。调用 Lambda 函数时，`FunctionName`、`InvocationType` 和 `LogType` 是用于在 Lambda 服务对象上调用 `invoke` 方法的参数。

将参数传递给服务对象方法调用时，将 JSON 对象提供给方法调用，如下面调用 Lambda 函数的示例中所示。

```
lambda = new AWS.Lambda({region: 'us-west-2', apiVersion: '2015-03-31'});
// create JSON object for service call parameters
var pullParams = {
   FunctionName : 'slotPull',
   InvocationType : 'RequestResponse',
   LogType : 'None'
};
// invoke Lambda function, passing JSON object
lambda.invoke(pullParams, function(err, data) {
   if (err) {
      console.log(err);
```

```
   } else {
      console.log(data);
   }
});
```

以 JSON 格式返回数据

JSON 提供了一种标准方式，针对需要同时发送多个值的情况，在应用程序的不同部分传递数据。API 中客户端类的方法通常在传递到其回调函数的 `data` 参数中返回 JSON。例如，以下是对 Amazon S3 客户端类的 `getBucketCors` 方法的调用。

```
// call S3 to retrieve CORS configuration for selected bucket
s3.getBucketCors(bucketParams, function(err, data) {
  if (err) {
    console.log(err);
  } else if (data) {
    console.log(JSON.stringify(data));
  }
});
```

`data` 的值是 JSON 对象，在此例中是描述指定 Amazon S3 桶的当前 CORS 配置的 JSON。

```
{
   "CORSRules": [
      {
          "AllowedHeaders":["*"],
          "AllowedMethods":["POST","GET","PUT","DELETE","HEAD"],
          "AllowedOrigins":["*"],
          "ExposeHeaders":[],
          "MaxAgeSeconds":3000
      }
   ]
}
```

在适用于 JavaScript 的 Amazon SDK v2 中重试策略

网络上的大量组件 (例如 DNS 服务器、交换机、负载均衡器等) 都可能在某个指定请求生命周期中的任一环节出现问题。在联网环境中，处理这些错误响应的常规技术是在客户应用程序中实施重试。该技术可以提高应用程序的可靠性和降低开发人员的操作成本。AmazonSDK 为您的 Amazon 请求实现自动重试逻辑。

基于指数回退的重试行为

适用于 JavaScript 的 Amazon SDK v2 使用指数回退及完全抖动实现重试逻辑，以获得更好的流量控制。指数回退的原理是对于连续错误响应，重试等待间隔越来越长。抖动（随机延迟）用于防止连续的冲突。

在 v2 中测试重试延迟

为了在 v2 中测试重试延迟，node_modules/aws-sdk/lib/event_listeners.js 中的代码已更新为 `console.log`。可变延迟中显示的值如下所示：

```
// delay < 0 is a signal from customBackoff to skip retries
if (willRetry && delay >= 0) {
  resp.error = null;
  console.log('retry delay: ' + delay);
  setTimeout(done, delay);
} else {
  done();
}
```

默认配置下的重试延迟

您可以在 AWS SDK 客户端上测试任何操作的延迟。我们使用以下代码在 DynamoDB 客户端上调用 `listTables` 操作：

```
import AWS from "aws-sdk";

const region = "us-east-1";
const client = new AWS.DynamoDB({ region });
await client.listTables({}).promise();
```

为了测试重试，我们通过断开互联网与运行测试代码的设备的连接来模拟 `NetworkingError`。还可以设置代理以返回自定义错误。

在运行代码时，您可以看到使用指数回退及抖动的重试延迟，如下所示：

```
retry delay: 7.39361151766359
retry delay: 9.0672860785882
retry delay: 134.89340825668168
retry delay: 398.53559817403965
```

```
retry delay: 523.8076165896343
retry delay: 1323.8789643058465
```

由于重试使用抖动，因此在运行示例代码时您将获得不同的值。

使用自定义基数的重试延迟

适用于 JavaScript 的 Amazon SDK v2 支持传递一个以毫秒为单位的自定义基数，用于操作重试的指数回退中。对于所有服务，其默认值均为 100 毫秒，但 DynamoDB 除外（默认为 50 毫秒）。

我们使用自定义基数 1000 毫秒来测试重试，如下所示：

```
...
const client = new AWS.DynamoDB({ region, retryDelayOptions: { base: 1000 } });
...
```

我们通过断开互联网与运行测试代码的设备的连接来模拟 `NetworkingError`。您可以看到，与上一次运行相比，重试延迟的值更高，默认值为 50 或 100 毫秒。

```
retry delay: 356.2841549924913
retry delay: 1183.5216495444615
retry delay: 2266.997988094194
retry delay: 1244.6948354966453
retry delay: 4200.323030066383
```

由于重试使用抖动，因此在运行示例代码时您将获得不同的值。

使用自定义回退算法的重试延迟

适用于 JavaScript 的 Amazon SDK v2 还支持传递一个自定义回退函数，该函数接受重试计数和错误，并以毫秒为单位返回延迟时间量。如果结果为非零负值，则不会再进行重试尝试。

我们测试的自定义回退函数使用基数值为 200 毫秒的线性回退，如下所示：

```
...
const client = new AWS.DynamoDB({
  region,
  retryDelayOptions: { customBackoff: (count, error) => (count + 1) * 200 },
});
...
```

我们通过断开互联网与运行测试代码的设备的连接来模拟 `NetworkingError`。您可以看到重试延迟的值是 200 的倍数。

```
retry delay: 200
retry delay: 400
retry delay: 600
retry delay: 800
retry delay: 1000
```

SDK for JavaScript 代码示例

此部分中的主题包含如何将适用于 JavaScript 的 Amazon SDK与各种服务的 API 结合使用来执行常见任务的示例。

您可以在 GitHub 上的代码示例库 Amazon 文档中找到这些示例及其它示例的源代码。要向 Amazon 文档团队提请考虑生成新的代码示例，请创建新的请求。该团队正在寻求生成涵盖更多应用场景和使用情形的代码示例，而不仅仅是涵盖个别 API 调用的简单代码片段。有关说明，请参阅 Contribution guidelines 中的 Authoring code 部分。

主题

- Amazon CloudWatch 示例
- Amazon DynamoDB 示例
- Amazon EC2 示例
- AWS Elemental MediaConvert 示例
- Amazon IAM 示例
- Amazon Kinesis 示例
- Amazon S3 示例
- Amazon Simple Email Service 示例
- Amazon Simple Notification Service 示例
- Amazon SQS 示例

Amazon CloudWatch 示例

Amazon CloudWatch（CloudWatch）是一项 Web 服务，可实时监控您的 Amazon Web Services 资源以及您在 Amazon 上运行的应用程序。您可以使用 CloudWatch 收集和跟踪指标，这些指标是您可衡量的相关资源和应用程序的变量。CloudWatch 警报可根据您定义的规则发送通知或者对您所监控的资源自动进行更改。

适用于 CloudWatch 的 JavaScript API 通过 `AWS.CloudWatch`、`AWS.CloudWatchEvents` 和 `AWS.CloudWatchLogs` 客户端类公开。有关使用 CloudWatch 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.CloudWatch`、`Class: AWS.CloudWatchEvents` 和 `Class: AWS.CloudWatchLogs`。

主题

- 在 Amazon CloudWatch 中创建警报
- 在 Amazon CloudWatch 中使用警报操作
- 从 Amazon CloudWatch 获取指标
- 将事件发送到 Amazon CloudWatch Events
- 在 Amazon CloudWatch Logs 中使用订阅筛选条件

在 Amazon CloudWatch 中创建警报

此 Node.js 代码示例演示：

- 如何检索有关 CloudWatch 警报的基本信息。
- 如何创建和删除 CloudWatch 警报。

情景

警报会每隔一段时间（由您指定）监控一个指标，并根据相对于给定阈值的指标值每隔若干个时间段执行一项或多项操作。

本示例使用一系列 Node.js 模块在 CloudWatch 中创建警报。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.CloudWatch` 客户端类的以下方法来创建警报：

- `describeAlarms`
- `putMetricAlarm`
- `deleteAlarms`

有关 CloudWatch 警报的更多信息，请参阅的《Amazon CloudWatch 用户指南》中的创建 Amazon CloudWatch 告警。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

描述警报

创建文件名为 `cw_describealarms.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。创建 JSON 对象，保存用于检索警报描述的参数，将返回的警报限制为具有状态 `INSUFFICIENT_DATA` 的警报。然后调用 `describeAlarms` 服务对象的 `AWS.CloudWatch` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });
```

```
cw.describeAlarms({ StateValue: "INSUFFICIENT_DATA" }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    // List the names of all current alarms in the console
    data.MetricAlarms.forEach(function (item, index, array) {
      console.log(item.AlarmName);
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_describealarms.js
```

此示例代码可在 GitHub 上的此处找到。

为 CloudWatch 指标创建警报

创建文件名为 `cw_putmetricalarm.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。针对基于指标创建警报所需的参数创建 JSON 对象，在本示例中指标为 Amazon EC2 实例的 CPU 利用率。其余参数设置为在指标超过 70% 的阈值时触发警报。然后调用 `describeAlarms` 服务对象的 `AWS.CloudWatch` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
  Period: 60,
  Statistic: "Average",
  Threshold: 70.0,
  ActionsEnabled: false,
```

```
  AlarmDescription: "Alarm when server CPU exceeds 70%",
  Dimensions: [
    {
      Name: "InstanceId",
      Value: "INSTANCE_ID",
    },
  ],
  Unit: "Percent",
};

cw.putMetricAlarm(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_putmetricalarm.js
```

此示例代码可在 GitHub 上的此处找到。

删除警报

创建文件名为 `cw_deletealarms.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。创建 JSON 对象以保存您要删除的警报的名称。然后调用 `deleteAlarms` 服务对象的 `AWS.CloudWatch` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmNames: ["Web_Server_CPU_Utilization"],
};
```

```
cw.deleteAlarms(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_deletealarms.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon CloudWatch 中使用警报操作

此 Node.js 代码示例演示：

- 如何根据 CloudWatch 警报自动更改 Amazon EC2 实例的状态。

情景

利用警报操作，您可以创建自动停止、终止、重启或恢复 Amazon EC2 实例的警报。当不再需要某个实例运行时，您可使用停止或终止操作。使用重启和恢复操作可以自动重启这些实例。

本示例使用一系列 Node.js 模块来定义 CloudWatch 中的警报操作，这些操作触发 Amazon EC2 实例的重启。这些 Node.js 模块使用 SDK for JavaScript，通过 `CloudWatch` 客户端类的以下方法来管理 Amazon EC2 实例：

- `enableAlarmActions`
- `disableAlarmActions`

有关 CloudWatch 警报操作的更多信息，请参阅《Amazon CloudWatch 用户指南》中的创建停止、终止、重启或恢复实例的警报。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个 IAM 角色，其策略授予描述、重启、停止或终止 Amazon EC2 实例的权限。有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

在创建 IAM 角色时，使用以下角色策略。

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "cloudwatch:Describe*",
            "ec2:Describe*",
            "ec2:RebootInstances",
            "ec2:StopInstances*",
            "ec2:TerminateInstances"
         ],
         "Resource": [
            "*"
         ]
      }
   ]
}
```

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var Amazon = require('aws-sdk');
```

```
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

创建并在警报上启用操作

创建文件名为 `cw_enablealarmactions.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。

创建 JSON 对象以保存用于创建警报的参数，将 `ActionsEnabled` 指定为 `true`，并为警报将触发的操作指定 ARN 数组。调用 `putMetricAlarm` 服务对象的 `AWS.CloudWatch` 方法，这会在警报不存在时创建警报，在警报存在时更新警报。

在 `putMetricAlarm` 的回调函数中，成功完成时，将创建一个包含 CloudWatch 警报名称的 JSON 对象。调用 `enableAlarmActions` 方法以启用警报操作。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
  Period: 60,
  Statistic: "Average",
  Threshold: 70.0,
  ActionsEnabled: true,
  AlarmActions: ["ACTION_ARN"],
  AlarmDescription: "Alarm when server CPU exceeds 70%",
  Dimensions: [
    {
      Name: "InstanceId",
      Value: "INSTANCE_ID",
    },
  ],
  Unit: "Percent",
};
```

```
cw.putMetricAlarm(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Alarm action added", data);
    var paramsEnableAlarmAction = {
      AlarmNames: [params.AlarmName],
    };
    cw.enableAlarmActions(paramsEnableAlarmAction, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Alarm action enabled", data);
      }
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_enablealarmactions.js
```

此示例代码可在 GitHub 上的此处找到。

禁用警报上的操作

创建文件名为 `cw_disablealarmactions.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。创建一个包含 CloudWatch 警报名称的 JSON 对象。调用 `disableAlarmActions` 方法以禁用此警报的操作。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

cw.disableAlarmActions(
  { AlarmNames: ["Web_Server_CPU_Utilization"] },
  function (err, data) {
```

```
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node cw_disablealarmactions.js
```

此示例代码可在 GitHub 上的此处找到。

从 Amazon CloudWatch 获取指标

此 Node.js 代码示例演示：

- 如何检索已发布 CloudWatch 指标的列表。
- 如何将数据点发布到 CloudWatch 指标。

情景

指标是关于您的系统性能的数据。您也可以启用对某些资源（例如 Amazon EC2 实例）或您自己的应用程序指标的详细监控。

本示例使用一系列 Node.js 模块从 CloudWatch 获取指标并将事件发送到 Amazon CloudWatch Events。这些 Node.js 模块使用 SDK for JavaScript，通过 `CloudWatch` 客户端类的以下方法从 CloudWatch 获取指标：

- `listMetrics`
- `putMetricData`

有关 CloudWatch 指标的更多信息，请参阅《Amazon CloudWatch 用户指南》中的使用 Amazon CloudWatch 指标。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

列出指标

创建文件名为 `cw_listmetrics.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 `AWS.CloudWatch` 服务对象。创建一个 JSON 对象，该对象包含在 `AWS/Logs` 命名空间中列出指标所需的参数。调用 `listMetrics` 方法以列出 `IncomingLogEvents` 指标。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  Dimensions: [
    {
      Name: "LogGroupName" /* required */,
    },
  ],
  MetricName: "IncomingLogEvents",
  Namespace: "AWS/Logs",
};

cw.listMetrics(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Metrics", JSON.stringify(data.Metrics));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_listmetrics.js
```

此示例代码可在 GitHub 上的此处找到。

提交自定义指标

创建文件名为 cw_putmetricdata.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch，请创建 AWS.CloudWatch 服务对象。创建一个 JSON 对象，其中包含为 PAGES_VISITED 自定义指标提交数据点所需的参数。调用 putMetricData 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

// Create parameters JSON for putMetricData
var params = {
  MetricData: [
    {
      MetricName: "PAGES_VISITED",
      Dimensions: [
        {
          Name: "UNIQUE_PAGES",
          Value: "URLS",
        },
      ],
      Unit: "None",
      Value: 1.0,
    },
  ],
  Namespace: "SITE/TRAFFIC",
};

cw.putMetricData(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data));
  }
```

```
});
```

要运行示例，请在命令行中键入以下内容。

```
node cw_putmetricdata.js
```

此示例代码可在 GitHub 上的此处找到。

将事件发送到 Amazon CloudWatch Events

此 Node.js 代码示例演示：

- 如何创建和更新用于触发事件的规则。
- 如何定义一个或多个目标以响应事件。
- 如何发送与目标匹配的事件以进行处理。

情景

CloudWatch Events 提供几乎实时的系统事件流，这些事件描述了 Amazon Web Services 资源中针对任何不同目标的更改。通过简单规则，您可以匹配事件并将事件路由到一个或多个目标函数或流。

本示例使用一系列 Node.js 模块将事件发送到 CloudWatch Events。这些 Node.js 模块使用 SDK for JavaScript，通过 `CloudWatchEvents` 客户端类的以下方法来管理实例：

- `putRule`
- `putTargets`
- `putEvents`

有关 CloudWatch Events 的更多信息，请参阅《Amazon CloudWatch Events User Guide》中的 Adding Events with PutEvents。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建 Lambda 函数，并将 hello-world 蓝图用作事件的目标。要了解如何操作，请参阅《Amazon CloudWatch Events User Guide》中的 Step 1: Create an Amazon Lambda function。
- 创建一个 IAM 角色，其策略向 CloudWatch Events 授予权限并包含 `events.amazonaws.com` 作为可信实体。有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

在创建 IAM 角色时，使用以下角色策略。

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Sid": "CloudWatchEventsFullAccess",
         "Effect": "Allow",
         "Action": "events:*",
         "Resource": "*"
      },
      {
         "Sid": "IAMPassRoleForCloudWatchEvents",
         "Effect": "Allow",
         "Action": "iam:PassRole",
         "Resource": "arn:aws:iam::*:role/AWS_Events_Invoke_Targets"
      }
   ]
}
```

在创建 IAM 角色时，使用以下信任关系。

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
```

```
      {
         "Effect": "Allow",
         "Principal": {
            "Service": "events.amazonaws.com"
         },
         "Action": "sts:AssumeRole"
      }
   ]
}
```

创建计划规则

创建文件名为 `cwe_putrule.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Events，请创建一个 `AWS.CloudWatchEvents` 服务对象。创建一个包含指定新计划规则所需的参数的 JSON 对象，其中包括下列内容：

- 规则的名称
- 您之前创建的 IAM 角色的 ARN
- 每五分钟触发计划的表达式

调用 `putRule` 方法以创建规则。回调返回新规则或更新后规则的 ARN。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });

var params = {
  Name: "DEMO_EVENT",
  RoleArn: "IAM_ROLE_ARN",
  ScheduleExpression: "rate(5 minutes)",
  State: "ENABLED",
};

cwevents.putRule(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Success", data.RuleArn);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwe_putrule.js
```

此示例代码可在 GitHub 上的此处找到。

添加 Amazon Lambda 函数目标

创建文件名为 `cwe_puttargets.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Events，请创建一个 `AWS.CloudWatchEvents` 服务对象。创建一个包含指定要附加到目标的规则所需的参数的 JSON 对象，其中包括您创建的 Lambda 函数的 ARN。调用 `putTargets` 服务对象的 `AWS.CloudWatchEvents` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });

var params = {
  Rule: "DEMO_EVENT",
  Targets: [
    {
      Arn: "LAMBDA_FUNCTION_ARN",
      Id: "myCloudWatchEventsTarget",
    },
  ],
};

cwevents.putTargets(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
```

```
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwe_puttargets.js
```

此示例代码可在 GitHub 上的此处找到。

发送事件

创建文件名为 cwe_putevents.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Events，请创建一个 AWS.CloudWatchEvents 服务对象。创建一个包含发送事件所需的参数的 JSON 对象。对于每个事件，包括事件源、受事件影响的任意资源的 ARN 以及事件详细信息。调用 putEvents 服务对象的 AWS.CloudWatchEvents 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });

var params = {
  Entries: [
    {
      Detail: '{ "key1": "value1", "key2": "value2" }',
      DetailType: "appRequestSubmitted",
      Resources: ["RESOURCE_ARN"],
      Source: "com.company.app",
    },
  ],
};

cwevents.putEvents(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Entries);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwe_putevents.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon CloudWatch Logs 中使用订阅筛选条件

此 Node.js 代码示例演示：

- 如何在 CloudWatch Logs 中为日志事件创建和删除筛选条件。

情景

通过订阅，可以从 CloudWatch Logs 提供对日志事件的实时源的访问，并将该源传输到 Amazon Kinesis 流或 Amazon Lambda 等其它服务，以对其进行自定义处理、分析或将其加载到其它系统。订阅筛选条件定义用于筛选传输到您的 Amazon 资源的日志事件的模式。

本示例使用一系列 Node.js 模块在 CloudWatch Logs 中列出、创建和删除订阅筛选条件。日志事件的目标是 Lambda 函数。这些 Node.js 模块使用 SDK for JavaScript，通过 `CloudWatchLogs` 客户端类的以下方法来管理订阅筛选条件：

- `putSubscriptionFilters`
- `describeSubscriptionFilters`
- `deleteSubscriptionFilter`

有关 CloudWatch Logs 订阅的更多信息，请参阅《Amazon CloudWatch Logs 用户指南》中的使用订阅实时处理日志数据。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。

- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建 Lambda 函数作为日志事件的目标。您将需要使用此函数的 ARN。有关设置 Lambda 函数的更多信息，请参阅《Amazon CloudWatch Logs 用户指南》中的 Amazon Lambda 订阅筛选条件。
- 创建一个 IAM 角色，其策略授予权限以调用您创建的 Lambda 函数并授予对 CloudWatch Logs 的完全访问权，或者将以下策略应用于您为 Lambda 函数创建的执行角色。有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

在创建 IAM 角色时，使用以下角色策略。

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
         ],
         "Resource": "arn:aws:logs:*:*:*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "lambda:InvokeFunction"
         ],
         "Resource": [
            "*"
         ]
      }
   ]
}
```

描述现有订阅筛选条件

创建文件名为 `cwl_describesubscriptionfilters.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Logs，请创建 `AWS.CloudWatchLogs` 服务对象。创建一个包含描述现有筛选条件所需参数的 JSON 对象，其中包括日志组的名称以及您所要描述的筛选条件最大数。调用 `describeSubscriptionFilters` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  logGroupName: "GROUP_NAME",
  limit: 5,
};

cwl.describeSubscriptionFilters(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.subscriptionFilters);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwl_describesubscriptionfilters.js
```

此示例代码可在 GitHub 上的此处找到。

创建订阅筛选条件

创建文件名为 `cwl_putsubscriptionfilter.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Logs，请创建 `AWS.CloudWatchLogs` 服务对象。创建一个 JSON 对象，其中包含创建筛选条件所需的参数，包括目标 Lambda 函数 ARN、筛选条件名称、要筛选的字符串模式以及日志组名称。调用 `putSubscriptionFilters` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  destinationArn: "LAMBDA_FUNCTION_ARN",
  filterName: "FILTER_NAME",
  filterPattern: "ERROR",
  logGroupName: "LOG_GROUP",
};

cwl.putSubscriptionFilter(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwl_putsubscriptionfilter.js
```

此示例代码可在 GitHub 上的此处找到。

删除订阅筛选条件

创建文件名为 `cwl_deletesubscriptionfilters.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 CloudWatch Logs，请创建 `AWS.CloudWatchLogs` 服务对象。创建一个包含删除筛选条件所需的参数的 JSON 对象，其中包括筛选条件的名称和日志组。调用 `deleteSubscriptionFilters` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  filterName: "FILTER",
  logGroupName: "LOG_GROUP",
};

cwl.deleteSubscriptionFilter(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node cwl_deletesubscriptionfilter.js
```

此示例代码可在 GitHub 上的此处找到。

Amazon DynamoDB 示例

Amazon DynamoDB 是一种完全托管的 NoSQL 云数据库，支持文档和键值两种存储模式。您可以为数据创建无架构表，而无需预配置或维护专用的数据库服务器。

适用于 DynamoDB 的 JavaScript API 通过 `AWS.DynamoDB`、`AWS.DynamoDBStreams` 和 `AWS.DynamoDB.DocumentClient` 客户端类公开。有关使用 DynamoDB 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.DynamoDB`、`Class: AWS.DynamoDBStreams` 和 `Class: AWS.DynamoDB.DocumentClient`。

主题

- 在 DynamoDB 中创建和使用表
- 在 DynamoDB 中读取和写入单个项目
- 在 DynamoDB 中批量读取和写入项目
- 查询和扫描 DynamoDB 表
- 使用 DynamoDB 文档客户端

在 DynamoDB 中创建和使用表

此 Node.js 代码示例演示：

- 如何创建和管理用于存储及从 DynamoDB 检索数据的表。

情景

类似于其他数据库系统，DynamoDB 将数据存储在表中。DynamoDB 表是数据的集合，这些数据按照类似于行的项目来排列。要在 DynamoDB 中存储或访问数据，您需要创建并使用表。

在本示例中，您使用一系列 Node.js 模块对 DynamoDB 表执行基本操作。代码使用 SDK for JavaScript，通过 `AWS.DynamoDB` 客户端类的以下方法来创建和处理表：

- createTable
- listTables
- describeTable
- deleteTable

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。

- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建表

创建文件名为 ddb_createtable.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB 服务对象。创建一个 JSON 对象，其中包含创建表所需的参数，在本示例中包括各个属性的名称和数据类型、关键架构、表的名称以及要预配置的吞吐量单位。调用 DynamoDB 服务对象的 createTable 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
```

```
  },
  TableName: "CUSTOMER_LIST",
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_createtable.js
```

此示例代码可在 GitHub 上的此处找到。

列出表

创建文件名为 `ddb_listtables.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。创建一个 JSON 对象，其中包含列出表所需的参数，在此示例中将列出的表数量限制为 10。调用 DynamoDB 服务对象的 `listTables` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

// Call DynamoDB to retrieve the list of tables
ddb.listTables({ Limit: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err.code);
  } else {
    console.log("Table names are ", data.TableNames);
  }
```

```
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_listtables.js
```

此示例代码可在 GitHub 上的此处找到。

描述表

创建文件名为 ddb_describetable.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB 服务对象。创建一个 JSON 对象，其中包含描述表所需的参数，在此示例中包括提供作为命令行参数的表的名称。调用 DynamoDB 服务对象的 describeTable 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to retrieve the selected table descriptions
ddb.describeTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Table.KeySchema);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_describetable.js TABLE_NAME
```

此示例代码可在 GitHub 上的此处找到。

删除表

创建文件名为 ddb_deletetable.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB 服务对象。创建一个 JSON 对象，其中包含删除表所需的参数，在此示例中包括提供作为命令行参数的表的名称。调用 DynamoDB 服务对象的 deleteTable 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_deletetable.js TABLE_NAME
```

此示例代码可在 GitHub 上的此处找到。

在 DynamoDB 中读取和写入单个项目

此 Node.js 代码示例演示：

- 如何在 DynamoDB 表中添加项目。
- 如何在 DynamoDB 表中检索项目。
- 如何从 DynamoDB 表中删除项目。

情景

在本示例中，您使用一系列 Node.js 模块，通过 AWS.DynamoDB 客户端类的以下方法，在 DynamoDB 表中读取和写入一个项目：

- putItem
- getItem
- deleteItem

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个您可以访问其项目的 DynamoDB 表。有关创建 DynamoDB 表的更多信息，请参阅在 DynamoDB 中创建和使用表。

写入项目

创建文件名为 ddb_putitem.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB 服务对象。创建一个 JSON 对象，其中包含添加项目所需的参数，在本示例中包括表的名称，定义要设置的属性的映射，以及各个属性的值。调用 DynamoDB 服务对象的 putItem 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_putitem.js
```

此示例代码可在 GitHub 上的此处找到。

获取项目

创建文件名为 `ddb_getitem.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。要标识所需获取的项目，您必须提供该项目在表中主键的值。默认情况下，`getItem` 方法返回为项目定义的所有属性值。要仅获取所有可能属性值的子集，请指定投影表达式。

创建一个 JSON 对象，其中包含获取某个项目所需的参数，在本示例中包括表的名称，所获取项目的键的值，以及确定要检索的项目属性的投影表达式。调用 DynamoDB 服务对象的 `getItem` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_getitem.js
```

此示例代码可在 GitHub 上的此处找到。

删除项目

创建文件名为 ddb_deleteitem.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB 服务对象。创建一个 JSON 对象，其中包含删除项目所需的参数，在本示例中包括表的名称，以及所删除的项目的键名和值。调用 DynamoDB 服务对象的 deleteItem 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_deleteitem.js
```

此示例代码可在 GitHub 上的此处找到。

在 DynamoDB 中批量读取和写入项目

此 Node.js 代码示例演示：

- 如何在 DynamoDB 表中批量读取和写入项目。

情景

在本示例中，您使用一系列 Node.js 模块在 DynamoDB 表中批量放置项目以及批量读取项目。代码使用 SDK for JavaScript，通过 DynamoDB 客户端类的以下方法来执行批量读取和写入操作：

- batchGetItem
- batchWriteItem

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个您可以访问其项目的 DynamoDB 表。有关创建 DynamoDB 表的更多信息，请参阅在 DynamoDB 中创建和使用表。

批量读取项目

创建文件名为 `ddb_batchgetitem.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。创建一个 JSON 对象，其中包含批量获取项目所需的参数，在此示例中包括要读取的一个或多个表的名称，在各个表中要读取的键的值，以及指定要返回的属性的投影表达式。调用 DynamoDB 服务对象的 `batchGetItem` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        { KEY_NAME: { N: "KEY_VALUE_1" } },
        { KEY_NAME: { N: "KEY_VALUE_2" } },
        { KEY_NAME: { N: "KEY_VALUE_3" } },
      ],
      ProjectionExpression: "KEY_NAME, ATTRIBUTE",
    },
  },
};

ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_batchgetitem.js
```

此示例代码可在 GitHub 上的此处找到。

批量写入项目

创建文件名为 `ddb_batchwriteitem.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。创建一个 JSON 对象，其中包含批量获取项目所需的参数，在本示例中包括要写入项目的表，要写入的各个项目的键，以及属性及值。调用 DynamoDB 服务对象的 `batchWriteItem` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: [
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
      {
        PutRequest: {
```

```
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
    ],
  },
};

ddb.batchWriteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_batchwriteitem.js
```

此示例代码可在 GitHub 上的此处找到。

查询和扫描 DynamoDB 表

此 Node.js 代码示例演示：

- 如何查询和扫描 DynamoDB 表的项目。

情景

仅使用主键属性值查找表或二级索引中的项目。您必须提供分区键名称和要搜索的值。您还可提供排序键名称和值，并使用比较运算符来优化搜索结果。扫描操作通过检查指定表中的每个项目来查找项目。

在本示例中，您使用一系列 Node.js 模块来标识要从 DynamoDB 表中检索的一个或多个项目。代码使用 SDK for JavaScript，通过 DynamoDB 客户端类的以下方法来查询和扫描表：

- query
- scan

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个您可以访问其项目的 DynamoDB 表。有关创建 DynamoDB 表的更多信息，请参阅在 DynamoDB 中创建和使用表。

查询表

此示例查询包含有关视频系列的剧集信息的表，返回第二季第九集之后，字幕中包含指定短语的每集的名称和字幕。

创建文件名为 `ddb_query.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。创建一个 JSON 对象，其中包含查询表所需的参数，在本示例中包括表名，查询所需的 `ExpressionAttributeValues`，使用这些值定义查询要返回的项目的 `KeyConditionExpression`，以及各个项目要返回的属性值的名称。调用 DynamoDB 服务对象的 `query` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": { N: "2" },
    ":e": { N: "09" },
```

```
    ":topic": { S: "PHRASE" },
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  ProjectionExpression: "Episode, Title, Subtitle",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

ddb.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    //console.log("Success", data.Items);
    data.Items.forEach(function (element, index, array) {
      console.log(element.Title.S + " (" + element.Subtitle.S + ")");
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_query.js
```

此示例代码可在 GitHub 上的此处找到。

扫描表

创建文件名为 `ddb_scan.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB` 服务对象。创建一个 JSON 对象，其中包含扫描表中项目所需的参数，在本示例中包括表的名称，各个匹配项目要返回的属性值的列表，以及用于筛选结果集来查找包含指定短语的项目的表达式。调用 DynamoDB 服务对象的 `scan` 方法。

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object.
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

const params = {
  // Specify which items in the results are returned.
```

```
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
  // Define the expression attribute value, which are substitutes for the values you
 want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: "SubTitle2" },
    ":s": { N: 1 },
    ":e": { N: 2 },
  },
  // Set the projection expression, which are the attributes that you want.
  ProjectionExpression: "Season, Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
    data.Items.forEach(function (element, index, array) {
      console.log(
        "printing",
        element.Title.S + " (" + element.Subtitle.S + ")"
      );
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddb_scan.js
```

此示例代码可在 GitHub 上的此处找到。

使用 DynamoDB 文档客户端

此 Node.js 代码示例演示：

- 如何使用文档客户端访问 DynamoDB 表。

情景

DynamoDB 文档客户端通过将属性值的概念抽象化，简化了项目的处理。此抽象化标注提供作为输入参数的原生 JavaScript 类型，以及将标注的响应数据转换为原生 JavaScript 类型。

有关 DynamoDB 文档客户端类的更多信息，请参阅 API 参考中的 `AWS.DynamoDB.DocumentClient`。有关使用 Amazon DynamoDB 编程的更多信息，请参阅《Amazon DynamoDB 开发人员指南》中的使用 DynamoDB 进行编程。

在本示例中，您使用一系列 Node.js 模块，通过文档客户端对 DynamoDB 表执行基本操作。代码使用 SDK for JavaScript，通过 DynamoDB 文档客户端类的以下方法来查询和扫描表：

- get
- put
- update
- query
- delete

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个您可以访问其项目的 DynamoDB 表。有关使用 SDK for JavaScript 创建 DynamoDB 表的更多信息，请参阅在 DynamoDB 中创建和使用表。您还可以使用 DynamoDB 控制台创建表。

从表中获取项目

创建文件名为 `ddbdoc_get.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB.DocumentClient` 对象。创建一个 JSON 对象，其中包含从表获取某个项目所需的参数，在本示例中包括表的名称，表中哈希键的名称，所要获取项目的哈希键的值。调用 DynamoDB 文档客户端的 `get` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddbdoc_get.js
```

此示例代码可在 GitHub 上的此处找到。

将项目放入表中

创建文件名为 `ddbdoc_put.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB.DocumentClient` 对象。创建一个 JSON 对象，其中包含将项目写入表中所需的参数，在本示例中包括表的名称，要添加或更新的项目的描述（包括哈希键和值），以及要在项目上设置的属性的名称和值。调用 DynamoDB 文档客户端的 `put` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });
```

```
var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddbdoc_put.js
```

此示例代码可在 GitHub 上的此处找到。

更新表中的项目

创建文件名为 ddbdoc_update.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 AWS.DynamoDB.DocumentClient 对象。创建一个 JSON 对象，其中包含将项目写入表中所需的参数，在本示例中包括表的名称，要更新的项目的键，定义要更新的项目的属性的一组 UpdateExpressions，以及您在 ExpressionAttributeValues 参数中将值分配到的令牌。调用 DynamoDB 文档客户端的 update 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

// Create variables to hold numeric key values
var season = SEASON_NUMBER;
```

```
var episode = EPISODES_NUMBER;

var params = {
  TableName: "EPISODES_TABLE",
  Key: {
    Season: season,
    Episode: episode,
  },
  UpdateExpression: "set Title = :t, Subtitle = :s",
  ExpressionAttributeValues: {
    ":t": "NEW_TITLE",
    ":s": "NEW_SUBTITLE",
  },
};

docClient.update(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddbdoc_update.js
```

此示例代码可在 GitHub 上的此处找到。

查询表

此示例查询包含有关视频系列的剧集信息的表，返回第二季第九集之后，字幕中包含指定短语的每集的名称和字幕。

创建文件名为 `ddbdoc_query.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB.DocumentClient` 对象。创建一个 JSON 对象，其中包含查询表所需的参数，在本示例中包括表名，查询所需的 `ExpressionAttributeValues`，以及使用这些值定义查询要返回的项目的 `KeyConditionExpression`。调用 DynamoDB 文档客户端的 `query` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Items);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ddbdoc_query.js
```

此示例代码可在 GitHub 上的此处找到。

从表中删除项目

创建文件名为 `ddbdoc_delete.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 DynamoDB，请创建一个 `AWS.DynamoDB.DocumentClient` 对象。创建一个 JSON 对象，其中包含从表中删除某个项目所需的参数，在本示例中包括表的名称，以及所要删除项目的哈希键的名称和值。调用 DynamoDB 文档客户端的 `delete` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

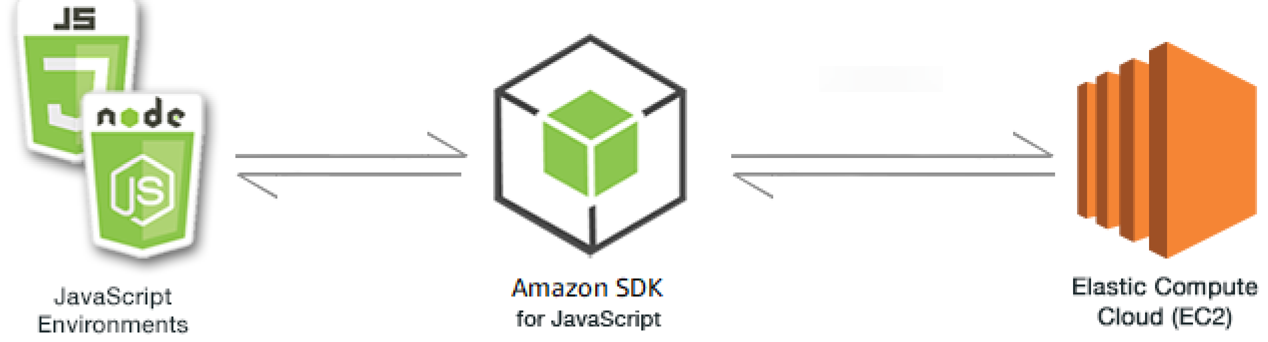
要运行示例，请在命令行中键入以下内容。

```
node ddbdoc_delete.js
```

此示例代码可在 GitHub 上的此处找到。

Amazon EC2 示例

Amazon Elastic Compute Cloud（Amazon EC2）是一项 Web 服务，可在云中提供虚拟服务器托管。该服务旨在通过提供大小可调整的计算容量来降低开发人员进行网络级云计算的难度。



适用于 Amazon EC2 的 JavaScript API 通过 `AWS.EC2` 客户端类公开。有关使用 Amazon EC2 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.EC2`。

主题

- 创建 Amazon EC2 实例
- 管理 Amazon EC2 实例
- 使用 &EC2; 密钥对
- 将区域和可用区用于 Amazon EC2
- 使用 Amazon EC2 中的安全组
- 在 Amazon EC2 中使用弹性 IP 地址

创建 Amazon EC2 实例

此 Node.js 代码示例演示：

- 如何从公有亚马逊机器映像（AMI）创建 Amazon EC2 实例。
- 如何创建标签并将其分配到新 Amazon EC2 实例。

关于示例

在本示例中，您使用 Node.js 模块来创建 Amazon EC2 实例，并将密钥对和标签分配到其上。代码使用 SDK for JavaScript，通过 Amazon EC2 客户端类的以下方法来创建和标记实例：

- `runInstances`
- `createTags`

先决条件任务

要设置和运行此示例，请先完成以下任务。

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建密钥对。有关详细信息，请参阅使用 &EC2; 密钥对。您可在此示例中使用密钥对的名称。

创建和标记实例

创建文件名为 `ec2_createinstances.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。

创建对象来传递 AWS.EC2 客户端类的 `runInstances` 方法的参数，包括要分配的密钥对的名称以及要运行的 AMI 的 ID。要调用 `runInstances` 方法，请创建一个 promise 来调用 Amazon EC2 服务对象并传递参数。然后处理 promise 回调中的 response。

接下来，代码将 Name 标签添加到新实例，Amazon EC2 控制台可识别标签并将其显示在实例列表的名称字段中。您可以添加最多 50 个标签添加到某个实例，所有标签可以通过调用一次 `createTags` 方法来添加。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// AMI is amzn-ami-2011.09.1.x86_64-ebs
var instanceParams = {
  ImageId: "AMI_ID",
  InstanceType: "t2.micro",
  KeyName: "KEY_PAIR_NAME",
  MinCount: 1,
  MaxCount: 1,
};

// Create a promise on an EC2 service object
var instancePromise = new AWS.EC2({ apiVersion: "2016-11-15" })
  .runInstances(instanceParams)
  .promise();

// Handle promise's fulfilled/rejected states
instancePromise
  .then(function (data) {
    console.log(data);
    var instanceId = data.Instances[0].InstanceId;
    console.log("Created instance", instanceId);
    // Add tags to the instance
    tagParams = {
      Resources: [instanceId],
```

```
      Tags: [
        {
          Key: "Name",
          Value: "SDK Sample",
        },
      ],
    };
    // Create a promise on an EC2 service object
    var tagPromise = new AWS.EC2({ apiVersion: "2016-11-15" })
      .createTags(tagParams)
      .promise();
    // Handle promise's fulfilled/rejected states
    tagPromise
      .then(function (data) {
        console.log("Instance tagged");
      })
      .catch(function (err) {
        console.error(err, err.stack);
      });
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_createinstances.js
```

此示例代码可在 GitHub 上的此处找到。

管理 Amazon EC2 实例

此 Node.js 代码示例演示：

- 如何检索有关 Amazon EC2 实例的基本信息。
- 如何启动和停止对 Amazon EC2 实例的详细监控。
- 如何启动和停止 Amazon EC2 实例。

- 如何重启 Amazon EC2 实例。

情景

在本示例中，您使用一系列 Node.js 模块执行多个基本实例管理操作。这些 Node.js 模块使用 SDK for JavaScript，通过以下 Amazon EC2 客户端类方法管理实例：

- `describeInstances`
- `monitorInstances`
- `unmonitorInstances`
- `startInstances`
- `stopInstances`
- `rebootInstances`

有关 Amazon EC2 实例的生命周期的更多信息，请参阅《Amazon EC2 用户指南》中的实例生命周期。

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建 Amazon EC2 实例。有关创建 Amazon EC2 实例的更多信息，请参阅《Amazon EC2 用户指南》中的 Amazon EC2 实例，或《Amazon EC2 用户指南》中的 Amazon EC2 实例。

描述实例

创建文件名为 `ec2_describeinstances.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。调用 Amazon EC2 服务对象的 `describeInstances` 方法来检索实例的详细说明。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  DryRun: false,
};

// Call EC2 to retrieve policy for selected bucket
ec2.describeInstances(params, function (err, data) {
  if (err) {
    console.log("Error", err.stack);
  } else {
    console.log("Success", JSON.stringify(data));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_describeinstances.js
```

此示例代码可在 GitHub 上的此处找到。

管理实例监控

创建文件名为 `ec2_monitorinstances.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。添加您要控制监控的实例的实例 ID。

根据命令行参数的值（`ON` 或 `OFF`），调用 Amazon EC2 服务对象的 `monitorInstances` 方法来启动对指定实例的详细监控，或者调用 `unmonitorInstances` 方法。使用 `DryRun` 参数，在您尝试更改这些实例的监控之前，测试您是否有权限更改实例监控。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
```

```
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "ON") {
  // Call EC2 to start monitoring the selected instances
  ec2.monitorInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.monitorInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.InstanceMonitorings);
        }
      });
    } else {
      console.log("You don't have permission to change instance monitoring.");
    }
  });
} else if (process.argv[2].toUpperCase() === "OFF") {
  // Call EC2 to stop monitoring the selected instances
  ec2.unmonitorInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.unmonitorInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.InstanceMonitorings);
        }
      });
    } else {
      console.log("You don't have permission to change instance monitoring.");
    }
  });
}
```

要运行此示例，请在命令行中键入以下内容，指定 ON 以启动详细监控，或者指定 OFF 以停止监控。

```
node ec2_monitorinstances.js ON
```

此示例代码可在 GitHub 上的此处找到。

启动和停止实例

创建文件名为 `ec2_startstopinstances.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。添加您要启动或停止的实例的实例 ID。

根据命令行参数的值（`START` 或 `STOP`），调用 Amazon EC2 服务对象的 `startInstances` 方法来启动指定实例，或者调用 `stopInstances` 方法来停止实例。首先使用 `DryRun` 参数来测试您是否有权限，然后再尝试启动或停止所选实例。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: [process.argv[3]],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "START") {
  // Call EC2 to start the selected instances
  ec2.startInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.startInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StartingInstances);
        }
      });
    } else {
      console.log("You don't have permission to start instances.");
    }
  });
} else if (process.argv[2].toUpperCase() === "STOP") {
  // Call EC2 to stop the selected instances
  ec2.stopInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
```

```
      ec2.stopInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StoppingInstances);
        }
      });
    } else {
      console.log("You don't have permission to stop instances");
    }
  });
}
```

要运行此示例，请在命令行中键入以下内容，指定 START 以启动实例，或者指定 STOP 以停止实例。

```
node ec2_startstopinstances.js START INSTANCE_ID
```

此示例代码可在 GitHub 上的此处找到。

重启实例

创建文件名为 `ec2_rebootinstances.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 Amazon EC2 服务对象。添加您要重启的实例的实例 ID。调用 `AWS.EC2` 服务对象的 `rebootInstances` 方法来重启指定的实例。首先使用 `DryRun` 参数来测试您是否有权限重启这些实例，然后再尝试重启它们。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

// Call EC2 to reboot instances
ec2.rebootInstances(params, function (err, data) {
  if (err && err.code === "DryRunOperation") {
```

```
    params.DryRun = false;
    ec2.rebootInstances(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else if (data) {
        console.log("Success", data);
      }
    });
  } else {
    console.log("You don't have permission to reboot instances.");
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_rebootinstances.js
```

此示例代码可在 GitHub 上的此处找到。

使用 &EC2; 密钥对

此 Node.js 代码示例演示：

- 如何检索有关密钥对的信息。
- 如何创建密钥对来访问 Amazon EC2 实例。
- 如何删除现有密钥对。

情景

Amazon EC2 使用公有密钥密码系统来加密和解密登录信息。公有密钥密码系统使用公有密钥加密数据，然后收件人可以使用私有密钥解密数据。公有和私有密钥被称为密钥对。

在本示例中，您使用一系列 Node.js 模块执行多个 Amazon EC2 密钥对管理操作。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon EC2 客户端类的以下方法来管理实例：

- `createKeyPair`

- deleteKeyPair
- describeKeyPairs

有关 Amazon EC2 密钥对的更多信息，请参阅《Amazon EC2 用户指南》中的 Amazon EC2 密钥对，或《Amazon EC2 用户指南》中的 Amazon EC2 密钥对和 Windows 实例。

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

描述密钥对

创建文件名为 ec2_describekeypairs.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 AWS.EC2 服务对象。创建空的 JSON 对象，以保存 describeKeyPairs 方法为返回全部密钥对的描述所需的参数。您还可以在 JSON 文件的参数的 KeyName 部分中将密钥对名称数组提供给 describeKeyPairs 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// Retrieve key pair descriptions; no params needed
ec2.describeKeyPairs(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.KeyPairs));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_describekeypairs.js
```

此示例代码可在 GitHub 上的此处找到。

创建密钥对

每个密钥对需要一个名称。Amazon EC2 将公有密钥与您指定的密钥名称相关联。创建文件名为 `ec2_createkeypair.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。创建 JSON 参数，指定密钥对的名称，然后传递它们来调用 `createKeyPair` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Create the key pair
ec2.createKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log(JSON.stringify(data));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_createkeypair.js
```

此示例代码可在 GitHub 上的此处找到。

删除密钥对

创建文件名为 `ec2_deletekeypair.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。创建 JSON 参数，指定您要删除的密钥对的名称。然后调用 `deleteKeyPair` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Delete the key pair
ec2.deleteKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Key Pair Deleted");
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_deletekeypair.js
```

此示例代码可在 GitHub 上的此处找到。

将区域和可用区用于 Amazon EC2

此 Node.js 代码示例演示：

- 如何检索区域和可用区的说明。

情景

Amazon EC2 托管在全球多个位置。这些位置由 区域和可用区构成。每个 区域 都是一个单独的地理区域。每个区域都有多个相互隔离的位置，称为可用区。Amazon EC2 提供了将实例和数据放在多个位置的功能。

在本示例中，您使用一系列 Node.js 模块检索有关区域和可用区的详细信息。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon EC2 客户端类的以下方法来管理实例：

- `describeAvailabilityZones`
- `describeRegions`

有关区域和可用区的更多信息，请参阅《Amazon EC2 用户指南》中的区域和可用区，或《Amazon EC2 用户指南》中的区域和可用区。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

描述区域和可用区域

创建文件名为 `ec2_describeregionsandzones.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。创建空 JSON 对象作为参数传递，这会返回所有可用说明。然后调用 `describeRegions` 和 `describeAvailabilityZones` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });
```

```
var params = {};

// Retrieves all regions/endpoints that work with EC2
ec2.describeRegions(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Regions: ", data.Regions);
  }
});

// Retrieves availability zones only for region of the ec2 service object
ec2.describeAvailabilityZones(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Availability Zones: ", data.AvailabilityZones);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_describeregionsandzones.js
```

此示例代码可在 GitHub 上的此处找到。

使用 Amazon EC2 中的安全组

此 Node.js 代码示例演示：

- 如何检索有关安全组的信息。
- 如何创建安全组以访问 Amazon EC2 实例。
- 如何删除现有安全组。

情景

Amazon EC2 安全组起着虚拟防火墙的作用，可控制一个或多个实例的流量。为每个安全组添加规则来规定流入或流出其关联实例的流量。您可以随时修改安全组的规则；新规则会自动应用于与该安全组关联的所有实例。

在本示例中，您使用一系列 Node.js 模块执行涉及到安全组的多个 Amazon EC2 操作。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon EC2 客户端类的以下方法来管理实例：

- `describeSecurityGroups`
- `authorizeSecurityGroupIngress`
- `createSecurityGroup`
- `describeVpcs`
- `deleteSecurityGroup`

有关 Amazon EC2 安全组的更多信息，请参阅《Amazon EC2 用户指南》中的适用于 Linux 实例的 Amazon EC2 安全组，或《Amazon EC2 用户指南》中的适用于 Windows 实例的 Amazon EC2 安全组。

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

描述您的安全组

创建文件名为 `ec2_describesecuritygroups.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。创建 JSON 对象作为参数传递，其中包括您要描述的安全组的组 ID。然后调用 Amazon EC2 服务对象的 `describeSecurityGroups` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  GroupIds: ["SECURITY_GROUP_ID"],
};

// Retrieve security group descriptions
ec2.describeSecurityGroups(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.SecurityGroups));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_describesecuritygroups.js
```

此示例代码可在 [GitHub 上的此处](#)找到。

创建安全组和规则

创建文件名为 `ec2_createsecuritygroup.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。为参数创建 JSON 对象，这些参数指定安全组的名称、描述以及 VPC 的 ID。将参数传递到 `createSecurityGroup` 方法。

成功创建安全组后，您可以定义规则以允许入站流量。为参数创建 JSON 对象，这些参数指定 IP 协议以及 Amazon EC2 实例接收流量的入站端口。将参数传递到 `authorizeSecurityGroupIngress` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// Variable to hold a ID of a VPC
var vpc = null;
```

```
// Retrieve the ID of a VPC
ec2.describeVpcs(function (err, data) {
  if (err) {
    console.log("Cannot retrieve a VPC", err);
  } else {
    vpc = data.Vpcs[0].VpcId;
    var paramsSecurityGroup = {
      Description: "DESCRIPTION",
      GroupName: "SECURITY_GROUP_NAME",
      VpcId: vpc,
    };
    // Create the instance
    ec2.createSecurityGroup(paramsSecurityGroup, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        var SecurityGroupId = data.GroupId;
        console.log("Success", SecurityGroupId);
        var paramsIngress = {
          GroupId: "SECURITY_GROUP_ID",
          IpPermissions: [
            {
              IpProtocol: "tcp",
              FromPort: 80,
              ToPort: 80,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
            {
              IpProtocol: "tcp",
              FromPort: 22,
              ToPort: 22,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
          ],
        };
        ec2.authorizeSecurityGroupIngress(paramsIngress, function (err, data) {
          if (err) {
            console.log("Error", err);
          } else {
            console.log("Ingress Successfully Set", data);
          }
        });
      }
```

```
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_createsecuritygroup.js
```

此示例代码可在 GitHub 上的此处找到。

正在删除安全组

创建文件名为 `ec2_deletesecuritygroup.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。创建 JSON 参数，指定您要删除的安全组的名称。然后调用 `deleteSecurityGroup` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  GroupId: "SECURITY_GROUP_ID",
};

// Delete the security group
ec2.deleteSecurityGroup(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Security Group Deleted");
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_deletesecuritygroup.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon EC2 中使用弹性 IP 地址

此 Node.js 代码示例演示：

- 如何检索您的弹性 IP 地址的描述。
- 如何分配和释放弹性 IP 地址。
- 如何将弹性 IP 地址与 Amazon EC2 实例关联。

情景

弹性 IP 地址 是专为动态云计算设计的静态 IP 地址。弹性 IP 地址与您的 Amazon 账户关联。它是公有 IP 地址，可从 Internet 访问。如果您的实例没有公有 IP 地址，则可以将弹性 IP 地址与您的实例关联以启用与 Internet 的通信。

在本示例中，您使用一系列 Node.js 模块执行涉及到弹性 IP 地址的多个 Amazon EC2 操作。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon EC2 客户端类的以下方法来管理弹性 IP 地址：

- `describeAddresses`
- `allocateAddress`
- `associateAddress`
- `releaseAddress`

有关 Amazon EC2 中的弹性 IP 地址的更多信息，请参阅《Amazon EC2 用户指南》中的弹性 IP 地址或《Amazon EC2 用户指南》中的弹性 IP 地址。

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

- 创建 Amazon EC2 实例。有关创建 Amazon EC2 实例的更多信息，请参阅《Amazon EC2 用户指南》中的 Amazon EC2 实例，或《Amazon EC2 用户指南》中的 Amazon EC2 实例。

描述弹性 IP 地址

创建文件名为 ec2_describeaddresses.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 AWS.EC2 服务对象。创建 JSON 对象作为参数传递，在 VPC 中筛选这些参数返回的地址。要检索所有弹性 IP 地址的说明，请忽略来自参数 JSON 的筛选条件。然后调用 Amazon EC2 服务对象的 describeAddresses 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  Filters: [{ Name: "domain", Values: ["vpc"] }],
};

// Retrieve Elastic IP address descriptions
ec2.describeAddresses(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.Addresses));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_describeaddresses.js
```

此示例代码可在 GitHub 上的此处找到。

分配弹性 IP 地址并将其与 Amazon EC2 实例关联

创建文件名为 ec2_allocateaddress.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 AWS.EC2 服务对象。为用于分配弹性 IP 地址的参数创建 JSON 对象，在此例中指定 Domain 为 VPC。调用 Amazon EC2 服务对象的 allocateAddress 方法。

如果调用成功，回调函数的 data 参数具有 AllocationId 属性，标识已分配的弹性 IP 地址。

为将弹性 IP 地址关联到 Amazon EC2 实例所用的参数创建一个 JSON 对象，其中包括新分配地址的 AllocationId 以及 Amazon EC2 实例的 InstanceId。然后调用 Amazon EC2 服务对象的 associateAddresses 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsAllocateAddress = {
  Domain: "vpc",
};

// Allocate the Elastic IP address
ec2.allocateAddress(paramsAllocateAddress, function (err, data) {
  if (err) {
    console.log("Address Not Allocated", err);
  } else {
    console.log("Address allocated:", data.AllocationId);
    var paramsAssociateAddress = {
      AllocationId: data.AllocationId,
      InstanceId: "INSTANCE_ID",
    };
    // Associate the new Elastic IP address with an EC2 instance
    ec2.associateAddress(paramsAssociateAddress, function (err, data) {
      if (err) {
        console.log("Address Not Associated", err);
      } else {
        console.log("Address associated:", data.AssociationId);
      }
    });
  }
```

```
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_allocateaddress.js
```

此示例代码可在 GitHub 上的此处找到。

释放弹性 IP 地址

创建文件名为 `ec2_releaseaddress.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon EC2，请创建 `AWS.EC2` 服务对象。为用于释放弹性 IP 地址的参数创建 JSON 对象，在本例中为弹性 IP 地址指定 `AllocationId`。释放弹性 IP 地址还会将其与任何 Amazon EC2 实例取消关联。调用 Amazon EC2 服务对象的 `releaseAddress` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsReleaseAddress = {
  AllocationId: "ALLOCATION_ID",
};

// Disassociate the Elastic IP address from EC2 instance
ec2.releaseAddress(paramsReleaseAddress, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Address released");
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_releaseaddress.js
```

此示例代码可在 GitHub 上的此处找到。

AWS Elemental MediaConvert 示例

AWS Elemental MediaConvert 是基于文件的视频转码服务与广播级的功能。您可以使用它来创建资产广播并为视频点播 (VOD) 交付整个 Internet。有关更多信息，请参阅 AWS Elemental MediaConvert 用户指南。

适用于 MediaConvert 的 JavaScript API 通过 `AWS.MediaConvert` 客户端类公开。有关更多信息，请参阅 API 参考中的 `Class: AWS.MediaConvert`。

主题

- 在 MediaConvert 中创建和管理转码作业
- 在 MediaConvert 中使用作业模板

在 MediaConvert 中创建和管理转码作业

此 Node.js 代码示例演示：

- 如何在 MediaConvert 中创建转码作业。
- 如何取消转码作业。
- 如何检索已完成转码作业的 JSON。
- 如何检索最多 20 个最新创建的作业的 JSON 数组。

情景

在此示例中，您使用 Node.js 模块调用 MediaConvert 来创建和管理转码作业。该代码使用 SDK for JavaScript，通过 MediaConvert 客户端类的以下方法来完成此操作：

- `createJob`
- `cancelJob`
- `getJob`
- `listJobs`

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建和配置 Amazon S3 桶，提供作业输入文件和输出文件的存储。有关详细信息，请参阅《AWS Elemental MediaConvert User Guide》中的 Create Storage for Files。
- 将输入视频上传到您为输入存储预置的 Amazon S3 存储桶。有关支持的输入视频编解码器和容器的列表，请参阅《AWS Elemental MediaConvert User Guide》中的 Supported Input Codecs and Containers。
- 创建一个 IAM 角色，该角色使 MediaConvert 能够访问输入文件以及存储输出文件的 Amazon S3 存储桶。有关详细信息，请参阅《AWS Elemental MediaConvert User Guide》中的 Set Up IAM Permissions。

定义简单的转码任务

创建文件名为 `emc_createjob.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。创建定义转码任务参数的 JSON。

这些参数有非常详细的说明。您可以使用 AWS Elemental MediaConvert 控制台生成 JSON 作业参数，方法是在控制台中选择您的作业设置，然后选择作业部分底部的显示作业 JSON。本示例说明了简单作业的 JSON。

```
var params = {
  Queue: "JOB_QUEUE_ARN",
  UserMetadata: {
    Customer: "Amazon",
  },
  Role: "IAM_ROLE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://OUTPUT_BUCKET_NAME/",
```

```
          },
        },
        Outputs: [
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
                Codec: "H_264",
                H264Settings: {
                  InterlaceMode: "PROGRESSIVE",
                  NumberReferenceFrames: 3,
                  Syntax: "DEFAULT",
                  Softness: 0,
                  GopClosedCadence: 1,
                  GopSize: 90,
                  Slices: 1,
                  GopBReference: "DISABLED",
                  SlowPal: "DISABLED",
                  SpatialAdaptiveQuantization: "ENABLED",
                  TemporalAdaptiveQuantization: "ENABLED",
                  FlickerAdaptiveQuantization: "DISABLED",
                  EntropyEncoding: "CABAC",
                  Bitrate: 5000000,
                  FramerateControl: "SPECIFIED",
                  RateControlMode: "CBR",
                  CodecProfile: "MAIN",
                  Telecine: "NONE",
                  MinIInterval: 0,
                  AdaptiveQuantization: "HIGH",
                  CodecLevel: "AUTO",
                  FieldEncoding: "PAFF",
                  SceneChangeDetect: "ENABLED",
                  QualityTuningLevel: "SINGLE_PASS",
                  FramerateConversionAlgorithm: "DUPLICATE_DROP",
                  UnregisteredSeiTimecode: "DISABLED",
                  GopSizeUnits: "FRAMES",
                  ParControl: "SPECIFIED",
                  NumberBFramesBetweenReferenceFrames: 2,
                  RepeatPps: "DISABLED",
                  FramerateNumerator: 30,
                  FramerateDenominator: 1,
```

```
                ParNumerator: 1,
                ParDenominator: 1,
              },
            },
            AfdSignaling: "NONE",
            DropFrameTimecode: "ENABLED",
            RespondToAfd: "NONE",
            ColorMetadata: "INSERT",
          },
          AudioDescriptions: [
            {
              AudioTypeControl: "FOLLOW_INPUT",
              CodecSettings: {
                Codec: "AAC",
                AacSettings: {
                  AudioDescriptionBroadcasterMix: "NORMAL",
                  RateControlMode: "CBR",
                  CodecProfile: "LC",
                  CodingMode: "CODING_MODE_2_0",
                  RawFormat: "NONE",
                  SampleRate: 48000,
                  Specification: "MPEG4",
                  Bitrate: 64000,
                },
              },
              LanguageCodeControl: "FOLLOW_INPUT",
              AudioSourceName: "Audio Selector 1",
            },
          ],
          ContainerSettings: {
            Container: "MP4",
            Mp4Settings: {
              CslgAtom: "INCLUDE",
              FreeSpaceBox: "EXCLUDE",
              MoovPlacement: "PROGRESSIVE_DOWNLOAD",
            },
          },
          NameModifier: "_1",
        },
      ],
    },
  ],
  AdAvailOffset: 0,
  Inputs: [
```

```
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "s3://INPUT_BUCKET_AND_FILE_NAME",
      },
    ],
    TimecodeConfig: {
      Source: "EMBEDDED",
    },
  },
};
```

创建转码作业

在创建作业参数 JSON 后，通过创建 promise 来调用 `createJob` 服务对象并传递参数，以此调用 `AWS.MediaConvert` 方法。然后处理 promise 回调中的 response。所创建作业的 ID 在响应 `data` 中返回。

```
// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job created! ", data);
```

```
  },
  function (err) {
    console.log("Error", err);
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_createjob.js
```

此示例代码可在 GitHub 上的此处找到。

取消转码作业

创建文件名为 emc_canceljob.js 的 Node.js 模块。请确保按前面所示配置开发工具包。创建包含要取消的作业的 ID 的 JSON。然后，通过创建一个 promise 来调用 cancelJob 服务对象并传递参数，以此调用 AWS.MediaConvert 方法。承诺处理响应中的回调。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set MediaConvert to customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Id: "JOB_ID" /* required */,
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .cancelJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job  " + params.Id + " is canceled");
  },
  function (err) {
    console.log("Error", err);
  }
```

```
);
```

要运行示例，请在命令行中键入以下内容。

```
node ec2_canceljob.js
```

此示例代码可在 GitHub 上的此处找到。

列出最近的转码任务

创建文件名为 emc_listjobs.js 的 Node.js 模块。请确保按前面所示配置开发工具包。

创建包括值的参数 JSON，这些值指定是按 ASCENDING 还是 DESCENDING 对列表排序、要检查的作业队列的 ARN，以及要包含的作业的状态。然后，通过创建一个 promise 来调用 listJobs 服务对象并传递参数，以此调用 AWS.MediaConvert 方法。承诺处理响应中的回调。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  MaxResults: 10,
  Order: "ASCENDING",
  Queue: "QUEUE_ARN",
  Status: "SUBMITTED",
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobs(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Jobs: ", data);
  },
  function (err) {
    console.log("Error", err);
```

```
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_listjobs.js
```

此示例代码可在 GitHub 上的此处找到。

在 MediaConvert 中使用作业模板

此 Node.js 代码示例演示：

- 如何创建 MediaConvert 作业模板。
- 如何使用作业模板来创建转码作业。
- 如何列出您的所有作业模板。
- 如何删除作业模板。

情景

在 MediaConvert 中创建转码作业所需的 JSON 有详细说明，包含大量设置。您可以将已知工作正常的设置保存在作业模板中并用于创建以后的作业，从而节省大量时间。在此示例中，您使用 Node.js 模块调用 MediaConvert 来创建、使用和管理作业模板。该代码使用 SDK for JavaScript，通过 MediaConvert 客户端类的以下方法来完成此操作：

- `createJobTemplate`
- `createJob`
- `deleteJobTemplate`
- `listJobTemplates`

先决条件任务

要设置和运行此示例，请先完成以下任务：

- 安装 Node.js。有关更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个 IAM 角色，该角色使 MediaConvert 能够访问输入文件以及存储输出文件的 Amazon S3 桶。有关详细信息，请参阅《AWS Elemental MediaConvert User Guide》中的 Set Up IAM Permissions。

创建作业模板

创建文件名为 `emc_create_jobtemplate.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。

指定用于创建模板的参数 JSON。您可以使用来自以前成功作业中的大部分 JSON 参数来指定模板中的 `Settings` 值。此示例使用来自 在 MediaConvert 中创建和管理转码作业 的作业设置。

通过创建一个 promise 来调用 `createJobTemplate` 服务对象并传递参数，以此调用 `AWS.MediaConvert` 方法。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Category: "YouTube Jobs",
  Description: "Final production transcode",
  Name: "DemoTemplate",
  Queue: "JOB_QUEUE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://BUCKET_NAME/",
          },
        },
        Outputs: [
```

```
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
                Codec: "H_264",
                H264Settings: {
                  InterlaceMode: "PROGRESSIVE",
                  NumberReferenceFrames: 3,
                  Syntax: "DEFAULT",
                  Softness: 0,
                  GopClosedCadence: 1,
                  GopSize: 90,
                  Slices: 1,
                  GopBReference: "DISABLED",
                  SlowPal: "DISABLED",
                  SpatialAdaptiveQuantization: "ENABLED",
                  TemporalAdaptiveQuantization: "ENABLED",
                  FlickerAdaptiveQuantization: "DISABLED",
                  EntropyEncoding: "CABAC",
                  Bitrate: 5000000,
                  FramerateControl: "SPECIFIED",
                  RateControlMode: "CBR",
                  CodecProfile: "MAIN",
                  Telecine: "NONE",
                  MinIInterval: 0,
                  AdaptiveQuantization: "HIGH",
                  CodecLevel: "AUTO",
                  FieldEncoding: "PAFF",
                  SceneChangeDetect: "ENABLED",
                  QualityTuningLevel: "SINGLE_PASS",
                  FramerateConversionAlgorithm: "DUPLICATE_DROP",
                  UnregisteredSeiTimecode: "DISABLED",
                  GopSizeUnits: "FRAMES",
                  ParControl: "SPECIFIED",
                  NumberBFramesBetweenReferenceFrames: 2,
                  RepeatPps: "DISABLED",
                  FramerateNumerator: 30,
                  FramerateDenominator: 1,
                  ParNumerator: 1,
                  ParDenominator: 1,
                },
```

```
              },
              AfdSignaling: "NONE",
              DropFrameTimecode: "ENABLED",
              RespondToAfd: "NONE",
              ColorMetadata: "INSERT",
            },
            AudioDescriptions: [
              {
                AudioTypeControl: "FOLLOW_INPUT",
                CodecSettings: {
                  Codec: "AAC",
                  AacSettings: {
                    AudioDescriptionBroadcasterMix: "NORMAL",
                    RateControlMode: "CBR",
                    CodecProfile: "LC",
                    CodingMode: "CODING_MODE_2_0",
                    RawFormat: "NONE",
                    SampleRate: 48000,
                    Specification: "MPEG4",
                    Bitrate: 64000,
                  },
                },
                LanguageCodeControl: "FOLLOW_INPUT",
                AudioSourceName: "Audio Selector 1",
              },
            ],
            ContainerSettings: {
              Container: "MP4",
              Mp4Settings: {
                CslgAtom: "INCLUDE",
                FreeSpaceBox: "EXCLUDE",
                MoovPlacement: "PROGRESSIVE_DOWNLOAD",
              },
            },
            NameModifier: "_1",
          },
        ],
      },
    ],
    AdAvailOffset: 0,
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
```

```
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
      },
    ],
    TimecodeConfig: {
      Source: "EMBEDDED",
    },
  },
};

// Create a promise on a MediaConvert object
var templatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
templatePromise.then(
  function (data) {
    console.log("Success!", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_create_jobtemplate.js
```

此示例代码可在 GitHub 上的此处找到。

从作业模板创建转码作业

创建文件名为 `emc_template_createjob.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。

创建作业创建参数 JSON，其中包括要使用的作业模板名称，以及所要使用的特定于您正在创建的作业的 `Settings`。然后，通过创建一个 promise 来调用 `createJobs` 服务对象并传递参数，以此调用 `AWS.MediaConvert` 方法。承诺处理响应中的回调。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Queue: "QUEUE_ARN",
  JobTemplate: "TEMPLATE_NAME",
  Role: "ROLE_ARN",
  Settings: {
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
```

```
          FileInput: "s3://BUCKET_NAME/FILE_NAME",
        },
      ],
    },
};

// Create a promise on a MediaConvert object
var templateJobPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
templateJobPromise.then(
  function (data) {
    console.log("Success! ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_template_createjob.js
```

此示例代码可在 GitHub 上的此处找到。

列出作业模板

创建文件名为 `emc_listtemplates.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。

创建一个对象以传递 `listTemplates` 客户端类的 `AWS.MediaConvert` 方法的请求参数。包含值以确定要列出哪些模板（`NAME`、`CREATION DATE`、`SYSTEM`）、要列出多少个模板及其排序顺序。要调用 `listTemplates` 方法，请创建一个 promise 来调用 MediaConvert 服务对象并传递参数。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };
```

```
var params = {
  ListBy: "NAME",
  MaxResults: 10,
  Order: "ASCENDING",
};

// Create a promise on a MediaConvert object
var listTemplatesPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobTemplates(params)
  .promise();

// Handle promise's fulfilled/rejected status
listTemplatesPromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_listtemplates.js
```

此示例代码可在 GitHub 上的此处找到。

删除作业模板

创建文件名为 `emc_deletetemplate.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。

创建一个对象，以将您要删除的作业模板的名称作为 `deleteJobTemplate` 客户端类的 `AWS.MediaConvert` 方法的参数传递。要调用 `deleteJobTemplate` 方法，请创建一个 promise 来调用 MediaConvert 服务对象并传递参数。承诺处理响应中的回调。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };
```

```
var params = {
  Name: "TEMPLATE_NAME",
};

// Create a promise on a MediaConvert object
var deleteTemplatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .deleteJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
deleteTemplatePromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node emc_deletetemplate.js
```

此示例代码可在 GitHub 上的此处找到。

Amazon IAM 示例

Amazon Identity and Access Management（IAM）是一项 Web 服务，支持 Amazon Web Services 客户在 Amazon 中管理用户和用户权限。该服务面向在云中有多个使用 Amazon 产品的用户或系统的组织。借助 IAM，您可以集中管理用户、安全凭证（如访问密钥），以及控制用户可访问何种 Amazon 资源的权限。

适用于 IAM 的 JavaScript API 通过 `AWS.IAM` 客户端类公开。有关使用 IAM 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.IAM`。

主题

- 管理 IAM 用户
- 使用 IAM 策略
- 管理 IAM 访问密钥
- 使用 IAM 服务器证书
- 管理 IAM 账户别名

管理 IAM 用户

此 Node.js 代码示例演示：

- 如何检索 IAM 用户列表。
- 如何创建和删除用户。
- 如何更新用户名。

情景

本示例使用一系列 Node.js 模块在 IAM 中创建和管理用户。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.IAM` 客户端类的以下方法创建、删除和更新用户：

- `createUser`
- `listUsers`
- `updateUser`
- `getUser`
- `deleteUser`

有关 IAM 用户的更多信息，请参阅《IAM 用户指南》中的 IAM 用户。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建用户

创建文件名为 `iam_createuser.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含所需参数的 JSON 对象，该对象包含要用于新用户的用户名作为命令行参数。

调用 `getUser` 服务对象的 `AWS.IAM` 方法，查看该用户名是否已存在。如果用户名称当前不存在，请调用 `createUser` 方法来创建它。如果名称已存在，则将说明该结果的消息写入控制台。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
};

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    iam.createUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  } else {
    console.log(
      "User " + process.argv[2] + " already exists",
```

```
      data.User.UserId
    );
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_createuser.js USER_NAME
```

此示例代码可在 GitHub 上的此处找到。

列出您的账户中的用户

创建文件名为 `iam_listusers.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含列出用户时所需参数的 JSON 对象，通过将 `MaxItems` 参数设置为 10 来限制返回的数量。调用 `listUsers` 服务对象的 `AWS.IAM` 方法。将第一个用户的名称和创建日期写入控制台。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 10,
};

iam.listUsers(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var users = data.Users || [];
    users.forEach(function (user) {
      console.log("User " + user.UserName + " created", user.CreateDate);
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_listusers.js
```

此示例代码可在 GitHub 上的此处找到。

更新用户名

创建文件名为 iam_updateuser.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 AWS.IAM 服务对象。创建一个包含列出用户时所需参数的 JSON 对象，并将当前用户名和新用户名都指定为命令行参数。调用 updateUser 服务对象的 AWS.IAM 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
  NewUserName: process.argv[3],
};

iam.updateUser(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行该示例，请在命令行中键入以下内容，指定用户的当前名称，后跟新用户名。

```
node iam_updateuser.js ORIGINAL_USERNAME NEW_USERNAME
```

此示例代码可在 GitHub 上的此处找到。

删除用户

创建文件名为 `iam_deleteuser.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含所需参数的 JSON 对象，该对象包含要删除的用户名作为命令行参数。

调用 `getUser` 服务对象的 `AWS.IAM` 方法，查看该用户名是否已存在。如果该用户名当前不存在，则将说明该结果的消息写入控制台。如果用户存在，则调用 `deleteUser` 方法删除该用户。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
};

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    console.log("User " + process.argv[2] + " does not exist.");
  } else {
    iam.deleteUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_deleteuser.js USER_NAME
```

此示例代码可在 GitHub 上的此处找到。

使用 IAM 策略

此 Node.js 代码示例演示：

- 如何创建和删除 IAM policy。
- 如何对角色附加和分离 IAM policy。

情景

您可通过创建策略（此文档列出用户可执行的操作以及操作可以影响的资源）向用户授予权限。默认情况下会拒绝未显式允许的任何操作或资源。可将策略附加到用户、用户组、用户代入的角色以及资源。

本示例使用一系列 Node.js 模块在 IAM 中管理策略。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.IAM` 客户端类的以下方法来创建和删除策略，以及附加和分离角色策略：

- `createPolicy`
- `getPolicy`
- `listAttachedRolePolicies`
- `attachRolePolicy`
- `detachRolePolicy`

有关 IAM 用户的更多信息，请参阅《IAM 用户指南》中的访问管理概述：权限和策略。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建一个您可以向其附加策略 IAM 角色。有关创建角色的更多信息，请参阅《IAM 用户指南》中的创建 IAM 角色。

创建 IAM Policy

创建文件名为 `iam_createpolicy.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建两个 JSON 对象，一个包含要创建的策略文档，另一个包含创建策略时所需的参数，该策略包括策略 JSON 以及要为策略指定的名称。确保在参数中对策略 JSON 对象进行字符串化。调用 `createPolicy` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var myManagedPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Effect: "Allow",
      Action: "logs:CreateLogGroup",
      Resource: "RESOURCE_ARN",
    },
    {
      Effect: "Allow",
      Action: [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
      ],
      Resource: "RESOURCE_ARN",
    },
  ],
};

var params = {
  PolicyDocument: JSON.stringify(myManagedPolicy),
  PolicyName: "myDynamoDBPolicy",
};

iam.createPolicy(params, function (err, data) {
```

```
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_createpolicy.js
```

此示例代码可在 GitHub 上的此处找到。

获取 IAM Policy

创建文件名为 `iam_getpolicy.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含检索策略时所需参数的 JSON 对象，该对象是您要获取的策略的 ARN。调用 `getPolicy` 服务对象的 `AWS.IAM` 方法。将策略描述写入控制台。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  PolicyArn: "arn:aws:iam::aws:policy/AWSLambdaExecute",
};

iam.getPolicy(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Policy.Description);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_getpolicy.js
```

此示例代码可在 GitHub 上的此处找到。

附加托管的角色策略

创建文件名为 iam_attachrolepolicy.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 AWS.IAM 服务对象。创建一个包含获取附加到角色的托管 IAM policy 列表时所需参数的 JSON 对象，其中包括角色名称。提供角色名称作为命令行参数。调用 listAttachedRolePolicies 服务对象的 AWS.IAM 方法，该方法向回调函数返回托管策略的数组。

检查数组成员，查看要附加到角色的策略是否已附加。如果策略未附加，则调用 attachRolePolicy 方法附加该策略。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var paramsRoleList = {
  RoleName: process.argv[2],
};

iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        console.log(
          "AmazonDynamoDBFullAccess is already attached to this role."
        );
        process.exit();
      }
    });
    var params = {
      PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
      RoleName: process.argv[2],
    };
    iam.attachRolePolicy(params, function (err, data) {
```

```
      if (err) {
        console.log("Unable to attach policy to role", err);
      } else {
        console.log("Role attached successfully");
      }
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_attachrolepolicy.js IAM_ROLE_NAME
```

分离托管的角色策略

创建文件名为 `iam_detachrolepolicy.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含获取附加到角色的托管 IAM policy 列表时所需参数的 JSON 对象，其中包括角色名称。提供角色名称作为命令行参数。调用 `listAttachedRolePolicies` 服务对象的 `AWS.IAM` 方法，该方法在回调函数中返回托管策略的数组。

检查数组成员，查看要从角色分离的策略是否已附加。如果策略已附加，则调用 `detachRolePolicy` 方法分离该策略。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var paramsRoleList = {
  RoleName: process.argv[2],
};

iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
```

```
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        var params = {
          PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
          RoleName: process.argv[2],
        };
        iam.detachRolePolicy(params, function (err, data) {
          if (err) {
            console.log("Unable to detach policy from role", err);
          } else {
            console.log("Policy detached from role successfully");
            process.exit();
          }
        });
      }
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_detachrolepolicy.js IAM_ROLE_NAME
```

管理 IAM 访问密钥

此 Node.js 代码示例演示：

- 如何管理您的用户的访问密钥。

情景

用户需要自己的访问密钥从 SDK for JavaScript 以编程方式调用 Amazon。要满足这一需要，您可以创建、修改、查看或轮换 IAM 用户的访问密钥 (访问密钥 ID 和秘密访问密钥)。默认情况下，当您创建访问密钥时，其状态为 `Active`，这意味着用户可使用该访问密钥执行 API 调用。

本示例使用一系列 Node.js 模块在 IAM 中管理访问密钥。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.IAM` 客户端类的以下方法来管理 IAM 访问密钥：

- createAccessKey
- listAccessKeys
- getAccessKeyLastUsed
- updateAccessKey
- deleteAccessKey

有关 IAM 访问密钥的更多信息，请参阅《IAM 用户指南》中的访问密钥。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

为用户创建访问密钥

创建文件名为 `iam_createaccesskeys.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含创建新访问密钥时所需参数的 JSON 对象，其中包括 IAM 用户的名称。调用 `createAccessKey` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.createAccessKey({ UserName: "IAM_USER_NAME" }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.AccessKey);
  }
});
```

要运行示例，请在命令行中键入以下内容。确保将返回的数据通过管道传输到文本文件中，以免丢失私有密钥，该密钥只能提供一次。

```
node iam_createaccesskeys.js > newuserkeys.txt
```

此示例代码可在 GitHub 上的此处找到。

列出用户的访问密钥

创建文件名为 iam_listaccesskeys.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 AWS.IAM 服务对象。创建一个包含检索用户访问密钥时所需参数的 JSON 对象，其中包括 IAM 用户的名称以及您希望列出的最大访问密钥对数（可选）。调用 listAccessKeys 服务对象的 AWS.IAM 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 5,
  UserName: "IAM_USER_NAME",
};

iam.listAccessKeys(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_listaccesskeys.js
```

此示例代码可在 GitHub 上的此处找到。

获取访问密钥的最后一次使用

创建文件名为 `iam_accesskeylastused.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含创建新访问密钥时所需参数的 JSON 对象，这是您希望获得最后一次使用信息的访问密钥 ID。调用 `getAccessKeyLastUsed` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.getAccessKeyLastUsed(
  { AccessKeyId: "ACCESS_KEY_ID" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data.AccessKeyLastUsed);
    }
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node iam_accesskeylastused.js
```

此示例代码可在 GitHub 上的此处找到。

更新访问密钥状态

创建文件名为 `iam_updateaccesskey.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含更新访问密钥状态时所需参数的 JSON 对象，其中包括访问密钥 ID 和更新后的状态。状态可以为 `Active` 或 `Inactive`。调用 `updateAccessKey` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  Status: "Active",
  UserName: "USER_NAME",
};

iam.updateAccessKey(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_updateaccesskey.js
```

此示例代码可在 GitHub 上的此处找到。

删除访问密钥

创建文件名为 `iam_deleteaccesskey.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含删除访问密钥时所需参数的 JSON 对象，其中包括访问密钥 ID 和用户的名称。调用 `deleteAccessKey` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });
```

```
var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  UserName: "USER_NAME",
};

iam.deleteAccessKey(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_deleteaccesskey.js
```

此示例代码可在 GitHub 上的此处找到。

使用 IAM 服务器证书

此 Node.js 代码示例演示：

- 如何执行管理 HTTPS 连接的服务器证书的基本任务。

情景

要在 Amazon 上启用网站或应用程序的 HTTPS 连接，需要 SSL/TLS 服务器证书。要在 Amazon 上将从外部提供程序获得的证书与网站或应用程序结合使用，必须将证书上传到 IAM 或者导入 Amazon Certificate Manager 中。

本示例使用一系列 Node.js 模块在 IAM 中处理服务器证书。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.IAM` 客户端类的以下方法来管理服务器证书：

- listServerCertificates
- getServerCertificate

- updateServerCertificate
- deleteServerCertificate

有关服务器证书的更多信息，请参阅《IAM 用户指南》中的使用服务器证书。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

列出服务器证书

创建文件名为 `iam_listservercerts.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。调用 `listServerCertificates` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.listServerCertificates({}, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_listservercerts.js
```

此示例代码可在 GitHub 上的此处找到。

获取服务器证书

创建文件名为 `iam_getservercert.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含获取证书时所需参数的 JSON 对象，该对象包含您需要的服务器证书的名称。调用 `getServerCertificates` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.getServerCertificate(
  { ServerCertificateName: "CERTIFICATE_NAME" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node iam_getservercert.js
```

此示例代码可在 GitHub 上的此处找到。

更新服务器证书

创建文件名为 `iam_updateservercert.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含更新证书时所需参数的 JSON 对象，其中包括现有服务器证书的名称以及新证书的名称。调用 `updateServerCertificate` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  ServerCertificateName: "CERTIFICATE_NAME",
  NewServerCertificateName: "NEW_CERTIFICATE_NAME",
};

iam.updateServerCertificate(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_updateservercert.js
```

此示例代码可在 GitHub 上的此处找到。

删除服务器证书

创建文件名为 `iam_deleteservercert.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含删除服务器证书时所需参数的 JSON 对象，该对象包含您要删除的证书的名称。调用 `deleteServerCertificates` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteServerCertificate(
  { ServerCertificateName: "CERTIFICATE_NAME" },
  function (err, data) {
```

```
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

要运行示例，请在命令行中键入以下内容。

```
node iam_deleteservercert.js
```

此示例代码可在 GitHub 上的此处找到。

管理 IAM 账户别名

此 Node.js 代码示例演示：

- 如何管理 Amazon 账户 ID 的别名。

情景

如果您希望登录页面的 URL 包含贵公司名称（或其它友好标识符）而不是 Amazon 账户 ID，则可以为 Amazon 账户 ID 创建别名。如果您创建 Amazon 账户别名，您的登录页面 URL 将更改以包含该别名。

本示例使用一系列 Node.js 模块创建和管理 IAM 账户别名。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.IAM` 客户端类的以下方法来管理别名：

- createAccountAlias
- listAccountAliases
- deleteAccountAlias

有关 IAM 账户别名的更多信息，请参阅《IAM 用户指南》中的您的 Amazon 账户 ID 及其别名。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建账户别名

创建文件名为 `iam_createaccountalias.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含创建账户别名时所需参数的 JSON 对象，其中包括您要创建的别名。调用 `createAccountAlias` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.createAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_createaccountalias.js ALIAS
```

此示例代码可在 GitHub 上的此处找到。

列出账户别名

创建文件名为 `iam_listaccountaliases.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含列出账户别名时所需参数的 JSON 对象，其中包括要返回的最大项数。调用 `listAccountAliases` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.listAccountAliases({ MaxItems: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_listaccountaliases.js
```

此示例代码可在 GitHub 上的此处找到。

删除账户别名

创建文件名为 `iam_deleteaccountalias.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 IAM，请创建 `AWS.IAM` 服务对象。创建一个包含删除账户别名时所需参数的 JSON 对象，其中包括您要删除的别名。调用 `deleteAccountAlias` 服务对象的 `AWS.IAM` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
```

```
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node iam_deleteaccountalias.js ALIAS
```

此示例代码可在 GitHub 上的此处找到。

Amazon Kinesis 示例

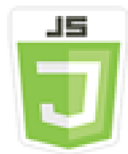
Amazon Kinesis 是一种在 Amazon 上流式处理数据的平台，可提供强大的服务来加载和分析流数据，同时还可让您根据具体需求来构建自定义流数据应用程序。

适用于 Kinesis 的 JavaScript API 通过 `AWS.Kinesis` 客户端类公开。有关使用 Kinesis 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.Kinesis`。

主题

- 使用 Amazon Kinesis 捕获网页滚动进度

使用 Amazon Kinesis 捕获网页滚动进度



此浏览器脚本示例演示：

- 如何使用 Amazon Kinesis 捕获网页中的滚动进度作为流式页面使用指标的示例，以供日后分析。

情景

在本示例中，使用一个简单的 HTML 页面模拟博客页面内容。随着读者在模拟博客帖子中滚动，浏览器脚本使用 SDK for JavaScript 来记录沿着页面向下滚动的距离，并使用 Kinesis 客户端类的 `putRecords` 方法将该数据发送到 Kinesis。然后，Amazon Kinesis Data Streams 捕获的流数据可以通过 Amazon EC2 实例加以处理，并存储在多个数据存储的任意一个中，包括 Amazon DynamoDB 和 Amazon Redshift。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 创建 Kinesis 流。您需要将流的资源 ARN 包含在浏览器脚本中。有关创建 Amazon Kinesis Data Streams 的更多信息，请参阅《Amazon Kinesis Data Streams Developer Guide》中的 Managing Kinesis Streams。
- 创建一个 Amazon Cognito 身份池，并包含为未经身份验证的身份启用的权限。您需要在代码中包含身份池 ID 以获取浏览器脚本的凭证。有关 Amazon Cognito 身份池的更多信息，请参阅《Amazon Cognito 开发人员指南》中的身份池。
- 创建一个 IAM 角色，该角色的策略授予将数据提交到 Kinesis 流的权限。有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

在创建 IAM 角色时，使用以下角色策略。

JSON

```
{
   "Version":"2012-10-17",
```

```
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mobileanalytics:PutEvents",
          "cognito-sync:*"
        ],
        "Resource": [
          "*"
        ]
      },
      {
        "Effect": "Allow",
        "Action": [
          "kinesis:Put*"
        ],
        "Resource": [
          "arn:aws:kinesis:us-east-1:111122223333:stream/stream-name"
        ]
      }
    ]
}
```

博客页面

博客页面的 HTML 主要由包含在 `<div>` 元素中的一系列段落构成。此 `<div>` 的可滚动高度用于帮助计算读者阅读时在内容中滚动的距离。该 HTML 还包含一对 `<script>` 元素。其中一个元素将 SDK for JavaScript 添加到页面，另一个元素则添加浏览器脚本来捕获页面中的滚动进度并报告给 Kinesis。

```
<!DOCTYPE html>
<html>
    <head>
        <title>AWS SDK for JavaScript - Amazon Kinesis Application</title>
    </head>
    <body>
        <div id="BlogContent" style="width: 60%; height: 800px; overflow: auto;margin: auto; text-align: center;">
            <div>
                <p>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum vitae nulla eget nisl bibendum feugiat. Fusce rhoncus felis at ultricies luctus. Vivamus fermentum cursus sem at interdum. Proin vel lobortis nulla. Aenean rutrum
```

```
odio in tellus semper rhoncus. Nam eu felis ac augue dapibus laoreet vel in erat.
Vivamus vitae mollis turpis. Integer sagittis dictum odio. Duis nec sapien diam.
In imperdiet sem nec ante laoreet, vehicula facilisis sem placerat. Duis ut metus
egestas, ullamcorper neque et, accumsan quam. Class aptent taciti sociosqu ad litora
torquent per conubia nostra, per inceptos himenaeos.
                </p>
                <!-- Additional paragraphs in the blog page appear here -->
            </div>
        </div>
        <script src="https://sdk.amazonaws.com/js/aws-sdk-2.283.1.min.js"></script>
        <script src="kinesis-example.js"></script>
    </body>
</html>
```

配置 SDK

通过调用 `CognitoIdentityCredentials` 方法并提供 Amazon Cognito 身份池 ID，获取配置 SDK 时所需的凭证。成功后，在回调函数中创建 Kinesis 服务对象。

下面的代码段演示了此步骤。（有关完整示例，请参阅捕获网页滚动进度代码。）

```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });
```

创建滚动记录

滚动进度是使用包含博客帖子内容的 `scrollHeight` 的 `scrollTop` 和 `<div>` 属性计算的。每个滚动记录在 `scroll` 事件的事件侦听器函数中创建，然后添加到记录数组中，以定期提交到 Kinesis。

下面的代码段演示了此步骤。（有关完整示例，请参阅[捕获网页滚动进度代码](#)。）

```
// Get the ID of the Web page element.
var blogContent = document.getElementById("BlogContent");

// Get Scrollable height
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener("scroll", function (event) {
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling
  TID = setTimeout(function () {
    // calculate percentage
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(
      ((scrollTop + scrollableHeight) / scrollHeight) * 100
    );

    // Create the Amazon Kinesis record
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date(),
      }),
      PartitionKey: "partition-" + AWS.config.credentials.identityId,
    };
    recordData.push(record);
  }, 100);
});
```

将记录提交到 Kinesis

在经过每一秒后，如果数组中有记录，则这些待处理记录会发送到 Kinesis。

下面的代码段演示了此步骤。（有关完整示例，请参阅捕获网页滚动进度代码。）

```
  // upload data to Amazon Kinesis every second if data exists
  setInterval(function () {
    if (!recordData.length) {
      return;
    }
    // upload data to Amazon Kinesis
    kinesis.putRecords(
      {
        Records: recordData,
        StreamName: "NAME_OF_STREAM",
      },
      function (err, data) {
        if (err) {
          console.error(err);
        }
      }
    );
    // clear record data
    recordData = [];
  }, 1000);
});
```

捕获网页滚动进度代码

下面是 Kinesis 捕获网页滚动进度的浏览器脚本代码示例。

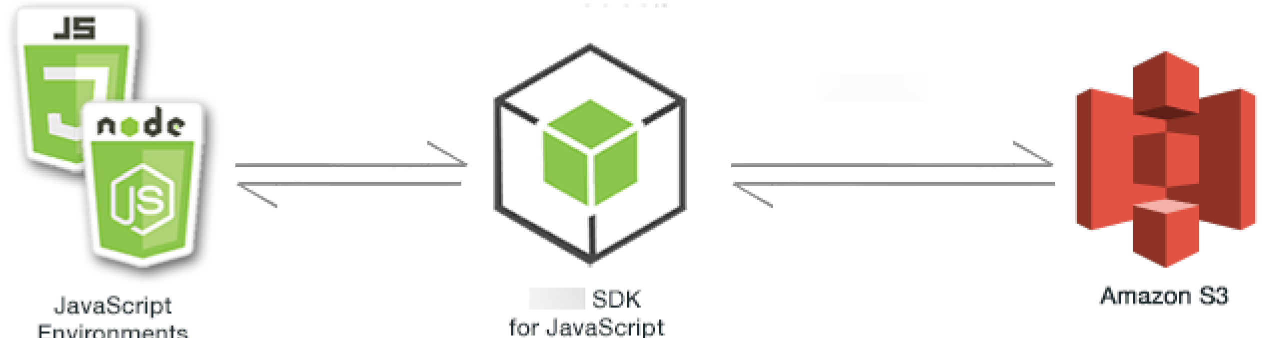
```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
```

```
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });

  // Get the ID of the Web page element.
  var blogContent = document.getElementById("BlogContent");

  // Get Scrollable height
  var scrollableHeight = blogContent.clientHeight;

  var recordData = [];
  var TID = null;
  blogContent.addEventListener("scroll", function (event) {
    clearTimeout(TID);
    // Prevent creating a record while a user is actively scrolling
    TID = setTimeout(function () {
      // calculate percentage
      var scrollableElement = event.target;
      var scrollHeight = scrollableElement.scrollHeight;
      var scrollTop = scrollableElement.scrollTop;

      var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
      var scrollBottomPercentage = Math.round(
        ((scrollTop + scrollableHeight) / scrollHeight) * 100
      );

      // Create the Amazon Kinesis record
      var record = {
        Data: JSON.stringify({
          blog: window.location.href,
          scrollTopPercentage: scrollTopPercentage,
          scrollBottomPercentage: scrollBottomPercentage,
          time: new Date(),
        }),
        PartitionKey: "partition-" + AWS.config.credentials.identityId,
      };
      recordData.push(record);
    }, 100);
  });
```

```
  // upload data to Amazon Kinesis every second if data exists
  setInterval(function () {
    if (!recordData.length) {
      return;
    }
    // upload data to Amazon Kinesis
    kinesis.putRecords(
      {
        Records: recordData,
        StreamName: "NAME_OF_STREAM",
      },
      function (err, data) {
        if (err) {
          console.error(err);
        }
      }
    );
    // clear record data
    recordData = [];
  }, 1000);
});
```

Amazon S3 示例

Amazon Simple Storage Service（Amazon S3）是提供高度可扩展的云存储的 Web 服务。Amazon S3 提供易于使用的对象存储，具有简单的 Web 服务接口，可用于从 Web 上的任何位置存储和检索任意规模的数据。



适用于 Amazon S3 的 JavaScript API 通过 `AWS.S3` 客户端类公开。有关使用 Amazon S3 客户端类的更多信息，请参阅 API 参考中的 [Class: AWS.S3]。

主题

- Amazon S3 浏览器示例
- Amazon S3 Node.js 示例

Amazon S3 浏览器示例

以下主题演示了有关如何在浏览器中使用 适用于 JavaScript 的 Amazon SDK 以与 Amazon S3 桶进行交互的两个示例。

- 第一个示例展示了一个简单场景，在其中，Amazon S3 桶中的现有照片可由任何（未经身份验证的）用户查看。
- 第二个示例展示了一个更复杂的场景，在其中，允许用户对桶中的照片执行操作（如上传、删除等）。

主题

- 从浏览器查看 Amazon S3 存储桶中的照片
- 从浏览器将照片上传到 Amazon S3

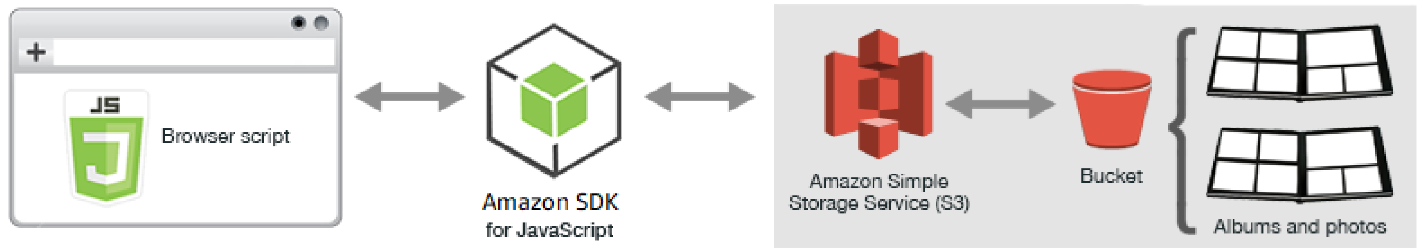
从浏览器查看 Amazon S3 存储桶中的照片

此浏览器脚本代码示例演示：

- 如何在 Amazon Simple Storage Service（Amazon S3）桶中创建相册并允许未验证身份用户查看照片。

情景

本示例通过简单的 HTML 页面提供了一个用于查看相册中的照片的基于浏览器的应用程序。相册位于一个作为照片上传目标的 Amazon S3 桶中。



浏览器脚本使用 SDK for JavaScript 与 Amazon S3 桶交互。该脚本使用 Amazon S3 客户端类的 `listObjects` 方法来允许您查看相册。

先决条件任务

要设置和运行此示例，请先完成以下任务。

> **Note**
>
> 在本示例中，您必须对 Amazon S3 桶和 Amazon Cognito 身份池使用相同的 Amazon 区域。

创建存储桶

在 Amazon S3 控制台中，创建一个可存储相册和照片的 Amazon S3 桶。有关使用控制台创建 S3 桶的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的创建桶。

在创建 S3 存储桶时，请务必执行以下操作：

- 记下存储桶名称，以便在后续先决条件任务“配置角色权限”中使用它。
- 选择一个要在其中创建存储桶的 Amazon 区域。此区域必须与您将在后续先决条件任务“创建身份池”中用于创建 Amazon Cognito 身份池的区域相同。
- 请按照《Amazon Simple Storage Service 用户指南》中的设置访问网站的权限配置存储桶权限。

创建身份池

在 Amazon Cognito 控制台中，创建 Amazon Cognito 身份池，如“浏览器脚本入门”主题中the section called “步骤 1：创建一个 Amazon Cognito 身份池”所述。

当您创建身份池时，请记下身份池名称以及未经身份验证的身份的角色名称。

配置角色权限

要允许查看相册和照片，您必须向刚刚创建的身份池的 IAM 角色添加权限。首先按下面所示创建策略。

1. 打开 IAM 控制台。
2. 在左侧的导航窗格中，选择 Policies (策略)，然后选择 Create policy (创建策略) 按钮。
3. 在 JSON 选项卡上，输入以下 JSON 定义，但将 BUCKET_NAME 替换为存储桶的名称。

   JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:ListBucket"
         ],
         "Resource": [
            "arn:aws:s3:::BUCKET_NAME"
         ]
      }
   ]
}
```

4. 选择 Review policy (查看策略) 按钮，为策略命名并提供描述（如果您需要），然后选择 Create policy (创建策略) 按钮。

   请务必记下该名称，以便稍后可以找到它并将它附加到 IAM 角色。

创建策略后，导航回 IAM 控制台。找到 Amazon Cognito 在上一个先决条件任务“创建身份池”中创建的未经身份验证的身份的 IAM 角色。您将使用刚刚创建的策略来向此身份添加权限。

尽管此任务的工作流与“浏览器脚本入门”主题中the section called “步骤 2：将策略添加到创建的 IAM 角色”的工作流基本相同，但需要注意一些区别：

- 使用您刚刚创建的新策略，而不是 Amazon Polly 的策略。
- 在 Attach Permissions (附加权限) 页面上，要快速查找新策略，请打开 Filter policies (筛选策略) 列表并选择 Customer managed (客户管理的)。

有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

配置 CORS

在浏览器脚本可以访问 Amazon S3 桶之前，您必须先按以下所示设置其 CORS 配置。

Important

在新的 S3 控制台中，CORS 配置必须是 JSON。

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ]
    }
]
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>HEAD</AllowedMethod>
        <AllowedHeader>*</AllowedHeader>
    </CORSRule>
</CORSConfiguration>
```

创建相册和上传照片

由于本示例仅允许用户查看已在存储桶中的照片，因此您需要在存储桶中创建一些相册并将照片上传到相册。

> **Note**
>
> 在本示例中，照片文件的文件名必须以一个下划线（“_”）开头。此字符对于稍后进行筛选很重要。此外，请务必尊重照片所有者的版权。

1. 在 Amazon S3 控制台中，打开您之前创建的桶。
2. 在概述选项卡中，选择创建文件夹按钮以创建文件夹。在本示例中，将文件夹命名为“album1”、“album2”和“album3”。
3. 对于 album1 和 album2，选择相应文件夹，然后将照片上传到其中，如下所示：
   a. 选择上传按钮。
   b. 拖动或选择要使用的照片文件，然后选择下一步。
   c. 在管理公共权限下，选择为此对象授予公共读取访问权限。
   d. 选择上传按钮（左下角）。
4. 将 album3 留空。

定义网页

照片查看应用程序的 HTML 包含一个 `<div>` 元素，在其中，浏览器脚本将创建查看界面。第一个 `<script>` 元素将开发工具包添加到浏览器脚本。第二个 `<script>` 元素添加保存浏览器脚本代码的外部 JavaScript 文件。

在本示例中，该文件名为 `PhotoViewer.js`，并且位于 HTML 文件所在的同一文件夹中。要查找当前的 SDK_VERSION_NUMBER，请参阅 适用于 JavaScript 的 Amazon SDK API Reference Guide 中适用于 SDK for JavaScript 的 API 参考。

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
```

```
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

配置 SDK

通过调用 `CognitoIdentityCredentials` 方法获取配置开发工具包所需的凭证。您需要提供 Amazon Cognito 身份池 ID。然后创建 `AWS.S3` 服务对象。

```
// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
 JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

// Create a new service object
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}
```

本示例中代码的其余部分定义了以下函数，用于收集和展示有关存储桶中的相册和照片的信息。

- `listAlbums`
- `viewAlbum`

列出存储桶中的相册

为了列出存储桶中的所有现有相册，应用程序的 `listAlbums` 函数将调用 `listObjects` 服务对象的 `AWS.S3` 方法。该函数使用 `CommonPrefixes` 属性，以使调用仅返回用作相册的对象（即文件夹）。

函数的剩余部分从 Amazon S3 桶获取相册列表，并生成在网页上显示相册列表所需的 HTML。

```
// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'' +
            albumName +
            "')\">",
          albumName,
          "</button>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml(["<p>Click on an album name to view it.</p>"])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    }
  });
```

```
}
```

查看相册

为显示 Amazon S3 桶中相册的内容，应用程序的 `viewAlbum` 函数获取相册名称并为该相册创建 Amazon S3 键。然后，函数调用 `listObjects` 服务对象的 `AWS.S3` 方法，以获取相册中所有对象（照片）的列表。

函数的剩余部分获取相册中的对象列表并生成在网页上显示照片所需的 HTML。

```
// Show the photos that exist in an album.
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Request instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>The following photos are present.</p>"
      : "<p>There are no photos in this album.</p>";
    var htmlTemplate = [
      "<div>",
```

```
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      "<h2>",
      "End of Album: " + albumName,
      "</h2>",
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    document
      .getElementsByTagName("img")[0]
      .setAttribute("style", "display:none;");
  });
}
```

查看 Amazon S3 桶中的照片：完整代码

本部分包含适用于可查看 Amazon S3 桶中照片的示例的完整 HTML 和 JavaScript 代码。有关详细信息和先决条件，请参阅父部分。

示例的 HTML：

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
```

```
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

此示例代码可在 GitHub 上的此处找到。

示例的浏览器脚本代码：

```
//
// Data constructs and initialization.
//

// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
 JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

// Create a new service object
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}

//
// Functions
//
```

```
// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'' +
            albumName +
            "')\">",
          albumName,
          "</button>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml(["<p>Click on an album name to view it.</p>"])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    }
  });
}

// Show the photos that exist in an album.
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Request instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";
```

```
    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>The following photos are present.</p>"
      : "<p>There are no photos in this album.</p>";
    var htmlTemplate = [
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      "<h2>",
      "End of Album: " + albumName,
      "</h2>",
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
```

```
    document
      .getElementsByTagName("img")[0]
      .setAttribute("style", "display:none;");
  });
}
```

此示例代码可在 GitHub 上的此处找到。

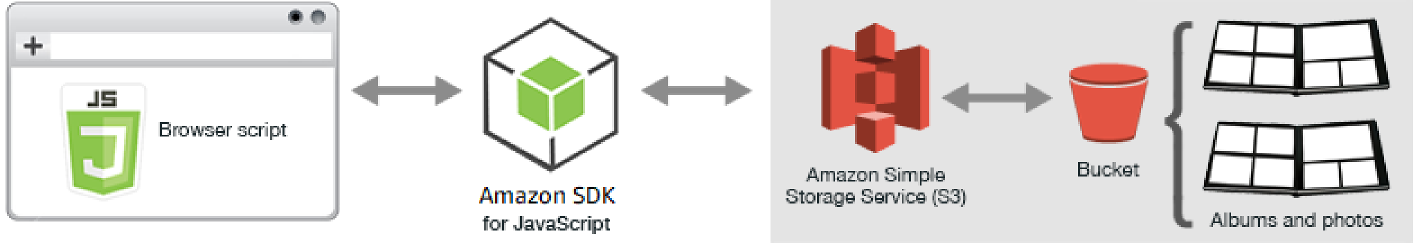
从浏览器将照片上传到 Amazon S3

此浏览器脚本代码示例演示：

- 如何创建允许用户在 Amazon S3 桶中创建相册并将照片上传到相册中的浏览器应用程序。

情景

在本示例中，一个简单的 HTML 页面提供了基于浏览器的应用程序，该应用程序在 Amazon S3 桶中创建相册，您可将照片上传到其中。通过该应用程序，您可以删除所添加的照片和相册。



浏览器脚本使用 SDK for JavaScript 与 Amazon S3 桶交互。使用 Amazon S3 客户端类的以下方法来启用相册应用程序：

- `listObjects`
- `headObject`
- `putObject`
- `upload`
- `deleteObject`

- `deleteObjects`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 在 Amazon S3 控制台中，创建一个 Amazon S3 桶，您将使用它在相册中存储照片。有关在控制台中创建桶的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的创建桶。确保您同时拥有对象的读取和写入权限。有关设置桶权限的更多信息，请参阅设置访问网站的权限。
- 在 Amazon Cognito 控制台中，使用联合身份创建 Amazon Cognito 身份池，其中包含为与 Amazon S3 桶位于相同区域中的未验证身份用户启用的访问权限。您需要在代码中包含身份池 ID 以获取浏览器脚本的凭证。有关 Amazon Cognito 联合身份的更多信息，请参阅《Amazon Cognito 开发人员指南》中的 Amazon Cognito 身份池（联合身份）。
- 在 IAM 控制台中，查找 Amazon Cognito 针对未验证身份用户创建的 IAM 角色。添加以下策略，授予对 Amazon S3 桶的读取和写入权限。有关创建 IAM 角色的更多信息，请参阅《IAM 用户指南》中的创建向 Amazon 服务委派权限的角色。

  为 Amazon Cognito 针对未验证身份用户创建的 IAM 角色使用此角色策略。

  > **Warning**
  >
  > 如果对未经身份验证的用户启用访问权限，您将向所有人授予对该存储桶及存储桶中的所有对象的写入访问权限。在本示例中，此安全状况很有用，可使其专注于示例的主要目标。但是，在许多实际情况下，强烈建议提高安全性（如使用经身份验证的用户和对象所有权）。

  JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:DeleteObject",
            "s3:GetObject",
            "s3:ListBucket",
            "s3:PutObject",
            "s3:PutObjectAcl"
```

```
            ],
            "Resource": [
               "arn:aws:s3:::BUCKET_NAME",
               "arn:aws:s3:::BUCKET_NAME/*"
            ]
         }
      ]
}
```

配置 CORS

在浏览器脚本可以访问 Amazon S3 桶之前，您必须先按以下所示设置其 CORS 配置。

Important

在新的 S3 控制台中，CORS 配置必须是 JSON。

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "ETag"
        ]
    }
]
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>POST</AllowedMethod>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>PUT</AllowedMethod>
        <AllowedMethod>DELETE</AllowedMethod>
        <AllowedMethod>HEAD</AllowedMethod>
        <AllowedHeader>*</AllowedHeader>
        <ExposeHeader>ETag</ExposeHeader>
    </CORSRule>
</CORSConfiguration>
```

网页

照片上传应用程序的 HTML 包含一个 <div> 元素，浏览器脚本在其中创建上传用户界面。第一个 <script> 元素将开发工具包添加到浏览器脚本。第二个 <script> 元素添加外部 JavaScript 文件，该文件保存浏览器脚本代码。

```
<!DOCTYPE html>
<html>
  <head>
     <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./s3_photoExample.js"></script>
    <script>
       function getHtml(template) {
          return template.join('\n');
       }
       listAlbums();
    </script>
  </head>
  <body>
    <h1>My Photo Albums App</h1>
    <div id="app"></div>
  </body>
</html>
```

配置 SDK

通过调用 `CognitoIdentityCredentials` 方法并提供 Amazon Cognito 身份池 ID，获取配置 SDK 时所需的凭证。接下来，创建 `AWS.S3` 服务对象。

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});
```

此示例中几乎所有剩余的代码组织成一系列函数，这些函数收集并提供存储桶中相册的相关信息，上传并显示上传到相册中的照片，以及删除照片和相册。这些函数包括：

- `listAlbums`
- `createAlbum`
- `viewAlbum`
- `addPhoto`
- `deleteAlbum`
- `deletePhoto`

列出存储桶中的相册

应用程序在 Amazon S3 桶中将相册创建为对象，其键以正斜杠字符开头，指示该对象用作文件夹。为列出存储桶中的所有现有相册，应用程序的 `listAlbums` 函数在使用 `listObjects` 时调用 `AWS.S3` 服务对象的 `commonPrefix` 方法，因此调用仅返回用作相册的对象。

函数的剩余部分从 Amazon S3 桶获取相册列表，并生成在网页中显示相册列表所需的 HTML。它还支持删除和打开单独的相册。

```
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml([
            "<p>Click on an album name to view it.</p>",
            "<p>Click on the X to delete the album.</p>",
          ])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create New Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    }
  });
}
```

在存储桶中创建相册

为在 Amazon S3 桶中创建相册，应用程序的 `createAlbum` 函数首先验证为新相册提供的名称，确保它包含合适的字符。然后，函数构成 Amazon S3 对象键，将其传递到 Amazon S3 服务对象的 `headObject` 方法。此方法返回指定键的元数据，因此如果它返回数据，则具有该键的对象已存在。

如果相册尚不存在，则函数调用 `putObject` 服务对象的 `AWS.S3` 方法来创建相册。然后，它调用 `viewAlbum` 函数以显示新的空相册。

```
function createAlbum(albumName) {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  s3.headObject({ Key: albumKey }, function (err, data) {
    if (!err) {
      return alert("Album already exists.");
    }
    if (err.code !== "NotFound") {
      return alert("There was an error creating your album: " + err.message);
    }
    s3.putObject({ Key: albumKey }, function (err, data) {
      if (err) {
        return alert("There was an error creating your album: " + err.message);
      }
      alert("Successfully created album.");
      viewAlbum(albumName);
    });
  });
}
```

查看相册

为显示 Amazon S3 桶中相册的内容，应用程序的 `viewAlbum` 函数获取相册名称并为该相册创建 Amazon S3 键。然后，函数调用 `listObjects` 服务对象的 `AWS.S3` 方法，获取相册中所有对象（照片）的列表。

函数的剩余部分从相册获取对象（照片）列表并生成在网页中显示照片所需的 HTML。它还支持删除单独的照片以及导航回相册列表。

```
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
```

```
    }
    // 'this' references the AWS.Response instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
        "X",
        "</span>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>Click on the X to delete the photo</p>"
      : "<p>You do not have any photos in this album. Please add photos.</p>";
    var htmlTemplate = [
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      '<input id="photoupload" type="file" accept="image/*">',
      '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
      "Add Photo",
      "</button>",
      '<button onclick="listAlbums()">',
```

```
      "Back To Albums",
      "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
  });
}
```

将照片添加到相册

为将照片上传到 Amazon S3 桶中的相册，应用程序的 addPhoto 函数在网页中使用文件选取器元素来标识要上传的文件。然后，它从当前相册名称和文件名，为上传的照片构成一个键。

函数调用 Amazon S3 服务对象的 upload 方法来上传照片。上传照片后，函数将重新显示相册，这样上传的照片会显示。

```
function addPhoto(albumName) {
  var files = document.getElementById("photoupload").files;
  if (!files.length) {
    return alert("Please choose a file to upload first.");
  }
  var file = files[0];
  var fileName = file.name;
  var albumPhotosKey = encodeURIComponent(albumName) + "/";

  var photoKey = albumPhotosKey + fileName;

  // Use S3 ManagedUpload class as it supports multipart uploads
  var upload = new AWS.S3.ManagedUpload({
    params: {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file,
    },
  });

  var promise = upload.promise();

  promise.then(
    function (data) {
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    },
    function (err) {
```

```
      return alert("There was an error uploading your photo: ", err.message);
    }
  );
}
```

删除照片

为从 Amazon S3 桶的相册中删除照片，应用程序的 deletePhoto 函数将调用 Amazon S3 服务对象的 deleteObject 方法。这会删除传递到函数的 photoKey 值指定的照片。

```
function deletePhoto(albumName, photoKey) {
  s3.deleteObject({ Key: photoKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your photo: ", err.message);
    }
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
  });
}
```

删除相册

为删除 Amazon S3 桶中的相册，应用程序的 deleteAlbum 函数将调用 Amazon S3 服务对象的 deleteObjects 方法。

```
function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    s3.deleteObjects(
      {
        Delete: { Objects: objects, Quiet: true },
      },
      function (err, data) {
        if (err) {
          return alert("There was an error deleting your album: ", err.message);
        }
        alert("Successfully deleted album.");
```

```
        listAlbums();
      }
    );
  });
}
```

将照片上传到 Amazon S3：完整代码

本部分包含适用于将照片上传到 Amazon S3 相册的示例的完整 HTML 和 JavaScript 代码。有关详细信息和先决条件，请参阅父部分。

示例的 HTML：

```
<!DOCTYPE html>
<html>
  <head>
     <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./s3_photoExample.js"></script>
    <script>
       function getHtml(template) {
          return template.join('\n');
       }
       listAlbums();
    </script>
  </head>
  <body>
    <h1>My Photo Albums App</h1>
    <div id="app"></div>
  </body>
</html>
```

此示例代码可在 GitHub 上的此处找到。

示例的浏览器脚本代码：

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
```

```
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml([
            "<p>Click on an album name to view it.</p>",
            "<p>Click on the X to delete the album.</p>",
          ])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create New Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
```

```
    }
  });
}

function createAlbum(albumName) {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  s3.headObject({ Key: albumKey }, function (err, data) {
    if (!err) {
      return alert("Album already exists.");
    }
    if (err.code !== "NotFound") {
      return alert("There was an error creating your album: " + err.message);
    }
    s3.putObject({ Key: albumKey }, function (err, data) {
      if (err) {
        return alert("There was an error creating your album: " + err.message);
      }
      alert("Successfully created album.");
      viewAlbum(albumName);
    });
  });
}

function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Response instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
```

```
        "<span>",
        "<div>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
        "X",
        "</span>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>Click on the X to delete the photo</p>"
      : "<p>You do not have any photos in this album. Please add photos.</p>";
    var htmlTemplate = [
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      '<input id="photoupload" type="file" accept="image/*">',
      '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
      "Add Photo",
      "</button>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
  });
}

function addPhoto(albumName) {
  var files = document.getElementById("photoupload").files;
```

```
  if (!files.length) {
    return alert("Please choose a file to upload first.");
  }
  var file = files[0];
  var fileName = file.name;
  var albumPhotosKey = encodeURIComponent(albumName) + "/";

  var photoKey = albumPhotosKey + fileName;

  // Use S3 ManagedUpload class as it supports multipart uploads
  var upload = new AWS.S3.ManagedUpload({
    params: {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file,
    },
  });

  var promise = upload.promise();

  promise.then(
    function (data) {
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    },
    function (err) {
      return alert("There was an error uploading your photo: ", err.message);
    }
  );
}

function deletePhoto(albumName, photoKey) {
  s3.deleteObject({ Key: photoKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your photo: ", err.message);
    }
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
  });
}

function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
```

```
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    s3.deleteObjects(
      {
        Delete: { Objects: objects, Quiet: true },
      },
      function (err, data) {
        if (err) {
          return alert("There was an error deleting your album: ", err.message);
        }
        alert("Successfully deleted album.");
        listAlbums();
      }
    );
  });
}
```

此示例代码可在 GitHub 上的此处找到。

Amazon S3 Node.js 示例

以下主题展示了有关如何通过 适用于 JavaScript 的 Amazon SDK 与使用 Node.js 的 Amazon S3 桶进行交互的示例。

主题

- 创建和使用 Amazon S3 存储桶
- 配置 Amazon S3 存储桶
- 管理 Amazon S3 存储桶访问权限
- 使用 Amazon S3 存储桶策略
- 使用 Amazon S3 存储桶作为静态 Web 主机

创建和使用 Amazon S3 存储桶

此 Node.js 代码示例演示：

- 如何获取和显示账户中 Amazon S3 桶的列表。
- 如何创建 Amazon S3 存储桶。
- 如何将对象上传到指定的存储桶。

情景

本示例使用一系列 Node.js 模块来获取现有 Amazon S3 桶的列表、创建桶并将文件上传到指定桶。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon S3 客户端类的以下方法从 Amazon S3 桶获取信息和将文件上传到该桶：

- `listBuckets`
- `createBucket`
- `listObjects`
- `upload`
- `deleteBucket`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

显示 Amazon S3 桶的列表

创建文件名为 `s3_listbuckets.js` 的 Node.js 模块。确保按前面所示配置开发工具包。要访问 Amazon Simple Storage Service，请创建一个 `AWS.S3` 服务对象。调用 Amazon S3 服务对象的 `listBuckets` 方法来检索桶列表。回调函数的 `data` 参数具有 `Buckets` 属性，包含用于表示存储桶的映射数组。通过登录到控制台显示存储桶列表。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Call S3 to list the buckets
s3.listBuckets(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Buckets);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_listbuckets.js
```

此示例代码可在 [GitHub 上的此处](#)找到。

创建 Amazon S3 存储桶

创建文件名为 `s3_createbucket.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。模块将获取单个命令行参数来指定新存储桶的名称。

添加变量来保存用于调用 Amazon S3 服务对象的 `createBucket` 方法的参数，其中包括新创建的桶的名称。在 Amazon S3 成功创建新桶后，回调函数将新桶的位置记录到控制台中。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling createBucket
var bucketParams = {
  Bucket: process.argv[2],
};

// call S3 to create the bucket
s3.createBucket(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Location);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_createbucket.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

将文件上传到 Amazon S3 桶

创建文件名为 `s3_upload.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。此模块将获取两个命令行参数，第一个用于指定目标存储桶，第二个用于指定要上传的文件。

使用调用 Amazon S3 服务对象的 `upload` 方法所需的参数创建变量。在 `Bucket` 参数中提供目标存储桶的名称。`Key` 参数设置为所选文件的名称，您可使用 Node.js `path` 模块来获取该名称。`Body` 参数设置为文件的内容，您可从 Node.js `createReadStream` 模块使用 `fs` 来获取。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create S3 service object
var s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// call S3 to retrieve upload file to specified bucket
var uploadParams = { Bucket: process.argv[2], Key: "", Body: "" };
var file = process.argv[3];

// Configure the file stream and obtain the upload parameters
var fs = require("fs");
var fileStream = fs.createReadStream(file);
fileStream.on("error", function (err) {
  console.log("File Error", err);
});
uploadParams.Body = fileStream;
var path = require("path");
uploadParams.Key = path.basename(file);

// call S3 to retrieve upload file to specified bucket
s3.upload(uploadParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  }
  if (data) {
    console.log("Upload Success", data.Location);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_upload.js BUCKET_NAME FILE_NAME
```

此示例代码可在 GitHub 上的此处找到。

列出 Amazon S3 桶中的对象

创建文件名为 `s3_listobjects.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。

添加变量来保存用于调用 Amazon S3 服务对象的 `listObjects` 方法的参数，其中包括要读取的桶的名称。回调函数记录对象（文件）列表或失败消息。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling listObjects
var bucketParams = {
  Bucket: "BUCKET_NAME",
};

// Call S3 to obtain a list of the objects in the bucket
s3.listObjects(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_listobjects.js
```

此示例代码可在 GitHub 上的此处找到。

删除 Amazon S3 桶

创建文件名为 `s3_deletebucket.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。

添加变量来保存用于调用 Amazon S3 服务对象的 `createBucket` 方法的参数，其中包括要删除的桶的名称。存储桶必须为空才能将其删除。回调函数记录成功或失败消息。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create params for S3.deleteBucket
```

```
var bucketParams = {
  Bucket: "BUCKET_NAME",
};

// Call S3 to delete the bucket
s3.deleteBucket(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_deletebucket.js
```

此示例代码可在 GitHub 上的此处找到。

配置 Amazon S3 存储桶

此 Node.js 代码示例演示：

- 如何为存储桶配置跨源资源共享 (CORS) 权限。

情景

在本示例中，使用一系列 Node.js 模块来列出您的 Amazon S3 存储桶以及配置 CORS 和存储桶日志记录。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon S3 客户端类的以下方法来配置选定的 Amazon S3 桶：

- `getBucketCors`
- `putBucketCors`

有关将 CORS 配置与 Amazon S3 桶配合使用的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的跨源资源共享（CORS）。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

检索存储桶 CORS 配置

创建文件名为 `s3_getcors.js` 的 Node.js 模块。模块将获取单个命令行参数来指定需要其 CORS 配置的存储桶。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。

在调用 `getBucketCors` 方法时，您需要传递的唯一参数是所选存储桶的名称。如果桶当前具有 CORS 配置，该配置由 Amazon S3 作为传递到回调函数的 `data` 参数的 `CORSRules` 属性返回。

如果所选存储桶没有 CORS 配置，该信息将在 `error` 参数中返回到回调函数。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Set the parameters for S3.getBucketCors
var bucketParams = { Bucket: process.argv[2] };

// call S3 to retrieve CORS configuration for selected bucket
s3.getBucketCors(bucketParams, function (err, data) {
  if (err) {
```

```
    console.log("Error", err);
  } else if (data) {
    console.log("Success", JSON.stringify(data.CORSRules));
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_getcors.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

设置存储桶 CORS 配置

创建文件名为 s3_setcors.js 的 Node.js 模块。该模块获取多个命令行参数，第一个参数指定要设置其 CORS 配置的存储桶。其他参数枚举您希望允许对存储桶使用的 HTTP 方法（POST、GET、PUT、PATCH、DELETE、POST）。按前面所示配置 SDK。

创建 AWS.S3 服务对象。接下来，根据 putBucketCors 服务对象的 AWS.S3 方法的要求，创建一个 JSON 对象来保存 CORS 配置的值。为 "Authorization" 值指定 AllowedHeaders，为 "*" 值指定 AllowedOrigins。最初，将 AllowedMethods 的值设置为空数组。

指定允许的方法作为 Node.js 模块的命令行参数，添加与参数之一匹配的各个方法。将生成的 CORS 配置添加到 CORSRules 参数中包含的配置的数组。在 Bucket 参数中指定您要为 CORS 配置的存储桶。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create initial parameters JSON for putBucketCors
var thisConfig = {
  AllowedHeaders: ["Authorization"],
  AllowedMethods: [],
  AllowedOrigins: ["*"],
  ExposeHeaders: [],
  MaxAgeSeconds: 3000,
};
```

```
// Assemble the list of allowed methods based on command line parameters
var allowedMethods = [];
process.argv.forEach(function (val, index, array) {
  if (val.toUpperCase() === "POST") {
    allowedMethods.push("POST");
  }
  if (val.toUpperCase() === "GET") {
    allowedMethods.push("GET");
  }
  if (val.toUpperCase() === "PUT") {
    allowedMethods.push("PUT");
  }
  if (val.toUpperCase() === "PATCH") {
    allowedMethods.push("PATCH");
  }
  if (val.toUpperCase() === "DELETE") {
    allowedMethods.push("DELETE");
  }
  if (val.toUpperCase() === "HEAD") {
    allowedMethods.push("HEAD");
  }
});

// Copy the array of allowed methods into the config object
thisConfig.AllowedMethods = allowedMethods;
// Create array of configs then add the config object to it
var corsRules = new Array(thisConfig);

// Create CORS params
var corsParams = {
  Bucket: process.argv[2],
  CORSConfiguration: { CORSRules: corsRules },
};

// set the new CORS configuration on the selected bucket
s3.putBucketCors(corsParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    // update the displayed CORS config for the selected bucket
    console.log("Success", data);
  }
```

```
});
```

要运行此示例，请在命令行中键入以下内容，包括一个或多个所示的 HTTP 方法。

```
node s3_setcors.js BUCKET_NAME get put
```

此示例代码可在 GitHub 上的此处找到。

管理 Amazon S3 存储桶访问权限

此 Node.js 代码示例演示：

- 如何检索或设置 Amazon S3 存储桶的访问控制列表。

情景

在本示例中，使用 Node.js 模块来显示选定存储桶的存储桶访问控制列表 (ACL)，并为选定存储桶应用对 ACL 的更改。该 Node.js 模块使用 SDK for JavaScript，通过 Amazon S3 客户端类的以下方法来管理 Amazon S3 桶访问权限：

- `getBucketAcl`
- `putBucketAcl`

有关 Amazon S3 桶访问控制列表的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的使用 ACL 管理访问权限。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

检索当前存储桶的访问控制列表

创建文件名为 `s3_getbucketacl.js` 的 Node.js 模块。该模块将获取单个命令行参数，指定需要其 ACL 配置的存储桶。确保按前面所示配置开发工具包。

创建 `AWS.S3` 服务对象。在调用 `getBucketAcl` 方法时，您需要传递的唯一参数是所选存储桶的名称。当前访问控制列表配置由 Amazon S3 在传递到回调函数的 `data` 参数中返回。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketAcl(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Grants);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_getbucketacl.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

使用 Amazon S3 存储桶策略

此 Node.js 代码示例演示：

- 如何检索 Amazon S3 桶的桶策略。
- 如何添加或更新 Amazon S3 桶的桶策略。
- 如何删除 Amazon S3 桶的桶策略。

情景

本示例使用一系列 Node.js 模块来检索、设置或删除 Amazon S3 桶上的桶策略。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon S3 客户端类的以下方法来配置选定的 Amazon S3 桶的策略：

- `getBucketPolicy`
- `putBucketPolicy`
- `deleteBucketPolicy`

有关 Amazon S3 桶的桶策略的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的使用桶策略和用户策略。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
```

```
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

检索当前存储桶策略

创建文件名为 `s3_getbucketpolicy.js` 的 Node.js 模块。该模块将获取单个命令行参数，指定需要其策略的存储桶。确保按前面所示配置开发工具包。

创建 `AWS.S3` 服务对象。在调用 `getBucketPolicy` 方法时，您需要传递的唯一参数是所选存储桶的名称。如果桶当前具有策略，该策略在由 Amazon S3 传递到回调函数的 `data` 参数中返回。

如果所选存储桶没有策略，该信息将在 `error` 参数中返回给回调函数。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketPolicy(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Policy);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_getbucketpolicy.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

设置简单存储桶策略

创建文件名为 `s3_setbucketpolicy.js` 的 Node.js 模块。该模块将获取单个命令行参数，指定需要应用其策略的存储桶。按前面所示配置 SDK。

创建 AWS.S3 服务对象。存储桶策略以 JSON 形式指定。首先，创建一个 JSON 对象，该对象包含用于指定策略的所有值，但标识存储桶的 Resource 值除外。

格式化策略所需的 Resource 字符串，在其中包含所选存储桶的名称。将该字符串插入到 JSON 对象中。准备 putBucketPolicy 方法的参数，包括存储桶的名称以及转换为字符串值的 JSON 策略。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var readOnlyAnonUserPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Sid: "AddPerm",
      Effect: "Allow",
      Principal: "*",
      Action: ["s3:GetObject"],
      Resource: [""],
    },
  ],
};

// create selected bucket resource string for bucket policy
var bucketResource = "arn:aws:s3:::" + process.argv[2] + "/*";
readOnlyAnonUserPolicy.Statement[0].Resource[0] = bucketResource;

// convert policy JSON into string and assign into params
var bucketPolicyParams = {
  Bucket: process.argv[2],
  Policy: JSON.stringify(readOnlyAnonUserPolicy),
};

// set the new policy on the selected bucket
s3.putBucketPolicy(bucketPolicyParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
```

```
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_setbucketpolicy.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

删除存储桶策略

创建文件名为 s3_deletebucketpolicy.js 的 Node.js 模块。该模块将获取单个命令行参数，指定需要删除其策略的存储桶。按前面所示配置 SDK。

创建 AWS.S3 服务对象。在调用 deleteBucketPolicy 方法时，您需要传递的唯一参数是所选存储桶的名称。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to delete policy for selected bucket
s3.deleteBucketPolicy(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_deletebucketpolicy.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

使用 Amazon S3 存储桶作为静态 Web 主机

此 Node.js 代码示例演示：

- 如何设置 Amazon S3 桶作为静态 Web 主机。

情景

在此示例中，使用一系列 Node.js 模块将您的任意存储桶配置作为静态 Web 主机。这些 Node.js 模块使用 SDK for JavaScript，通过 Amazon S3 客户端类的以下方法来配置选定的 Amazon S3 桶：

- `getBucketWebsite`
- `putBucketWebsite`
- `deleteBucketWebsite`

有关使用 Amazon S3 桶作为静态 Web 主机的更多信息，请参阅《Amazon Simple Storage Service 用户指南》中的在 Amazon S3 上托管静态网站。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
```

```
AWS.config.update({region: 'us-west-2'});
```

检索当前存储桶网站配置

创建文件名为 `s3_getbucketwebsite.js` 的 Node.js 模块。该模块将获取单个命令行参数，指定需要其网站配置的存储桶。按前面所示配置 SDK。

创建 `AWS.S3` 服务对象。创建一个函数，检索在存储桶列表中选择的存储桶的当前存储桶网站配置。在调用 `getBucketWebsite` 方法时，您需要传递的唯一参数是所选存储桶的名称。如果桶当前具有网站配置，该配置在由 Amazon S3 传递到回调函数的 `data` 参数中返回。

如果所选存储桶没有网站配置，该信息将在 `err` 参数中返回给回调函数。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };

// call S3 to retrieve the website configuration for selected bucket
s3.getBucketWebsite(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_getbucketwebsite.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

设置存储桶的网站配置

创建文件名为 `s3_setbucketwebsite.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。

创建应用存储桶网站配置的函数。该配置允许将所选存储桶用作静态 Web 主机。网站配置以 JSON 形式指定。首先，创建包含用于指定网站配置的所有值的 JSON 对象，但标识错误文档的 `Key` 值以及标识索引文档的 `Suffix` 值除外。

将文本输入元素的值插入到 JSON 对象中。准备 `putBucketWebsite` 方法的参数，包括存储桶的名称以及 JSON 网站配置。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create JSON for putBucketWebsite parameters
var staticHostParams = {
  Bucket: "",
  WebsiteConfiguration: {
    ErrorDocument: {
      Key: "",
    },
    IndexDocument: {
      Suffix: "",
    },
  },
};

// Insert specified bucket name and index and error documents into params JSON
// from command line arguments
staticHostParams.Bucket = process.argv[2];
staticHostParams.WebsiteConfiguration.IndexDocument.Suffix = process.argv[3];
staticHostParams.WebsiteConfiguration.ErrorDocument.Key = process.argv[4];

// set the new website configuration on the selected bucket
s3.putBucketWebsite(staticHostParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    // update the displayed website configuration for the selected bucket
    console.log("Success", data);
  }
```

```
});
```

要运行示例，请在命令行中键入以下内容。

```
node s3_setbucketwebsite.js BUCKET_NAME INDEX_PAGE ERROR_PAGE
```

此示例代码可在 GitHub 上的此处找到。

删除存储桶网站配置

创建文件名为 `s3_deletebucketwebsite.js` 的 Node.js 模块。确保按前面所示配置开发工具包。创建 `AWS.S3` 服务对象。

创建删除所选存储桶的网站配置的函数。在调用 `deleteBucketWebsite` 方法时，您需要传递的唯一参数是所选存储桶的名称。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };

// call S3 to delete website configuration for selected bucket
s3.deleteBucketWebsite(bucketParams, function (error, data) {
  if (error) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

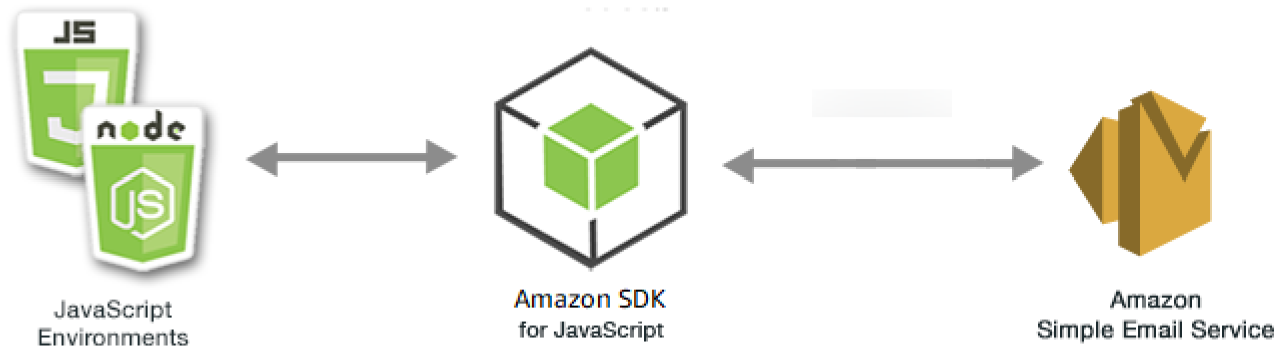
要运行示例，请在命令行中键入以下内容。

```
node s3_deletebucketwebsite.js BUCKET_NAME
```

此示例代码可在 GitHub 上的此处找到。

Amazon Simple Email Service 示例

Amazon Simple Email Service（Amazon SES）是一项基于云的电子邮件发送服务，旨在帮助数字营销人员和应用程序开发人员发送营销、通知和事务电子邮件。对于使用电子邮件联系客户的所有规模的企业来说，它是一种可靠且经济实用的服务。



适用于 Amazon SES 的 JavaScript API 通过 `AWS.SES` 客户端类公开。有关使用 Amazon SES 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.SES`。

主题

- 管理 Amazon SES 身份
- 在 Amazon SES 中使用电子邮件模板
- 使用 Amazon SES 发送电子邮件
- 为 Amazon SES 中的电子邮件接收使用 IP 地址筛选条件
- 在 Amazon SES 中使用接收规则

管理 Amazon SES 身份

此 Node.js 代码示例演示：

- 如何验证用于 Amazon SES 的电子邮件地址和域。
- 如何将 IAM policy 分配到您的 Amazon SES 身份。
- 如何列出您 Amazon 账户中的所有 Amazon SES 身份。

- 如何删除用于 Amazon SES 的身份。

Amazon SES 身份 是 Amazon SES 用来发送电子邮件的电子邮件地址或域。Amazon SES 要求您验证电子邮件身份，以确认您拥有该身份，并防止他人使用。

有关如何在 Amazon SES 中验证电子邮件地址和域的详细信息，请参阅《Amazon Simple Email Service 开发人员指南》中的在 Amazon SES 中验证电子邮件地址和域。有关 Amazon SES 中发送授权的信息，请参阅 Amazon SES 发送授权概览。

情景

在本示例中，您使用一系列 Node.js 模块验证和管理 Amazon SES 身份。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SES` 客户端类的以下方法来验证电子邮件地址和域：

- `listIdentities`
- `deleteIdentity`
- `verifyEmailIdentity`
- `verifyDomainIdentity`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');

// Set the Region
AWS.config.update({region: 'us-west-2'});
```

列出身份

在本示例中，使用 Node.js 模块列出用于 Amazon SES 的电子邮件地址和域。创建文件名为 `ses_listidentities.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，为 `AWS.SES` 客户端类的 `listIdentities` 方法传递 `IdentityType` 及其他参数。要调用 `listIdentities` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数对象。

然后处理 promise 回调中的 `response`。由 promise 返回的 `data` 包含 `IdentityType` 参数所指定的域身份数组。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create listIdentities params
var params = {
  IdentityType: "Domain",
  MaxItems: 10,
};

// Create the promise and SES service object
var listIDsPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listIdentities(params)
  .promise();

// Handle promise's fulfilled/rejected states
listIDsPromise
  .then(function (data) {
    console.log(data.Identities);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node ses_listidentities.js
```

此示例代码可在 GitHub 上的此处找到。

验证电子邮件地址身份

本示例使用 Node.js 模块验证用于 Amazon SES 的电子邮件发送方。创建文件名为 `ses_verifyemailidentity.js` 的 Node.js 模块。按前面所示配置 SDK。要访问 Amazon SES，请创建 `AWS.SES` 服务对象。

创建对象，为 `AWS.SES` 客户端类的 `verifyEmailIdentity` 方法传递 `EmailAddress` 参数。要调用 `verifyEmailIdentity` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SES service object
var verifyEmailPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .verifyEmailIdentity({ EmailAddress: "ADDRESS@DOMAIN.EXT" })
  .promise();

// Handle promise's fulfilled/rejected states
verifyEmailPromise
  .then(function (data) {
    console.log("Email verification initiated");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。域会添加到 Amazon SES 等待验证。

```
node ses_verifyemailidentity.js
```

此示例代码可在 GitHub 上的此处找到。

验证域身份

本示例使用 Node.js 模块验证用于 Amazon SES 的电子邮件域。创建文件名为 `ses_verifydomainidentity.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，为 AWS.SES 客户端类的 verifyDomainIdentity 方法传递 Domain 参数。要调用 verifyDomainIdentity 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数对象。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var verifyDomainPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .verifyDomainIdentity({ Domain: "DOMAIN_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
verifyDomainPromise
  .then(function (data) {
    console.log("Verification Token: " + data.VerificationToken);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。域会添加到 Amazon SES 等待验证。

```
node ses_verifydomainidentity.js
```

此示例代码可在 GitHub 上的此处找到。

删除身份

本示例使用 Node.js 模块删除用于 Amazon SES 的电子邮件地址或域。创建文件名为 ses_deleteidentity.js 的 Node.js 模块。按前面所示配置 SDK。

创建对象，为 AWS.SES 客户端类的 deleteIdentity 方法传递 Identity 参数。要调用 deleteIdentity 方法，请创建一个 request 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var deletePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteIdentity({ Identity: "DOMAIN_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
deletePromise
  .then(function (data) {
    console.log("Identity Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node ses_deleteidentity.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SES 中使用电子邮件模板

此 Node.js 代码示例演示：

- 获取所有电子邮件模板的列表。
- 检索和更新电子邮件模板。
- 创建和删除电子邮件模板。

通过 Amazon SES，您可以使用电子邮件模板发送个性化的电子邮件。有关如何在 Amazon Simple Email Service 中创建和使用电子邮件模板的详细信息，请参阅《Amazon Simple Email Service 开发人员指南》中的通过 Amazon SES API 发送个性化电子邮件。

情景

在本示例中，您使用一系列 Node.js 模块来处理电子邮件模板。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SES` 客户端类的以下方法来创建和使用电子邮件模板：

- `listTemplates`
- `createTemplate`
- `getTemplate`
- `deleteTemplate`
- `updateTemplate`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关创建凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

列出电子邮件模板

本示例使用 Node.js 模块创建用于 Amazon SES 的电子邮件模板。创建文件名为 `ses_listtemplates.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，为 `AWS.SES` 客户端类的 `listTemplates` 方法传递参数。要调用 `listTemplates` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listTemplates({ MaxItems: ITEMS_COUNT })
  .promise();
```

```
// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 会返回模板列表。

```
node ses_listtemplates.js
```

此示例代码可在 GitHub 上的此处找到。

获取电子邮件模板

本示例使用 Node.js 模块获取用于 Amazon SES 的电子邮件模板。创建文件名为 `ses_gettemplate.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，为 `AWS.SES` 客户端类的 `getTemplate` 方法传递 `TemplateName` 参数。要调用 `getTemplate` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create the promise and Amazon Simple Email Service (Amazon SES) service object.
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .getTemplate({ TemplateName: "TEMPLATE_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data.Template.SubjectPart);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 会返回模板详细信息。

```
node ses_gettemplate.js
```

此示例代码可在 GitHub 上的此处找到。

创建电子邮件模板

本示例使用 Node.js 模块创建用于 Amazon SES 的电子邮件模板。创建文件名为 `ses_createtemplate.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象来为 `AWS.SES` 客户端类的 `createTemplate` 方法传递参数，其中包括 `TemplateName`、`HtmlPart`、`SubjectPart` 和 `TextPart`。要调用 `createTemplate` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createTemplate params
var params = {
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
```

```
  });
```

要运行示例，请在命令行中键入以下内容。将在 Amazon SES 中添加模板。

```
node ses_createtemplate.js
```

此示例代码可在 GitHub 上的此处找到。

更新电子邮件模板

本示例使用 Node.js 模块创建用于 Amazon SES 的电子邮件模板。创建文件名为 `ses_updatetemplate.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象来传递您在模板中要更新的 `Template` 参数值，并将必需的 `TemplateName` 参数传递到 `AWS.SES` 客户端类的 `updateTemplate` 方法。要调用 `updateTemplate` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create updateTemplate parameters
var params = {
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .updateTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log("Template Updated");
  })
```

```
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 会返回模板详细信息。

```
node ses_updatetemplate.js
```

此示例代码可在 GitHub 上的此处找到。

删除电子邮件模板

本示例使用 Node.js 模块创建用于 Amazon SES 的电子邮件模板。创建文件名为 `ses_deletetemplate.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，将必需的 `TemplateName` 参数传递到 `AWS.SES` 客户端类的 `deleteTemplate` 方法。要调用 `deleteTemplate` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteTemplate({ TemplateName: "TEMPLATE_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log("Template Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 会返回模板详细信息。

```
node ses_deletetemplate.js
```

此示例代码可在 GitHub 上的此处找到。

使用 Amazon SES 发送电子邮件

此 Node.js 代码示例演示：

- 发送文本或 HTML 电子邮件。
- 根据电子邮件模板发送电子邮件。
- 根据电子邮件模板批量发送电子邮件。

Amazon SES API 为您提供了两种不同的方法来发送电子邮件，具体取决于您对电子邮件内容的控制程度：格式化和原始。有关详细信息，请参阅使用 Amazon SES API 发送格式化的电子邮件和使用 Amazon SES API 发送原始电子邮件。

情景

在本示例中，您使用一系列 Node.js 模块以多种方式发送电子邮件。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SES` 客户端类的以下方法来创建和使用电子邮件模板：

- `sendEmail`
- `sendTemplatedEmail`
- `sendBulkTemplatedEmail`

先决条件任务

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

电子邮件发送要求

Amazon SES 编写电子邮件并立即将其加入队列等待发送。要使用 `SES.sendEmail` 方法发送电子邮件，您的邮件必须满足以下要求：

- 您必须从已验证的电子邮件地址或域发送邮件。如果您尝试使用未验证的地址或域发送电子邮件，则操作会导致 "Email address not verified" 错误。
- 如果您的账户仍在 Amazon SES 沙盒中，则只能发送到经验证的地址或域，或者与 Amazon SES 邮箱模拟器关联的电子邮件地址。有关更多信息，请参阅《Amazon Simple Email Service 开发人员指南》中的验证电子邮件地址和域。
- 邮件（包括附件）的总大小必须小于 10 MB。
- 邮件必须包含至少一个收件人电子邮件地址。收件人地址可以是“收件人：”地址、“抄送：”地址或“密件抄送：”地址。如果某个收件人的电子邮件地址无效（即，未使用格式 UserName@[SubDomain.]Domain.TopLevelDomain），则将拒绝整个邮件，即使邮件包含的其他收件人有效。
- 邮件在“收件人：”、“抄送：”和“密件抄送：”字段中包含的收件人不能超过 50 个。如果您需要将电子邮件发送给更多的受众，可以将收件人列表划分为不超过 50 个人的组，然后多次调用 sendEmail 方法来发送邮件到各个组。

发送电子邮件

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 ses_sendemail.js 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象以将定义要发送的电子邮件的参数值传递到 AWS.SES 客户端类的 sendEmail 方法，这些参数值包括发件人和收件人地址、主题、纯文本和 HTML 格式的电子邮件正文。要调用 sendEmail 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more items */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
```

```
      /* more items */
    ],
  },
  Message: {
    /* required */
    Body: {
      /* required */
      Html: {
        Charset: "UTF-8",
        Data: "HTML_FORMAT_BODY",
      },
      Text: {
        Charset: "UTF-8",
        Data: "TEXT_FORMAT_BODY",
      },
    },
    Subject: {
      Charset: "UTF-8",
      Data: "Test email",
    },
  },
  Source: "SENDER_EMAIL_ADDRESS" /* required */,
  ReplyToAddresses: [
    "EMAIL_ADDRESS",
    /* more items */
  ],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。电子邮件会排队，等待由 Amazon SES 发送。

```
node ses_sendemail.js
```

此示例代码可在 GitHub 上的此处找到。

使用模板发送电子邮件

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_sendtemplatedemail.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象以将定义要发送的电子邮件的参数值传递到 `AWS.SES` 客户端类的 `sendTemplatedEmail` 方法，这些参数值包括发件人和收件人地址、主题、纯文本和 HTML 格式的电子邮件正文。要调用 `sendTemplatedEmail` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendTemplatedEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more CC email addresses */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
      /* more To email addresses */
    ],
  },
  Source: "EMAIL_ADDRESS" /* required */,
  Template: "TEMPLATE_NAME" /* required */,
  TemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }' /* required */,
  ReplyToAddresses: ["EMAIL_ADDRESS"],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendTemplatedEmail(params)
  .promise();
```

```
// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。电子邮件会排队，等待由 Amazon SES 发送。

```
node ses_sendtemplatedemail.js
```

此示例代码可在 GitHub 上的此处找到。

使用模板批量发送电子邮件

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_sendbulktemplatedemail.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象以将定义要发送的电子邮件的参数值传递到 `AWS.SES` 客户端类的 `sendBulkTemplatedEmail` 方法，这些参数值包括发件人和收件人地址、主题、纯文本和 HTML 格式的电子邮件正文。要调用 `sendBulkTemplatedEmail` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendBulkTemplatedEmail params
var params = {
  Destinations: [
    /* required */
    {
      Destination: {
        /* required */
        CcAddresses: [
          "EMAIL_ADDRESS",
          /* more items */
        ],
```

```
          ToAddresses: [
            "EMAIL_ADDRESS",
            "EMAIL_ADDRESS",
            /* more items */
          ],
        },
        ReplacementTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
      },
  ],
  Source: "EMAIL_ADDRESS" /* required */,
  Template: "TEMPLATE_NAME" /* required */,
  DefaultTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
  ReplyToAddresses: ["EMAIL_ADDRESS"],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendBulkTemplatedEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。电子邮件会排队，等待由 Amazon SES 发送。

```
node ses_sendbulktemplatedemail.js
```

此示例代码可在 GitHub 上的此处找到。

为 Amazon SES 中的电子邮件接收使用 IP 地址筛选条件

此 Node.js 代码示例演示：

- 创建 IP 地址筛选条件以接受或拒绝来自某个 IP 地址或 IP 地址范围的邮件。
- 列出您当前的 IP 地址筛选条件。
- 删除 IP 地址筛选条件。

在 Amazon SES 中，筛选条件 是一个数据结构，包括名称、IP 地址范围以及允许还是阻止来自该范围的邮件。对于您希望阻止或允许的 IP 地址，以无类域间路由 (CIDR) 表示法指定单个 IP 地址或 IP 地址范围。有关 Amazon SES 如何接收电子邮件的详细信息，请参阅《Amazon Simple Email Service 开发人员指南》中的 Amazon SES 电子邮件接收概念。

情景

在本示例中，使用了一系列 Node.js 模块以多种方式发送电子邮件。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SES` 客户端类的以下方法来创建和使用电子邮件模板：

- `createReceiptFilter`
- `listReceiptFilters`
- `deleteReceiptFilter`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

配置 SDK

通过创建全局配置对象然后为代码设置区域，来配置 SDK for JavaScript。在此示例中，区域设置为 `us-west-2`。

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

创建 IP 地址筛选条件

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_createreceiptfilter.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个传递参数值的对象，这些参数值定义 IP 筛选条件，包括筛选条件名称，要筛选的 IP 地址或地址范围，以及允许还是阻止来自所筛选地址的电子邮件流量。要调用 `createReceiptFilter` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createReceiptFilter params
var params = {
  Filter: {
    IpFilter: {
      Cidr: "IP_ADDRESS_OR_RANGE",
      Policy: "Allow" | "Block",
    },
    Name: "NAME",
  },
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptFilter(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。将在 Amazon SES 中创建筛选条件。

```
node ses_createreceiptfilter.js
```

此示例代码可在 GitHub 上的此处找到。

列出 IP 地址筛选条件

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_listreceiptfilters.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个空参数对象。要调用 `listReceiptFilters` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listReceiptFilters({})
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data.Filters);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 会返回筛选条件列表。

```
node ses_listreceiptfilters.js
```

此示例代码可在 GitHub 上的此处找到。

删除 IP 地址筛选器

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_deletereceiptfilter.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，传递要删除的 IP 筛选条件的名称。要调用 deleteReceiptFilter 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptFilter({ FilterName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log("IP Filter deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。将从 Amazon SES 中删除筛选条件。

```
node ses_deletereceiptfilter.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SES 中使用接收规则

此 Node.js 代码示例演示：

- 创建和删除接收规则。
- 将接收规则组织为接收规则集。

Amazon SES 中的接收规则指定从您拥有的电子邮件地址或域接收电子邮件后，将执行哪些操作。接收规则 包含一个条件和一个有序操作列表。如果传入电子邮件的收件人与接收规则条件中指定的收件人相匹配，则 Amazon SES 执行接收规则指定的操作。

要使用 Amazon SES 作为电子邮件接收方，您必须至少具有一个有效接收规则集。接收规则集是接收规则的有序集合，用于指定 Amazon SES 如何处理从您的已验证域接收的邮件。有关更多信息，请参阅《Amazon Simple Email Service 开发者指南》中的为 Amazon SES 电子邮件接收创建接收规则和为 Amazon SES 电子邮件接收创建接收规则集。

情景

在本示例中，使用了一系列 Node.js 模块以多种方式发送电子邮件。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SES` 客户端类的以下方法来创建和使用电子邮件模板：

- `createReceiptRule`
- `deleteReceiptRule`
- `createReceiptRuleSet`
- `deleteReceiptRuleSet`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建 Amazon S3 接收规则

Amazon SES 的每个接收规则都包含一组有序操作。此示例创建具有 Amazon S3 操作的接收规则，该操作将邮件传送到 Amazon S3 存储桶。有关接收规则操作的详细信息，请参阅《Amazon Simple Email Service 开发人员指南》中的操作选项。

要使 Amazon SES 将电子邮件写入 Amazon S3 存储桶，请创建向 Amazon SES 提供 `PutObject` 权限的存储桶策略。有关创建此桶策略的信息，请参阅《Amazon Simple Email Service 开发人员指南》中的授予 Amazon SES 写入 S3 存储桶的权限。

本示例使用 Node.js 模块在 Amazon SES 中创建接收规则，将收到的邮件保存到 Amazon S3 桶中。创建文件名为 `ses_createreceiptrule.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个参数对象来传递创建接收规则集所需的值。要调用 `createReceiptRuleSet` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createReceiptRule params
var params = {
  Rule: {
    Actions: [
      {
        S3Action: {
          BucketName: "S3_BUCKET_NAME",
          ObjectKeyPrefix: "email",
        },
      },
    ],
    Recipients: [
      "DOMAIN | EMAIL_ADDRESS",
      /* more items */
    ],
    Enabled: true | false,
    Name: "RULE_NAME",
    ScanEnabled: true | false,
    TlsPolicy: "Optional",
  },
  RuleSetName: "RULE_SET_NAME",
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log("Rule created");
```

```
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 将创建接收规则。

```
node ses_createreceiptrule.js
```

此示例代码可在 GitHub 上的此处找到。

删除接收规则

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_deletereceiptrule.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个参数对象以传递要删除的接收规则的名称。要调用 `deleteReceiptRule` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create deleteReceiptRule params
var params = {
  RuleName: "RULE_NAME" /* required */,
  RuleSetName: "RULE_SET_NAME" /* required */,
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log("Receipt Rule Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
```

```
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 将创建接收规则集列表。

```
node ses_deletereceiptrule.js
```

此示例代码可在 GitHub 上的此处找到。

创建接收规则集

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 `ses_createreceiptruleset.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个参数对象以传递新接收规则集的名称。要调用 `createReceiptRuleSet` 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptRuleSet({ RuleSetName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 将创建接收规则集列表。

```
node ses_createreceiptruleset.js
```

此示例代码可在 GitHub 上的此处找到。

删除接收规则集

在本示例中，使用 Node.js 模块通过 Amazon SES 发送电子邮件。创建文件名为 ses_deletereceiptruleset.js 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象以传递要删除的接收规则集的名称。要调用 deleeReceiptRuleSet 方法，请创建一个 promise 来调用 Amazon SES 服务对象并传递参数。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptRuleSet({ RuleSetName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。Amazon SES 将创建接收规则集列表。

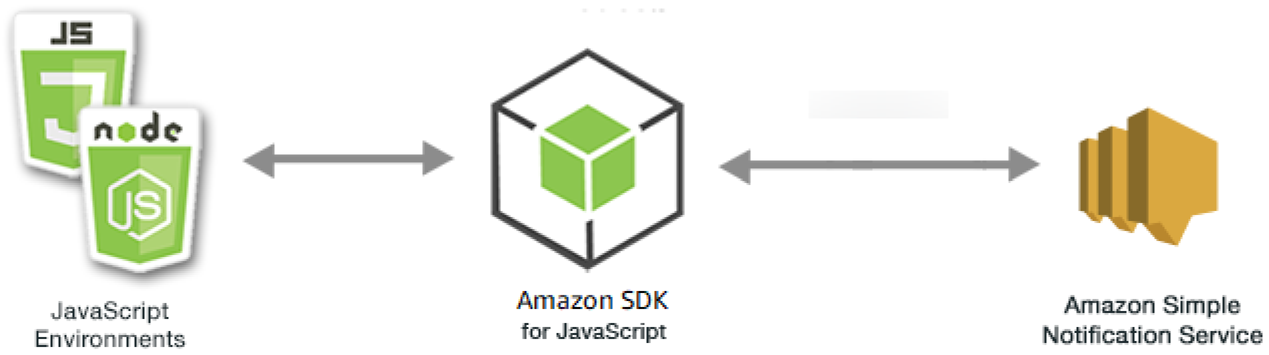
```
node ses_deletereceiptruleset.js
```

此示例代码可在 [GitHub 上的此处](#)找到。

Amazon Simple Notification Service 示例

Amazon Simple Notification Service (Amazon SNS) 是一项 Web 服务，用于协调和管理向订阅端点或客户端交付或发送消息的过程。

在 Amazon SNS 中有两种类型的客户端：发布者和订阅者，也称为生产者和消费者。



发布者通过创建消息并将消息发送至主题与订阅者进行异步交流，主题是一个逻辑访问点和通信渠道。订阅者（即 Web 服务器、电子邮件地址、Amazon SQS 队列、Lambda 函数）在其订阅主题后通过受支持协议（Amazon SQS、HTTP/S、电子邮件、SMS、Amazon Lambda）中的一种来使用或接收消息或通知。

适用于 Amazon SNS 的 JavaScript API 通过 `Class: AWS.SNS` 公开。

主题

- 在 Amazon SNS 中管理主题
- 在 Amazon SNS 中发布消息
- 在 Amazon SNS 中管理订阅
- 使用 Amazon SNS 发送 SMS 消息

在 Amazon SNS 中管理主题

此 Node.js 代码示例演示：

- 如何在 Amazon SNS 中创建可以将通知发布到的主题。
- 如何删除在 Amazon SNS 中创建的主题。
- 如何获取可用主题的列表。
- 如何获取和设置主题属性。

情景

在本示例中，您使用一系列 Node.js 模块来创建、列出和删除 Amazon SNS 主题，以及处理主题属性。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SNS` 客户端类的以下方法管理主题：

- `createTopic`
- `listTopics`
- `deleteTopic`
- `getTopicAttributes`
- `setTopicAttributes`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建主题

本示例使用 Node.js 模块创建 Amazon SNS 主题。创建文件名为 `sns_createtopic.js` 的 Node.js 模块。按前面所示配置 SDK。

创建对象，将新主题的 `Name` 传递到 `createTopic` 客户端类的 `AWS.SNS` 方法。要调用 `createTopic` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。由 promise 返回的 `data` 包含主题的 ARN。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var createTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .createTopic({ Name: "TOPIC_NAME" })
  .promise();
```

```
// Handle promise's fulfilled/rejected states
createTopicPromise
  .then(function (data) {
    console.log("Topic ARN is " + data.TopicArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_createtopic.js
```

此示例代码可在 GitHub 上的此处找到。

列出主题

本示例使用 Node.js 模块列出所有 Amazon SNS 主题。创建文件名为 `sns_listtopics.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个空对象以传递到 `listTopics` 客户端类的 `AWS.SNS` 方法。要调用 `listTopics` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。由 promise 返回的 `data` 包含您的主题 ARN 的数组。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var listTopicsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listTopics({})
  .promise();

// Handle promise's fulfilled/rejected states
listTopicsPromise
  .then(function (data) {
    console.log(data.Topics);
  })
  .catch(function (err) {
    console.error(err, err.stack);
```

```
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_listtopics.js
```

此示例代码可在 GitHub 上的此处找到。

删除主题

本示例使用 Node.js 模块删除 Amazon SNS 主题。创建文件名为 `sns_deletetopic.js` 的 Node.js 模块。按前面所示配置 SDK。

创建包含要删除的主题的 `TopicArn` 的对象，将其传递到 `deleteTopic` 客户端类的 `AWS.SNS` 方法。要调用 `deleteTopic` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var deleteTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .deleteTopic({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
deleteTopicPromise
  .then(function (data) {
    console.log("Topic Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_deletetopic.js
```

此示例代码可在 GitHub 上的此处找到。

获取主题属性

本示例使用 Node.js 模块检索 Amazon SNS 主题的属性。创建文件名为 `sns_gettopicattributes.js` 的 Node.js 模块。按前面所示配置 SDK。

创建包含要删除主题的 `TopicArn` 的对象，将其传递到 `getTopicAttributes` 客户端类的 `AWS.SNS` 方法。要调用 `getTopicAttributes` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var getTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .getTopicAttributes({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
getTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_gettopicattributes.js
```

此示例代码可在 GitHub 上的此处找到。

设置主题属性

本示例使用 Node.js 模块设置 Amazon SNS 主题的可变属性。创建文件名为 `sns_settopicattributes.js` 的 Node.js 模块。按前面所示配置 SDK。

创建包含用于属性更新参数的对象，这包括要设置其属性的主题的 `TopicArn`、要设置的属性的名称以及该属性的新值。您只能设置 `Policy`、`DisplayName` 和 `DeliveryPolicy` 属性。将参数传

递到 `setTopicAttributes` 客户端类的 `AWS.SNS` 方法。要调用 `setTopicAttributes` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create setTopicAttributes parameters
var params = {
  AttributeName: "ATTRIBUTE_NAME" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  AttributeValue: "NEW_ATTRIBUTE_VALUE",
};

// Create promise and SNS service object
var setTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setTopicAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_settopicattributes.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SNS 中发布消息

此 Node.js 代码示例演示：

- 如何将消息发布到 Amazon SNS 主题。

情景

在本示例中，您使用一系列 Node.js 模块，将消息从 Amazon SNS 发布到主题端点、电子邮件或电话号码。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SNS` 客户端类的以下方法发送消息：

- `publish`

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

将消息发布到 Amazon SNS 主题

本示例使用 Node.js 模块将消息发布到 Amazon SNS 主题。创建文件名为 `sns_publishtotopic.js` 的 Node.js 模块。按前面所示配置 SDK。

创建包含用于发布消息的参数的对象，这包括消息文本以及 Amazon SNS 主题的 ARN。有关可用 SMS 属性的详细信息，请参阅SetSMSAttributes。

将参数传递到 `publish` 客户端类的 `AWS.SNS` 方法。创建用于调用 Amazon SNS 服务对象的 promise 并传递参数对象。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create publish parameters
var params = {
  Message: "MESSAGE_TEXT" /* required */,
  TopicArn: "TOPIC_ARN",
};
```

```
// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log(
      `Message ${params.Message} sent to the topic ${params.TopicArn}`
    );
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_publishtotopic.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SNS 中管理订阅

此 Node.js 代码示例演示：

- 如何列出对 Amazon SNS 主题的所有订阅。
- 如何将电子邮件地址、应用程序端点或 Amazon Lambda 函数订阅到 Amazon SNS 主题。
- 如何从 Amazon SNS 主题取消订阅。

情景

在本示例中，您使用一系列 Node.js 模块将通知消息发布到 Amazon SNS 主题。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SNS` 客户端类的以下方法管理主题：

- subscribe
- confirmSubscription
- listSubscriptionsByTopic
- unsubscribe

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

列出对主题的订阅

本示例使用 Node.js 模块列出对 Amazon SNS 主题的所有订阅。创建文件名为 `sns_listsubscriptions.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个对象，其中包含您要列出其订阅的主题的 `TopicArn` 参数。将参数传递到 `listSubscriptionsByTopic` 客户端类的 `AWS.SNS` 方法。要调用 `listSubscriptionsByTopic` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

const params = {
  TopicArn: "TOPIC_ARN",
};

// Create promise and SNS service object
var subslistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listSubscriptionsByTopic(params)
  .promise();

// Handle promise's fulfilled/rejected states
subslistPromise
```

```
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_listsubscriptions.js
```

此示例代码可在 GitHub 上的此处找到。

将电子邮件地址订阅到主题

本示例使用 Node.js 模块将电子邮件地址订阅到 Amazon SNS 主题，以从该主题接收 SMTP 电子邮件。创建文件名为 `sns_subscribeemail.js` 的 Node.js 模块。按前面所示配置 SDK。

创建包含 `Protocol` 参数的对象，用于指定 `email` 协议、要订阅到的主题的 `TopicArn` 以及作为邮件 `Endpoint` 的电子邮件地址。将参数传递到 `subscribe` 客户端类的 `AWS.SNS` 方法。您可以使用 `subscribe` 方法，根据在所传递参数中使用的值，将多种不同的端点订阅到某个 Amazon SNS 主题，如本主题中的其它示例所示。

要调用 `subscribe` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "EMAIL" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "EMAIL_ADDRESS",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
```

```
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_subscribeemail.js
```

此示例代码可在 GitHub 上的此处找到。

将应用程序端点订阅到主题

本示例使用 Node.js 模块将移动应用程序端点订阅到 Amazon SNS 主题，以接收通知。创建文件名为 `sns_subscribeapp.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个包含 `Protocol` 参数的对象，用于指定 `application` 协议、要订阅到的主题的 `TopicArn` 以及 `Endpoint` 参数的移动应用程序终端节点的 ARN。将参数传递到 `subscribe` 客户端类的 `AWS.SNS` 方法。

要调用 `subscribe` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "application" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "MOBILE_ENDPOINT_ARN",
};
```

```
// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_subscribeapp.js
```

此示例代码可在 GitHub 上的此处找到。

将 Lambda 函数订阅到主题

本示例使用 Node.js 模块将 Amazon Lambda 函数订阅到 Amazon SNS 主题，以接收通知。创建文件名为 `sns_subscribelambda.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个包含 `Protocol` 参数的对象，指定 `lambda` 协议、要订阅到的主题的 `TopicArn` 以及作为 Amazon Lambda 参数的 `Endpoint` 函数的 ARN。将参数传递到 `subscribe` 客户端类的 `AWS.SNS` 方法。

要调用 `subscribe` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "lambda" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
```

```
  Endpoint: "LAMBDA_FUNCTION_ARN",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_subscribelambda.js
```

此示例代码可在 GitHub 上的此处找到。

从主题取消订阅

本示例使用 Node.js 模块取消对 Amazon SNS 主题的订阅。创建文件名为 `sns_unsubscribe.js` 的 Node.js 模块。按前面所示配置 SDK。

创建一个包含 `SubscriptionArn` 参数的对象，指定要取消订阅的订阅的 ARN。将参数传递到 `unsubscribe` 客户端类的 `AWS.SNS` 方法。

要调用 `unsubscribe` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .unsubscribe({ SubscriptionArn: TOPIC_SUBSCRIPTION_ARN })
```

```
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_unsubscribe.js
```

此示例代码可在 GitHub 上的此处找到。

使用 Amazon SNS 发送 SMS 消息

此 Node.js 代码示例演示：

- 如何获取和设置 Amazon SNS 的 SMS 消息发送首选项。
- 如何检查电话号码以确定是否选择退出接收 SMS 消息。
- 如何获取已选择退出接收 SMS 消息的电话号码列表。
- 如何发送 SMS 消息。

情景

您可以使用 Amazon SNS 将文本消息或 SMS 消息发送到支持 SMS 的设备上。您可以直接向电话号码发送消息，也可以使用多个电话号码订阅主题，然后通过向该主题发送消息来一次向这些电话号码发送消息。

在本示例中，您使用一系列 Node.js 模块将 SMS 文本消息从 Amazon SNS 发送到支持 SMS 的设备。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SNS` 客户端类的以下方法发布 SMS 消息：

- getSMSAttributes
- setSMSAttributes
- checkIfPhoneNumberIsOptedOut
- listPhoneNumbersOptedOut
- publish

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供凭证 JSON 文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

获取 SMS 属性

使用 Amazon SNS 来指定发送 SMS 消息的首选项，例如如何优化消息传输（在成本或可靠传输方面）、您的每月支出限额、如何记录消息传输以及是否要订阅每日 SMS 使用率报告。这些首选项通过检索得到，并设置为 Amazon SNS 的 SMS 属性。

本示例使用 Node.js 模块获取 Amazon SNS 中的当前 SMS 属性。创建文件名为 sns_getsmstype.js 的 Node.js 模块。按前面所示配置 SDK。创建包含用于获取 SMS 属性的参数的对象，包括要获取的单个属性的名称。有关可用 SMS 属性的详细信息，请参阅《Amazon Simple Notification Service API 参考》中的 SetSMSAttributes。

此示例获取 DefaultSMSType 属性，该属性控制 SMS 消息是作为 Promotional 发送（这将优化消息传送以尽可能降低成本）还是作为 Transactional 发送（这将优化消息传送以实现最高的可靠性）。将参数传递到 setTopicAttributes 客户端类的 AWS.SNS 方法。要调用 getSMSAttributes 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 response。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create SMS Attribute parameter you want to get
var params = {
```

```
  attributes: [
    "DefaultSMSType",
    "ATTRIBUTE_NAME",
    /* more items */
  ],
};

// Create promise and SNS service object
var getSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .getSMSAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
getSMSTypePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_getsmstype.js
```

此示例代码可在 GitHub 上的此处找到。

设置 SMS 属性

本示例使用 Node.js 模块获取 Amazon SNS 中的当前 SMS 属性。创建文件名为 `sns_setsmstype.js` 的 Node.js 模块。按前面所示配置 SDK。创建包含用于设置 SMS 属性的参数的对象，其中包括要设置的单个属性的名称以及为各个属性设置的值。有关可用 SMS 属性的详细信息，请参阅《Amazon Simple Notification Service API 参考》中的 SetSMSAttributes。

此示例将 `DefaultSMSType` 属性设置为 `Transactional`，这会优化消息传送以实现最高的可靠性。将参数传递到 `setTopicAttributes` 客户端类的 `AWS.SNS` 方法。要调用 `getSMSAttributes` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set region
AWS.config.update({ region: "REGION" });

// Create SMS Attribute parameters
var params = {
  attributes: {
    /* required */
    DefaultSMSType: "Transactional" /* highest reliability */,
    //'DefaultSMSType': 'Promotional' /* lowest cost */
  },
};

// Create promise and SNS service object
var setSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setSMSAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setSMSTypePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_setsmstype.js
```

此示例代码可在 GitHub 上的此处找到。

检查电话号码是否已选择不接收消息

在本示例中，使用 Node.js 模块检查电话号码，确定该号码是否已退出接收 SMS 消息。创建文件名为 `sns_checkphoneoptout.js` 的 Node.js 模块。按前面所示配置 SDK。创建一个对象，其中将要检查的电话号码包含作为参数。

此示例设置 `PhoneNumber` 参数以指定要检查的电话号码。将对象发布到 `checkIfPhoneNumberIsOptedOut` 客户端类的 `AWS.SNS` 方法。要调用 `checkIfPhoneNumberIsOptedOut` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var phonenumPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .checkIfPhoneNumberIsOptedOut({ phoneNumber: "PHONE_NUMBER" })
  .promise();

// Handle promise's fulfilled/rejected states
phonenumPromise
  .then(function (data) {
    console.log("Phone Opt Out is " + data.isOptedOut);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_checkphoneoptout.js
```

此示例代码可在 GitHub 上的此处找到。

列出已退出的电话号码

在本示例中，使用 Node.js 模块获取已退出接收 SMS 消息的电话号码列表。创建文件名为 `sns_listnumbersoptedout.js` 的 Node.js 模块。按前面所示配置 SDK。创建一个空对象作为参数。

将对象发布到 `listPhoneNumbersOptedOut` 客户端类的 `AWS.SNS` 方法。要调用 `listPhoneNumbersOptedOut` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
```

```
var phonelistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listPhoneNumbersOptedOut({})
  .promise();

// Handle promise's fulfilled/rejected states
phonelistPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_listnumbersoptedout.js
```

此示例代码可在 GitHub 上的此处找到。

发布 SMS 消息

在本示例中，使用 Node.js 模块将 SMS 消息发布到电话号码。创建文件名为 `sns_publishsms.js` 的 Node.js 模块。按前面所示配置 SDK。创建一个包含 `Message` 和 `PhoneNumber` 参数的对象。

在发送 SMS 消息时，请使用 E.164 格式指定电话号码。E.164 是用于国际电信的电话号码结构标准。遵循此格式的电话号码最多可包含 15 位，并以加号 (+) 和国家/地区代码作为前缀。例如，E.164 格式的美国电话号码将显示为 +1001XXX5550100。

此示例设置 `PhoneNumber` 参数以指定将消息发送到的电话号码。将对象发布到 `publish` 客户端类的 `AWS.SNS` 方法。要调用 `publish` 方法，请创建一个 promise 来调用 Amazon SNS 服务对象并传递参数对象。然后处理 promise 回调中的 `response`。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create publish parameters
var params = {
  Message: "TEXT_MESSAGE" /* required */,
  PhoneNumber: "E.164_PHONE_NUMBER",
};
```

```
// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

要运行示例，请在命令行中键入以下内容。

```
node sns_publishsms.js
```

此示例代码可在 GitHub 上的此处找到。

Amazon SQS 示例

Amazon Simple Queue Service（Amazon SQS）是一种快速、可靠、可扩展且完全托管的消息队列服务。Amazon SQS 使您能够分离云应用程序的组件。Amazon SQS 包括具有高吞吐量和“至少一次”处理的标准队列，以及提供 FIFO（先进先出）交付和“仅一次”处理的 FIFO 队列。

适用于 Amazon SQS 的 JavaScript API 通过 AWS.SQS 客户端类公开。有关使用 Amazon SQS 客户端类的更多信息，请参阅 API 参考中的 `Class: AWS.SQS`。

主题

- 在 &SQS; 中使用队列
- 在 Amazon SQS 中发送和接收消息
- 在 &SQS; 中管理可见性超时
- 在 Amazon SQS 中启用长轮询
- 在 &SQS; 中使用死信队列

在 &SQS; 中使用队列

此 Node.js 代码示例演示：

- 如何获取所有消息队列的列表
- 如何获取特定队列的 URL
- 如何创建和删除队列

关于示例

在本示例中，使用了一系列 Node.js 模块来处理队列。这些 Node.js 模块使用 SDK for JavaScript 以启用队列来调用 `AWS.SQS` 客户端类的以下方法：

- `listQueues`
- `createQueue`
- `getQueueUrl`
- `deleteQueue`

有关 Amazon SQS 消息的更多信息，请参阅《Amazon Simple Queue Service Developer Guide》中的 How Queues Work。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

列出队列

创建文件名为 sqs_listqueues.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 AWS.SQS 服务对象。创建一个包含列出队列所需的参数的 JSON 对象，默认情况下这是个空对象。调用 listQueues 方法以检索队列的列表。回调函数将返回所有队列的 URL。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {};

sqs.listQueues(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrls);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_listqueues.js
```

此示例代码可在 GitHub 上的此处找到。

创建队列

创建文件名为 sqs_createqueue.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 AWS.SQS 服务对象。创建一个包含列出队列所需的参数的 JSON 对象，这必须包括所创建队列的名称。参数也可以包含队列的属性，例如消息发送延迟的秒数或者保留所收到消息的秒数。调用 createQueue 方法。回调函数将返回已创建队列的 URL。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    DelaySeconds: "60",
    MessageRetentionPeriod: "86400",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_createqueue.js
```

此示例代码可在 GitHub 上的此处找到。

获取队列的 URL

创建文件名为 `sqs_getqueueurl.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 `AWS.SQS` 服务对象。创建一个包含列出队列所需的属性的 JSON 对象，这必须包括您需要其 URL 的队列的名称。调用 `getQueueUrl` 方法。回调函数将返回指定队列的 URL。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
```

```
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
};

sqs.getQueueUrl(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_getqueueurl.js
```

此示例代码可在 GitHub 上的此处找到。

删除队列

创建文件名为 `sqs_deletequeue.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 `AWS.SQS` 服务对象。创建一个包含删除队列所需的参数的 JSON 对象，这包括您要删除的队列的 URL。调用 `deleteQueue` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.deleteQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
```

```
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_deletequeue.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SQS 中发送和接收消息

此 Node.js 代码示例演示：

- 如何在队列中发送消息。
- 如何在队列中接收消息。
- 如何在队列中删除消息。

情景

在本示例中，使用了一系列 Node.js 模块来发送和接收消息。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SQS` 客户端类的以下方法来发送和接收消息：

- `sendMessage`
- `receiveMessage`
- `deleteMessage`

有关 Amazon SQS 消息的更多信息，请参阅《Amazon Simple Queue Service Developer Guide》中的 Sending a Message to an Amazon SQS Queue 和 Receiving and Deleting a Message from an Amazon SQS Queue。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建 Amazon SQS 队列。有关创建队列的示例，请参阅在 &SQS; 中使用队列。

向队列发送消息

创建文件名为 sqs_sendmessage.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 AWS.SQS 服务对象。创建一个包含消息所需的参数的 JSON 对象，其中包括您要将此消息发送到的队列的 URL。在本示例中，消息提供了最畅销科幻小说排行榜中某本书籍的详细信息，包括书名、作者以及上榜周数。

调用 sendMessage 方法。回调函数将返回消息的唯一 ID。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  // Remove DelaySeconds parameter and value for FIFO queues
  DelaySeconds: 10,
  MessageAttributes: {
    Title: {
      DataType: "String",
      StringValue: "The Whistler",
    },
    Author: {
      DataType: "String",
      StringValue: "John Grisham",
    },
    WeeksOn: {
      DataType: "Number",
      StringValue: "6",
    },
  },
  MessageBody:
    "Information about current NY Times fiction bestseller for week of 12/11/2016.",
```

```
  // MessageDeduplicationId: "TheWhistler",  // Required for FIFO queues
  // MessageGroupId: "Group1",  // Required for FIFO queues
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.sendMessage(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.MessageId);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_sendmessage.js
```

此示例代码可在 GitHub 上的此处找到。

接收和删除来自队列的消息

创建文件名为 `sqs_receivemessage.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 `AWS.SQS` 服务对象。创建一个包含消息所需的参数的 JSON 对象，其中包括您要用于接收消息的队列的 URL。在本示例中，参数指定所有消息属性的收件人，以及不超过 10 条消息的收件人。

调用 `receiveMessage` 方法。回调函数将返回 `Message` 对象的数组，从中您可以检索各个消息的 `ReceiptHandle`，您可在以后用来删除该消息。创建包含删除消息所需参数的另一个 JSON 对象，这些参数为队列的 URL 和 `ReceiptHandle` 值。调用 `deleteMessage` 方法以删除您收到的消息。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";

var params = {
```

```
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 10,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  VisibilityTimeout: 20,
  WaitTimeSeconds: 0,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Receive Error", err);
  } else if (data.Messages) {
    var deleteParams = {
      QueueUrl: queueURL,
      ReceiptHandle: data.Messages[0].ReceiptHandle,
    };
    sqs.deleteMessage(deleteParams, function (err, data) {
      if (err) {
        console.log("Delete Error", err);
      } else {
        console.log("Message Deleted", data);
      }
    });
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_receivemessage.js
```

此示例代码可在 GitHub 上的此处找到。

在 &SQS; 中管理可见性超时

此 Node.js 代码示例演示：

- 如何指定队列接收的消息不可见的时间间隔。

情景

在此示例中，使用了 Node.js 模块来管理可见性超时。该 Node.js 模块使用 SDK for JavaScript，通过 AWS.SQS 客户端类的以下方法来管理可见性超时：

- changeMessageVisibility

有关 Amazon SQS 可见性超时的更多信息，请参阅《Amazon Simple Queue Service Developer Guide》中的 Visibility Timeout。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建 Amazon SQS 队列。有关创建队列的示例，请参阅在 &SQS; 中使用队列。
- 将消息发送到队列。有关将消息发送到队列的示例，请参阅在 Amazon SQS 中发送和接收消息。

更改可见性超时

创建文件名为 sqs_changingvisibility.js 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon Simple Queue Service，请创建一个 AWS.SQS 服务对象。从队列接收消息。

从队列收到消息时，创建一个包含设置超时所需的参数的 JSON 对象，其中包括消息所在的队列的 URL、收到消息时返回的 ReceiptHandle，以及新的超时（以秒为单位）。调用 changeMessageVisibility 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region to us-west-2
AWS.config.update({ region: "us-west-2" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "https://sqs.REGION.amazonaws.com/ACCOUNT-ID/QUEUE-NAME";
```

```
var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Receive Error", err);
  } else {
    // Make sure we have a message
    if (data.Messages != null) {
      var visibilityParams = {
        QueueUrl: queueURL,
        ReceiptHandle: data.Messages[0].ReceiptHandle,
        VisibilityTimeout: 20, // 20 second timeout
      };
      sqs.changeMessageVisibility(visibilityParams, function (err, data) {
        if (err) {
          console.log("Delete Error", err);
        } else {
          console.log("Timeout Changed", data);
        }
      });
    } else {
      console.log("No messages to change");
    }
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_changingvisibility.js
```

此示例代码可在 GitHub 上的此处找到。

在 Amazon SQS 中启用长轮询

此 Node.js 代码示例演示：

- 如何为新创建的队列启用长轮询
- 如何为现有队列启用长轮询
- 如何在收到消息时启用长轮询

情景

长轮询使 Amazon SQS 在发送响应之前等待指定的时间，以便消息在队列中变得可用，从而减少空响应的数量。此外，长轮询通过查询所有服务器而不是执行服务器采样，消除了假的空响应。要启用长轮询，您必须为接收的消息指定非零等待时间。为此，您可以设置队列的 `ReceiveMessageWaitTimeSeconds` 参数或者设置在收到消息时的 `WaitTimeSeconds` 参数。

在本示例中，使用了一系列 Node.js 模块来启用长轮询。这些 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SQS` 客户端类的以下方法来启用长轮询：

- `setQueueAttributes`
- `receiveMessage`
- `createQueue`

有关 Amazon SQS 长轮询的更多信息，请参阅《Amazon Simple Queue Service Developer Guide》中的 Long Polling。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。

创建队列时启用长轮询

创建文件名为 `sqs_longpolling_createqueue.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 `AWS.SQS` 服务对象。创建一个包含创建队列所需的参数的 JSON 对象，其中包括 `ReceiveMessageWaitTimeSeconds` 参数的非零值。调用 `createQueue` 方法。接下来为队列启用长轮询。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_longpolling_createqueue.js
```

此示例代码可在 GitHub 上的此处找到。

在现有队列上启用长轮询

创建文件名为 `sqs_longpolling_existingqueue.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon Simple Queue Service，请创建一个 `AWS.SQS` 服务对象。创建一个包含设置队列属性所需的参数的 JSON 对象，其中包括 `ReceiveMessageWaitTimeSeconds` 参数的非零值和队列的 URL。调用 `setQueueAttributes` 方法。接下来为队列启用长轮询。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
```

```
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.setQueueAttributes(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_longpolling_existingqueue.js
```

此示例代码可在 GitHub 上的此处找到。

在接收消息时启用长轮询

创建文件名为 `sqs_longpolling_receivemessage.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon Simple Queue Service，请创建一个 `AWS.SQS` 服务对象。创建一个包含接收消息所需的参数的 JSON 对象，其中包括 `WaitTimeSeconds` 参数的非零值和队列的 URL。调用 `receiveMessage` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";

var params = {
  AttributeNames: ["SentTimestamp"],
```

```
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  WaitTimeSeconds: 20,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_longpolling_receivemessage.js
```

此示例代码可在 GitHub 上的此处找到。

在 &SQS; 中使用死信队列

此 Node.js 代码示例演示：

- 如何使用队列接收和保留其他队列无法处理的消息。

情景

其他 (源) 队列可将无法成功处理的消息转到死信队列。您可以在死信队列中留出和隔离这些消息以确定其处理失败的原因。您必须单独配置将消息发送到死信队列的每个源队列。多个队列可将一个死信队列作为目标。

在此示例中，使用 Node.js 模块将消息路由到死信队列。该 Node.js 模块使用 SDK for JavaScript，通过 `AWS.SQS` 客户端类的以下方法来使用死信队列：

- `setQueueAttributes`

有关 Amazon SQS 死信队列的更多信息，请参阅《Amazon Simple Queue Service Developer Guide》中的 Using Amazon SQS Dead Letter Queues。

先决条件任务

要设置和运行此示例，您必须先完成以下任务：

- 安装 Node.js。有关安装 Node.js 的更多信息，请参阅 Node.js 网站。
- 使用用户凭证创建共享配置文件。有关提供共享凭证文件的更多信息，请参阅从共享凭证文件加载 Node.js 中的凭证。
- 创建要用作死信队列的 Amazon SQS 队列。有关创建队列的示例，请参阅在 &SQS; 中使用队列。

配置源队列

创建充当死信队列的队列之后，必须配置其他队列以将未处理的消息路由到死信队列。要执行此操作，请指定重新驱动策略，标识用作死信队列的队列，以及单个消息的接收次数上限 (超过该次数后将消息路由到死信队列)。

创建文件名为 `sqs_deadletterqueue.js` 的 Node.js 模块。请确保按前面所示配置开发工具包。要访问 Amazon SQS，请创建 `AWS.SQS` 服务对象。创建一个包含更新队列属性所需的参数的 JSON 对象，其中包括同时指定死信队列 ARN 和 `maxReceiveCount` 值的 `RedrivePolicy` 参数。另请指定您要配置的 URL 源队列。调用 `setQueueAttributes` 方法。

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    RedrivePolicy:
      '{"deadLetterTargetArn":"DEAD_LETTER_QUEUE_ARN","maxReceiveCount":"10"}',
  },
  QueueUrl: "SOURCE_QUEUE_URL",
};

sqs.setQueueAttributes(params, function (err, data) {
```

```
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

要运行示例，请在命令行中键入以下内容。

```
node sqs_deadletterqueue.js
```

此示例代码可在 GitHub 上的此处找到。

教程

以下教程为您演示如何执行与使用适用于 JavaScript 的 Amazon SDK相关的不同任务。

主题

- 教程：在 Amazon EC2 实例上设置 Node.js

教程：在 Amazon EC2 实例上设置 Node.js

将 Node.js 与 SDK for JavaScript 结合使用的一个常见场景是在 Amazon Elastic Compute Cloud (Amazon EC2) 实例上设置和运行 Node.js Web 应用程序。在本教程中，您将创建一个 Linux 实例，使用 SSH 连接到该实例，然后安装 Node.js 以在该实例上运行。

先决条件

本教程假定您已经使用公有 DNS 名称启动 Linux 实例，该实例可从 Internet 访问并且您可以使用 SSH 来连接。有关更多信息，请参阅《Amazon EC2 用户指南》中的步骤 1：启动实例。

> **Important**
>
> 在启动新的 Amazon EC2 实例时，请使用 Amazon Linux 2023 Amazon 机器映像 (AMI)。

还必须将安全组配置为允许 SSH（端口 22）、HTTP（端口 80）和 HTTPS（端口 443）连接。有关这些先决条件的更多信息，请参阅《Amazon EC2 用户指南》中的设置 Amazon EC2。

过程

以下过程可帮助您在 Amazon Linux 实例上安装 Node.js。您可以使用此服务器来托管 Node.js Web 应用程序。

在 Linux 实例上设置 Node.js

1. 使用 SSH 以 `ec2-user` 身份连接您的 Linux 实例。
2. 通过在命令行中键入以下内容，安装节点版本管理器 (nvm)。

> **Warning**
>
> Amazon 不控制以下代码。在运行之前，请务必验证其真实性和完整性。有关此代码的更多信息，请参阅 nvm GitHub 存储库。

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

由于 nvm 可以安装多个版本的 Node.js 并允许您在各个版本之间切换，我们将使用 nvm 安装 Node.js。

3. 通过在命令行中键入以下内容来加载 nvm。

```
source ~/.bashrc
```

4. 通过在命令行键入以下命令，使用 nvm 安装 Node.js 的最新 LTS 版本。

```
nvm install --lts
```

安装 Node.js 还会安装 Node Package Manager（npm），以便您可以根据需要安装其它模块。

5. 通过在命令行键入以下内容，测试 Node.js 已安装并正确运行。

```
node -e "console.log('Running Node.js ' + process.version)"
```

这将显示以下消息，其中显示正在运行的 Node.js 的版本。

`Running Node.js` *`VERSION`*

> **Note**
>
> 节点安装仅适用于当前的 Amazon EC2 会话。如果您重启 CLI 会话，则需要使用 nvm 来启用已安装的节点版本。如果实例终止，则需要重新安装节点。另一种方法是在获得要保留的配置后，制作一个 Amazon EC2 实例的亚马逊机器映像（AMI），如以下主题所述。

创建 Amazon 系统映像

在 Amazon EC2 实例上安装 Node.js 后，您可以从该实例创建亚马逊机器映像（AMI）。创建 AMI 可通过同一个 Node.js 安装，轻松地预置多个 Amazon EC2 实例。有关从现有实例创建 AMI 的更多信息，请参阅《Amazon EC2 用户指南》中的创建一个由 Amazon EBS 支持的 Linux AMI。

相关资源

有关本主题中使用的命令和软件的更多信息，请参阅以下网页：

- 节点版本管理器 (nvm)：请参阅 GitHub 上的 nvm 存储库。
- 节点程序包管理器 (npm)：请参阅 npm 网站。

JavaScript API 参考

适用于最新版 SDK for JavaScript 的 API 参考信息位于：

适用于 JavaScript 的 Amazon SDK API Reference Guide。

GitHub 上的开发工具包更改日志

版本 2.4.8 及更高版本的更改日志位于：

更改日志。

迁移到适用于 JavaScript 的 Amazon SDK 的 v3

适用于 JavaScript 的 Amazon SDK 版本 3 是对版本 2 的重大重写。有关迁移到版本 3 的更多信息，请参阅《适用于 JavaScript 的 Amazon SDK Developer Guide v3》中的 Migrate from version 2.x to 3.x of the 适用于 JavaScript 的 Amazon SDK。

此 Amazon 产品或服务的安全性

云安全性一直是 Amazon Web Services (Amazon) 的重中之重。作为 Amazon 客户，您将从专为满足大多数安全敏感型企业的要求而打造的数据中心和网络架构中受益。安全性是 Amazon 和您的共同责任。责任共担模式将其描述为云的安全性和云中的安全性。

云的安全性：Amazon 负责保护运行在 Amazon 云中提供的所有服务的基础设施，并向您提供可安全使用的服务。我们的安全责任是 Amazon 中的最高优先级，作为 Amazon 合规性计划的一部分，我们安全性的有效性由第三方审计员定期进行测试和验证。

云中的安全性：您的责任由您所使用的 Amazon 服务以及其他因素决定，包括您数据的敏感性、您企业的要求以及适用的法律法规。

此 Amazon 产品或服务通过它支持的特定 Amazon Web Services (Amazon) 服务遵循责任共担模式。有关 Amazon 服务安全性信息，请参阅 Amazon 服务安全性文档页面和合规性计划规定的 Amazon 合规性工作范围内的 Amazon 服务。

主题

- 本 Amazon 产品或服务中的数据保护
- 身份和访问管理
- 此 Amazon 产品或服务的合规性验证
- 此 Amazon 产品或服务的故障恢复能力
- 此 Amazon 产品或服务的基础设施安全性
- 强制实施最低版本 TLS

本 Amazon 产品或服务中的数据保护

Amazon 责任共担模式适用于本 Amazon 产品或服务中的数据保护。如该模式中所述，Amazon 负责保护运行所有 Amazon Web Services 云 的全球基础结构。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 Amazon Web Services 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。

出于数据保护目的，建议您保护 Amazon Web Services 账户 凭证并使用 Amazon IAM Identity Center 或 Amazon Identity and Access Management（IAM）设置单个用户。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）。
- 使用 SSL/TLS 与 Amazon 资源进行通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用 Amazon CloudTrail 设置 API 和用户活动日记账记录。有关使用 CloudTrail 跟踪来捕获 Amazon 活动的信息，请参阅《Amazon CloudTrail 用户指南》中的使用 CloudTrail 跟踪。
- 使用 Amazon 加密解决方案以及 Amazon Web Services 服务中的所有默认安全控制。
- 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。
- 如果在通过命令行界面或 API 访问 Amazon 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://www.amazonaws.cn/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）。这包括使用本 Amazon 产品或服务或者其他使用控制台、API、Amazon CLI 或 Amazon SDK 的 Amazon Web Services 服务时。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。

身份和访问管理

Amazon Identity and Access Management（IAM）是一项 Amazon Web Services 服务，可以帮助管理员安全地控制对 Amazon 资源的访问。IAM 管理员控制谁可以通过身份验证（登录）和授权（具有权限）来使用 Amazon 资源。IAM 是一项无需额外费用即可使用的。Amazon Web Services 服务

主题

- 受众
- 使用身份进行身份验证
- 使用策略管理访问
- Amazon Web Services 服务如何与 IAM 协同工作
- 对 Amazon 身份和访问进行问题排查

受众

使用 Amazon Identity and Access Management（IAM）的方式因您可以在 Amazon 中执行的操作而异。

服务用户 – 如果使用 Amazon Web Services 服务来执行任务，则管理员会为您提供所需的凭证和权限。当您使用更多 Amazon 特征来完成工作时，您可能需要额外权限。了解如何管理访问权限有助于您向管理员请求适合的权限。如果您无法访问 Amazon 中的功能，请参阅对 Amazon 身份和访问进行问题排查或您所使用的 Amazon Web Services 服务的用户指南。

服务管理员：如果您在公司负责管理 Amazon 资源，则您可能具有 Amazon 的完全访问权限。您有责任确定您的服务用户应访问哪些 Amazon 特征和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要详细了解您的公司如何将 IAM 与 Amazon 结合使用，请参阅您所使用的 Amazon Web Services 服务的用户指南。

IAM 管理员：如果您是 IAM 管理员，您可能希望了解如何编写策略以管理对 Amazon 的访问权限的详细信息。要查看您可在 IAM 中使用的 Amazon 基于身份的策略示例，请参阅您所使用的 Amazon Web Services 服务的用户指南。

使用身份进行身份验证

身份验证是您使用身份凭证登录 Amazon 的方法。您必须作为 Amazon Web Services 账户根用户、IAM 用户或通过担任 IAM 角色进行身份验证。

对于编程访问，Amazon 提供了 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 Amazon 签名版本 4。

Amazon Web Services 账户 根用户

当您创建 Amazon Web Services 账户 时，最初使用的是一个对所有 Amazon Web Services 服务和资源拥有完全访问权限的登录身份（称为 Amazon Web Services 账户根用户）。我们强烈建议不要使用根用户进行日常任务。有关需要根用户凭证的任务，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，请要求人类用户必须使用带有身份提供者的联合身份验证才能使用临时凭证访问 Amazon Web Services 服务。

联合身份是来自企业目录、Web 身份提供者的用户，或 Amazon Directory Service 中的用户（这些用户使用来自身份源的凭证访问 Amazon Web Services 服务）。联合身份代入可提供临时凭证的角色。

IAM 用户和群组

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带有身份提供商的联合身份验证才能使用临时凭证访问 Amazon。

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 Amazon CLI 或 Amazon API 操作来担任角色。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。

使用策略管理访问

您将创建策略并将其附加到 Amazon 身份或资源，以控制 Amazon 中的访问。策略在与身份或资源关联时定义权限。当主体发出请求时，Amazon 会评估这些策略。大多数策略在 Amazon 中存储为 JSON 文档。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。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。示例包括 IAM 角色信任策略和 Amazon S3 存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。您必须在基于资源的策略中指定主体。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用来自 IAM 的 Amazon 托管式策略。

访问控制列表（ACL）

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

Amazon S3、Amazon WAF 和 Amazon VPC 是支持 ACL 的服务示例。要了解有关 ACL 的更多信息，请参阅《Amazon Simple Storage Service 开发人员指南》中的访问控制列表（ACL）概览。

其他策略类型

Amazon 支持额外的策略类型，这些策略类型可以设置由更常用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（SCP）– 指定 Amazon Organizations 中组织或组织单元的最大权限。有关更多信息，请参阅《Amazon Organizations 用户指南》中的服务控制策略。
- 资源控制策略（RCP）– 设置对账户中资源的最大可用权限。有关更多信息，请参阅《Amazon Organizations 用户指南》中的资源控制策略（RCP）。
- 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 Amazon 如何确定在涉及多种策略类型时是否允许请求，请参阅《IAM 用户指南》中的策略评估逻辑。

Amazon Web Services 服务如何与 IAM 协同工作

要大致了解 Amazon Web Services 服务如何与大多数 IAM 功能结合使用，请参阅《IAM 用户指南》中的与 IAM 结合使用的 Amazon 服务。

要学习如何将特定的 Amazon Web Services 服务与 IAM 结合使用，请参阅相关服务的《用户指南》的安全部分。

对 Amazon 身份和访问进行问题排查

使用以下信息可帮助您诊断和修复在使用 Amazon 和 IAM 时可能遇到的常见问题。

主题

- 我无权在 Amazon 中执行操作
- 我无权执行 iam:PassRole
- 我希望允许我的 Amazon Web Services 账户以外的人访问我的 Amazon 资源

我无权在 Amazon 中执行操作

如果您收到错误提示，指明您无权执行某个操作，则必须更新策略以允许执行该操作。

当 `mateojackson` IAM 用户尝试使用控制台查看有关虚构 *`my-example-widget`* 资源的详细信息，但不拥有虚构 `awes:`*`GetWidget`* 权限时，会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

在此情况下，必须更新 `mateojackson` 用户的策略，以允许使用 `awes:`*`GetWidget`* 操作访问 *`my-example-widget`* 资源。

如果您需要帮助，请联系 Amazon 管理员。您的管理员是提供登录凭证的人。

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。Amazon

有些 Amazon Web Services 服务 允许您将现有角色传递到该服务，而不是创建新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 Amazon 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 Amazon 管理员。您的管理员是提供登录凭证的人。

我希望允许我的 Amazon Web Services 账户以外的人访问我的 Amazon 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（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解 Amazon 是否支持这些特征，请参阅 Amazon Web Services 服务如何与 IAM 协同工作
- 要了解如何为您拥有的 Amazon Web Services 账户 中的资源提供访问权限，请参阅《IAM 用户指南》中的为您拥有的另一个 Amazon Web Services 账户 中的 IAM 用户提供访问权限。
- 要了解如何为第三方 Amazon Web Services 账户 提供您的资源的访问权限，请参阅《IAM 用户指南》中的为第三方拥有的 Amazon Web Services 账户 提供访问权限。
- 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。

此 Amazon 产品或服务的合规性验证

要了解某个 Amazon Web Services 服务 是否在特定合规性计划范围内，请参阅合规性计划范围内的 Amazon Web Services 服务，然后选择您感兴趣的合规性计划。有关常规信息，请参阅 、、Amazon Web Services 合规性计划。

您可以使用 Amazon Artifact 下载第三方审计报告。有关更多信息，请参阅在 Amazon Artifact 中下载报告。

您在使用 Amazon Web Services 服务 时的合规性责任由您的数据的敏感性、您公司的合规性目标以及适用的法律法规决定。有关您在使用 Amazon Web Services 服务时的合规责任的更多信息，请参阅 Amazon 安全性文档。

此 Amazon 产品或服务通过它支持的特定 Amazon Web Services（Amazon）服务遵循责任共担模式。有关 Amazon 服务安全性信息，请参阅 Amazon 服务安全性文档页面和合规性计划规定的 Amazon 合规性工作范围内的 Amazon 服务。

此 Amazon 产品或服务的故障恢复能力

Amazon 全球基础设施围绕 Amazon Web Services 区域和可用区而构建。

Amazon Web Services 区域 提供多个在物理上独立且隔离的可用区，这些可用区与延迟率低、吞吐量高且冗余性高的网络连接在一起。

利用可用区，您可以设计和运营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 Amazon 区域和可用区的更多信息，请参阅 Amazon 全球基础架构。

此 Amazon 产品或服务通过它支持的特定 Amazon Web Services (Amazon) 服务遵循责任共担模式。有关 Amazon 服务安全性信息，请参阅 Amazon 服务安全性文档页面和合规性计划规定的 Amazon 合规性工作范围内的 Amazon 服务。

此 Amazon 产品或服务的基础设施安全性

此 Amazon 产品或服务使用托管式服务，因此受 Amazon 全球网络安全保护。有关 Amazon 安全服务以及 Amazon 如何保护基础设施的信息，请参阅 Amazon 云安全。要按照基础设施安全最佳实践设计您的 Amazon 环境，请参阅《安全性支柱 Amazon Well‐Architected Framework》中的基础设施保护。

您可以使用 Amazon 发布的 API 调用通过网络访问此 Amazon 产品或服务。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 Amazon Security Token Service（Amazon STS）生成临时安全凭证来对请求进行签名。

此 Amazon 产品或服务通过它支持的特定 Amazon Web Services（Amazon）服务遵循责任共担模式。有关 Amazon 服务安全性信息，请参阅 Amazon 服务安全性文档页面和合规性计划规定的 Amazon 合规性工作范围内的 Amazon 服务。

强制实施最低版本 TLS

要提高与 Amazon 服务通信时的安全性，请将 适用于 JavaScript 的 Amazon SDK 配置为使用 TLS 1.2 或更高版本。

传输层安全性协议（TLS）是 Web 浏览器和其它应用程序使用的一种协议，用于确保通过网络交换的数据的隐私和完整性。

> **Important**
>
> 自 2024 年 6 月 10 日起，我们宣布 TLS 1.3 在每个 Amazon 区域的 Amazon 服务 API 端点上均可用。适用于 JavaScript 的 Amazon SDK v2 本身不会协商 TLS 版本。相反，它使用由 Node.js 确定的 TLS 版本，该版本可通过 `https.Agent` 进行配置。Amazon 建议使用 Node.js 的当前活动 LTS 版本。

在 Node.js 中验证并强制执行 TLS

将 适用于 JavaScript 的 Amazon SDK 与 Node.js 结合使用时，将使用底层 Node.js 安全层设置 TLS 版本。

Node.js 12.0.0 及更高版本使用支持 TLS 1.3 的最低版本 OpenSSL 1.1.1b。适用于 JavaScript 的 Amazon SDK v2 在 TLS 1.3 可用时默认使用该版本，但如果需要，可默认使用较低的版本。

验证 OpenSSL 和 TLS 的版本

要获取计算机上的 Node.js 使用的 OpenSSL 版本，请运行以下命令。

```
node -p process.versions
```

列表中的 OpenSSL 版本是 Node.js 使用的版本，如以下示例所示。

```
openssl: '1.1.1b'
```

要获取计算机上的 Node.js 使用的 TLS 版本，请启动 Node shell 并按顺序运行以下命令。

```
> var tls = require("tls");
> var tlsSocket = new tls.TLSSocket();
```

```
> tlsSocket.getProtocol();
```

最后一条命令输出 TLS 版本，如以下示例所示。

```
'TLSv1.3'
```

Node.js 默认使用此版本的 TLS，如果调用不成功，则会尝试协商另一个版本的 TLS。

检查支持的最低和最高 TLS 版本

开发人员可以使用以下脚本来检查 Node.js 中支持的最低和最高 TLS 版本：

```
var tls = require("tls");
console.log("Supported TLS versions:", tls.DEFAULT_MIN_VERSION + " to " +
 tls.DEFAULT_MAX_VERSION);
```

最后一个命令输出默认的最低和最高 TLS 版本，如以下示例所示。

```
Supported TLS versions: TLSv1.2 to TLSv1.3
```

强制使用最低版本的 TLS

当调用失败时，Node.js 会协商 TLS 的版本。您可以在此协商期间强制执行允许的最低 TLS 版本，无论是在命令行运行脚本时，还是根据您的 JavaScript 代码中的请求。

要通过命令行指定最低 TLS 版本，必须使用 Node.js 版本 11.4.0 或更高版本。要安装特定的 Node.js 版本，请先按照 [Node Version Manager Installing and Updating](https://github.com/nvm-sh/nvm) 中的步骤安装 Node Version Manager（nvm）。然后运行以下命令来安装并使用特定版本的 Node.js。

```
nvm install 11
nvm use 11
```

Enforcing TLS 1.2

要强制规定 TLS 1.2 是允许的最低版本，请在运行脚本时指定 `--tls-min-v1.2` 参数，如以下示例所示。

```
node --tls-min-v1.2 yourScript.js
```

要在 JavaScript 代码中为特定请求指定允许的最低 TLS 版本，请使用 `httpOptions` 参数指定协议，如以下示例所示。

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
    region: "us-west-2",
    requestHandler: new NodeHttpHandler({
        httpsAgent: new https.Agent(
            {
                secureProtocol: 'TLSv1_2_method'
            }
        )
    })
});
```

Enforcing TLS 1.3

要强制规定 TLS 1.3 是允许的最低版本，请在运行脚本时指定 `--tls-min-v1.3` 参数，如以下示例所示。

```
node --tls-min-v1.3 yourScript.js
```

要在 JavaScript 代码中为特定请求指定允许的最低 TLS 版本，请使用 `httpOptions` 参数指定协议，如以下示例所示。

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
    region: "us-west-2",
    requestHandler: new NodeHttpHandler({
        httpsAgent: new https.Agent(
            {
                secureProtocol: 'TLSv1_3_method'
            }
        )
    })
});
```

在浏览器脚本中验证并强制执行 TLS

当您在浏览器脚本中使用 SDK for JavaScript 时，浏览器设置会控制所使用的 TLS 版本。浏览器使用的 TLS 版本无法通过脚本发现或设置，必须由用户配置。要验证和强制执行浏览器脚本中使用的 TLS 版本，请参阅特定浏览器的说明。

Microsoft Internet Explorer

1. 打开 Internet Explorer。
2. 从菜单栏中选择工具 - Internet 选项 - 高级选项卡。
3. 向下滚动到安全类别，手动选中使用 TLS 1.2 选项框。
4. 单击确定。
5. 关闭浏览器并重新启动 Internet Explorer。

Microsoft Edge

1. 在 Windows 菜单搜索框中，键入 `Internet options`。
2. 在最佳匹配下，单击 Internet 选项。
3. 在 Internet 属性窗口的高级选项卡上，向下滚动到安全部分。
4. 选中用户 TLS 1.2 复选框。
5. 单击确定。

Google Chrome

1. 打开 Google Chrome。
2. 按 Alt F 并选择设置。
3. 向下滚动并选择显示高级设置...。
4. 向下滚动到系统部分，然后单击打开代理设置...。
5. 选择高级选项卡。
6. 向下滚动到安全类别，手动选中使用 TLS 1.2 选项框。
7. 单击确定。
8. 关闭浏览器并重启 Google Chrome。

Mozilla Firefox

1. 打开 Firefox。
2. 在地址栏中键入 about:config，然后按 Enter。
3. 在搜索字段中输入 tls。找到并双击 security.tls.version.min 条目。
4. 将整数值设置为 3 以强制将 TLS 1.2 协议设为默认协议。
5. 单击确定。
6. 关闭浏览器并重启 Mozilla Firefox。

Apple Safari

没有启用 SSL 协议的选项。如果您使用的是 Safari 浏览器 7 或更高版本，则会自动启用 TLS 1.2。

其他资源

下列链接提供了可与适用于 JavaScript 的 Amazon SDK结合使用的其他资源。

Amazon SDKs and Tools Reference Guide

Amazon SDKs and Tools Reference Guide 还包含许多 Amazon SDK 中常见的设置、功能和其它基础概念。

JavaScript 开发工具包论坛

您可以在 JavaScript SDK 论坛中找到 SDK for JavaScript 用户感兴趣的问题和讨论。

GitHub 上的 JavaScript 开发工具包和开发人员指南

GitHub 上有多个针对 SDK for JavaScript 的存储库。

- SDK 存储库中提供了最新的 SDK for JavaScript。
- 《SDK for JavaScript 开发人员指南》（本文档）以 Markdown 格式在自己的文档存储库中提供。
- 本指南中包含的一些示例代码可在开发工具包示例代码存储库中提供。

Gitter 上的 JavaScript 开发工具包

还可以在 Gitter 上的 JavaScript SDK 社区中找到有关 SDK for JavaScript 的问题和讨论。

适用于 JavaScript 的 Amazon SDK 的文档历史记录

- 开发工具包版本：请参阅 JavaScript API 参考
- 最近主要文档更新时间：2022 年 3 月 31 日

文档历史记录

下表描述了在 2018 年 5 月后每次发布 适用于 JavaScript 的 Amazon SDK时进行的重要更改。如需获取对此文档的更新的通知，您可以订阅 RSS 源。

变更	说明	日期
现在，所有区域中的所有 Amazon 服务 API 端点都支持 TLS 1.3	更新了支持的 TLS 版本和用于记录 TLS 版本的方法。	2025 年 4 月 10 日
强制实施最低版本 TLS	添加了有关 TLS 1.3 的信息。	2022 年 3 月 31 日
从浏览器查看 Amazon S3 桶中的照片	添加了一个仅查看现有相册中的照片的示例。	2019 年 5 月 13 日
在 Node.js 中设置凭证，新凭证加载选项	添加了有关从 ECS 凭证提供程序或已配置凭证过程加载的凭证的信息。	2019 年 4 月 25 日
使用已配置凭证过程的凭证	添加了有关从已配置凭证过程加载的凭证的信息。	2019 年 4 月 25 日
新的浏览器脚本入门	浏览器脚本入门经过了重新编写，简化了示例以及访问 Amazon Polly 服务来发送文本和返回可在浏览器中播放的合成语音的过程。有关新内容，请参阅浏览器脚本入门。	2018 年 7 月 14 日
新的 Amazon SNS 代码示例	增加了与 Amazon SNS 配合使用的四个新的 Node.js 代码	2018 年 6 月 29 日

	示例。有关示例代码，请参阅 Amazon SNS Examples。	
新的 Node.js 入门	Node.js 入门经过了重新编写，使用更新的示例代码并更详细地介绍了如何创建 `package.json` 文件以及 Node.js 代码本身。有关新内容，请参阅 Node.js 入门。	2018 年 4 月 6 日

早期更新

下表列出了 2018 年 6 月之前每次发布 适用于 JavaScript 的 Amazon SDK 时进行的重要更改。

更改	描述	日期
新 AWS Elemental MediaConvert 代码示例	增加了三个用于处理 AWS Elemental MediaConvert 的新 Node.js 代码示例。有关示例代码，请参阅AWS Elemental MediaConvert 示例。	2018 年 5 月 21 日
新的“在 GitHub 上编辑”按钮	每个主题的标题现在提供了一个按钮，可以将您转至 GitHub 上相同主题的 Markdown 版本，这样您可以提供编辑来改进指南的准确性和完整性。	2018 年 2 月 21 日
自定义终端节点上的新主题	增加了有关格式以及使用自定义终端节点进行 API 调用的信息。请参阅指定自定义终端节点。	2018 年 2 月 20 日
GitHub 上的 SDK for JavaScript 开发人员指南	SDK for JavaScript 开发人员指南以 Markdown 格式在自己的文档存储库中提供。您可以发布希望指南解决的问题，或	2018 年 2 月 16 日

更改	描述	日期
	者提交拉取请求以提交建议的更改。	
新的 Amazon DynamoDB 代码示例	增加了使用文档客户端来更新 DynamoDB 表的新 Node.js 代码示例。有关示例代码，请参阅使用 DynamoDB 文档客户端。	2018 年 2 月 14 日
有关开发工具包日志记录的新主题	增加了描述如何记录通过 SDK for JavaScript 进行的 API 调用的主题，其中包括有关使用第三方日志记录程序的信息。请参阅记录适用于 JavaScript 的 Amazon SDK调用。	2018 年 2 月 5 日
更新了有关区域设置的主题	此主题描述对“如何设置用于开发工具包的区域”进行的更新和扩展，包括有关设置区域的优先顺序的信息。请参阅设置 Amazon 区域。	2017 年 12 月 12 日
新的 Amazon SES 代码示例	SDK 代码示例部分进行了更新，包括了五个与 Amazon SES 配合使用的新示例。有关这些代码示例的更多信息，请参阅Amazon Simple Email Service 示例。	2017 年 11 月 9 日

更改	描述	日期
可用性改进	根据最近的可用性测试，进行了多项更改以改进文档的可用性。 • 更明确地将代码示例标识为定位于浏览器或 Node.js 执行。 • TOC 链接不再立即跳转到其他 Web 内容，包括 API 参考。 • 在“入门”部分中包括了更多链接，提供有关获取 Amazon 凭证的详细信息。 • 提供有关使用开发工具包所需的常见 Node.js 功能的更多信息。有关更多信息，请参阅 Node.js 注意事项。	2017 年 8 月 9 日
新的 DynamoDB 代码示例	SDK 代码示例部分进行了更新，重写了之前的两个示例并增加了三个与 DynamoDB 配合使用的全新示例。有关这些代码示例的更多信息，请参阅 Amazon DynamoDB 示例。	2017 年 6 月 21 日
新的 IAM 代码示例	SDK 代码示例部分进行了更新，包括了五个与 IAM 配合使用的新示例。有关这些代码示例的更多信息，请参阅Amazon IAM 示例。	2016 年 12 月 23 日

更改	描述	日期
新的 CloudWatch 和 Amazon SQS 代码示例	SDK 代码示例部分进行了更新，包括了与 CloudWatch 和 Amazon SQS 配合使用的新示例。有关这些代码示例的更多信息，请参阅 Amazon CloudWatch 示例和 Amazon SQS 示例。	2016 年 12 月 20 日
新的 Amazon EC2 代码示例	SDK 代码示例部分进行了更新，包括了五个与 Amazon EC2 配合使用的新示例。有关这些代码示例的更多信息，请参阅Amazon EC2 示例。	2016 年 12 月 15 日
改进了所支持浏览器列表的可见性	SDK for JavaScript 支持的浏览器列表以前在“先决条件”主题中提供，现在有了自己的主题，在目录中更容易看到。	2016 年 11 月 16 日
新的开发人员指南初次发布	以前的开发者指南现已弃用。新的开发人员指南经过重新整理，可以更方便地查找信息。当 Node.js 或浏览器 JavaScript 场景存在特殊注意事项时，将相应地标识这些情况。本指南还提供了其他以更好方式整理的代码示例，可以更方便更快速地查找这些示例。	2016 年 10 月 28 日