应用与数据集成平台

开发指南

文档版本 01

发布日期 2023-04-23

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 服务集成开发指导

1.1 如何选择认证方式

如果您是 API 提供方

调用接口有如下认证方式，您可以选择其中一种进行认证鉴权。

- **APP认证**（推荐）

  支持简易认证和非简易认证两种。

  - 非简易认证：通过集成应用的Key和Secret认证调用请求。
  - 简易认证：通过AppCode认证调用请求。

  APP认证支持对API进行IP地址方式的访问权限控制。

- **IAM认证**

  支持Token认证和AK/SK认证两种。

  - Token认证：通过Token认证调用请求。Token认证无需使用SDK签名，优先使用Token认证。
  - AK/SK认证：通过AK（Access Key ID）/SK（Secret Access Key）认证调用请求。其签名方式和APP认证相似。

  IAM认证支持对API进行IP地址方式和帐号名方式的访问权限控制。

- **自定义认证**

  如果您希望使用自己的认证方式，您可以创建一个函数后端，将其作为您的认证服务。

  自定义认证支持对API进行IP地址方式的访问权限控制。

- **无认证**

  对API请求不进行认证。

  无认证支持对API进行IP地址方式的访问权限控制。

如果您是 API 调用方

请联系API提供方获取API的调用信息，并确定认证方式，然后参考本指南中相应认证方式调用API。

1.2 APP 认证开发

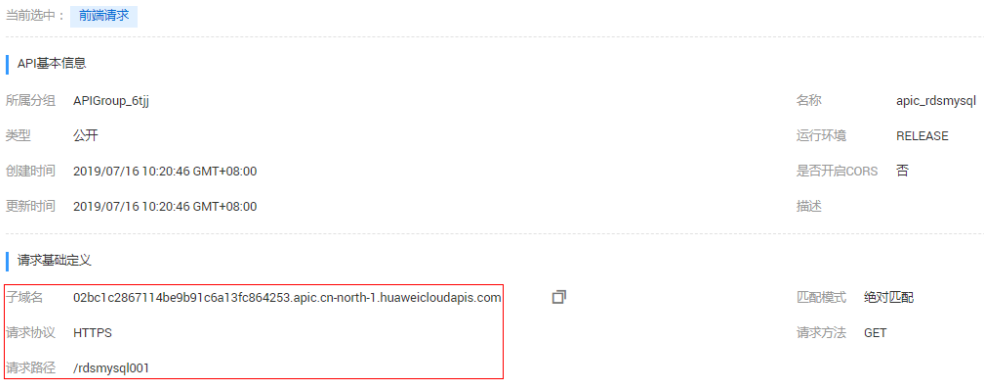
1.2.1 认证前准备

通过SDK访问APP认证前，需要获取如下信息：

- 访问服务前，首先需要得到API的子域名、请求路径和请求协议。

  在“服务集成APIC > API管理 > API列表”中找到具体的API，单击进入API详情，在“调用信息”中单击“前端请求”，查看API的子域名、请求路径和请求协议。

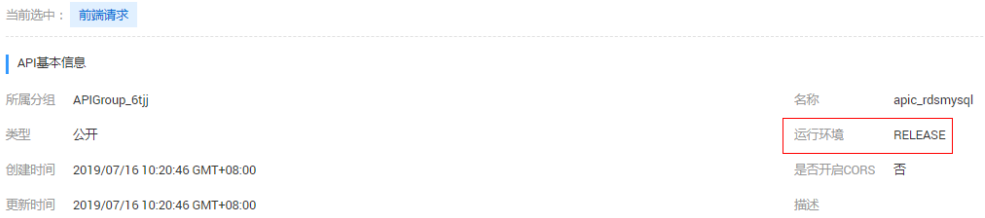
  图 1-1 API 基础定义

- 您必须将API发布到环境才能访问。

  同上，在“前端请求”信息中，可查看API运行环境，如无运行环境，可返回API列表界面，将API发布到指定环境。

  图 1-2 运行环境信息

- 对于APP认证的API，您必须提供有效的AppKey、AppSecret才能够生成认证签名。

  在“集成应用”页面中，单击“创建集成应用”，生成一个APP，单击集成应用名称，可在“基本信息”中查看Key和Secret。在“服务集成 APIC > API管理 > API列表”中把API授权给APP，就可以使用APP对应的Key和Secert访问该API。

  说明

  - Key：APP访问密钥ID。与私有访问密钥关联的唯一标识符；访问密钥ID和私有访问密钥一起使用，对请求进行加密签名。
  - Secret：与访问密钥ID结合使用的密钥，对请求进行加密签名，可标识发送方，并防止请求被修改。

- 发送API请求时，SDK会将当前时间置于HTTP的X-Sdk-Date头，将签名信息置于Authorization头。签名只在一个有限的时间内是有效的，超时即无效。

> **注意**
>
> 客户端须注意本地与NTP服务器的时间同步，避免请求消息头X-Sdk-Date的值出现较大误差。
>
> ROMA Connect除了校验X-Sdk-Date的时间格式外，还会校验该时间值与收到请求的时间差，如果时间差超过15分钟，ROMA Connect将拒绝请求。

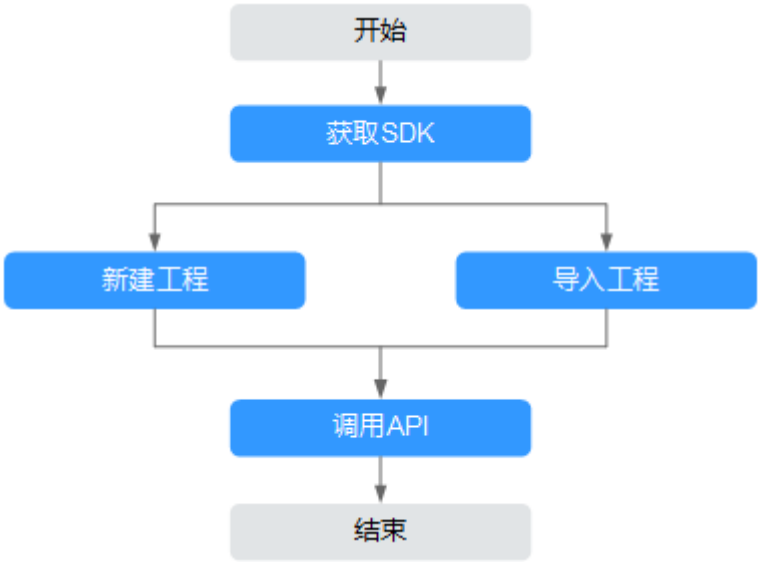
1.2.2 Java SDK 使用说明

操作场景

使用Java语言调用APP认证的API时，您需要先获取SDK，然后新建工程或导入工程，最后参考调用API示例调用API。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

图 1-3 调用流程



前提条件

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 已安装Java Development Kit 1.8.111或以上版本，如果未安装，请至**Oracle官方下载页面**下载。

获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

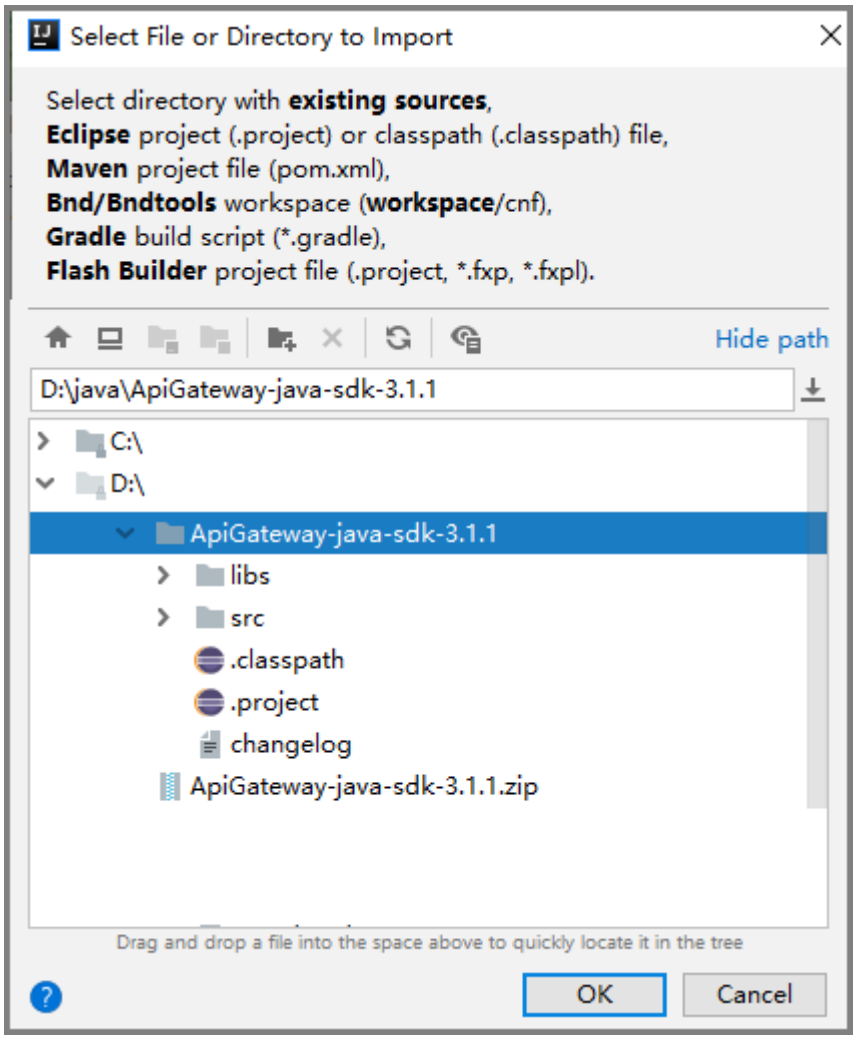
名称	说明
libs\	SDK依赖库。
libs\java-sdk-core-*x.x.x*.jar	SDK包。
src\com\apig\sdk\demo\Main.java	使用SDK签名请求示例代码。
src\com\apig\sdk\demo\OkHttpDemo.java	
src\com\apig\sdk\demo\LargeFileUploadDemo.java	
.classpath	Java工程配置文件。
.project	

导入工程

步骤1 打开IntelliJ IDEA，在菜单栏选择“Import Project”。

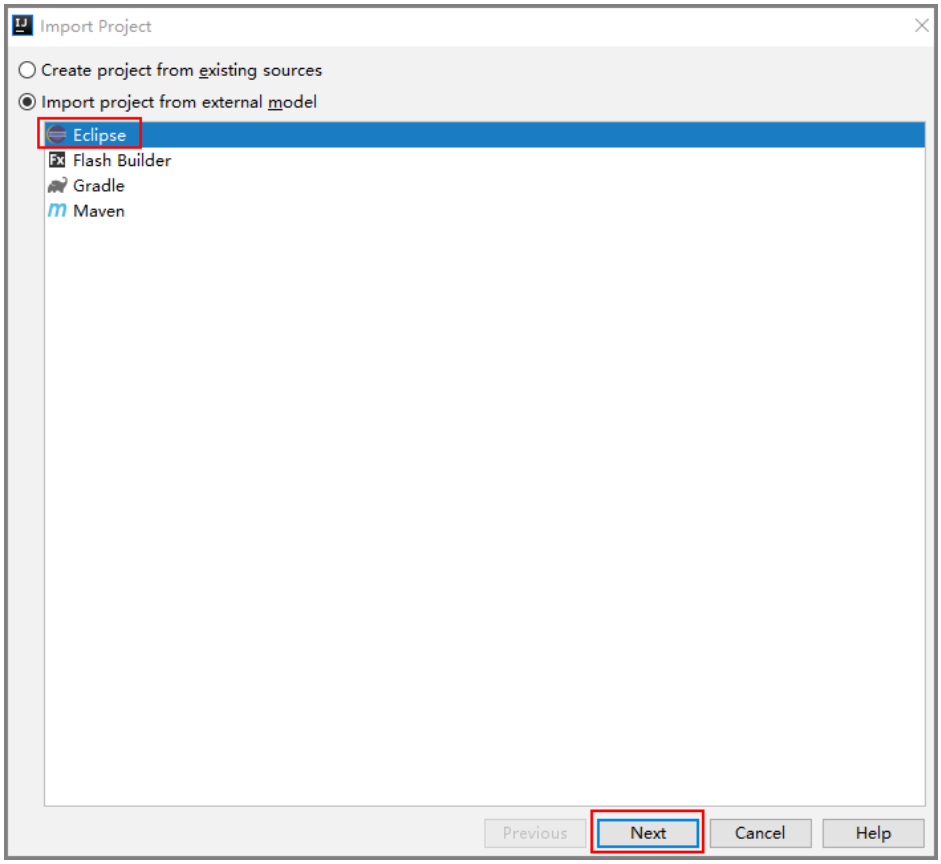
弹出“Select File or Directory to Import”对话框。

步骤2 在弹出的对话框中选择解压后的SDK路径，单击“OK”。



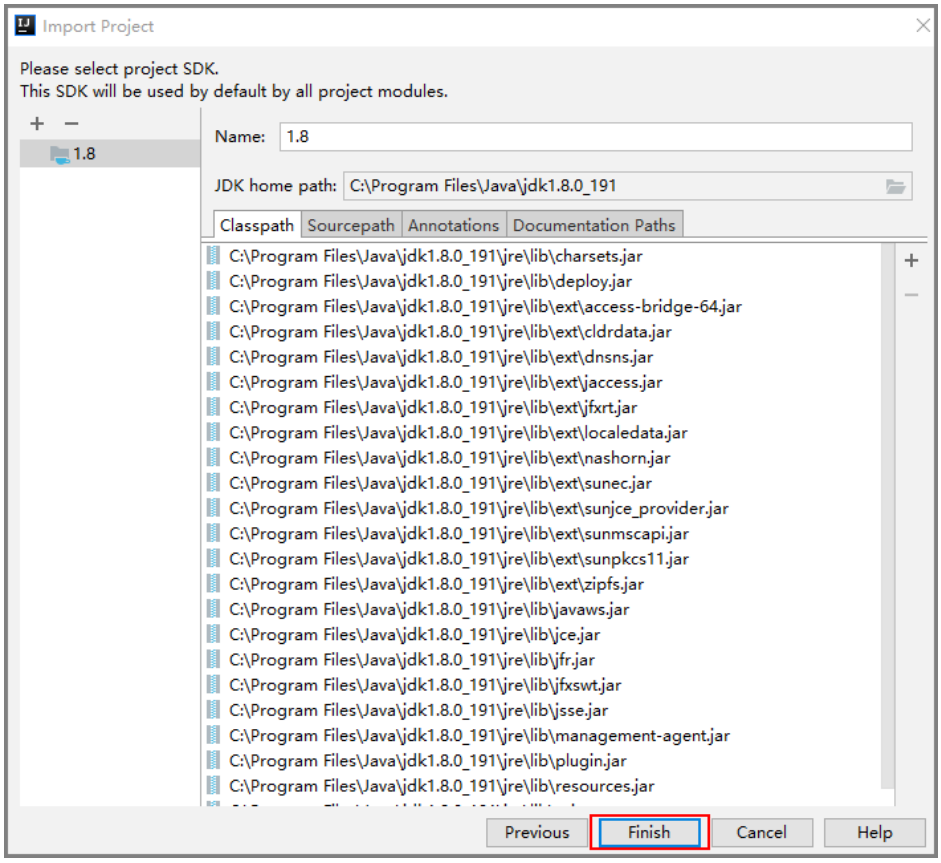
步骤3 “Import project from external model”选择“Eclipse”，单击“Next”，进入下一页后保持默认连续单击“Next”，直到“Please select project SDK”页面。

图 1-4 Import Project



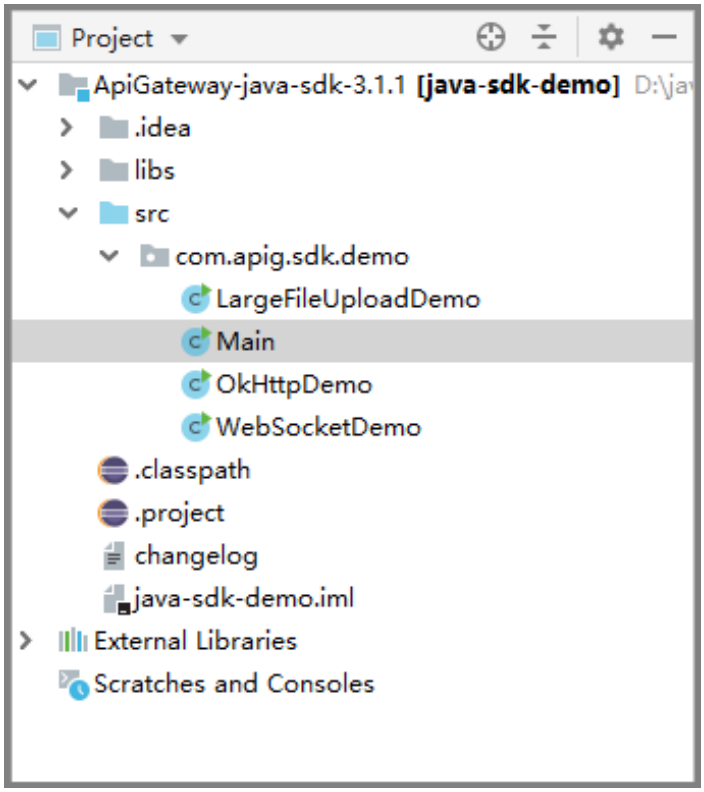
步骤4 单击“Finish”，完成工程导入。

图 1-5 Finish



步骤5 完成导入后，目录结构如下图。

图 1-6 目录结构



----结束

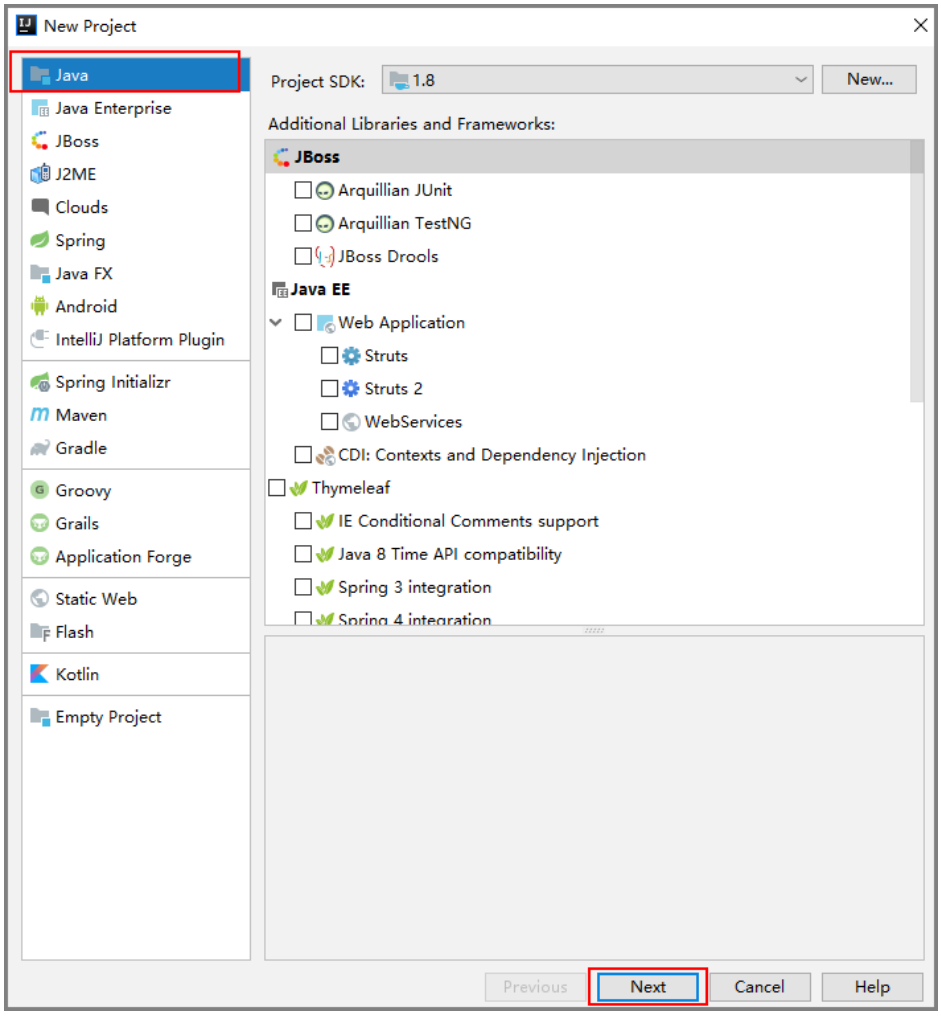
新建工程

步骤1 打开IntelliJ IDEA，在菜单栏选择“Create New Project”。

弹出“New Project”对话框。

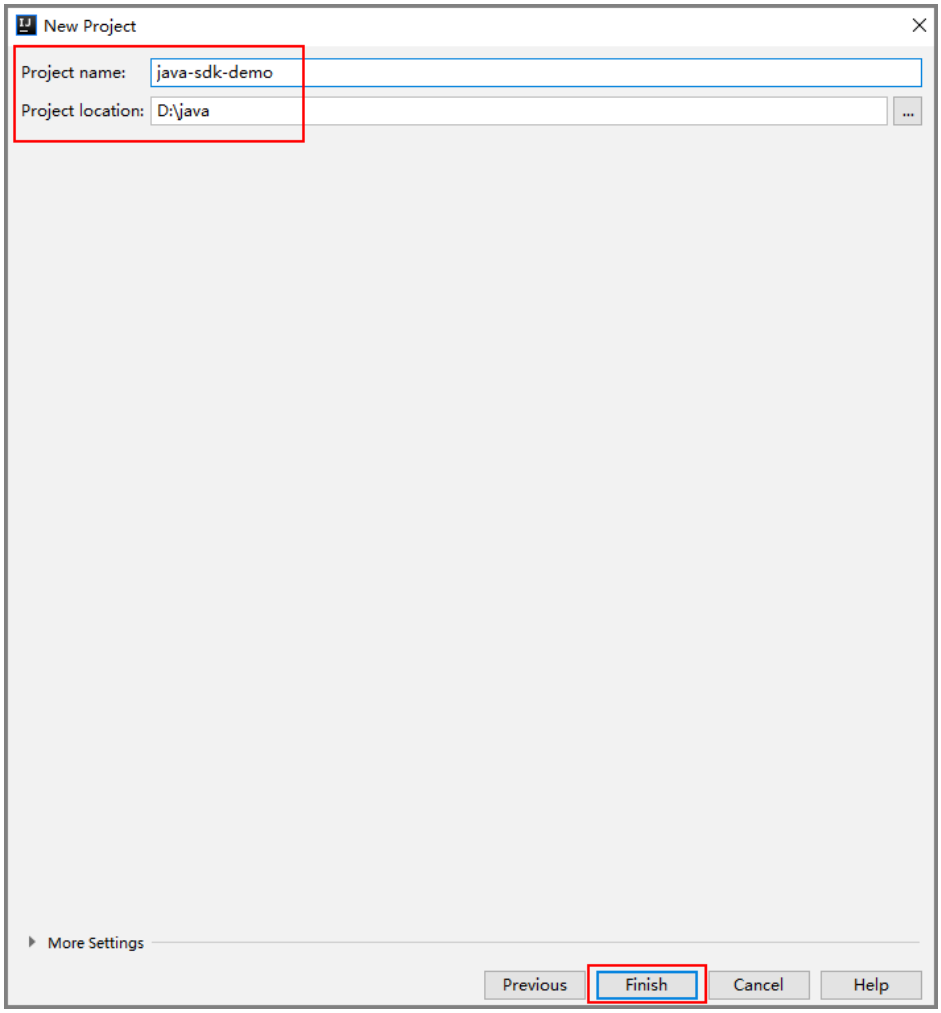
步骤2 在右侧栏中选择“Java”，单击“Next”，进入下一页。

图 1-7 New Project



步骤3 保持默认继续单击“Next”，进入下一页，自定义“Project name”，并选择创建工程所在本地目录“Project location”。

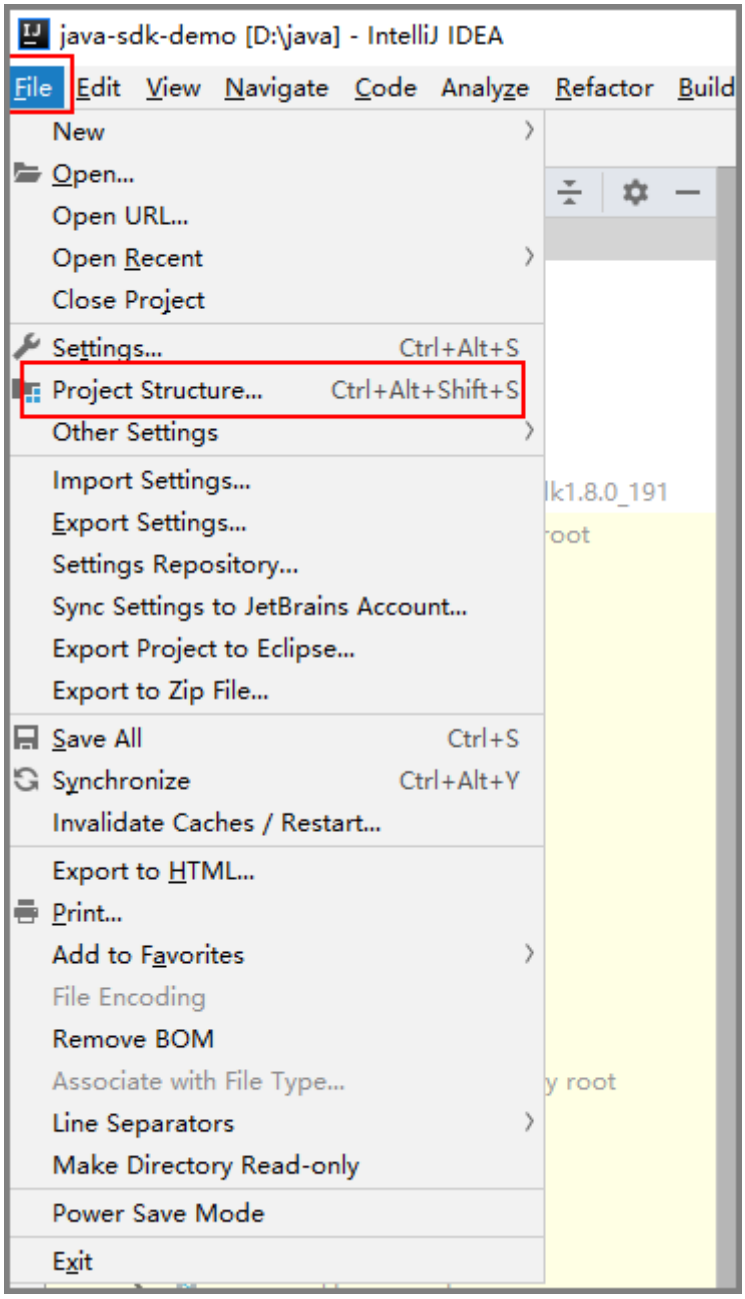
图 1-8 新建工程



步骤4 导入Java SDK的“jar”文件。

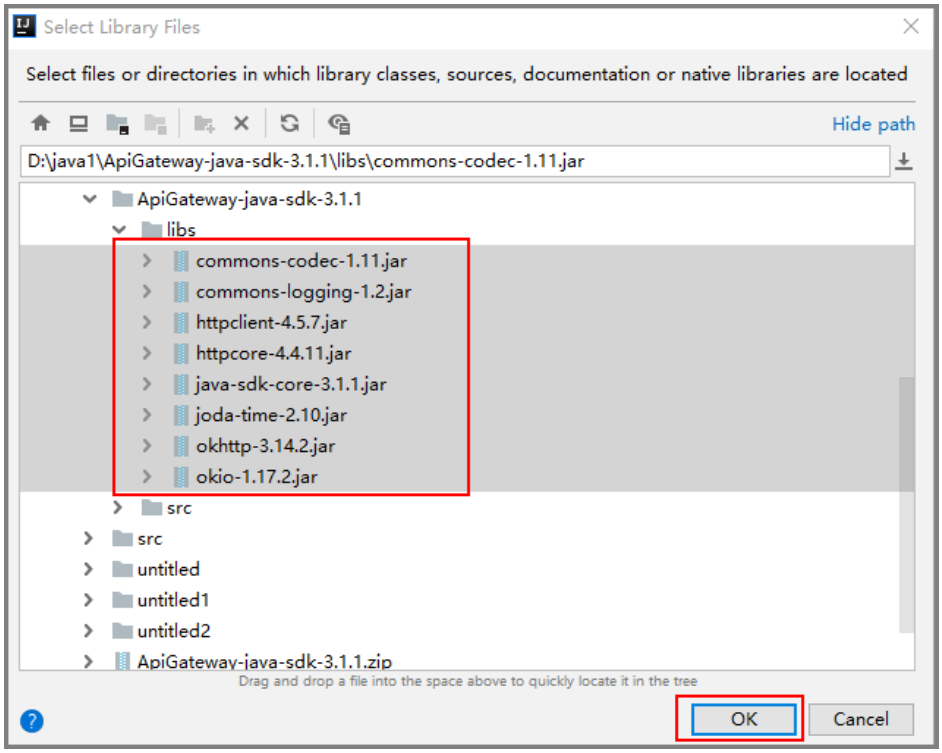
1. 选择“File > Project Structure”，弹出“Project Structure”对话框。

图 1-9 导入 jar 文件



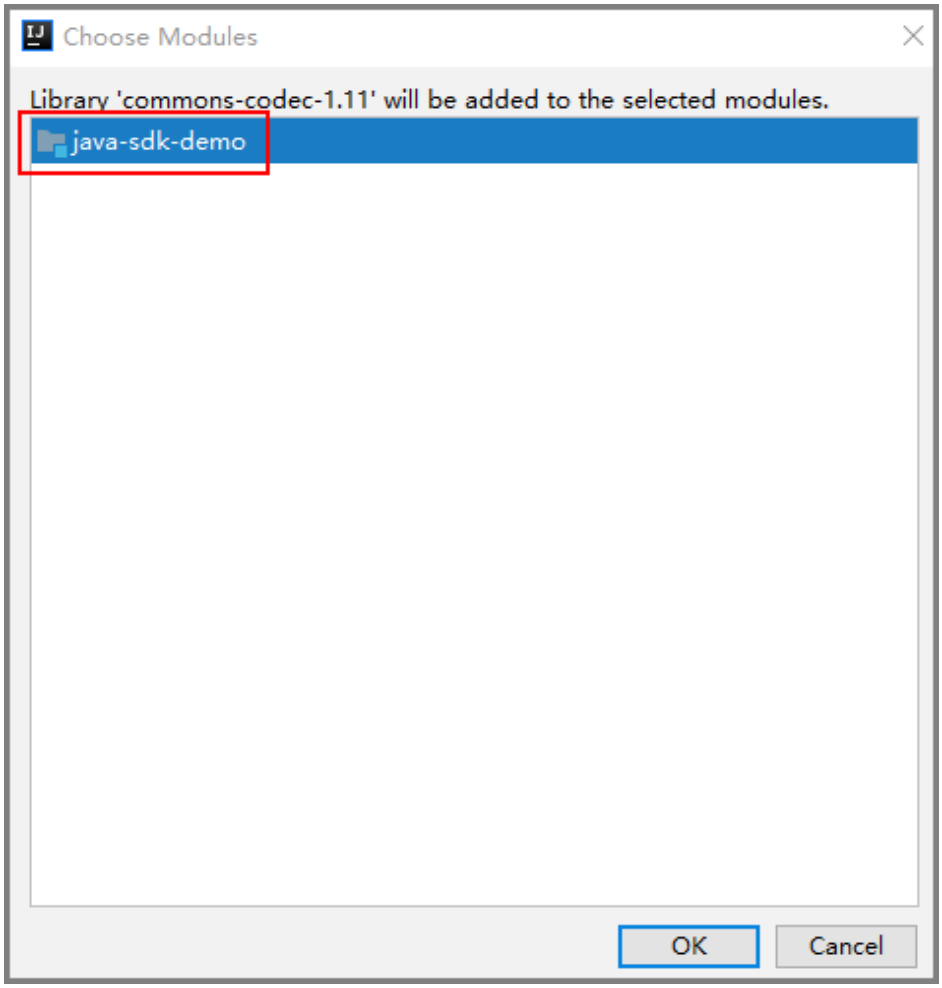
2. 在“Project Structure”对话框中选择“Libraries > + >Java”，界面弹出“Select Library Files”对话框。
3. 选择SDK所在目录中“\libs”目录下所有以“jar”结尾的文件，单击“ok”。

图 1-10 选择 jar 文件



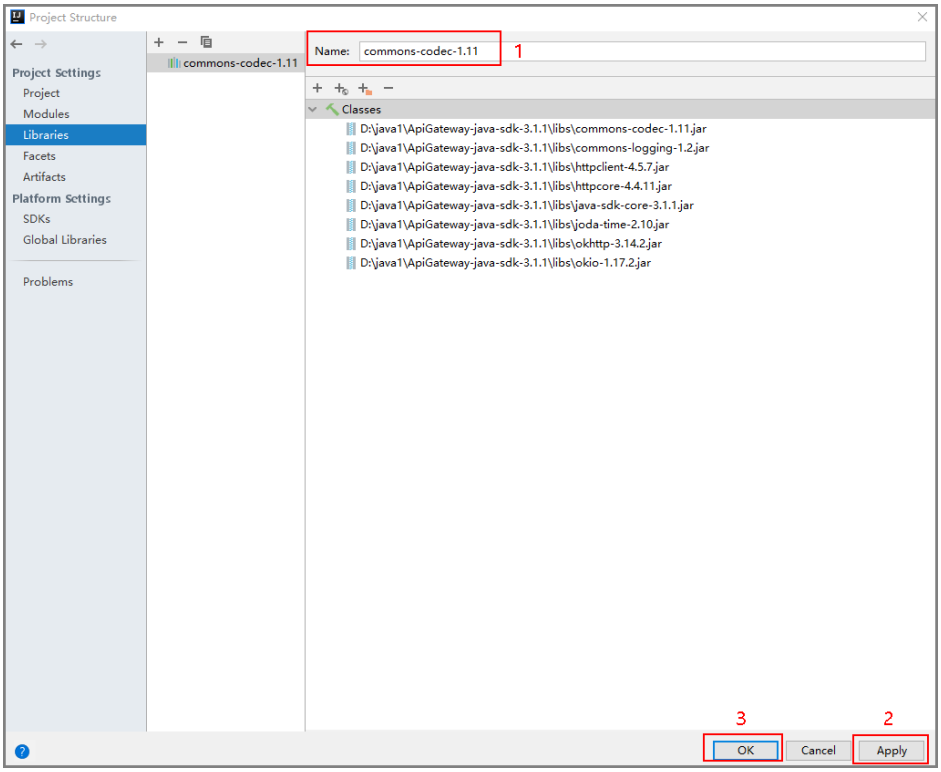
4. 选择步骤3已创建的工程，单击“ok”。

图 1-11 选择工程



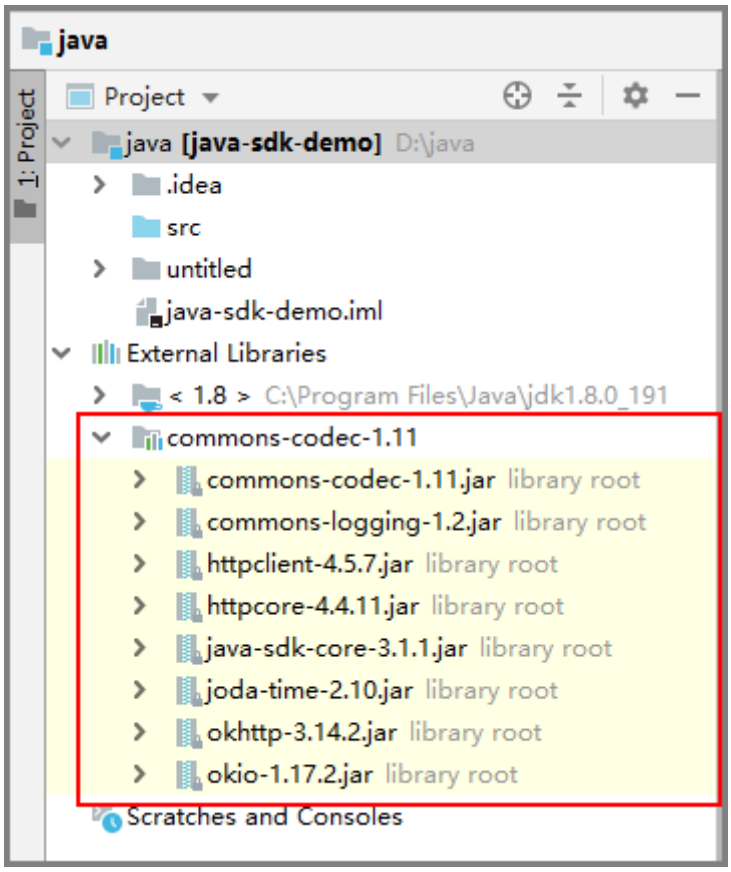
5. 填写jar文件所在目录的名称，单击“Apply > OK”。

图 1-12 jar 文件目录



6. 完成jar文件导入，导入后目录结构如下图。

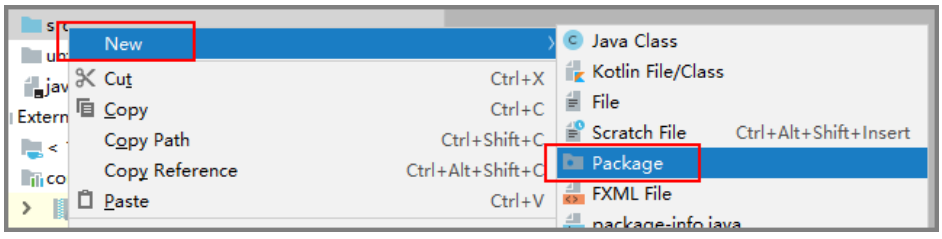
图 1-13 目录结果



步骤5 新建“Package”及“Main”文件。

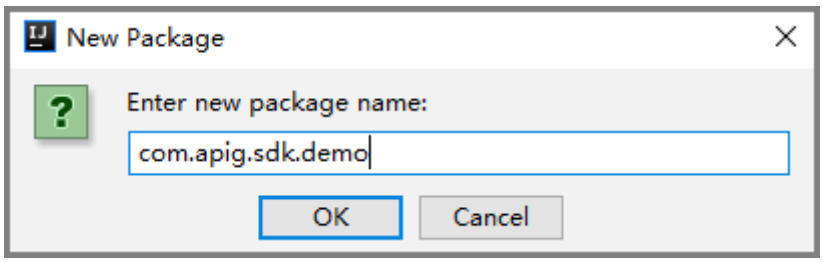
1. 选择“src”，单击鼠标右键，选择“New > Package”。

图 1-14 新建 Package



2. 在“Name”中输入“com.apig.sdk.demo”。

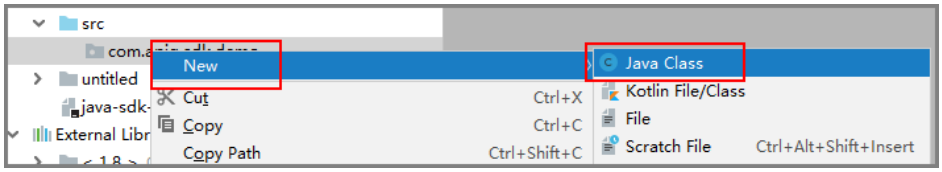
图 1-15 设置 Package 的名称



3. 单击“OK”，完成“Package”的创建。

4. 选择“com.apig.sdk.demo”，单击鼠标右键，选择“New > Java Class”，在“Name”中输入“Main”单击“OK”，完成“Main”文件创建。

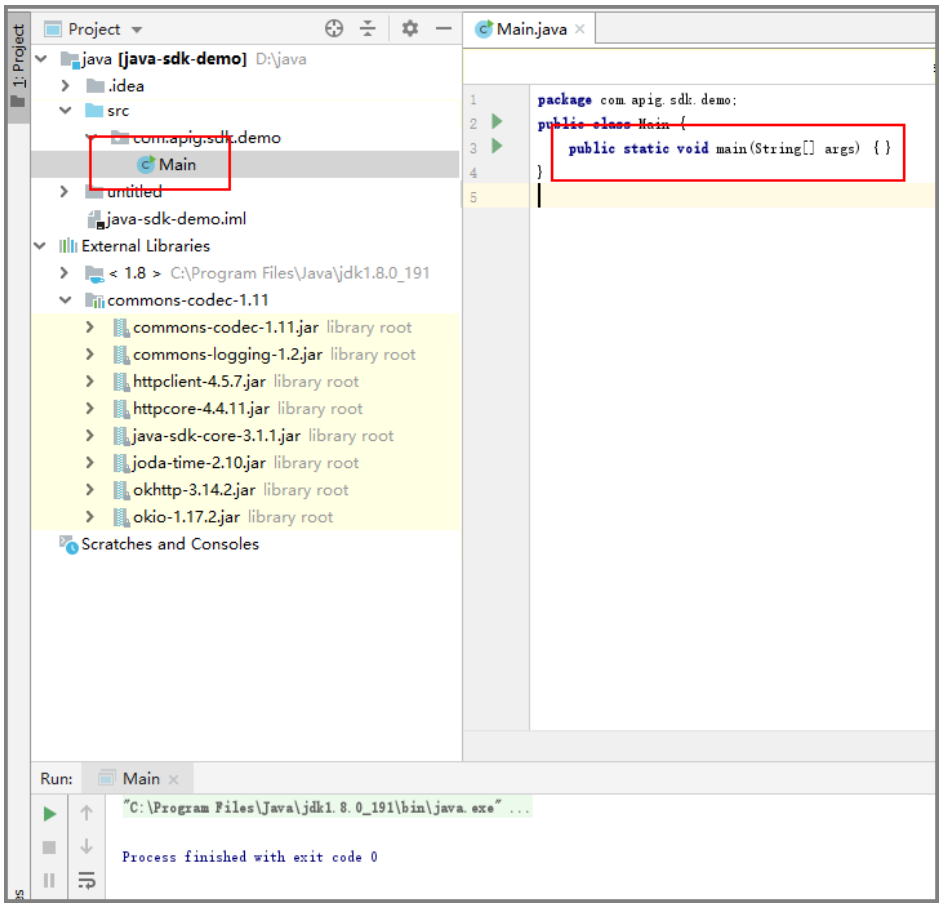
图 1-16 新建 Class



5. 配置Class。

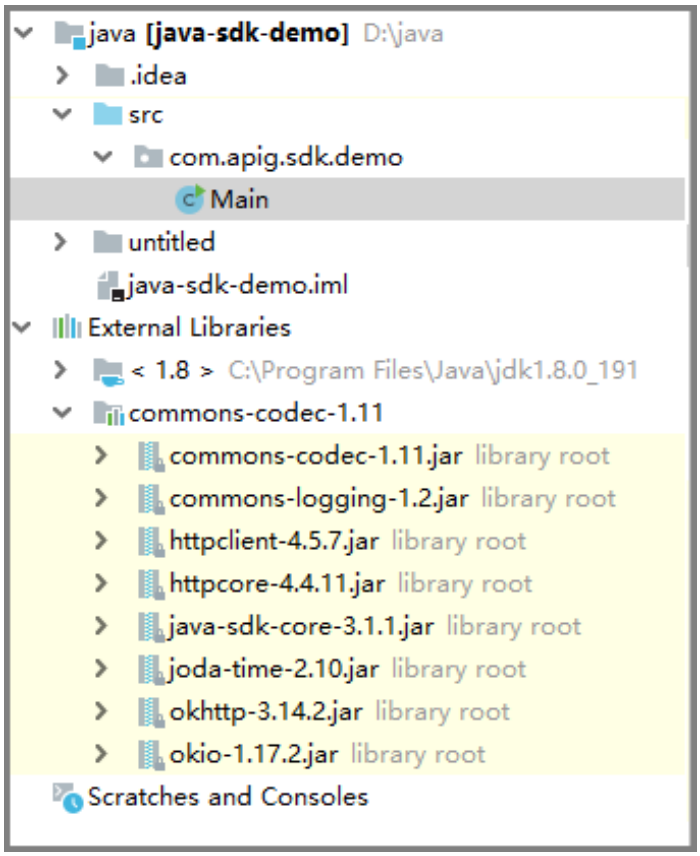
   创建完成后，打开“Main”文件，添加“public static void main(String[] args)”。

图 1-17 设置 Class 的配置



步骤6 完成工程创建后，最终目录结构如下。

图 1-18 新建工程的目录结构



“Main.java”无法直接使用，请根据实际情况参考**调用API示例**输入所需代码。

----结束

调用 API 示例

说明

- 示例演示如何访问发布的API。
- 您需要在APIC的管理控制台自行创建和发布一个API，可以选择Mock模式。
- 示例API的后端为打桩的HTTP服务，此后端返回一个“200”响应码及“Congratulations, sdk demo is running”消息体。

步骤1 在“Main.java”中加入以下引用。

```
import java.io.IOException;
import javax.net.ssl.SSLContext;

import org.apache.http.Header;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpRequestBase;
import org.apache.http.conn.ssl.AllowAllHostnameVerifier;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.conn.ssl.SSLContexts;
import org.apache.http.conn.ssl.TrustSelfSignedStrategy;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;
```

```
import com.cloud.apigateway.sdk.utils.Client;
import com.cloud.apigateway.sdk.utils.Request;
```

步骤2 创建request，过程中需要用到如下参数。

- AppKey：通过**认证前准备**获取。根据实际情况填写，示例代码使用“4f5f626b-073f-402f-a1e0-e52171c6100c”作为样例。
- AppSecret：通过**认证前准备**获取。根据实际情况填写，示例代码使用“******”作为样例。
- Method：请求的方法名。根据API实际情况填写，示例代码使用“POST”作为样例。
- url：请求的url，不包含QueryString及fragment部分。域名部分请使用API所在的分组绑定的您自己的独立域名。示例代码使用“http://serviceEndpoint/java-sdk”作为样例。
- queryString: url携带参数的部分，根据API实际情况填写。支持的字符集为[0-9a-zA-Z./;[]\-=~#%^&_+: "]。示例代码使用“name=value”作为样例。
- header：请求的头域。根据API实际情况填写，不支持中文和下划线。示例代码使用“Content-Type:text/plain”作为样例。如果API发布到非RELEASE环境时，需要增加自定义的环境名称，示例代码使用“x-stage:publish_env_name”作为样例。
- body：请求的正文。根据API实际情况填写，示例代码使用“demo”作为样例。

样例代码如下：

```
    Request request = new Request();
    try
    {
        request.setKey("4f5f626b-073f-402f-a1e0-e52171c6100c"); //创建集成应用时得到
        request.setSecret("*****"); //创建集成应用时得到
        request.setMethod("POST");
        request.setUrl("http://serviceEndpoint/java-sdk");
         //url地址在创建API时得到
         //子域名在创建API分组时得到
        request.addQueryStringParam("name", "value");
        request.addHeader("Content-Type", "text/plain");
        //request.addHeader("x-stage", "publish_env_name"); //如果API发布到非RELEASE环境，需要增加自定义的环境名称
        request.setBody("demo");
    } catch (Exception e)
    {
        e.printStackTrace();
        return;
    }
```

步骤3 对请求进行签名，访问API并打印结果。

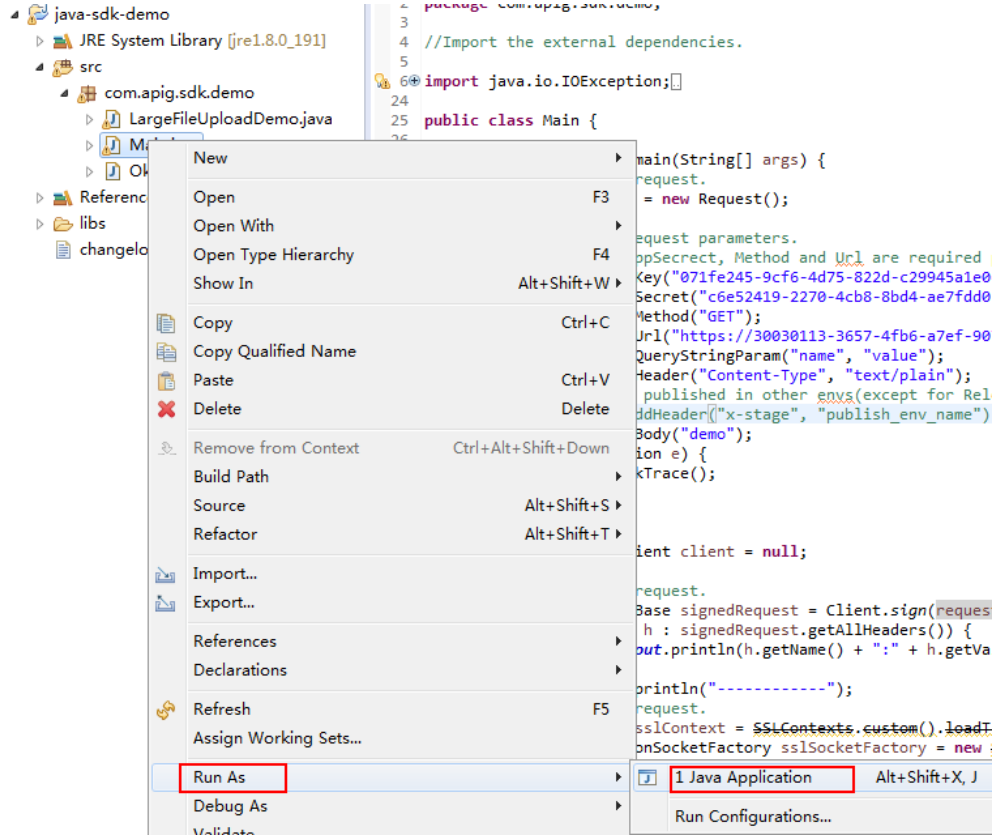
样例代码如下：

```
    CloseableHttpClient client = null;
    try
    {
        HttpRequestBase signedRequest = Client.sign(request);

        //若使用系统分配的子域名访问https请求的API时，需要取消这两行代码的注释，用来忽略证书校验
        // SSLContext sslContext = SSLContexts.custom().loadTrustMaterial(null, new TrustSelfSignedStrategy()).useTLS().build();
        // SSLConnectionSocketFactory sslSocketFactory = new SSLConnectionSocketFactory(sslContext, new AllowAllHostnameVerifier());

        //若使用系统分配的子域名访问https请求的API时，需要在custom()后添加“.setSSLSocketFactory(sslSocketFactory)”，用来忽略证书校验
        client = HttpClients.custom().build();
```

```
        HttpResponse response = client.execute(signedRequest);
        System.out.println(response.getStatusLine().toString());
        Header[] resHeaders = response.getAllHeaders();
        for (Header h : resHeaders)
        {
            System.out.println(h.getName() + ":" + h.getValue());
        }
        HttpEntity resEntity = response.getEntity();
        if (resEntity != null)
        {
            System.out.println(System.getProperty("line.separator") + EntityUtils.toString(resEntity, "UTF-8"));
        }

    } catch (Exception e)
    {
        e.printStackTrace();
    } finally
    {
        try
        {
            if (client != null)
            {
                client.close();
            }
        } catch (IOException e)
        {
            e.printStackTrace();
        }
    }
```

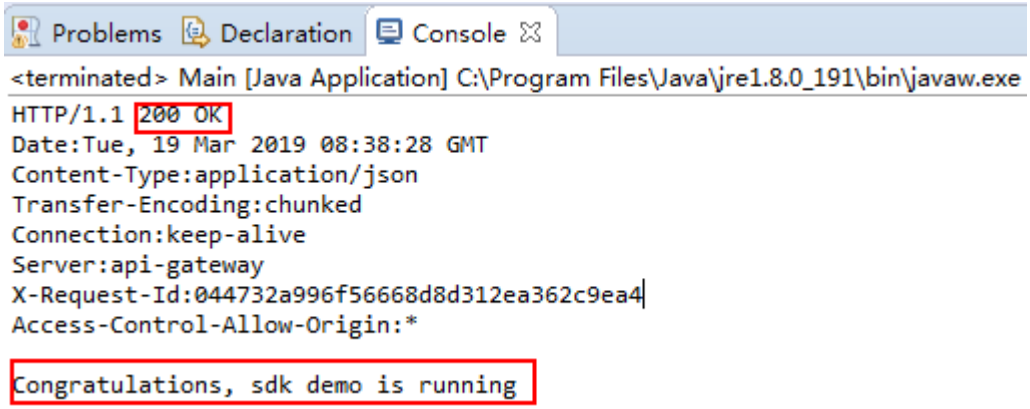
步骤4 选择“Main.java”，单击鼠标右键，选择“Run As > Java Application”，运行工程测试代码。

图 1-19 运行工程测试代码



步骤5 在“Console”页签，查看运行结果。

图 1-20 调用成功后的返回信息



----结束

1.2.3 Go SDK 使用说明

操作场景

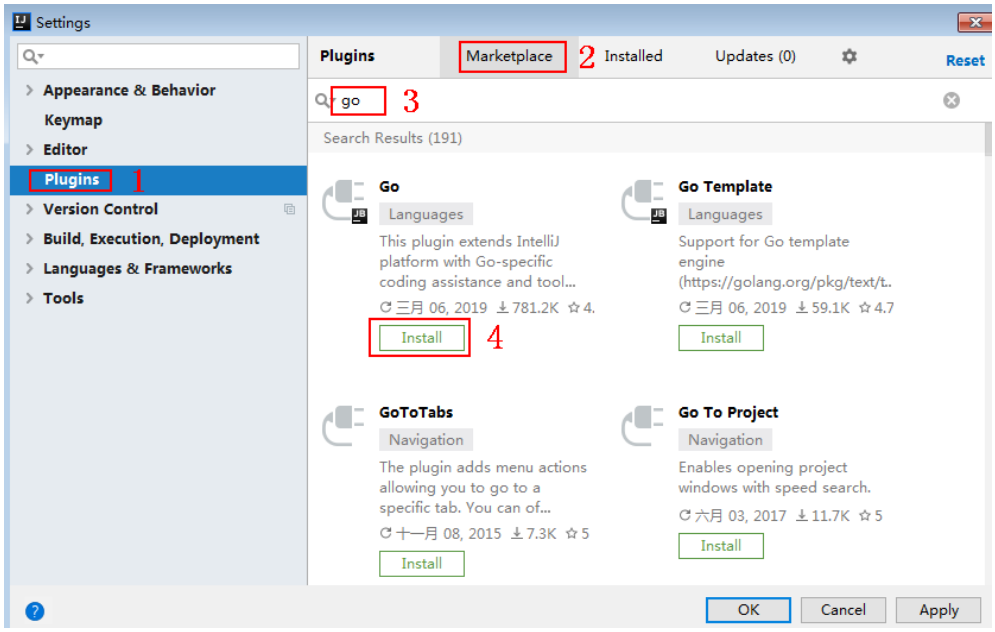
使用Go语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考调用API示例调用API。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

前提条件

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装1.14及以上版本的Go安装包，如果未安装，请至**Go官方下载页面**下载。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 已在IntelliJ IDEA中安装Go插件，如果未安装，请按照**图1-21**所示安装。

图 1-21 安装 Go 插件



获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

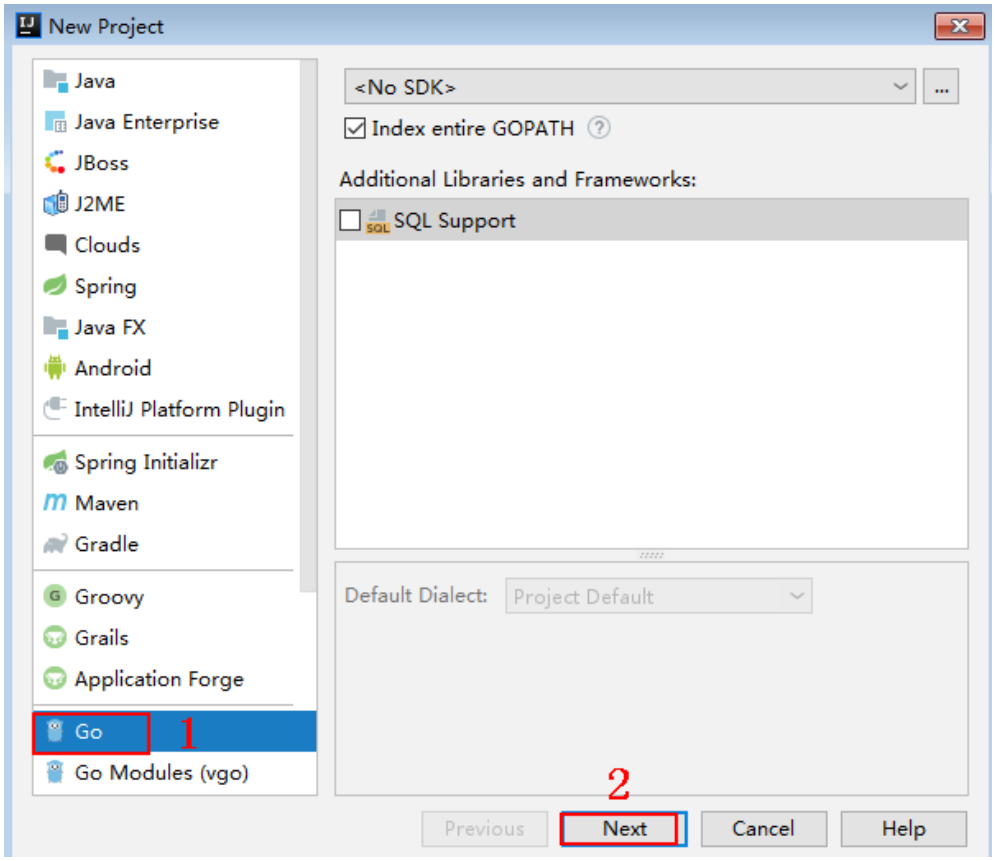
名称	说明
core\escape.go	SDK代码
core\signer.go	
demo.go	示例代码

新建工程

步骤1 打开IntelliJ IDEA，选择菜单“File > New > Project”。

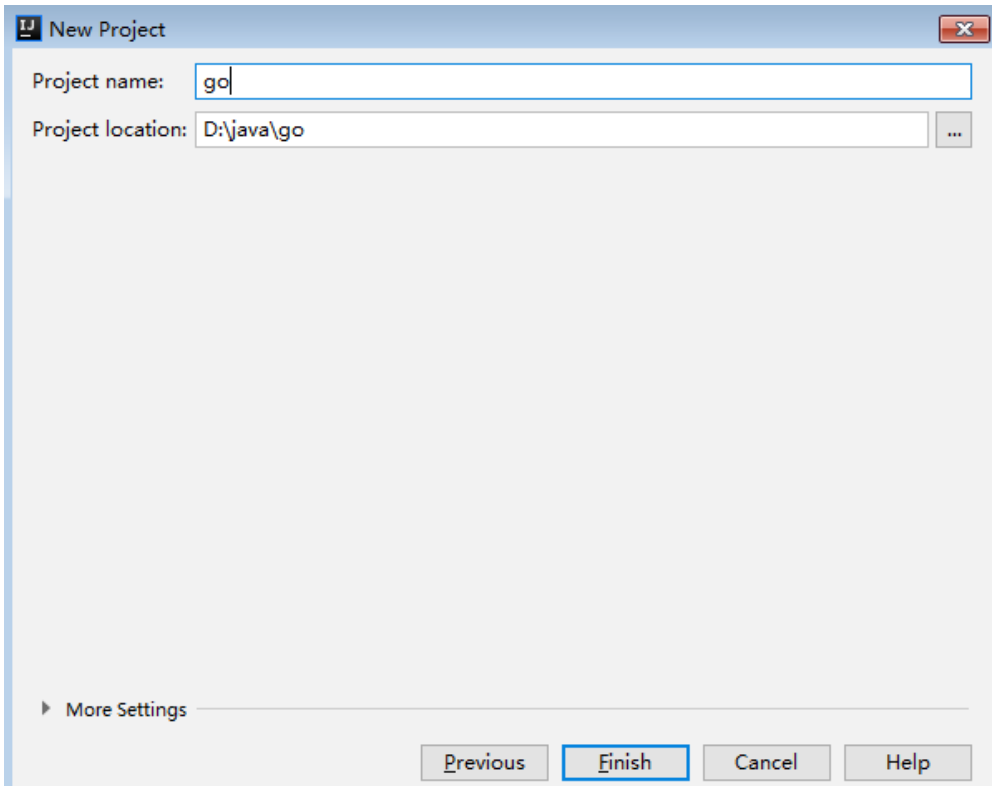
弹出“New Project”对话框，选择“Go”，单击“Next”。

图 1-22 New Project



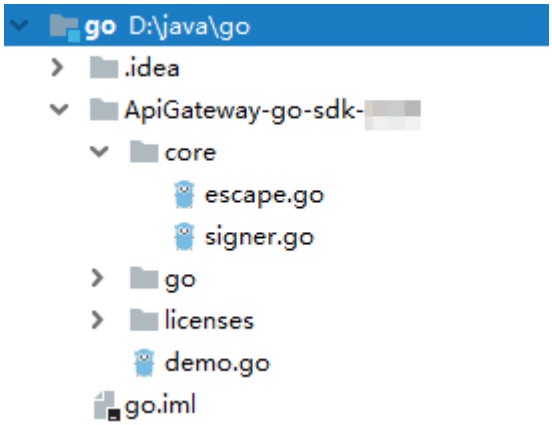
步骤2 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 1-23 选择解压后的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

图 1-24 新建工程的目录结构



“demo.go”为示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**调用API示例**。

----结束

调用 API 示例

步骤1 在工程中引入sdk（signer.go）。

```
import "apig-sdk/go/core"
```

步骤2 生成一个新的Signer，输入集成应用的Key和Secret。

```
s := core.Signer{
        Key: "4f5f626b-073f-402f-a1e0-e52171c6100c",
        Secret: "******",
}
```

步骤3 生成一个新的Request，指定域名、方法名、请求url、query和body。

```
r, _ := http.NewRequest("POST", "http://c967a237-cd6c-470e-906f-
a8655461897e.apigw.exampleRegion.com/api?a=1&b=2",
                        ioutil.NopCloser(bytes.NewBuffer([]byte("foo=bar"))))
```

步骤4 给请求添加x-stage头，内容为环境名。如有需要，添加需要签名的其他头域。

```
r.Header.Add("x-stage", "RELEASE")
```

步骤5 进行签名，执行此函数会在请求中添加用于签名的X-Sdk-Date头和Authorization头。

```
s.Sign(r)
```

步骤6 若使用系统分配的子域名访问https请求的API时，需要忽略证书校验，否则跳过此步。

```
client:=&http.Client{
  Transport:&http.Transport{
    TLSClientConfig:&tls.Config{InsecureSkipVerify:true},
  },
}
```

步骤7 访问API，查看访问结果。

```
resp, err := http.DefaultClient.Do(r)
body, err := ioutil.ReadAll(resp.Body)
```

----结束

1.2.4 Python SDK 使用说明

操作场景

使用Python语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考调用API示例调用API。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

准备环境

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装2.7或3.X版本的Python安装包，如果未安装，请至**Python官方下载页面**下载。

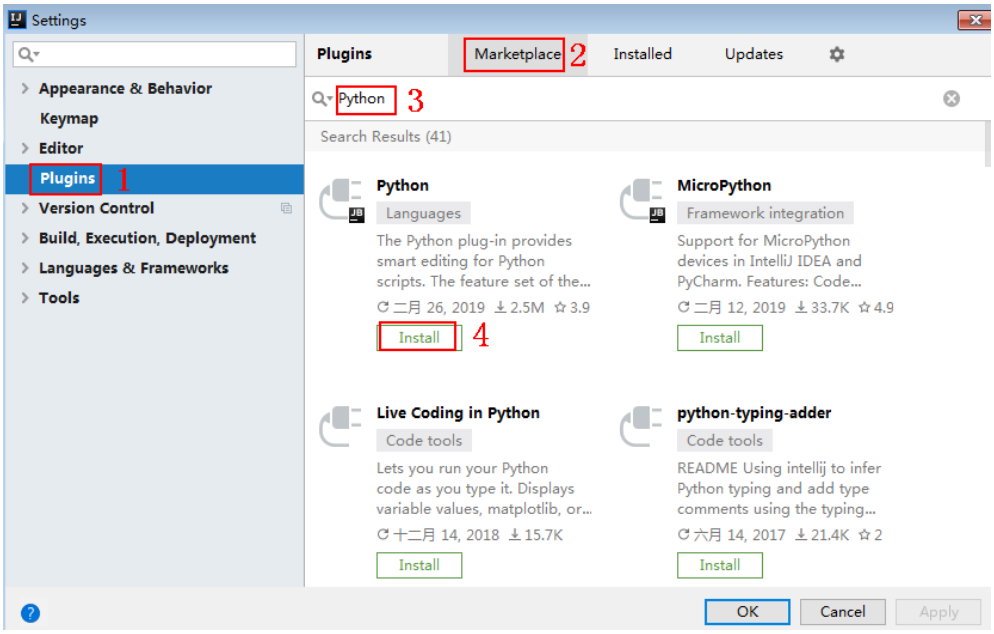
  Python安装完成后，在命令行中使用pip安装“requests”库。

  ```
  pip install requests
  ```

  说明

  如果pip安装requests遇到证书错误，请下载并使用Python执行**此文件**，升级pip，然后再执行以上命令安装。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照**图1-25**所示安装。

图 1-25 安装 Python 插件



获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

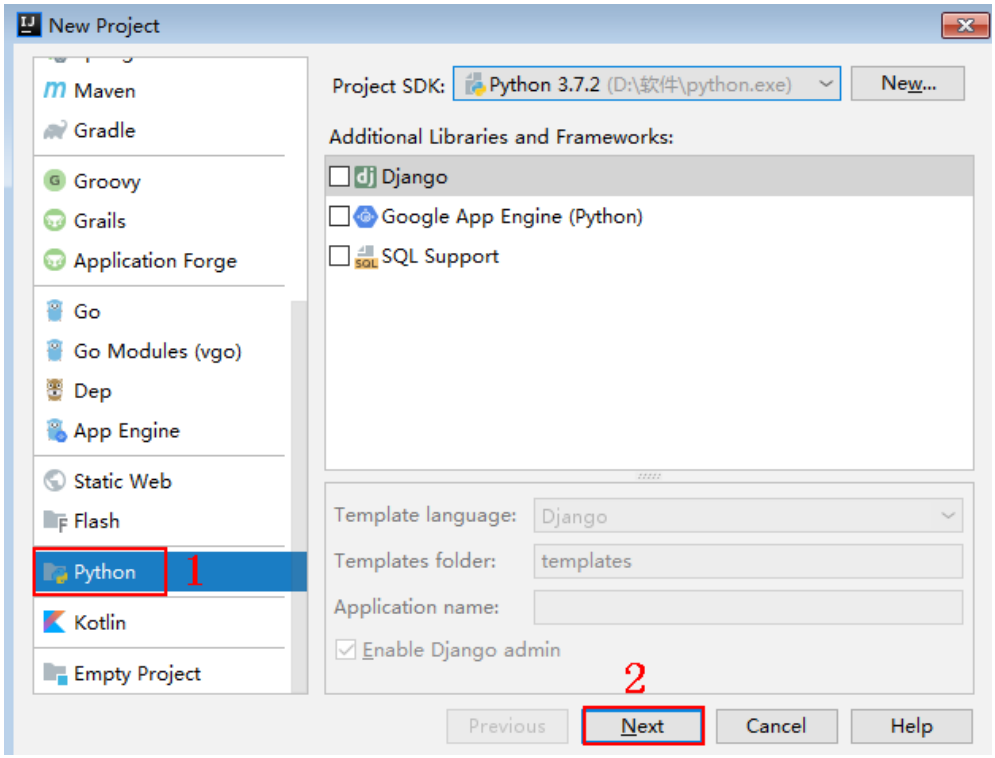
名称	说明
apig_sdk__init__.py	SDK代码
apig_sdk\signer.py	
main.py	示例代码
backend_signature.py	后端签名示例代码
licenses\license-requests	第三方库license文件

新建工程

步骤1 打开IDEA，选择菜单“File > New > Project”。

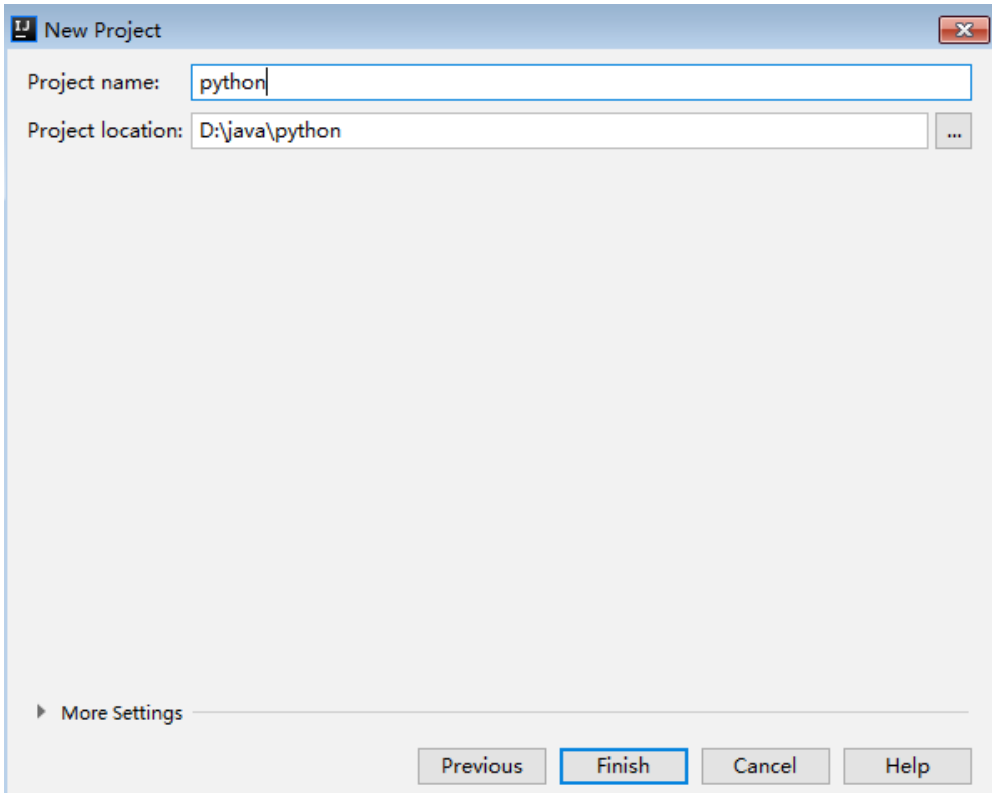
弹出“New Project”对话框，选择“Python”，单击“Next”。

图 1-26 New Project



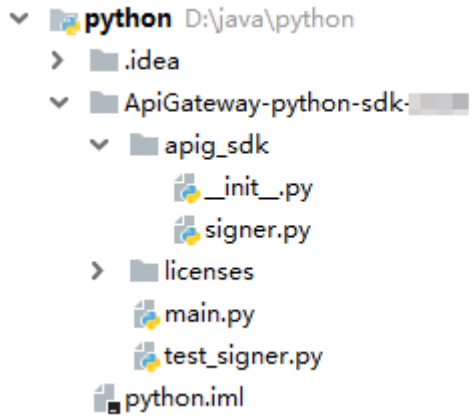
步骤2 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 1-27 选择解压后的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

图 1-28 新建工程的目录结构



“main.py”为示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**调用API示例**。

----结束

调用 API 示例

步骤1 在工程中引入apig_sdk。

```
from apig_sdk import signer
import requests
```

步骤2 生成一个新的Signer，输入集成应用的Key和Secret。

```
sig = signer.Signer()
sig.Key = "4f5f626b-073f-402f-a1e0-e52171c6100c"
sig.Secret = "******"
```

步骤3 生成一个Request对象，指定方法名、请求uri、header和body。

```
r = signer.HttpRequest("POST",
                       "https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1?a=1",
                       {"x-stage": "RELEASE"},
                       "body")
```

步骤4 进行签名，执行此函数会在请求参数中添加用于签名的X-Sdk-Date头和Authorization头。

说明

X-Sdk-Date是一个必须参与签名的请求消息头参数。

```
sig.Sign(r)
```

步骤5 访问API，查看访问结果。

```
//若使用系统分配的子域名访问https请求的API时，需要在data=r.body后添加“,verify=False”，用来忽略证书校验
resp = requests.request(r.method, r.scheme + "://" + r.host + r.uri, headers=r.headers, data=r.body)
print(resp.status_code, resp.reason)
print(resp.content)
```

----结束

1.2.5 C# SDK 使用说明

操作场景

使用C#语言调用APP认证的API时，您需要先获取SDK，然后打开SDK包中的工程文件，最后参考API调用示例调用API。

准备环境

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装2019 version 16.8.4及以上版本的Visual Studio，如果未安装，请至**Visual Studio官方网站**下载。

获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

名称	说明
apigateway-signature\Signer.cs	SDK代码
apigateway-signature\HttpEncoder.cs	
sdk-request\Program.cs	签名请求示例代码
backend-signature\	后端签名示例工程
csharp.sln	工程文件
licenses\license-referencesource	第三方库license文件

打开工程

双击SDK包中的“csharp.sln”文件，打开工程。工程中包含如下3个项目：

- apigateway-signature：实现签名算法的共享库，可用于.Net Framework与.Net Core项目。
- backend-signature：后端服务签名示例。
- sdk-request：签名算法的调用示例，请根据实际情况修改参数后使用。具体代码说明请参考**调用API示例**。

调用 API 示例

步骤1 在工程中引入sdk。

```
using APIGATEWAY_SDK;
```

步骤2 生成一个新的Signer， 输入集成应用的Key和Secret。

```
Signer signer = new Signer();
signer.Key = "4f5f626b-073f-402f-a1e0-e52171c6100c";
signer.Secret = "******";
```

步骤3 生成一个HttpRequest对象，指定域方法名、请求url和body。

```
HttpRequest r = new HttpRequest("POST",
                new Uri("https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1?query=value"));
r.body = "{\"a\":1}";
```

步骤4 给请求添加x-stage头，内容为环境名。如有需要，添加需要签名的其他头域。

```
r.headers.Add("x-stage", "RELEASE");
```

步骤5 进行签名，执行此函数会生成一个新的HttpWebRequest，并在请求参数中添加用于签名的X-Sdk-Date头和Authorization头。

```
HttpWebRequest req = signer.Sign(r);
```

步骤6 若使用系统分配的子域名访问https请求的API，需要忽略证书校验，否则跳过此步。

```
System.Net.ServicePointManager.ServerCertificateValidationCallback = new
System.Net.Security.RemoteCertificateValidationCallback(delegate { return true; });
```

步骤7 访问API，查看访问结果。

```
var writer = new StreamWriter(req.GetRequestStream());
writer.Write(r.body);
writer.Flush();
HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
var reader = newStreamReader(resp.GetResponseStream());
Console.WriteLine(reader.ReadToEnd());
```

----结束

1.2.6 JavaScript SDK 使用说明

操作场景

使用JavaScript语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考API调用示例调用API。

JavaScript SDK支持Node.js和浏览器两种运行环境。

本章节以IntelliJ IDEA 2018.3.5版本、搭建Node.js开发环境为例介绍。

准备环境

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 浏览器版本为chrome 89.0 或以上版本。
- 获取并安装15.10.0及以上版本的Nodejs安装包，如果未安装，请至**Nodejs官方下载页面**下载。

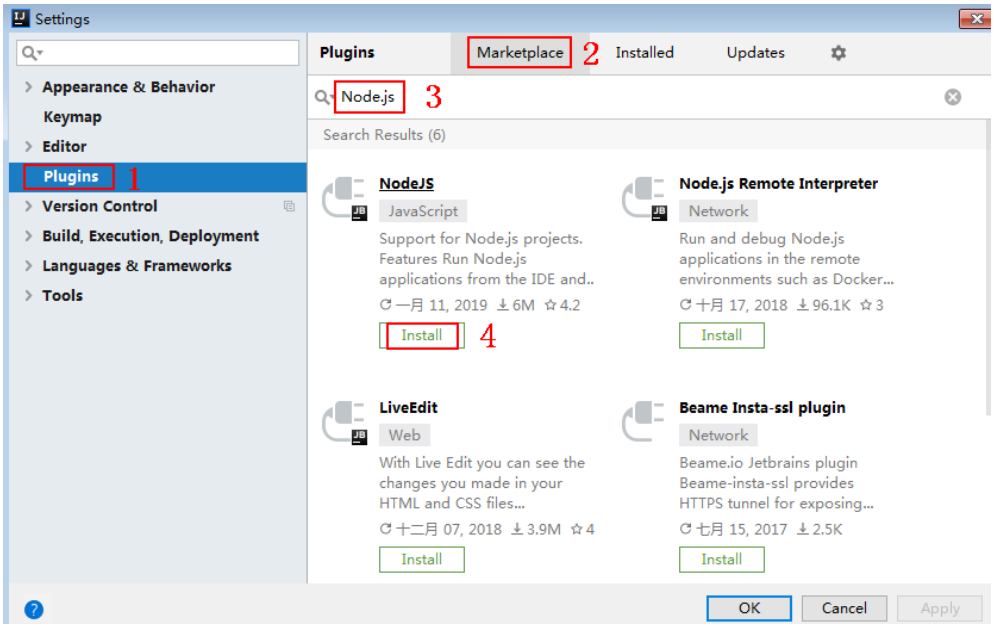
  Nodejs安装后，在命令行中，用npm安装“moment”和“moment-timezone”模块。

  ```
  npm install moment --save
  npm install moment-timezone --save
  ```

- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。

- 已在IntelliJ IDEA中安装NodeJS插件，如果未安装，请按照图1-29所示安装。

图 1-29 安装 NodeJS 插件



获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

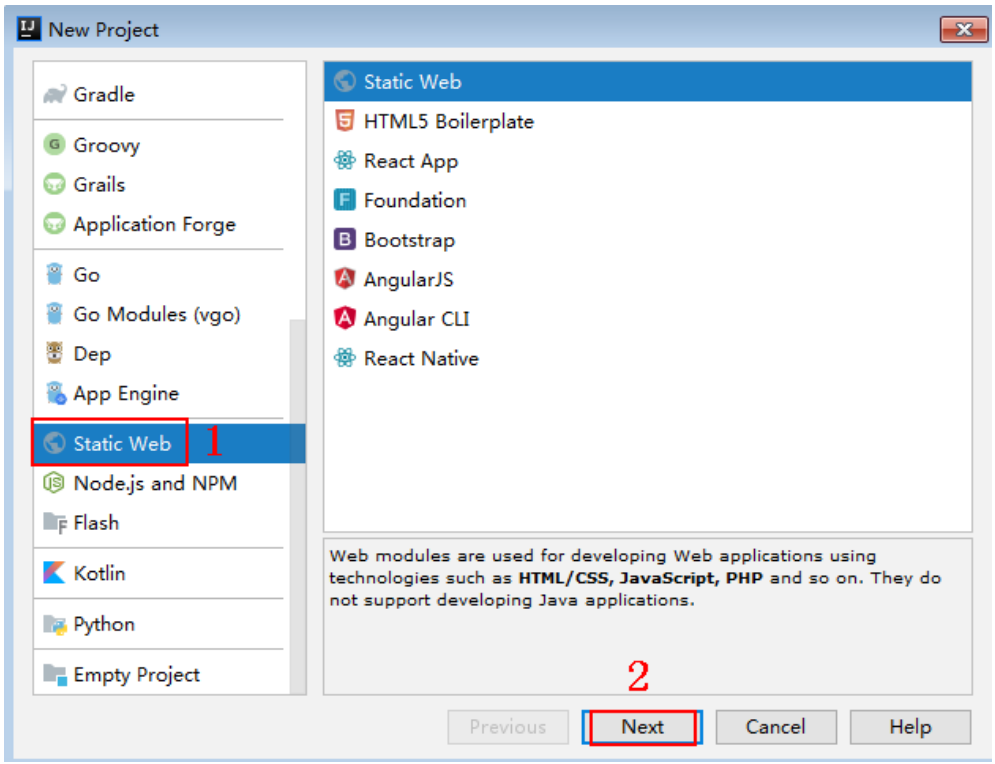
名称	说明
signer.js	SDK代码
node_demo.js	Nodejs示例代码
demo.html	浏览器示例代码
demo_require.html	浏览器示例代码（使用require加载）
test.js	测试用例
js\hmac-sha256.js	依赖库
licenses\license-crypto-js	第三方库license文件
licenses\license-node	

创建工程

步骤1 打开IntelliJ IDEA，选择菜单“File > New > Project”。

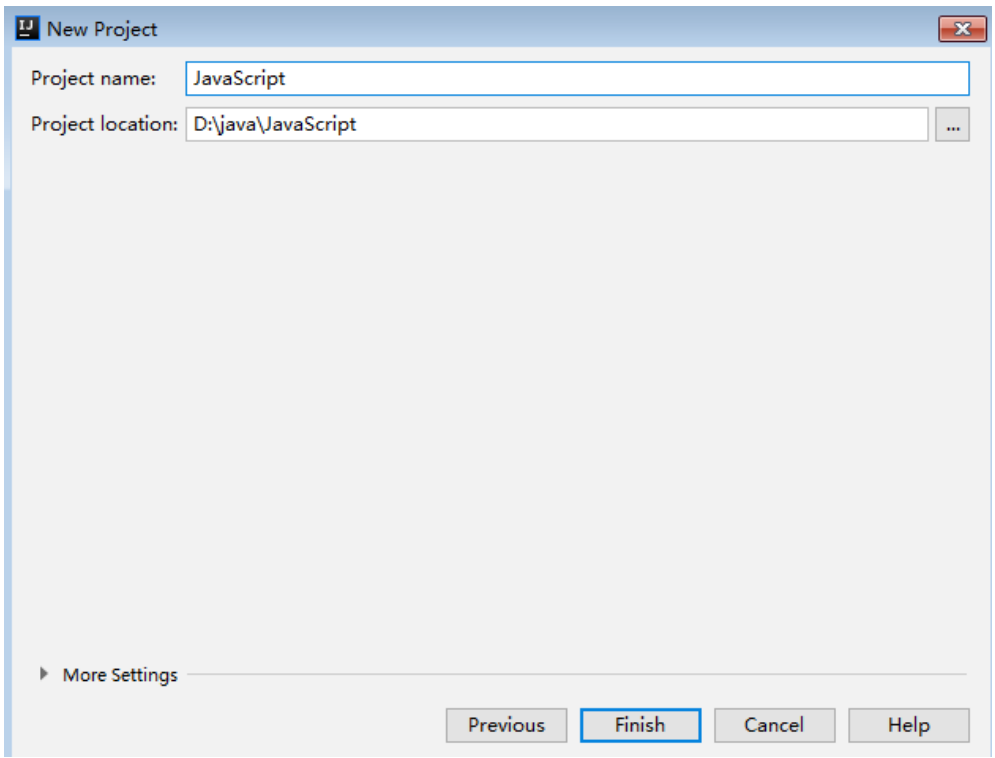
弹出“New Project”对话框。选择“Static Web”，单击“Next”。

图 1-30 New Project



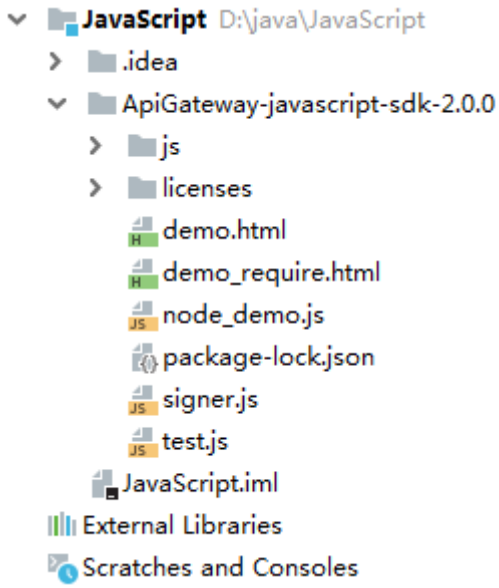
步骤2 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 1-31 选择解压后的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

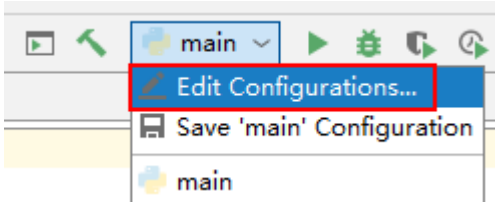
图 1-32 新建工程的目录结构



- node_demo.js：Nodejs示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**调用API（Node.js）示例**。
- demo.html：浏览器示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**调用API（浏览器）示例**。

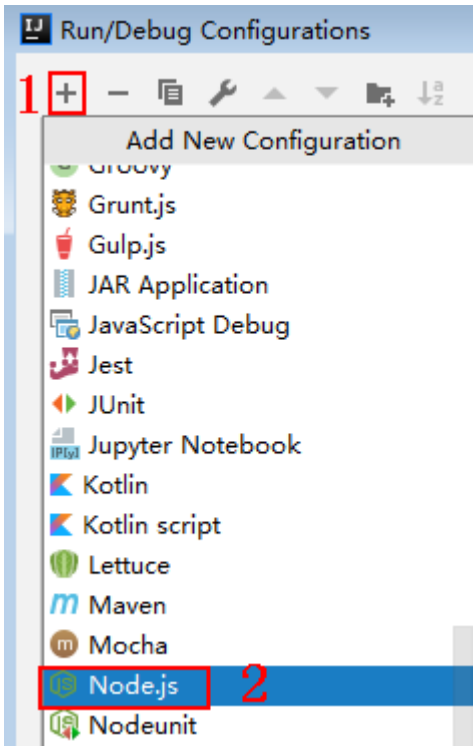
步骤4 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 1-33 Edit Configurations



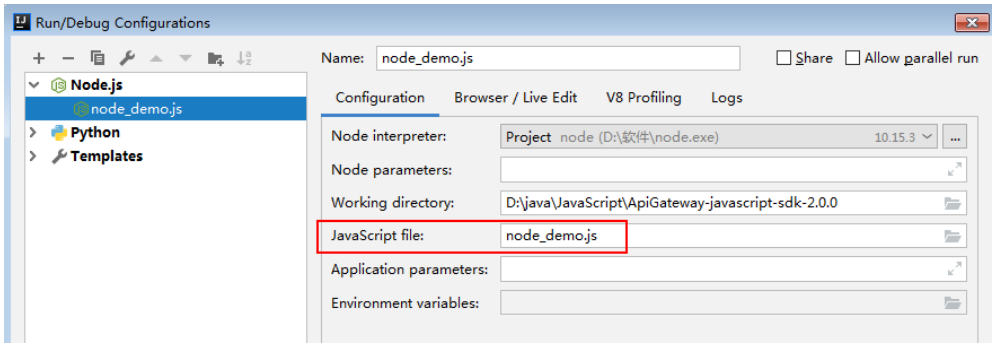
步骤5 单击“+”，选择“Node.js”。

图 1-34 选择 Node.js



步骤6 “JavaScript file”选择“node_demo.js”，单击“OK”，完成配置。

图 1-35 选择 node_demo.js



----结束

调用 API（Node.js）示例

步骤1 在工程中引入signer.js。

```
var signer = require('./signer')
var http = require('http')
```

步骤2 生成一个新的Signer，填入AppKey和AppSecret。

```
var sig = new signer.Signer()
sig.Key = "4f5f626b-073f-402f-a1e0-e52171c6100c"
sig.Secret = "******"
```

步骤3 生成一个Request对象，指定方法名、请求uri和body。

```
var r = new signer.HttpRequest("POST", "c967a237-cd6c-470e-906f-
a8655461897e.apigw.exampleRegion.com/app1?a=1");
r.body = '{"a":1}'
```

步骤4 给请求添加x-stage头，内容为环境名。如有需要，添加需要签名的其他头域。

```
r.headers = { "x-stage":"RELEASE" }
```

步骤5 进行签名，执行此函数会生成请求参数，用于创建http(s)请求，请求参数中添加了用于签名的X-Sdk-Date头和Authorization头。

```
var opts = sig.Sign(r)
```

步骤6 访问API，查看访问结果。如果使用https访问，则将“http.request”改为“https.request”。

```
var req=http.request(opts, function(res){
        console.log(res.statusCode)
        res.on("data",    function(chunk){
        console.log(chunk.toString())
    })
})
req.on("error",function(err){
    console.log(err.message)
})
req.write(r.body)
req.end()
```

----结束

调用 API（浏览器）示例

使用浏览器访问API，需要注册支持OPTIONS方法的API，具体步骤请参见创建OPTIONS方式的API，且返回头中带有“Access-Control-Allow-*”相关访问控制头域，可在创建API时通过开启CORS来添加这些头域。

步骤1 在html中引入signer.js及依赖。

```
<script src="js/hmac-sha256.js"></script>
<script src="js/moment.min.js"></script>
<script src="js/moment-timezone-with-data.min.js"></script>
<script src='signer.js'></script>
```

步骤2 进行签名和访问。

```
var sig = new signer.Signer()
sig.Key = "4f5f626b-073f-402f-a1e0-e52171c6100c"
sig.Secret = "******"
var r= new signer.HttpRequest()
r.host = "c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com"
r.method = "POST"
r.uri = "/app1"
r.body = '{"a":1}'
r.query = { "a":"1","b":"2" }
r.headers = { "Content-Type":"application/json" }
var opts = sig.Sign(r)
var scheme = "https"
$.ajax({
    type: opts.method,
    data: req.body,
    processData: false,
    url: scheme + "://" + opts.hostname + opts.path,
    headers: opts.headers,
    success: function (data) {
        $('#status').html('200')
        $('#recv').html(data)
    },
    error: function (resp) {
        if (resp.readyState === 4) {
            $('#status').html(resp.status)
```

```
            $('#recv').html(resp.responseText)
        } else {
            $('#status').html(resp.state())
        }
    },
    timeout: 1000
});
```

----结束

1.2.7 PHP SDK 使用说明

操作场景

使用PHP语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考API调用示例调用API。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

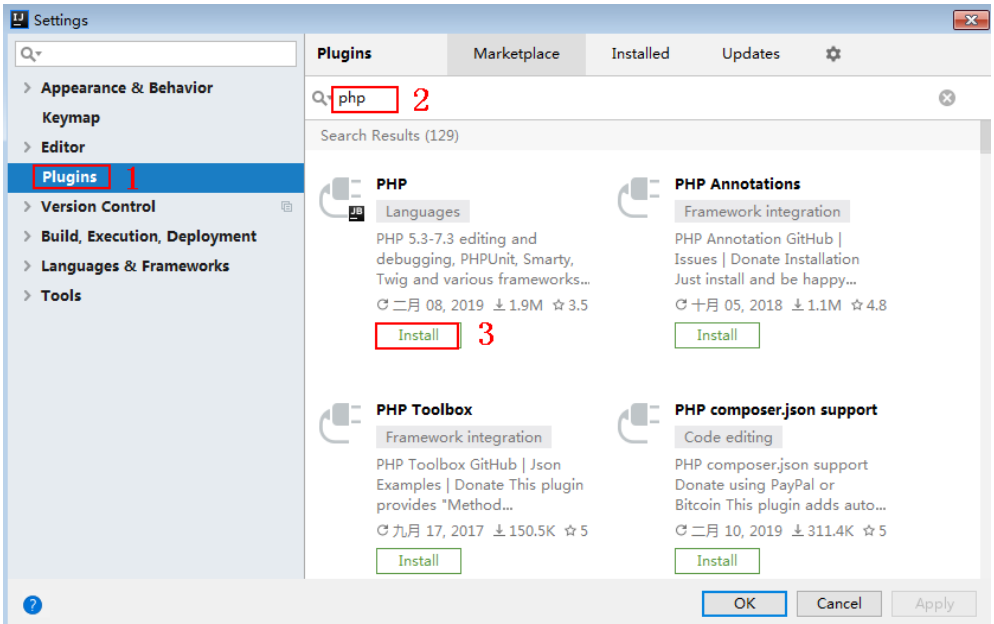
准备环境

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 获取并安装8.0.3及以上版本的PHP安装包，如果未安装，请至**PHP官方下载页面**下载。
- 将PHP安装目录中的“php.ini-production”文件复制到“C:\windows”，改名为“php.ini”，并在文件中增加如下内容。

```
extension_dir = "php安装目录/ext"
extension=openssl
extension=curl
```

- 已在IntelliJ IDEA中安装PHP插件，如果未安装，请按照**图1-36**所示安装。

图 1-36 安装 PHP 插件



获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

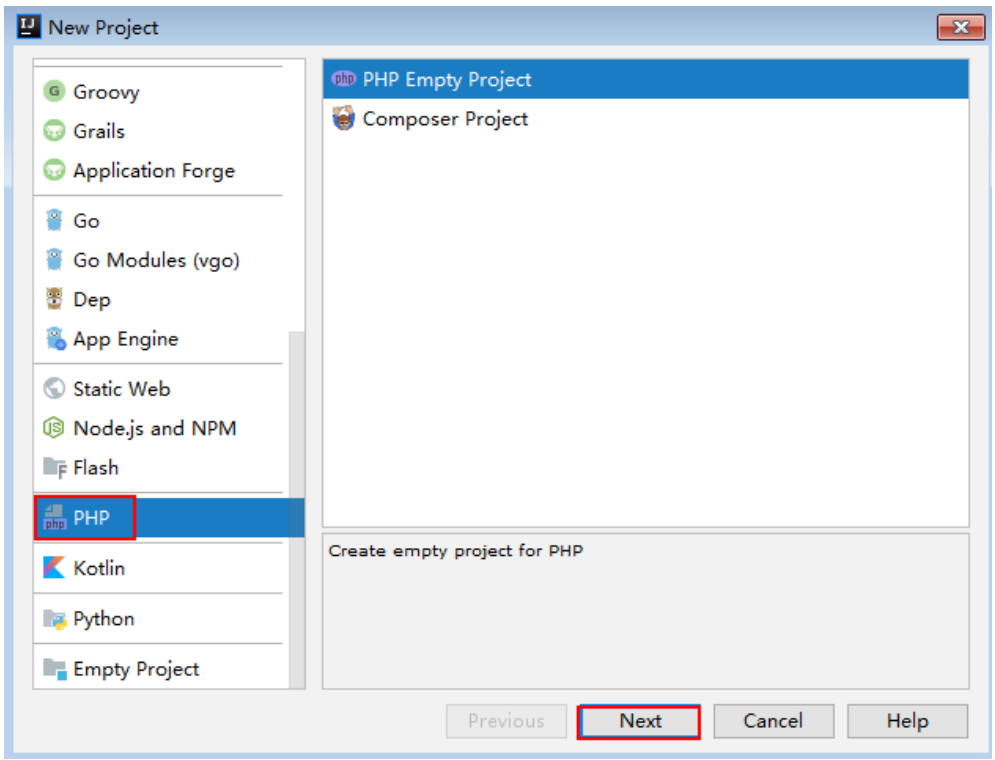
名称	说明
signer.php	SDK代码
index.php	示例代码

新建工程

步骤1 打开IDEA，选择菜单“File > New > Project”。

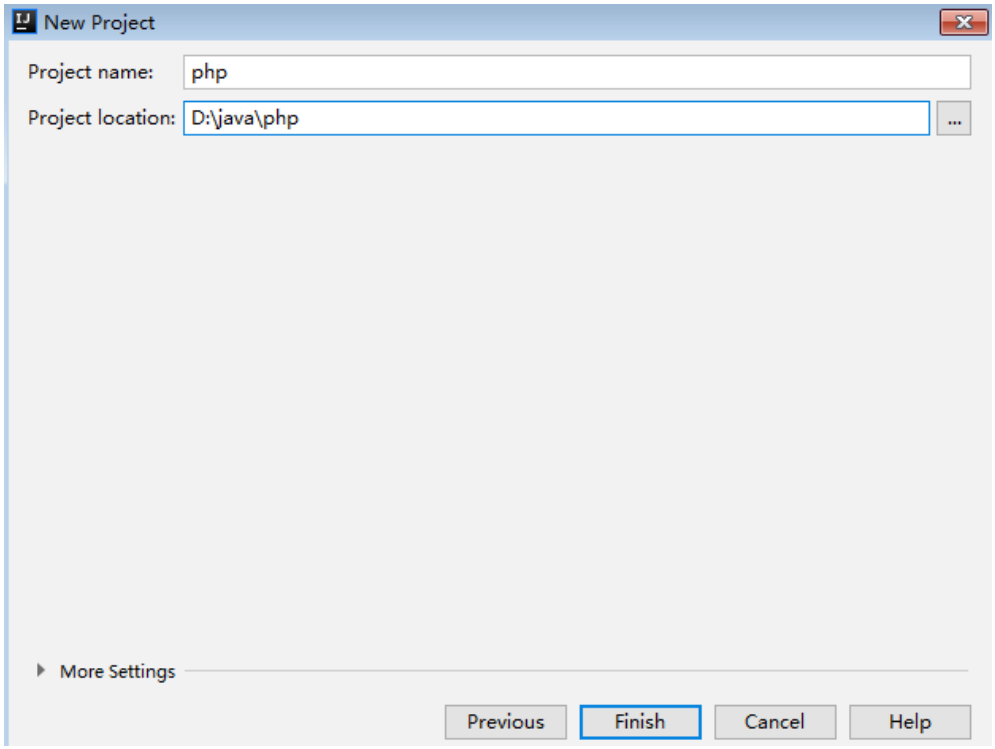
弹出“New Project”对话框，选择“PHP”，单击“Next”。

图 1-37 New Project



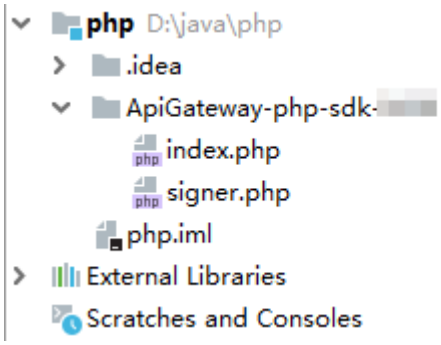
步骤2 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 1-38 选择解压后的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

图 1-39 新建工程的目录结构



“signer.php”为示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**调用API示例**。

----结束

调用 API 示例

步骤1 在代码中引入sdk。

```
require 'signer.php';
```

步骤2 生成一个新的Signer， 输入集成应用的Key和Secret。

```
$signer = new Signer();
$signer->Key = '4f5f626b-073f-402f-a1e0-e52171c6100c';
$signer->Secret = "******";
```

步骤3 生成一个新的Request，指定方法名、请求url和body。

```
$req = new Request('GET', "https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1?a=1");
$req->body = '';
```

步骤4 给请求添加x-stage头，内容为环境名。如果有需要，添加需要签名的其他头域。

```
$req->headers = array(
    'x-stage' => 'RELEASE',
);
```

步骤5 进行签名，执行此函数会生成一个$curl上下文变量。

```
$curl = $signer->Sign($req);
```

步骤6 若使用系统分配的子域名访问https请求的API，需要忽略证书校验，否则跳过此步。

```
curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, 0);
curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, 0);
```

步骤7 访问API，查看访问结果。

```
$response = curl_exec($curl);
echo curl_getinfo($curl, CURLINFO_HTTP_CODE);
echo $response;
curl_close($curl);
```

----结束

1.2.8 C++ SDK 使用说明

操作场景

使用C++语言调用APP认证的API时，您需要先获取SDK，参考API调用示例调用API。

准备环境

1. 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
2. 安装openssl库。
   ```
   apt-get install libssl-dev
   ```
3. 安装curl库。
   ```
   apt-get install libcurl4-openssl-dev
   ```

获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

名称	说明
hasher.cpp	SDK代码
hasher.h	
header.h	
RequestParams.cpp	
RequestParams.h	

名称	说明
signer.cpp	
signer.h	
Makefile	Makefile文件
main.cpp	示例代码

调用 API 示例

步骤1 在main.cpp中加入以下引用。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>
#include "signer.h"
```

步骤2 生成一个新的Signer， 填入AppKey和AppSecret。

```
Signer signer("4f5f626b-073f-402f-a1e0-e52171c6100c", "******");
```

步骤3 生成一个新的RequestParams，指定方法名、域名、请求uri、查询字符串和body。

```
 RequestParams* request = new RequestParams("POST", "c967a237-cd6c-470e-906f-
a8655461897e.apigw.exampleRegion.com", "/app1",
        "Action=ListUsers&Version=2010-05-08", "demo");
```

步骤4 给请求添加x-stage头，内容为环境名。如果有需要，添加需要签名的其他头域。

```
request->addHeader("x-stage", "RELEASE");
```

步骤5 进行签名，执行此函数会将生成的签名头加入request变量中。

```
signer.createSignature(request);
```

步骤6 使用curl库访问API，查看访问结果。

```
static size_t
WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)
{
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;

    mem->memory = (char*)realloc(mem->memory, mem->size + realsize + 1);
    if (mem->memory == NULL) {
        /* out of memory! */
        printf("not enough memory (realloc returned NULL)\n");
        return 0;
    }

    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;

    return realsize;
}

//send http request using curl library
int perform_request(RequestParams* request)
{
    CURL *curl;
    CURLcode res;
    struct MemoryStruct resp_header;
    resp_header.memory = (char*)malloc(1);
    resp_header.size = 0;
```



```
    struct MemoryStruct resp_body;
    resp_body.memory = (char*)malloc(1);
    resp_body.size = 0;

    curl_global_init(CURL_GLOBAL_ALL);
    curl = curl_easy_init();

    curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, request->getMethod().c_str());
    std::string url = "http://" + request->getHost() + request->getUri() + "?" + request->getQueryParams();
    curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
    struct curl_slist *chunk = NULL;
    std::set<Header>::iterator it;
    for (auto header : *request->getHeaders()) {
        std::string headerEntry = header.getKey() + ": " + header.getValue();
        printf("%s\n", headerEntry.c_str());
        chunk = curl_slist_append(chunk, headerEntry.c_str());
    }
    printf("-------------\n");
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, chunk);
    curl_easy_setopt(curl, CURLOPT_COPYPOSTFIELDS, request->getPayload().c_str());
    curl_easy_setopt(curl, CURLOPT_NOBODY, 0L);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
    curl_easy_setopt(curl, CURLOPT_HEADERDATA, (void *)&resp_header);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)&resp_body);
    //curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
    res = curl_easy_perform(curl);
    if (res != CURLE_OK) {
        fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
    }
    else {
        long status;
        curl_easy_getinfo(curl, CURLINFO_HTTP_CODE, &status);
        printf("status %d\n", status);
        printf(resp_header.memory);
        printf(resp_body.memory);
    }
    free(resp_header.memory);
    free(resp_body.memory);
    curl_easy_cleanup(curl);

    curl_global_cleanup();

    return 0;
}
```

步骤7 运行make命令编译，得到可执行文件main，执行main文件，查看结果。

----结束

1.2.9 C SDK 使用说明

操作场景

使用C语言调用APP认证的API时，您需要先获取SDK，参考API调用示例调用API。

准备环境

1. 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
2. 安装openssl库。
   ```
   apt-get install libssl-dev
   ```
3. 安装curl库。
   ```
   apt-get install libcurl4-openssl-dev
   ```

获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

名称	说明
signer_common.c	SDK代码
signer_common.h	
signer.c	
signer.h	
Makefile	Makefile文件
main.c	示例代码

调用 API 示例

步骤1 在main.c中加入以下引用。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>
#include "signer.h"
```

步骤2 生成一个sig_params_t类型的变量，填入AppKey和AppSecret。

```
sig_params_t params;
sig_params_init(&params);
sig_str_t app_key = sig_str("4f5f626b-073f-402f-a1e0-e52171c6100c");
sig_str_t app_secret = sig_str("******");
params.key = app_key;
params.secret = app_secret;
```

步骤3 指定方法名、域名、请求uri、查询字符串和body。

```
sig_str_t host = sig_str("c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com");
sig_str_t method = sig_str("GET");
sig_str_t uri = sig_str("/app1");
sig_str_t query_str = sig_str("a=1&b=2");
sig_str_t payload = sig_str("");
params.host = host;
params.method = method;
params.uri = uri;
params.query_str = query_str;
params.payload = payload;
```

步骤4 给请求添加x-stage头，内容为环境名。如果有需要，添加需要签名的其他头域。

```
sig_headers_add(&params.headers, "x-stage", "RELEASE");
```

步骤5 进行签名，执行此函数会将生成的签名头加入request变量中。

```
sig_sign(&params);
```

步骤6 使用curl库访问API，查看访问结果。

```
static size_t
WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)
{
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;
```



```
    mem->memory = (char*)realloc(mem->memory, mem->size + realsize + 1);
    if (mem->memory == NULL) {
        /* out of memory! */
        printf("not enough memory (realloc returned NULL)\n");
        return 0;
    }

    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;

    return realsize;
}

//send http request using curl library
int perform_request(RequestParams* request)
{
    CURL *curl;
    CURLcode res;
    struct MemoryStruct resp_header;
    resp_header.memory = malloc(1);
    resp_header.size = 0;
    struct MemoryStruct resp_body;
    resp_body.memory = malloc(1);
    resp_body.size = 0;

    curl_global_init(CURL_GLOBAL_ALL);
    curl = curl_easy_init();

    curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, params.method.data);
    char url[1024];
    sig_snprintf(url, 1024, "http://%V%V?%V", &params.host, &params.uri, &params.query_str);
    curl_easy_setopt(curl, CURLOPT_URL, url);
    struct curl_slist *chunk = NULL;
    for (int i = 0; i < params.headers.len; i++) {
        char header[1024];
        sig_snprintf(header, 1024, "%V: %V", &params.headers.data[i].name, &params.headers.data[i].value);
        printf("%s\n", header);
        chunk = curl_slist_append(chunk, header);
    }
    printf("-------------\n");
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, chunk);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, params.payload.data);
    curl_easy_setopt(curl, CURLOPT_NOBODY, 0L);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);
    curl_easy_setopt(curl, CURLOPT_HEADERDATA, (void *)&resp_header);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *)&resp_body);
    //curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);
    res = curl_easy_perform(curl);
    if (res != CURLE_OK) {
        fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
    }
    else {
        long status;
        curl_easy_getinfo(curl, CURLINFO_HTTP_CODE, &status);
        printf("status %d\n", status);
        printf(resp_header.memory);
        printf(resp_body.memory);
    }
    free(resp_header.memory);
    free(resp_body.memory);
    curl_easy_cleanup(curl);

    curl_global_cleanup();

    //free signature params
    sig_params_free(&params);
```

```
    return 0;
}
```

步骤7 运行make命令编译，得到可执行文件main，执行main文件，查看结果。

----结束

1.2.10 Android SDK 使用说明

操作场景

使用Android语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考API调用示例调用API。

准备环境

- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 获取并安装4.1.2及以上版本的Android Studio，如果未安装，请至**Android Studio官方网站**下载。

获取 SDK

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK。解压后目录结构如下：

名称	说明
app\	安卓工程代码
gradle\	gradle相关文件
build.gradle	gradle配置文件
gradle.properties	
settings.gradle	
gradlew	gradle wrapper执行脚本
gradlew.bat	

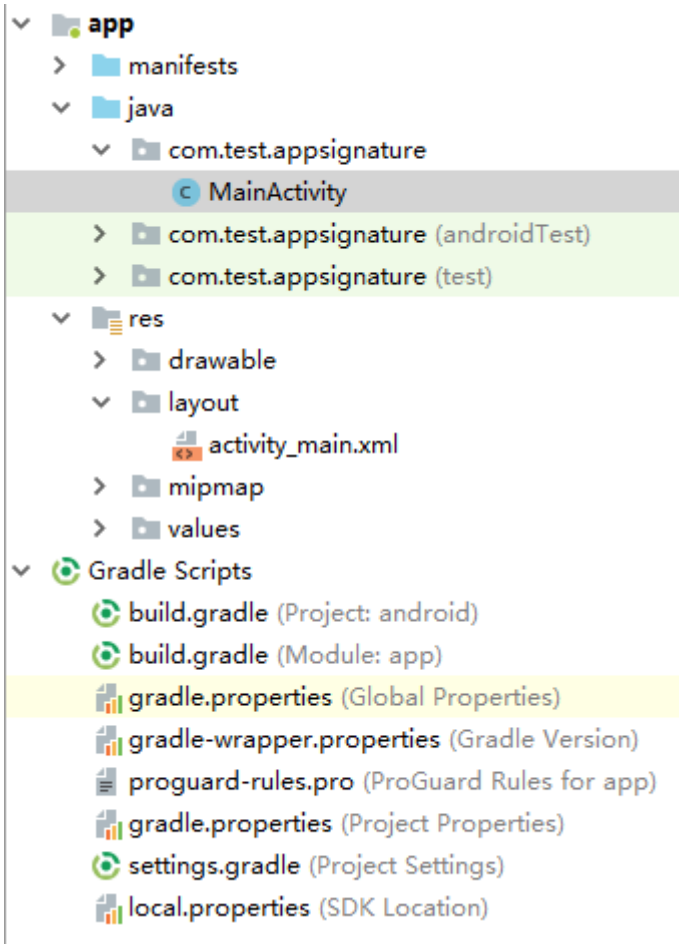
打开工程

步骤1 打开Android Studio，选择“File > Open”。

在弹出的对话框中选择解压后的SDK路径。

步骤2 打开工程后，目录结构如下。

图 1-40 工程目录结构



----结束

调用 API 示例

步骤1 在Android工程中的“app/libs”目录下，加入SDK所需jar包。其中jar包必须包括：

- java-sdk-core-x.x.x.jar
- joda-time-2.10.jar

步骤2 在“build.gradle”文件中加入okhttp库的依赖。

在“build.gradle”文件中的“dependencies”下加入“implementation 'com.squareup.okhttp3:okhttp:3.14.2'”。

```
dependencies {
    ...
    ...
    implementation 'com.squareup.okhttp3:okhttp:3.14.3'
}
```

步骤3 创建request，输入AppKey和AppSecret，并指定域名、方法名、请求uri和body。

```
Request request = new Request();
try {
    request.setKey("4f5f626b-073f-402f-a1e0-e52171c6100c");
    request.setSecrect("******");
    request.setMethod("POST");
    request.setUrl("https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1");
    request.addQueryStringParam("name", "value");
```

```
    request.addHeader("Content-Type", "text/plain");
    request.setBody("demo");
} catch (Exception e) {
    e.printStackTrace();
    return;
}
```

步骤4 对请求进行签名，生成okhttp3.Request对象来访问API。

```
okhttp3.Request signedRequest = Client.signOkhttp(request);
OkHttpClient client = new OkHttpClient.Builder().build();
Response response = client.newCall(signedRequest).execute();
```

----结束

1.2.11 curl SDK 使用说明

操作场景

使用curl命令调用APP认证的API时，您需要先下载JavaScript SDK生成curl命令，然后将curl命令复制到命令行调用API。

前提条件

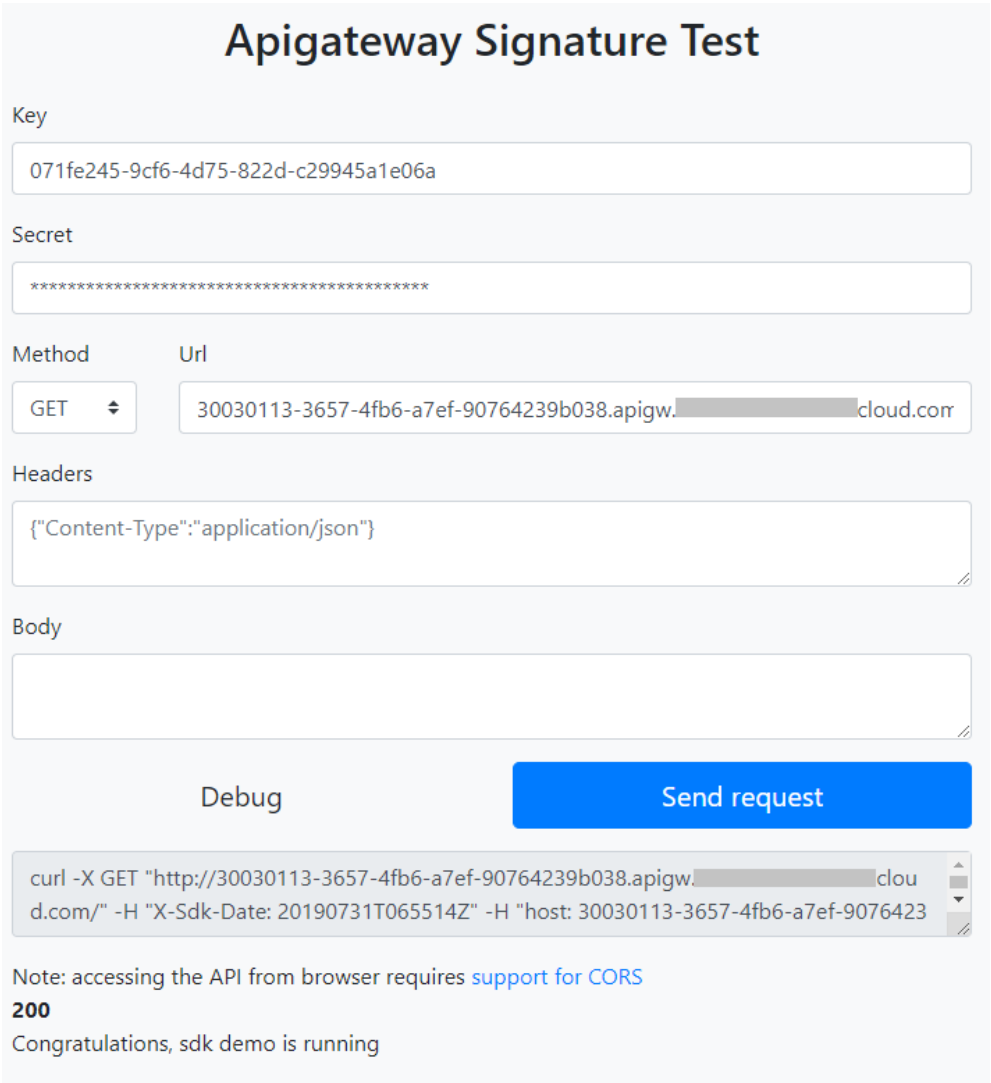
- 已获取API的域名、请求url、请求方法、集成应用的Key和Secret（或客户端的AppKey和AppSecret）等信息，具体参见**认证前准备**。
- 浏览器版本为chrome 89.0 或以上版本。

调用 API 示例

步骤1 使用JavaScript SDK生成curl命令。

请登录ROMA Connect实例控制台，在“服务集成 APIC > API调用”页面中下载SDK并解压。

在浏览器中打开demo.html，页面如下图所示。



步骤2 填入Key、Secret、方法名、请求协议、域名和url。例如：

```
Key=4f5f626b-073f-402f-a1e0-e52171c6100c
Secret=******
Method=POST
Url=https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com
```

步骤3 填入json格式的Query和Headers，填入Body。

步骤4 单击“Send request”，生成curl命令。将curl命令复制到命令行，访问API。

```
//若使用系统分配的子域名访问https请求的API时，需要忽略证书校验，在-d后添加“ -k”
$ curl -X POST "https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/" -H "X-Sdk-Date: 20180530T115847Z" -H "Authorization: SDK-HMAC-SHA256 Access=071fe245-9cf6-4d75-822d-c29945a1e06a, SignedHeaders=host;x-sdk-date, Signature=9e5314bd156d517******dd3e5765fdde4" -d ""
Congratulations, sdk demo is running
```

说明

SDK生成的curl命令不符合Window下cmd终端格式，请在git bash下执行生成的curl命令。

----结束

1.2.12 其他编程语言

APP 认证工作原理

1. **构造规范请求**。
   将待发送的请求内容按照与APIC后台约定的规则组装，确保客户端签名、APIC后台认证时使用的请求内容一致。
2. 使用规范请求和其他信息**创建待签字符串**。
3. 使用AK/SK和待签字符串**计算签名**。
4. 将生成的**签名信息作为请求消息头添加**到HTTP请求中，或者作为查询字符串参数添加到HTTP请求中。
5. APIC收到请求后，执行**1**~**3**，计算签名。
6. 将**3**中的生成的签名与**5**中生成的签名进行比较，如果签名匹配，则处理请求，否则将拒绝请求。

说明

APP签名仅支持Body体12M及以下的请求签名。

步骤 1：构造规范请求

使用APP方式进行签名与认证，首先需要规范请求内容，然后再进行签名。客户端与APIC使用相同的请求规范，可以确保同一个HTTP请求的前后端得到相同的签名结果，从而完成身份校验。

HTTP请求规范伪代码如下：

```
CanonicalRequest =
    HTTPRequestMethod + '\n' +
    CanonicalURI + '\n' +
    CanonicalQueryString + '\n' +
    CanonicalHeaders + '\n' +
    SignedHeaders + '\n' +
    HexEncode(Hash(RequestPayload))
```

我们通过以下示例来说明规范请求的构造步骤。

假设原始请求如下：

```
GET https://30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com/app1?b=2&a=1 HTTP/1.1
Host: 30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com
X-Sdk-Date: 20180330T123600Z
```

1. 构造**HTTP请求方法**（**HTTPRequestMethod**），以换行符结束。
   HTTP请求方法，如GET、PUT、POST等。请求方法示例：
   ```
   GET
   ```
2. 添加规范**URI参数**（**CanonicalURI**），以换行符结束。
   释义：
   规范URI，即请求资源路径，是URI的绝对路径部分的URI编码。
   格式：
   根据RFC 3986标准化URI路径，移除冗余和相对路径部分，路径中每个部分必须为URI编码。如果URI路径不以“/”结尾，则在尾部添加“/”。
   举例：
   示例中的URI：/app1，此时规范的URI编码为：

```
GET
/app1/
```

说明

计算签名时，URI必须以“/”结尾。发送请求时，可以不以“/”结尾。

3. 添加**规范查询字符串（CanonicalQueryString）**，以换行符结束。

释义：

查询字符串，即查询参数。如果没有查询参数，则为空字符串，即规范后的请求为**空行**。

格式：

规范查询字符串需要满足以下要求：

- 根据以下规则对每个参数名和值进行URI编码：
  - 请勿对RFC 3986定义的任何非预留字符进行URI编码，这些字符包括：A-Z、a-z、0-9、-、_、.和~。
  - 使用%XY对所有非预留字符进行百分比编码，其中X和Y为十六进制字符（0-9和A-F）。例如，空格字符必须编码为%20，扩展UTF-8字符必须采用“%XY%ZA%BC”格式。
- 对于每个参数，追加“*URI编码的参数名称*=*URI编码的参数值*”。如果没有参数值，则以空字符串代替，但不能省略“=”。

  例如以下含有两个参数，其中第二个参数parm2的值为空。

  ```
  parm1=value1&parm2=
  ```
- 按照字符代码以升序顺序对参数名进行排序。例如，以大写字母F开头的参数名排在以小写字母b开头的参数名之前。
- 以排序后的第一个参数名开始，构造规范查询字符串。

举例：

示例中包含两个可选参数：a、b

```
GET
/app1/
a=1&b=2
```

4. 添加规范**消息头**（**CanonicalHeaders**），以换行符结束。

释义：

规范消息头，即请求消息头列表。包括签名请求中的所有HTTP消息头列表。消息头必须包含X-Sdk-Date，用于校验签名时间，格式为ISO8601标准的UTC时间格式：YYYYMMDDTHHMMSSZ。如果API发布到非RELEASE环境时，需要增加自定义的环境名称。

注意

客户端须注意本地时间与时钟服务器的同步，避免请求消息头X-Sdk-Date的值出现较大误差。

ROMA Connect除了校验X-Sdk-Date的时间格式外，还会校验该时间值与收到请求的时间差，如果时间差超过15分钟，ROMA Connect将拒绝请求。

格式：

CanonicalHeaders由多个请求消息头共同组成，**CanonicalHeadersEntry0 + CanonicalHeadersEntry1 + ...**，其中每个请求消息头

（CanonicalHeadersEntry）**的格式为Lowercase(HeaderName) + ':' + Trimall(HeaderValue) + '\n'**

说明

- Lowercase表示将所有字符转换为小写字母的函数。
- Trimall表示删除值前后的多余空格的函数。
- 最后一个请求消息头也会携带一个换行符。叠加规范中**CanonicalHeaders**自身携带的换行符，因此会出现一个空行。

举例：

```
GET
/app1/
a=1&b=2
host:30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com
x-sdk-date:20180330T123600Z
```

须知

规范消息头需要满足以下要求：

- 将消息头名称转换为小写形式，并删除前导空格和尾随空格。
- 按照字符代码对消息头名称进行升序排序。

例如原始消息头为：

```
Host:30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com\n
Content-Type: application/json;charset=utf8\n
My-header1:    a   b   c  \n
X-Sdk-Date:20180330T123600Z\n
My-Header2:    "a   b   c"  \n
```

规范消息头为：

```
content-type:application/json;charset=utf8\n
host:30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com\n
my-header1:a   b   c\n
my-header2:"a   b   c"\n
x-sdk-date:20180330T123600Z\n
```

5. 添加用于**签名的消息头声明**（**SignedHeaders**），以换行符结束。

释义：

用于签名的请求消息头列表。通过添加此消息头，向APIC告知请求中哪些消息头是签名过程的一部分，以及在验证请求时APIC可以忽略哪些消息头。X-Sdk-date必须作为已签名的消息头。

格式：

SignedHeaders = Lowercase(HeaderName0) + ';' + Lowercase(HeaderName1) + ";" + ...

已签名的消息头需要满足以下要求：将已签名的消息头名称转换为小写形式，按照字符代码对消息头进行排序，并使用“;”来分隔多个消息头。

Lowercase表示将所有字符转换为小写字母。

举例：

以下表示有两个消息头参与签名：host、x-sdk-date

```
GET
/app1/
a=1&b=2
host:30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com
x-sdk-date:20180330T123600Z
```

```
host;x-sdk-date
```

6. 使用SHA 256哈希函数以基于HTTP或HTTPS请求正文中的body体（**RequestPayload**），创建哈希值。

   释义：

   请求消息体。消息体需要做两层转换：HexEncode(Hash(*RequestPayload*))，其中Hash表示生成消息摘要的函数，当前支持SHA-256算法。HexEncode表示以小写字母形式返回摘要的Base-16编码的函数。例如，HexEncode("m") 返回值为“6d”而不是“6D”。输入的每一个字节都表示为两个十六进制字符。

   说明

   计算RequestPayload的哈希值时，对于“RequestPayload==null”的场景，直接使用空字符串""来计算。

   举例：

   本示例为GET方法，body体为空。经过哈希处理的body（空字符串）如下：

   ```
   GET
   /app1/
   a=1&b=2
   host:30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com
   x-sdk-date:20180330T123600Z

   host;x-sdk-date
   e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
   ```

7. 对构造好的规范请求进行哈希处理，算法与对RequestPayload哈希处理的算法相同。经过哈希处理的规范请求必须以小写十六进制字符串形式表示。

   算法伪代码：**Lowercase(HexEncode(Hash.SHA256(CanonicalRequest)))**

   经过哈希处理的规范请求示例：

   ```
   4bd8e1afe76738a332ecff075321623fb90ebb181fe79ec3e23dcb081ef15906
   ```

步骤 2：创建待签字符串

对HTTP请求进行规范并取得请求的哈希值后，将其与签名算法、签名时间一起组成待签名字符串。

```
StringToSign =
  Algorithm + \n +
  RequestDateTime + \n +
  HashedCanonicalRequest
```

伪代码中参数说明如下。

- **Algorithm**

  签名算法。对于SHA 256，算法为SDK-HMAC-SHA256。

- **RequestDateTime**

  请求时间戳。与请求消息头X-Sdk-Date的值相同，格式为YYYYMMDDTHHMMSSZ。

- **HashedCanonicalRequest**

  经过哈希处理的规范请求。

上述例子得到的待签字符串为：

```
SDK-HMAC-SHA256
20180330T123600Z
4bd8e1afe76738a332ecff075321623fb90ebb181fe79ec3e23dcb081ef15906
```


步骤 3：计算签名

将APP secret和创建的待签字符串作为加密哈希函数的输入，计算签名，将二进制值转换为十六进制表示形式。

伪代码如下：

```
signature = HexEncode(HMAC(APP secret, string to sign))
```

其中HMAC指密钥相关的哈希运算，HexEncode指转十六进制。伪代码中参数说明如表1-1所示。

表 1-1 参数说明

参数名称	参数解释
APP secret	签名密钥
string to sign	创建的待签字符串

假设APP secret为12345678-1234-1234-1234-123456781234，则计算得到的signature为：

```
cb978df7c06ac242bab1d1b39d697ef7df4806664a6e09d5f5308a6b25043ea2
```

步骤 4：添加签名信息到请求头

在计算签名后，将它添加到Authorization的HTTP消息头。Authorization消息头未包含在已签名消息头中，主要用于身份验证。

伪代码如下：

```
Authorization header创建伪码：
Authorization: algorithm Access=APP key, SignedHeaders=SignedHeaders, Signature=signature
```

需要注意的是算法与Access之前没有逗号，但是SignedHeaders与Signature之前需要使用逗号隔开。

得到的签名消息头为：

```
Authorization: SDK-HMAC-SHA256 Access=071fe245-9cf6-4d75-822d-c29945a1e06a, SignedHeaders=host;x-sdk-date, Signature=cb978df7c06ac242bab1d1b39d697ef7df4806664a6e09d5f5308a6b25043ea2
```

得到签名消息头后，将其增加到原始HTTP请求内容中，请求将被发送给APIC，由APIC完成身份认证。身份认证通过后，该请求才会发送给后端服务进行业务处理。

1.3 IAM 认证开发

1.3.1 IAM 认证开发（Token）

操作场景

当您使用Token认证方式完成认证鉴权时，需要获取用户Token并在调用接口时增加“X-Auth-Token”到业务接口请求消息头中。

说明

调用接口有如下两种认证方式，您可以选择其中一种进行认证鉴权。

- Token认证：通过Token认证调用用请求。
- AK/SK认证：通过AK（Access Key ID）/SK（Secret Access Key）对调用请求内容进行签名认证。

调用接口步骤

1. 获取Token。

   请参考《统一身份认证服务 API参考》的“获取用户Token”接口，获取Token。

   请求响应成功后在响应消息头中包含的“X-Subject-Token”的值即为Token值。

   请求内容示例如下：

```
POST https://{iam_endpoint}/v3/auth/tokens
Content-Type: application/json

{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
            "password": {
                "user": {
                    "name": "username",
                    "password": "********",
                    "domain": {
                        "name": "domainname"
                    }
                }
            }
        },
        "scope": {
            "project": {
                "id": "xxxxxxxx"
            }
        }
    }
}
```

   其中：

   - **{iam_endpoint}**请参见**地区和终端节点**获取。
   - ***username***为用户名。
   - **domainname**为用户所属的帐号名称。
   - ********为用户登录密码。
   - ***xxxxxxxx***为项目ID。

     项目ID可以在管理控制台上，单击用户名，在下拉列表中单击“我的凭证”，查看“项目ID”。

2. 调用业务接口，在请求消息头中增加“X-Auth-Token”，“X-Auth-Token”的取值为1中获取的Token。

1.3.2 IAM 认证开发（AK/SK）

使用AK（Access Key ID）、SK（Secret Access Key）对请求进行签名。

说明

- AK：访问密钥ID。与私有访问密钥关联的唯一标识符，访问密钥ID和私有访问密钥一起使用，对请求进行加密签名。
- SK：与访问密钥ID结合使用的密钥，对请求进行加密签名，可标识发送方，并防止请求被修改。

生成 AK、SK

如果已生成过AK/SK，则可跳过此步骤，找到原来已下载的AK/SK文件，文件名一般为：credentials.csv。

如下图所示，文件包含了租户名（User Name），AK（Access Key Id），SK（Secret Access Key）。

图 1-41 credential.csv 文件内容

	A	B	C
1	User Name	Access Key Id	Secret Access Key
2	hu[redacted]dg	QTWA[redacted]UT2QVKYUC	MFyfvK41ba2[redacted]npdUKGpownRZlmVmHc

AK/SK生成步骤：

1. 注册并登录管理控制台。
2. 单击右上角的用户名，在下拉列表中单击“我的凭证”。
3. 单击“访问密钥”。
4. 单击“新增访问密钥”，进入“新增访问密钥”页面。
5. 输入登录密码和验证码，单击“确定”，下载密钥，请妥善保管。

图 1-42 访问密钥获取页面示意

生成签名

生成签名的方式和**APP认证**相同，用AK代替APP认证中的AppKey，SK替换APP认证中的AppSecret，即可完成签名和请求。

注意

客户端须注意本地时间与时钟服务器的同步，避免请求消息头X-Sdk-Date的值出现较大误差。

ROMA Connect除了校验X-Sdk-Date的时间格式外，还会校验该时间值与收到请求的时间差，如果时间差超过15分钟，ROMA Connect将拒绝请求。

1.4 后端服务签名开发

1.4.1 Java SDK 使用说明

操作场景

使用Java语言进行后端服务签名时，您需要先获取SDK，然后导入工程，最后参考校验后端签名示例校验签名是否一致。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

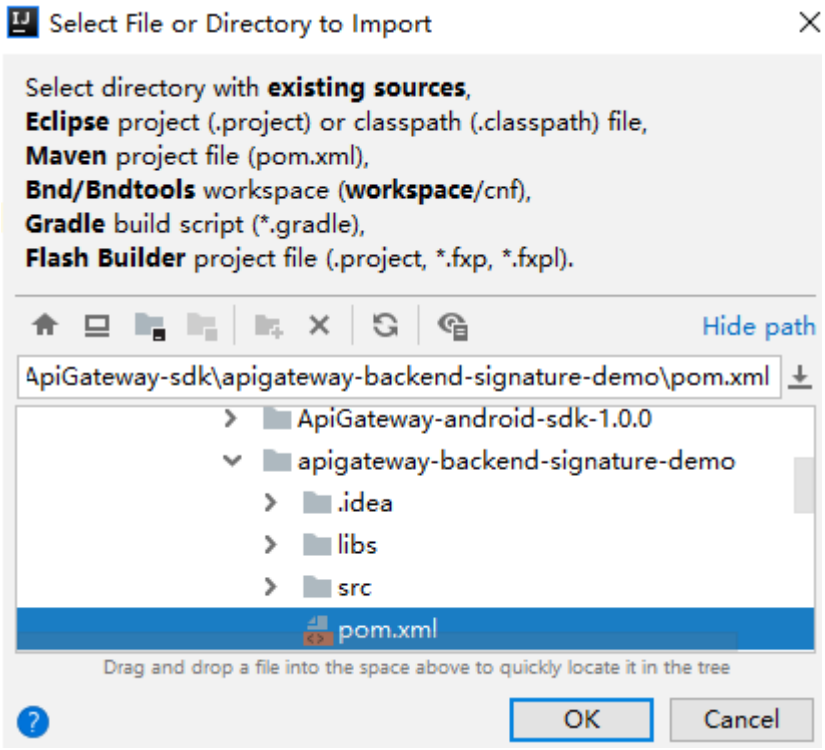
前提条件

- 准备待用的签名密钥的Key和Secret。
- 已在控制台创建签名密钥，并绑定API，具体请参见**配置后端服务的签名校验**。
- 已获取后端签名实例代码，您可以在ROMA Connect实例控制台的“服务集成APIC > API管理”页面中，选择“签名密钥”页签，单击“下载SDK”下载签名示例代码。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 已安装Java Development Kit 1.8.111或以上版本，如果未安装，请至**Oracle官方下载页面**下载。

导入工程

步骤1 打开IntelliJ IDEA，在菜单栏选择“File > New > Project from Existing Sources”，选择解压后的“apigateway-backend-signature-demo\pom.xml”文件，单击“OK”。

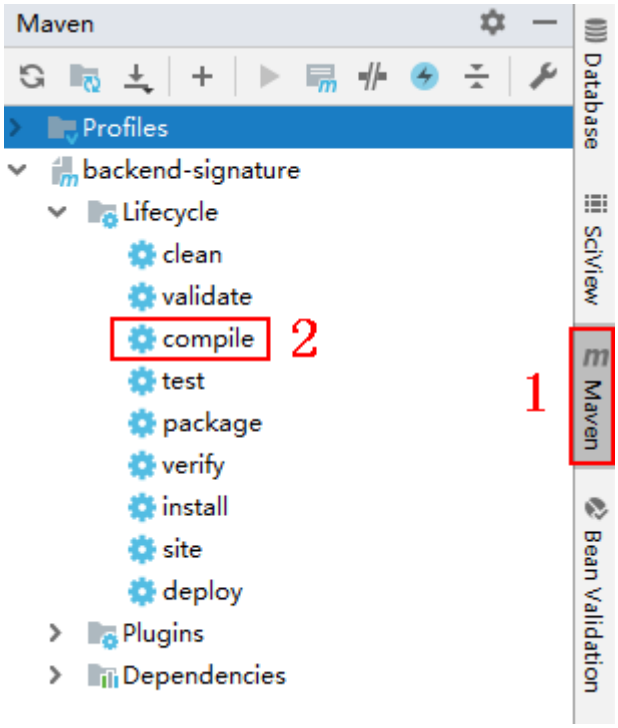
图 1-43 Select File or Directory to Import



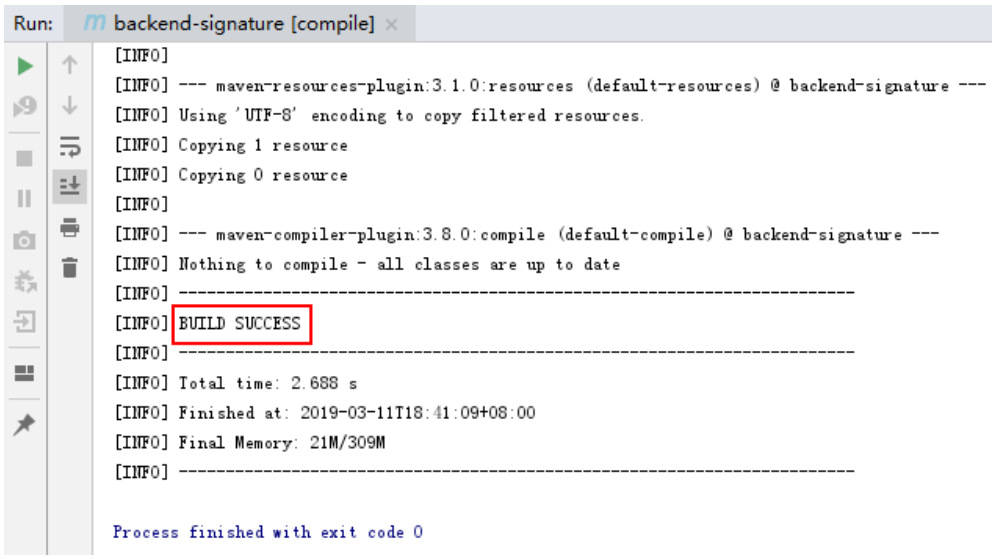
步骤2 保持默认设置，单击“Next > Next > Next > Next > Finish”，完成工程导入。

步骤3 在右侧Maven页签，双击“compile”进行编译。

图 1-44 编译工程

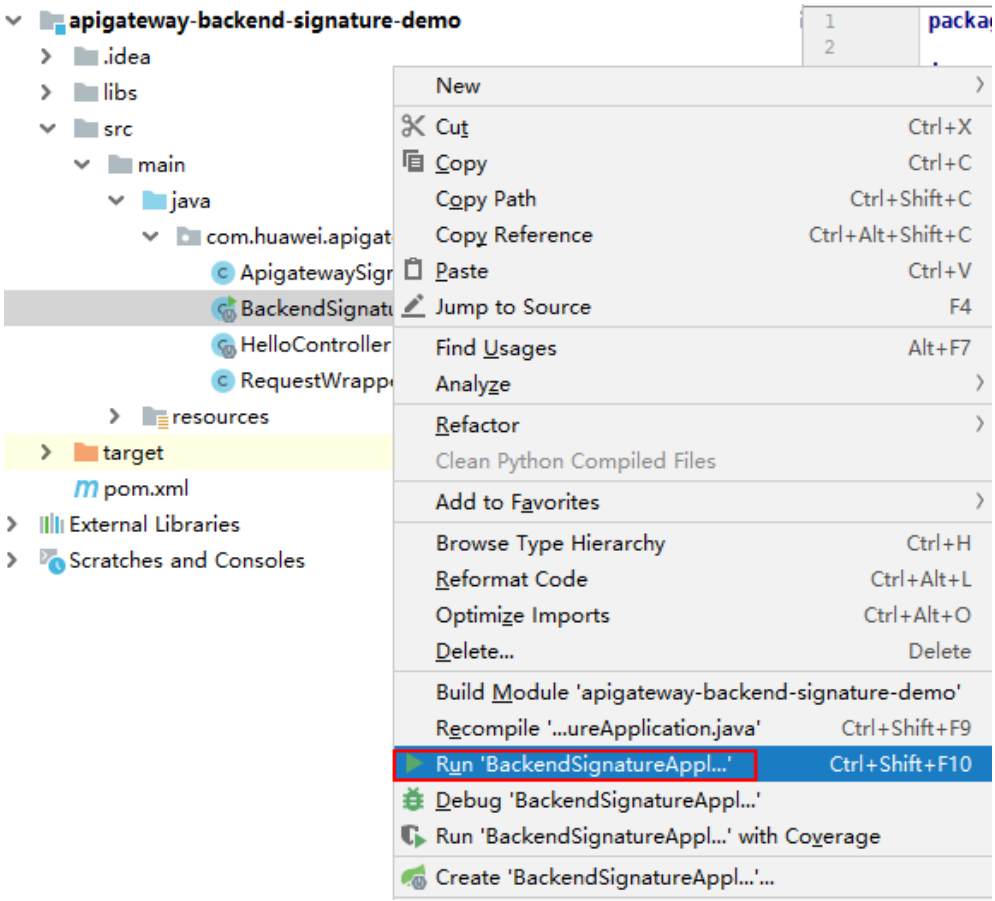


返回“BUILD SUCCESS”，表示编译成功。



步骤4 右键单击BackendSignatureApplication，选择“Run”运行服务。

图 1-45 运行服务



“ApigatewaySignatureFilter.java”为示例代码，请根据实际情况修改参数后使用。具体代码说明请参考**校验hmac类型后端签名示例**。

----结束

校验 hmac 类型后端签名示例

说明

- 示例演示如何编写一个基于Spring boot的服务器，作为API的后端，并且实现一个Filter，对APIC的请求做签名校验。
- API绑定hmac类型签名密钥后，发给后端的请求中会添加签名信息。

步骤1 编写一个Controller，路径为/hmac。

```
// HelloController.java

@RestController
@EnableAutoConfiguration
public class HelloController {

    @RequestMapping("/hmac")
    private String hmac() {
        return "Hmac authorization success";
    }
}
```

步骤2 编写一个Filter，匹配所有路径和方法。将允许的签名key和secret对放入一个Map中。

```
public class ApigatewaySignatureFilter implements Filter {
    private static Map<String, String> secrets = new HashMap<>();
    static {
        secrets.put("signature_key1", "signature_secret1");
        secrets.put("signature_key2", "signature_secret2");
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain chain) {
        //签名校验代码
        ...
    }
}
```

步骤3 doFilter函数为签名校验代码。校验流程如下：由于filter中需要读取body，为了使得body可以在后续的filter和controller中再次读取，把request包装起来传给后续的filter和controller。包装类的具体实现可见RequestWrapper.java。

```
RequestWrapper request = new RequestWrapper((HttpServletRequest) servletRequest);
```

步骤4 使用正则表达式解析Authorization头，得到signingKey和signedHeaders。

```
private static final Pattern authorizationPattern = Pattern.compile("SDK-HMAC-SHA256\\s+Access=([^,]+),\
\s?SignedHeaders=([^,]+),\\s?Signature=(\\w+)");

...

String authorization = request.getHeader("Authorization");
if (authorization == null || authorization.length() == 0) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization not found.");
    return;
}

Matcher m = authorizationPattern.matcher(authorization);
if (!m.find()) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization format incorrect.");
    return;
}
String signingKey = m.group(1);
String signingSecret = secrets.get(signingKey);
if (signingSecret == null) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Signing key not found.");
```

```
    return;
}
String[] signedHeaders = m.group(2).split(";");
```

例如，Authorization头为：

```
SDK-HMAC-SHA256 Access=signature_key1, SignedHeaders=host;x-sdk-date,
Signature=e11adf65a20d1b82c25419b5********8d0ba12fed1ceb13ed00
```

则解析的结果为：

```
signingKey=signature_key1
signedHeaders=host;x-sdk-date
```

步骤5 通过signingKey找到signingSecret，如果不存在signingKey，则返回认证失败。

```
String signingSecret = secrets.get(signingKey);
if (signingSecret == null) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Signing key not found.");
    return;
}
```

步骤6 新建一个Request对象，将请求method、url、query、signedHeaders对应的请求头放入其中。判断是否需要设置body并设置。

需要读取body的条件为：不存在值为UNSIGNED-PAYLOAD的x-sdk-content-sha256头。

```
Request apiRequest = new DefaultRequest();
apiRequest.setHttpMethod(HttpMethodName.valueOf(request.getMethod()));
String url = request.getRequestURL().toString();
String queryString = request.getQueryString();
try {
    apiRequest.setEndpoint((new URL(url)).toURI());
    Map<String, String> parametersmap = new HashMap<>();
    if (null != queryString && !"".equals(queryString)) {
        String[] parameterarray = queryString.split("&");
        for (String p : parameterarray) {
            String[] p_split = p.split("=", 2);
            String key = p_split[0];
            String value = "";
            if (p_split.length >= 2) {
                value = p_split[1];
            }
            parametersmap.put(URLDecoder.decode(key, "UTF-8"), URLDecoder.decode(value, "UTF-8"));
        }
        apiRequest.setParameters(parametersmap); //set query
    }
} catch (URISyntaxException e) {
    e.printStackTrace();
}

boolean needbody = true;
String dateHeader = null;
for (int i = 0; i < signedHeaders.length; i++) {
    String headerValue = request.getHeader(signedHeaders[i]);
    if (headerValue == null || headerValue.length() == 0) {
        ((HttpServletResponse) response).sendError(HttpServletResponse.SC_UNAUTHORIZED, "signed
header" + signedHeaders[i] + " not found.");
    } else {
        apiRequest.addHeader(signedHeaders[i], headerValue);//set header
        if (signedHeaders[i].toLowerCase().equals("x-sdk-content-sha256") &&
headerValue.equals("UNSIGNED-PAYLOAD")) {
            needbody = false;
        }
        if (signedHeaders[i].toLowerCase().equals("x-sdk-date")) {
            dateHeader = headerValue;
        }
    }
}
```

```
if (needbody) {
        apiRequest.setContent(new ByteArrayInputStream(request.getBody()));    //set body
}
```

步骤7 校验签名是否过期。从X-Sdk-Date头中取出时间，判断与服务器时间是否相差在15分钟以内。如果signedHeaders中不包含X-Sdk-Date，也返回认证失败。

```
private static final DateTimeFormatter timeFormatter =
DateTimeFormat.forPattern("yyyyMMdd'T'HHmmss'Z'").withZoneUTC();

...

if (dateHeader == null) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Header x-sdk-date not found.");
    return;
}
long date = timeFormatter.parseMillis(dateHeader);
long duration = Math.abs(DateTime.now().getMillis() - date);
if (duration > 15 * 60 * 1000) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Signature expired.");
    return;
}
```

步骤8 将Authorization头也放入Request对象中，调用verify方法校验请求签名。如果校验通过，则执行下一个filter，否则返回认证失败。

```
DefaultSigner signer = (DefaultSigner) SignerFactory.getSigner();
boolean verify = signer.verify(apiRequest, new BasicCredentials(signingKey, signingSecret));
if (verify) {
    chain.doFilter(request, response);
} else {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "verify authroization failed.");
}
```

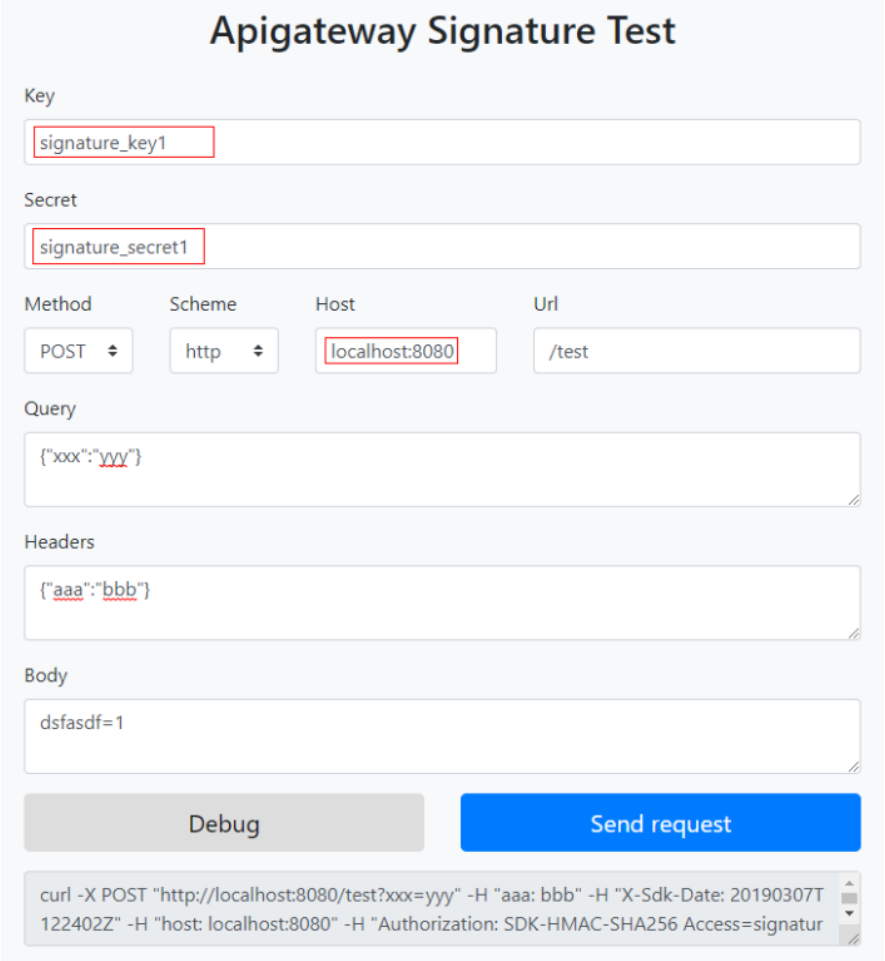
步骤9 注册filter和路径的映射关系。

```
@Configuration
public class FilterConfig {
    @Bean
    public FilterRegistrationBean registApigatewaySignatureFilter() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(new ApigatewaySignatureFilter());
        registration.addUrlPatterns("/hmac");
        registration.setName("ApigatewaySignatureFilter");
        return registration;
    }
}
```

步骤10 运行服务器，验证代码正确性。下面示例使用JavaScript SDK中的html签名工具生成签名。

填入如图所示字段后，单击“Send request”，复制生成的curl命令，并在命令行中执行，服务器返回“Hello World!”。

如果使用错误的Key和Secret访问，服务器返回401认证不通过。



----结束

校验 basic 类型后端签名示例

说明

- 示例演示如何编写一个基于Spring boot的服务器，作为API的后端，并且实现一个Filter，对APIC的请求做签名校验。
- API绑定basic类型签名密钥后，发给后端的请求中会添加basic认证信息，其中basic认证的用户名为签名密钥的key，密码为签名密钥的secret。

步骤1 编写一个Controller，路径为/basic。

```
// HelloController.java

@RestController
@EnableAutoConfiguration
public class HelloController {

    @RequestMapping("/basic")
    private String basic() {
        return "Basic authorization success";
    }
}
```

步骤2 编写一个Filter，按照basic认证的规则，Authorization头格式为"Basic "+base64encode(username+":"+password)。以下为根据规则编写的校验代码。

```
// BasicAuthFilter.java
public class BasicAuthFilter implements Filter {
```



```
    private static final String CREDENTIALS_PREFIX = "Basic ";
    private static Map<String, String> secrets = new HashMap<>();

    static {
        secrets.put("signature_key1", "signature_secret1");
        secrets.put("signature_key2", "signature_secret2");
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain chain) {
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        try {
            String credentials = request.getHeader("Authorization");
            if (credentials == null || credentials.length() == 0) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization not found.");
                return;
            }

            if (!credentials.startsWith(CREDENTIALS_PREFIX)) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization format incorrect.");
                return;
            }
            String authInfo = credentials.substring(CREDENTIALS_PREFIX.length());
            String decoded;
            try {
                decoded = new String(Base64.getDecoder().decode(authInfo));
            } catch (IllegalArgumentException e) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization format incorrect.");
                return;
            }
            String[] spl = decoded.split(":", 2);
            if (spl.length < 2) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization format incorrect.");
                return;
            }
            String signingSecret = secrets.get(spl[0]);
            if (signingSecret == null) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Username not found.");
                return;
            }
            if (signingSecret.equals(spl[1])) {
                chain.doFilter(request, response);
            } else {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Incorrect username or password");
            }
        } catch (Exception e) {
            e.printStackTrace();
            try {
                response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            } catch (IOException e1) {
            }
        }
    }
}
```

步骤3 注册filter和路径的映射关系。

```
@Configuration
public class FilterConfig {
    @Bean
    public FilterRegistrationBean registBasicAuthFilter() {
        FilterRegistrationBean registration = new FilterRegistrationBean();
        registration.setFilter(new BasicAuthFilter());
        registration.addUrlPatterns("/basic");
        registration.setName("BasicAuthFilter");
        return registration;
    }
}
```


步骤4 运行服务器，验证代码正确性。将用户名和密码生成basic认证的Authorization头域传给请求接口。如果使用错误的用户名和密码访问，服务器返回401认证不通过。

----结束

1.4.2 Python SDK 使用说明

操作场景

使用Python语言进行后端服务签名时，您需要先获取SDK，然后导入工程，最后参考校验后端签名示例校验签名是否一致。

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

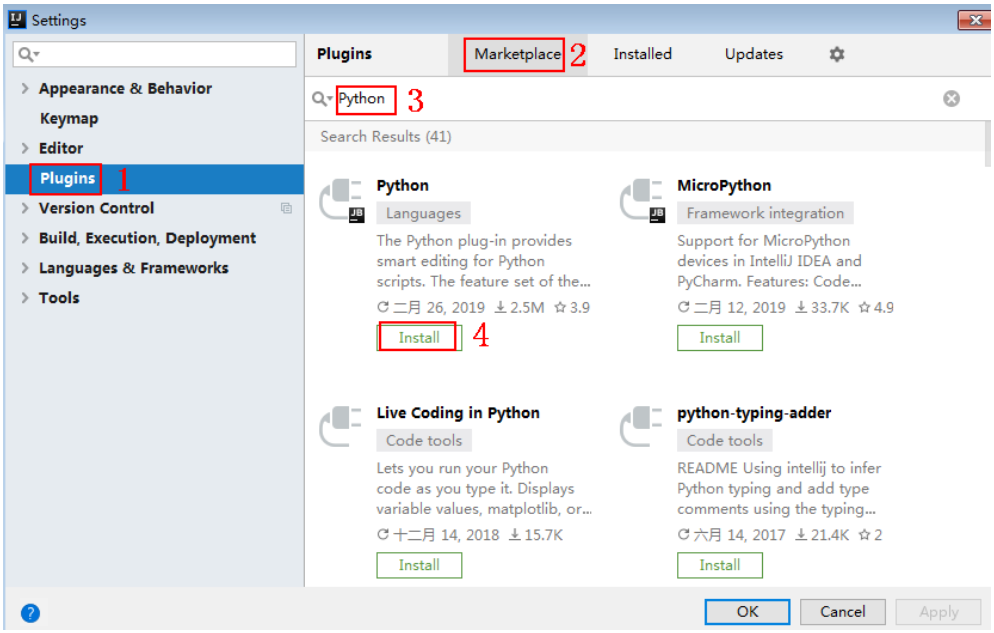
说明

Python SDK仅支持hmac类型的后端服务签名。

准备环境

- 准备待用的签名密钥的Key和Secret。
- 已在控制台创建签名密钥，并绑定API，具体请参见**配置后端服务的签名校验**。
- 已获取后端签名SDK，您可以在ROMA Connect实例控制台的“服务集成 APIC > API调用”页面中下载Python SDK。
- 获取并安装2.7或3.X版本的Python安装包，如果未安装，请至**Python官方下载页面**下载。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，如果未安装，请至**IntelliJ IDEA官方网站**下载。
- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照**图1-46**所示安装。

图 1-46 安装 Python 插件

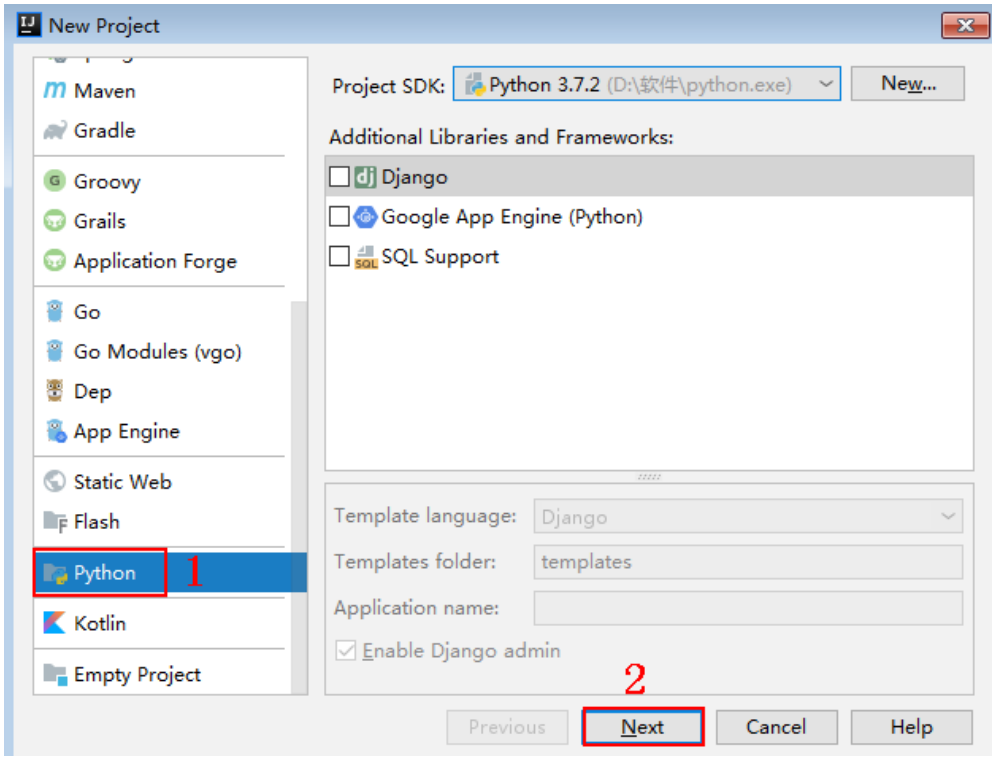


导入工程

步骤1 打开IntelliJ IDEA，在菜单栏选择“File > New > Project”。

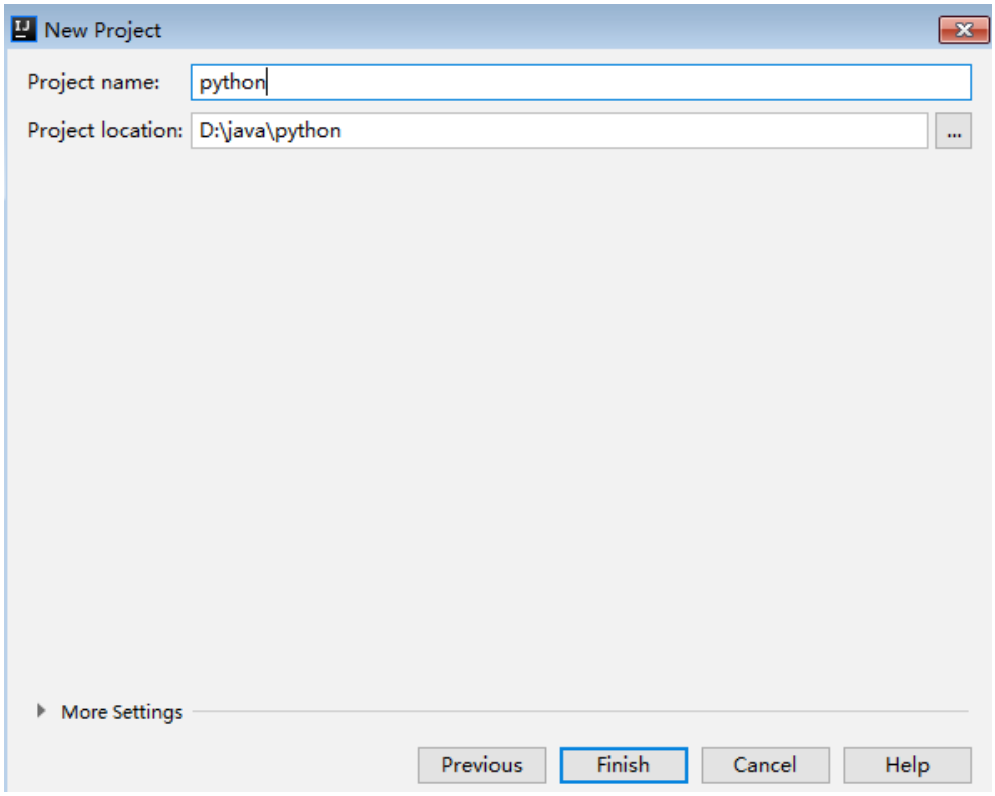
弹出“New Project”对话框，选择“Python”，单击“Next”。

图 1-47 New Project



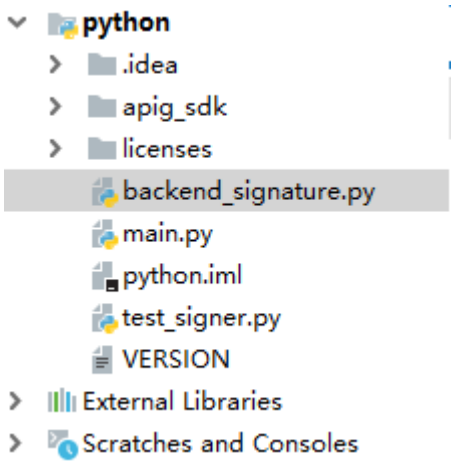
步骤2 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 1-48 选择解压后的 SDK 路径



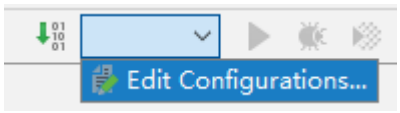
步骤3 完成工程创建后，目录结构如下。

图 1-49 目录结构



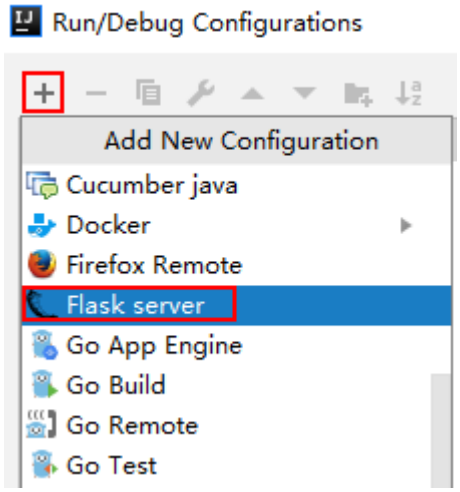
步骤4 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 1-50 Edit Configurations

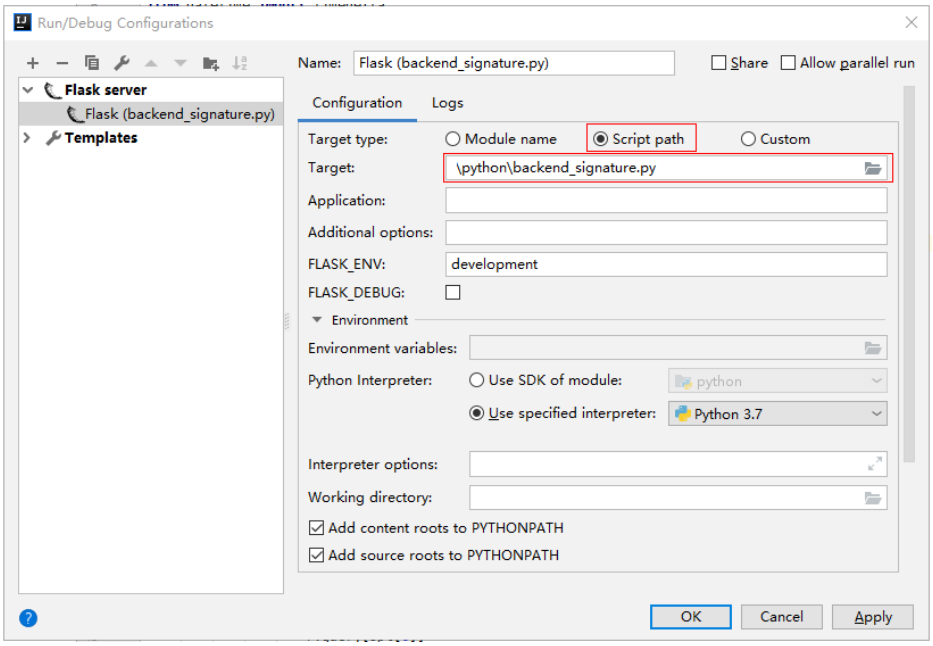


步骤5 单击“+”，选择“Flask server”。

图 1-51 选择 Flask server



步骤6 “Taget type”选择“Script path”，“Target”选择工程下的“backend_signature.py”文件，单击“OK”，完成工程配置。



----结束

校验后端签名示例

说明

- 示例演示如何编写一个基于Flask的服务器，作为API的后端，并且实现一个wrapper，对APIC的请求做签名校验。
- API绑定签名密钥后，发给后端的请求中才会添加签名信息。

步骤1 编写一个返回“Hello World!”的接口，方法为GET、POST、PUT和DELETE，且使用requires_apigateway_signature的wrapper。

```
app = Flask(__name__)

@app.route("/<id>", methods=['GET', 'POST', 'PUT', 'DELETE'])
```

```
@requires_apigateway_signature()
def hello(id):
    return "Hello World!"
```

步骤2 实现requires_apigateway_signature。将允许的签名key和secret对放入一个dict中。

```
def requires_apigateway_signature():
    def wrapper(f):

        secrets = {
            "signature_key1": "signature_secret1",
            "signature_key2": "signature_secret2",
        }
        authorizationPattern = re.compile(
            r'SDK-HMAC-SHA256\s+Access=([^,]+),\s?SignedHeaders=([^,]+),\s?Signature=(\w+)')
        BasicDateFormat = "%Y%m%dT%H%M%SZ"

        @wraps(f)
        def wrapped(*args, **kwargs):
            //签名校验代码
            ...

            return f(*args, **kwargs)
        return wrapped
    return wrapper
```

步骤3 wrapped函数为签名校验代码。校验流程如下：使用正则表达式解析Authorization头。得到key和signedHeaders。

```
if "authorization" not in request.headers:
    return 'Authorization not found.', 401
authorization = request.headers['authorization']
m = authorizationPattern.match(authorization)
if m is None:
    return 'Authorization format incorrect.', 401
signingKey = m.group(1)
signedHeaders = m.group(2).split(";")
```

例如，Authorization头为：

```
SDK-HMAC-SHA256 Access=signature_key1, SignedHeaders=host;x-sdk-date,
Signature=e11adf65a20d1b82c25419b5********8d0ba12fed1ceb13ed00
```

则解析的结果为：

```
signingKey=signature_key1
signedHeaders=host;x-sdk-date
```

步骤4 通过key找到secret，如果不存在key，则返回认证失败。

```
if signingKey not in secrets:
    return 'Signing key not found.', 401
signingSecret = secrets[signingKey]
```

步骤5 新建一个HttpRequest对象，将请求method、url、query、signedHeaders对应的请求头放入其中。判断是否需要设置body并设置。

需要读取body的条件为：不存在值为UNSIGNED-PAYLOAD的x-sdk-content-sha256头。

```
r = signer.HttpRequest()
r.method = request.method
r.uri = request.path
r.query = {}
for k in request.query_string.decode('utf-8').split('&'):
    spl = k.split("=", 1)
    if len(spl) < 2:
        r.query[spl[0]] = ""
    else:
        r.query[spl[0]] = spl[1]
r.headers = {}
```



```
needbody = True
dateHeader = None
for k in signedHeaders:
    if k not in request.headers:
        return 'Signed header ' + k + ' not found', 401
    v = request.headers[k]
    if k.lower() == 'x-sdk-content-sha256' and v == 'UNSIGNED-PAYLOAD':
        needbody = False
    if k.lower() == 'x-sdk-date':
        dateHeader = v
    r.headers[k] = v
if needbody:
    r.body = request.get_data()
```

步骤6 校验签名是否过期。从X-Sdk-Date头中取出时间，判断与服务器时间是否相差在15分钟以内。如果signedHeaders中不包含X-Sdk-Date，也返回认证失败。

```
if dateHeader is None:
    return 'Header x-sdk-date not found.', 401
t = datetime.strptime(dateHeader, BasicDateFormat)
if abs(t - datetime.utcnow()) > timedelta(minutes=15):
    return 'Signature expired.', 401
```

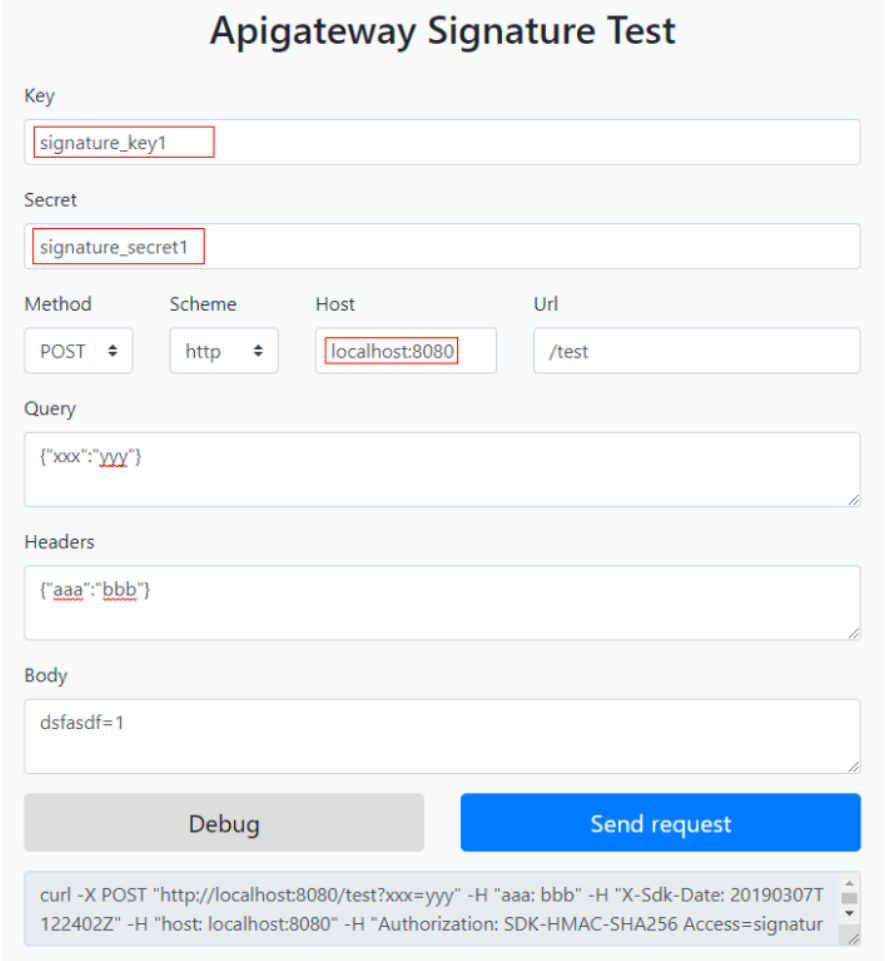
步骤7 调用verify方法校验请求签名。判断校验是否通过。

```
sig = signer.Signer()
sig.Key = signingKey
sig.Secret = signingSecret
if not sig.Verify(r, m.group(3)):
    return 'Verify authroization failed.', 401
```

步骤8 运行服务器，验证代码正确性。下面示例使用JavaScript SDK中的html签名工具生成签名。

填入如图所示字段后，单击“Send request”，复制生成的curl命令，并在命令行中执行，服务器返回200。

如果使用错误的Key和Secret访问，服务器返回401认证不通过。



----结束

1.4.3 C# SDK 使用说明

操作场景

使用C#语言进行后端服务签名时，您需要先获取SDK，然后打开工程，最后参考校验后端签名示例校验签名是否一致。

说明

C# SDK仅支持hmac类型的后端服务签名。

准备环境

- 准备待用的签名密钥的Key和Secret。
- 已在控制台创建签名密钥，并绑定API，具体请参见**配置后端服务的签名校验**。
- 已获取后端签名SDK，您可以在ROMA Connect实例控制台的“服务集成 APIC > API调用”页面中下载C# SDK。
- 获取并安装2019 version 16.8.4及以上版本的Visual Studio，如果未安装，请至**Visual Studio官方网站**下载。

打开工程

双击SDK包中的“csharp.sln”文件，打开工程。工程中包含如下3个项目：

- apigateway-signature：实现签名算法的共享库，可用于.Net Framework与.Net Core项目。
- backend-signature：后端服务签名示例，请根据实际情况修改参数后使用。具体代码说明请参考**校验后端签名示例**。
- sdk-request：签名算法的调用示例。

校验后端签名示例

说明

- 示例演示如何编写一个基于ASP.Net Core的服务器，作为API的后端，并且实现一个IAuthorizationFilter，对APIC的请求做签名校验。
- API绑定签名密钥后，发给后端的请求中才会添加签名信息。

步骤1 编写一个Controller，提供GET、POST、PUT和DELETE四个接口，且加入ApigatewaySignatureFilter的Attribute。

```
// ValuesController.cs

namespace backend_signature.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [ApigatewaySignatureFilter]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }

        // PUT api/values/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}
```

步骤2 实现一个ApigatewaySignatureFilter。将允许的签名key和secret对放入一个Dictionary中。

```
// ApigatewaySignatureFilter.cs

namespace backend_signature.Filters
{
    public class ApigatewaySignatureFilter : Attribute, IAuthorizationFilter
```



```
  {
      private Dictionary<string, string> secrets = new Dictionary<string, string>
      {
          {"signature_key1", "signature_secret1" },
          {"signature_key2", "signature_secret2" },
      };

      public void OnAuthorization(AuthorizationFilterContext context) {
          //签名校验代码
          ...
      }
  }
}
```

步骤3 OnAuthorization函数为签名校验代码。校验流程如下：使用正则表达式解析Authorization头。得到key和signedHeaders。

```
private Regex authorizationPattern = new Regex("SDK-HMAC-SHA256\\s+Access=([^,]+),\\s?
SignedHeaders=([^,]+),\\s?Signature=(\\w+)");

...

string authorization = request.Headers["Authorization"];
if (authorization == null)
{
    context.Result = new UnauthorizedResult();
    return;
}
var matches = authorizationPattern.Matches(authorization);
if (matches.Count == 0)
{
    context.Result = new UnauthorizedResult();
    return;
}
var groups = matches[0].Groups;
string key = groups[1].Value;
string[] signedHeaders = groups[2].Value.Split(';');
```

例如，Authorization头为：

```
SDK-HMAC-SHA256 Access=signature_key1, SignedHeaders=host;x-sdk-date,
Signature=e11adf65a20d1b82c25419b5********8d0ba12fed1ceb13ed00
```

则解析的结果为：

```
signingKey=signature_key1
signedHeaders=host;x-sdk-date
```

步骤4 通过key找到secret，如果不存在key，则返回认证失败。

```
if (!secrets.ContainsKey(key))
{
    context.Result = new UnauthorizedResult();
    return;
}
string secret = secrets[key];
```

步骤5 新建一个HttpRequest对象，将请求method、url、query、signedHeaders对应的请求头放入其中。判断是否需要设置body并设置。

需要读取body的条件为：不存在值为UNSIGNED-PAYLOAD的x-sdk-content-sha256头。

```
HttpRequest sdkRequest = new HttpRequest();
sdkRequest.method = request.Method;
sdkRequest.host = request.Host.Value;
sdkRequest.uri = request.Path;
Dictionary<string, string> query = new Dictionary<string, string>();
foreach (var pair in request.Query)
{
```

```
    query[pair.Key] = pair.Value;
}
sdkRequest.query = query;
WebHeaderCollection headers = new WebHeaderCollection();
string dateHeader = null;
bool needBody = true;
foreach (var h in signedHeaders)
{
    var value = request.Headers[h];
    headers[h] = value;
    if (h.ToLower() == "x-sdk-date")
    {
        dateHeader = value;
    }
    if (h.ToLower() == "x-sdk-content-sha256" && value == "UNSIGNED-PAYLOAD")
    {
        needBody = false;
    }
}
sdkRequest.headers = headers;
if (needBody)
{
    request.EnableRewind();
    using (MemoryStream ms = new MemoryStream())
    {
        request.Body.CopyTo(ms);
        sdkRequest.body = Encoding.UTF8.GetString(ms.ToArray());
    }
    request.Body.Position = 0;
}
```

步骤6 校验签名是否过期。从X-Sdk-Date头中取出时间，判断与服务器时间是否相差在15分钟以内。如果signedHeaders中不包含X-Sdk-Date，也返回认证失败。

```
private const string BasicDateFormat = "yyyyMMddTHHmmssZ";

...

if(dateHeader == null)
{
    context.Result = new UnauthorizedResult();
    return;
}
DateTime t = DateTime.ParseExact(dateHeader, BasicDateFormat, CultureInfo.CurrentCulture);
if (Math.Abs((t - DateTime.Now).Minutes) > 15)
{
    context.Result = new UnauthorizedResult();
    return;
}
```

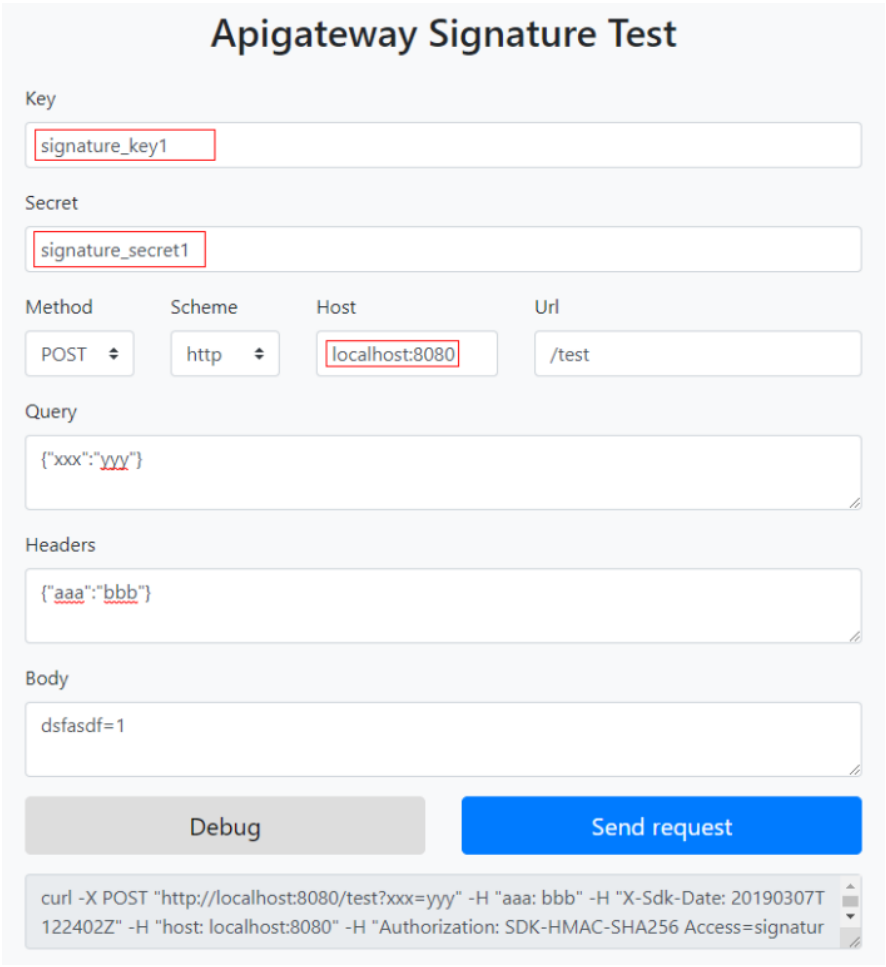
步骤7 调用verify方法校验请求签名。判断校验是否通过。

```
Signer signer = new Signer();
signer.Key = key;
signer.Secret = secret;
if (!signer.Verify(sdkRequest, groups[3].Value))
{
    context.Result = new UnauthorizedResult();
}
```

步骤8 运行服务器，验证代码正确性。下面示例使用JavaScript SDK中的html签名工具生成签名。

填入如图所示字段后，单击“Send request”，复制生成的curl命令，并在命令行中执行，服务器返回200。

如果使用错误的Key和Secret访问，服务器返回401认证不通过。



----结束

1.5 函数 API 脚本开发

1.5.1 编写函数 API 脚本（Java Script）

本节主要介绍在APIC自定义后端中开发函数API时，对于定义脚本的编写指导。

函数API通过编写函数脚本实现将多个服务封装成一个服务。

函数API使用Javascript编写函数，Javascript的运行采用java Nashorn的运行标准，支持**ECMAScript Edition 5.1**规范。Javascript引擎运行于java虚拟机，可调用自定义后端提供的**Java类**实现具体功能。

示例一 Helloworld

定义execute函数作为入口，execute函数返回的内容将作为Function API的响应Body。

```
function execute(data) {
    return "Hello world!"
}
```


示例二 获取请求参数

execute函数传入的data参数中包含了请求参数。

```
function execute(data) {
    data = JSON.parse(data)
    return {
        "method":data["method"],
        "uri":data["uri"],
        "headers":data["headers"],
        "param":data["param"],
        "body":data["body"],
    }
}
```

示例三 设置响应参数

在execute函数中返回APIConnectResponse对象，可以指定调用函数API接口返回的HTTP状态码、返回头和body体。

```
importClass(com.roma.apic.livedata.provider.v1.APIConnectResponse);
function execute(data) {
    return new APIConnectResponse(401, {"X-Type":"Demo"}, "unauthorized", false);
}
```

此时，调用Function API返回的HTTP状态码为401，响应头中包含了“X-Type: Demo”，且响应body为“unauthorized”。

示例四 调用 Java 函数

以下示例中，使用importClass引入了Java类，并调用其中的函数。所有提供的Java类可参见**函数API脚本的类说明**。

```
importClass(com.roma.apic.livedata.common.v1.Md5Utils);
function execute(data) {
    var sourceCode = "Hello world!";
    return Md5Utils.encode(sourceCode);
}
```

引用公共配置

1. 登录ROMA Connect控制台，在“实例”页面单击实例上的“查看控制台”，进入实例控制台。
2. 在左侧的导航栏选择“服务集成 APIC > 自定义后端”，在“配置管理”页面单击“添加配置”。
3. 在添加配置弹窗中配置相关信息，完成后单击“确定”。

表 1-2 公共引用配置

参数	配置说明
配置名称	填写配置的名称，根据规划自定义。
配置类型	选择配置的类型，可选择“模板变量”、“密码”和“证书”。
所属集成应用	选择配置所归属的集成应用。

参数	配置说明
配置值	**仅当“配置类型”选择“模板变量”和“密码”时需要配置。** 填写模板变量或密码的值。
确认配置值	**仅当“配置类型”选择“密码”时需要配置。** 填写密码的值，需与“配置值”保持一致。
证书	**仅当“配置类型”选择“证书”时需要配置。** 填写pem编码格式的证书内容。
私钥	**仅当“配置类型”选择“证书”时需要配置。** 填写pem编码格式的证书私钥。
密码	**仅当“配置类型”选择“证书”时需要配置。** 填写证书私钥的密码。
确认密码	**仅当“配置类型”选择“证书”时需要配置。** 填写证书私钥的密码，需与“密码”保持一致。
描述	填写配置的描述信息。

4. 在Java Script函数脚本中引用配置。

   假如配置名称为example，则引用格式如下：

   - 模板变量：#{example}
   - 密码：CipherUtils.getPlainCipherText("example")
   - 证书：CipherUtils.getPlainCertificate("example")

1.5.2 APIConnectResponse 类说明

路径

com.roma.apic.livedata.provider.v1.APIConnectResponse

说明

在execute函数中返回此类对象，可以指定调用函数API接口返回的HTTP状态码、返回头和body体。

使用示例

```
importClass(com.roma.apic.livedata.provider.v1.APIConnectResponse);
function execute(data) {
    return new APIConnectResponse(401, {"X-Type":"Demo"}, "unauthorized", false);
}
```

此时，调用Function API返回的HTTP状态码为401，响应头中包含了"X-Type: Demo"，且响应body为"unauthorized"

构造器详情

- **public APIConnectResponse(Integer statusCode)**

  构造一个APIConnectResponse

  参数：statusCode表示响应状态码

- **public APIConnectResponse(Integer statusCode, Map<String,String> headers)**

  构造一个APIConnectResponse

  参数：statusCode表示响应状态码，headers表示响应请求头

- **public APIConnectResponse(Integer statusCode, Map<String,String> headers, Object body)**

  构造一个APIConnectResponse

  参数：statusCode表示响应状态码，headers表示响应请求头，body表示响应body体

- **public APIConnectResponse(Integer statusCode, Map<String,String> headers, String body, Boolean base64Encoded)**

  构造一个APIConnectResponse

  参数：statusCode表示响应状态码，headers表示响应请求头，body表示响应body体，base64Encoded表示传入的body是否已经base64编码

方法列表

返回类型	方法和说明
Object	**getBody**() 获取响应的返回体
Map<String,String>	**getHeaders**() 获取响应的返回头
Integer	**getStatusCode**() 获取响应返回码
Boolean	**isBase64Encoded**() 获取body是否已经base64编码
void	**setBase64Encoded**(Boolean base64Encoded) 设置body是否已经base64编码
void	**setBody**(Object body) 设置响应的body体
void	**setHeaders**(Map<String,String> headers) 设置响应的返回头
void	**setStatusCode**(Integer statusCode) 设置响应的返回码

方法详情

- **public Object getBody()**
  获取响应的返回体
  返回信息
  返回响应返回体对象
- **public Map<String,String> getHeaders()**
  获取响应的返回头
  返回信息
  返回请求头的Map集合
- **public Integer getStatusCode()**
  获取响应返回码
  返回信息
  返回响应返回码
- **public Boolean isBase64Encoded()**
  获取body是否已经base64编码
  返回信息
  - true：已经进行base64编码
  - false：没有进行base64编码
- **public void setBase64Encoded(Boolean base64Encoded)**
  设置body是否已经base64编码
  输入参数
  base64Encoded：取值为true表示已经base64编码，取值为false表示没有base64编码
- **public void setBody(Object body)**
  设置响应的body体
  输入参数
  body：body体对象
- **public void setHeaders(Map<String,String> headers)**
  设置响应的返回头
  输入参数
  headers：返回头的map集合
- **public void setStatusCode(Integer statusCode)**
  设置响应返回码
  输入参数
  statusCode：返回码

1.5.3 Base64Utils 类说明

路径

com.roma.apic.livedata.common.v1.Base64Utils

说明

提供Base64Utils编码和解码功能。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.Base64Utils);
function execute(data) {
    var sourceCode = "Hello world!";
    return Base64Utils.encode(sourceCode);
}
```

方法列表

返回类型	方法和说明
static java.lang.String	**decode**(java.lang.String content) 对字符串进行Base64解码
static java.lang.String	**decodeUrlSafe**(java.lang.String content) 对字符串进行Base64解码（使用url兼容的字符集）
static java.lang.String	**encode**(byte[] content) 对字符数组进行Base64编码
static java.lang.String	**encode**(java.lang.String content) 对字符串进行Base64编码
static java.lang.String	**encodeUrlSafe**(byte[] content) 对字符数组进行Base64编码（使用url兼容的字符集）
static java.lang.String	**encodeUrlSafe**(java.lang.String content) 对字符串进行Base64编码（使用url兼容的字符集）

方法详情

- **public static java.lang.String decode(java.lang.String content)**
  对字符串进行Base64解密
  输入参数
  content：Base64加密后的字符串
  返回信息
  返回解密后的字符串
- **public static java.lang.String decodeUrlSafe(java.lang.String content)**
  对字符数组进行Base64解密（使用url兼容的字符集）
  输入参数
  content：Base64加密后的字符串

返回信息

返回解密后的字符串

- **public static java.lang.String encode(byte[] content)**

  对字符数组进行Base64加密

  输入参数

  content：待加密的字符数组

  返回信息

  返回加密后的字符串

- **public static java.lang.String encode(java.lang.String content)**

  对字符串进行Base64加密

  输入参数

  content：待加密的字符串

  返回信息

  返回加密后的字符串

- **public static java.lang.String encodeUrlSafe(byte[] content)**

  对字符数组进行Base64加密（使用url兼容的字符集）

  输入参数

  content：待加密的字符数组

  返回信息

  返回加密后的字符串

- **public static java.lang.String encodeUrlSafe(java.lang.String content)**

  对字符串进行Base64加密（使用url兼容的字符集）

  输入参数

  content：待加密的字符串

  返回信息

  返回加密后的字符串

1.5.4 CacheUtils 类说明

路径

com.huawei.livedata.lambdaservice.util.CacheUtils

说明

提供缓存的存储和查询功能。

方法列表

返回类型	方法和说明
static boolean	**putCache**(String key, String value) 存入缓存信息

返回类型	方法和说明
static boolean	**putCache**(String key, String value, int time) 存入带超时时间缓存信息
static String	**getCache**(String key) 获取缓存信息
static long	**removeCache**(String key) 移除缓存信息
static String	**get**(String key) 获取字典缓存信息

方法详情

- **public static boolean putCache(String key, String value)**
  存入缓存信息
  输入参数
  - key：缓存信息的key值。
  - value：缓存的信息。

  返回信息
  返回对应的boolean值
- **public static boolean putCache(String key, String value, int time)**
  存入带超时时间缓存信息
  输入参数
  - key：缓存信息的key值。
  - value：缓存的信息。
  - time：超时时间

  返回信息
  返回对应的boolean值
- **public static String getCache(String key)**
  获取缓存信息
  输入参数
  key：缓存信息的key值
  返回信息
  返回key值对应的缓存信息
- **public static long removeCache(String key)**
  移除缓存信息
  输入参数
  key：待移除缓存信息的key值
  返回信息
  返回执行结果

- **public static String get(String key)**
  获取字典缓存信息
  输入参数
  key：字典缓存信息的key值
  返回信息
  返回key值对应的字典缓存信息

1.5.5 CipherUtils 类说明

路径

com.huawei.livedata.lambdaservice.security.CipherUtils

说明

解密密码箱中密码的key值。

说明

在获取解密密码箱中普通密码的key值时，注意敏感信息保护，避免敏感信息泄露。

方法列表

返回类型	方法和说明
static String	**getPlainCipherText**(String key) 解密密码箱中普通密码的key值
static Response	**getPlainCertificate**(String key) 解密密码箱中证书密码的key值

方法详情

- **public static String getPlainCipherText(String key)**
  解密密码箱中普通密码的key值
  输入参数
  key：普通密码的key值
  返回信息
  返回解密后的密码
- **public static Response getPlainCertificate(String key)**
  解密密码箱中证书密码的key值
  输入参数
  key：证书密码的key值
  返回信息
  返回解密后的证书密码消息体，消息体如下格式：

```
{
"cipherType": "CERTIFICATE",
```

```
"passphrase": "xxx",
"privateKey": "xx",
"privateKey": "xx",
}
```

1.5.6 ConnectionConfig 类说明

路径

com.roma.apic.livedata.config.v1.ConnectionConfig

说明

与**RabbitMqConfig**和**RabbitMqProducer**配合使用，对RabbitMQ客户端进行连接配置。

构造器详情

public ConnectionConfig(String host, int port, String userName, String pw)

构造一个RabbitMQ客户端连接配置。

1.5.7 DataSourceClient 类说明

路径

com.roma.apic.livedata.client.v1.DataSourceClient

说明

连接数据源，执行SQL语句、存储过程或NOSQL查询语句。

使用示例

SQL数据源示例：

```
importClass(com.roma.apic.livedata.client.v1.DataSourceClient);
importClass(com.roma.apic.livedata.config.v1.DataSourceConfig);
function execute(data){
    var config = new DataSourceConfig()
    config.setType("mysql")
    config.setUrl("jdbc:mysql://127.0.0.1:3306/db?allowPublicKeyRetrieval=true")
    config.setUser("username")
    config.setPassword("password")
    var ds = new DataSourceClient(config)
    return ds.execute("SELECT * FROM person where name = ? and age = ?", "Tom", 20);
}
```

NOSQL数据源示例：

```
importClass(com.roma.apic.livedata.client.v1.DataSourceClient);
importClass(com.roma.apic.livedata.config.v1.DataSourceConfig);
function execute(data){
    var config = new DataSourceConfig()
    config.setType("redis")
    config.setUrl("127.0.0.1:6379")
    config.setPassword("password")
    var ds = new DataSourceClient(config)
    return ds.execute("GET key");
}
```

构造器详情

public DataSourceClient(DataSourceConfig config)

传入数据源配置，构造一个数据源连接器

方法列表

返回类型	方法和说明
Object	execute(String sql, Object... prepareValue) 执行SQL语句、存储过程或NOSQL查询语句

方法详情

- **public Object execute(String sql, Object... prepareValue)**

  执行SQL语句、存储过程或NOSQL查询语句

  输入参数

  - prepareValue：仅在SQL语句中生效，用于替换SQL语句中的"?"参数，可以防止SQL注入。

  返回信息

  返回语句执行结果

1.5.8 DataSourceConfig 类说明

路径

com.roma.apic.livedata.config.v1.DataSourceConfig

说明

配合DataSourceClient使用，对数据源进行配置。

构造器详情

public DataSourceConfig()

构造一个无参数的DataSourceConfig

public DataSourceConfig(String type, String url, String user, String password)

传入数据源类型，连接字符串，用户名和密码，构造一个DataSourceConfig

方法列表

返回类型	方法和说明
String	getType() 获取数据源的类型

返回类型	方法和说明
String	**getUrl**() 获取连接字符串。
String	**getUser**() 获取用户名
String	**getMaxPassword**() 获取密码
void	**setType**() 设置数据源类型，可以为"mysql","mssql","oracle","postgresql","hive","redis","mongodb"
void	**setUrl**() 设置数据源连接字符串。
void	**setUser**() 设置数据源用户名
void	**setPassword**() 设置数据源密码

方法详情

- **public String getType()**
  获取数据源的类型
  返回信息
  返回数据源类型
- **public String getUrl()**
  获取连接字符串。
  返回信息
  返回连接字符串
- **public String getUser()**
  获取用户名
  返回信息
  返回用户名
- **public String getPassword()**
  获取密码
  返回信息
  返回密码
- **public void setType(String type)**
  设置数据源类型，可以为"mysql","mssql","oracle","postgresql","hive","redis","mongodb"

输入参数

- type：类型

- **public void setUrl(String url)**

  设置数据源连接字符串。

  如果数据源类型为"mysql","mssql","oracle","postgresql","hive"，则填写jdbc连接字符串。例如："jdbc:mysql://127.0.0.1:8888/db?useUnicode=true&characterEncoding=utf8"。

  如果为数据源类型为"redis"，则格式为"127.0.0.1:6379@0"，其中，"@0"可省略，为redis数据库编号。

  如果为数据源类型为"mongodb"，则格式为"127.0.0.1:27017@db"，其中，db为数据库名称。

  输入参数

  - url：连接字符串

- **public void setUser(String user)**

  设置数据源用户名。如果为数据源类型为"redis"，则不需要填写。

  输入参数

  - user：用户名

- **public void setPassword(String password)**

  设置数据源密码。

  输入参数

  - password：密码

1.5.9 ExchangeConfig 类说明

路径

com.roma.apic.livedata.config.v1.ExchangeConfig

说明

与**RabbitMqConfig**和**RabbitMqProducer**配合使用，对交换器进行配置。

构造器详情

public ExchangeConfig(String exchange, String type, boolean durable, boolean autoDelete, boolean internal, Map<String, Object> arguments)

构造一个交换器配置。

参数：

- exchange表示交换器名称。
- type表示交换器类型。
- durable表示是否持久化，true表示持久化，false表示非持久化。
- autoDelete表示是否自动删除，true表示自动删除。自动删除的前提是至少有一个队列或者交换器与该交换器绑定，之后所有与该交换器绑定的队列或者交换器都会解绑。

- internal表示是否为内置交换器，true表示是内置的交换器。客户端程序无法直接发送消息到这个交换器，只能通过交换器路由到这个交换器。
- arguments表示其他属性。

1.5.10 HttpClient 类说明

路径

- com.roma.apic.livedata.client.v1.HttpClient
- com.huawei.livedata.lambdaservice.livedataprovider.HttpClient

说明

使用此类进行HTTP请求。

使用示例

- com.roma.apic.livedata.client.v1.HttpClient

```
importClass(com.roma.apic.livedata.client.v1.HttpClient);
importClass(com.roma.apic.livedata.provider.v1.APIConnectResponse);
function execute(data) {
    var httpClient = new HttpClient();
    var resp = httpClient.request('GET', 'http://apigdemo.exampleRegion.com/api/echo', {}, null,
'application/json');
    myHeaders = resp.headers();
    proxyHeaders = {};
    for (var key in myHeaders) {
        proxyHeaders[key] = myHeaders.get(key);
    }
    return new APIConnectResponse(resp.code(), proxyHeaders, resp.body().string(), false);
}
```

- com.huawei.livedata.lambdaservice.livedataprovider.HttpClient

```
importClass(com.huawei.livedata.lambdaservice.livedataprovider.HttpClient);
function excute(data) {
    var httpExecutor = new HttpClient();
    var obj = JSON.parse(data);
    var host = 'xx.xx.xxx.xx:xxxx';
    var headers = {
        'clientapp' : 'FunctionStage'
    };
    var params = {
        'employ_no' :'00xxxxxx'
    };
    var result = httpExecutor.callGETAPI(host,'/livews/rest/apiservice/iData/personInfo/
batch',JSON.stringify(params),JSON.stringify(headers));
    return result;
}
```

构造器详情

- com.roma.apic.livedata.client.v1.HttpClient

  public HttpClient()

  构造一个无参数的HttpClient。

  public HttpClient(HttpConfig config)

  构造一个包含**HttpConfig**配置信息的HttpClient。

  参数：config表示传入HttpClient的配置信息。

- com.huawei.livedata.lambdaservice.livedataprovider.HttpClient

public HttpClient()

构造一个无参数的HttpClient。

方法列表

- com.roma.apic.livedata.client.v1.HttpClient

返回类型	方法和说明
okhttp3.Response	**request**(**HttpConfig** config) 用于发送rest请求
okhttp3.Response	**request**(String method, String url) 通过指定请求方法、请求路径的方式发送rest请求
okhttp3.Response	**request**(String method, String url, Map<String,String> headers) 通过指定请求方法、请求路径、请求消息头的方式发送rest请求
okhttp3.Response	**request**(String method, String url, Map<String,String> headers, String body) 通过指定请求方法、请求路径、请求消息头、请求body体的方式发送rest请求
okhttp3.Response	**request**(String method, String url, Map<String,String> headers, String body, String contentType) 通过指定请求方法、请求路径、请求消息头、请求body体、contentType的方式发送rest请求

- com.huawei.livedata.lambdaservice.livedataprovider.HttpClient

返回类型	方法和说明
String	**callGETAPI**(String url) 使用get方法调用http或https服务
String	**callGETAPI**(String host, String service, String params, String header) 使用get方法调用http或https服务
Response	**get**(String url, String header) 使用get方法调用http或https服务
String	**callPostAPI**(String host, String service, String content, String header, String contentType) 使用post方法调用http或https服务
String	**callPostAPI**(String url, String header, String requestBody, String type) 使用post方法调用http或https服务

返回类型	方法和说明
Response	**post**(String url, String header, String content, String type) 使用post方法调用http或https服务
String	**callFormPost**(String url, String header, String/Map param) formdata格式调用http或https服务
Response	**callFormPost**(String url, String header, String param, FormDataMultiPart form) formdata格式调用http或https服务
String	**callDelAPI**(String url, String header, String content, String type) 使用delete方法请求http或https服务
String	**callPUTAPI**(String url, String header, String content, String type) 通过put方法调用http或https服务
String	**callPatchAPI**(String url, String header, String content, String type) 使用patch方法调用http或https服务
Response	**put**(String url, String header, String content, String type) 使用put方法调用http或https服务

方法详情

- com.roma.apic.livedata.client.v1.HttpClient
  - **public okhttp3.Response request(HttpConfig config)**
    用于发送rest请求
    输入参数
    config：传入**HttpConfig**的配置信息
    返回信息
    返回响应的消息体
  - **public okhttp3.Response request(String method, String url)**
    通过指定请求方法、请求路径的方式发送rest请求
    输入参数
    - method：请求方法
    - url：请求路径

    返回信息
    返回响应的消息体
  - **public okhttp3.Response request(String method, String url, Map<String,String> headers)**
    通过指定请求方法、请求路径、请求消息头的方式发送rest请求

输入参数

- method：请求方法
- url：请求路径
- headers：Map类型，请求消息头信息

返回信息

返回响应的消息体

- **public okhttp3.Response request(String method, String url, Map<String,String> headers, String body)**

  通过指定请求方法、请求路径、请求消息头、请求body体的方式发送rest请求

  输入参数

  - method：请求方法
  - url：请求路径
  - headers：Map类型，请求消息头信息
  - body：请求body体信息

  返回信息

  返回响应的消息体

- **public okhttp3.Response request(String method, String url, Map<String,String> headers, String body, String contentType)**

  通过指定请求方法、请求路径、请求消息头、请求body体、contentType的方式发送rest请求

  输入参数

  - method：请求方法
  - url：请求路径
  - headers：Map类型，请求消息头信息
  - body：请求body体信息
  - contentType：请求体的Content-type类型

  返回信息

  返回响应的消息体

- com.huawei.livedata.lambdaservice.livedataprovider.HttpClient
  - **public String callGETAPI(String url)**

    使用get方法调用http或https服务

    输入参数

    url：服务地址

    返回信息

    返回响应的消息体

- **public String callGETAPI(String host, String service, String params, String header)**

  使用get方法调用http或https服务

  输入参数

  - host：服务地址
  - service：服务路径
  - params：http参数信息
  - header：http头部信息

  返回信息

  返回响应的消息体

- **public Response get(String url, String header)**

  使用get方法调用http或https服务

  输入参数

  - url：服务地址
  - header：请求头信息

  返回信息

  返回响应的消息体

- **public String callPostAPI(String host, String service, String content, String header, String contentType)**

  使用post方法调用http或https服务

  输入参数

  - host：服务地址
  - service：服务路径
  - content：消息体
  - header：请求头信息
  - contentType：内容类型

  返回信息

  返回响应的消息体

- **public String callPostAPI(String url, String header, String requestBody, String type)**

  使用post方法调用http或https服务

  输入参数

  - url：服务地址
  - header：请求头信息
  - requestBody：消息体

- type：MIME类型

返回信息

返回响应的消息体

– **public Response post(String url, String header, String content, String type)**

使用post方法调用http或https服务

输入参数

- url：服务地址
- header：请求头信息
- content：消息体
- type：MIME类型

返回信息

返回响应的消息体

– **public String callFormPost(String url, String header, String/Map param)**

formdata格式调用http或https服务

输入参数

- url：服务地址
- header：请求头信息
- param：参数信息

返回信息

返回响应的消息体

– **public Response callFormPost(String url, String header, String param, FormDataMultiPart form)**

formdata格式调用http或https服务

输入参数

- url：服务地址
- header：请求头信息
- param：参数信息
- form：Body参数

返回信息

返回响应的消息体

– **public String callDelAPI(String url, String header, String content, String type)**

使用delete方法请求http或https服务

输入参数

- url：服务地址
- header：请求头信息
- content：消息体
- type：MIME类型

返回信息

返回响应的消息体

- **public String callPUTAPI(String url, String header, String content, String type)**

  使用put方法调用http或https服务

  输入参数

  - url：服务地址
  - header：请求头信息
  - content：消息体
  - type：MIME类型

  返回信息

  返回响应的消息体

- **public String callPatchAPI(String url, String header, String content, String type)**

  使用patch方法调用http或https服务

  输入参数

  - url：服务地址
  - header：请求头信息
  - content：消息体
  - type：MIME类型

  返回信息

  返回响应的消息体

- **public Response put(String url, String header, String content, String type)**

  使用put方法调用http或https服务

  输入参数

  - url：服务地址
  - header：请求头信息
  - content：消息体
  - type：MIME类型

  返回信息

返回响应的消息体

1.5.11 HttpConfig 类说明

路径

com.roma.apic.livedata.config.v1.HttpConfig

说明

配合**HttpClient**使用，对HTTP请求进行配置。

使用示例

```
importClass(com.roma.apic.livedata.client.v1.HttpClient);
importClass(com.roma.apic.livedata.config.v1.HttpConfig);
function execute(data) {
    var requestConfig = new HttpConfig();

    requestConfig.setAccessKey("071fe245-9cf6-4d75-822d-c29945a1e06a");
    requestConfig.setSecretKey("c6e52419-2270-****-****-ae7fdd01dcd5");

    requestConfig.setMethod('POST');
    requestConfig.setUrl("https://30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com/
app1");
    requestConfig.setContent("body");
    requestConfig.setContentType('application/json');

    var client = new HttpClient();
    var resp =  client.request(requestConfig);
    return resp.body().string()
}
```

构造器详情

public HttpConfig()

构造一个无参数的HttpConfig

方法列表

返回类型	方法和说明
void	**addHeaderToSign**(String headerName) 添加待签名的请求头
String	**getAccessKey**() 获取签名的AccessKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
String	**getCaCertData**() 获取Ca证书
String	**getCharset**() 获取HTTP请求编码格式

返回类型	方法和说明
String	**getClientCertData**() 获取客户端证书
String	**getClientKeyAlgo**() 获取客户端私钥加密算法
String	**getClientKeyData**() 获取客户端私钥
String	**getClientKeyPassphrase**() 获取客户端私钥密码
int	**getConnectionTimeout**() 获取连接超时时间
int	**getConnectTimeout**() 获取连接超时时间
Object	**getContent**() 获取HTTP请求内容
String	**getContentType**() 获取HTTP请求内容格式
String	**getHeader**(String name) 获取指定名称的HTTP请求头
Set<String>	**getHeaderNames**() 获取请求头名称
Map<String,String[]>	**getHeaders**() 获取所有请求头
String[]	**getHeaders**(String name) 获取指定名称的所有HTTP请求头
Set<String>	**getHeadersToSign**() 获取待签名的请求头
String	**getHttpProxy**() 获取Http代理
String	**getHttpsProxy**() 获取Https代理
int	**getMaxConcurrentRequests**() 获取maxConcurrentRequests
int	**getMaxConcurrentRequests...**() 获取maxConcurrentRequestsPerHost

返回类型	方法和说明
String	**getMethod**() 获取HTTP方法
String[]	**getNoProxy**() 获取不使用代理的IP地址列表
String	**getParameter**(String name) 获取指定名称的HTTP请求参数
Set<String>	**getParameterNames**() 获取所有HTTP请求参数名称
Map<String,String>	**getParameters**() 获取HTTP请求参数
String	**getProxyPassword**() 获取代理密码
String	**getProxyUsername**() 获取代理用户名
RequestConfig	**getRequestConfig**() 获取请求配置信息
String	**getRequestId**() 获取请求编号
int	**getRequestTimeout**() 获取请求超时时间
long	**getRollingTimeout**() 获取rolling超时时间
long	**getScaleTimeout**() 获取scale超时时间
String	**getSecretKey**() 获取请求签名的SecretKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
okhttp3.TlsVersion[]	**getTlsVersions**() 获取TLS版本
String	**getUrl**() 获取Url
String	**getUserAgent**() 获取userAgent

返回类型	方法和说明
long	**getWebsocketPingInterval**() 获取websocket心跳时间
long	**getWebsocketTimeout**() 获取websocket超时时间
boolean	**isRedirects**() 是否允许重定向
boolean	**isSsl**() 是否使用HTTPS，默认false
boolean	**isSslRedirects**() 获取sslRedirects的值，取值：true or false
boolean	**isTrustCerts**() 是否信任所有证书，取值：true or false
void	**setAccessKey**(String accessKey) 设置请求签名的AccessKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
void	**setBodyForm**(Map<String,String> content) 设置map类型的HTTP请求内容
void	**setBodyText**(String content) 设置String格式的HTTP请求内容
void	**setCaCertData**(String caCertData) 设置Ca证书
void	**setCharset**(String charset) 设置HTTP请求编码格式
void	**setClientCertData**(String clientCertData) 设置客户端证书
void	**setClientKeyAlgo**(String clientKeyAlgo) 设置客户端私钥加密算法
void	**setClientKeyData**(String clientKeyData) 设置客户端私钥
void	**setClientKeyPassphrase**(String clientKeyPassphrase) 设置客户端私钥密码
void	**setConnectionTimeout**(int connectionTimeout) 设置连接超时时间

返回类型	方法和说明
void	**setConnectTimeout**(int connectTimeout) 设置连接超时时间
void	**setContent**(Object content) 设置Object格式的HTTP请求内容
void	**setContentType**(String contentType) 设置HTTP请求内容格式
void	**setHeader**(String name, String value) 设置指定名称和值的请求头
void	**setHeader**(String name, String[] value) 设置指定名称和值的请求头
void	**setHeaders**(Map<String,String> headers) 设置请求头
void	**setHeaderValues**(Map<String,String[]> headers) 设置请求头
void	**setHttpProxy**(String httpProxy) 设置http代理
void	**setHttpsProxy**(String httpsProxy) 设置Https代理
void	**setMaxConcurrentRequests**(int maxConcurrentRequests) 设置最大并发数
void	**setMaxConcurrentRequests...**(int maxConcurrentRequestsPerHost) 设置单域名最大并发数
void	**setMethod**(String method) 设置HTTP方法
void	**setNoProxy**(String[] noProxy) 设置不使用代理的地址列表
void	**setParameter**(String name, String value) 设置HTTP请求参数
void	**setParameters**(java.util.Map<String,String> parameters) 设置HTTP请求参数
void	**setProxyPassword**(String proxyPassword) 设置proxy密码

返回类型	方法和说明
void	**setProxyUsername**(String proxyUsername) 设置proxy用户名
void	**setRedirects**(boolean redirects) 设置是否允许重定向
void	**setRequestId**(String requestId) 设置请求编号
void	**setRequestTimeout**(int requestTimeout) 设置请求超时时间
void	**setRollingTimeout**(long rollingTimeout) 设置rolling超时时间
void	**setScaleTimeout**(long scaleTimeout) 设置scale超时时间
void	**setSecretKey**(String secretKey) 设置请求签名的SecretKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
void	**setSsl**(boolean ssl) 设置是否使用HTTPS
void	**setSslRedirects**(boolean sslRedirects) 设置sslRedirects的值
void	**setTlsVersions**(okhttp3.TlsVersion[] tlsVersions) 设置TLS版本
void	**setTrustCerts**(boolean trustCerts) 设置是否信任所有证书
void	**setUrl**(String url) 设置Url
void	**setUserAgent**(String userAgent) 设置userAgent
void	**setWebsocketPingInterval**(long websocketPingInterval) 设置websocket心跳时间
void	**setWebsocketTimeout**(long websocketTimeout) 设置websocket超时时间

方法详情

- **public void addHeaderToSign(String headerName)**

  向签名中添加header

  输入参数

  headerName：请求头名称

- **public String getAccessKey()**

  获取请求签名的AccessKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。

  返回信息

  返回请求签名的AccessKey

- **public String getCaCertData()**

  获取Ca证书

  返回信息

  返回Ca证书

- **public String getCharset()**

  获取HTTP请求编码格式

  返回信息

  返回HTTP请求编码格式

- **public String getClientCertData()**

  获取客户端证书

  返回信息

  返回客户端证书

- **public String getClientKeyAlgo()**

  获取客户端私钥加密算法

  返回信息

  返回客户端私钥加密算法

- **public String getClientKeyData()**

  获取客户端私钥

  返回信息

  返回客户端私钥

- **public String getClientKeyPassphrase()**

  获取客户端私钥密码

  返回信息

  返回客户端私钥密码

- **public int getConnectionTimeout()**

  获取连接超时时间

  返回信息

  返回连接超时时间

- **public int getConnectTimeout()**

  获取连接超时时间

返回信息

返回连接超时时间

- **public Object getContent()**

  获取HTTP请求内容

  返回信息

  返回HTTP请求内容

- **public String getContentType()**

  获取HTTP请求内容的格式

  返回信息

  返回HTTP请求内容的格式

- **public String getHeader(String name)**

  获取指定名称的HTTP请求头

  输入参数

  name：请求头名称

  返回信息

  返回指定名称的请求头信息

- **public Set<String> getHeaderNames()**

  获取请求头名称

  返回信息

  返回请求头名称

- **public Map<String,String[]> getHeaders()**

  获取所有请求头

  返回信息

  返回所有请求头

- **public String[] getHeaders(String name)**

  获取指定名称的所有HTTP请求头

  输入参数

  name：请求头名称

  返回信息

  返回指定名称的所有HTTP请求头

- **public Set<String> getHeadersToSign()**

  获取签名中的请求头

  返回信息

  返回签名中的请求头

- **public String getHttpProxy()**

  获取Http代理

  返回信息

  返回Http代理

- **public String getHttpsProxy()**

  获取Https代理

返回信息

返回Https代理

- **public int getMaxConcurrentRequests()**

  获取最大并发数

  返回信息

  返回最大并发数

- **public int getMaxConcurrentRequestsPerHost()**

  获取单域名最大并发数

  返回信息

  返回单域名最大并发数

- **public String getMethod()**

  获取HTTP方法

  返回信息

  返回HTTP方法

- **public String[] getNoProxy()**

  获取不使用代理的地址列表

  返回信息

  返回不使用代理的地址列表

- **public String getParameter(String name)**

  获取指定名称的参数

  输入参数

  name：HTTP的名称

  返回信息

  返回指定名称的参数

- **public Set<String> getParameterNames()**

  获取HTTP请求参数

  返回信息

  返回HTTP请求参数

- **public Map<String,String> getParameters()**

  获取HTTP请求参数

  返回信息

  返回HTTP请求参数

- **public String getProxyPassword()**

  获取Proxy密码

  返回信息

  返回Proxy密码

- **public String getProxyUsername()**

  获取Proxy用户名

  返回信息

  返回Proxy用户名

- **public RequestConfig getRequestConfig()**
  获取requestConfig
  返回信息
  返回sslRedirects的布尔值
- **public String getRequestId()**
  获取请求编号
  返回信息
  返回请求编号
- **public int getRequestTimeout()**
  获取请求超时时间
  返回信息
  返回请求超时时间
- **public long getRollingTimeout()**
  获取rolling超时时间
  返回信息
  返回rolling超时时间
- **public long getScaleTimeout()**
  获取scale超时时间
  返回信息
  返回scale超时时间
- **public String getSecretKey()**
  获取请求签名的SecretKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
  返回信息
  返回请求签名的SecretKey
- **public okhttp3.TlsVersion[] getTlsVersions()**
  获取TLS版本号
  返回信息
  返回TLS版本号
- **public String getUrl()**
  获取Url
  返回信息
  返回Url
- **public String getUserAgent()**
  获取userAgent
  返回信息
  返回userAgent
- **public long getWebsocketPingInterval()**
  获取websocket心跳时间
  返回信息
  返回websocket心跳时间

- **public long getWebsocketTimeout()**
  获取Websocket超时时间
  返回信息
  返回Websocket超时时间
- **public boolean isRedirects()**
  是否允许重定向
  返回信息
  返回允许/不允许重定向
- **public boolean isSsl()**
  是否使用HTTPS，默认false
  返回信息
  返回允许/不允许使用HTTPS
- **public boolean isSslRedirects()**
  获取sslRedirects的值，取值为true或者false
  返回信息
  返回sslRedirects的值
- **public boolean isTrustCerts()**
  获取是否信任所有证书
  返回信息
  返回是否信任所有证书
- **public void setAccessKey(String accessKey)**
  设置请求签名的AccessKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
  输入参数
  accessKey：请求签名的AccessKey
- **public void setBodyForm(Map<String,String> content)**
  设置map类型的HTTP请求内容
  输入参数
  content：HTTP请求内容
- **public void setBodyText(String content)**
  设置String类型的HTTP请求内容
  输入参数
  content：HTTP请求内容
- **public void setCaCertData(String caCertData)**
  设置ca证书
  输入参数
  caCertData：ca证书
- **public void setCharset(String charset)**
  设置HTTP请求编码格式
  输入参数
  charset：HTTP请求编码格式

- **public void setClientCertData(String clientCertData)**
  设置客户端证书
  输入参数
  clientCertData：客户端证书
- **public void setClientKeyAlgo(String clientKeyAlgo)**
  设置客户端私钥加密算法
  输入参数
  clientKeyAlgo：客户端私钥加密算法
- **public void setClientKeyData(String clientKeyData)**
  设置客户端私钥
  输入参数
  clientKeyData：客户端私钥
- **public void setClientKeyPassphrase(String clientKeyPassphrase)**
  设置客户端私钥密码
  输入参数
  clientKeyPassphrase：客户端私钥密码
- **public void setConnectionTimeout(int connectionTimeout)**
  设置连接超时时间
  输入参数
  connectionTimeout：连接超时时间
- **public void setConnectTimeout(int connectTimeout)**
  设置连接超时时间
  输入参数
  connectTimeout：连接超时时间
- **public void setContent(Object content)**
  设置String和File类型的HTTP请求内容
  输入参数
  content：HTTP请求内容
- **public void setContentType(String contentType)**
  设置HTTP请求内容格式
  输入参数
  contentType：HTTP请求内容格式
- **public void setHeader(String name, String value)**
  设置请求头
  输入参数
  - name：请求头名称
  - value：请求头值
- **public void setHeader(String name, String[] value)**
  设置请求头
  输入参数

  - name：请求头名称
  - value：请求头值
- **public void setHeaders(Map<String,String> headers)**
  设置请求头
  输入参数
  headers：请求头信息
- **public void setHeaderValues(Map<String,String[]> headers)**
  设置请求头
  输入参数
  headers：请求头信息
- **public void setHttpProxy(String httpProxy)**
  设置http代理
  输入参数
  httpProxy：http代理
- **public void setHttpsProxy(String httpsProxy)**
  设置https代理
  输入参数
  httpsProxy：https代理
- **public void setMaxConcurrentRequests(int maxConcurrentRequests)**
  设置maxConcurrentRequests
  输入参数
  maxConcurrentRequests：
- **public void setMaxConcurrentRequestsPerHost(int maxConcurrentRequestsPerHost**
  设置maxConcurrentRequestsPerHost
  输入参数
  maxConcurrentRequestsPerHost：
- **public void setMethod(String method)**
  设置HTTP方法
  输入参数
  method：HTTP方法
- **public void setNoProxy(String[] noProxy)**
  设置不使用代理的地址列表
  输入参数
  noProxy：不使用代理的地址列表
- **public void setParameter(String name, String value)**
  设置HTTP请求参数
  输入参数
  - name：HTTP请求参数名
  - value：HTTP请求参数值

- **public void setParameters(Map<String,String> parameters)**
  设置HTTP请求参数
  输入参数
  parameters：HTTP请求参数
- **public void setProxyPassword(String proxyPassword)**
  设置proxy密码
  输入参数
  proxyPassword：proxy密码
- **public void setProxyUsername(String proxyUsername)**
  设置proxy用户名
  输入参数
  proxyUsername：proxy用户名
- **public void setRedirects(boolean redirects)**
  设置是否允许重定向
  输入参数
  redirects：是否允许重定向
- **public void setRequestId(String requestId)**
  设置请求编号
  输入参数
  requestId：请求编号
- **public void setRequestTimeout(int requestTimeout)**
  设置请求超时时间
  输入参数
  requestTimeout：请求超时时间
- **public void setRollingTimeout(long rollingTimeout)**
  设置rolling超时时间
  输入参数
  rollingTimeout：rolling超时时间
- **public void setScaleTimeout(long scaleTimeout)**
  设置scale超时时间
  输入参数
  scaleTimeout：scale超时时间
- **public void setSecretKey(String secretKey)**
  设置请求签名的SecretKey。设置了AccessKey和SecretKey的请求将使用AKSK签名算法对请求签名。
  输入参数
  secretKey：请求签名的SecretKey
- **public void setSsl(boolean ssl)**
  设置是否使用HTTPS
  输入参数
  ssl：是否使用HTTPS

- **public void setSslRedirects(boolean sslRedirects)**
  设置sslRedirects的值，取值为true或者false
  输入参数
  sslRedirects：true或者false
- **public void setTlsVersions(okhttp3.TlsVersion[] tlsVersions)**
  设置TLS版本号
  输入参数
  tlsVersions：TLS版本号
- **public void setTrustCerts(boolean trustCerts)**
  设置是否信任所有证书
  输入参数
  trustCerts：是否信任所有证书
- **public void setUrl(String url)**
  设置Url
  输入参数
  url：URL
- **public void setUserAgent(String userAgent)**
  设置userAgent
  输入参数
  userAgent：userAgent值
- **public void setWebsocketPingInterval(long websocketPingInterval)**
  设置websocket心跳时间
  输入参数
  websocket心跳时间：
- **public void setWebsocketTimeout(long websocketTimeout)**
  设置Websocket超时时间
  输入参数
  websocketTimeout：Websocket超时时间

1.5.12 JedisConfig 类说明

路径

com.roma.apic.livedata.config.v1.JedisConfig

说明

配合**RedisClient**使用，对Redis连接进行配置。

使用示例

```
importClass(com.roma.apic.livedata.client.v1.RedisClient);
importClass(com.roma.apic.livedata.config.v1.JedisConfig);
function execute(data) {
    var config = new JedisConfig();
```

```
    config.setIp(["1.1.1.1"]);
    config.setPort(["6379"]);
    config.setMode("SINGLE");
    var redisClient = new RedisClient(config);
    var count = redisClient.get("visit_count")
    if (!count)
    {
        redisClient.put("visit_count", 1);
    }else {
        redisClient.put("visit_count", parseInt(count) + 1);
    }
    return redisClient.get("visit_count");
}
```

构造器详情

public JedisConfig()

构造一个无参数的JedisConfig

方法列表

返回类型	方法和说明
int	**getDatabase**() 获取jedis的database，默认为0
String[]	**getIp**() 获取redis的IP地址列表
String	**getMaster**() 获取jedis的master名称（当mode为"MASTER_SLAVE"时有效）
int	**getMaxAttempts**() 获取jedis的重试次数，默认10000
int	**getMaxIdel**() 获取jedis连接池中空闲连接数的上限，默认5
int	**getMaxWait**() 获取jedis连接池耗尽时等待时间上限，默认60
String	**getMode**() 获取jedis的类型，可设置为“SINGLE”/“CLUSTER”/“MASTER_SLAVE”
String	**getPassPhrase**() 获取jedis的密码
String[]	**getPort**() 获取所有端口号
int	**getSoTimeout**() 获取jedis的读取超时时间，默认600

返回类型	方法和说明
int	**getTimeout**() 获取jedis的超时时间，默认1000
void	**setDatabase**(int database) 设置jedis的database
void	**setIp**(String[] ip) 设置IP地址
void	**setMaster**(String master) 设置jedis的master名称（当mode为“MASTER_SLAVE”时有效）
void	**setMaxAttempts**(int maxAttempts) 设置jedis的重试次数，默认10000
void	**setMaxIdel**(int maxIdel) 设置jedis连接池中空闲连接数的上限，默认5
void	**setMaxWait**(int maxWait) 设置jedis连接池耗尽时等待时间上限，默认60
void	**setMode**(String mode) 设置jedis的类型，可设置为“SINGLE”/“CLUSTER”/“MASTER_SLAVE”
void	**setPassPhrase**(String passPhrase) 设置jedis的密码
void	**setPort**(String[] port) 设置端口号
void	**setSoTimeout**(int soTimeout) 设置jedis的读取超时时间
void	**setTimeout**(int timeout) 设置jedis的超时时间

方法详情

- **public int getDatabase()**
  获取redis的database，默认为0
  返回信息
  返回database
- **public String[] getIp()**
  获取所有IP地址
  返回信息

返回IP地址的String数组

- **public String getMaster()**
  获取redis的master名称（当mode为“MASTER_SLAVE”时有效）
  返回信息
  返回master名称
- **public int getMaxAttempts()**
  获取redis的重试次数，默认10000
  返回信息
  返回重试次数
- **public int getMaxIdel()**
  获取jedis连接池中空闲连接数的上限，默认5
  返回信息
  返回连接池中空闲连接数的上限
- **public int getMaxWait()**
  获取jedis连接池耗尽时等待时间上限，默认60
  返回信息
  返回连接池耗尽时等待时间上限
- **public String getMode()**
  获取redis的类型，可设置为“SINGLE”/“CLUSTER”/“MASTER_SLAVE”
  返回信息
  返回redis的类型
- **public String getPassPhrase()**
  获取redis的密码
  返回信息
  返回redis的密码
- **public String[] getPort()**
  获取所有端口号
  返回信息
  返回端口号的String数组
- **public int getSoTimeout()**
  获取jedis的读取超时时间，默认600
  返回信息
  返回soTimeout
- **public int getTimeout()**
  获取jedis的超时时间，默认1000
  返回信息
  返回超时时间
- **public void setDatabase(int database)**
  设置redis的database
  输入参数

database：database

- **public void setIp(String[] ip)**
  设置ip地址
  输入参数
  ip：IP地址
- **public void setMaster(String master)**
  设置redis的master名称（当mode为“MASTER_SLAVE”时有效）
  输入参数
  master：redis的master名称
- **public void setMaxAttempts(int maxAttempts)**
  设置jedis的重试次数
  输入参数
  maxAttempts：重试次数
- **public void setMaxIdel(int maxIdel)**
  设置jedis的连接池中空闲连接数的上限，默认5
  输入参数
  maxIdel：连接池中空闲连接数的上限
- **public void setMaxWait(int maxWait)**
  设置jedis的连接池耗尽时等待时间上限，默认60
  输入参数
  maxWait：连接池耗尽时等待时间上限
- **public void setMode(String mode)**
  设置redis的类型，可设置为“SINGLE”/“CLUSTER”/“MASTER_SLAVE”
  输入参数
  mode：类型
- **public void setPassPhrase(String passPhrase)**
  设置redis的密码
  输入参数
  passPhrase：密码
- **public void setPort(String[] port)**
  设置端口号
  输入参数
  port：端口号
- **public void setSoTimeout(int soTimeout)**
  设置jedis的读取超时时间，默认600
  输入参数
  soTimeout：读取超时时间
- **public void setTimeout(int timeout)**
  设置jedis的超时时间
  输入参数

timeout：超时时间

1.5.13 JSON2XMLHelper 类说明

路径

com.huawei.livedata.util.JSON2XMLHelper

说明

提供Json与Xml之间的相互转换。

方法列表

返回类型	方法和说明
static String	**JSON2XML**(String json, boolean returnFormat) json转xml
static String	**XML2JSON**(String xml) xml转json

方法详情

- **public static String JSON2XML(String json, boolean returnFormat)**
  json转xml
  输入参数
  - json：json格式的字符串
  - returnFormat：返回格式

  返回信息
  返回xml格式字符串
- **public static String XML2JSON(String xml)**
  xml转json
  输入参数
  xml：xml格式的字符串
  返回信息
  返回xml格式字符串

1.5.14 JSONHelper 类说明

路径

com.huawei.livedata.lambdaservice.util.JSONHelper

说明

提供Json与Xml、Map之间的相互转换。

方法列表

返回类型	方法和说明
static String	**json2Xml**(String json) json转xml
static String	**xml2Json**(String xml) xml转json
static String	**json2XmlWithoutType**(String json) json转xml
static HashMap	**jsonToMap**(String json) json转map

方法详情

- **public static String json2Xml(String json)**

  json转xml

  输入参数

  json：json格式的字符串

  返回信息

  返回xml格式字符串

- **public static String xml2Json(String xml)**

  xml转json

  输入参数

  xml：xml格式的字符串

  返回信息

  返回json格式字符串

- **public static String json2XmlWithoutType(String json)**

  json转xml

  输入参数

  json：json格式的字符串

  返回信息

  返回xml格式字符串

- **public static HashMap jsonToMap(String json)**

  json转map

  输入参数

  json：json格式的字符串

  返回信息

  返回map格式字符串

1.5.15 JsonUtils 类说明

路径

com.roma.apic.livedata.common.v1.JsonUtils

说明

提供Json与对象、Xml之间的相互转换。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.JsonUtils);
function execute(data) {
    return JsonUtils.convertJsonToXml('{"a":1}')
}
```

方法列表

返回类型	方法和说明
static String	**convertJsonToXml**(String json) json转换成xml
static String	**convertJsonToXml**(String json, String rootName) json转换成xml
static <T> T	**toBean**(String json, Class<T> clazz) json转换成对象
static String	**toJson**(Object object) 将对象转换为json字符串
static String	**toJson**(Object object, Map<String,Object> config) 将对象转换为Json字符串，使用config中的配置。 例如，config中可设置“date-format”为“yyyy-MM-dd HH:mm:ss”
static Map<String,Object>	**toMap**(String json) json转换成map

方法详情

- **public static String convertJsonToXml(String json)**

  json转换成xml

  输入参数

  json：json格式的字符串

  返回信息

  返回xml格式的字符串

- **public static String convertJsonToXml(String json, String rootName)**
  json转换成xml
  输入参数
  - json：json格式的字符串
  - rootName：xml根节点名称

  返回信息
  返回xml格式的字符串
- **public static <T> T toBean(String json, Class<T> clazz)**
  json转换成对象
  输入参数
  - json ： json格式的字符串
  - clazz：类

  返回信息
  返回类对象
- **public static String toJson(Object object)**
  将对象转换为Json字符串
  输入参数
  object：输入对象
  返回信息
  转换得到的json字符串
- **public static String toJson(Object object, Map<String,Object> config)**
  将对象转换为json字符串，使用config中的配置。
  例如，config中可设置"date-format"为"yyyy-MM-dd HH:mm:ss"
  输入参数
  - object：输入对象
  - config：转换使用的配置

  返回信息
  转换得到的json字符串
- **public static Map<String,Object> toMap(String json)**
  json转换成map
  输入参数
  json：json格式的字符串
  返回信息
  map格式的字符串

1.5.16 JWTUtils 类说明

路径

com.huawei.livedata.util.JWTUtils

说明

提供sha256签名。

方法列表

返回类型	方法和说明
static String	createToken(String appId, String appKey, String timestamp) 生成sha256签名

方法详情

public static String createToken(String appId, String appKey, String timestamp)

生成sha256签名

输入参数

- appId：集成应用ID
- appKey：集成应用Key
- timestamp：时间戳

返回信息

返回sha256签名信息

1.5.17 KafkaConsumer 类说明

路径

com.roma.apic.livedata.client.v1.KafkaConsumer

说明

消费Kafka消息。

使用示例

```
importClass(com.roma.apic.livedata.client.v1.KafkaConsumer);
importClass(com.roma.apic.livedata.config.v1.KafkaConfig);

var kafka_brokers = '1.1.1.1:26330,2.2.2.2:26330'
var topic = 'YourKafkaTopic'
var group = 'YourKafkaGroupId'

function execute(data) {
    var config = KafkaConfig.getConfig(kafka_brokers, group)
    var consumer = new KafkaConsumer(config)
    var records = consumer.consume(topic, 5000, 10);
    var res = []
    var iter = records.iterator()
    while (iter.hasNext()) {
        res.push(iter.next())
    }
```

```
    return JSON.stringify(res);
}
```

构造器详情

public KafkaConsumer(Map configs)

构造一个Kafka消息消费者

参数：configs表示Kafka的配置信息

方法列表

返回类型	方法和说明
List<String>	**consume**(String topic, long timeout, long maxItems) 消费消息

方法详情

- **public List<String> consume(String topic, long timeout, long maxItems)**
  消费消息
  输入参数
  - topic：消息队列
  - timeout：读取超时时间
  - maxItems：读取消息的最大数量

  返回信息
  Kafka已消费的消息数组，消息数组即将多条消息的内容组成一个数组

1.5.18 KafkaProducer 类说明

路径

com.roma.apic.livedata.client.v1.KafkaProducer

说明

生产Kafka消息。

使用示例

```
importClass(com.roma.apic.livedata.client.v1.KafkaProducer);
importClass(com.roma.apic.livedata.config.v1.KafkaConfig);

var kafka_brokers = '1.1.1.1:26330,2.2.2.2:26330'
var topic = 'YourKafkaTopic'

function execute(data) {
    var config = KafkaConfig.getConfig(kafka_brokers, null)
    var producer = new KafkaProducer(config)
    var record = producer.produce(topic, "hello, kafka.")
    return {
        offset: record.offset(),
```

```
        partition: record.partition(),
        code: 0,
        message: "OK"
    }
}
```

构造器详情

public KafkaProducer(Map configs)

构造一个Kafka消息生产者

参数：configs表示Kafka的配置信息

方法列表

返回类型	方法和说明
org.apache.kafka.clients.producer.RecordMetadata	produce(String topic, String message) 生产消息

说明

不能直接返回方法produce(String topic, String message)，否则会导致返回信息为空。例如在使用示例中，不能直接使用“return record”句式，否则返回的信息为空。

方法详情

- **public org.apache.kafka.clients.producer.RecordMetadata produce(String topic, String message)**

  生产消息

  输入参数

  - topic：消息队列
  - message：消息内容

  返回信息

  消息记录

1.5.19 KafkaConfig 类说明

路径

com.roma.apic.livedata.config.v1.KafkaConfig

extends

java.util.Properties

说明

与KafkaProducer或KafkaConsumer配合使用，对Kafka客户端进行配置。

构造器详情

public KafkaConfig()

构造一个无参数的KafkaConfig

方法列表

返回类型	方法和说明
static KafkaConfig	getConfig(String servers, String groupId) 获取一个配置，用于访问MQS提供的Kafka（不开启sasl_ssl），具体参见MQS开发指南。
static KafkaConfig	getSaslConfig(String servers, String groupId, String username, String password) 获取一个配置，用于访问MQS提供的Kafka（开启sasl_ssl），具体参见MQS开发指南。

方法详情

- **public static KafkaConfig getConfig(String servers, String groupId)**

  访问MQS提供的kafka（不开启sasl_ssl），具体参见MQS开发指南。

  输入参数

  - servers：kafkaConfig中的bootstrap.servers信息
  - groupId：kafkaConfig中的group.id信息

  返回信息

  返回KafkaConfig对象

- **public static KafkaConfig getSaslConfig(String servers, String groupId, String username, String password)**

  访问MQS提供的kafka（开启sasl_ssl），具体参见MQS开发指南。

  输入参数

  - servers：kafkaConfig中的bootstrap.servers信息
  - groupId：kafkaConfig中的group.id信息
  - username：用户名
  - password：密码

  返回信息

  返回KafkaConfig对象

1.5.20 MD5Encoder 类说明

路径

com.huawei.livedata.lambdaservice.util.MD5Encoder

说明

计算Md5值。

方法列表

返回类型	方法和说明
static String	md5(String source) 计算字符串的Md5值

方法详情

public static String md5(String source)

计算字符串的Md5值

输入参数

source：需要计算Md5的字符串

返回信息

字符串的Md5值

1.5.21 Md5Utils 类说明

路径

com.roma.apic.livedata.common.v1.Md5Utils

说明

计算Md5值。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.Md5Utils);
function execute(data) {
    var sourceCode = "Hello world!";
    return Md5Utils.encode(sourceCode);
}
```

方法列表

返回类型	方法和说明
static String	**encode**(String content) 计算字符串的Md5值

方法详情

- **public static String encode(String content)**
  计算字符串的Md5值
  输入参数
  content：需要计算Md5的字符串
  返回信息
  字符串的Md5值

1.5.22 ObjectUtils 类说明

路径

com.roma.apic.livedata.common.v1.ObjectUtils

说明

将Java对象序列化为字节数组或从字节数组反序列化为Java对象。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.ObjectUtils);
function execute(data) {
    var sourceCode = "Hello world";
    var bytes = ObjectUtils.getBytes(sourceCode);
    return ObjectUtils.getObjectFromBytes(bytes)
}
```

方法列表

返回类型	方法和说明
static byte[]	**getBytes**(Object obj) 将对象序列化为字节数组
static Object	**getObjectFromBytes**(byte[] objBytes) 将字节数组反序列化为对象
static int	**size**(Object obj) 获取object的大小

方法详情

- **public static byte[] getBytes(Object obj)**
  将对象序列化为字节数组
  输入参数
  obj：指定的object对象
  返回信息
  返回object序列化的字节数组

- **public static Object getObjectFromBytes(byte[] objBytes)**
  将字节数组反序列化为对象
  输入参数
  objBytes：指定的字节数组
  返回信息
  返回反序列化的object对象
- **public static int size(Object obj)**
  获取object的大小
  输入参数
  obj：指定的object对象
  返回信息
  返回object的大小

1.5.23 QueueConfig 类说明

路径

com.roma.apic.livedata.config.v1.QueueConfig

说明

与**RabbitMqConfig**和**RabbitMqProducer**配合使用，对队列进行配置。

构造器详情

public QueueConfig(String queueName, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)

构造一个队列配置。

参数：

- queueName表示队列名称。
- durable表示是否持久化，true表示持久化，false表示非持久化。
- exclusive表示是否排外，true表示排外，即一个队列只能有一个消费者来消费。
- autoDelete表示是否自动删除，true表示自动删除。
- arguments表示其他属性。

1.5.24 RabbitMqConfig 类说明

路径

com.roma.apic.livedata.config.v1.RabbitMqConfig

说明

与**ConnectionConfig**，**QueueConfig**，**ExchangeConfig**和**RabbitMqProducer**配合使用，对RabbitMQ客户端进行配置。

构造器详情

public RabbitMqConfig(ConnectionConfig connectionConfig, QueueConfig queueConfig, ExchangeConfig exchangeConfig)

构造一个RabbitMQ客户端配置。

参数：

- connectionConfig表示客户端连接配置。
- queueConfig表示队列配置。
- exchangeConfig表示交换器配置。

1.5.25 RabbitMqProducer 类说明

路径

com.roma.apic.livedata.client.v1.RabbitMqProducer

说明

生产RabbitMQ消息。若发送消息没有异常，则消息发送成功；若发送消息抛出异常，则消息发送失败。

使用示例

- 用direct交换器生产消息，把消息路由到bindingKey与routingKey完全匹配的Queue中。

```
importClass(com.roma.apic.livedata.client.v1.RabbitMqProducer);
importClass(com.roma.apic.livedata.config.v1.RabbitMqConfig);
importClass(com.roma.apic.livedata.config.v1.QueueConfig);
importClass(com.roma.apic.livedata.config.v1.ExchangeConfig);
importClass(com.roma.apic.livedata.config.v1.ConnectionConfig);

function execute(data) {
    var connectionConfig = new ConnectionConfig("10.10.10.10", 5672, "admin", "123456");
    var queueConfig = new QueueConfig("directQueue", false, false, false, null);
    var exchangeConfig = new ExchangeConfig("directExchange", "direct", true, false, false, null);
    var config = new RabbitMqConfig(connectionConfig, queueConfig, exchangeConfig);

    var producer = new RabbitMqProducer(config);
    producer.produceWithDirectExchange("direct.exchange", "PERSISTENT_TEXT_PLAIN", "direct
exchange message");

    return "produce successful.";
}
```

- 用topic交换器生产消息，把消息路由到bindingKey与routingKey模糊匹配的Queue中。

```
importClass(com.roma.apic.livedata.client.v1.RabbitMqProducer);
importClass(com.roma.apic.livedata.config.v1.RabbitMqConfig);
importClass(com.roma.apic.livedata.config.v1.QueueConfig);
importClass(com.roma.apic.livedata.config.v1.ExchangeConfig);
importClass(com.roma.apic.livedata.config.v1.ConnectionConfig);

function execute(data) {
    var connectionConfig = new ConnectionConfig("10.10.10.10", 5672, "admin", "123456");
    var queueConfig = new QueueConfig(“topicQueue”, false, false, false, null);
    var exchangeConfig = new ExchangeConfig("topicExchange", "topic", true, false, false, null);
    var config = new RabbitMqConfig(connectionConfig, queueConfig, exchangeConfig);
```

```
    var producer = new RabbitMqProducer(config);
    producer.produceWithTopicExchange("topic.#", “topic.A”, null, “message”);
    return "produce successful.";
}
```

- 用fanout交换器生产消息，把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。

```
importClass(com.roma.apic.livedata.client.v1.RabbitMqProducer);
importClass(com.roma.apic.livedata.config.v1.RabbitMqConfig);
importClass(com.roma.apic.livedata.config.v1.QueueConfig);
importClass(com.roma.apic.livedata.config.v1.ExchangeConfig);
importClass(com.roma.apic.livedata.config.v1.ConnectionConfig);

function execute(data) {
    var connectionConfig = new ConnectionConfig("10.10.10.10", 5672, "admin", "123456");
    var queueConfig = new QueueConfig(“fanoutQueue”, false, false, false, null);
    var exchangeConfig = new ExchangeConfig(“fanoutExchange”, "fanout", true, false, null);
    var config = new RabbitMqConfig(connectionConfig, queueConfig, exchangeConfig);

    var producer = new RabbitMqProducer(config);
    producer.produceWithFanoutExchange( null, “message”);

    return "produce successfull"
}
```

构造器详情

public RabbitMqProducer(RabbitMqConfig rabbitMqConfig)

构造一个RabbitMQ消息生产者。

参数：rabbitMqConfig表示RabbitMQ的配置信息。

方法列表

返回类型	方法和说明
void	**produceWithDirectExchange**(String routingKey, String props, String message) 用direct交换器生产消息，把消息路由到bindingKey与routingKey完全匹配的Queue中。
void	**produceWithTopicExchange**(String bindingKey, String routingKey, String props, String message) 用topic交换器生产消息，把消息路由到bindingKey与routingKey模糊匹配的Queue中。
void	**produceWithFanoutExchange**(String props, String message) 用fanout交换器生产消息，把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。

方法详情

- **public void produceWithDirectExchange(String routingKey, String props, String message)**

  用direct交换器生产消息，把消息路由到bindingKey与routingKey完全匹配的Queue中。

输入参数

- routingKey：消息路由键
- props：消息持久化设置，非必填
- message：消息内容

- **public void produceWithTopicExchange(String bindingKey, String routingKey, String props, String message)**

  用topic交换器生产消息，把消息路由到bindingKey与routingKey模糊匹配的Queue中。

  输入参数

  - bindingKey：队列绑定键
  - routingKey：消息路由键
  - props：消息持久化设置，非必填
  - message：消息内容

- **produceWithFanoutExchange(String props, String message)**

  用fanout交换器生产消息，把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。

  输入参数

  - props：消息持久化设置，非必填
  - message：消息内容

1.5.26 RedisClient 类说明

路径

com.roma.apic.livedata.client.v1.RedisClient

说明

连接Redis设置或读取缓存（如果不指定JedisConfig，则连接到自定义后端的Function API提供的默认Redis）。

使用示例

```
importClass(com.roma.apic.livedata.client.v1.RedisClient);
function execute(data) {
    var redisClient = new RedisClient;
    var count = redisClient.get("visit_count")
    if (!count)
    {
        redisClient.put("visit_count", 1);
    }else {
        redisClient.put("visit_count", parseInt(count) + 1);
    }
    return redisClient.get("visit_count");
}
```

构造器详情

public RedisClient()

构造一个RedisClient，连接到自定义后端Function API（livedata）提供的默认Redis

public RedisClient(JedisConfig jedisConfig)

通过jedisConfig构造一个RedisClient

参数：jedisConfig

方法列表

返回类型	方法和说明
String	**get**(String key) 返回redis缓存中key对应的value值
String	**put**(String key, int expireTime, String value) 更新redis缓存内容、过期时间，返回执行结果
String	**put**(String key, String value) 更新redis缓存内容，返回执行结果
Long	**remove**(String key) 删除指定key值的缓存消息

方法详情

- **public String get(String key)**
  返回redis缓存中key对应的value值
  输入参数
  key：key值
  返回信息
  reids缓存中key对应的value值
- **public String put(String key, int expireTime, String value)**
  更新redis缓存内容、过期时间，返回执行结果
  输入参数
  - key：待更新缓存的key值
  - expireTime：待更新缓存内容的过期时间
  - value：待更新缓存的value值

  返回信息
  返回执行结果
- **public String put(String key, String value)**
  更新redis缓存内容，返回执行结果
  输入参数
  - key：待更新缓存的key值
  - value：待更新缓存的value值

  返回信息
  返回执行结果

- **public Long remove(String key)**
  删除指定key值的缓存消息
  输入参数
  key：待删除缓存的key值
  返回信息
  返回执行结果

1.5.27 RomaWebConfig 类说明

路径

com.huawei.livedata.lambdaservice.config.RomaWebConfig

说明

获取roma配置。

方法列表

返回类型	方法和说明
static String	getAppConfig(String key) 根据config key获取集成应用的配置

方法详情

public

根据config key获取配置

输入参数

key：集成应用的Key

返回信息

返回集成应用的配置

1.5.28 RSAUtils 类说明

路径

com.roma.apic.livedata.common.v1.RSAUtils

说明

提供RSA加解密方法。

使用示例

通过以下java代码生成公钥和私钥：

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64;

public class Main {

    public static void main(String[] args) {
        try {
            KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
            keyPairGenerator.initialize(1024);
            KeyPair keyPair = keyPairGenerator.generateKeyPair();
            PublicKey publicKey = keyPair.getPublic();
            System.out.println("publicKey:" + new
String(Base64.getEncoder().encode(publicKey.getEncoded())));

            PrivateKey privateKey = keyPair.getPrivate();
            System.out.println("privateKey:" + new
String(Base64.getEncoder().encode(privateKey.getEncoded())));
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
    }
}
```

将上述公钥和私钥填入到下面代码中：

```
importClass(com.roma.apic.livedata.common.v1.RSAUtils);
importClass(com.roma.apic.livedata.common.v1.Base64Utils);

function execute(data) {
    var publicKeyString = "MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDd4CRRppmYVlFl3dX4iVGN
+2Twy5gLeEPRbvhOko/xFipGF7XV0weTp4wCakgdnm+DR4gBBrQtfAuKwYIBPIr
+C1FI5sKYA3NxazDWUcXR3xlPM5D0DWjacjcMjnaj2v21WZxGpwHZHQ9TLd4OBBq3fva1r/
cE8s1Lji5QeFiklwIDAQAB";

    var privateKeyString = "**********";

    var publicKey = RSAUtils.getPublicKey(publicKeyString)
    var privateKey = RSAUtils.getPrivateKey(privateKeyString)

    var origin = "hello rsa"
    var encrypted = RSAUtils.encrypt(Base64Utils.encode(origin), publicKey)

    var decrypted = RSAUtils.decrypt(encrypted, privateKey)
    return decrypted
}
```

构造器详情

public RSAUtils()

构造一个无参数的RSAUtils

方法列表

返回类型	方法和说明
static byte[]	**decodeBase64**(String base64) BASE64字符串解码为二进制数据

返回类型	方法和说明
static byte[]	**decrypt**(java.security.PrivateKey privateKey, byte[] encryptData) RSA解密
static String	**decrypt**(String source, java.security.interfaces.RSAPrivateKey privateKey) RSA解密（source为base64编码）
static String	**decrypt**(String source, java.security.interfaces.RSAPrivateKey privateKey, Map<String, String> config) RSA解密（source为base64编码）
static String	**decrypt**(byte[] source, java.security.interfaces.RSAPrivateKey privateKey) RSA解密
static String	**decrypt**(byte[] source, java.security.interfaces.RSAPrivateKey privateKey, Map<String, String> config) RSA解密
static String	**encodeBase64**(byte[] bytes) 二进制数据编码为BASE64字符串
static byte[]	**encrypt**(java.security.PublicKey publicKey, byte[] source) RSA加密
static String	**encrypt**(String source, java.security.PublicKey publicKey) RSA加密（source为base64编码，返回加密数据为base64编码）
static String	**encrypt**(String source, java.security.PublicKey publicKey, Map<String, String> config) RSA加密（source为base64编码，返回加密数据为base64编码）
static String	**encrypt**(byte[] source, java.security.PublicKey publicKey) RSA加密（返回加密数据为base64编码）
static String	**encrypt**(byte[] source, java.security.PublicKey publicKey, Map<String, String> config) RSA加密（返回加密数据为base64编码）
static java.security.interfaces.RSAPrivateKey	**getPrivateKey**(byte[] privateKeyByte) 通过私钥字节数组创建RSA私钥

返回类型	方法和说明
static java.security.interfaces.RSAPrivateKey	**getPrivateKey**(String privateKeyByte) 通过base64编码的私钥创建RSA私钥
static java.security.interfaces.RSAPrivateKey	**getPrivateKey**(String modulus, String exponent) 通过模数和指数创建RSA私钥
static java.security.interfaces.RSAPublicKey	**getPublicKey**(byte[] publicKeyByte) 通过公钥字节数组创建RSA公钥
static java.security.interfaces.RSAPublicKey	**getPublicKey**(String publicKeyByte) 通过base64编码的公钥创建RSA公钥
static java.security.PublicKey	**getPublicKey**(String modulus, String exponent) 通过模数和指数创建RSA公钥

方法详情

- **public static byte[] decodeBase64(String base64)**

  BASE64字符串解码为二进制数据

  输入参数

  base64：base64编码的数据

  返回信息

  返回base64解码后的数据

- **public static byte[] decrypt(java.security.PrivateKey privateKey, byte[] encryptData)**

  RSA解密

  输入参数

  - privateKey：私钥
  - encryptData：待解密数据

  返回信息

  解密后的数据

- **public static String decrypt(String source, java.security.interfaces.RSAPrivateKey privateKey)**

  RSA解密

  输入参数

  - source: 待解密数据的base64编码
  - privateKey：私钥

  返回信息

解密后的数据

- **public static String decrypt(String source, java.security.interfaces.RSAPrivateKey privateKey, Map<String,String>config)**

  RSA解密

  输入参数

  - source: 带解密数据的base64编码
  - privateKey：私钥
  - config: 解密配置，配置项可以为：
    - transformation：指定解密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见**https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html#getInstance-java.lang.String-**

  返回信息

  解密后的数据

- **public static String decrypt(byte[] source, java.security.interfaces.RSAPrivateKey privateKey)**

  RSA解密

  输入参数

  - source: 带解密数据
  - privateKey：私钥

  返回信息

  解密后的数据

- **public static String decrypt(byte[] source, java.security.interfaces.RSAPrivateKey privateKey, Map<String,String>config)**

  RSA解密

  输入参数

  - source: 带解密数据
  - privateKey：私钥
  - config: 解密配置，配置项可以为：
    - transformation：指定解密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见**https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html#getInstance-java.lang.String-**

  返回信息

  解密后的数据

- **public static String encodeBase64(byte[] bytes)**

  二进制数据编码为BASE64字符串

  输入参数

  bytes：待编码数据

  返回信息

  BASE64编码

- **public static byte[] encrypt(java.security.PublicKey publicKey, byte[] source)**

  RSA加密

  输入参数

  - publicKey：公钥
  - source：需要解密的加密数据

  返回信息

  加密后的数据内容

- **public static String encrypt(String source, java.security.PublicKey publicKey)**

  RSA加密

  输入参数

  - source：待加密数据的base64编码
  - publicKey：公钥

  返回信息

  加密后的数据内容的base64编码

- **public static String encrypt(String source, java.security.PublicKey publicKey, Map<String, String> config)**

  RSA加密

  输入参数

  - source：待加密数据的base64编码
  - publicKey：公钥
  - config：加密选项，配置项可以为：
    - transformation：指定解密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见**https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html#getInstance-java.lang.String-**

- **public static String encrypt(byte[] source, java.security.PublicKey publicKey)**

  RSA加密

  输入参数

  - source：需要加密的内容
  - publicKey：公钥

  返回信息

  加密后的数据内容的base64编码

- **public static String encrypt(byte[] source, java.security.PublicKey publicKey, Map<String, String> config)**

  RSA加密

  输入参数

  - source：需要加密的内容
  - publicKey：公钥
  - config：加密选项，配置项可以为：

  - transformation：指定解密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html#getInstance-java.lang.String-

返回信息

加密后的数据内容的base64编码

- **public static java.security.interfaces.RSAPrivateKey getPrivateKey(byte[] privateKeyByte)**

  通过x509格式编码的私钥创建RSA私钥

  输入参数

  privateKeyByte：通过x509格式编码的私钥

  返回信息

  私钥

- **public static java.security.interfaces.RSAPrivateKey getPrivateKey(String privateKeyByte)**

  通过x509格式编码的私钥创建RSA私钥

  输入参数

  privateKeyByte：通过x509格式编码的私钥

  返回信息

  私钥

- **public static java.security.interfaces.RSAPrivateKey getPrivateKey(String modulus, String exponent)**

  通过模数和指数创建RSA私钥

  输入参数

  - modulus：生成私钥需要的模数
  - exponent：生成私钥需要的指数

  返回信息

  返回RSA私钥

- **public static java.security.interfaces.RSAPublicKey getPublicKey(byte[] publicKeyByte)**

  通过x509格式编码的公钥创建RSA公钥

  输入参数

  publicKeyByte：x509格式编码的公钥

  返回信息

  公钥

- **public static java.security.PublicKey getPublicKey(String modulus, String exponent)**

  通过模数和指数创建RSA公钥

  输入参数

  - modulus：生成公钥需要的模数
  - exponent：生成公钥需要的指数

  返回信息

  返回RSA公钥

1.5.29 SapRfcClient 类说明

路径

com.roma.apic.livedata.client.v1.SapRfcClient

说明

使用RFC方式访问SAP函数

使用示例

```
importClass(com.roma.apic.livedata.client.v1.SapRfcClient);
importClass(com.roma.apic.livedata.config.v1.SapRfcConfig);

function execute(data) {
    var config = new SapRfcConfig();
    config.put("jco.client.ashost", "10.95.152.107");//服务器
    config.put("jco.client.sysnr", "00"); //实例编号
    config.put("jco.client.client", "400"); //SAP集团
    config.put("jco.client.user", "SAPIDES");//SAP用户名
    config.put("jco.client.passwd", "****");//密码
    config.put("jco.client.lang", "zh");//登录语言
    config.put("jco.destination.pool_capacity", "3");//最大连接数
    config.put("jco.destination.peak_limit", "10");//最大连接线程
    var client = new SapRfcClient(config);
    var res = client.executeFunction("FUNCTION1", {
        "A":"200",
        "B":"2",
    })
    return res
}
```

构造器详情

public SapRfcClient(SapRfcConfig config)

构造一个包含**SapRfcConfig**配置信息的SapRfcClient。

参数：config表示传入SapRfcClient的配置信息。

方法列表

返回类型	方法和说明
Map<String, Object>	**executeFunction**(String functionName, Map<String, Object> params) 使用RFC方式访问SAP函数

方法详情

- **executeFunction(String functionName, Map<String, Object> params)**

  使用RFC方式访问SAP函数

  输入参数

  - functionName：函数名

- params：SAP函数的输入参数列表

返回信息

返回SAP函数的输出参数列表

1.5.30 SapRfcConfig 类说明

路径

com.roma.apic.livedata.config.v1.SapRfcConfig

extends

java.util.Properties

说明

与**SapRfcClient**配合使用，对sap客户端进行配置。

方法列表

返回类型	方法和说明
Object	**put**(String key, Object value) 设置配置参数

方法详情

- **public Object put(String key, Object value)**

  设置配置参数

  输入参数

  - key：配置信息key
  - value：配置信息value

    支持以下配置参数：

    - jco.client.ashost：SAP服务器IP
    - jco.client.sysnr：系统编号
    - jco.client.client：SAP集团
    - jco.client.user：SAP用户名
    - jco.client.passwd：密码
    - jco.client.lang：登录语言
    - jco.destination.pool_capacity：最大连接数
    - jco.destination.peak_limit：最大连接线程

- apic.async：是否为异步调用。"true"表示异步调用，默认为同步调用

返回信息

返回value

1.5.31 SoapClient 类说明

路径

com.roma.apic.livedata.client.v1.SoapClient

使用示例

```
importClass(com.roma.apic.livedata.client.v1.SoapClient);
importClass(com.roma.apic.livedata.config.v1.SoapConfig);
importClass(com.roma.apic.livedata.common.v1.XmlUtils);

function execute(data) {
    var soap = new SoapConfig();
    soap.setUrl("http://test.webservice.com/ws");
    soap.setNamespace("http://spring.io/guides/gs-producing-web-service");
    soap.setOperation("getCountryRequest");

    soap.setNamespacePrefix("ser");
    soap.setBodyPrefix("ser");
    soap.setEnvelopePrefix("soapenv");
    var content = {
        "getCountryRequest": {
            "ser:name": "Spain"
        },
    };
    soap.setContent(content);

    var client = new SoapClient(soap);
    var result = client.execute();
    var body = result.getBody();

    return XmlUtils.toJson(body);
}
```

构造器详情

SoapClient(SoapConfig soapCfg)

public SoapClient(**SoapConfig** soapCfg)：

方法列表

返回类型	方法和说明
okhttp3.Response	**request**(**HttpConfig** config) 用于发送rest请求

1.5.32 SoapConfig 类说明

路径

com.roma.apic.livedata.config.v1.SoapConfig

构造器详情

public SoapConfig()

构造一个无参数的SoapConfig

方法列表

返回类型	方法和说明
String	**buildSoapMessage**() 构造SOAP请求报文
String	**getBodyPrefix**() 获取请求报文节点前缀
String	**getCharset**() 获取HTTP请求编码格式
int	**getConnectTimeout**() 获取连接超时时间
Object	**getContent**() 获取请求内容
String	**getContentType**() 获取报文参数类型
String	**getEnvelopePrefix**() 获取信封前缀
String	**getHeader**(String name) 通过请求头名获取对应的请求头值
Map<String,String>	**getHeaders**() 获取请求头信息
String	**getMethod**() 获取请求方法
String	**getNamespace**() 获取命名空间
String	**getNamespacePrefix**() 获取命名空间前缀

返回类型	方法和说明
String	**getOperation**() 获取操作名称
String	**getParameter**(String name) 通过指定名称获取SOAP请求参数
Map<String,String>	**getParameters**() 获取SOAP请求参数
String	**getProtocol**() 获取请求协议
int	**getReadTimeout**() 获取读取超时时间
String	**getSoapAction**() 获取操作请求地址
String	**getUrl**() 获取请求地址
boolean	**isRedirects**() 是否允许重定向
void	**setBodyPrefix**(String bodyPrefix) 设置请求报文节点前缀
void	**setCharset**(String charset) 设置HTTP请求编码格式
void	**setConnectTimeout**(int connectTimeout) 设置连接超时时间
void	**setContent**(Object content) 设置请求内容
void	**setContentType**(String contentType) 设置报文参数类型
void	**setEnvelopePrefix**(String envelopePrefix) 设置信封前缀
void	**setHeader**(String name, String value) 设置请求头信息
void	**setHeaders**(Map<String,String> headers) 设置请求头信息
void	**setMethod**(String method) 设置请求方法

返回类型	方法和说明
void	**setNamespace**(String namespace) 设置命名空间
void	**setNamespacePrefix**(String namespacePrefix) 设置命名空间前缀
void	**setOperation**(String operation) 设置操作名称
void	**setParameter**(String name, String value) 设置SOAP请求参数
void	**setParameters**(Map<String,String> parameters) 设置SOAP请求参数
void	**setProtocol**(String protocol) 设置请求协议
void	**setReadTimeout**(int readTimeout) 设置读取超时时间
void	**setRedirects**(boolean redirects) 设置是否重定向
void	**setSoapAction**(String soapAction) 设置操作请求地址
void	**setUrl**(String url) 设置请求地址

方法详情

- **public String buildSoapMessage()**
  构造SOAP请求报文
  返回信息
  返回SOAP请求报文
- **public String getBodyPrefix()**
  获取请求报文节点前缀
  返回信息
  返回请求报文节点前缀bodyPrefix
- **public String getCharset()**
  获取HTTP请求编码格式
  返回信息
  返回HTTP请求编码格式
- **public int getConnectTimeout()**

获取连接超时时间

返回信息

返回连接超时时间

- **public Object getContent()**
  获取请求内容
  返回信息
  返回请求内容
- **public String getContentType()**
  获取报文参数类型
  返回信息
  返回报文参数类型
- **public String getEnvelopePrefix()**
  获取信封前缀
  返回信息
  返回信封前缀
- **public String getHeader(String name)**
  通过请求头名称获取对应的请求头值
  输入参数
  name：请求头名称
  返回信息
  请求头名称对应的请求头值
- **public Map<String,String> getHeaders()**
  获取请求头信息
  返回信息
  返回请求头信息
- **public String getMethod()**
  获取请求方法
  返回信息
  返回请求方法
- **public String getNamespace()**
  获取命名空间
  返回信息
  返回命名空间
- **public String getNamespacePrefix()**
  获取命名空间前缀
  返回信息
  返回命名空间前缀
- **public String getOperation()**
  获取操作名称
  返回信息

返回操作名称

- **public String getParameter(String name)**
  通过指定名称获取SOAP请求参数
  输入参数
  name：SOAP请求参数的名称
  返回信息
  返回SOAP请求参数
- **public Map<String,String> getParameters()**
  获取SOAP请求参数
  返回信息
  返回SOAP请求参数
- **public String getProtocol()**
  获取请求协议
  返回信息
  返回请求协议
- **public int getReadTimeout()**
  获取读取超时时间
  返回信息
  返回读取超时时间
- **public String getSoapAction()**
  获取操作请求地址
  返回信息
  返回操作请求地址
- **public String getUrl()**
  获取请求地址
  返回信息
  返回请求地址
- **public boolean isRedirects()**
  是否允许重定向
  返回信息
  返回允许/不允许重定向
- **public void setBodyPrefix(String bodyPrefix)**
  设置请求报文节点前缀
  输入参数
  bodyPrefix：请求报文节点前缀
- **public void setCharset(String charset)**
  设置HTTP请求编码格式
  输入参数
  charset：HTTP请求编码格式
- **public void setConnectTimeout(int connectTimeout)**

设置连接超时时间

输入参数

connectTimeout：连接超时时间

- **public void setContent(Object content)**
  设置请求内容
  输入参数
  content：请求内容
- **public void setContentType(String contentType)**
  设置报文参数类型
  输入参数
  contentType：报文参数类型
- **public void setEnvelopePrefix(String envelopePrefix)**
  设置信封前缀
  输入参数
  envelopePrefix：信封前缀
- **public void setHeader(String name, String value)**
  设置请求头信息
  输入参数
  - name：请求头名称
  - value：请求头值
- **public void setHeaders(Map<String,String> headers)**
  设置请求头信息
  输入参数
  headers：请求头信息
- **public void setMethod(String method)**
  设置请求方法
  输入参数
  method：请求方法
- **public void setNamespace(String namespace)**
  设置命名空间
  输入参数
  namespace：命名空间
- **public void setNamespacePrefix(String namespacePrefix)**
  设置命名空间前缀
  输入参数
  namespacePrefix：命名空间前缀
- **public void setOperation(String operation)**
  设置操作名称
  输入参数
  operation：操作名称

- **public void setParameter(String name, String value)**
  设置SOAP请求参数
  输入参数
  - name：SOAP请求参数的名称
  - value：SOAP请求参数的值
- **public void setParameters(Map<String,String> parameters)**
  设置SOAP请求参数
  输入参数
  parameters：SOAP请求参数
- **public void setProtocol(String protocol)**
  设置请求协议
  输入参数
  protocol：请求协议
- **public void setReadTimeout(int readTimeout)**
  设置读取超时时间
  输入参数
  readTimeout：读取超时时间
- **public void setRedirects(boolean redirects)**
  设置是否重定向
  输入参数
  redirects：是否重定向
- **public void setSoapAction(String soapAction)**
  设置操作请求地址
  输入参数
  soapAction：操作请求地址
- **public void setUrl(String url)**
  设置请求地址
  输入参数
  url：请求地址

1.5.33 StringUtils 类说明

路径

com.roma.apic.livedata.common.v1.StringUtils

说明

提供字符串转换功能。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.StringUtils);
function execute(data){
```

```
    return StringUtils.toString([97,96,95,94,93,92], "UTF-8")
}
```

方法列表

返回类型	方法和说明
static String	**toString**(byte[] bytes, String encoding) 将字节数组转换为字符串
static String	**toString**(byte[] bytes) 将字节数组转换成UTF-8编码字符串
static String	**toHexString**(byte[] data) 将字节数组转成十六进制小写字符串
static byte[]	**hexToByteArray**(String hex) 将十六进制字符串转换成字节数组

方法详情

- **public static String toString(byte[] bytes, String encoding)**
  将字节数组转换为字符串
  输入参数
  - bytes：需要转换的字节数组
  - encoding：编码

  返回信息
  返回转换后的字符串
- **public static String toString(byte[] bytes)**
  将字节数组转换成UTF-8编码字符串
  输入参数
  bytes：需要转换的字节数组
  返回信息
  返回转换后的字符串
- **public static String toHexString(byte[] data)**
  将字节数组转成十六进制小写字符串
  输入参数
  data：需要转换的字节数组
  返回信息
  返回转换后的十六进制字符串
- **public static byte[] hexToByteArray(String hex)**
  将十六进制字符串转换成字节数组
  输入参数
  hex：需要转换的十六进制字符串
  返回信息

返回转换后的字节数组

1.5.34 TextUtils 类说明

路径

com.roma.apic.livedata.common.v1.TextUtils

说明

提供格式化等功能

方法列表

返回类型	方法和说明
static Map<String,String>	**encodeByUrlEncoder**(Map<String,String> map) 对map中的key、value进行url编码
static boolean	**parseBoolean**(String value, boolean defaultValue) 字符串转换成对应的boolean类型
static String	**toHttpParameters**(Map<String,String> map) 将map内容转换成http url parameters

方法详情

- **public static Map<String,String> encodeByUrlEncoder(Map<String,String> map)**
  对映射表中的key、value进行url编码
  输入参数
  map：含有url参数的map
  返回信息
  返回url编码后的map
- **public static boolean parseBoolean(String value, boolean defaultValue)**
  字符串转换成对应的boolean类型
  输入参数
  - value：需要转换的字符内容
  - defaultValue：默认的boolean值，在value值非法情况下，返回

  返回信息
  返回对应的boolean值
- **public static String toHttpParameters(Map<String,String> map)**
  将映射表内容转换成http请求url中parameters字符串
  输入参数
  map：含有url参数的map

返回信息

返回http请求url中parameters字符串

1.5.35 XmlUtils 类说明

路径

com.roma.apic.livedata.common.v1.XmlUtils

说明

提供Xml转换功能。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.XmlUtils);
function execute(data) {
    var xml = '<a><id>2</id><name>1</name></a>'
    return XmlUtils.toMap(xml)
}
```

方法列表

返回类型	方法和说明
static String	**toJson**(String xml) 将xml格式的字符串转化成json格式
static Map<String,Object>	**toMap**(String xml) xml转化成Map
static String	**toXml**(Object object) object转化成xml
static String	**toXml**(Object object, Map<String,Object> config) object转化成xml

方法详情

- **public static String toJson(String xml)**

  将xml格式的字符串转化成json格式

  输入参数

  xml：xml格式的字符串

  返回信息

  返回json格式的字符串

- **public static Map<String,Object> toMap(String xml)**

  xml转化成Map

  输入参数

  xml：xml格式的字符串

返回信息

返回map格式的字符串

- **public static String toXml(Object object)**

  object转化成xml

  输入参数

  object：待转换的对象

  返回信息

  返回xml格式的字符串

- **public static String toXml(Object object, Map<String,Object> config)**

  object转换成xml

  输入参数

  - object：待转换的对象
  - config：转换配置

  返回信息

  返回xml格式的字符串

1.6 数据 API 执行语句开发

SQL 语法

数据API与各数据库的SQL语法差异：

- 如果需要把后端服务请求中携带的参数传递给SQL，使用**${参数名}**的方式传递，其中String类型的参数需要用单引号括起来，int类型的参数则不需要。

  如以下示例，name为String类型参数，id为int类型参数。

  ```
  select * from table01 where name='${name}' and id=${id}
  ```

- 参数可以在后端服务请求的Headers、Parameters或者Body中传递。
- 如果SQL中的字符串含关键字，需要对字符串转义。

  如某个字段名为delete，则SQL需要按如下格式写：

  ```
  select `delete` from table01
  ```

  说明

  若在SQL语句中同时引用多种数据类型的后端请求参数时，系统会默认把传入的参数转换为String类型。因此在执行SQL语句时，需要调用相应的函数对非String类型参数做数据类型转换。

  以上面的name（String类型）和id（int类型）参数为例，在同时传入SQL语句时，id参数会被转换为String类型，需要在SQL语句中，使用转换函数把id参数再转换回int类型。此处以cast()函数为例，不同数据库使用的转换函数会有所不同。

  ```
  select * from table01 where name='${name}' and id=cast('${id}' as int)
  ```

SQL查询样例（update、insert等命令类似）：

- 全量查询

  ```
  select * from table01;
  ```

- 带参数查询

  指把后端服务请求中携带的参数（Headers、Parameters或者Body参数）传递给SQL，为SQL语句提供灵活的条件查询或数据处理能力。

  - GET、DELETE方法的API，从请求URL中获取参数。
  - POST、PUT方法的API，从Body中获取参数。**注意：**Body体为application/x-www-form-urlencoded格式。

```
select * from table01 where 1=1 and col01 = ${param01};
```

- 可选参数查询

```
select * from table01 where 1=1 [and col01 = ${param01}] [and col02 = ${param02}]
```

- IN查询

```
select * from table01 where 1=1 and col01 in ('${param01}','${param02}');
```

- UNION查询

  默认删除重复数据，如需返回全部数据，使用关键字：union all

```
select * from table01
union [all | distinct]
select * from table02;
```

- 嵌套查询

```
select * from table01 where 1=1 and col01 in (select col02 from table02 where col03 is not null);
```

NoSQL（MongoDB、Redis等）兼容的源生命令：

- Redis数据源支持的命令：

  GET、HGET、HGETALL、LRANGE、SMEMBERS、ZRANGE、ZREVRANGE、SET、LPUSH、SADD、ZADD、HMSET、DEL

- MongoDB数据源支持的命令：

  find

NoSQL样例：

- 插入String类型的key，value从请求参数中获取。

```
set hello ${parm01}
```

- 查询String类型的key

```
get hello
```

存储过程调用

当前数据API不支持直接创建存储过程，但是可以执行MySQL、Oracle、PostgreSQL这三种数据源的存储过程，以Oracle数据库为例说明。

- **数据源说明**

  假设数据库里面有一张表，表结构如下建表语句所示：

```
create table sp_test(id number,name varchar2(50),sal number);
```

  往表中并插入数据，数据集如下表所示：

  表 1-3 sp_test 表数据集

ID	NAME	SAL
1	ZHANG	5000
2	LI	6000
3	ZHAO	7000
4	WANG	8000

在Oracle数据库中调用存储过程，根据name查询sal的值。

```
create or replace procedure APICTEST.sb_test(nname in varchar, nsal out number) as
begin
   select sal into nsal from sp_test where name = nname;
end;
```

- **数据API中的执行语句说明**

  数据API调用存储过程时，参数可通过后端服务请求的Headers、Parameters或者Body传递，参数名的语法为：**{参数名}.{数据类型}.{传输类型}**。

  – 数据类型包括String和int。

  – 传输类型指入参或出参声明，入参使用**in**，出参使用**out**。

  数据API中调用存储过程的执行语句示例：

  ```
  call sb_test(${nname.String.in},${nsal.int.out})
  ```

  该脚本示例中，nname为字符串类型的入参，参数名为nname.String.in ，value则是你要查询的参数值。nsal为数值类型的出参，参数名为nsal.int.out，由于格式限定，出参的value也需要填写，可填写符合数据类型的任意值，不影响输出结果。

  说明

  - 数据API中对存储过程的调用，用String和int来区分字符串和数值，无需加单引号，这一点和SQL要求不一样。
  - 在后端服务的Headers、Parameters或者Body中定义的参数名不能相同，否则将被覆盖。

  – Body传递参数示例：

  后端服务请求的Body内容

  ```
  {
   "nname.String.in": "zhang",
   "nsal": 0
  }
  ```

  响应结果

  ```
  {
   "test": [
    5000
   ]
  }
  ```

  – Parameters传递参数示例：

  后端服务请求的Parameters内容

  ```
  https://example.com?nname.String.in=zhang&nsal=0
  ```

  响应结果

  ```
  {
   "test": [
    5000
   ]
  }
  ```

多个数据源编排

一个数据API可以包含多个数据源，因此一次API请求可以涉及多个数据源，例如取第一个数据源查询结果作为第二个数据源的参数。

以MySQL为例说明，假设数据API有数据源1和数据源2，user01是数据源1的数据表和user02是数据源2的数据表，两张表的结构如下：

表 1-4 表结构

数据源	表名	参数
数据源1	user01	• id（int） • name（varchar）
数据源2	user02	• user_id（int） • user_name（varchar） • user_age（int） • user_sex（varchar）

数据源SQL设计如下：

数据源1，在表user01中查找name为“zhang”的数据记录id。

```
select id from user01 where name='zhang';
```

数据源2，根据user01中找到的id，在user02中找到对应的数据记录，并把该记录的user_name的值更新为“zhang”。

```
update user02 set user_name='zhang' where user_id=${result1[0].id};
```

可选参数的使用

数据API中使用中括号[]标记可选参数，例如以下SQL执行语句：

```
select * from table01 where id=${id} [or sex='${sex}']
```

用[]括起来的那部分语句，表示当后端服务请求中携带参数${sex}时候，才会生效；不带参数${sex}的时候，[]括起来的语句在执行时将被忽略。

- 后端服务请求携带了参数id=88，可选参数sex没有携带，则执行SQL语句：
  ```
  select * from table01 where id=88;
  ```
- 后端服务请求携带了参数id=88和sex=female，则执行SQL语句：
  ```
  select * from table01 where id=88 or sex='female';
  ```

2 消息集成开发指导

2.1 概述与网络环境准备

本文概述

ROMA MQS完全兼容开源Kafka协议，可以直接使用**kafka开源客户端**连接。如果使用SASL认证方式，则在开源客户端基础上使用云服务提供的证书文件。

本指南主要介绍MQS连接信息的收集，如获取MQS连接地址与端口、SASL连接使用的证书、公网访问信息等，然后提供Java、Python等语言的连接示例。

本指南的示例仅展示Kafka的API调用，生产与消费的API集，请参考**Kafka官网**。

客户端网络环境说明

客户端有3种方式访问MQS：

- VPC内子网地址访问

  如果客户端是云上ECS，与MQS处于同region同VPC，则可以直接访问MQS提供的**VPC内子网地址**。

- VPC对等连接方式访问

  如果客户端是云上ECS，与MQS处于相同region但不同VPC，则可以通过建立VPC对等连接后，访问MQS提供的VPC内子网地址。

- 公网访问

  客户端在其他网络环境，或者与ROMA MQS处于不同region，则访问实例的公网地址。

  公网访问时，需要**修改ROMA Connect实例绑定的安全组，增加入方向规则，允许端口9094、9095、9096被外部网络访问**。

说明

不同网络环境，对于客户端配置来说，只是连接地址的差异，其他都一样。因此，本手册以同一VPC内子网地址的方式，介绍客户端开发环境搭建。

遇到连接超时或失败时，请注意确认网络是否连通。可使用telnet方式，检测实例连接地址与端口。

2.2 收集连接信息

MQS 信息准备

- 实例连接地址与端口

  在实例控制台的“实例信息”页面中，选择“基本信息”页签，查看MQS连接地址。

  - 消息集成 MQS内网连接地址：使用Kafka客户端连接MQS，且客户端与MQS内网互通时使用。
  - 消息集成 MQS公网连接地址：使用Kafka客户端连接MQS，且客户端与MQS公网互通时使用
  - 消息RESTful API：使用RESTful API连接MQS时使用。

- Topic名称

  在ROMA Connect实例控制台的“消息集成 MQS > Topic管理”页面，查看Topic名称。

- SASL认证信息

  若ROMA Connect实例启用了“MQS SASL_SSL”，则需要获取用户名、密码与客户端证书。

  - 用户名和密码

    在ROMA Connect实例控制台的“集成应用”页面，单击Topic所属的集成应用名称，在集成应用概览页的基本信息中，可查看Key和Secret，即为用户名和密码。

  - 客户端证书

    在ROMA Connect实例控制台的“消息集成 MQS > Topic管理”页面，单击“下载SSL证书”下载客户端证书。

2.3 使用客户端连接 MQS

2.3.1 客户端使用建议

Consumer 使用建议

- Consumer的owner线程需确保不会异常退出，避免客户端无法发起消费请求，阻塞消费。
- 确保处理完消息后再做消息commit，避免业务消息处理失败，无法重新拉取处理失败的消息。
- Consumer不能频繁加入和退出group，频繁加入和退出，会导致Consumer频繁做rebalance，阻塞消费。
- Consumer数量不能超过topic分区数，否则会有Consumer拉取不到消息。
- Consumer需周期poll，维持和server的心跳，避免心跳超时，导致Consumer频繁加入和退出，阻塞消费。
- Consumer拉取的消息本地缓存应有大小限制，避免OOM（Out of Memory）。

- Consumer session的超时时间设置为30秒，session.timeout.ms=30000，避免时间过短导致Consumer频繁超时做rebalance，阻塞消费。
- ROMA Connect不能保证消费重复的消息，业务侧需保证消息处理的幂等性。
- 消费线程退出要调用Consumer的close方法，避免同一个组的其他消费者阻塞sesstion.timeout.ms的时间。

Producer 使用建议

- 同步复制客户端需要配合使用：acks=all
- 配置发送失败重试：retries=3
- 发送优化：linger.ms=0
- 生产端的JVM内存要足够，避免内存不足导致发送阻塞

Topic 使用建议

- 配置要求：推荐3副本，同步复制，最小同步副本数为2，且同步副本数不能等于Topic副本数，否则宕机1个副本会导致无法生产消息。
- 创建方式：支持选择是否开启kafka自动创建Topic的开关。选择开启后，表示生产或消费一个未创建的Topic时，会自动创建一个包含3个分区和3个副本的Topic。
- 单Topic最大分区数建议为20。
- Topic副本数为3（当前版本限制，不可调整）。

其他建议

- 连接数限制：3000
- 消息大小：不能超过10MB
- 使用sasl_ssl协议访问ROMA Connect：确保DNS具有反向解析能力，或者在hosts文件配置ROMA Connect所有节点ip和主机名映射，避免Kafka client做反向解析，阻塞连接建立。
- 磁盘容量申请超过业务量 * 副本数的2倍，即保留磁盘空闲50%左右。
- 业务进程JVM内存使用确保无频繁FGC，否则会阻塞消息的生产和消费。

说明

- 若ROMA Connect实例的消息集成在开启SASL_SSL的同时，也开启了VPC内网明文访问，则VPC内无法使用SASL方式连接消息集成的Topic。
- 使用SASL方式连接消息集成的Topic时，建议在客户端所在主机的“/etc/hosts”文件中配置host和IP的映射关系，否则会引入时延。

  其中，IP地址必须为消息集成的连接地址，host为每个实例主机的名称，可以自定义，但不能重复。例如：

  10.10.10.11 host01

  10.10.10.12 host02

  10.10.10.13 host03

2.3.2 客户端参数配置建议

Kafka客户端的配置参数很多，以下提供Producer和Consumer几个常用参数配置。

表 2-1 Producer 参数

参数	默认值	推荐值	说明
acks	1	高可靠：all 高吞吐：1	收到Server端确认信号个数，表示procuder需要收到多少个这样的确认信号，算消息发送成功。acks参数代表了数据备份的可用性。常用选项： ● acks=**0**：表示producer不需要等待任何确认收到的信息，副本将立即加到socket buffer并认为已经发送。没有任何保障可以保证此种情况下server已经成功接收数据，同时重试配置不会发生作用（因为客户端不知道是否失败），回馈的offset会总是设置为-1。 ● acks=**1**：这意味着至少要等待leader已经成功将数据写入本地log，但是并没有等待所有follower是否成功写入。如果follower没有成功备份数据，而此时leader又无法提供服务，则消息会丢失。 ● acks=**all**：这意味着leader需要等待所有备份都成功写入日志，只有任何一个备份存活，数据都不会丢失。
retries	0	结合实际业务调整	客户端发送消息的重试次数。值大于0时，这些数据发送失败后，客户端会重新发送。 注意，这些重试与客户端接收到发送错误时的重试没有什么不同。允许重试将潜在的改变数据的顺序，如果这两个消息记录都是发送到同一个partition，则第一个消息失败第二个发送成功，则第二条消息会比第一条消息出现要早。
request.timeout.ms	30000	结合实际业务调整	设置一个请求最大等待时间，超过这个时间则会抛Timeout异常。 超时时间如果设置大一些，如120000（120秒），高并发的场景中，能减少发送失败的情况。
block.on.buffer.full	TRUE	TRUE	TRUE表示当我们内存用尽时，停止接收新消息记录或者抛出错误。 默认情况下，这个设置为TRUE。然而某些阻塞可能不值得期待，因此立即抛出错误更好。如果设置为false，则producer抛出一个异常错误：BufferExhaustedException

参数	默认值	推荐值	说明
batch.size	16384	262144	默认的批量处理消息字节数上限。producer将试图批处理消息记录，以减少请求次数。这将改善client与server之间的性能。不会试图处理大于这个字节数的消息字节数。 发送到brokers的请求将包含多个批量处理，其中会包含对每个partition的一个请求。 较小的批量处理数值比较少用，并且可能降低吞吐量（0则会仅用批量处理）。较大的批量处理数值将会浪费更多内存空间，这样就需要分配特定批量处理数值的内存大小。
buffer.memory	33554432	67108864	producer可以用来缓存数据的内存大小。如果数据产生速度大于向broker发送的速度，producer会阻塞或者抛出异常，以“block.on.buffer.full”来表明。 这项设置将和producer能够使用的总内存相关，但并不是一个硬性的限制，因为不是producer使用的所有内存都是用于缓存。一些额外的内存会用于压缩（如果引入压缩机制），同样还有一些用于维护请求。

表 2-2 Consumer 参数

参数	默认值	推荐值	说明
auto.commit.enable	TRUE	FALSE	如果为TRUE，consumer所fetch的消息的offset将会自动的同步到zookeeper。这项提交的offset将在进程无法提供服务时，由新的consumer使用。 约束： 设置为FALSE后，需要先成功消费再提交，这样可以避免消息丢失。
auto.offset.reset	latest	earliest	没有初始化offset或者offset被删除时，可以设置以下值： earliest：自动复位offset为最早 latest：自动复位offset为最新 none：如果没有发现offset则向消费者抛出异常 anything else：向消费者抛出异常。
connections.max.idle.ms	600000	30000	空连接的超时时间，设置为30000可以在网络异常场景下减少请求卡顿的时间。

2.3.3 Java 开发环境搭建

基于**收集连接信息**的介绍，假设您已经获取了实例连接相关的信息，以及配置好客户端的网络环境。本章节以生产与发送消息的Demo为例，介绍Kafka客户端的环境配置。

开发环境准备

- Maven：

  获取并安装Apache Maven 3.0.3及以上版本，可至**Maven官方下载页面**下载。
- JDK：

  获取并安装Java Development Kit 1.8.111及以上版本，可至**Oracle官方下载页面**下载。

  安装后注意配置Java的环境变量。
- 获取并安装2018.3.5或以上版本的IntelliJ IDEA，可至**IntelliJ IDEA官方网站**下载。

操作步骤

步骤1 下载Demo包。

在ROMA Connect实例控制台的“消息集成 MQS > Topic管理”页面，单击右上角的“用户指南 > 下载Kafka客户端 Java Demo包”下载Demo。

解压后，有如下文件：

表 2-3 Kafka Demo 文件清单

文件名	路径	说明
MqsConsumer.java	.\src\main\java\com\mqs\consumer	消费消息的API。
MqsProducer.java	.\src\main\java\com\mqs\producer	生产消息的API。
mqs.sdk.consumer.properties	.\src\main\resources	消费消息的配置信息。
mqs.sdk.producer.properties	.\src\main\resources	生产消息的配置信息。
client.truststore.jks	.\src\main\resources	SSL证书，用于SASL方式连接。
MqsConsumerTest.java	.\src\test\java\com\mqs\consumer	消费消息的测试代码。
MqsProducerTest.java	.\src\test\java\com\mqs\producer	生产消息的测试代码。
pom.xml	.\	maven配置文件，包含Kafka客户端引用。

步骤2 打开IntelliJ IDEA，导入Demo。

Demo是一个Maven构建的Java工程，因此需要配置JDK环境，以及IDEA的Maven插件。

图 2-1 选择“导入工程”

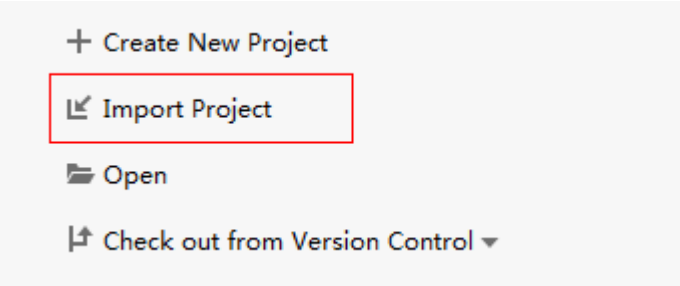


图 2-2 选择“Maven”

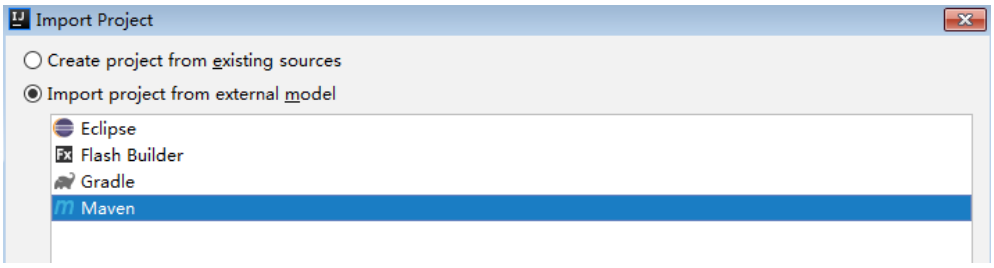
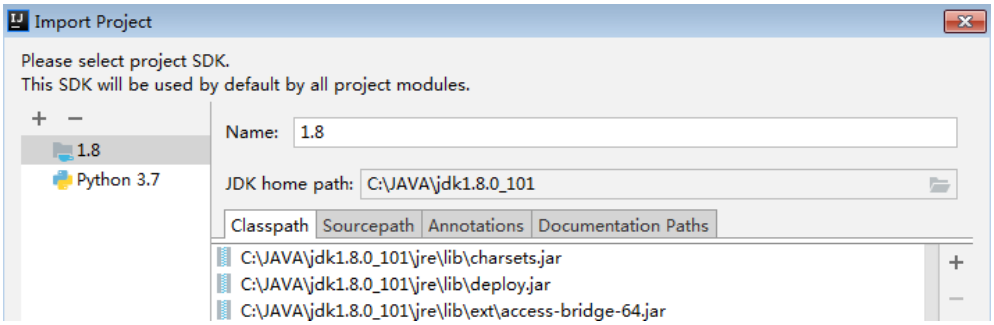
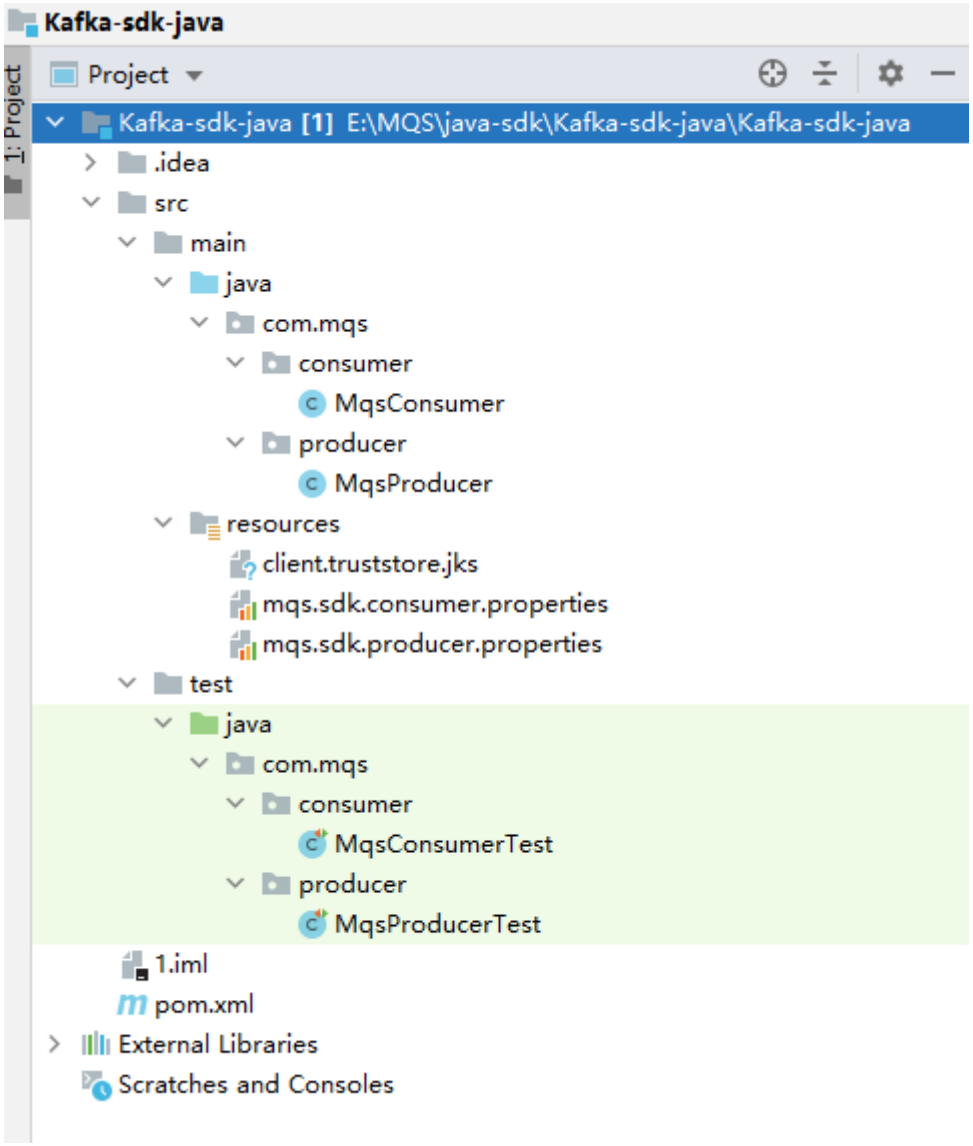


图 2-3 选择 Java 环境



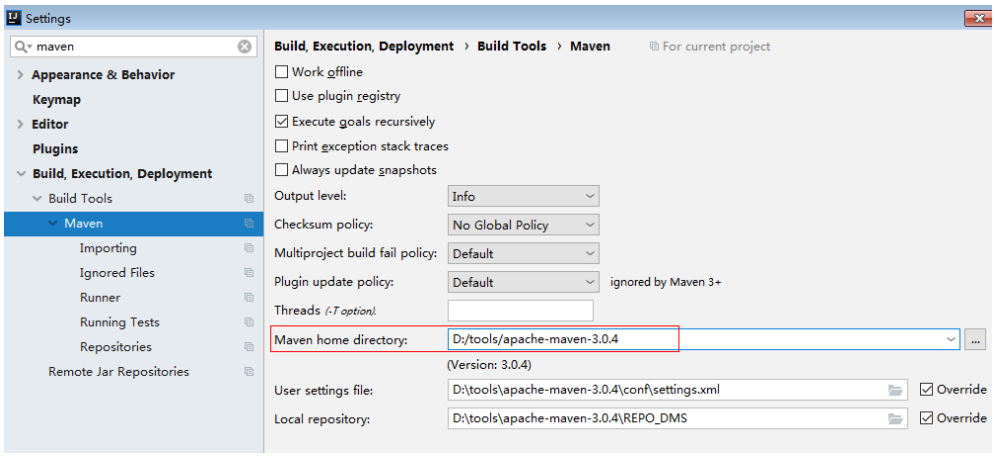
其他选项可默认或自主选择。然后单击Finish，完成Demo导入。

导入后Demo工程如下：



步骤3 配置Maven路径。

打开“File > Settings”，找到“Maven home directory”信息项，选择正确的Maven路径，以及Maven所需的settings.xml文件。



步骤4 修改客户端配置信息。

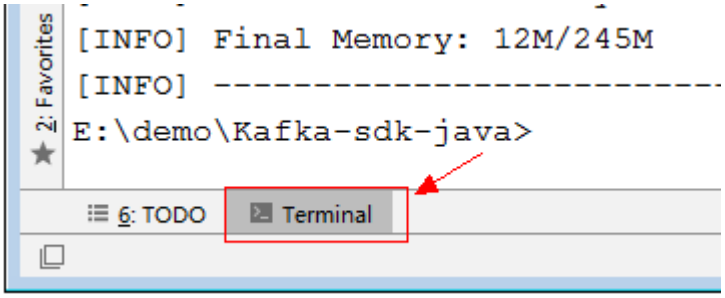
以生产消息为例，需配置以下信息，其中加粗内容必须修改。

```
#以下粗体部分为不同MQS特有的信息，必须修改。客户端其他参数，可以自主添加
#topic名称在具体的生产与消费代码中。
########################
#broker信息请从控制台界面获取。
#举例：bootstrap.servers=192.168.0.196:9095,192.168.0.196:9096,192.168.0.196:9094
bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
#发送确认参数
acks=all
#键的序列化方式
key.serializer=org.apache.kafka.common.serialization.StringSerializer
#值的序列化方式
value.serializer=org.apache.kafka.common.serialization.StringSerializer
#producer可以用来缓存数据的内存大小
buffer.memory=33554432
#重试次数
retries=0
########################
#如果不使用SASL认证，以下参数请注释掉。
########################
#设置jaas用户和密码，通过控制台设置
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
#SASL鉴权方式
sasl.mechanism=PLAIN
#加密协议，目前支持SASL_SSL协议
security.protocol=SASL_SSL
#ssl truststore文件的位置，证书文件在demo包的\src\main\resources路径下。
ssl.truststore.location=E:\\temp\\client.truststore.jks
#ssl truststore文件的密码
ssl.truststore.password=dms@kafka
ssl.endpoint.identification.algorithm=
```

步骤5 打开IDEA的Terminal窗口，执行**mvn test**命令体验demo。

Terminal窗口默认在IDEA工具的**左下角**：

图 2-4 IDEA 的 Terminal 窗口位置



生产消息会得到以下回显信息：

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mqs.producer.MqsProducerTest
produce msg:The msg is 0
produce msg:The msg is 1
produce msg:The msg is 2
produce msg:The msg is 3
produce msg:The msg is 4
produce msg:The msg is 5
```

```
produce msg:The msg is 6
produce msg:The msg is 7
produce msg:The msg is 8
produce msg:The msg is 9
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 138.877 sec
```

消费消息会得到以下回显信息：

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mqs.consumer.MqsConsumerTest
the numbers of topic:0
the numbers of topic:0
the numbers of topic:6
ConsumerRecord(topic = topic-0, partition = 2, offset = 0, CreateTime = 1557059377179, serialized key size = -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value = The msg is 2)
ConsumerRecord(topic = topic-0, partition = 2, offset = 1, CreateTime = 1557059377195, serialized key size = -1, serialized value size = 12, headers = RecordHeaders(headers = [], isReadOnly = false), key = null, value = The msg is 5)
```

----结束

2.3.4 Java 客户端使用说明

本文介绍Maven方式下为MQS引入Kafka客户端，并完成客户端连接以及消息生产与消费的相关示例。如果您需要在查看Demo具体表现，请查看**Java开发环境搭建**。

下文所有配置信息，如MQS连接地址、Topic名称、用户信息等，请参考**收集连接信息**获取。

Maven 中引入 Kafka 客户端

MQS基于Kafka社区版本1.1.0和2.3.0，请使用相同版本的客户端。您可以在ROMA Connect实例控制台的“实例信息”页面，在“MQS基本信息”下查看Kafka版本信息。

若使用1.1.0版本客户端，则version参数设置为1.1.0，若使用2.3.0版本客户端，则version参数设置为2.3.0，此处以2.3.0版本客户端为例进行说明。

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.3.0</version>
</dependency>
```

准备配置信息

为了方便，下文分生产与消费两个配置文件介绍。如果ROMA Connect实例开启了SASL认证，在Java客户端的配置文件中必须配置涉及SASL认证的相关信息，否则无法连接。如果没有使用SASL认证，请注释掉相关配置。

- 生产消息配置文件（对应生产消息代码中的mqs.sdk.producer.properties文件）

  以下粗体部分为不同MQS特有的信息，必须修改。客户端其他参数，可以自主添加。所有配置信息，如MQS连接地址、Topic名称、用户信息等，请参考**收集连接信息**获取。

  ```
  #topic名称在具体的生产与消费代码中。
  ########################
  #broker信息请从控制台界面获取。
  #举例：bootstrap.servers=192.168.0.196:9095,192.168.0.196:9096,192.168.0.196:9094
  bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
  ```

```
#发送确认参数
acks=all
#键的序列化方式
key.serializer=org.apache.kafka.common.serialization.StringSerializer
#值的序列化方式
value.serializer=org.apache.kafka.common.serialization.StringSerializer
#producer可以用来缓存数据的内存大小
buffer.memory=33554432
#重试次数
retries=0
########################
#如果不使用SASL认证，以下参数请注释掉。
########################
#设置jaas用户和密码，通过控制台设置
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
#SASL鉴权方式
sasl.mechanism=PLAIN
#加密协议，目前支持SASL_SSL协议
security.protocol=SASL_SSL
#ssl truststore文件的位置
ssl.truststore.location=E:\\temp\\client.truststore.jks
#ssl truststore文件的密码，固定，请勿修改
ssl.truststore.password=dms@kafka
ssl.endpoint.identification.algorithm=
```

- 消费消息配置文件（对应消费消息代码中的mqs.sdk.consumer.properties文件）

  以下粗体部分为不同MQS特有的信息，必须修改。客户端其他参数，可以自主添加。所有配置信息，如MQS连接地址、Topic名称、用户信息等，请参考 **收集连接信息**获取。

```
#topic名称在具体的生产与消费代码中。
########################
#broker信息请从控制台界面获取。
#举例：bootstrap.servers=192.168.0.196:9095,192.168.0.196:9096,192.168.0.196:9094
bootstrap.servers=ip1:port1,ip2:port2,ip3:port3
#用来唯一标识consumer进程所在组的字符串，请您自行设定。
#如果设置同样的group id，表示这些processes都是属于同一个consumer group
group.id=1
#键的序列化方式
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#值的序列化方式
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
#偏移量的方式
auto.offset.reset=earliest
########################
#如果不使用SASL认证，以下参数请注释掉。
########################
#设置jaas帐号和密码，通过控制台设置
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="password";
#SASL鉴权方式
sasl.mechanism=PLAIN
#加密协议，目前支持SASL_SSL协议
security.protocol=SASL_SSL
#ssl truststore文件的位置
ssl.truststore.location=E:\\temp\\client.truststore.jks
#ssl truststore文件的密码
ssl.truststore.password=dms@kafka
ssl.endpoint.identification.algorithm=
```

生产消息

- 测试代码：

```
package com.mqs.producer;

import org.apache.kafka.clients.producer.Callback;
```

```
import org.apache.kafka.clients.producer.RecordMetadata;
import org.junit.Test;

public class MqsProducerTest {
    @Test
    public void testProducer() throws Exception {
        MqsProducer<String, String> producer = new MqsProducer<String, String>();
        int partiton = 0;
        try {
            for (int i = 0; i < 10; i++) {
                String key = null;
                String data = "The msg is " + i;
                // 注意填写您创建的topic名称。另外，生产消息的API有多个，具体参见Kafka官网或者下文的
生产消息代码。
                producer.produce("topic-0", partiton, key, data, new Callback() {
                    public void onCompletion(RecordMetadata metadata,
                        Exception exception) {
                        if (exception != null) {
                            exception.printStackTrace();
                            return;
                        }
                        System.out.println("produce msg completed");
                    }
                });
                System.out.println("produce msg:" + data);
            }
        } catch (Exception e) {
            // TODO: 异常处理
            e.printStackTrace();
        } finally {
            producer.close();
        }
    }
}
```

- 生产消息代码：

```
package com.mqs.producer;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.Properties;

import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class MqsProducer<K, V> {
    //引入生产消息的配置信息，具体内容参考上文
    public static final String CONFIG_PRODUCER_FILE_NAME = "mqs.sdk.producer.properties";

    private Producer<K, V> producer;

    MqsProducer(String path)
    {
        Properties props = new Properties();
        try {
            InputStream in = new BufferedInputStream(new FileInputStream(path));
            props.load(in);
        }catch (IOException e)
        {
            e.printStackTrace();
            return;
        }
```



```
    producer = new KafkaProducer<K,V>(props);
}
MqsProducer()
{
    Properties props = new Properties();
    try {
        props = loadFromClasspath(CONFIG_PRODUCER_FILE_NAME);
    }catch (IOException e)
    {
        e.printStackTrace();
        return;
    }
    producer = new KafkaProducer<K,V>(props);
}

/**
 * 生产消息
 *
 * @param topic        topic对象
 * @param partition    partition
 * @param key          消息key
 * @param data         消息数据
 */
public void produce(String topic, Integer partition, K key, V data)
{
    produce(topic, partition, key, data, null, (Callback)null);
}

/**
 * 生产消息
 *
 * @param topic        topic对象
 * @param partition    partition
 * @param key          消息key
 * @param data         消息数据
 * @param timestamp    timestamp
 */
public void produce(String topic, Integer partition, K key, V data, Long timestamp)
{
    produce(topic, partition, key, data, timestamp, (Callback)null);
}
/**
 * 生产消息
 *
 * @param topic        topic对象
 * @param partition    partition
 * @param key          消息key
 * @param data         消息数据
 * @param callback     callback
 */
public void produce(String topic, Integer partition, K key, V data, Callback callback)
{
    produce(topic, partition, key, data, null, callback);
}

public void produce(String topic, V data)
{
    produce(topic, null, null, data, null, (Callback)null);
}

/**
 * 生产消息
 *
 * @param topic        topic对象
 * @param partition    partition
 * @param key          消息key
 * @param data         消息数据
 * @param timestamp    timestamp
 * @param callback     callback
```

```
     */
    public void produce(String topic, Integer partition, K key, V data, Long timestamp, Callback
callback)
    {
        ProducerRecord<K, V> kafkaRecord =
                timestamp == null ? new ProducerRecord<K, V>(topic, partition, key, data)
                        : new ProducerRecord<K, V>(topic, partition, timestamp, key, data);
        produce(kafkaRecord, callback);
    }

    public void produce(ProducerRecord<K, V> kafkaRecord)
    {
        produce(kafkaRecord, (Callback)null);
    }

    public void produce(ProducerRecord<K, V> kafkaRecord, Callback callback)
    {
        producer.send(kafkaRecord, callback);
    }

    public void close()
    {
        producer.close();
    }

    /**
     * get classloader from thread context if no classloader found in thread
     * context return the classloader which has loaded this class
     *
     * @return classloader
     */
    public static ClassLoader getCurrentClassLoader()
    {
        ClassLoader classLoader = Thread.currentThread()
                .getContextClassLoader();
        if (classLoader == null)
        {
            classLoader = MqsProducer.class.getClassLoader();
        }
        return classLoader;
    }

    /**
     * 从classpath 加载配置信息
     *
     * @param configFileName 配置文件名称
     * @return 配置信息
     * @throws IOException
     */
    public static Properties loadFromClasspath(String configFileName) throws IOException
    {
        ClassLoader classLoader = getCurrentClassLoader();
        Properties config = new Properties();

        List<URL> properties = new ArrayList<URL>();
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
```

```
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

消费消息

- 测试代码：

```
package com.mqs.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.junit.Test;
import java.util.Arrays;

public class MqsConsumerTest {
    @Test
    public void testConsumer() throws Exception {
        MqsConsumer consumer = new MqsConsumer();
        consumer.consume(Arrays.asList("topic-0"));
        try {
            for (int i = 0; i < 10; i++){
                ConsumerRecords<Object, Object> records = consumer.poll(1000);
                System.out.println("the numbers of topic:" + records.count());
                for (ConsumerRecord<Object, Object> record : records)
                {
                    System.out.println(record.toString());
                }
            }
        }catch (Exception e)
        {
            // TODO: 异常处理
            e.printStackTrace();
        }finally {
            consumer.close();
        }
    }
}
```

- 消费消息代码：

```
package com.mqs.consumer;

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.*;

public class MqsConsumer {

    public static final String CONFIG_CONSUMER_FILE_NAME = "mqs.sdk.consumer.properties";

    private KafkaConsumer<Object, Object> consumer;

    MqsConsumer(String path)
    {
```

```
    Properties props = new Properties();
    try {
        InputStream in = new BufferedInputStream(new FileInputStream(path));
        props.load(in);
    }catch (IOException e)
    {
        e.printStackTrace();
        return;
    }
    consumer = new KafkaConsumer<Object, Object>(props);
}

MqsConsumer()
{
    Properties props = new Properties();
    try {
        props = loadFromClasspath(CONFIG_CONSUMER_FILE_NAME);
    }catch (IOException e)
    {
        e.printStackTrace();
        return;
    }
    consumer = new KafkaConsumer<Object, Object>(props);
}
public void consume(List topics)
{
    consumer.subscribe(topics);
}

public ConsumerRecords<Object, Object> poll(long timeout)
{
    return consumer.poll(timeout);
}

public void close()
{
    consumer.close();
}

/**
 * get classloader from thread context if no classloader found in thread
 * context return the classloader which has loaded this class
 *
 * @return classloader
 */
public static ClassLoader getCurrentClassLoader()
{
    ClassLoader classLoader = Thread.currentThread()
            .getContextClassLoader();
    if (classLoader == null)
    {
        classLoader = MqsConsumer.class.getClassLoader();
    }
    return classLoader;
}

/**
 * 从classpath 加载配置信息
 *
 * @param configFileName 配置文件名称
 * @return 配置信息
 * @throws IOException
 */
public static Properties loadFromClasspath(String configFileName) throws IOException
{
    ClassLoader classLoader = getCurrentClassLoader();
    Properties config = new Properties();

    List<URL> properties = new ArrayList<URL>();
```

```
        Enumeration<URL> propertyResources = classLoader
                .getResources(configFileName);
        while (propertyResources.hasMoreElements())
        {
            properties.add(propertyResources.nextElement());
        }

        for (URL url : properties)
        {
            InputStream is = null;
            try
            {
                is = url.openStream();
                config.load(is);
            }
            finally
            {
                if (is != null)
                {
                    is.close();
                    is = null;
                }
            }
        }

        return config;
    }
}
```

2.3.5 Python 客户端使用说明

本文以Linux Centos环境为例，介绍Python版本的Kafka客户端连接指导，包括Kafka客户端安装，以及生产、消费消息。

说明

使用前请参考**收集连接信息**收集Kafka所需的连接信息。

准备环境

- Python：

  一般系统预装了Python。在命令行输入**python**，得到如下回显，说明Python已安装，版本号为3.7.1。

  ```
  [root@ecs-test python-kafka]# python3
  Python 3.7.1 (default, Jul  5 2020, 14:37:24)
  [GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  ```

  如果未安装Python，请使用以下命令安装：

  ```
  yum install python
  ```

- Python版的Kafka客户端：

  执行以下命令，安装推荐版本的kafka-python：

  ```
  pip install kafka-python
  ```

生产消息

- SASL认证方式

  ```
  from kafka import KafkaProducer
  import ssl
  ##连接信息
  conf = {
  ```

```
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic_name',
    'sasl_plain_username': 'username',
    'sasl_plain_password': 'password'
}

context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.verify_mode = ssl.CERT_REQUIRED
##证书文件
context.load_verify_locations("phy_ca.crt")

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'],
                         sasl_mechanism="PLAIN",
                         ssl_context=context,
                         security_protocol='SASL_SSL',
                         sasl_plain_username=conf['sasl_plain_username'],
                         sasl_plain_password=conf['sasl_plain_password'])

data = bytes("hello kafka!", encoding="utf-8")
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
from kafka import KafkaProducer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
}

print('start producer')
producer = KafkaProducer(bootstrap_servers=conf['bootstrap_servers'])

data = bytes("hello kafka!", encoding="utf-8")
producer.send(conf['topic_name'], data)
producer.close()
print('end producer')
```

消费消息

- SASL认证方式

```
from kafka import KafkaConsumer
import ssl
##连接信息
conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic_name',
    'sasl_plain_username': 'username',
    'sasl_plain_password': 'password',
    'consumer_id': 'consumer_id'
}

context = ssl.create_default_context()
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.verify_mode = ssl.CERT_REQUIRED
##证书文件
context.load_verify_locations("phy_ca.crt")

print('start consumer')
consumer = KafkaConsumer(conf['topic_name'],
                         bootstrap_servers=conf['bootstrap_servers'],
                         group_id=conf['consumer_id'],
                         sasl_mechanism="PLAIN",
                         ssl_context=context,
```

```
                    security_protocol='SASL_SSL',
                    sasl_plain_username=conf['sasl_plain_username'],
                    sasl_plain_password=conf['sasl_plain_password'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset,
message.key,message.value))

print('end consumer')
```

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
from kafka import KafkaConsumer

conf = {
    'bootstrap_servers': ["ip1:port1","ip2:port2","ip3:port3"],
    'topic_name': 'topic-name',
    'consumer_id': 'consumer-id'
}

print('start consumer')
consumer = KafkaConsumer(conf['topic_name'],
                    bootstrap_servers=conf['bootstrap_servers'],
                    group_id=conf['consumer_id'])

for message in consumer:
    print("%s:%d:%d: key=%s value=%s" % (message.topic, message.partition,message.offset,
message.key,message.value))

print('end consumer')
```

2.3.6 其他语言客户端使用说明

MQS完全兼容Kafka开源客户端，如果您使用其他语言，也可以从**Kafka官网**获取客户端，按照Kafka官网提供的连接说明，与MQS对接。

2.3.7 附录：如何提高消息处理效率

消息生产和消费的可靠性必须由ROMA Connect、生产者和消费者协同工作才能保证，对使用ROMA Connect的生产者和消费者有如下的使用建议。

重视消息生产与消费的确认过程

消息生产

生产消息后，生产者需要根据ROMA Connect的返回信息确认消息是否发送成功，如果返回失败需要重新发送。

每次生产消息，生产者都需要等待消息发送API的应答信号，以确认消息是否成功发送。在消息传递过程中，如果发生异常，生产者没有接收到发送成功的信号，生产者自己决策是否需要重复发送消息。如果接收到发送成功的信号，则表明该消息已经被ROMA Connect可靠存储。

消息消费

消息消费时，消费者需要确认消息是否已被成功消费。

生产的消息被依次存储在ROMA Connect的存储介质中。消费时依次获取ROMA Connect中存储的消息。消费者获取消息后，进行消费并记录消费成功或失败的状态，并将消费状态提交到ROMA Connect，由ROMA Connect决定消费下一批消息或回滚重新消费消息。

在消费过程中，如果出现异常，没有提交消费确认，该批消息会在后续的消费请求中再次被获取。

消息生产与消费的幂等传递

ROMA Connect设计了一系列可靠性保障措施，确保消息不丢失。例如使用消息同步存储机制防止系统与服务器层面的异常重启或者掉电，使用消息确认（ACK）机制解决消息传输过程中遇到的异常。

考虑到网络异常等极端情况，用户除了做好消息生产与消费的确认，还需要配合ROMA Connect完成消息发送与消费的重复传输设计。

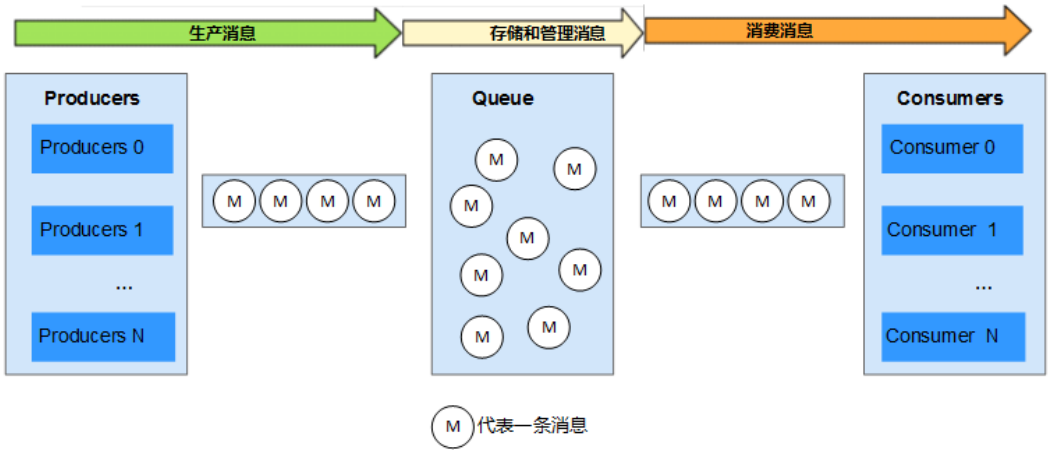
- 当无法确认消息是否已发送成功，生产者需要将消息重复发送给ROMA Connect。
- 当重复收到已处理过的消息，消费者需要告诉ROMA Connect消费成功且保证不重复处理。

消息可以批量生产和消费

为提高消息发送和消息消费效率，推荐使用批量消息发送和消费。通常，默认消息消费为批量消费，而消息发送尽可能采用批量发送，可以有效减少API调用次数。

如下面两张示意图对比所示，消息批量生产与消费，可以减少API调用次数，节约资源。

图 2-5 消息批量生产与消费

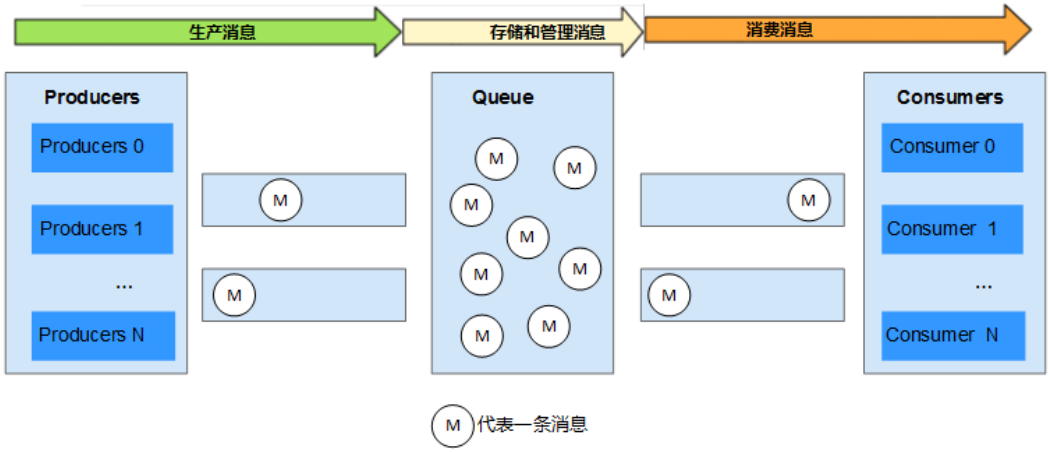


须知

批量发送消息时，单次不能超过10条消息，总大小不能超过512KB。

批量生产（发送）消息可以灵活使用，在消息并发多的时候，批量发送，并发少时，单条发送。这样能够在减少调用次数的同时保证消息发送的实时性。

图 2-6 消息逐条生产与消费



此外，批量消费消息时，消费者应按照接收的顺序对消息进行处理、确认，当对某一条消息处理失败时，不再需要继续处理本批消息中的后续消息，直接对已正确处理的消息进行确认即可。

巧用消费组协助运维

用户使用ROMA Connect作为消息管理系统，查看队列的消息内容对于定位问题与调试服务是至关重要的。

当消息的生产和消费过程中遇到疑难问题时，通过创建不同消费组可以帮助定位分析问题或调试服务对接。用户可以创建一个新的消费组，对主题中的消息进行消费并分析消费过程，这样不会影响其他服务对消息的处理。

2.3.8 附录：spring-kafka 对接限制

概述

spring-kafka兼容开源Kafka客户端，其与开源Kafka客户端的版本对应关系可参见Spring官网。spring-kafka兼容的Kafka客户端版本主要为2.x.x版本，而ROMA Connect消息集成的Kafka服务端版本为1.1.0和2.3.0版本。因此在Spring Boot项目工程中使用spring-kafka连接ROMA Connect时，请确保客户端与服务端的Kafka版本一致。

若spring-kafka连接的ROMA Connect实例为Kafka 1.1.0版本时，大部分的功能可以正常使用，仅少数新增功能不支持。经排查验证，以下为不支持的功能，除此以外的其他功能暂未发现问题。如果在使用过程中遇到其他问题，请联系技术支持。

不支持 zstd 压缩类型

Kafka在2.1.0版本新增了zstd压缩类型，而1.1.0版本的Kafka不支持zstd压缩类型。

- 配置文件：

  src/main/resources/application.yml

- 配置项：

```
spring:
  kafka:
    producer:
      compression-type: xxx
```

- 使用限制：
  “compression-type”的值不能设置为“zstd”。

不支持消费者组静态成员功能

Kafka客户端在2.3版本新增了Consumer参数“group.instance.id”，设置了该ID的消费者被视为一个静态成员。

- 配置文件：
  src/main/resources/application.yml
- 配置项：
```
spring:
  kafka:
    consumer:
      properties:
        group.instance.id: xxx
```
- 使用限制：
  不能添加“group.instance.id”参数配置。

2.4 使用 RESTful API 连接 MQS

2.4.1 Java Demo 使用说明

除了前面章节介绍的使用原生Kafka客户端，MQS（Kafka）实例还可以通过HTTP RESTful方式访问，包括向指定Topic发送消息、消费消息以及确认消费。

这种方式主要用于适配原有业务系统架构，方便统一使用HTTP协议接入。

如何使用

1. 收集连接信息
   包括MQS连接地址与端口、Topic名称、SASL用户名与密码。具体请参考**收集连接信息**。

   说明

   - 若ROMA Connect实例的消息集成在开启SASL_SSL的同时，也开启了VPC内网明文访问，则VPC内无法使用SASL方式连接消息集成的Topic。
   - 使用SASL方式连接消息集成的Topic时，建议在客户端所在主机的“/etc/hosts”文件中配置host和IP的映射关系，否则会引入时延。
     其中，IP地址必须为消息集成的连接地址，host为每个实例主机的名称，可以自定义，但不能重复。例如：
     10.10.10.11 host01
     10.10.10.12 host02
     10.10.10.13 host03
2. 参考示例代码，组装API请求，包括对API请求的签名。
   对API请求签名，指使用SASL的用户名与密码作为密钥对，将请求URL、消息头时间戳等内容进行签名，供后端服务进行校验。
3. 使用Demo向指定Topic生产消息、消费消息和确认消息时，返回的响应消息结构请参考**生产消息接口说明**、**消费消息接口说明**和**消费确认接口说明**。

示例工程搭建

本指南提供了Java语言版本的RESTful API请求发送示例。

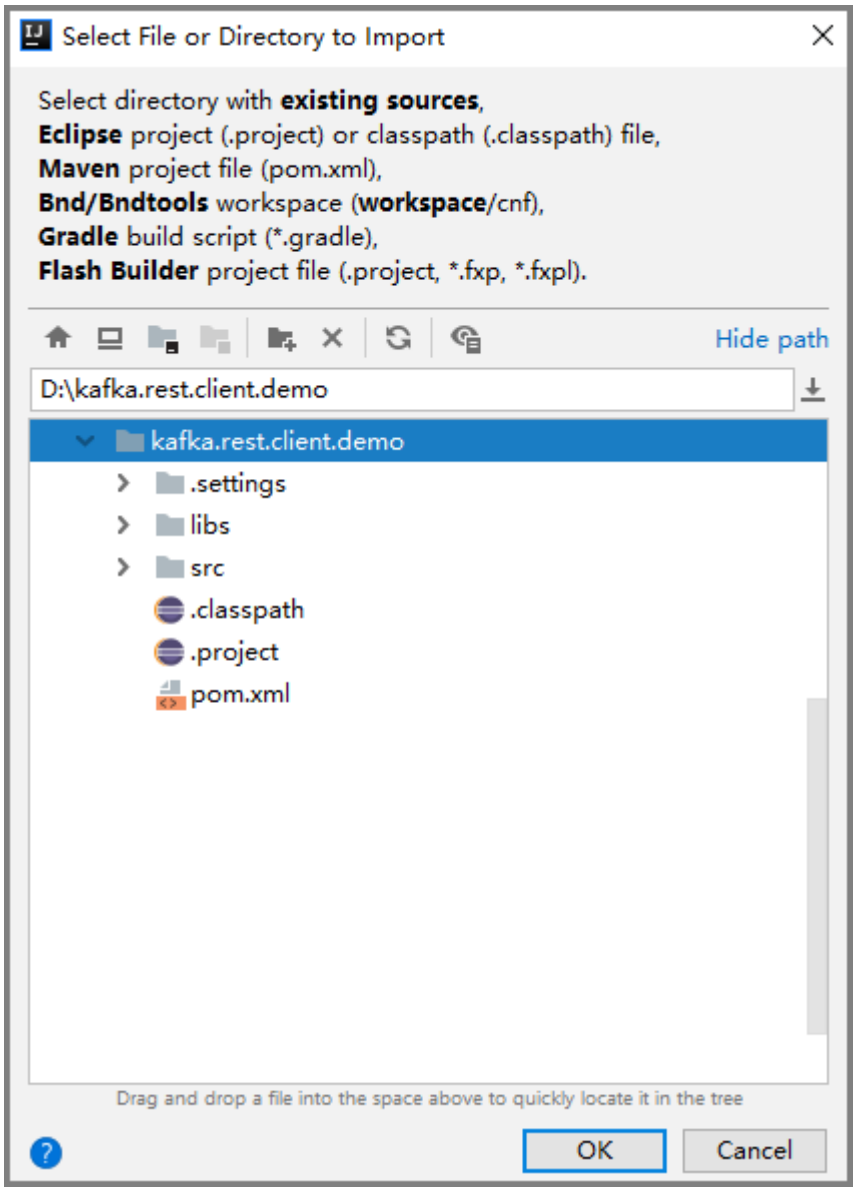
示例为一个在IntelliJ IDEA工具中开发的Maven工程，因此，您如果在本地环境使用，请先安装并配置以下环境（以Windows 10系统为例）：

- Maven：
  Apache Maven 3.0.3及以上版本，可至**Maven官方下载页面**下载。
- JDK：
  Java Development Kit 1.8.111及以上版本，可至**Oracle官方下载页面**下载。
  安装后注意配置JAVA的环境变量。
- IntelliJ IDEA工具：
  IntelliJ IDEA 2018.3.5及以上版本，可至**IntelliJ IDEA官方网站**下载。
- Demo：
  在ROMA Connect实例控制台的“消息集成 MQS > Topic管理”页面，单击右上角的“用户指南 > 下载RESTful API Java Demo包”下载Demo。

步骤1 打开IntelliJ IDEA，在菜单栏选择“Import Project”。

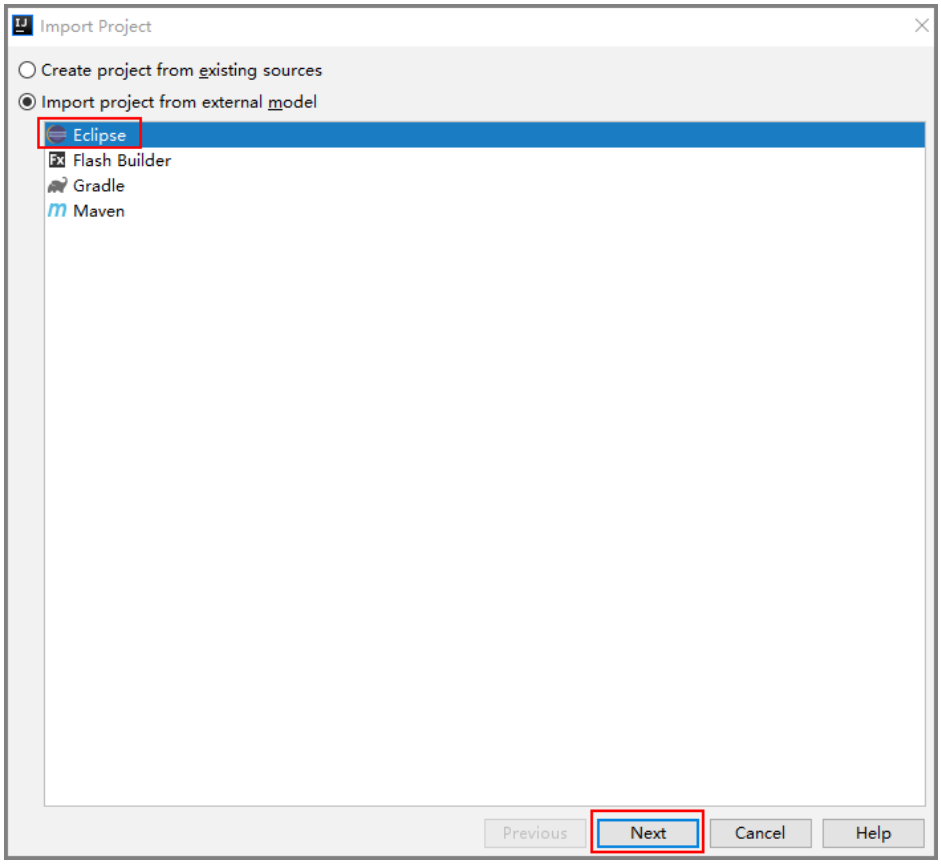
弹出“Select File or Directory to Import”对话框。

步骤2 在弹出的对话框中选择解压后的RESTful API Java Demo路径，单击“OK”。



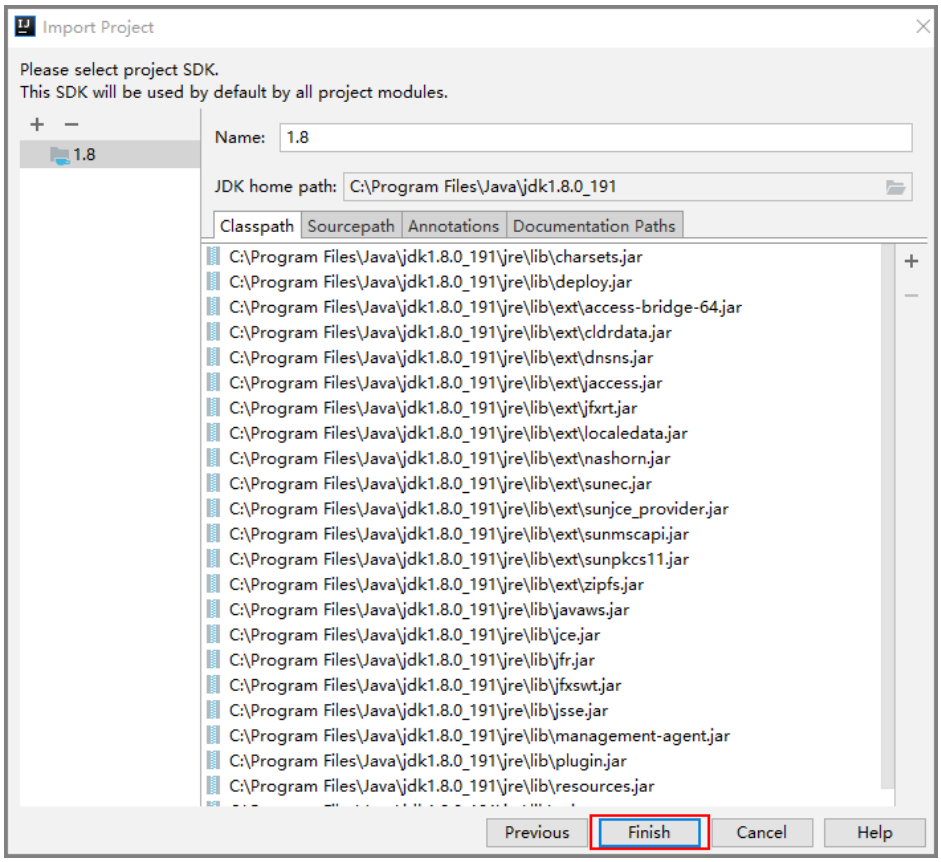
步骤3 “Import project from external model”选择“Eclipse”，单击“Next”，进入下一页后保持默认连续单击“Next”，直到“Please select project SDK”页面。

图 2-7 Import Project



步骤4 单击“Finish”，完成工程导入。

图 2-8 Finish



步骤5 编辑rest-config.properties

文件在src/main/resources目录下。将获取到的Kafka实例连接地址、Topic名称，以及SASL信息填写到下述配置中。其中参数kafka.rest.group为消费组ID，可在客户端指定。

```
# Kafka rest endpoint.
kafka.rest.endpoint=https://{MQS_Instance_IP_Addr}:9292
# Kafka topic name.
kafka.rest.topic=topic_name_demo
# Kafka consume group.
kafka.rest.group=group_id_demo
# Kafka sasl username.
kafka.rest.username=sasl_username_demo
# Kafka sasl password.
kafka.rest.password=sasl_user_passwd_demo
```

步骤6 编辑log4j.properties

修改日志存储目录：

```
log.directory=D://workspace/logs
```

步骤7 运行示例工程，查看消息生产与消费样例。

消息生成与消费的Main方法在RestMain.java中，以Java Application的方式运行即可。

----结束

示例代码解读

- 工程入口：

  工程入口在RestMain.java文件中。

```
public class RestMain
{
    private static final Logger LOGGER = LoggerFactory.getLogger(RestMain.class);

    public static void main(String[] args) throws InterruptedException
    {
        //初始化请求对象。在RestServiceImpl类文件中还包含RESTful API组装，以及对请求签名
        IRestService restService = new RestServiceImpl();
        Base64.Decoder decoder = Base64.getDecoder();
        //以下分别为生产消息、消费消息与消费确认
        // Produce message
        ProduceReq messages = new ProduceReq();
        messages.addMessage("{[{'id': '00001', 'name': 'John'}, {'id': '00002', 'name':
'Mike'}]}").addMessage("Kafka rest client demo!");
        LOGGER.debug("produce message: {}", JsonUtils.convertObject2Str(messages));
        restService.produce(messages);

        // Consume message
        List<ConsumeResp> consumeResps = restService.consume();
        CommitReq commitReq = new CommitReq();
        consumeResps.forEach(resp ->
        {
            LOGGER.debug("handler: {}, content: {}", resp.getHandler(), new
String(decoder.decode(resp.getMessage().getContent())));
            commitReq.addCommit(resp.getHandler());
        });

        // Commit message
        if (commitReq.getMessages().size() != 0)
        {
            CommitResp resp = restService.commit(commitReq);
            LOGGER.info("Commit resp: success: {}, failed: {}", resp.getSuccess(), resp.getFail());
        }
        else
        {
            LOGGER.warn("Commit is empty.");
        }
    }
}
```

- 消息组装与发送：

  以生产消息为例，在下述方法中完成消息组装和签名。其中签名方法调用后，会返回两个消息头：Authorization和X-Sdk-Date，Authorization包含了对请求内容的签名信息。消息头的另一个参数Content-Type需要在代码中添加，参考示例的createRequest()方法。

```
 public List<ProduceResp> produce(ProduceReq messages)
    {
        List<ProduceResp> prodResp = null;
        try
        {
            Request request = createRequest();
            request.setUrl(produceURI);
            request.setMethod("POST");
            request.setBody(JsonUtils.convertObject2Str(messages));
            //对请求内容签名，签名后，请求头部参数会新增两个参数：Authorization和X-Sdk-Date，
Authorization包含了对请求内容的签名信息。
            HttpRequestBase signedRequest = Client.sign(request);
            LOGGER.debug("Request uri: {}, headers: {}", signedRequest.getURI(),
signedRequest.getAllHeaders());
            LOGGER.debug("Request body: {}", request.getBody());

            HttpResponse response = HttpUtils.execute(signedRequest);
            if (response.getStatusLine().getStatusCode() == HttpStatus.SC_CREATED)
```

```
        {
            String jsonStr = EntityUtils.toString(response.getEntity(), "UTF-8");
            prodResp = JsonUtils.convertStr2ListObject(jsonStr, new
TypeReference<List<ProduceResp>>() { });
            LOGGER.info("Produce response: {}", jsonStr);
            return prodResp;
        }
        else
        {
            LOGGER.error("Produce message failed. statusCode: {}, error msg: {}",
                    response.getStatusLine().getStatusCode(),
                    EntityUtils.toString(response.getEntity(), "UTF-8"));
        }
    }
    catch (Exception e)
    {
        LOGGER.error("Produce message failed.");
    }
    return prodResp;
  }
```

2.4.2 生产消息接口说明

功能介绍

向指定队列发送消息，可同时发送多条消息。

- 每次最多发送10条。
- 每次发送的消息总负载不超过2MB。
- endpoint为https://{rest_connect_address}:9292，通过指定实例接口查询rest_connect_address的IP地址。

URI

POST /v1/topic/{topic_name}/messages

表 2-4 参数说明

参数	类型	必选	说明
topic_name	String	是	主题名称。

请求消息

请求参数

参数	类型	必选	说明
messages	Array	是	消息列表，数组大小不能超过10，且不能为空数组。

表 2-5 messages 参数

参数	类型	必选	说明
content	Object	是	消息内容。
id	String	是	消息序号，序列号不能重复。

请求示例

```
{
  "messages": [
    {
      "content": "hello roma-1",
      "id": "1"
    },
    {
      "content": "hello roma-2",
      "id": "2"
    },
    {
      "content": "hello roma-3",
      "id": "3"
    }
  ]
}
```

响应消息

响应参数

参数	类型	说明
state	String	结果状态。成功为success，失败为fail。
id	String	消息序号。

响应示例

```
[
  {
    "state": "success",
    "id": "1"
  },
  {
    "state": "success",
    "id": "2"
  },
  {
    "state": "success",
    "id": "3"
  }
]
```

2.4.3 消费消息接口说明

功能介绍

消费指定队列中的消息，可同时消费多条消息。

- 当队列中消息较少时，单次消费返回的消息数量可能会少于指定条数，但多次消费最终可获取全部消息。当返回的消息为空数组时，表示未消费到消息。
- endpoint为https://{rest_connect_address}:9292，通过指定实例接口查询rest_connect_address的IP地址。

URI

GET /v1/topic/{topic_name}/group/{group_name}/messages?ack_wait={ack_wait}&time_wait={time_wait}&max_msgs={max_msgs}

表 2-6 参数说明

参数	类型	必选	说明
topic_name	String	是	主题名称。
group_name	String	是	消费组名称。长度不超过249位的字符串，包含a~z，A~Z，0~9、中划线（-）和下划线（_）。
ack_wait	Integer	否	提交确认消费的超时时间，客户端需要在该时间内提交消费确认，如果超过指定时间，没有确认消费，系统会报消息确认超时或handler无效，则默认为消费失败。取值范围：1~300s。默认值：15s
time_wait	Integer	否	设定队列可消费的消息为0时的读取消息等待时间。 如果在等待时间内有新的消息，则立即返回消费结果，如果等待时间内没有新的消息，则到等待时间后返回消费结果。取值范围：1~30s。 默认值：3s
max_msgs	Integer	否	获取可消费的消息的条数。取值范围：1~10。默认值：10
max_bytes	Integer	否	每次消费的消息总负载最大值。取值范围：1~2097152。默认值：524288。

请求消息

请求参数

无。

请求示例

无。

响应消息

响应参数

参数	类型	说明
handler	String	消息handler。
message	JSON对象	消息的内容。

表 2-7 message 参数

参数	类型	说明
content	JSON	消息体的内容。Base64加密密文。

响应示例

```
[
  {
    "handler": "NCMxMDAjMTgjMA==",
    "message": {
      "content": "ImhlbGxvIGh1YXdlaWNsb3VkLTIi"
    }
  }
]
```

2.4.4 消费确认接口说明

功能介绍

确认已经消费指定消息。

- 在消费者消费消息期间，消息仍然停留在队列中，但消息从被消费开始的30秒内不能被该消费组再次消费，若在这30秒内没有被消费者确认消费，则MQS认为消息未消费成功，将可以被继续消费。
- endpoint为https://{rest_connect_address}:9292，通过指定实例接口查询rest_connect_address的IP地址。

URI

POST /v1/topic/{topic_name}/group/{group_name}/messages

表 2-8 参数说明

参数	类型	必选	说明
topic_name	String	是	主题名称。

参数	类型	必选	说明
group_name	String	是	消费组名称。

请求消息

请求参数

参数	类型	必选	说明
messages	Array	是	消息列表，数组大小不能超过10，且不能为空数组。

表 2-9 参数说明

参数	类型	必选	说明
handler	String	是	消息handler。
status	String	是	消费状态。只能为success，或者fail。

请求示例

```
{
  "messages": [
    {
      "handler": "NCMxMDAjMTgjMA==",
      "status": "success"
    }
  ]
}
```

响应消息

响应参数

参数	类型	说明
success	Integer	确认消费成功的数目。
fail	Integer	确认消费失败的数目。

响应示例

```
{
  "success": 1,
  "fail": 0
}
```

3 设备集成开发指导

3.1 设备集成开发

概述

本文提供了通过设备集成实现设备的消息收发功能的具体步骤，主要分为配置Demo的设备连接信息、收发消息两部分。

准备工作

- 下载SDK：
  设备集成支持标准的MQTT协议，您可以使用开源Eclipse paho MQTT Client与设备集成进行对接。
  下载地址：**Eclipse paho MQTT Client**
- 下载设备集成Demo：
  在ROMA Connect实例控制台的“设备集成 > 设备管理”页面，单击“下载Demo”，下载LINK Demo。
  本例中Demo使用Java版本的SDK。Demo包括两个文件，DeviceConnectDemo.java用于连接设备，DeviceControlDemo.java用于调用控制设备的API。
- 创建产品：
  具体步骤请参考**创建产品**。
- 创建设备：
  具体步骤请参考**注册设备**。

配置 Demo 的设备连接信息

1. 进入ROMA Connect控制台，单击“查看控制台”。
2. 单击左侧“设备集成”，单击“设备管理”。
3. 在“设备管理”页面单击已创建的设备名称，进入设备详情页面，获取设备连接信息。
   需要编辑的设备信息包括：设备连接地址、设备客户端ID、用户名、密码、具有发布权限的Topic、具有订阅权限的Topic。

4. 解压Demo压缩包，在“romalink_demo > src > com > demo > romalink”路径下找到DeviceConnectDemo.java文件。
5. 使用Java编辑工具打开这个文件，编辑设备连接的信息。运行成功后，在“设备管理”页面可以看到在线设备状态。

说明

设备连接地址的格式为：“tcp://*ip*:*port*”，请按照格式输入设备连接地址。

```
// 设备连接地址
final String host = "";
// 设备客户端ID
final String clientId = "";
// 设备认证的用户名
final String userName = "";
// 设备认证的密码
final String password = "";
// 设备Publish权限的Topic
final String pubTopic = "";
// 设备Subscribe权限的Topic
final String subTopic = "";
// 设备发送的消息内容
final String payload = "hello world.";
```

收发消息

DeviceConnectDemo.java预先设置了具有发布权限的topic消息。如果您调用设备集成发送控制设备消息的API，设备可以立即收到消息。

```
client.subscribe(subTopic, (s, mqttMessage) -> {
            String recieveMsg = "Receive message from topic:" + s + "\n";
            System.out.println(recieveMsg + new String(mqttMessage.getPayload(),
StandardCharsets.UTF_8));
        });
```

1. **调用控制设备的API**。
   a. 使用Java编辑器打开DeviceControlDemo.java，把发送控制消息的API的参数修改为您创建好的设备信息。

      需要填写的信息包括：appKey、appSecret、设备客户端ID、具有订阅权限的Topic、发送控制消息的API的访问地址、访问端口、消息内容。

```
public static void main(String[] args)
  {
     // API认证的appKey
     String appKey = "";
     // API认证的appSecret
     String appSecret = "";
     // 要发送控制消息的设备客户端ID
     String clientId = "";
     // 要发送控制消息的设备的sub权限的Topic
     String subTopic = "";
     // 发送控制消息的API的访问地址
     String host = "";
     // 发送控制消息的API的访问端口
     String port = "";
     // 要给设备发送的消息内容
     String payload = "hello world.";

     String url = "https://" + host + ":" + port + "/v1/devices/" + clientId;
     controlDevice(url, appKey, appSecret, clientId, subTopic, payload);
  }
```

      - appKey、appSecret参数可以从ROMA Connect实例控制台的“集成应用”页面中，单击设备所属集成应用的名称，在集成应用的基本信息中获取Key和Secret。

- Port为7443；clientId、subTopic、host可以直接从ROMA Connect实例控制台的“设备集成 > 设备管理”页面中，单击对应设备的名称，在设备详情页面获取。

b. 重新编译并运行DeviceControlDemo这个类，如果此时该设备处于连接状态并订阅了sub权限的Topic，设备会立即收到一条消息，并在IDE控制台打印出来。API的请求IP和设备连接的IP是相同的，端口是7443。

2. **发送消息**。

您可以设置设备发送消息的内容和频率。例如，您可以命令设备每隔10秒钟向设备集成发送一条消息，代码运行后，设备集成每隔10秒会收到消息。

```
try
    {
        final MqttClient client = new MqttClient(host, clientId);
        client.connect(mqttConnectOptions);
        System.out.println("Device connect success. client id is " + clientId + ", host is " + host);

        final MqttMessage message = new MqttMessage();
        message.setQos(1);
        message.setRetained(false);
        message.setPayload(payload.getBytes(StandardCharsets.UTF_8));

        Runnable pubTask = () -> {
            try
            {
                client.publish(pubTopic, message);
            }
            catch (MqttException e)
            {
                System.out.println(e.getMessage());
            }
        };

        client.subscribe(subTopic, (s, mqttMessage) -> {
            String recieveMsg = "Receive message from topic:" + s + "\n";
            System.out.println(recieveMsg + new String(mqttMessage.getPayload(),
StandardCharsets.UTF_8));
        });

        ScheduledExecutorService service = Executors
                .newSingleThreadScheduledExecutor();
        service.scheduleAtFixedRate(pubTask, 0, 10, TimeUnit.SECONDS);
    }
```

说明

- Connect代码模拟的是MQTT.fx客户端连接设备的功能，所以连接以后，设备会显示“已经连接”。
- 为防止因网络不稳定或实例升级等原因导致设备出现连接中断的情况，需要在设备开发过程中增加设备状态检测和设备断开自动重连机制。

3.2 MQTT 协议 Topic 规范

3.2.1 使用前必读

须知

CoAP协议当前仅支持直连设备命令下发和数据上报。

- 物联网平台作为消息接收方时，已默认订阅了相关Topic，设备只要向对应Topic发送消息，物联网平台就可以接收。
- 设备作为消息接收方时，需要先订阅相关Topic，这样物联网平台向对应Topic发送消息时，设备才能接收到。设备需要根据具体实现的业务来决定订阅哪些Topic。

Topic	支持的协议	消息发送方（Publisher）	消息接收方(Subscriber)	用途
/v1/devices/{gatewayId}/topo/add	MQTT	边设备	物联网平台	边设备添加子设备
/v1/devices/{gatewayId}/topo/addResponse		物联网平台	边设备	物联网平台返回的添加子设备的响应
/v1/devices/{gatewayId}/topo/update		边设备	物联网平台	边设备更新子设备状态
/v1/devices/{gatewayId}/topo/updateResponse		物联网平台	边设备	物联网平台返回的更新子设备状态的响应
/v1/devices/{gatewayId}/topo/delete		物联网平台	边设备	物联网平台删除子设备
/v1/devices/{gatewayId}/topo/query		边设备	物联网平台	边设备查询网关信息
/v1/devices/{gatewayId}/topo/queryResponse		物联网平台	边设备	物联网平台返回的网关信息响应
/v1/devices/{gatewayId}/command		物联网平台	边设备	物联网平台给设备或边设备下发命令
/v1/devices/{gatewayId}/commandResponse		边设备	物联网平台	边设备返回给物联网平台的命令响应
/v1/devices/{gatewayId}/datas		边设备	物联网平台	边设备上报数据

说明

{gatewayId}指设备标识。其中delete延用了之前局点的规范，deleteResponse暂未提供。

3.2.2 网关登录

平台支持MQTT协议的connect消息接口，获取鉴权信息“clientId”、“Username”、“Password”。

参数说明

参数	必选/可选	类型	参数描述
clientId	必选	String(256)	一机一密的设备clientId由4个部分组成：deviceId/nodeId、鉴权类型、密码签名类型、时间戳，通过下划线“_”分隔。 ● 鉴权类型：长度1字节，当前支持2个类型： – “0”，表示使用一机一密设备的deviceId接入。 – “2”，表示使用一机一密设备的nodeId接入。 ● 密码签名类型：长度1字节，当前支持2种类型。 – “0”代表HMACSHA256不校验时间戳。 – “1”代表HMACSHA256校验时间戳。 ● 时间戳：为设备连接平台时的UTC时间，格式为YYYYMMDDHH，如UTC 时间2018/7/24 17:56:20，则应表示为2018072417。 以deviceId为例的clientId示例为：D39564861q3gDa_0_0_2018072417
Username	必选	String(256)	一机一密的设备“Username”。 ● 使用deviceId接入时填写为设备注册成功后返回的“deviceId”值。 ● 使用nodeId接入时填写为设备注册成功时的“nodeId”值。
Password	必选	String(256)	Password的值为使用“HMACSHA256”算法以时间戳为密钥，对secret进行加密后的值。 secret为注册设备时平台返回的secret，或者是设备自身的secret。

3.2.3 添加网关子设备

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/add
消息发送方	边设备
消息接收方	物联网平台

参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。
deviceInfos	必选	List<DeviceInfos>	子设备信息列表，列表大小1~100。

DeviceInfos结构体说明

字段名	必选/可选	类型	参数描述
nodeId	必选	String	设备标识。 支持英文大小写，数字和中划线，长度2-64。
name	可选	String	设备名称。 支持中英文大小写，数字，中划线和#，长度2-64。
description	可选	String	设备描述。 描述长度不能超过200。
manufacturerId	必选	String	厂商ID。 支持英文大小写，数字，中划线和下划线，长度2-50。
model	必选	String	产品型号。 支持英文大小写，数字，中划线和下划线，长度2-50。

示例

```
{
    "deviceInfos": [{
        "manufacturerId": "Test_n",
        "model": "A_n",
        "nodeId": "n-device"
    }],
    "mid": 7
}
```

3.2.4 添加网关子设备响应

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/addResponse

消息发送方	物联网平台
消息接收方	边设备

参数说明

子设备添加成功后会返回响应，其中包含新增的子设备信息，二次开发需自行在本地保存新增的子设备信息，其中返回的deviceId字段将用于子设备数据上报、设备状态更新和子设备删除。

响应参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。
statusCode	必选	Integer	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
statusDesc	可选	String	响应状态描述。
data	必选	List< AddDeviceRsp >	添加子设备的结果信息。

AddDeviceRsp结构体说明

字段名	必选/可选	类型	参数描述
statusCode	必选	Integer	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
statusDesc	可选	String	响应状态描述。
deviceInfo	可选	DeviceInfo	设备信息。

DeviceInfo结构体说明

字段名	必选/可选	类型	参数描述
nodeId	必选	String	设备标识。 支持英文大小写，数字和中划线，长度2-64。
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。
name	必选	String	设备名称。 支持中英文大小写，数字，中划线和#，长度2-64。
description	可选	String	设备描述。 描述长度不能超过200。
manufacturerId	必选	String	厂商ID。 支持英文大小写，数字，中划线和下划线，长度2-50。
model	必选	String	产品型号。 支持英文大小写，数字，中划线和下划线，长度2-50。

示例

```
{
    "data": [{
        "deviceInfo": {
            "manufacturerId": "Test_n",
            "name": "n-device",
            "model": "A_n",
            "nodeId": "n-device",
            "deviceId": "D59eGSxy"
        },
        "statusCode": 0
    }],
    "mid": 7,
    "statusCode": 0
}
```

3.2.5 更新网关子设备状态

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/update
消息发送方	边设备
消息接收方	物联网平台

参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。
deviceStatuses	必选	List<DeviceStatus>	设备状态列表，列表大小1~100。

deviceStatus

字段名	必选/可选	类型	参数描述
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。
status	必选	String	子设备状态： ● OFFLINE：设备离线 ● ONLINE： 设备上线

示例

```
{
    "deviceStatuses": [{
        "deviceId": "D59eGSxy",
        "status": "ONLINE"
    }],
    "mid": 9
}
```

3.2.6 更新网关子设备状态响应

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/updateResponse
消息发送方	物联网平台
消息接收方	边设备

参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。

字段名	必选/可选	类型	参数描述
statusCode	必选	Integer	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
statusDesc	可选	String	响应状态描述。
data	可选	List< UpdateStatusRsp >	更新设备状态信息。

UpdateStatusRsp结构体说明

字段名	必选/可选	类型	参数描述
statusCode	必选	Integer	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
statusDesc	可选	String	结果描述。
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。

示例

```
{
    "data": [{
        "deviceId": "D59eGSxy",
        "statusCode": 0
    }],
    "mid": 9,
    "statusCode": 0
}
```

3.2.7 删除网关子设备

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/delete
消息发送方	物联网平台
消息接收方	边设备

参数说明

字段名	必选/可选	类型	参数描述
id	必选	Integer	删除子设备命令ID。
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。
requestTime	必选	Timestamp	请求时间戳。
request	必选	JsonObject	子设备信息。

JsonObject结构体说明

字段名	必选/可选	类型	参数描述
manufacturerName	必选	String	厂商名称。 支持长度2-64。
manufacturerId	必选	String	厂商ID。 支持英文大小写，数字，中划线和下划线，长度2-50。
model	必选	String	产品型号。 支持英文大小写，数字，中划线和下划线，长度2-50。

示例

```
{
    "requestTime": 1576639584536,
    "request": {
        "manufacturerName": "ATest_n",
        "manufacturerId": "Test_n",
        "model": "A_n"
    },
    "id": 8,
    "deviceId": "n-device"
}
```

3.2.8 查询网关信息

主题 Topic

Topic	/v1/devices/{gatewayId}/topo/query
消息发送方	边设备

消息接收方	物联网平台

参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。
nodeId	必选	String	设备标识。 支持英文大小写，数字和中划线，长度2-64。

示例

```
{
    "mid": 2,
    "nodeId": "test123"
}
```

3.2.9 查询网关信息响应

主题 Topic

Topic	/v1/devices/{gatewayId}/queryResponse
消息发送方	物联网平台
消息接收方	边设备

参数说明

字段名	必选/可选	类型	参数描述
mid	必选	Integer	命令ID。
statusCode	必选	Integer	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
statusDesc	可选	String	响应状态描述。

字段名	必选/可选	类型	参数描述
data	可选	List<DeviceInfo>	查询设备信息。
count	可选	String	设备数量。
marker	可选	String	标签。

表 3-1 DeviceInfo 结构体说明

字段名	必选/可选	类型	参数描述
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。
nodeId	必选	String	设备标识。 支持英文大小写，数字和中划线，长度2-64。
name	必选	String	设备名称。 支持中英文大小写，数字，中划线和#，长度2-64。
description	可选	String	设备描述。 描述长度不能超过200。
manufactureId	必选	String	厂商ID。 支持英文大小写，数字，中划线和下划线，长度2-50。
model	必选	String	产品型号。 支持英文大小写，数字，中划线和下划线，长度2-50。

示例

```
{
   "mid": 2,
   "statusCode": 0,
   "statusDesc": "",
      "marker": "",
      "count": "1",
   "data": [
      {
         "deviceId": "D59eGSxy",
         "nodeId": "test123",
         "name": "n-device",
         "description": "addsSubDevice",
         "manufacturerId": "Test_n",
         "model": "A_n"
      }
   ]
}
```

3.2.10 设备命令下发

主题 Topic

Topic	/v1/devices/{gatewayId}/command
消息发送方	物联网平台
消息接收方	边设备

参数说明

字段名	必选/可选	类型	参数描述
deviceId	必选	String	平台生成的设备唯一标识，对应设备客户端ID。
msgType	必选	String	固定值"cloudReq"，表示平台下发的请求。
serviceId	必选	String	服务ID。
cmd	必选	String	服务的命令名。
paras	必选	ObjectNode	命令的参数。
mid	必选	Int	命令ID。

示例

```
{
	"msgType": "cloudReq",
	"mid": 54132,
	"cmd": "command1",
	"paras": {
		"temperature": 123
	},
	"serviceId": "service1",
	"deviceId": "D23pigXo"
}
```

3.2.11 设备命令下发响应

主题 Topic

Topic	/v1/devices/{gatewayId}/commandResponse

消息发送方	边设备
消息接收方	物联网平台

参数说明

字段名	必选/可选	类型	参数描述
msgType	必选	String	固定值"deviceRsp"，表示设备的应答消息。
mid	必选	Int	命令ID。
errcode	必选	Int	请求处理的结果码。 ● “0”表示成功。 ● 非“0”表示失败。
body	可选	ObjectNode	命令的应答。

示例

```
{
	"body": {
		"orginParameters": {
			"temperature": 123
		},
		"state": "ok"
	},
	"errcode": 0,
	"mid": 54132,
	"msgType": "deviceRsp"
}
```

3.2.12 设备数据上报

主题 Topic

Topic	/v1/devices/{gatewayId}/datas
消息发送方	边设备
消息接收方	物联网平台

参数说明

字段名	必选/可选	类型	参数描述
devices	必选	DeviceS[]	设备数据。

DeviceS结构体说明

字段名	必选/可选	类型	参数描述
deviceId	必选	String(256)	平台生成的设备唯一标识，对应设备客户端ID。
services	必选	List<Services>	服务列表。

Services结构体说明

字段名	必选/可选	类型	参数描述
serviceId	必选	String(256)	服务ID。
data	必选	ObjectNode	服务数据。
eventTime	必选	String(256)	时间格式：yyyyMMdd’T’HHmmss’Z’ 如： 20151212T121212Z。

示例

```
{
    "devices": [{
        "deviceId": "D68NZxB4",
        "services": [{
            "data": {
                "key": "value"
            },
            "eventTime": "20191023T173625Z",
            "serviceId": "serviceName"
        }]
    }]
}
```