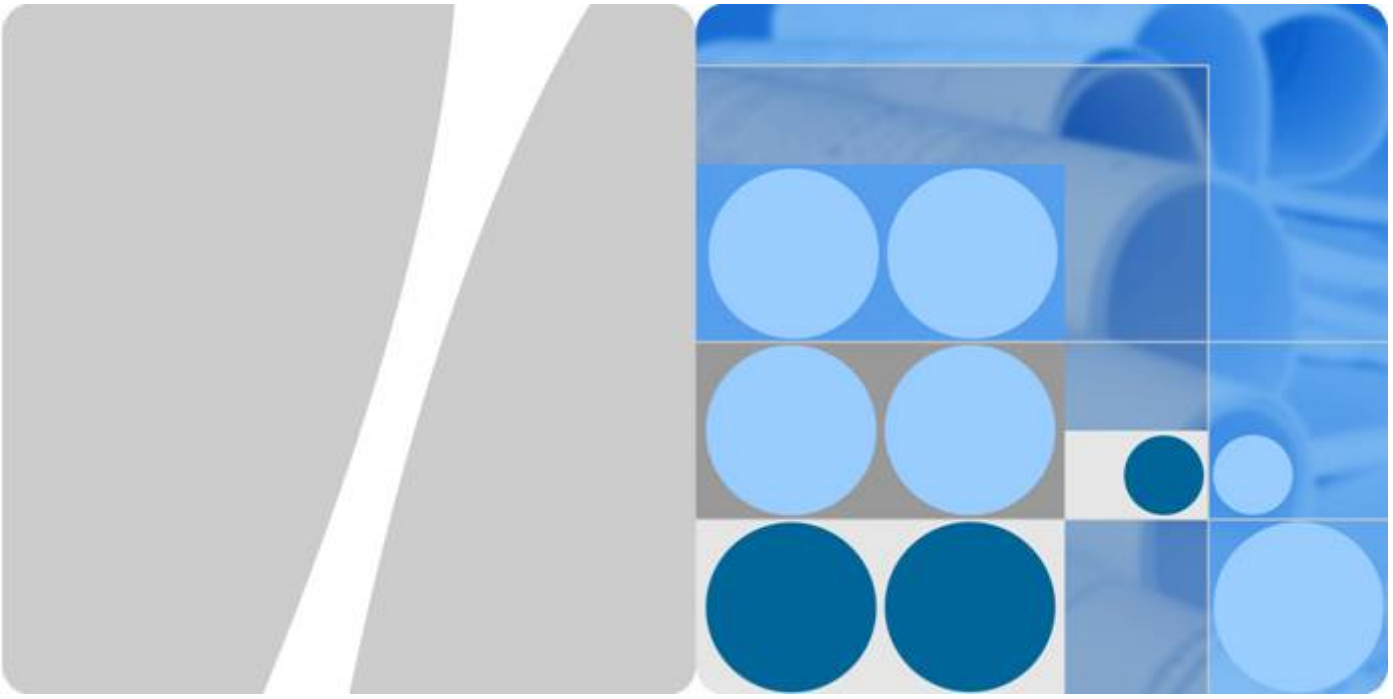


应用与数据集成平台(ROMA Connect)

2.1.5

开发指南

发布日期　2024-04-30

目 录

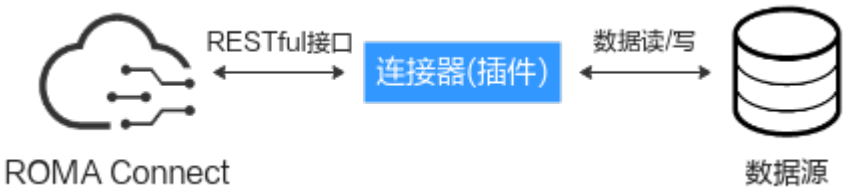
1 数据集成开发指导

1.1 开发说明

1.1.1 开发场景

场景说明

如果ROMA Connect支持的数据源类型无法接入您使用的数据源，您可以自行开发一个数据源插件，用于实现对数据源的读写。同时，这个数据源插件以标准RESTful接口形式供ROMA Connect接入，最终实现ROMA Connect对数据源的读写。这个数据源插件称为连接器，这类通过连接器接入的数据源称为自定义数据源。



ROMA Connect的数据集成开发主要是对连接器的RESTful接口做了规范定义，您在开发连接器的RESTful接口时应遵循**连接器RESTful接口规范**去实现接口。

1.1.2 开发规范

开发建议

- ROMA Connect仅对连接器的RESTful接口做了规范定义，具体连接器对数据源的读写功能以及RESTful接口均需要您自行开发，您可以自行选择开发语言进行连接器开发。
- 连接器开发完成后，您需要自行部署连接器，并确保连接器所在网络与ROMA Connect所在网络互通。

1.2 连接器 RESTful 接口规范

数据读取接口

接口规范定义

- 接口URI

  POST /reader

- 接口请求

```
{
   "job_name": "job_name",
   "datasource": {
      "para1": "******",
      "para2": "******",
       ...
   },
   "params": {
      "extend": {
         "ex_para1": "******",
         "ex_para2": "******",
          ...
      },
      "pagination": {
         "offset": 1,
         "limit": "10"
      },
      "migration": {
         "begin": **********,
         "end": **********
      }
   }
}
```

- 接口响应

```
{
   "datas": [
      {
         "para1": "******",
         "para2": "******",
          ...
      },
      {
         "para1": "******",
         "para2": "******",
          ...
      },
      ...
   ]
}
```

接口参数说明

- 请求参数

  表 1-1 请求参数

参数	是否必选	参数类型	描述
job_name	是	String	任务名称。由英文字母、数字、下划线、中划线组成，长度为4~64字符。

参数	是否必选	参数类型	描述
datasource	是	Object	数据源参数对象。包括连接器接数据库依赖的参数，所有参数组成的Json体。
params	是	**Params**	连接器所需参数对象。

表 1-2 Params 说明

参数	是否必选	参数类型	描述
pagination	否	**Pagination**	分页对象。
migration	否	**Migration**	增量迁移对象。
extend	否	Object	连接器所属的扩展参数，其值为各个扩展参数组成的Json体。

表 1-3 Pagination 说明

参数	是否必选	参数类型	描述
offset	否	Integer	分页查询的偏移量，表示从此偏移量开始查询。
limit	否	Integer	每页显示的数据记录数。

表 1-4 Migration 说明

参数	是否必选	参数类型	描述
begin	否	Date	迁移数据开始时间。
end	否	Date	迁移数据结束时间。

- 响应参数

表 1-5 响应参数

参数	参数类型	描述
datas	List<Object>	读取数据组成的列表。该字段要求满足JsonArray的格式，里面的字段由连接器侧根据实际情况而定。

数据写入接口

接口规范定义

- 接口URI

  POST /writer

- 接口请求

```
{
   "job_name": "job_name",
   "datasource": {
      "para1": "******",
      "para2": "******",
       ...
   },
   "params": {
      "extend": {
         "ex_para1": "******",
         "ex_para2": "******",
          ...
      }
   },
   "meta-data": [
      {
         "name": "id",
         "type": "String",
         "format": "",
         "path": "datas[i].id"
      },
      {
         "name": "company",
         "type": "String",
         "format": "",
         "path": "datas[i].company"
      },
      ...
   ],
   "datas": [
      {
         "data1": "******",
         "data2": "******",
          ...
      },
      {
         "data1": "******",
         "data2": "******",
          ...
      },
      ...
   ]
}
```

- 接口响应

```
{
   "num_success": "2",
   "num_fail": "0",
   "fail_datas": [
      {}
   ]
}
```

接口参数说明

- 请求参数

表 1-6 请求参数

参数	是否必选	参数类型	描述
job_name	是	String	任务名称。由英文字母、数字、下划线、中划线组成，长度为4~64字符。
datasource	是	Object	数据源参数对象。包括连接器接数据库依赖的参数，所有参数组成的Json体。
params	是	**Params**	连接器所需参数对象。
meta-data	是	List<**Meta-data**>	元数据参数列表。
datas	是	List<Object>	连接器处理的数据列表。

表 1-7 Params 说明

参数	是否必选	参数类型	描述
extend	否	Object	连接器所属的扩展参数，其值为各个扩展参数组成的Json体。

表 1-8 Meta-data 说明

参数	是否必选	参数类型	描述
name	是	String	数据字段名称。
type	是	String	字段值类型。支持的类型有：String，Integer，Date和Long。
format	否	String	数据的格式化字符串。当字段值为Date类型时，需要填写该值，用于描述字符串的格式。
path	是	String	字段在源数据中的获取路径。

- 响应参数

表 1-9 响应参数

参数	参数类型	描述
num_success	Integer	数据写入成功数。
num_fail	Integer	数据写入失败数。
fail_datas	List<Object>	处理失败的数据列表。

2 服务集成开发指导

2.1 开发说明

2.1.1 开发场景

场景说明

ROMA Connect的服务集成主要涉及以下几个开发场景：

- **API调用认证开发**：业务系统在调用服务集成APIC开放的API时，若API的认证方式为APP认证或IAM认证，则需要在业务系统侧进行调用认证的开发，为API请求添加认证信息。
  - APP认证（签名认证）：调用侧集成ROMA Connect提供的SDK，对API请求进行签名处理。
  - IAM认证（Token认证）：调用侧先获取云服务平台的认证Token，然后把认证Token携带到API请求中。
  - IAM认证（AK/SK认证）：调用侧集成ROMA Connect提供的SDK，对API请求进行签名处理。
- **自定义后端开发**：用户在使用服务集成APIC的自定义后端创建函数后端和数据后端时，涉及函数脚本或数据库执行语句的编写开发。
  - 函数后端：创建函数后端时需要编写函数脚本，进行函数功能开发，ROMA Connect提供了部分Java类函数供用户直接使用。
  - 数据后端：创建数据后端时需要编写执行语句，用于对数据源进行操作。

- **后端服务签名校验开发**：若API在ROMA Connect上绑定了签名密钥，则ROMA Connect向该API的后端服务发送请求时会携带相应的签名信息。该API的后端服务需要集成ROMA Connect提供的SDK，并对请求中的签名信息进行校验。

2.1.2 开发规范

API 调用认证开发要求

- **开发工具版本要求**：
  - IntelliJ IDEA：2018.3.5或以上版本。
  - Eclipse：3.6.0或以上版本。
  - Visual Studio：2019 version 16.8.4或以上版本。
- **开发语言环境版本要求**：
  - Java：1.8.111或以上版本的Java Development Kit。
  - Go：1.14及以上版本的Go安装包。
  - Python：2.7或3.X版本的Python安装包。
  - JavaScript：15.10.0及以上版本的Nodejs安装包。
  - PHP：8.0.3及以上版本的PHP安装包。
  - Android：4.1.2及以上版本的Android Studio。
- **浏览器版本要求**：Chrome 89.0或以上版本。
- **SDK签名限制**：
  - 使用SDK对API请求签名时，仅支持对Body体为12M及以下的请求进行签名。
  - 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。
  - 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。因此，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。

自定义后端开发要求

- **函数后端开发要求**：
  - 仅支持使用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。
  - 函数后端支持的脚本大小最大为32KB。
- **数据后端开发要求**：
  - 若数据后端执行语句查询获取的数据量较大，建议携带offset和limit参数进行分页，避免因响应数据太大导致响应超时。使用方式如下：
    ```
    select * from table01 limit '${limit}' offset ${offset}
    ```
    其中offset和limit参数key在后端服务请求的Headers、Parameters或Body中传递。
  - 不建议开启结果分页选项，结果分页当前仅支持2000条以内数据的分页。
  - 数据后端支持的执行语句大小最大为32KB。

后端服务签名校验开发要求

- **开发工具版本要求**：
  - IntelliJ IDEA：2018.3.5或以上版本。
  - Eclipse：3.6.0或以上版本。
  - Visual Studio：2019 version 16.8.4或以上版本。
- **开发语言环境版本要求**：
  - Java：1.8.111或以上版本的Java Development Kit。
  - Python：2.7或3.X版本的Python安装包。
- **SDK使用限制**：
  - Python SDK仅支持hmac类型的后端服务签名。
  - C# SDK仅支持hmac类型的后端服务签名。

2.2 API 调用认证开发（APP 认证）

2.2.1 开发准备

获取 API 的调用信息

- 获取API的请求信息

  在ROMA Connect实例控制台选择“服务集成 APIC > API列表”，在页面中可获取API的“域名”、“请求方法”和“请求路径”。单击API的名称进入API详情页面，在“API运行”页签下，获取API的“请求协议”，在“前端配置”中可获取“入参定义”。
- 获取API的认证信息

  APP认证（签名认证）在对API请求进行加密签名时，需要用到API所授权凭据的Key和Secret。

  在ROMA Connect实例控制台选择“服务集成 APIC > 凭据管理”，在“凭据管理”页签下单击API所授权凭据的名称，进入凭据详情页面，获取凭据的Key和Secret。

准备开发环境

- 安装开发工具

  请根据所使用语言选择适合的开发工具。
  - 安装2018.3.5或以上版本的IntelliJ IDEA，安装包请至**IntelliJ IDEA官方网站**下载。
  - 安装2019 version 16.8.4及以上版本的Visual Studio，安装包请至**Visual Studio官方网站**下载。
- 安装开发语言环境
  - Java：安装1.8.111或以上版本的Java Development Kit，安装包请至**Oracle官方下载页面**下载。
  - Go：安装1.14及以上版本的Go安装包，安装包请至**Go官方下载页面**下载。
  - Python：安装2.7或3.X版本的Python安装包，安装包请至**Python官方下载页面**下载。

- JavaScript：安装15.10.0及以上版本的Nodejs安装包，安装包请至**Nodejs官方下载页面**下载。
- PHP：安装8.0.3及以上版本的PHP安装包，安装包请至**PHP官方下载页面**下载。
- Android：安装4.1.2及以上版本的Android Studio，安装包请至**Android Studio官方网站**下载。

2.2.2 APP 认证工作原理

APP 认证流程

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))
```

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。

须知

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：

   ```
   aa521bbe74d13cd8cf536c1a03a5dd85d1934179d33d47110b528eae8b7251e1
   ```

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
aa521bbe74d13cd8cf536c1a03a5dd85d1934179d33d47110b528eae8b7251e1
```

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2-1所示。

表 2-1 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：

```
121c2501e8951ff7d5574423939b9acaa283e55a27c0107d767bb0d68b5ffcab
```

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=121c2501e8951ff7d5574423939b9acaa283e55a27c0107d767bb0d68b5ffcab
```

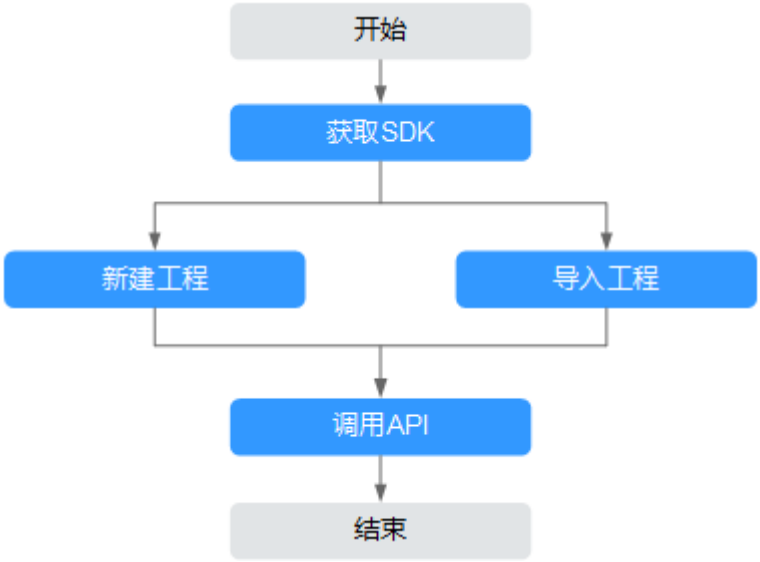
得到签名消息头后，将其增加到原始HTTP请求内容中，请求将被发送给APIC，由APIC完成身份认证。身份认证通过后，该请求才会发送给后端服务进行业务处理。

2.2.3 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。

图 2-1 调用流程



前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

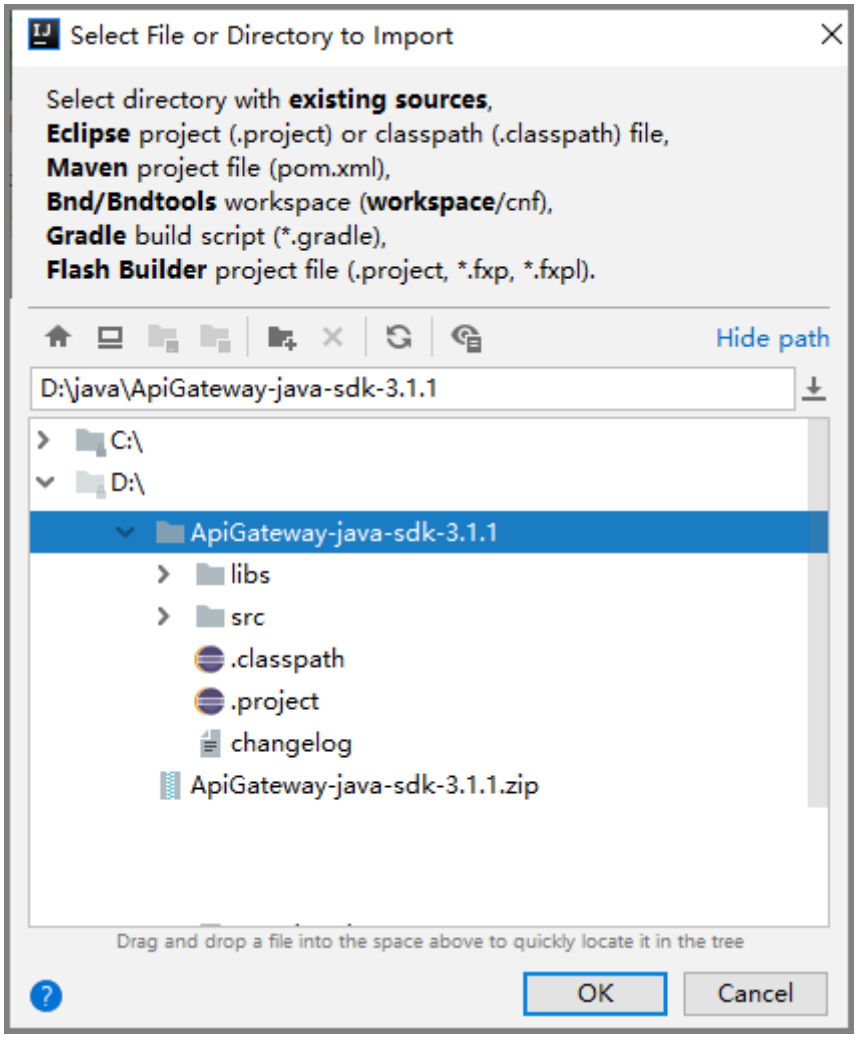
解压后目录结构如下：

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	

名称	说明
src\com\apig\sdk\demo\LargeFileUploadDemo.java	
.classpath	Java工程配置文件。
.project	

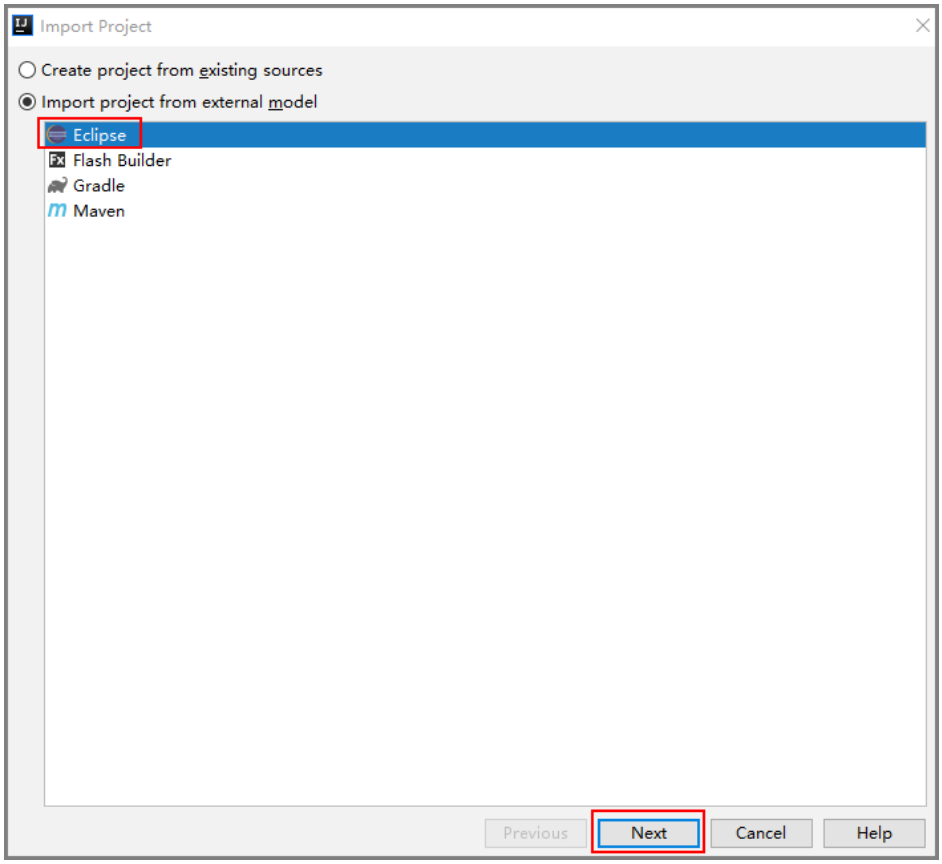
导入工程

1. 打开IntelliJ IDEA，在菜单栏选择“Import Project”。
   弹出“Select File or Directory to Import”对话框。
2. 在弹出的对话框中选择解压后的SDK路径，单击“OK”。



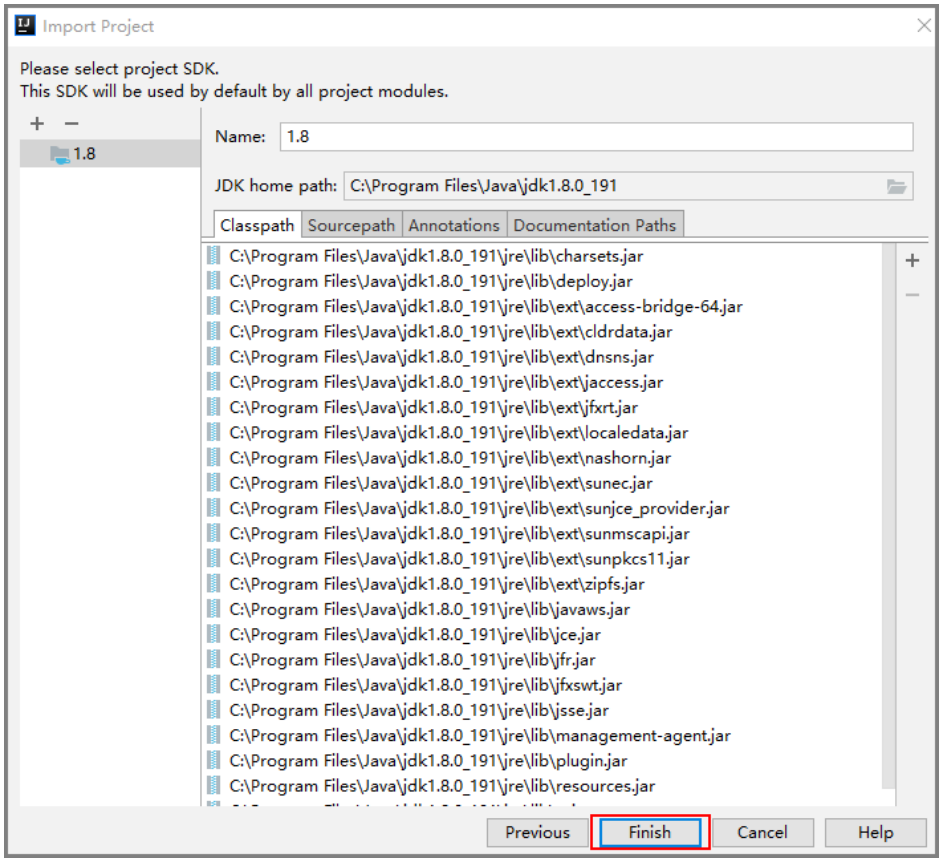
3. “Import project from external model”选择“Eclipse”，单击“Next”，进入下一页后保持默认连续单击“Next”，直到“Please select project SDK”页面。

图 2-2 Import Project



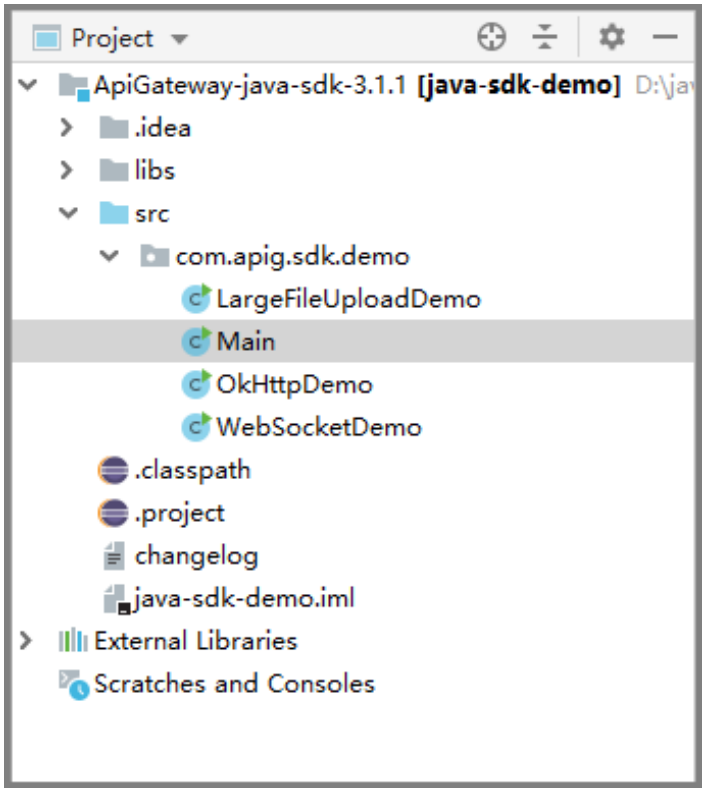
4. 单击“Finish”，完成工程导入。

图 2-3 Finish



5. 完成导入后，目录结构如下图。

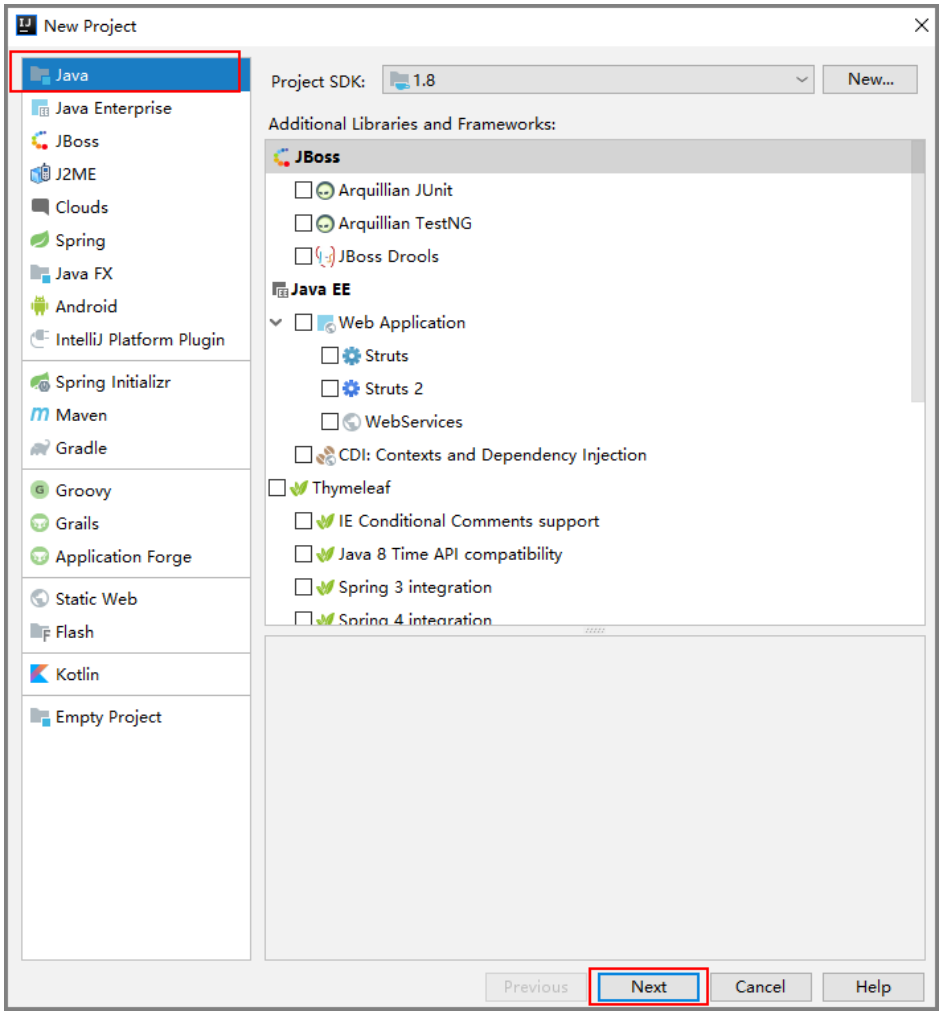
图 2-4 目录结构



新建工程

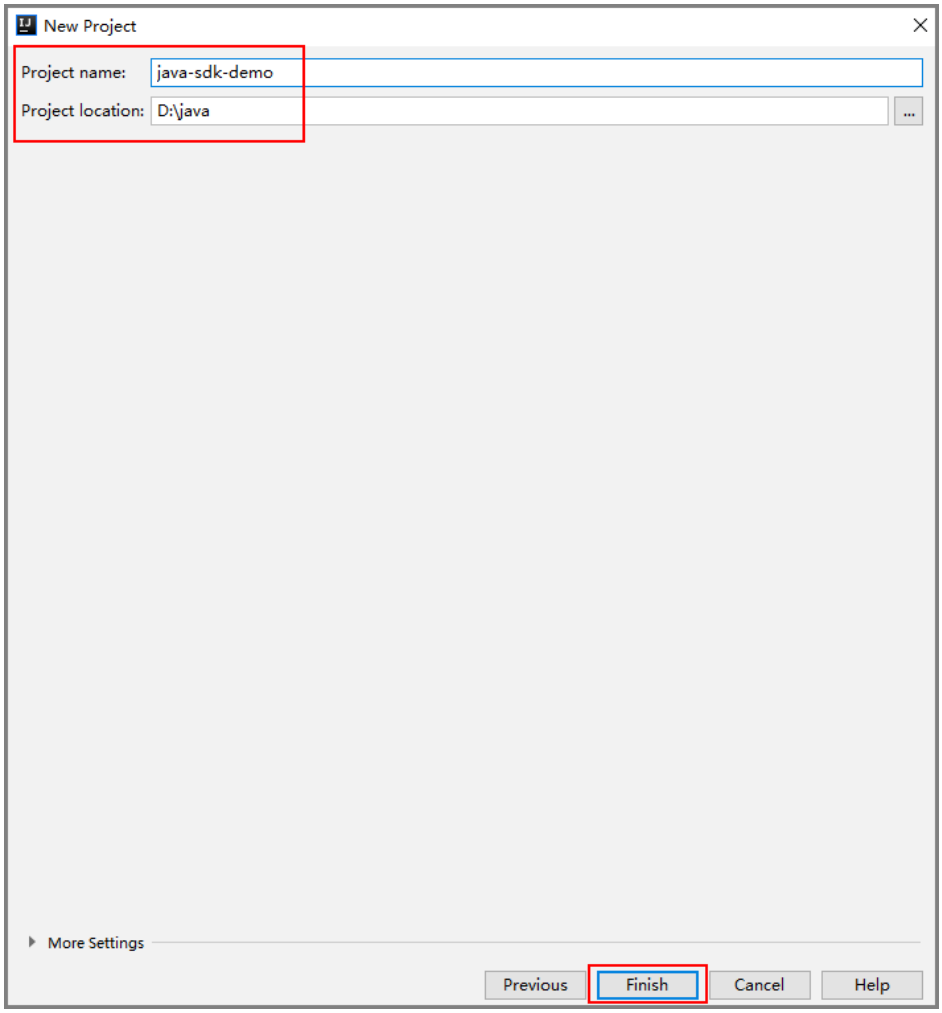
1. 打开IntelliJ IDEA，在菜单栏选择“Create New Project”。
   弹出“New Project”对话框。
2. 在右侧栏中选择“Java”，单击“Next”，进入下一页。

图 2-5 New Project



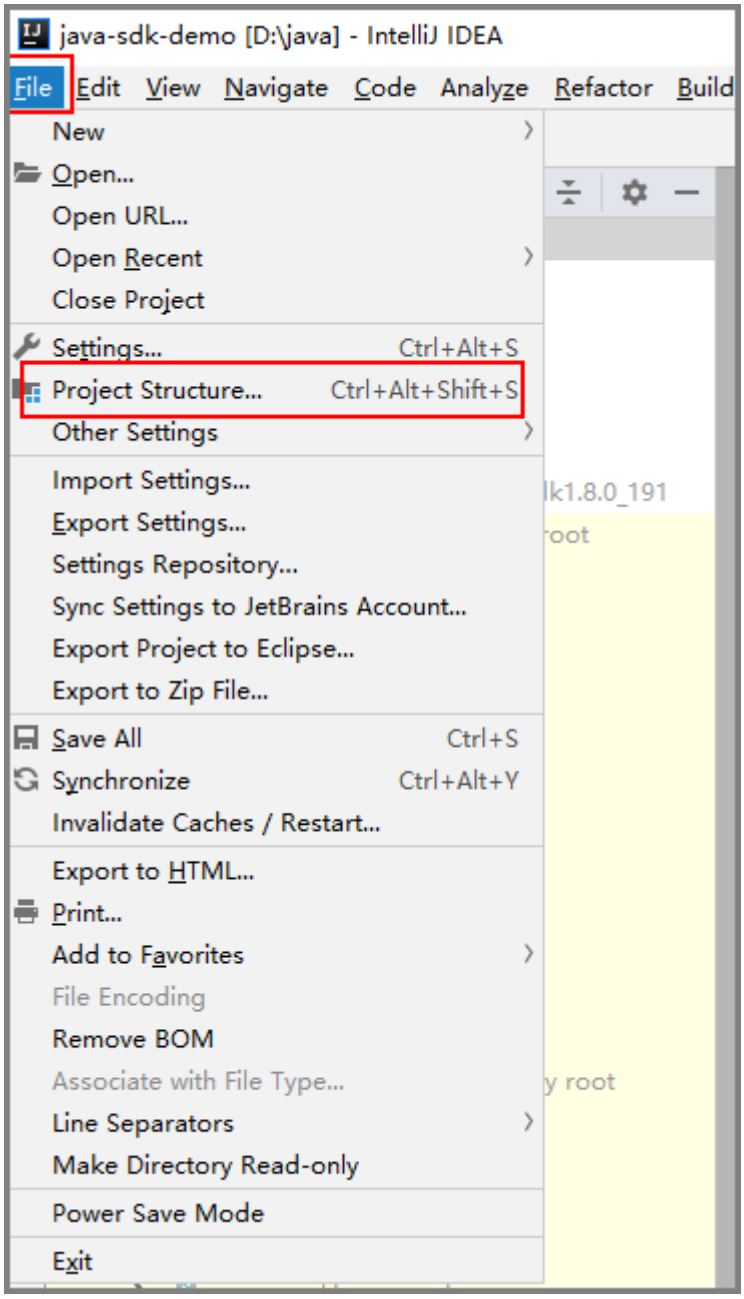
3. 保持默认继续单击“Next”，进入下一页，自定义“Project name”，并选择创建工程所在本地目录“Project location”。

图 2-6 新建工程



4. 导入Java SDK的“jar”文件。
   a. 选择“File > Project Structure”，弹出“Project Structure”对话框。

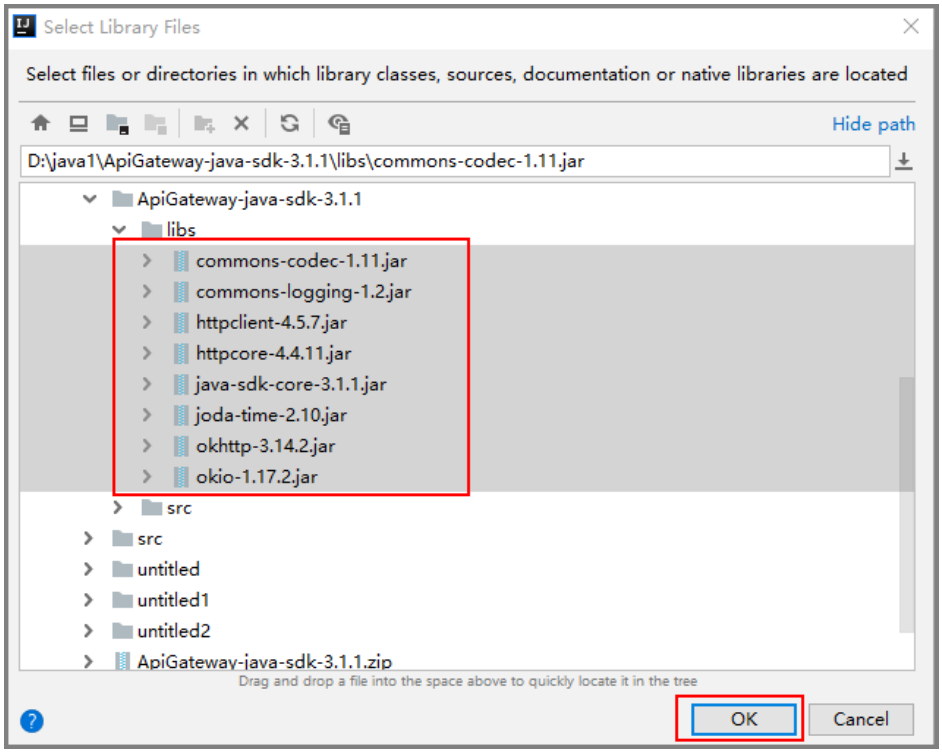
图 2-7 导入 jar 文件



b. 在“Project Structure”对话框中选择“Libraries > + >Java”，界面弹出“Select Library Files”对话框。

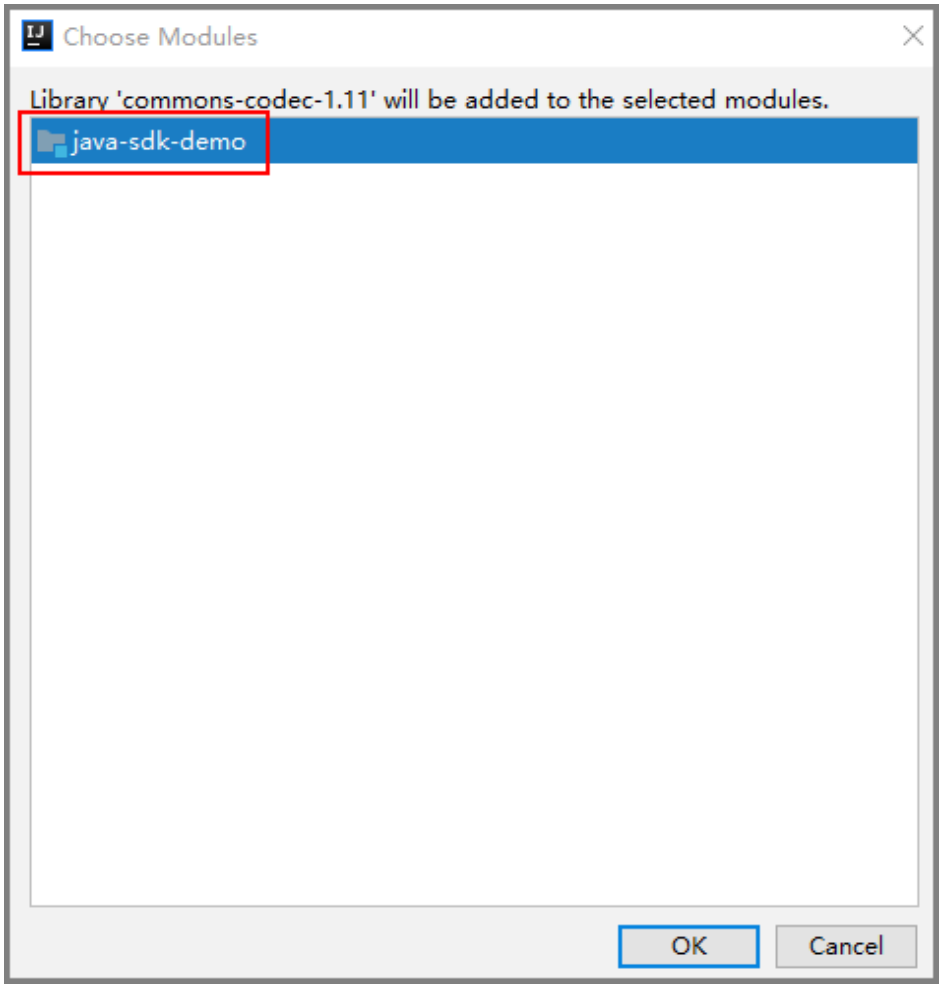
c. 选择SDK所在目录中“\libs”目录下所有以“jar”结尾的文件，单击“ok”。

图 2-8 选择 jar 文件



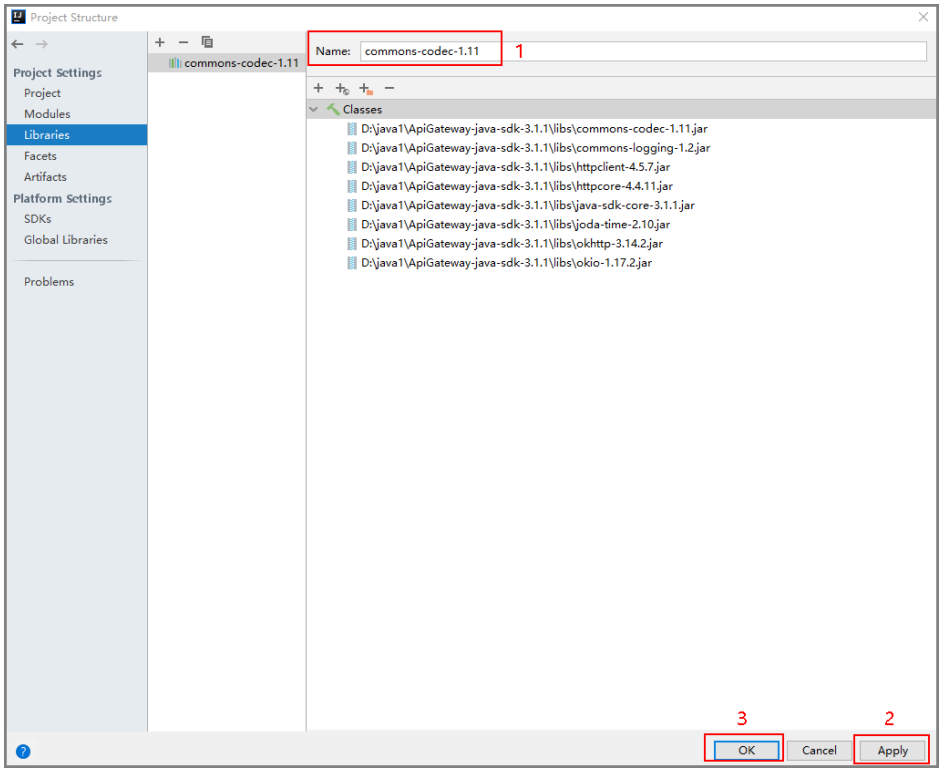
d. 选择步骤3已创建的工程，单击“ok”。

图 2-9 选择工程



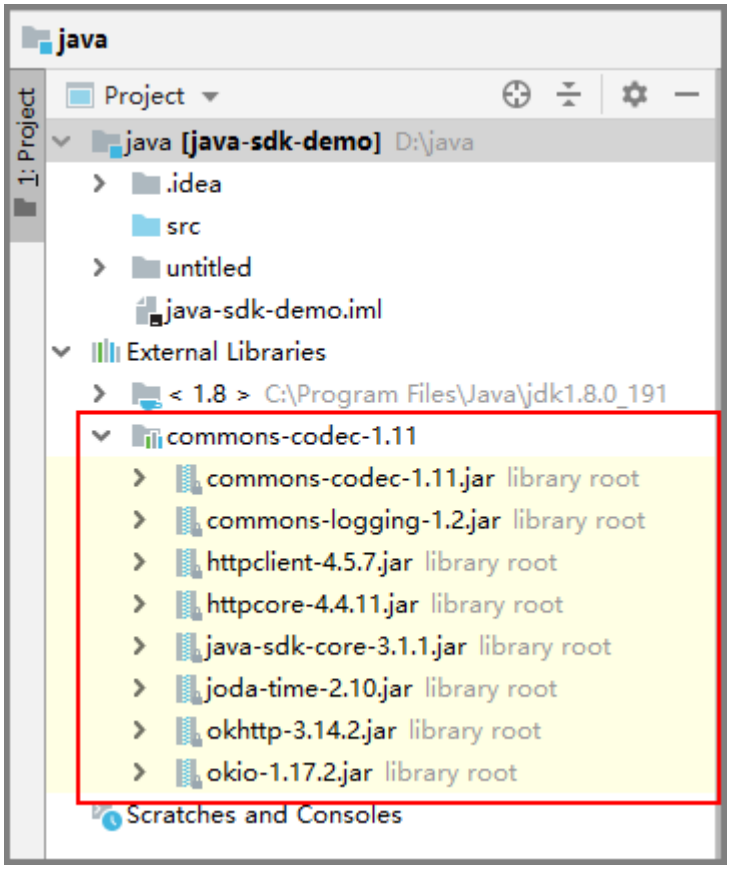
e. 填写jar文件所在目录的名称，单击“Apply > OK”。

图 2-10 jar 文件目录



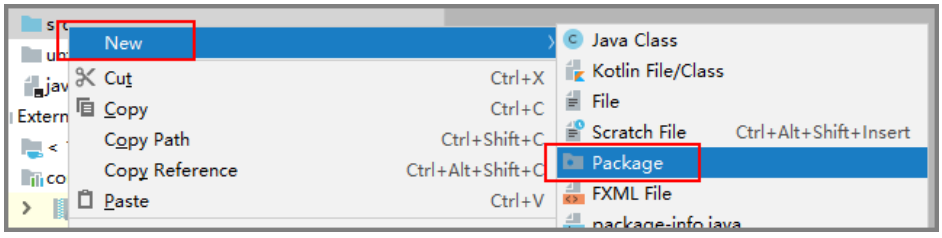
f. 完成jar文件导入，导入后目录结构如下图。

图 2-11 目录结果



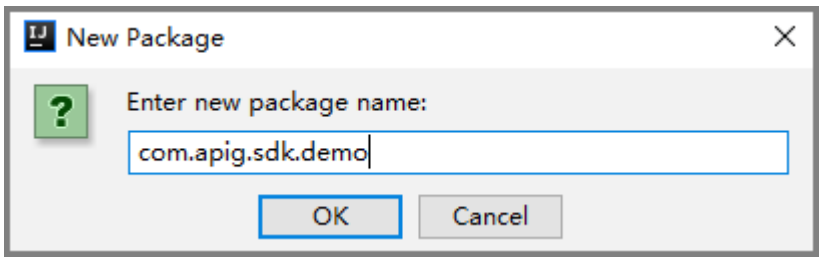
5. 新建“Package”及“Main”文件。
   a. 选择“src”，单击鼠标右键，选择“New > Package”。

图 2-12 新建 Package



   b. 在“Name”中输入“com.apig.sdk.demo”。

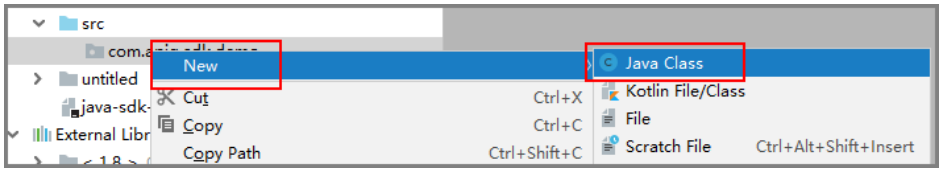
图 2-13 设置 Package 的名称



   c. 单击“OK”，完成“Package”的创建。

d. 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。

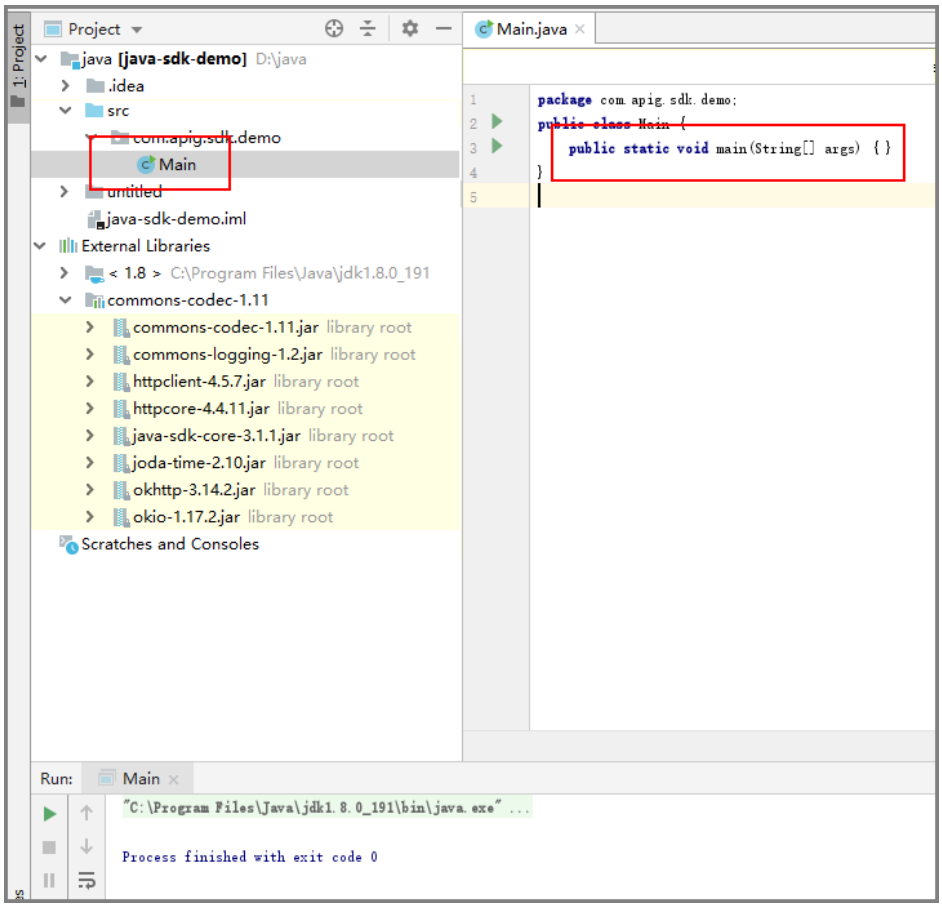
图 2-14 新建 Class



e. 配置Class。

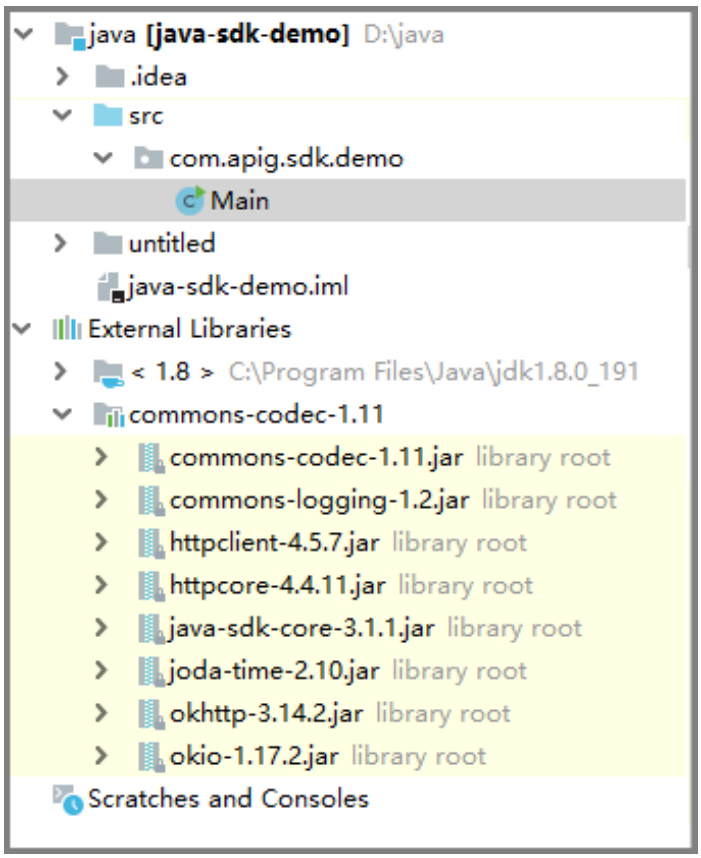
创建完成后，打开“Main”文件，添加“public static void main(String[] args)”。

图 2-15 设置 Class 的配置



6. 完成工程创建后，最终目录结构如下。

图 2-16 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。

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。

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;
```

2. 创建request，过程中需要用到如下参数，参数值可参考**获取API的调用信息**获取。
   - Key：API所授权凭据的Key。根据实际情况填写。
   - Secret：API所授权凭据的Secret。根据实际情况填写。

     说明

     认证使用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全；本示例以AK和SK保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量CLOUD_SDK_AK和CLOUD_SDK_SK。
   - Method：请求的方法。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。
   - QueryStringParam: 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
    {
        // Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
        // In this example, the AK/SK are stored in environment variables for identity authentication.
        // Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
        request.setKey(System.getenv("CLOUD_SDK_AK"));
        request.setSecret(System.getenv("CLOUD_SDK_SK"));
        request.setMethod("POST");
        request.setUrl("http://serviceEndpoint/java-sdk");
        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，需要取消本行代码的注释，增加自定义的环境名称
        request.setBody("demo");
    } catch (Exception e)
    {
        e.printStackTrace();
        return;
    }
```

3. 对请求进行签名，访问API并打印结果。

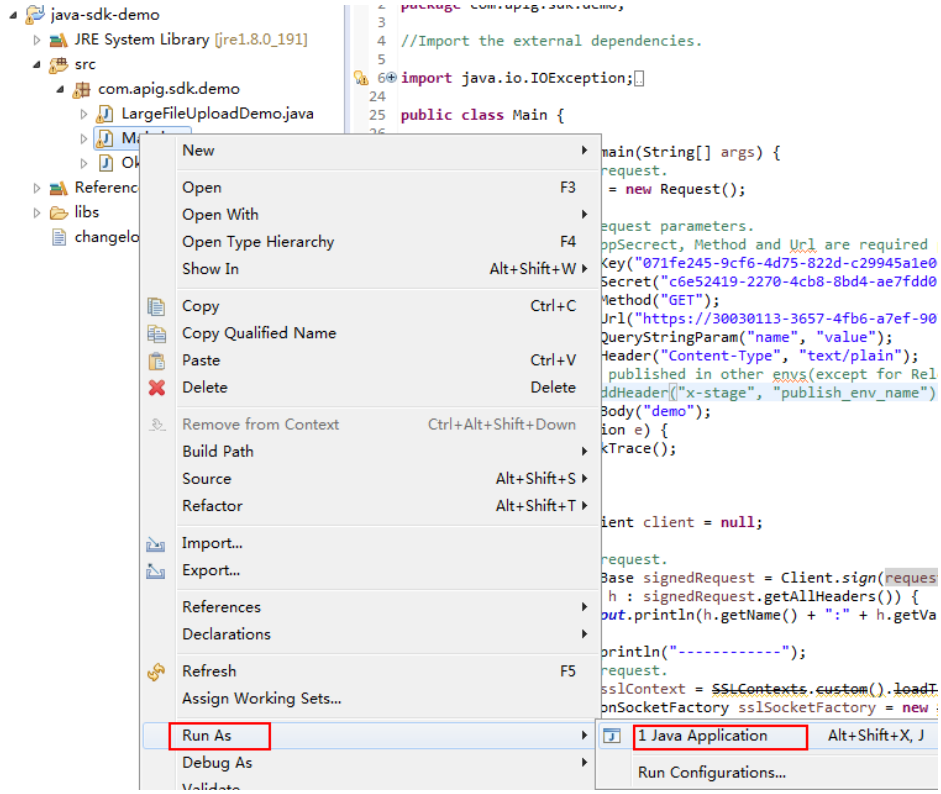
   样例代码如下：

```
    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();
        }
    }
```

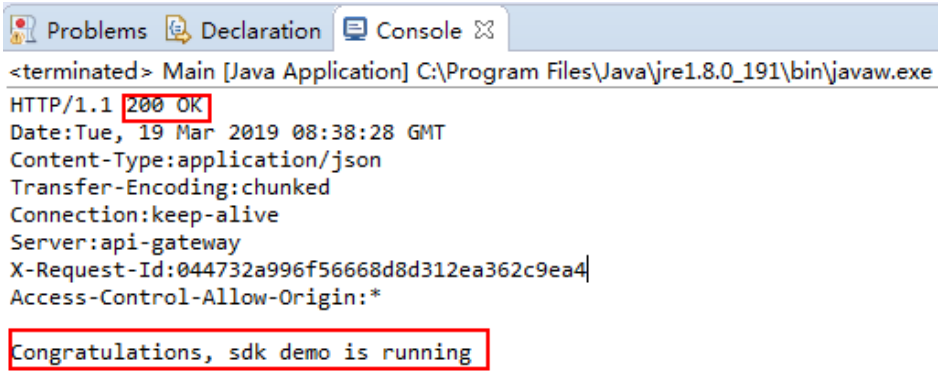
4. 选择“Main.java”，单击鼠标右键，选择“Run As > Java Application”，运行工程测试代码。

图 2-17 运行工程测试代码



5. 在“Console”页签，查看运行结果。

图 2-18 调用成功后的返回信息



2.2.4 Go SDK 使用说明

操作场景

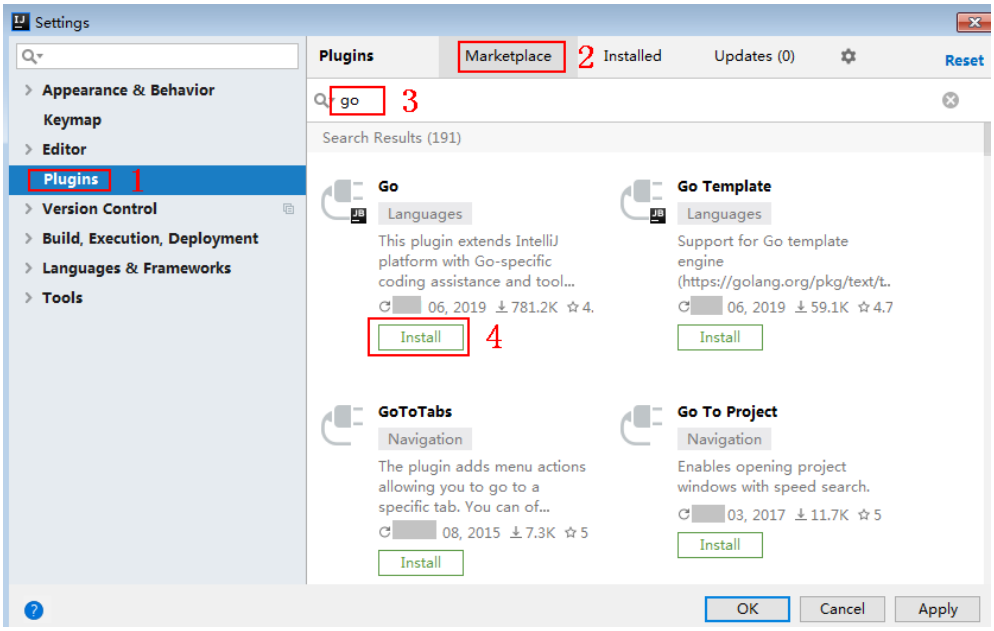
使用Go语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考调用API示例调用API。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

- 已在IntelliJ IDEA中安装Go插件，如果未安装，请按照图2-19所示安装。

图 2-19 安装 Go 插件



获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

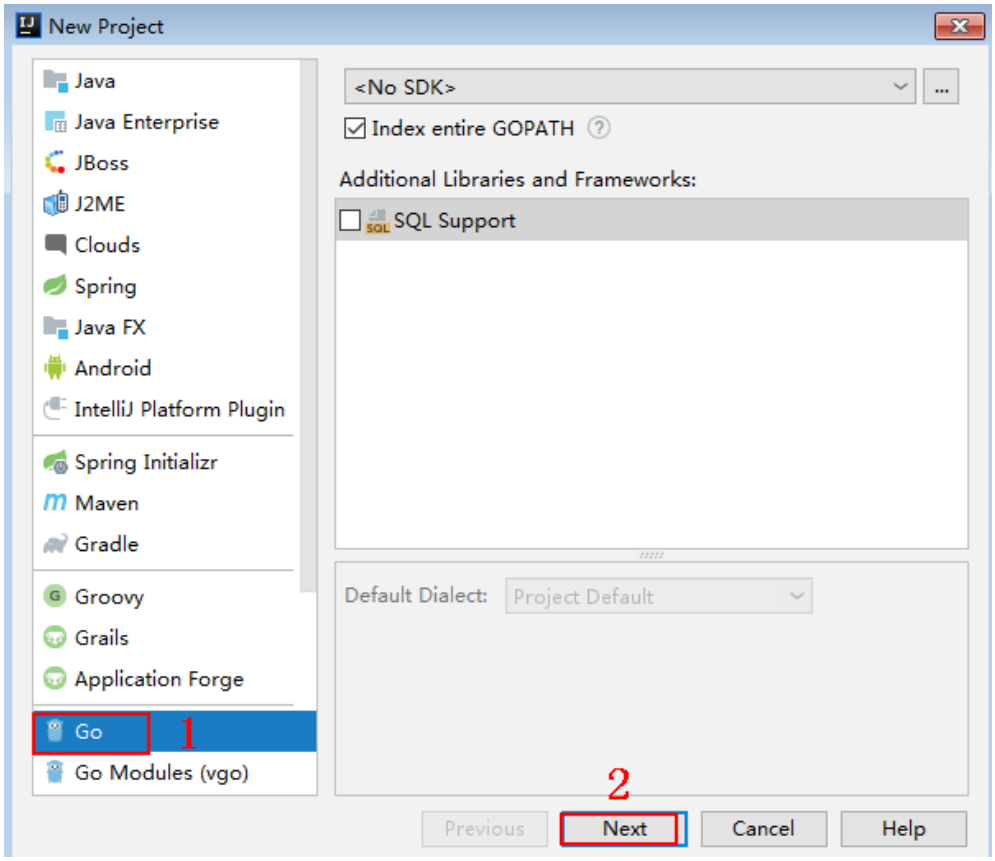
解压后目录结构如下：

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

新建工程

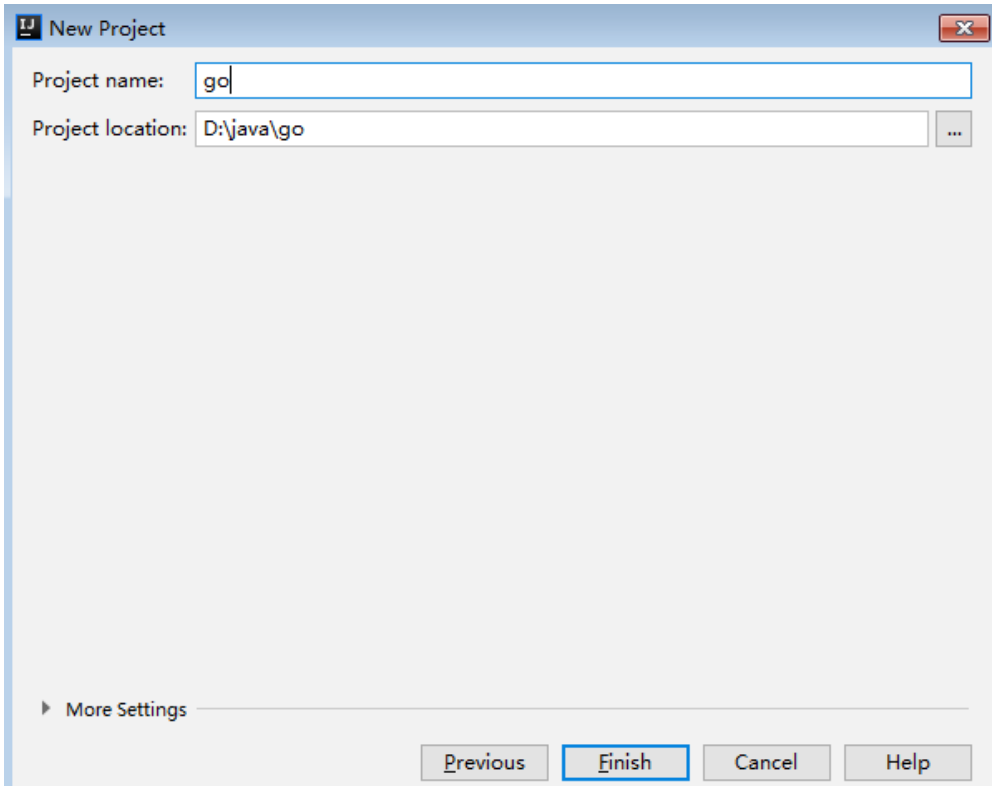
1. 打开IntelliJ IDEA，选择菜单“File > New > Project”。
   弹出“New Project”对话框，选择“Go”，单击“Next”。

图 2-20 New Project



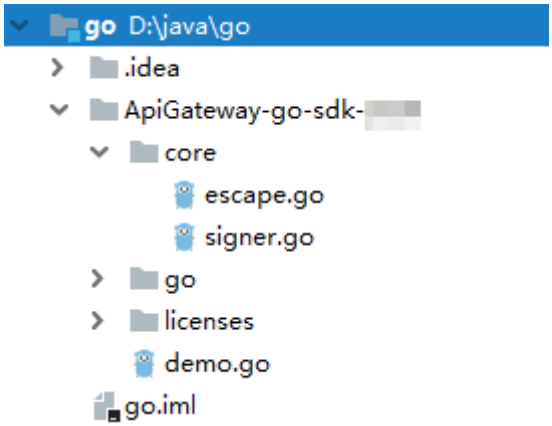
2. 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-21 选择解压后的 SDK 路径



3. 完成工程创建后，目录结构如下。

图 2-22 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**。

调用 API 示例

1. 在工程中引入sdk（signer.go）。
```
import "apig-sdk/go/core"
```
2. 生成一个新的Signer，输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。
```
s := core.Signer{
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the
```

```
configuration file or environment variables.
 // In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
        Key: os.Getenv("CLOUD_SDK_AK"),
        Secret:os.Getenv("CLOUD_SDK_SK"),
}
```

3. 生成一个新的Request，指定域名、方法名、请求url、query和body，可参考**获取API的调用信息**获取。

```
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"))))
```

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")
```

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)
```

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},
  },
}
```

7. 访问API，查看访问结果。

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

2.2.5 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。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。

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

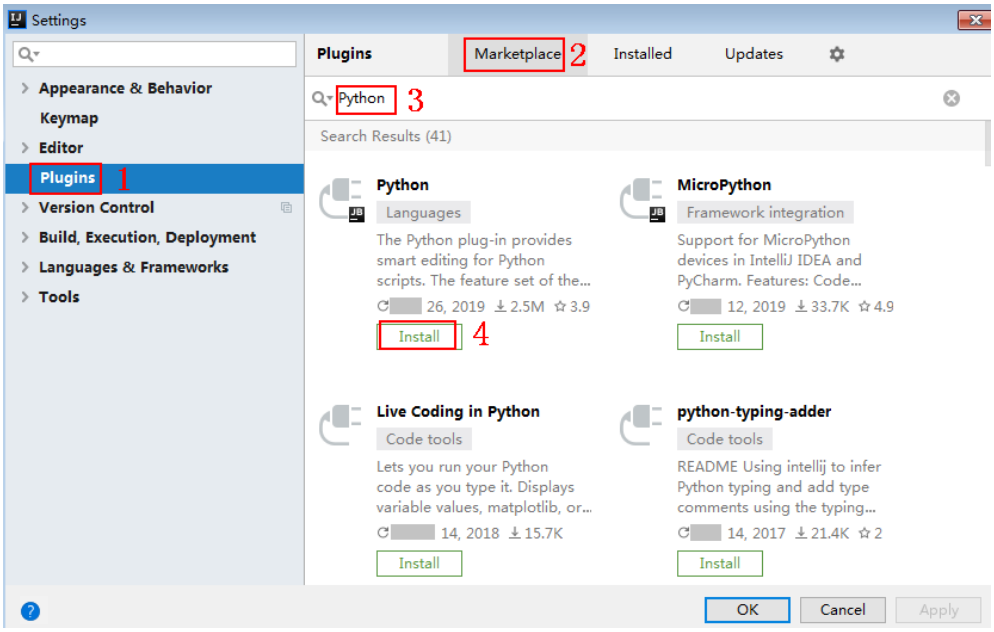
```
pip install requests
```

  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。

- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照**图2-23**所示安装。

图 2-23 安装 Python 插件



获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

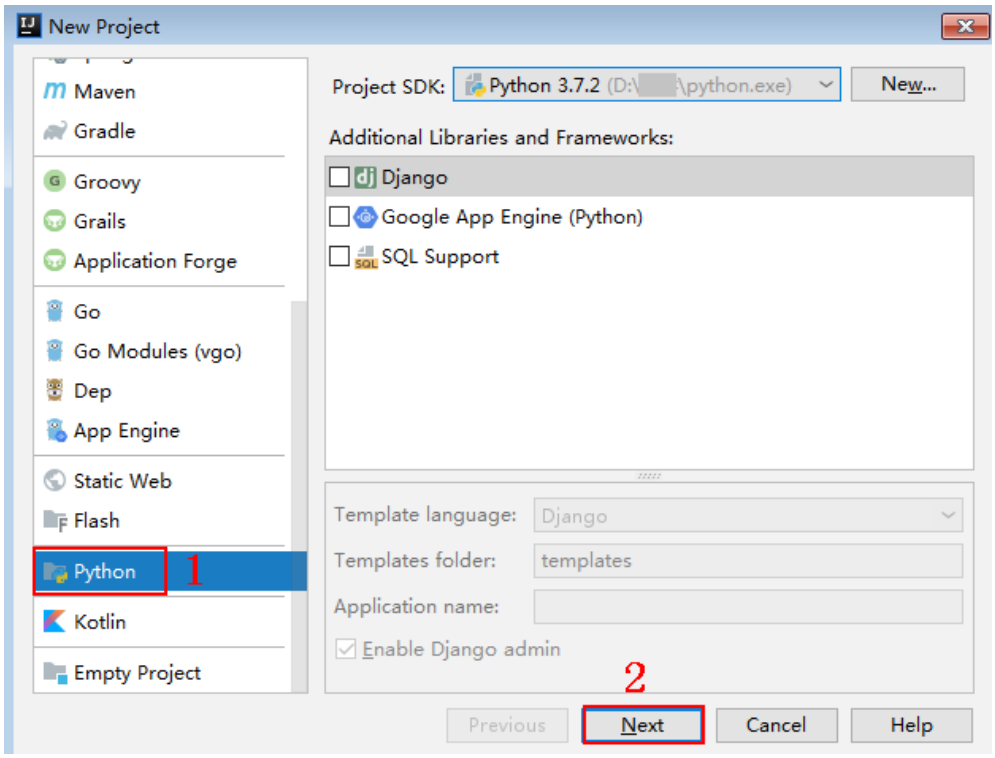
解压后目录结构如下：

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

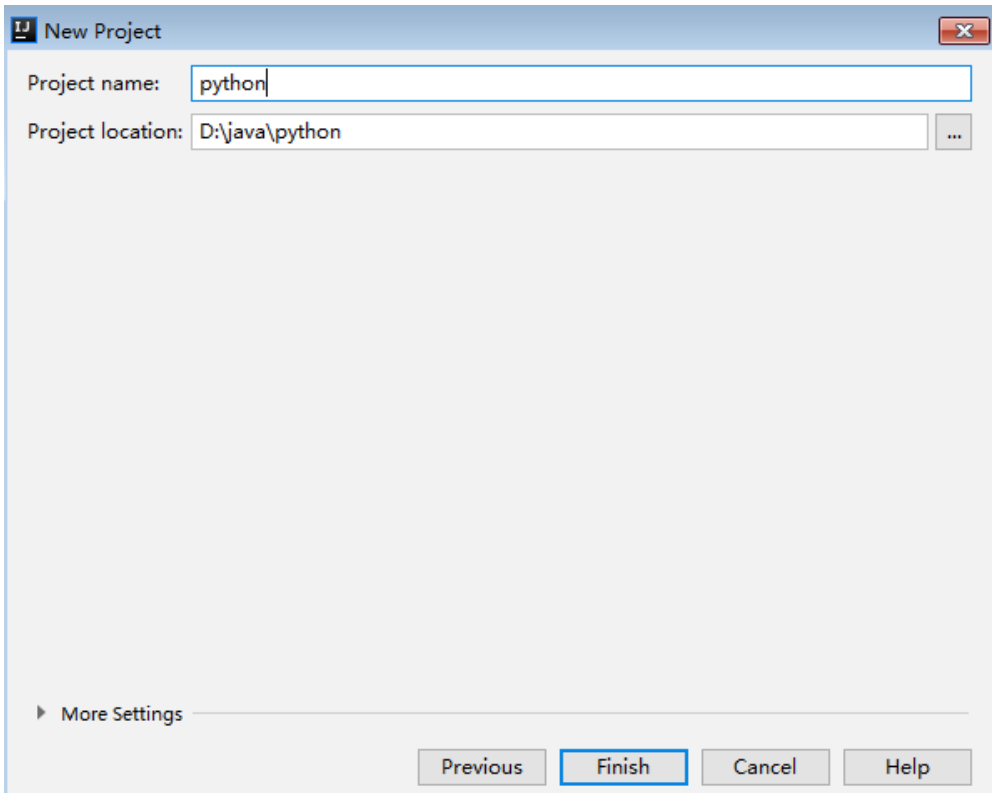
1. 打开IDEA，选择菜单“File > New > Project”。
   弹出“New Project”对话框，选择“Python”，单击“Next”。

图 2-24 New Project



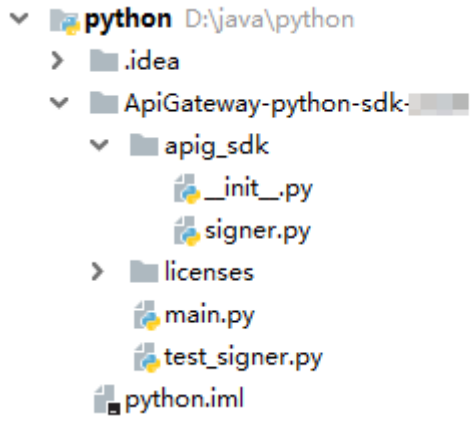
2. 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-25 选择解压后的 SDK 路径



3. 完成工程创建后，目录结构如下。

图 2-26 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**。

调用 API 示例

1. 在工程中引入apig_sdk。

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

2. 生成一个新的Signer，输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
sig = signer.Signer()
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
sig.Key = os.getenv('CLOUD_SDK_AK')
sig.Secret = os.getenv('CLOUD_SDK_SK')
```

3. 生成一个Request对象，指定方法名、请求url、header和body，可参考**获取API的调用信息**获取。

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

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)
```

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)
```

2.2.6 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。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装C#开发语言环境，具体请参见**开发准备**。

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载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

1. 在工程中引入sdk。
```
using APIGATEWAY_SDK;
```
2. 生成一个新的Signer， 输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
Signer signer = new Signer();
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
signer.Key = Environment.GetEnvironmentVariable("CLOUD_SDK_AK");
signer.Secret = Environment.GetEnvironmentVariable("CLOUD_SDK_SK");
```

3. 生成一个Request对象，指定方法名、请求url和body，可参考**获取API的调用信息**获取。

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

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");
```

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);
```

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; });
```

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());
```

2.2.7 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Node.js开发环境为例介绍。

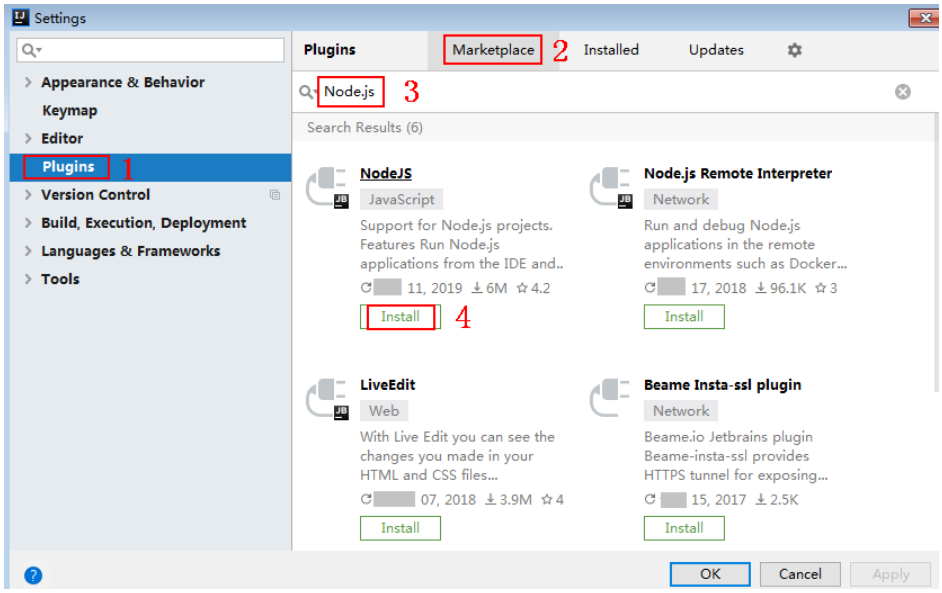
前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装开发工具和JavaScript开发语言环境，具体请参见**开发准备**。
  - 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
    ```

  - 已在IntelliJ IDEA中安装NodeJS插件，如果未安装，请按照**图2-27**所示安装。

图 2-27 安装 NodeJS 插件



- 浏览器版本为Chrome 89.0或以上版本。

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

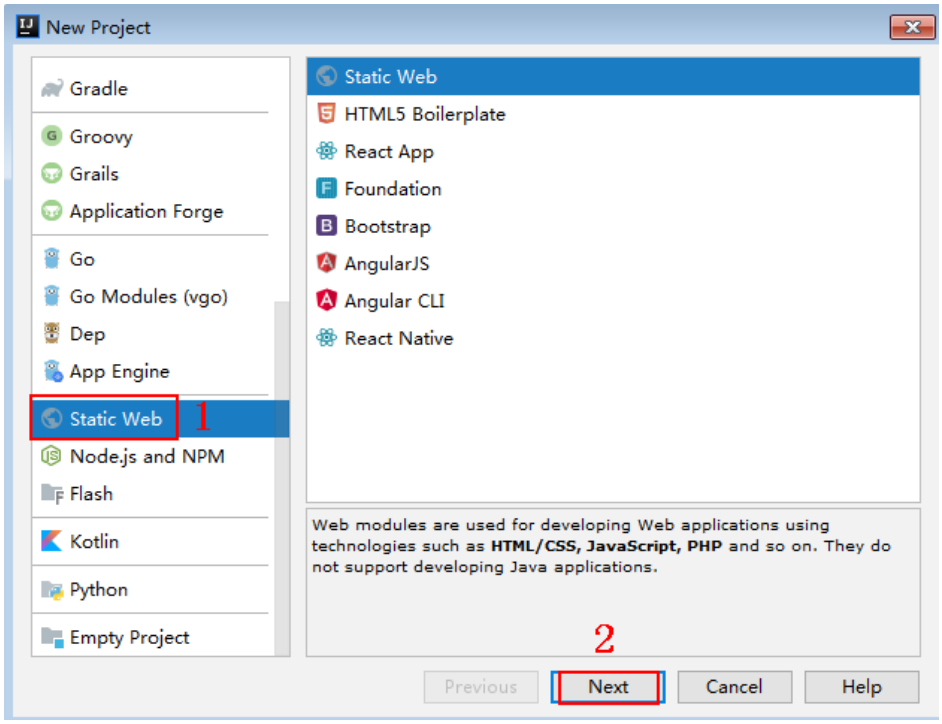
解压后目录结构如下：

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

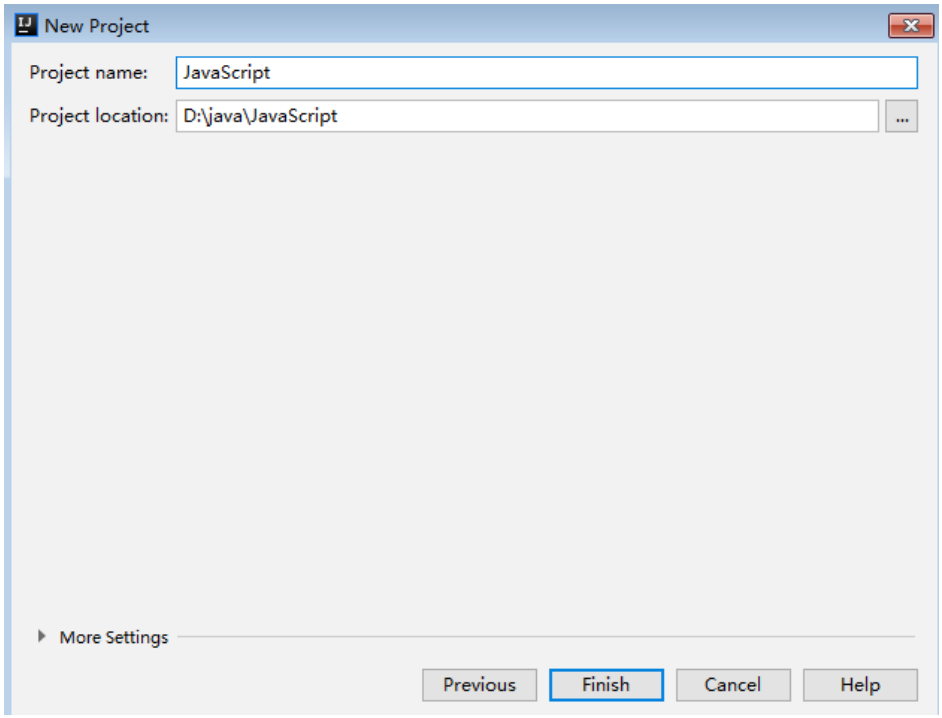
1. 打开IntelliJ IDEA，选择菜单“File > New > Project”。
   弹出“New Project”对话框。选择“Static Web”，单击“Next”。

图 2-28 New Project



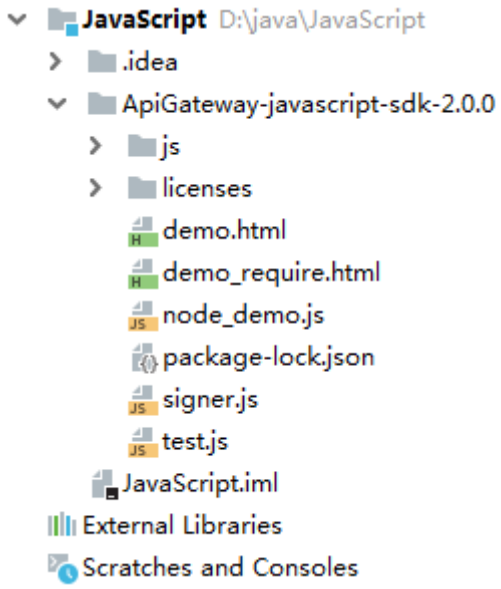
2. 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-29 选择解压后的 SDK 路径



3. 完成工程创建后，目录结构如下。

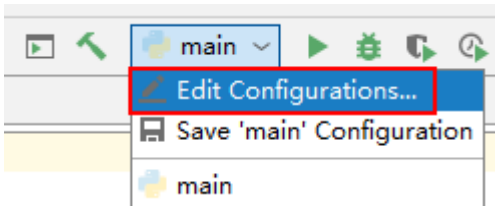
图 2-30 新建工程的目录结构



- 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**。

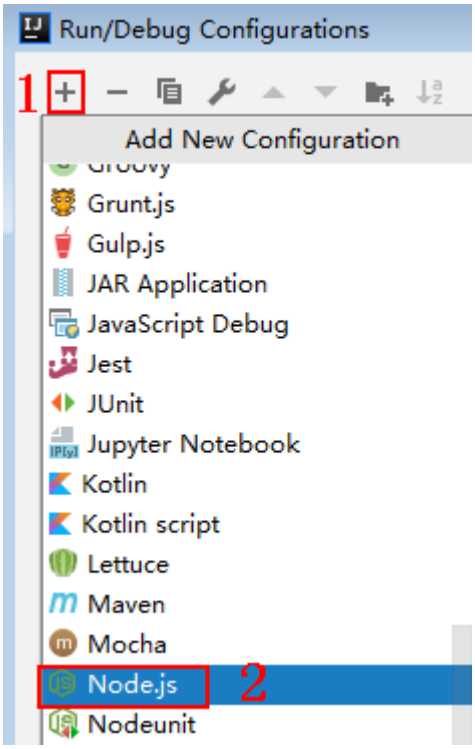
4. 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 2-31 Edit Configurations



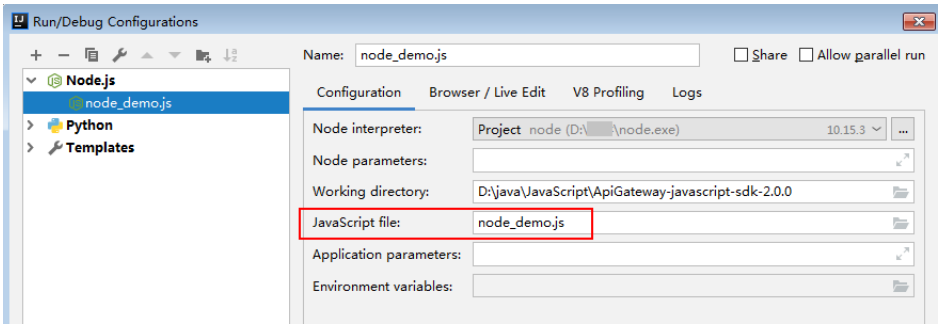
5. 单击“+”，选择“Node.js”。

图 2-32 选择 Node.js



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

图 2-33 选择 node_demo.js



调用 API（Node.js）示例

1. 在工程中引入signer.js。

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

2. 生成一个新的Signer，输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
var sig = new signer.Signer()
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
sig.Key = process.env.CLOUD_SDK_AK
sig.Secret = process.env.CLOUD_SDK_SK
```

3. 生成一个Request对象，指定方法名、请求url和body，可参考**获取API的调用信息**获取。

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

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" }
```

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)
```

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()
```

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。

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>
```

2. 进行签名和访问。

```
var sig = new signer.Signer()
sig.Key = process.env.CLOUD_SDK_AK
sig.Secret = process.env.CLOUD_SDK_SK
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
});
```

2.2.8 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。

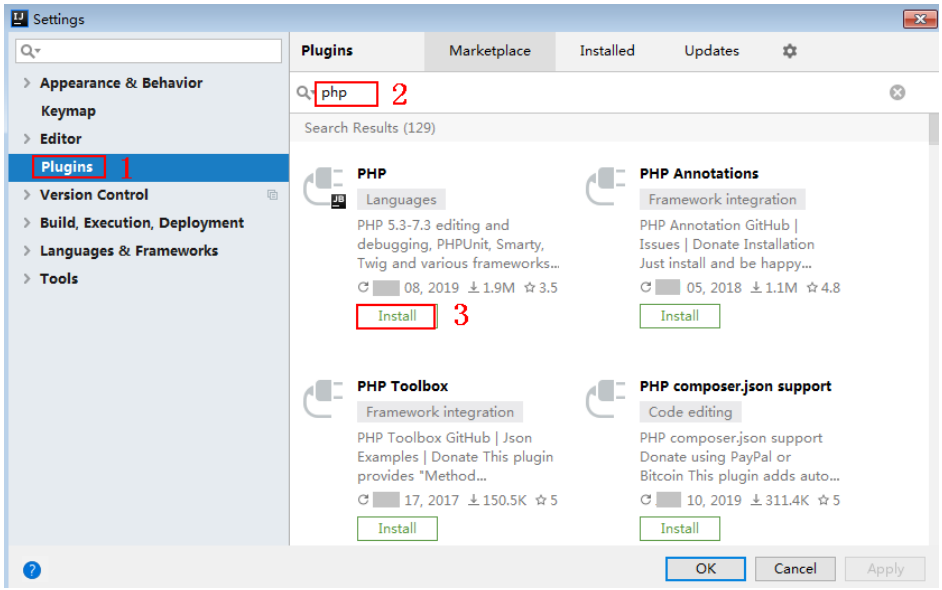
本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装开发工具和PHP开发语言环境，具体请参见**开发准备**。
  - 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2-34**所示安装。

    图 2-34 安装 PHP 插件

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

解压后目录结构如下：

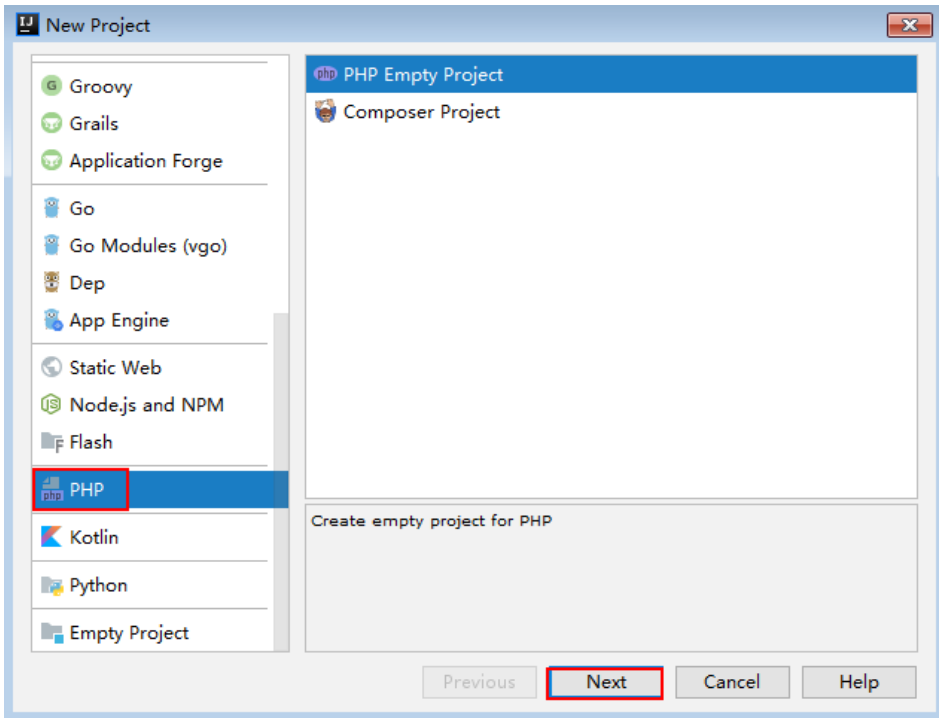
名称	说明
signer.php	SDK代码

名称	说明
index.php	示例代码

新建工程

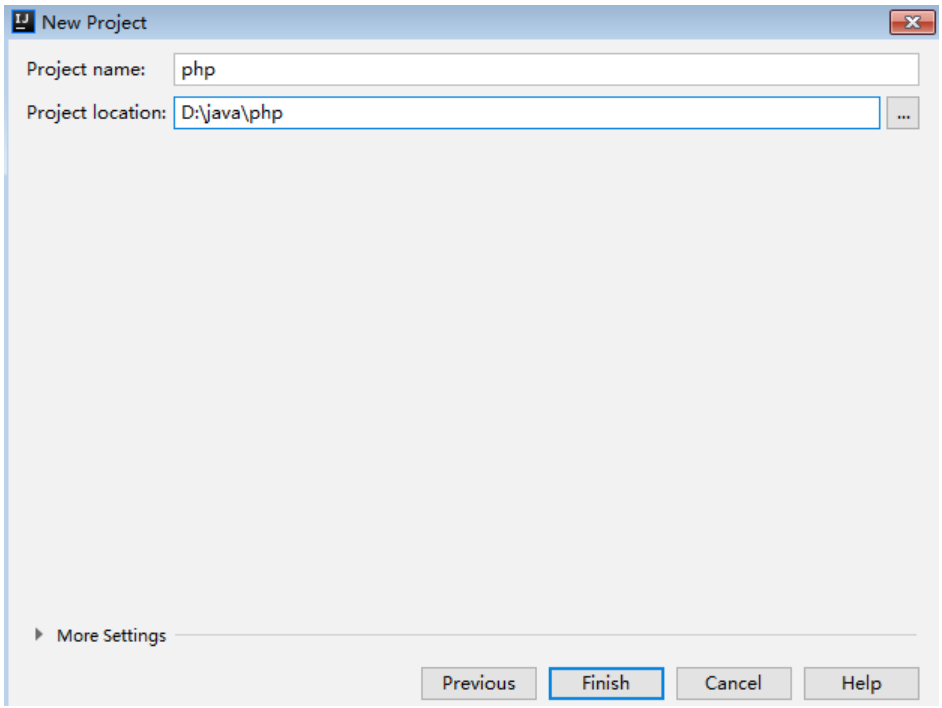
1. 打开IDEA，选择菜单“File > New > Project”。
   弹出“New Project”对话框，选择“PHP”，单击“Next”。

   图 2-35 New Project



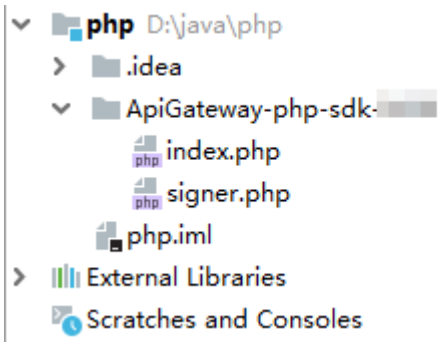
2. 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-36 选择解压后的 SDK 路径



3. 完成工程创建后，目录结构如下。

图 2-37 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**。

调用 API 示例

1. 在代码中引入sdk。

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

2. 生成一个新的Signer， 输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
$signer = new Signer();
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
$signer->Key = getenv('CLOUD_SDK_AK');
$signer->Secret = getenv('CLOUD_SDK_SK');
```

3. 生成一个新的Request，指定方法名、请求url和body，可参考**获取API的调用信息**获取。
```
$req = new Request('GET', "https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1?a=1");
$req->body = '';
```
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。
```
$req->headers = array(
    'x-stage' => 'RELEASE',
);
```
5. 进行签名，执行此函数会生成一个$curl上下文变量。
```
$curl = $signer->Sign($req);
```
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);
```
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);
```

2.2.9 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。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 安装openssl库。
```
apt-get install libssl-dev
```
- 安装curl库。
```
apt-get install libcurl4-openssl-dev
```

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载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	
signer.cpp	

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

调用 API 示例

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"
```

2. 生成一个新的Signer， 输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
Signer signer(getenv("CLOUD_SDK_AK"), getenv("CLOUD_SDK_SK"));
```

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，可参考**获取API的调用信息**获取。

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

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");
```

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);
```

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;
}
```

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。

2.2.10 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。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 安装openssl库。
  ```
  apt-get install libssl-dev
  ```
- 安装curl库。
  ```
  apt-get install libcurl4-openssl-dev
  ```

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载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

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"
```

2. 生成一个sig_params_t类型的变量，输入API所授权凭据的Key和Secret，可参考**获取API的调用信息**获取。

```
sig_params_t params;
sig_params_init(&params);
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
sig_str_t app_key = sig_str(getenv("CLOUD_SDK_AK"));
sig_str_t app_secret = sig_str(getenv("CLOUD_SDK_SK"));
params.key = app_key;
params.secret = app_secret;
```

3. 指定方法名、域名、请求uri、查询字符串和body，可参考**获取API的调用信息**获取。

```
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;
```

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");
```

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);
```

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;
}
```

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。

2.2.11 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。

前提条件

- 已获取API的调用信息，具体请参见**开发准备**。
- 已安装Android开发语言环境，具体请参见**开发准备**。

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

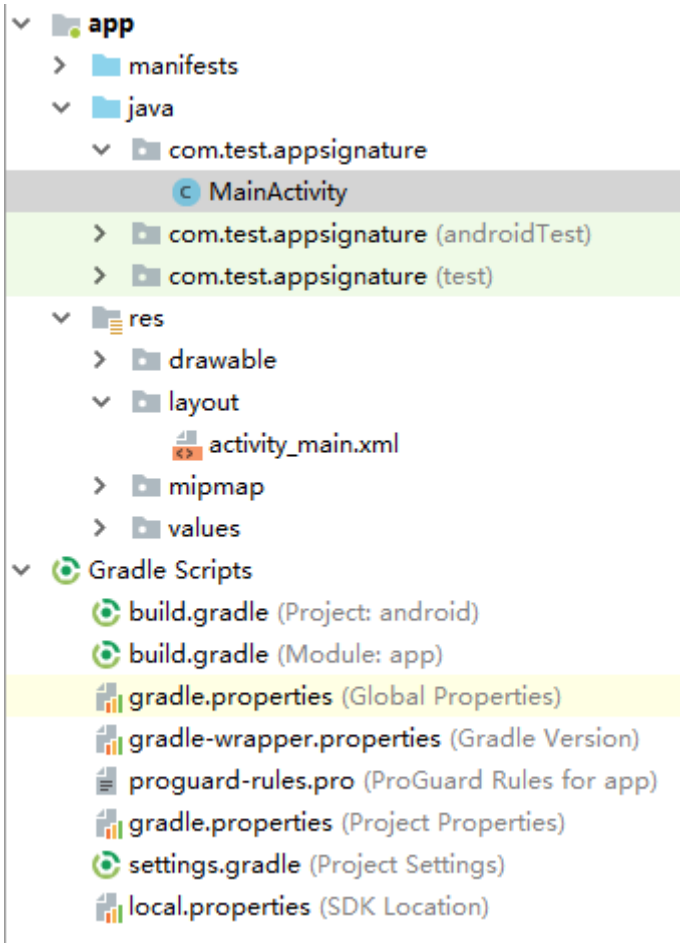
解压后目录结构如下：

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

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。
2. 打开工程后，目录结构如下。

图 2-38 工程目录结构



调用 API 示例

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
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'
}
```

3. 创建request，输入API所授权凭据的Key和Secret，并指定方法名、请求url和body，可参考**获取API的调用信息**获取。

```
Request request = new Request();
try {
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
    request.setKey(System.getenv("CLOUD_SDK_AK"));
    request.setSecrect(System.getenv("CLOUD_SDK_SK"));
```

```
    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;
}
```

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();
```

2.2.12 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的调用信息，具体请参见**开发准备**。
- 浏览器版本为Chrome 89.0或以上版本。

获取 SDK

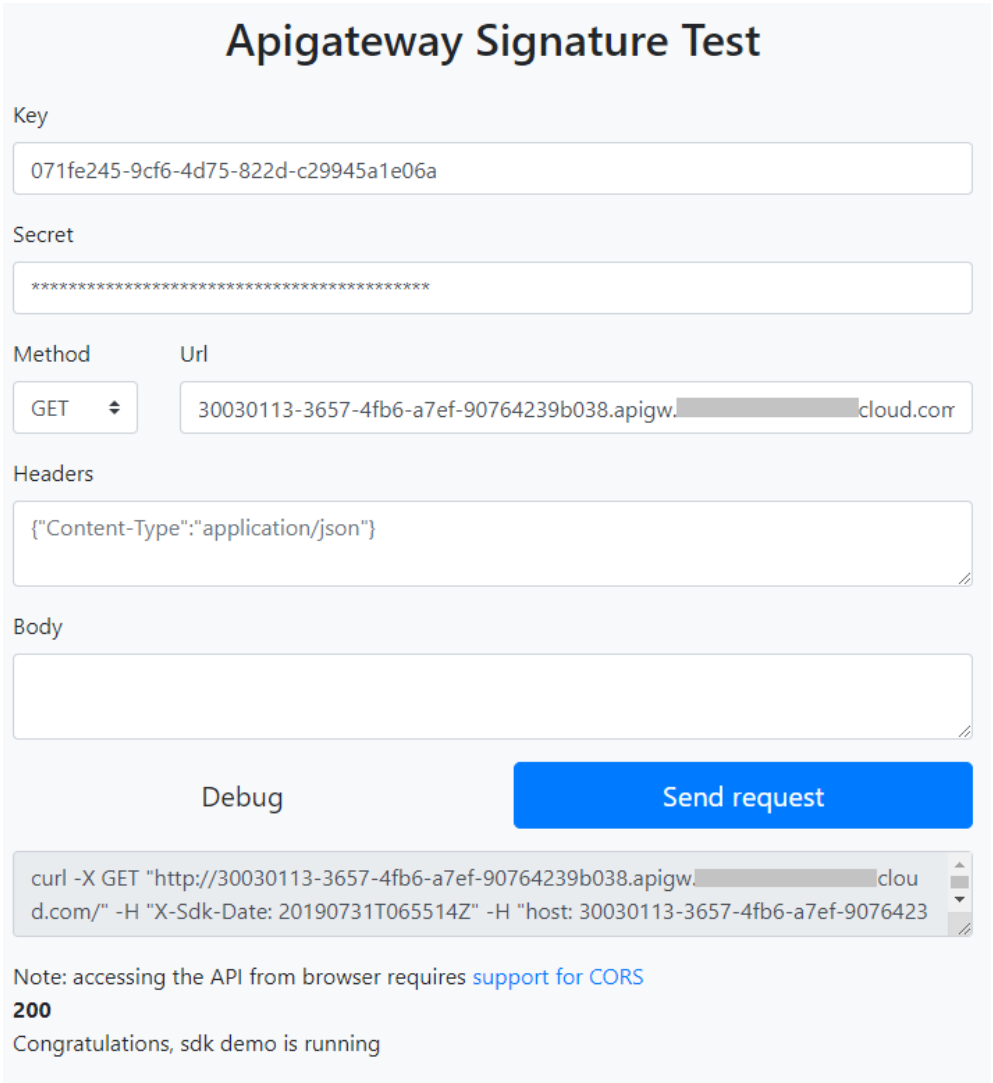
登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

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	

调用 API 示例

1. 使用JavaScript SDK生成curl命令。
   解压SDK，在浏览器中打开demo.html，页面如下图所示。



2. 输入API所授权凭据的Key和Secret，并指定方法名、请求url，可参考**获取API的调用信息**获取。例如：

```
// Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
// In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK and CLOUD_SDK_SK.
Key=4f5f626b-073f-402f-a1e0-e52171c6100c
Secret=******
Method=POST
Url=https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1
```

3. 输入json格式的Query和Headers，填入Body。
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/app1" -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。

2.3 API 调用认证开发（IAM 认证）

2.3.1 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}**请联系环境管理员获取。
   - ***username***为用户名。
   - **domainname**为用户所属的账号名称。
   - ********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。

2.3.2 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 2-39 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 2-40 访问密钥获取页面示意

生成签名

生成签名的方式和**API调用认证开发（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。

2.4 自定义后端开发（函数后端）

2.4.1 函数后端脚本开发说明

开发说明

函数后端通过编写函数脚本实现将多个服务封装成一个服务，在实际开发前请务必提前了解**自定义后端开发要求**。

函数后端当前仅支持使用JavaScript编写函数，JavaScript引擎运行于Java虚拟机，可调用ROMA Connect提供的**Java类**实现具体功能。

公共配置引用

自定义后端支持添加变量、密码、证书等全局公共配置，并在函数脚本中引用。公共配置的添加操作请参考《ROMA Connect 用户指南》的“服务集成指导 > 自定义后端管理 > 添加自定义后端公共配置”章节。

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")

2.4.2 AesUtils 类说明

路径

com.roma.apic.livedata.common.v1.AesUtils

说明

提供AES加解密方法。

使用示例

加密示例

```
importClass(com.roma.apic.livedata.common.v1.AesUtils);
function execute(data) {
  var plainText = "plainText";
  //加解密用的密钥硬编码到代码中有较大的安全风险，建议在配置文件中存放，本示例以在自定义后端的配置管理中配置的secretKey为例进行说明
  var secretKey = "#{secretKey}";
  var initialVector = "initialVector";
  var encryptText = AesUtils.encrypt(plainText, secretKey, initialVector, AesUtils.KEYLENGTH.KL_128, AesUtils.MODE.GCM, AesUtils.PAD.NOPADDING);
  return encryptText
}
```

解密示例

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

```
 var encryptText = "encryptText";
 //加解密用的密钥硬编码到代码中有较大的安全风险，建议在配置文件中存放，本示例以在自定义后端的配置管理中配置的secretKey为例进行说明
 var secretKey = "#{secretKey}";
 var initialVector = "initialVector";
 var decryptText = AesUtils.decrypt(encryptText, secretKey, initialVector, AesUtils.KEYLENGTH.KL_128, AesUtils.MODE.GCM, AesUtils.PAD.NOPADDING);
 return decryptText
}
```

方法列表

返回类型	方法和说明
static java.lang.String	decrypt(java.lang.String encryptText, java.lang.String secretKey) 使用密钥对密文进行AES解密。
static java.lang.String	encrypt(java.lang.String plainText, java.lang.String secretKey) 使用密钥对明文进行AES加密。
static java.lang.String	decrypt(java.lang.String encryptText, java.lang.String secretKey, java.lang.String initialVector, AesUtils.KEYLENGTH length, AesUtils.MODE mode, AesUtils.PAD pad) 使用指定模式和长度的密钥对密文进行AES解密。
static java.lang.String	encrypt(java.lang.String plainText, java.lang.String secretKey, java.lang.String initialVector, AesUtils.KEYLENGTH length, AesUtils.MODE mode, AesUtils.PAD pad) 使用指定指定模式和长度的密钥对明文进行AES加密。

方法详情

- **public static java.lang.String decrypt(java.lang.String encryptText, java.lang.String secretKey)**

  使用密钥对密文进行AES解密

  输入参数

  - encryptText：待解密的密文，长度不超过8192
  - secretKey：密钥

  返回信息

  返回解密后的数据

- **public static java.lang.String encrypt(java.lang.String plainText, java.lang.String secretKey)**

  使用密钥对明文进行AES加密

  输入参数

  - plainText：待加密的明文，长度不超过4096
  - secretKey：密钥

  返回信息

  返回加密后的数据

- **public static java.lang.String decrypt(java.lang.String encryptText, java.lang.String secretKey, java.lang.String initialVector, AesUtils.KEYLENGTH length, AesUtils.MODE mode, AesUtils.PAD pad)**

  使用指定模式和长度的密钥对密文进行AES解密

  输入参数

  - encryptText：待解密的密文，长度不超过8192
  - secretKey：密钥
  - initialVector：初始向量
  - length：密钥的长度，取值范围：AesUtils.KEYLENGTH.KL_0、AesUtils.KEYLENGTH.KL_128、AesUtils.KEYLENGTH.KL_192、AesUtils.KEYLENGTH.KL_256
  - mode：工作模式，取值范围：AesUtils.MODE.GCM、AesUtils.MODE.CTR
  - pad：填充模式，取值范围：AesUtils.PAD.PKCS5PADDING，AesUtils.PAD.NOPADDING

  返回信息

  返回解密后的数据

- **public static java.lang.String encrypt(java.lang.String plainText, java.lang.String secretKey, java.lang.String initialVector, AesUtils.KEYLENGTH length, AesUtils.MODE mode, AesUtils.PAD pad)**

  使用指定指定模式和长度的密钥对明文进行AES加密

  输入参数

  - plainText：待加密的明文，长度不超过4096
  - secretKey：密钥
  - initialVector：初始向量
  - length：密钥的长度，取值范围：AesUtils.KEYLENGTH.KL_0、AesUtils.KEYLENGTH.KL_128、AesUtils.KEYLENGTH.KL_192、AesUtils.KEYLENGTH.KL_256
  - mode：工作模式，取值范围：AesUtils.MODE.GCM、AesUtils.MODE.CTR
  - pad：填充模式，取值范围：AesUtils.PAD.PKCS5PADDING，AesUtils.PAD.NOPADDING

  返回信息

  返回加密后的数据

2.4.3 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

返回类型	方法和说明
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

2.4.4 ApicShaUtils 类说明

路径

com.roma.apic.livedata.common.v1.ApicShaUtils

说明

提供SHA算法的摘要处理。

使用示例

```
importClass(com.roma.apic.livedata.common.v1.ApicShaUtils);
function execute(data) {
  return ApicShaUtils.encode(ApicShaUtils.SHA_256, "Hello world!");
}
```

方法列表

返回类型	方法和说明
static String	**encode**(String algorithm, String content) 对字符串进行指定SHA算法的摘要处理
static String	**sha256**(String content) 对字符串进行SHA256算法的摘要处理
static String	**sha384**(String content) 对字符串进行SHA384算法的摘要处理
static String	**sha512**(String content) 对字符串进行SHA512算法的摘要处理

方法详情

- **public static String encode(String algorithm, String content)**

  对字符串进行指定SHA算法的摘要处理

  输入参数

  - algorithm：指定使用的SHA算法，可设置为“ApicShaUtils.SHA_256”、“ApicShaUtils.SHA_384”和“ApicShaUtils.SHA_512”
  - content：需要进行SHA算法处理的字符串

  返回信息

  返回处理后的HEX字符串

- **public static String sha256(String content)**

  对字符串进行SHA256算法的摘要处理

  输入参数

  content：需要进行SHA256算法处理的字符串

  返回信息

返回处理后的HEX字符串

- **public static String sha384(String content)**
  对字符串进行SHA384算法的摘要处理
  输入参数
  content：需要进行SHA384算法处理的字符串
  返回信息
  返回处理后的HEX字符串
- **public static String sha512(String content)**
  对字符串进行SHA512算法的摘要处理
  输入参数
  content：需要进行SHA512算法处理的字符串
  返回信息
  返回处理后的HEX字符串

2.4.5 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

Base64编码示例：

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

multipart-formdata上传文件示例：

```
importClass(com.roma.apic.livedata.common.v1.Base64Utils);
function execute(data) {
  var image = data.body.get("image")
  return {
    size: image.getSize(),
    name: image.getFileItem().getName(),
    base64: Base64Utils.encode(image.getFileItem().get())
  }
}
```

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

返回类型	方法和说明
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

2.4.6 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。

使用示例

在使用CacheUtils之前，需要new一个新对象方可调用。

CacheUtils的get方法只允许使用白名单里的几项作为键：

```
"DICT:api_gw_rest_addr"、"DICT:api_gw_rest_float_addr"、"DICT:api_gw_rest_eip_addr"、
"DICT:livedata_private_address"
```

示例：

```
importClass(com.huawei.livedata.lambdaservice.util.CacheUtils);
function execute(data) {
  var cacheUtils = new CacheUtils
  var value = cacheUtils.get("DICT:livedata_private_address")
  return value
}
```

返回的结果为livedata的内网地址。

putCache方法和getCache方法可以实现缓存的使用示例：

```
importClass(com.huawei.livedata.lambdaservice.util.CacheUtils);
function execute(data) {
  var cacheUtils = new CacheUtils
  code = cacheUtils.putCache("age", "20")
  if (code != true) {
    return code
  }
  var name = cacheUtils.getCache("age")
```

```
    return name
}
```

返回的结果为20。

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
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，单位秒。超时后缓存的信息将被删除，获取该缓存信息将返回空值。

  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

2.4.7 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",
}
```

2.4.8 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。

2.4.9 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);
}
```

```
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("oracle")
  config.setUrl("jdbc:oracle:thin:@127.0.0.1:1521/db")
  config.setUser("username")
  config.setPassword("password")
  var ds = new DataSourceClient(config)
  return ds.execute("select table_name from user_tables");
}
```

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

2.4.10 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
String	**getUrl**() 获取连接字符串。
String	**getUser**() 获取用户名
String	**getPassword**() 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

2.4.11 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。

2.4.12 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。

说明

部分HTTP通用响应头不支持返回，如Location等。

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
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

2.4.13 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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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

2.4.14 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

返回类型	方法和说明
int	**getMaxWait**() 获取jedis连接池耗尽时等待时间上限，默认60秒
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
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

返回类型	方法和说明
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秒
  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秒

  返回信息

  返回soTimeout

- **public int getTimeout()**

  获取jedis的超时时间，默认1000秒

  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秒

  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秒
  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，单位秒

2.4.15 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

2.4.16 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

2.4.17 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

返回类型	方法和说明
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

2.4.18 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

2.4.19 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，最大值为30000毫秒，建议timeout设置值小于前端API的“后端超时”时间值

- 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

2.4.20 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

2.4.21 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）。
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）。

方法详情

- **public static KafkaConfig getConfig(String servers, String groupId)**
  访问MQS提供的kafka（不开启sasl_ssl）。
  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）。

  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

2.4.22 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

2.4.23 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

2.4.24 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。

2.4.25 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。

2.4.26 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

2.4.27 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表示传入RedisClient的配置信息

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
  redis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
  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，单位为秒
  - 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
  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

2.4.28 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

2.4.29 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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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**参数说明**。

  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**参数说明**。

  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：需要加密的内容

  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：指定加密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见**参数说明**。

- **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：指定加密的算法/模式/填充。例如："RSA/ECB/OAEPPadding"。详见**参数说明**。

  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

2.4.30 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

2.4.31 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

2.4.32 SoapClient 类说明

路径

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

说明

使用此类进行SOAP请求。

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

public SoapClient(SoapConfig soapCfg)

构造一个包含**SoapConfig**配置信息的SOAP请求。

参数：soapCfg表示传入SoapClient的配置信息。

方法列表

返回类型	方法和说明
APIConnectResponse	execute() 用于发送SOAP请求

2.4.33 SoapConfig 类说明

路径

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

说明

配合**SoapClient**使用，对SOAP请求进行配置。

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
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

返回类型	方法和说明
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
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

返回类型	方法和说明
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

2.4.34 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

返回类型	方法和说明
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

2.4.35 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

2.4.36 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

2.5 自定义后端开发（数据后端）

2.5.1 SQL 语法

数据后端与各数据库的 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
  ```

- 如果配置数据后端时勾选了“预编译”，在使用传入的参数进行模糊匹配查询时，若匹配字段中带有字符“%”，则需要使用concat函数进行拼接。

  如以下示例，name为String类型参数。

  ```
  select * from table01 where name like concat('%',${name})
  ```

  说明

  若数据源开启了“预编译”，且在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类型参数进行数据类型转换。

  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）

- 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
```

2.5.2 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 2-2 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
    ]
  }
  ```

2.5.3 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 2-3 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_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。假设数据源1的返回对象为“default1”。

```
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user_age。假设数据源2的返回对象为“default2”。

```
select user_age from user02 where user_id=${default1[0].id};
```

其中，${**default1**[0].id}为数据源1的查询结果（default1为数据源1的返回对象，id为数据源1的查询字段）。

假设，数据表user01和user02中有如下数据记录：

user01		user02		
id	**name**	**user_id**	**user_age**	**user_sex**
1	zhang	2	17	女
2	li	3	18	男
3	wang	1	18	男

则调用该数据API将返回如下响应：

```
{
    "default1":[{
        "id":1
    }],
    "default2":[{
        "user_age":18
    }]
}
```

2.5.4 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';
  ```

2.6 后端服务签名校验开发

2.6.1 开发准备

获取签名密钥信息

登录ROMA Connect实例控制台，在“服务集成 APIC > API策略”的“策略管理”页签中，在右上角的过滤条件选择“签名密钥”，单击API所绑定签名密钥的名称进入签名密钥详情页面，获取签名密钥的Key和Secret。

准备开发环境

- 安装开发工具

  请根据所使用语言选择适合的开发工具。
  - 安装2018.3.5或以上版本的IntelliJ IDEA，安装包请至**IntelliJ IDEA官方网站**下载。
  - 安装2019 version 16.8.4及以上版本的Visual Studio，安装包请至**Visual Studio官方网站**下载。
- 安装开发语言环境
  - Java：安装1.8.111或以上版本的Java Development Kit，安装包请至**Oracle官方下载页面**下载。
  - Python：安装2.7或3.X版本的Python安装包，安装包请至**Python官方下载页面**下载。

2.6.2 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。

前提条件

- 已ROMA Connect控制台创建签名密钥，并绑定API，具体请参见《ROMA Connect用户指南》的“配置后端服务的签名校验”章节。
- 已获取签名密钥的Key和Secret，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

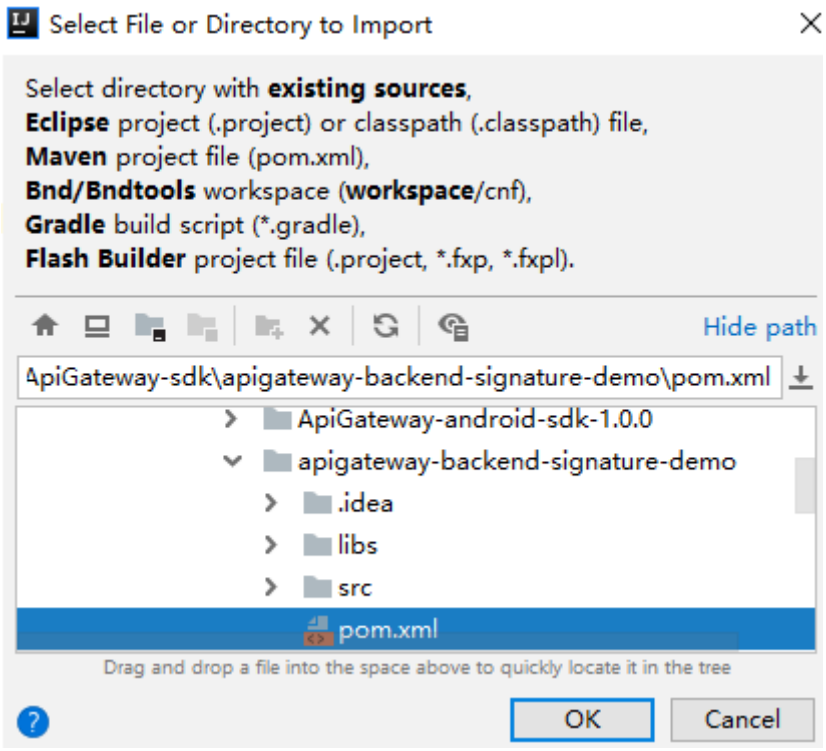
获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

导入工程

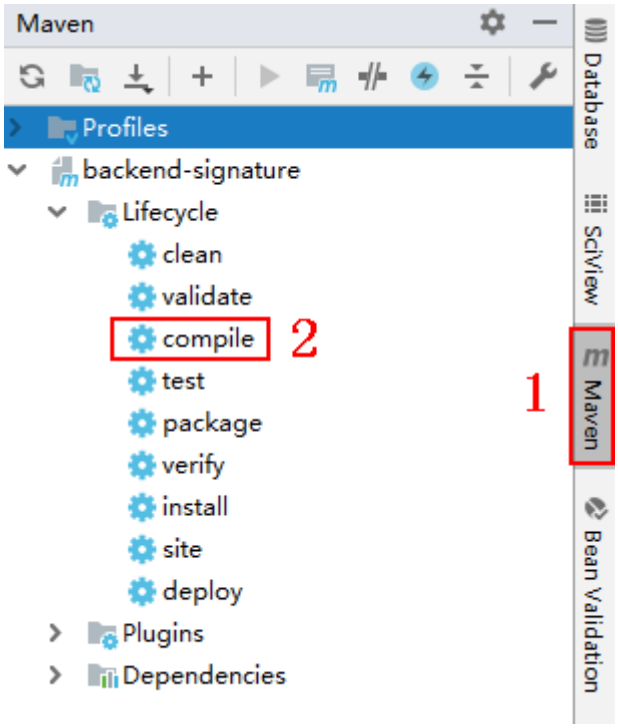
1. 打开IntelliJ IDEA，在菜单栏选择“File > New > Project from Existing Sources”，选择解压后的“apigateway-backend-signature-demo\pom.xml”文件，单击“OK”。

   图 2-41 Select File or Directory to Import

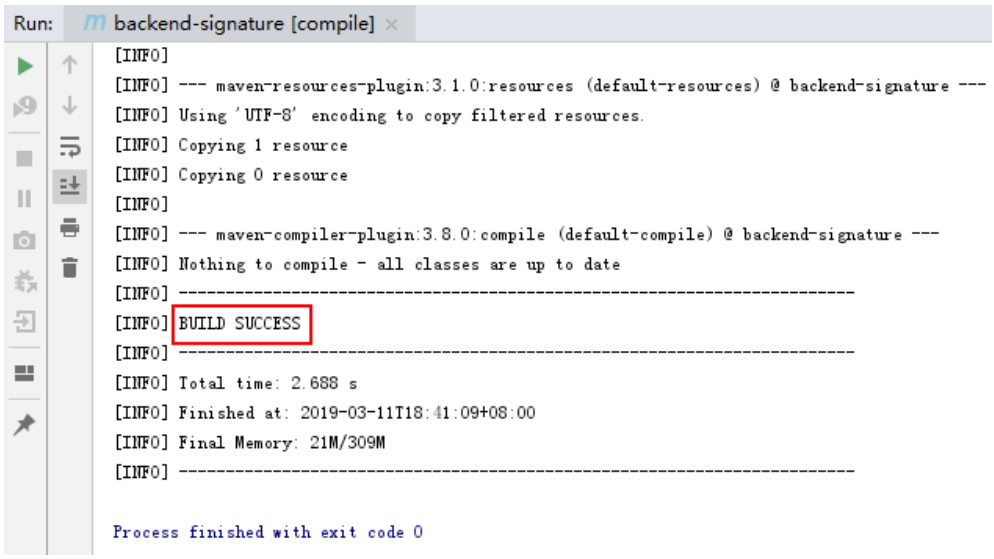
   

2. 保持默认设置，单击“Next > Next > Next > Next > Finish”，完成工程导入。
3. 在右侧Maven页签，双击“compile”进行编译。

图 2-42 编译工程

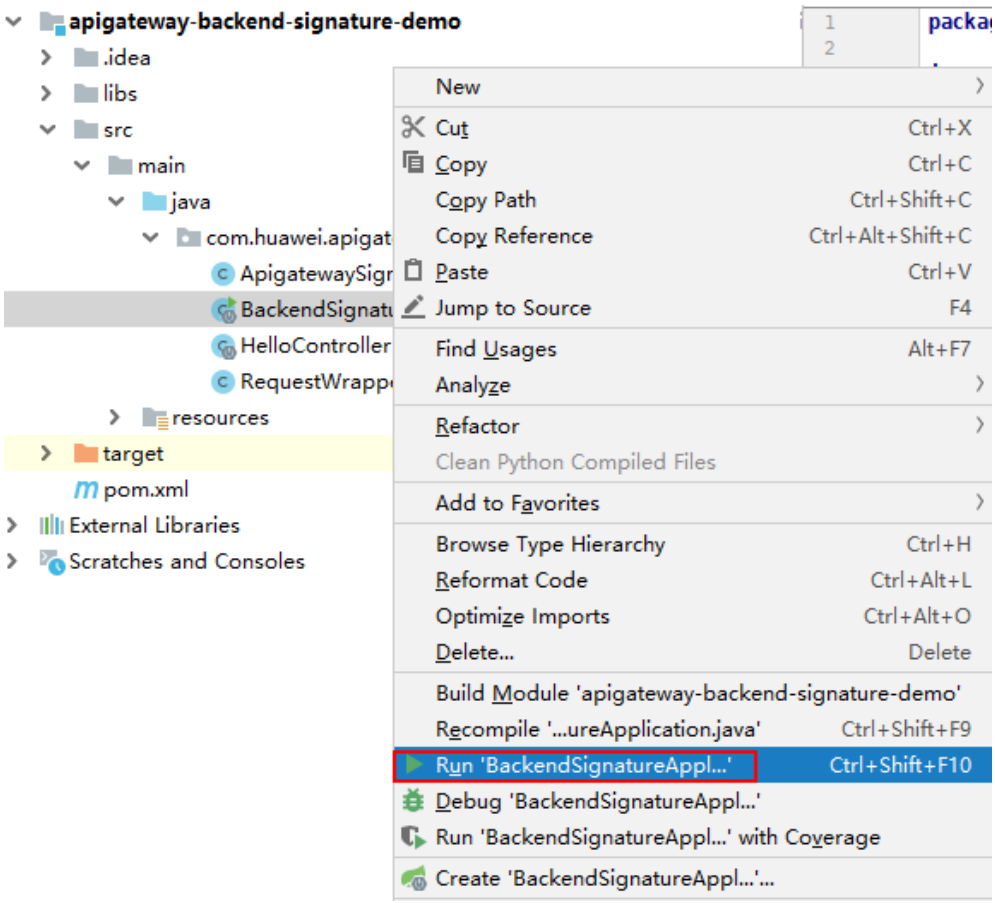


返回“BUILD SUCCESS”，表示编译成功。



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

图 2-43 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**。

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。

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";
    }
}
```

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 {
        // Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the
configuration file or environment variables.
        // In this example, the AK/SK are stored in environment variables for identity authentication.
Before running this example, set environment variables CLOUD_SDK_AK1, CLOUD_SDK_SK1, and
CLOUD_SDK_AK2, CLOUD_SDK_SK2.
        secrets.put(System.getenv("CLOUD_SDK_AK1"), System.getenv("CLOUD_SDK_SK1"));
        secrets.put(System.getenv("CLOUD_SDK_AK2"), System.getenv("CLOUD_SDK_SK2"));
    }

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

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

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);
```

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
```

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;
}
```

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
}
```

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;
}
```

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.");
}
```

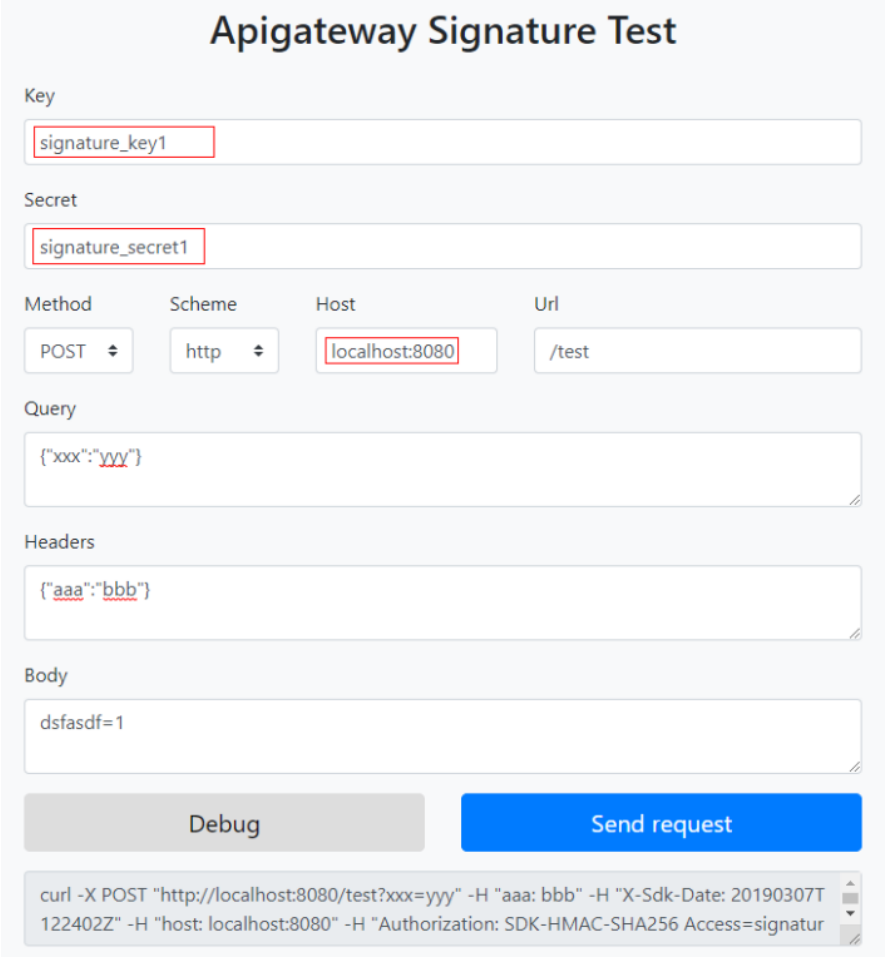
9. 注册filter和路径的映射关系。

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

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。



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。

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";
    }
}
```

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) {
            }
        }
    }
}
```

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;
    }
}
```

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。

2.6.3 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。

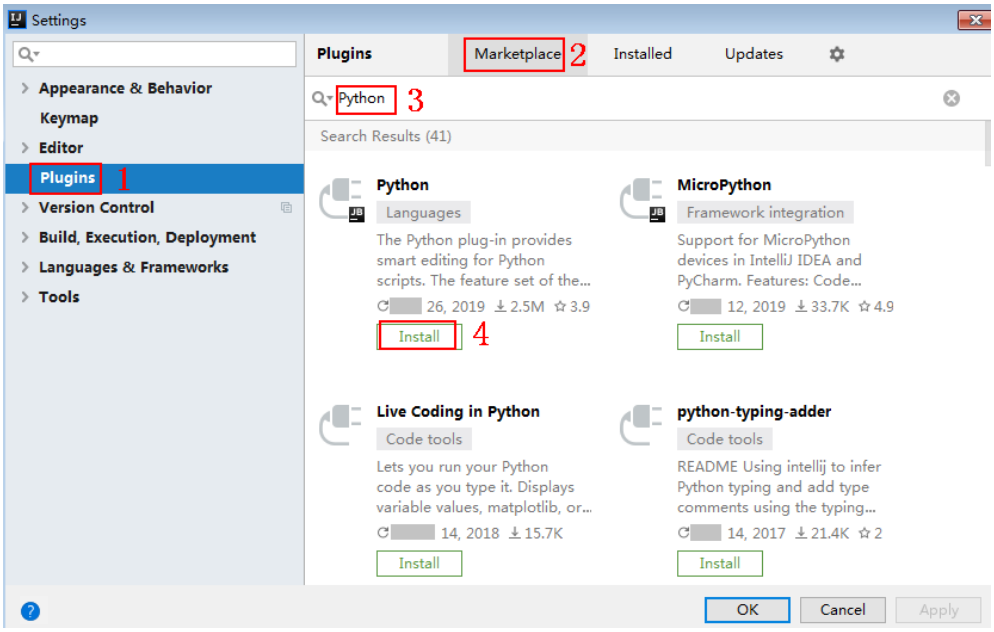
说明

Python SDK仅支持hmac类型的后端服务签名。

前提条件

- 已在控制台创建签名密钥，并绑定API，具体请参见《 ROMA Connect用户指南 》的“配置后端服务的签名校验”章节。
- 已获取签名密钥的Key和Secret，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。
- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照**图2-44**所示安装。

图 2-44 安装 Python 插件



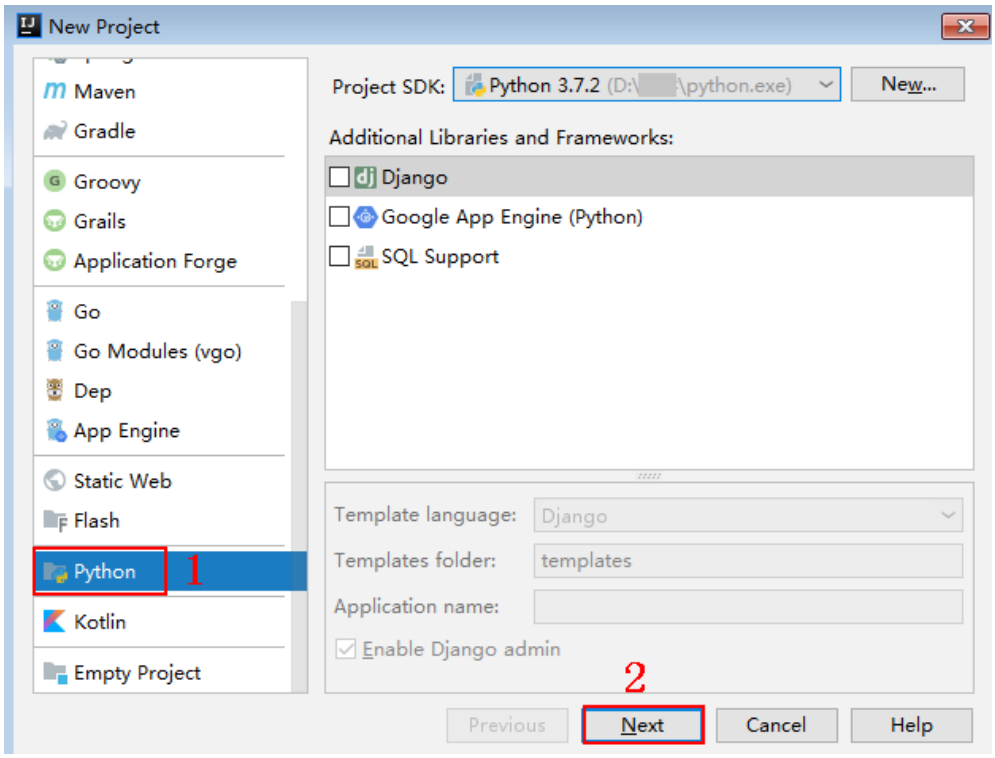
获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

导入工程

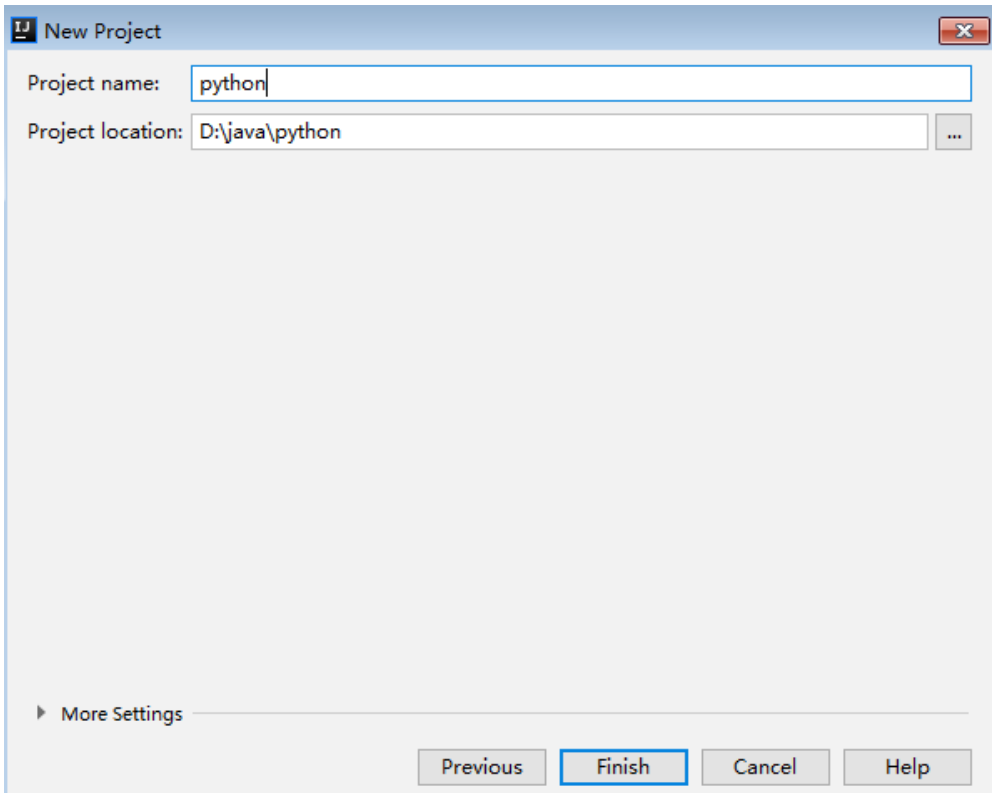
1. 打开IntelliJ IDEA，在菜单栏选择“File > New > Project”。
   弹出“New Project”对话框，选择“Python”，单击“Next”。

图 2-45 New Project



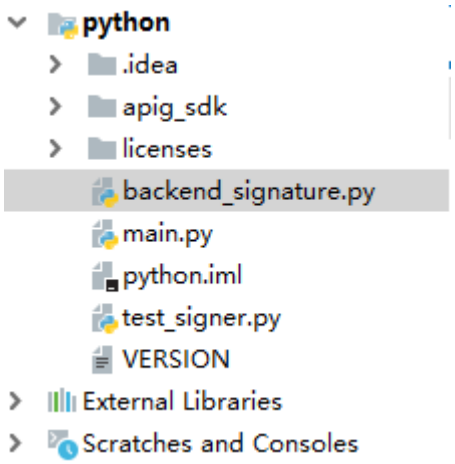
2. 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-46 选择解压后的 SDK 路径



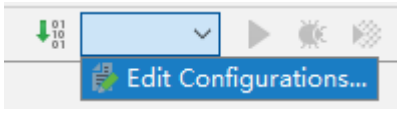
3. 完成工程创建后，目录结构如下。

图 2-47 目录结构



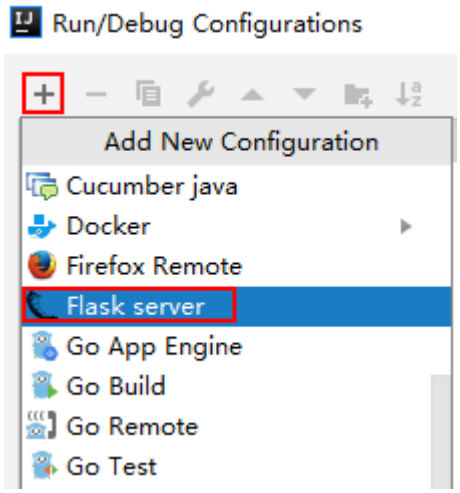
4. 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 2-48 Edit Configurations

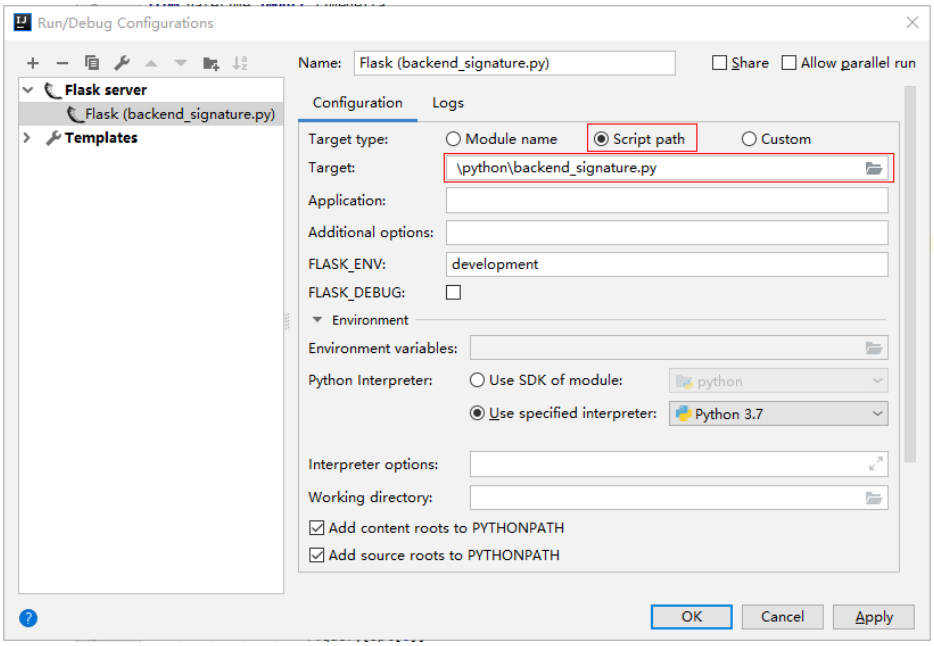


5. 单击“+”，选择“Flask server”。

图 2-49 选择 Flask server



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。



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。

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!"
```

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):

        # Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
        # In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK1, CLOUD_SDK_SK1, and CLOUD_SDK_AK2, CLOUD_SDK_SK2.
        secrets = {
            os.getenv('CLOUD_SDK_AK1'): os.getenv('CLOUD_SDK_SK1'),
            os.getenv('CLOUD_SDK_AK2'): os.getenv('CLOUD_SDK_SK2'),
        }
        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
```

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
```

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]
```

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()
```

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
```

7. 调用verify方法校验请求签名。判断校验是否通过。

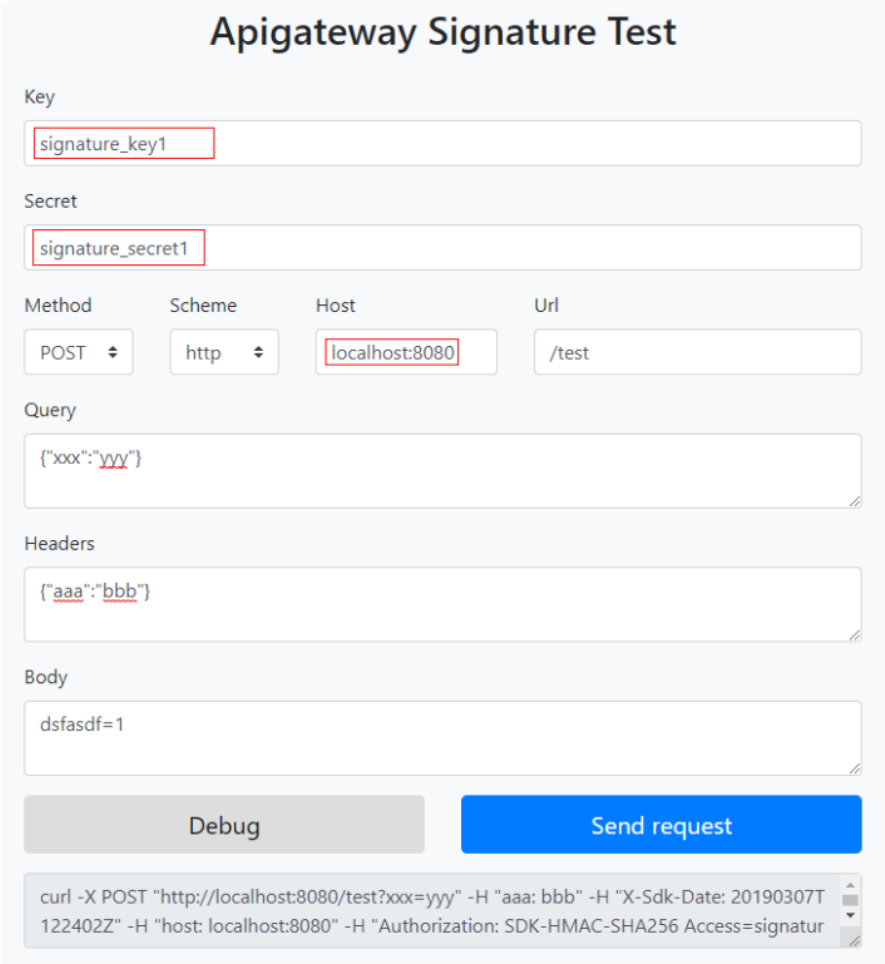
```
sig = signer.Signer()
sig.Key = signingKey
sig.Secret = signingSecret
```

```
if not sig.Verify(r, m.group(3)):
    return 'Verify authroization failed.', 401
```

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。



2.6.4 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。

前提条件

- 已在控制台创建签名密钥，并绑定API，具体请参见《ROMA Connect用户指南》的“配置后端服务的签名校验”章节。

- 已获取签名密钥的Key和Secret，具体请参见**开发准备**。
- 已安装C#开发语言环境，具体请参见**开发准备**。

获取 SDK

登录ROMA Connect实例控制台，在“服务集成 APIC > 凭据管理”的“SDKs”页签中下载SDK。

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。

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)
        {
```

```
        }
    }
}
```

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>
        {
        // Directly writing AK/SK in code is risky. For security, encrypt your AK/SK and store them in the configuration file or environment variables.
        // In this example, the AK/SK are stored in environment variables for identity authentication. Before running this example, set environment variables CLOUD_SDK_AK1, CLOUD_SDK_SK1, and CLOUD_SDK_AK2, CLOUD_SDK_SK2.
            {Environment.GetEnvironmentVariable("CLOUD_SDK_AK1"), Environment.GetEnvironmentVariable("CLOUD_SDK_SK1")},
            {Environment.GetEnvironmentVariable("CLOUD_SDK_AK2"), Environment.GetEnvironmentVariable("CLOUD_SDK_SK2")},
        };

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

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
```

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];
```

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;
}
```

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;
}
```

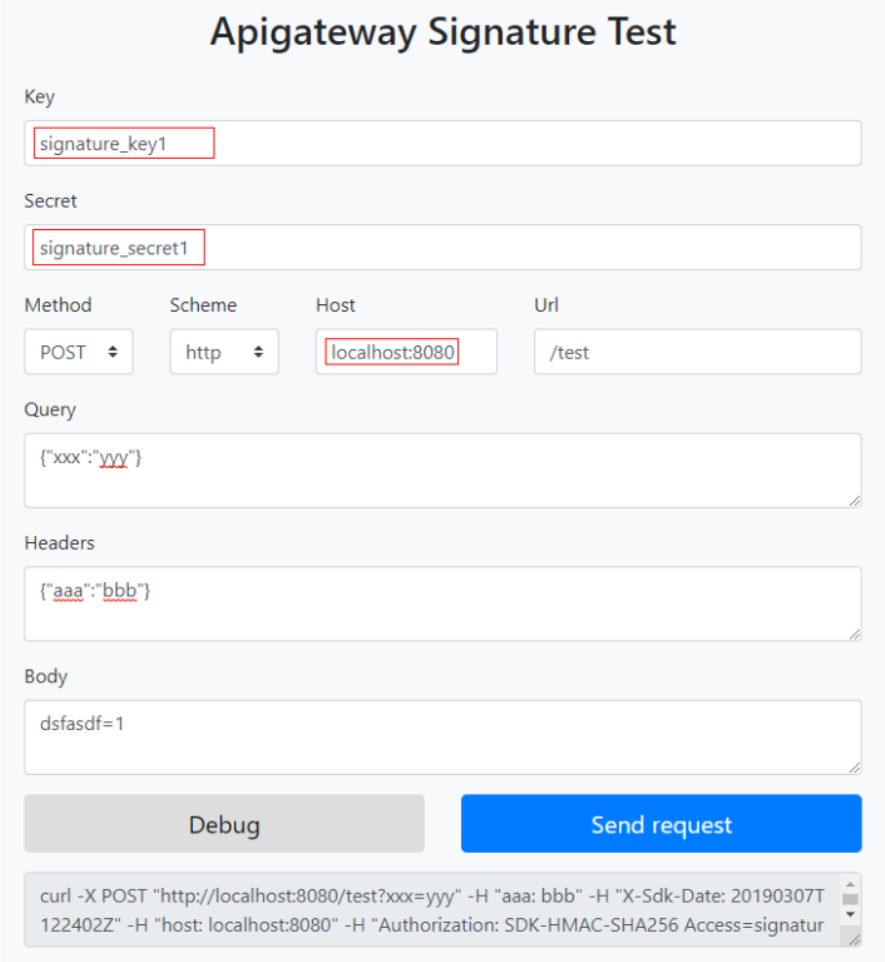
7. 调用verify方法校验请求签名。判断校验是否通过。

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

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。



3 消息集成开发指导（Kafka 引擎）

3.1 开发说明

3.1.1 开发场景

场景说明

ROMA Connect的消息集成MQS完全兼容开源Kafka协议，业务应用需要开发并集成**开源Kafka客户端**或ROMA Connect提供的RESTful API，实现与MQS的消息连接。

- 集成开源客户端：业务应用直接集成开源Kafka客户端，并通过客户端连接MQS，实现消息的生产和消费。
- 集成RESTful API：业务应用通过调用RESTful API，连接MQS并实现消息的生产和消费。

3.1.2 开发规范

- **开发工具版本要求**：
  - IntelliJ IDEA：2018.3.5或以上版本。
  - Eclipse：3.6.0或以上版本。
  - Visual Studio：2019 version 16.8.4或以上版本。
- **开发语言环境版本要求**：
  - Java：1.8.111或以上版本的Java Development Kit。
  - Python：2.7或3.X版本的Python。
  - Go：1.14及以上版本的Go。
  - C#：6.0及以上版本的.NET。

- **客户端版本使用建议：**

ROMA Connect的Kafka版本	Kafka客户端建议版本
1.1.0	• Java：1.1.0及以上版本 • Python：2.0.1及以上版本 • Go：1.8.2及以上版本 • C#：1.5.2及以上版本
2.3.0	• Java：2.3.0及以上版本 • Python：2.0.1及以上版本 • Go：1.8.2及以上版本 • C#：1.5.2及以上版本
2.7	• Java：2.7.2及以上版本 • Python：2.0.1及以上版本 • Go：1.8.2及以上版本 • C#：1.5.2及以上版本

- **客户端开发和配置建议：**
  请参见**客户端使用建议**和**客户端参数配置建议**。

3.1.3 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=-1
- 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。
- 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。
- 建议在Kafka客户端侧配置日志转储，否则容易引起磁盘被日志打满的情况。
- 一个消费组下的消费者在连接同一个MQS时，最多允许该消费组下的500个消费者进行连接，超过数量的消费者将连接失败。如果一个消费组下有超过500个消费者，且需要连接同一个MQS，需要把消费者拆分到多个消费组下。

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

3.1.4 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 3-1 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，只要任何一个备份存活，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。
max.request.size	1048576	5242880	生产者给服务端单次发送的最大消息字节数，该参数影响单次生产的消息记录数量。

表 3-2 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。

3.2 开发准备

获取 MQS 连接信息

- 获取连接地址与端口

  在ROMA Connect实例控制台的“实例信息”页面，在“基本信息”页签查看MQS连接地址。

  - 使用Kafka客户端内网连接MQS：在“连接地址”下查看消息集成 MQS内网连接地址。
  - 使用Kafka客户端公网连接MQS：在“连接地址”下查看消息集成 MQS公网连接地址。
  - 使用RESTful API连接MQS：在“MQS基本信息”下查看消息RESTful API。

- 获取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所属集成应用的名称进入集成应用详情页面，在基本信息中可查看集成应用的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。

准备开发环境

- 安装开发工具

  请根据所使用语言选择适合的开发工具。

  - 安装2018.3.5或以上版本的IntelliJ IDEA，安装包请至**IntelliJ IDEA官方网站**下载。
  - 安装3.0.3及以上版本Apache Maven，安装包请至**Maven官方下载页面**下载。
  - 安装2019 version 16.8.4及以上版本的Visual Studio，安装包请至**Visual Studio官方网站**下载。

- 安装开发语言环境

  - Java：安装1.8.111或以上版本的Java Development Kit，安装包请至**Oracle官方下载页面**下载。
  - Python：安装2.7或3.X版本的Python，安装包请至**Python官方下载页面**下载。
  - Go：安装1.14及以上版本的Go，安装包请至**Go官方下载页面**下载。
  - C#：安装6.0及以上版本的.NET，安装包请至**.NET官方下载页面**下载。

3.3 MQS 连接开发（开源客户端）

3.3.1 Java 客户端使用说明

操作场景

本文介绍Java版本的Kafka客户端连接指导，并完成客户端连接以及消息生产与消费的相关示例。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

引入 Kafka 客户端

MQS基于Kafka社区版本1.1.0、2.3.0、2.7，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。Java开源客户端的版本使用请参见**客户端版本使用建议**。

根据实例的Kafka版本信息使用对应版本的客户端，此处以2.7.2版本客户端为例进行说明。

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.7.2</version>
</dependency>
```

修改配置信息

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。

  ```
  #Topic名称在具体的生产与消费代码中。
  ########################
  #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。
  ########################
  #设置用户名和密码
  ```

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    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。配置此密码是为了访问Java生成的jks文件。
ssl.truststore.password=dms@kafka
ssl.endpoint.identification.algorithm=
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap.servers：MQS连接地址和端口。
- username和password：开启SASL_SSL认证时所使用的用户名和密码。
- ssl.truststore.location：开启SASL_SSL认证时所使用的客户端证书。

- 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。

```
#Topic名称在具体的生产与消费代码中。
########################
#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账号和密码，通过控制台设置
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \
    username="username" \
    password="********";
#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，配置此密码是为了访问Java生成的jks文件。
ssl.truststore.password=dms@kafka
ssl.endpoint.identification.algorithm=
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap.servers：MQS连接地址和端口。
- group.id：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- username和password：开启SASL_SSL认证时所使用的用户名和密码。
- ssl.truststore.location：开启SASL_SSL认证时所使用的客户端证书。

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("topicName", 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();
        // 注意填写要消费消息的Topic名称。
        consumer.consume(Arrays.asList("topicName"));
        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;
    }
}
```

3.3.2 Python 客户端使用说明

操作场景

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。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。

引入 Kafka 客户端

MQS基于Kafka社区版本1.1.0、2.3.0、2.7，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。Python开源客户端的版本使用请参见**客户端版本使用建议**。

执行以下命令，安装对应版本的Python Kafka客户端：

```
pip install kafka-python==2.0.1
```

生产消息

- 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
  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')
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap_servers：MQS连接地址和端口。
- topic_name：要生产消息的Topic名称。
- sasl_plain_username和sasl_plain_password：开启SASL_SSL认证时所使用的用户名和密码。
- context.load_verify_locations：开启SASL_SSL认证时所使用的客户端证书。

- 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')
```

  示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

  - bootstrap_servers：MQS连接地址和端口。
  - topic_name：要生产消息的Topic名称。

消费消息

- 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
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')
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap_servers：MQS连接地址和端口。
- topic_name：要消费消息的Topic名称。
- sasl_plain_username和sasl_plain_password：开启SASL_SSL认证时所使用的用户名和密码。
- consumer_id：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- context.load_verify_locations：开启SASL_SSL认证时所使用的客户端证书。

- 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')
```

  示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

  - bootstrap_servers：MQS连接地址和端口。
  - topic_name：要消费消息的Topic名称。
  - consumer_id：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。

3.3.3 Go 客户端使用说明

操作场景

本文以Linux CentOS环境为例，介绍Go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。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。

引入 Kafka 客户端

MQS基于Kafka社区版本1.1.0、2.3.0、2.7，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。Go开源客户端的版本使用请参见**客户端版本使用建议**。

执行以下命令，安装对应版本的Go Kafka客户端。

```
go get github.com/confluentinc/confluent-kafka-go/kafka
```

生产消息

- SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
package main

import (
    "bufio"
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    topics   = "topic_name"
    user     = "username"
    password = "password"
    caFile   = "phy_ca.crt"
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "security.protocol": "SASL_SSL",
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }
```

```
    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
    return data
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- brokers：MQS连接地址和端口。
- topics：要生产消息的Topic名称。
- user和password：开启SASL_SSL认证时所使用的用户名和密码。
- caFile：开启SASL_SSL认证时所使用的客户端证书。

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
package main

import (
    "bufio"
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)
```

```
var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka producer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
    }
    producer, err := kafka.NewProducer(config)
    if err != nil {
        log.Panicf("producer error, err: %v", err)
        return
    }

    go func() {
        for e := range producer.Events() {
            switch ev := e.(type) {
            case *kafka.Message:
                if ev.TopicPartition.Error != nil {
                    log.Printf("Delivery failed: %v\n", ev.TopicPartition)
                } else {
                    log.Printf("Delivered message to %v\n", ev.TopicPartition)
                }
            }
        }
    }()

    // Produce messages to topic (asynchronously)
    fmt.Println("please enter message:")
    go func() {
        for {
            err := producer.Produce(&kafka.Message{
                TopicPartition: kafka.TopicPartition{Topic: &topics, Partition: kafka.PartitionAny},
                Value:          GetInput(),
            }, nil)
            if err != nil {
                log.Panicf("send message fail, err: %v", err)
                return
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    // Wait for message deliveries before shutting down
    producer.Flush(15 * 1000)
    producer.Close()
}

func GetInput() []byte {
    reader := bufio.NewReader(os.Stdin)
    data, _, _ := reader.ReadLine()
    return data
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- brokers：MQS连接地址和端口。
- topics：要生产消息的Topic名称。

消费消息

- SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group_id"
    topics   = "topic_name"
    user     = "username"
    password = "password"
    caFile   = "phy_ca.crt"
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
        "security.protocol": "SASL_SSL",
        "sasl.mechanism":    "PLAIN",
        "sasl.username":     user,
        "sasl.password":     password,
        "ssl.ca.location":   caFile,
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
```

```
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- brokers：MQS连接地址和端口。
- group：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- topics：要消费消息的Topic名称。
- user和password：开启SASL_SSL认证时所使用的用户名和密码。
- caFile：开启SASL_SSL认证时所使用的客户端证书。

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
package main

import (
    "fmt"
    "github.com/confluentinc/confluent-kafka-go/kafka"
    "log"
    "os"
    "os/signal"
    "syscall"
)

var (
    brokers  = "ip1:port1,ip2:port2,ip3:port3"
    group    = "group_id"
    topics   = "topic_name"
)

func main() {
    log.Println("Starting a new kafka consumer")

    config := &kafka.ConfigMap{
        "bootstrap.servers": brokers,
        "group.id":          group,
        "auto.offset.reset": "earliest",
    }

    consumer, err := kafka.NewConsumer(config)
    if err != nil {
        log.Panicf("Error creating consumer: %v", err)
        return
    }

    err = consumer.SubscribeTopics([]string{topics}, nil)
    if err != nil {
        log.Panicf("Error subscribe consumer: %v", err)
        return
    }

    go func() {
        for {
            msg, err := consumer.ReadMessage(-1)
            if err != nil {
                log.Printf("Consumer error: %v (%v)", err, msg)
            } else {
                fmt.Printf("Message on %s: %s\n", msg.TopicPartition, string(msg.Value))
            }
        }
    }()

    sigterm := make(chan os.Signal, 1)
    signal.Notify(sigterm, syscall.SIGINT, syscall.SIGTERM)
```

```
    select {
    case <-sigterm:
        log.Println("terminating: via signal")
    }
    if err = consumer.Close(); err != nil {
        log.Panicf("Error closing consumer: %v", err)
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- brokers：MQS连接地址和端口。
- group：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- topics：要消费消息的Topic名称。

3.3.4 C#客户端使用说明

操作场景

本文介绍C#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。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和C#开发语言环境，具体请参见**开发准备**。

引入 Kafka 客户端

MQS基于Kafka社区版本1.1.0、2.3.0、2.7，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。C#开源客户端的版本使用请参见**客户端版本使用建议**。

执行以下命令下载安装C#的Kafka依赖库。

```
dotnet add package -v 1.5.2 Confluent.Kafka
```

生产消息

- SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
using System;
using Confluent.Kafka;

class Producer
{
    public static void Main(string[] args)
    {
        var conf = new ProducerConfig {
            bootstrap_servers = “ip1:port1,ip2:port2,ip3:port3”,
            context.load_verify_locations = “phy_ca.crt“,
            sasl_mechanism = “PLAIN”,
            security_protocol= “SASL_SSL”,
            SaslUsername = “username”,
            SaslPassword = “password”,
            };

        Action<DeliveryReport<Null, string>> handler = r =>
            Console.WriteLine(!r.Error.IsError
```

```
            ? $”Delivered message to {r.TopicPartitionOffset}”
            : $”Delivery Error: {r.Error.Reason}”);

        string topic = “topic_name”;

        using (var p = new ProducerBuilder<Null, string>(conf).Build())
        {
            for (int i=0; i<100; ++i)
            {
                p.Produce(topic, new Message<Null, string> { Value = i.ToString() }, handler);
            }
            p.Flush(TimeSpan.FromSeconds(10));
        }
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap_servers：MQS连接地址和端口。
- topic：要生产消息的Topic名称。
- SaslUsername和SaslPassword：开启SASL_SSL认证时所使用的用户名和密码。
- context.load_verify_locations：开启SASL_SSL认证时所使用的客户端证书。

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
using System;
using Confluent.Kafka;

class Producer
{
    public static void Main(string[] args)
    {
        var conf = new ProducerConfig {
            bootstrap_servers = “ip1:port1,ip2:port2,ip3:port3”,
            };

        Action<DeliveryReport<Null, string>> handler = r =>
            Console.WriteLine(!r.Error.IsError
                ? $”Delivered message to {r.TopicPartitionOffset}”
                : $”Delivery Error: {r.Error.Reason}”);

        string topic = “topic_name”;

        using (var p = new ProducerBuilder<Null, string>(conf).Build())
        {
            for (int i=0; i<100; ++i)
            {
                p.Produce(topic, new Message<Null, string> { Value = i.ToString() }, handler);
            }
            p.Flush(TimeSpan.FromSeconds(10));
        }
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- bootstrap_servers：MQS连接地址和端口。
- topic：要生产消息的Topic名称。

消费消息

- SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
using System;
using System.Threading;
```

```
using Confluent.Kafka;

class Consumer
{
    public static void Main(string[] args)
    {
        var conf = new ConsumerConfig {
            GroupId = “group_id”,
            BootstrapServers = “ip1:port1,ip2:port2,ip3:port3”,
            SslCaLocation = “phy_ca.crt”,
            SaslMechanism = “PLAIN”,
            SecurityProtocol = SASL_SSL,
            SaslUsername = “username”,
            SaslPassword = “password”,
            AutoOffsetReset = “earliest”
        };

        string topic = “topic_name”;

        using (var c = new ConsumerBuilder<Ignore, string>(conf).Build())
        {
            c.Subscribe(topic);

            CancellationTokenSource cts = new CancellationTokenSource();
            Console.CancelKeyPress += (_, e) => {
                e.Cancel = true;
                cts.Cancel();
            };

            try
            {
                while (true)
                {
                    try
                    {
                        var cr = c.Consume(cts.Token);
                        Console.WriteLine($"Consumed message '{cr.Value}' at: '{cr.TopicPartitionOffset}'.");
                    }
                    catch (ConsumeException e)
                    {
                        Console.WriteLine($"Error occured: {e.Error.Reason}");
                    }
                }
            }
            catch (OperationCanceledException)
            {
                c.Close();
            }
        }
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- BootstrapServers：MQS连接地址和端口。
- GroupId：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- topic：要消费消息的Topic名称。
- SaslUsername和SaslPassword：开启SASL_SSL认证时所使用的用户名和密码。
- SslCaLocation：开启SASL_SSL认证时所使用的客户端证书。

- 非SASL认证方式

  注意，加粗内容需要替换为实例自有信息。

```
using System;
using System.Threading;
```

```
using Confluent.Kafka;

class Consumer
{
    public static void Main(string[] args)
    {
        var conf = new ConsumerConfig {
            GroupId = “group_id”,
            BootstrapServers = “ip1:port1,ip2:port2,ip3:port3”,
            AutoOffsetReset = “earliest”
        };

        string topic = “topic_name”;

        using (var c = new ConsumerBuilder<Ignore, string>(conf).Build())
        {
            c.Subscribe(topic);

            CancellationTokenSource cts = new CancellationTokenSource();
            Console.CancelKeyPress += (_, e) => {
                e.Cancel = true;
                cts.Cancel();
            };

            try
            {
                while (true)
                {
                    try
                    {
                        var cr = c.Consume(cts.Token);
                        Console.WriteLine($"Consumed message '{cr.Value}' at: '{cr.TopicPartitionOffset}'.");
                    }
                    catch (ConsumeException e)
                    {
                        Console.WriteLine($"Error occured: {e.Error.Reason}");
                    }
                }
            }
            catch (OperationCanceledException)
            {
                c.Close();
            }
        }
    }
}
```

示例代码中的参数说明，可参考**获取MQS连接信息**获取参数值。

- BootstrapServers：MQS连接地址和端口。
- GroupId：消费组名称。根据业务需要，自定义消费组名称，如果设置的消费组不存在，系统会自动创建。
- topic：要消费消息的Topic名称。

3.3.5 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。

3.3.6 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。

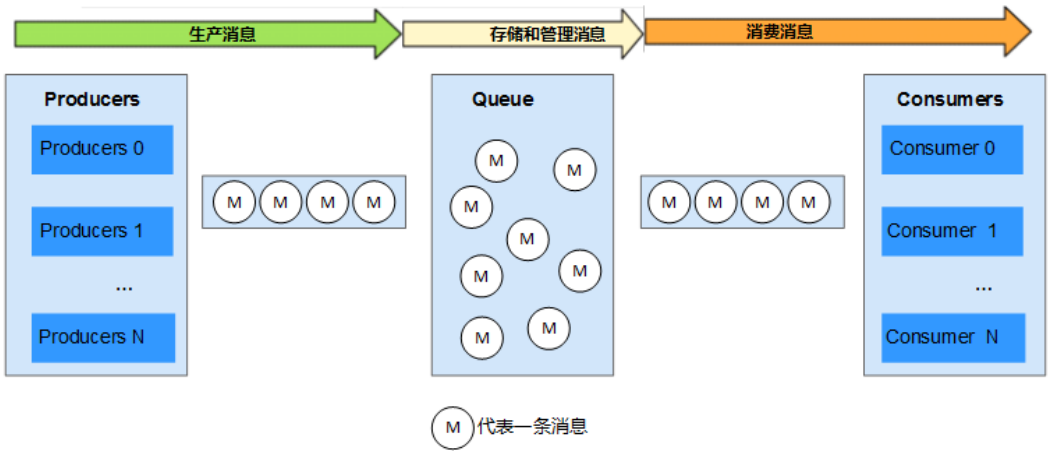
- 当无法确认消息是否已发送成功，生产者需要将消息重复发送给ROMA Connect。
- 当重复收到已处理过的消息，消费者需要告诉ROMA Connect消费成功且保证不重复处理。

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 3-1 消息批量生产与消费

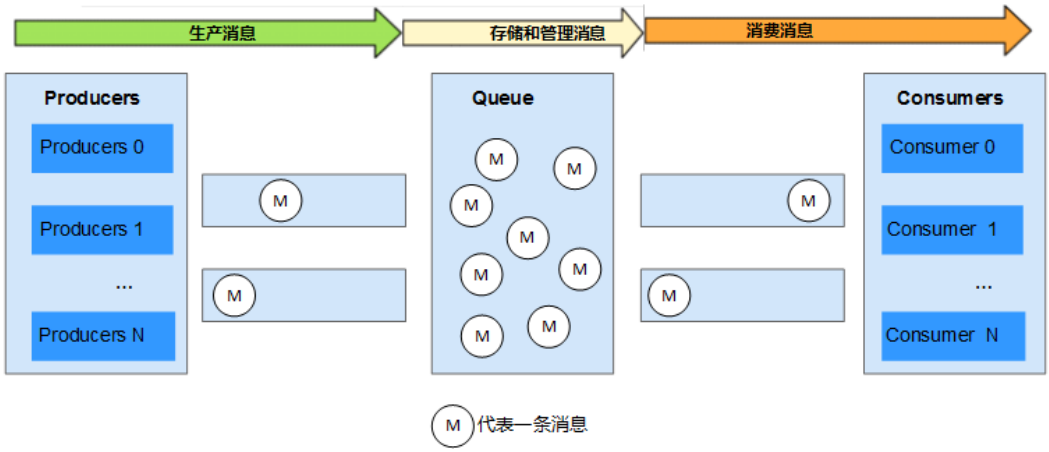


须知

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 3-2 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。

3.3.7 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、2.3.0、2.7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。

3.4 MQS 连接开发（RESTful API）

3.4.1 Java Demo 使用说明

操作场景

除了前面章节介绍的使用原生Kafka客户端，MQS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。

操作流程

1. 获取MQS连接信息，具体请参考**开发准备**。

   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**。

前提条件

本指南提供了Java语言的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）：

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。
- 获取Demo。

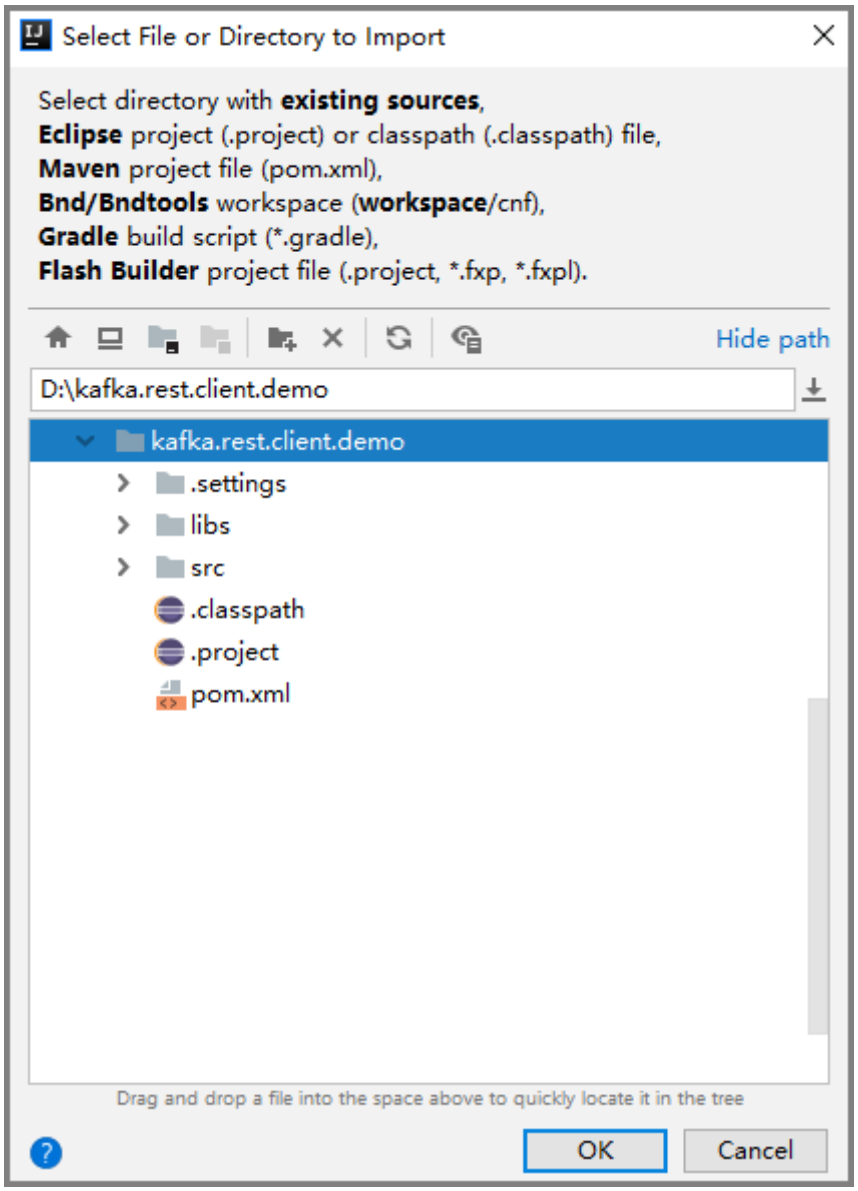
  在ROMA Connect实例控制台的“消息集成 MQS > Topic管理”页面，单击右上角的“下载示例 > 下载RESTful API Java Demo包”下载Demo。

导入工程

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

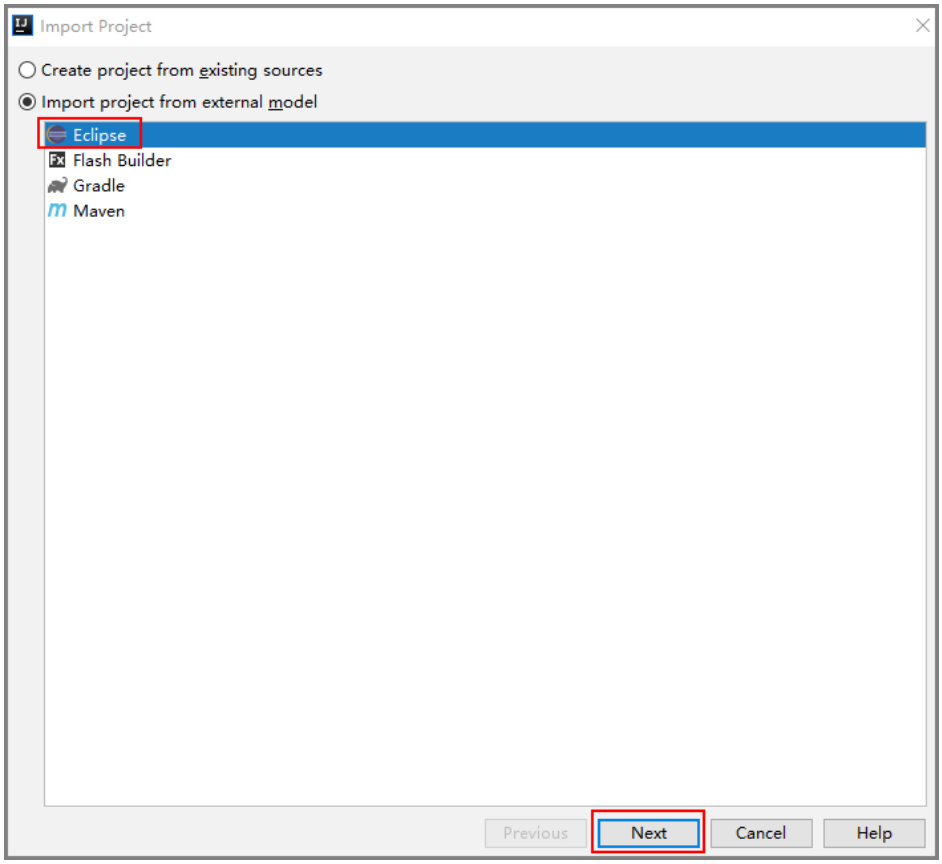
   弹出“Select File or Directory to Import”对话框。

2. 在弹出的对话框中选择解压后的RESTful API Java Demo路径，单击“OK”。



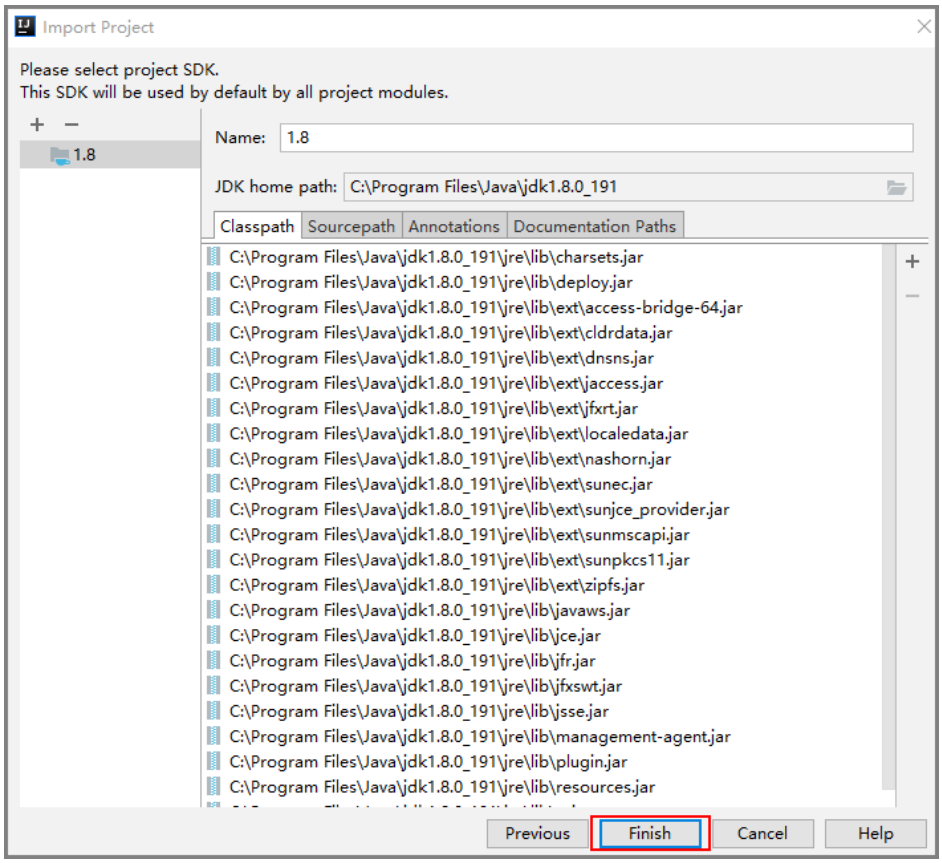
3. “Import project from external model”选择“Eclipse”，单击“Next”，进入下一页后保持默认连续单击“Next”，直到“Please select project SDK”页面。

图 3-3 Import Project



4. 单击“Finish”，完成工程导入。

图 3-4 Finish



5. 编辑rest-config.properties

   文件在src/main/resources目录下。将获取到的MQS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
   ```

6. 编辑log4j.properties

   修改日志存储目录：

   ```
   log.directory=D://workspace/logs
   ```

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。

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;
}
```

3.4.2 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 3-3 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 3-4 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"
  }
]
```

3.4.3 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 3-5 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	Object	消息的内容。

表 3-6 message 参数

参数	类型	说明
content	String	消息体的内容。Base64加密密文。

响应示例

```
[
  {
    "handler": "NCMxMDAjMTgjMA==",
    "message": {
      "content": "ImhlbGxvIGh1YXdlaWNsb3VkLTIi"
    }
  }
]
```

3.4.4 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 3-7 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。

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 3-8 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
}
```

4 消息集成开发指导（RocketMQ 引擎）

4.1 开发说明

4.1.1 开发场景

场景说明

ROMA Connect的消息集成MQS完全兼容开源RocketMQ协议，业务应用需要开发并集成开源RocketMQ客户端，实现与MQS的消息连接。

4.1.2 开发规范

- **开发工具版本要求**：
  - IntelliJ IDEA：2018.3.5或以上版本。
  - Eclipse：3.6.0或以上版本。
- **开发语言环境版本要求**：
  - Java：1.8.111或以上版本的Java Development Kit。
  - Python：2.7或3.X版本的Python安装包。
  - Go：1.14及以上版本的Go安装包。
- **客户端版本使用建议：**

ROMA Connect的RocketMQ版本	RocketMQ客户端建议版本
4.8.0	• Java：支持4.2.0及以上版本，推荐使用4.9.0及以上版本 • Python：支持1.2.0及以上版本，推荐使用1.2.4及以上版本 • Go：支持2.0.0及以上版本，推荐使用2.1.0及以上版本

4.2 开发准备

获取 MQS 连接信息

- 获取连接地址与端口

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

  - 消息集成 MQS元数据连接地址：使用RocketMQ客户端连接MQS，且客户端与MQS内网互通时使用。
  - 消息集成 MQS元数据公网地址：使用RocketMQ客户端连接MQS，且客户端与MQS公网互通时使用

- 获取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。

- ACL认证信息

  若ROMA Connect实例启用了ACL访问控制，则需要获取用户名和密码。

  在ROMA Connect实例控制台的“集成应用”页面，单击Topic所属集成应用的名称进入集成应用详情页面，在基本信息中可查看集成应用的Key和Secret，即为用户名和密码。

准备开发环境

- 安装开发工具
  - 安装2018.3.5或以上版本的IntelliJ IDEA，安装包请至**IntelliJ IDEA官方网站**下载。
  - 安装3.0.3及以上版本Apache Maven，安装包请至**Maven官方下载页面**下载。
- 安装开发语言环境
  - Java：安装1.8.111或以上版本的Java Development Kit，安装包请至**Oracle官方下载页面**下载。
  - Python：安装2.7或3.X版本的Python安装包，安装包请至**Python官方下载页面**下载。
  - Go：安装1.14及以上版本的Go安装包，安装包请至**Go官方下载页面**下载。

4.3 Java 客户端使用说明

4.3.1 收发普通消息

操作场景

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.0**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。
```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```
- 下载**依赖JAR包**。

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码，或者通过**Producer.java**获取更多示例代码。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。

```
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.common.RemotingHelper;

public class Main {
    public static void main(String[] args) {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");  //修改为实际的生产组名称

        producer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
        //producer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
        try {
```

```
        producer.start();
        Message msg = new Message("TopicName",  //修改为实际的Topic名称
            "TagName",
            "OrderID",
            "Message".getBytes(RemotingHelper.DEFAULT_CHARSET));
        SendResult sendResult = producer.send(msg);
        System.out.printf("%s%n", sendResult);

    } catch (Exception e) {
        e.printStackTrace();

    }
    producer.shutdown();
  }
}
```

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

使用异步发送需要客户端实现异步发送回调接口（SendCallback）。即消息发送方在发送了一条消息后，不需要等待服务端响应接着发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。

参考如下示例代码，或者通过**AsyncProducer.java**获取更多示例代码。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。

```
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendCallback;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.common.RemotingHelper;


public class Main {
    public static void main(String[] args) throws InterruptedException {
        DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");  //修改为实际的生产组名称

        producer.setNamesrvAddr("192.168.120.45:8200;192.168.123.150:8200");  //修改为实际的元数据连接地址
        //producer.setUseTLS(true);   //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
        try {
            producer.start();
            Message msg = new Message("TopicName",  //修改为实际的Topic名称
                    "TagName",
                    "OrderID",
                    "Message".getBytes(RemotingHelper.DEFAULT_CHARSET));
            producer.send(msg, new SendCallback() {
                @Override
                public void onSuccess(SendResult result) {
                    // 消息发送成功。
                    System.out.println("send message success. msgId= " + result.getMsgId());
                }


                @Override
                public void onException(Throwable throwable) {
                    // 消息发送失败，需要进行重试处理，可重新发送这条消息或持久化这条数据进行补偿处理。
                    System.out.println("send message failed.");
                    throwable.printStackTrace();
                }
            });


        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
        Thread.sleep(2000);
        producer.shutdown();
    }}
```

订阅普通消息

参考如下示例代码，或者通过**PushConsumer.java**获取更多示例代码。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。

```
import java.util.List;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("ConsumerGroupName");  //修改为实际的消费组名称

        consumer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
        //consumer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
        consumer.subscribe("TopicName", "*");  //修改为实际的Topic名称
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

4.3.2 收发顺序消息

操作场景

顺序消息是RocketMQ提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。

- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.0**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- 下载**依赖JAR包**。

发送顺序消息

参考如下示例代码，或者通过**Producer.java**获取更多示例代码。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。

```
import java.nio.charset.StandardCharsets;
import java.util.List;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;
import org.apache.rocketmq.remoting.exception.RemotingException;

public class Producer {

    public static void main(String[] args) {
        try {
            DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");  //修改为实际的生产组名称

            producer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
            //producer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
            producer.start();

            String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD", "TagE"};
            for (int i = 0; i < 100; i++) {
                String orderId = "order" + (i % 10);
                Message msg = new Message("TopicName", tags[i % tags.length], "KEY" + i,  //修改为实际的Topic名称
                    ("Message " + i).getBytes(StandardCharsets.UTF_8));
                SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
                    @Override
                    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {
                        String orderId = (String) arg;
                        int index = Math.abs(orderId.hashCode() % mqs.size());
                        return mqs.get(index);
                    }
                }, orderId);

                System.out.printf("%s%n", sendResult);
            }

            producer.shutdown();
        } catch (MQClientException | RemotingException | MQBrokerException | InterruptedException e) {
            e.printStackTrace();
```

```
    }
  }}
```

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以在分区选择算法中以“orderId/队列个数的余数”作为消息发送的队列。

订阅顺序消息

参考如下示例代码，或者通过**Consumer.java**获取更多示例代码。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。

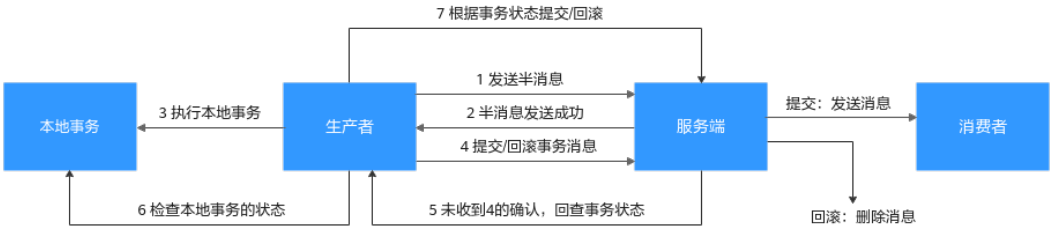
```
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;

import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeOrderlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerOrderly;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.message.MessageExt;

public class Consumer {

    public static void main(String[] args) throws MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("ConsumerGroupName");  //修改为实际的消费组名称

        consumer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
        //consumer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。

        consumer.subscribe("TopicName", "*");  //修改为实际的Topic名称

        consumer.registerMessageListener(new MessageListenerOrderly() {
            AtomicLong consumeTimes = new AtomicLong(0);

            @Override
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {
                context.setAutoCommit(true);
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(), msgs);
                this.consumeTimes.incrementAndGet();
                if ((this.consumeTimes.get() % 3) == 0) {
                    context.setSuspendCurrentQueueTimeMillis(3000);
                    return ConsumeOrderlyStatus.SUSPEND_CURRENT_QUEUE_A_MOMENT;
                }

                return ConsumeOrderlyStatus.SUCCESS;
            }
        });

        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

4.3.3 收发事务消息

操作场景

RocketMQ的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如**图4-1**所示。

图 4-1 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.0**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- 下载**依赖JAR包**。

发送事务消息

参考如下示例代码，或者通过**TransactionProducer.java**获取更多示例代码。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。

```
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.LocalTransactionState;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.client.producer.TransactionListener;
import org.apache.rocketmq.client.producer.TransactionMQProducer;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageExt;
import org.apache.rocketmq.remoting.common.RemotingHelper;

import java.io.UnsupportedEncodingException;

public class Main {
    public static void main(String[] args) throws MQClientException, UnsupportedEncodingException {
        TransactionListener transactionListener = new TransactionListener() {
            @Override
            public LocalTransactionState executeLocalTransaction(Message message, Object o) {
                System.out.println("开始执行本地事务：  " + message);
                return LocalTransactionState.COMMIT_MESSAGE;
            }
```

```
        @Override
        public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {
            System.out.println("收到回查，重新查询事务状态： " + messageExt);
            return LocalTransactionState.COMMIT_MESSAGE;
        }
    };

    TransactionMQProducer producer = new TransactionMQProducer("ProducerGroupName");  //修改为实际的生产组名称

    producer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
    //producer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
    producer.setTransactionListener(transactionListener);
    producer.start();

    Message msg =
        new Message("TopicName", "TagName", "KeyName",  //修改为实际的Topic名称
            "Message ".getBytes(RemotingHelper.DEFAULT_CHARSET));
    SendResult sendResult = producer.sendMessageInTransaction(msg, null);
    System.out.printf("%s%n", sendResult);

    producer.shutdown();
  }}
```

事务消息生产者需要实现两个回调函数，其中executeLocalTransaction回调函数在发送完半事务消息后被调用，即上图中的第3阶段，checkLocalTransaction回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- LocalTransactionState.COMMIT_MESSAGE：提交事务，允许消费者消费该消息。
- LocalTransactionState.ROLLBACK_MESSAGE：回滚事务，消息将被丢弃不允许消费。
- LocalTransactionState.UNKNOW：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息**相同。

4.3.4 发送定时消息

操作场景

RocketMQ支持**任意时间**的定时消息，最大推迟时间可达到1年。同时也支持定时消息的取消。

定时消息即生产者生产消息到RocketMQ后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在秒级以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在秒级的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距在秒级以内，它们投递的顺序与它们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受2天的消息保存时长限制。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

开源的Java客户端支持连接分布式消息服务RocketMQ版，推荐使用的客户端版本为**4.9.0**。

通过以下任意一种方式引入依赖：

- 使用Maven方式引入依赖。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.9.0</version>
</dependency>
```

- 下载**依赖JAR包**。

发送定时消息

发送定时消息的示例代码如下。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。

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;
```

```
public class ScheduledMessageProducer1 {
    public static final String TOPIC_NAME = "ScheduledTopic";

    public static void main(String[] args) throws MQClientException, InterruptedException,
MQBrokerException, RemotingException {

    DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");  //修改为实际的生
产组名称

    producer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
    //producer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
    producer.start();

    // 定时消息投递时间戳，该消息10秒后投递
    final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
    // 创建消息对象
    Message msg = new Message(TopicName,  //修改为实际的Topic名称
        "TagName",
        "KeyName",
        "scheduled message".getBytes(StandardCharsets.UTF_8));
    // 设置消息定时投递的时间戳属性
    msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
    // 发送消息，该消息将会在10秒后投递
    SendResult sendResult = producer.send(msg);
    // 打印发送结果和预计投递时间
    System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));


    producer.shutdown();
  }
}
```

取消定时消息

取消定时消息的示例代码如下。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。

```
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.UtilAll;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;

public class ScheduledMessageProducer1 {
    public static final String TOPIC_NAME = "ScheduledTopic";

    public static void main(String[] args) throws MQClientException, InterruptedException,
MQBrokerException, RemotingException {

    DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");  //修改为实际的生
产组名称

    producer.setNamesrvAddr("192.168.0.1:8100");  //修改为实际的元数据连接地址
    //producer.setUseTLS(true);  //创建实例时，如果开启了数据加密传输SSL，请取消此行代码注释。
    producer.start();

    // 定时消息投递时间戳，该消息10秒后投递
    final long deliverTimestamp = Instant.now().plusSeconds(10).toEpochMilli();
    // 创建消息对象
    Message msg = new Message(TopicName,  //修改为实际的Topic名称
        "TagName",
        "KeyName",
        "scheduled message".getBytes(StandardCharsets.UTF_8));
```

```
// 设置消息定时投递的时间戳属性
msg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
// 发送消息，该消息将会在10秒后投递
SendResult sendResult = producer.send(msg);
// 打印发送结果和预计投递时间
System.out.printf("%s %s%n", sendResult, UtilAll.timeMillisToHumanString2(deliverTimestamp));

// ====== 发送取消消息逻辑 ======

// 创建取消消息对象
Message cancelMsg = new Message(TopicName,  //修改为实际的Topic名称
    "",
    "",
    "cancel".getBytes(StandardCharsets.UTF_8));
// 设置取消消息的时间戳，该时间戳必须与要取消的定时消息的定时时间戳一致
cancelMsg.putUserProperty("__STARTDELIVERTIME", String.valueOf(deliverTimestamp));
// 设置要取消消息的ID，为发送消息的唯一ID（UNIQUE_KEY），可以从发送消息的结果中获取
cancelMsg.putUserProperty("__CANCEL_SCHEDULED_MSG", sendResult.getMsgId());
// 发送取消消息，必须在定时消息被投递之前发送才可以取消
SendResult cancelSendResult = producer.send(cancelMsg, sendResult.getMessageQueue());
System.out.printf("cancel %s%n", cancelSendResult);

producer.shutdown();    }}
```

4.3.5 使用 ACL 权限访问

操作场景

若ROMA Connect实例开启了MQS的ACL访问控制，客户端向Topic生产和消费消息时，需要使用授权集成应用的Key和Secret进行安全认证。

前提条件

已获取Topic所授权集成应用的Key和Secret，具体请参见**开发准备**。

生产者增加认证信息

生产者初始化时需要增加“rpcHook”参数。

```
RPCHook rpcHook = new AclClientRPCHook(new SessionCredentials(ACL_ACCESS_KEY, ACL_SECRET_KEY));
DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName", rpcHook);
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

消费者增加认证信息

消费者初始化时需要增加“rpcHook”参数。

```
RPCHook rpcHook = new AclClientRPCHook(new SessionCredentials(ACL_ACCESS_KEY, ACL_SECRET_KEY));
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(null, "ConsumerGroupName",
rpcHook);
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

4.3.6 消费端限流

在RocketMQ中，消费者消费消息时，可能会出现消费过快导致下游业务来不及处理的情况，进而影响系统的稳定性。本章节介绍在消费端进行限流的示例代码，以保障系统的稳定。

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。

```
package org.apache.rocketmq.example.simple;
```

```
import java.util.List;
import java.util.concurrent.TimeUnit;

import com.google.common.util.concurrent.RateLimiter;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.common.consumer.ConsumeFromWhere;
import org.apache.rocketmq.common.message.MessageExt;

public class PushConsumer {

    public static void main(String[] args) throws InterruptedException, MQClientException {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("ConsumerGroupName");  //修改为实际的消费组名称
        consumer.subscribe("TopicName", "*");  //修改为实际的Topic名称
        consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        RateLimiter rateLimiter = RateLimiter.create(200);
        consumer.registerMessageListener(new MessageListenerConcurrently() {

            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,
ConsumeConcurrentlyContext context) {
                if (!rateLimiter.tryAcquire(msgs.size(),3, TimeUnit.SECONDS)) {
                    return ConsumeConcurrentlyStatus.RECONSUME_LATER;
                }
                System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(),
msgs);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        consumer.start();
        System.out.printf("Consumer Started.%n");
    }
}
```

4.4 Go 客户端使用说明

4.4.1 收发普通消息

操作场景

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

在“go.mod”中增加以下代码，添加依赖。

```
module rocketmq-example-go
go 1.13
require (
    github.com/apache/rocketmq-client-go/v2 v2.1.0
)
```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
    "os"
)

// implements a simple producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    msg := &primitive.Message{
        Topic: "TopicName",  //修改为实际的Topic名称
        Body:  []byte("Message"),
    }
    msg.WithTag("TagName")
    msg.WithKeys([]string{"KeyName"})
    res, err := p.SendSync(context.Background(), msg)

    if err != nil {
        fmt.Printf("send message error: %s\n", err)
    } else {
        fmt.Printf("send message success: result=%s\n", res.String())
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

异步发送

异步发送是指消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息的通讯方式。

使用异步发送需要客户端实现异步发送回调接口（SendCallback）。即消息发送方在发送了一条消息后，不需要等待服务端响应接着发送第二条消息。发送方通过回调接口接收服务端响应，并处理响应结果。

参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
    "os"
    "sync"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

// implements a async producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
        producer.WithRetry(2))

    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    var wg sync.WaitGroup
    wg.Add(1)

    callback := func(ctx context.Context, result *primitive.SendResult, e error) {
        if e != nil {
            fmt.Printf("receive message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", result.String())
        }
        wg.Done()
    }
    message := primitive.NewMessage("TopicName", []byte("Message"))   //修改为实际的Topic名称
    err = p.SendAsync(context.Background(), callback, message)
    if err != nil {
        fmt.Printf("send message error: %s\n", err)
        wg.Done()
    }

    wg.Wait()
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

订阅普通消息

参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
    "os"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/consumer"
```

```
    "github.com/apache/rocketmq-client-go/v2/primitive"
)

func main() {
    c, _ := rocketmq.NewPushConsumer(
        consumer.WithGroupName("ConsumerGroupName"),  //修改为实际的消费组名称
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
    )
    err := c.Subscribe("TopicName", consumer.MessageSelector{}, func(ctx context.Context,  //修改为实际的Topic名称
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        for i := range msgs {
            fmt.Printf("subscribe callback: %v \n", msgs[i])
        }

        return consumer.ConsumeSuccess, nil
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("shutdown Consumer error: %s", err.Error())
    }
}
```

4.4.2 收发顺序消息

操作场景

顺序消息是RocketMQ提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

在“go.mod”中增加以下代码，添加依赖。

```
module rocketmq-example-go
go 1.13
require (
    github.com/apache/rocketmq-client-go/v2 v2.1.0
)
```

发送顺序消息

参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

// Package main implements a simple producer to send message.
func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    topic := "TopicName"   //修改为实际的Topic名称

    for i := 0; i < 100; i++ {
        msg := &primitive.Message{
            Topic: topic,
            Body:  []byte("Message " + strconv.Itoa(i)),
        }
        orderId := strconv.Itoa(i % 10)
        msg.WithShardingKey(orderId)
        res, err := p.SendSync(context.Background(), msg)

        if err != nil {
            fmt.Printf("send message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", res.String())
        }
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以将orderId作为选择队列的sharding key。

订阅顺序消息

只需要在订阅普通消息的代码基础上增加consumer.WithConsumerOrder(true)，参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
```

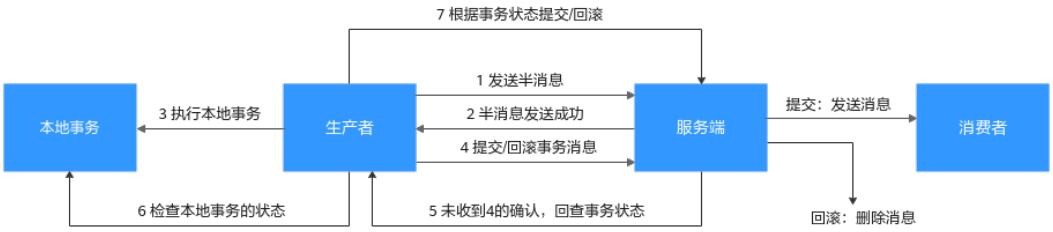
```
    "os"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/consumer"
    "github.com/apache/rocketmq-client-go/v2/primitive"
)

func main() {
    c, _ := rocketmq.NewPushConsumer(
        consumer.WithGroupName("ConsumerGroupName"),      //修改为实际的消费组名称
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),      //修改为实际的元数据连接地址
        consumer.WithConsumerModel(consumer.Clustering),
        consumer.WithConsumeFromWhere(consumer.ConsumeFromFirstOffset),
        consumer.WithConsumerOrder(true),
    )
    err := c.Subscribe("TopicName", consumer.MessageSelector{}, func(ctx context.Context,   //修改为实际的Topic名称
        msgs ...*primitive.MessageExt) (consumer.ConsumeResult, error) {
        orderlyCtx, _ := primitive.GetOrderlyCtx(ctx)
        fmt.Printf("orderly context: %v\n", orderlyCtx)
        fmt.Printf("subscribe orderly callback: %v \n", msgs)
        return consumer.ConsumeSuccess, nil
    })
    if err != nil {
        fmt.Println(err.Error())
    }
    // Note: start after subscribe
    err = c.Start()
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(-1)
    }
    time.Sleep(time.Hour)
    err = c.Shutdown()
    if err != nil {
        fmt.Printf("Shutdown Consumer error: %s", err.Error())
    }
}
```

4.4.3 收发事务消息

操作场景

RocketMQ的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如图4-2所示。

图 4-2 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

在“go.mod”中增加以下代码，添加依赖。

```
module rocketmq-example-go
go 1.13
require (
    github.com/apache/rocketmq-client-go/v2 v2.1.0
)
```

发送事务消息

参考如下示例代码。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。

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "sync"
    "sync/atomic"
    "time"

    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
)

type DemoListener struct {
    localTrans       *sync.Map
    transactionIndex int32
}

func NewDemoListener() *DemoListener {
    return &DemoListener{
        localTrans: new(sync.Map),
    }
}

func (dl *DemoListener) ExecuteLocalTransaction(msg *primitive.Message) primitive.LocalTransactionState {
    nextIndex := atomic.AddInt32(&dl.transactionIndex, 1)
    fmt.Printf("nextIndex: %v for transactionID: %v\n", nextIndex, msg.TransactionId)
    status := nextIndex % 3
    dl.localTrans.Store(msg.TransactionId, primitive.LocalTransactionState(status+1))

    fmt.Printf("dl")
    return primitive.UnknowState
}

func (dl *DemoListener) CheckLocalTransaction(msg *primitive.MessageExt) primitive.LocalTransactionState
{
    fmt.Printf("%v msg transactionID : %v\n", time.Now(), msg.TransactionId)
    v, existed := dl.localTrans.Load(msg.TransactionId)
    if !existed {
        fmt.Printf("unknow msg: %v, return Commit", msg)
        return primitive.CommitMessageState
    }
    state := v.(primitive.LocalTransactionState)
    switch state {
```

```
    case 1:
        fmt.Printf("checkLocalTransaction COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    case 2:
        fmt.Printf("checkLocalTransaction ROLLBACK_MESSAGE: %v\n", msg)
        return primitive.RollbackMessageState
    case 3:
        fmt.Printf("checkLocalTransaction unknow: %v\n", msg)
        return primitive.UnknowState
    default:
        fmt.Printf("checkLocalTransaction default COMMIT_MESSAGE: %v\n", msg)
        return primitive.CommitMessageState
    }
}

func main() {
    p, _ := rocketmq.NewTransactionProducer(
        NewDemoListener(),
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
        producer.WithRetry(1),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s\n", err.Error())
        os.Exit(1)
    }

    for i := 0; i < 10; i++ {
        res, err := p.SendMessageInTransaction(context.Background(),
            primitive.NewMessage("TopicName", []byte("Message "+strconv.Itoa(i))))  //修改为实际的Topic名称

        if err != nil {
            fmt.Printf("send message error: %s\n", err)
        } else {
            fmt.Printf("send message success: result=%s\n", res.String())
        }
    }
    time.Sleep(5 * time.Minute)
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

事务消息生产者需要实现两个回调函数，其中ExecuteLocalTransaction回调函数在发送完半事务消息后被调用，即上图中的第3阶段，CheckLocalTransaction回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- primitive.CommitMessageState：提交事务，允许消费者消费该消息。
- primitive.RollbackMessageState：回滚事务，消息将被丢弃不允许消费。
- primitive.UnknowState：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息**相同。

4.4.4 发送定时消息

操作场景

RocketMQ支持**任意时间**的定时消息，最大推迟时间可达到1年。同时也支持定时消息的取消。

定时消息即生产者生产消息到RocketMQ后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在秒级以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在秒级的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距在秒级以内，它们投递的顺序与它们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受2天的消息保存时长限制。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

在“go.mod”中增加以下代码，添加依赖。

```
module rocketmq-example-go
go 1.13
require (
    github.com/apache/rocketmq-client-go/v2 v2.1.0
)
```

发送定时消息

发送定时消息的示例代码如下。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。

```
package main

import (
    "context"
    "fmt"
    "github.com/apache/rocketmq-client-go/v2"
    "github.com/apache/rocketmq-client-go/v2/primitive"
    "github.com/apache/rocketmq-client-go/v2/producer"
    "os"
)

func main() {
    p, _ := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),  //修改为实际的元数据连接地址
        producer.WithRetry(2),
    )
    err := p.Start()
    if err != nil {
        fmt.Printf("start producer error: %s", err.Error())
        os.Exit(1)
    }
    msg := primitive.NewMessage("TopicName", []byte("Message"))  //修改为实际的Topic名称
    msg.WithProperty("__STARTDELIVERTIME", strconv.FormatInt(time.Now().UnixMilli()+3000, 10))
    res, err := p.SendSync(context.Background(), msg)

    if err != nil {
        fmt.Printf("send message error: %s\n", err)
    } else {
        fmt.Printf("send message success: result=%s\n", res.String())
    }
    err = p.Shutdown()
    if err != nil {
        fmt.Printf("shutdown producer error: %s", err.Error())
    }
}
```

4.4.5 使用 ACL 权限访问

操作场景

若ROMA Connect实例开启了MQS的ACL访问控制，客户端向Topic生产和消费消息时，需要使用授权集成应用的Key和Secret进行安全认证。

前提条件

已获取Topic所授权集成应用的Key和Secret，具体请参见**开发准备**。

生产者增加认证信息

生产消息的函数中增加如下加粗代码，示例如下。

```
func main() {
    p, err := rocketmq.NewProducer(
        producer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        producer.WithRetry(2),
        producer.WithCredentials(primitive.Credentials{
            ACL_ACCESS_KEY: "RocketMQ",
            ACL_SECRET_KEY: "xxxxxxxx",
```

```
        }),
    )
}
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

消费者增加认证信息

消费消息的函数中增加如下加粗代码，示例如下。

```
func main() {
    c, err := rocketmq.NewPushConsumer(
        consumer.WithGroupName("ConsumerGroupName"),
        consumer.WithNsResolver(primitive.NewPassthroughResolver([]string{"192.168.0.1:8100"})),
        consumer.WithCredentials(primitive.Credentials{
            ACL_ACCESS_KEY: "RocketMQ",
            ACL_SECRET_KEY: "xxxxxxxx",
        }),
    )
}
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

4.5 Python 客户端使用说明

4.5.1 收发普通消息

操作场景

本章节介绍普通消息的收发方法和示例代码。其中，普通消息发送方式分为同步发送和异步发送。

- 同步发送：消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息。
- 异步发送：消息发送方发出一条消息后，不等服务端返回响应，接着发送下一条消息。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

1. 安装librocketmq库和rocketmq-client-python，具体操作请参考**rocketmq-client-python**。
2. 将librocketmq.so添加到系统动态库搜索路径。
   a. 查找librocketmq.so的路径。
      ```
      find / -name librocketmq.so
      ```
   b. 将librocketmq.so添加到系统动态库搜索路径。
      ```
      ln -s /查找到的librocketmq.so路径/librocketmq.so /usr/lib
      sudo ldconfig
      ```

同步发送

同步发送是指消息发送方发出一条消息到服务端，服务端接收并处理消息，然后返回响应给发送方，发送方收到响应后才会发送下一条消息的通讯方式。

参考如下示例代码。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。

```
from rocketmq.client import Producer, Message

topic = 'TopicName'    //修改为实际的Topic名称
gid = 'ProducerGroupName'    //修改为实际的生产组名称
name_srv = '192.168.0.1:8100'    //修改为实际的元数据连接地址

def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg

def send_message_sync():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()

if __name__ == '__main__':
    send_message_sync()
```

订阅普通消息

参考如下示例代码。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。

```
import time

from rocketmq.client import PushConsumer, ConsumeStatus

def callback(msg):
    print(msg.id, msg.body, msg.get_property('property'))
    return ConsumeStatus.CONSUME_SUCCESS

def start_consume_message():
    consumer = PushConsumer('ConsumerGroupName')    //修改为实际的消费组名称
    consumer.set_name_server_address('192.168.0.1:8100')    //修改为实际的元数据连接地址
    consumer.subscribe('TopicName', callback)    //修改为实际的Topic名称
    print('start consume message')
    consumer.start()

    while True:
        time.sleep(3600)

if __name__ == '__main__':
    start_consume_message()
```

4.5.2 收发顺序消息

操作场景

顺序消息是RocketMQ提供的一种严格按照顺序来发布和消费的消息类型。

顺序消息分为全局顺序消息和分区顺序消息：

- 全局顺序消息：对于指定的一个Topic，将队列数量设置为1，这个队列内所有消息按照严格的先入先出FIFO（First In First Out）的顺序进行发布和订阅。
- 分区顺序消息：对于指定的一个Topic，同一个队列内的消息按照严格的FIFO顺序进行发布和订阅。生产者指定分区选择算法，保证需要按顺序消费的消息被分配到同一个队列。

全局顺序消息和分区顺序消息的区别仅为队列数量不同，代码没有区别。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Python开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

1. 安装librocketmq库和rocketmq-client-python，具体操作请参考**rocketmq-client-python**。
2. 将librocketmq.so添加到系统动态库搜索路径。
   a. 查找librocketmq.so的路径。
   ```
   find / -name librocketmq.so
   ```
   b. 将librocketmq.so添加到系统动态库搜索路径。
   ```
   ln -s /查找到的librocketmq.so路径/librocketmq.so /usr/lib
   sudo ldconfig
   ```

发送顺序消息

参考如下示例代码。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。

```
from rocketmq.client import Producer, Message

topic = 'TopicName'    //修改为实际的Topic名称
gid = 'ProducerGroupName'    //修改为实际的生产组名称
name_srv = '192.168.0.1:8100'    //修改为实际的元数据连接地址

def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg

def send_orderly_with_sharding_key():
    producer = Producer(gid, True)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    ret = producer.send_orderly_with_sharding_key(msg, 'orderId')
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()

if __name__ == '__main__':
    send_orderly_with_sharding_key()
```

上述代码中，相同orderId的消息需要保证顺序，不同orderId的消息不需要保证顺序，所以将orderId作为选择队列的sharding key。

订阅顺序消息

只需要在订阅普通消息的代码基础上增加orderly=True，参考如下示例代码。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。

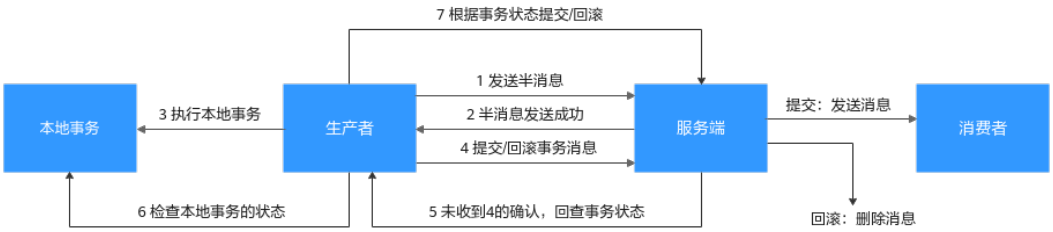
```
import time

from rocketmq.client import PushConsumer, ConsumeStatus

def callback(msg):
    print(msg.id, msg.body, msg.get_property('property'))
    return ConsumeStatus.CONSUME_SUCCESS

def start_consume_message():
    consumer = PushConsumer('ConsumerGroupName', orderly=True)  //修改为实际的消费组名称
    consumer.set_name_server_address('192.168.0.1:8100')  //修改为实际的元数据连接地址
    consumer.subscribe('TopicName', callback)  //修改为实际的Topic名称
    print('start consume message')
    consumer.start()

    while True:
        time.sleep(3600)

if __name__ == '__main__':
    start_consume_message()
```

4.5.3 收发事务消息

操作场景

RocketMQ的事务消息支持在业务逻辑与发送消息之间提供事务保证，通过两阶段的方式提供对事务消息的支持，事务消息交互流程如图4-3所示。

图 4-3 事务消息交互流程



事务消息生产者首先发送半消息，然后执行本地事务。如果执行成功，则发送事务提交，否则发送事务回滚。服务端在一段时间后如果一直收不到提交或回滚，则发起回查，生产者在收到回查后重新发送事务提交或回滚。消息只有在提交之后才投递给消费者，消费者对回滚的消息不可见。

前提条件

- 已获取MQS连接信息，具体请参见开发准备。
- 已安装开发工具和Python开发语言环境，具体请参见开发准备。

引入 RocketMQ 客户端

1. 安装librocketmq库和rocketmq-client-python，具体操作请参考rocketmq-client-python。

2. 将librocketmq.so添加到系统动态库搜索路径。
   a. 查找librocketmq.so的路径。
      ```
      find / -name librocketmq.so
      ```
   b. 将librocketmq.so添加到系统动态库搜索路径。
      ```
      ln -s /查找到的librocketmq.so路径/librocketmq.so /usr/lib
      sudo ldconfig
      ```

发送事务消息

参考如下示例代码。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。

```
import time

from rocketmq.client import Message, TransactionMQProducer, TransactionStatus

topic = 'TopicName'  //修改为实际的Topic名称
gid = 'ProducerGroupName'  //修改为实际的生产组名称
name_srv = '192.168.0.1:8100'  //修改为实际的元数据连接地址

def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg

def check_callback(msg):
    print('check: ' + msg.body.decode('utf-8'))
    return TransactionStatus.COMMIT

def local_execute(msg, user_args):
    print('local:   ' + msg.body.decode('utf-8'))
    return TransactionStatus.UNKNOWN

def send_transaction_message(count):
    producer = TransactionMQProducer(gid, check_callback)
    producer.set_name_server_address(name_srv)
    producer.start()
    for n in range(count):
        msg = create_message()
        ret = producer.send_message_in_transaction(msg, local_execute, None)
        print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    print('send transaction message done')

    while True:
        time.sleep(3600)

if __name__ == '__main__':
    send_transaction_message(10)
```

事务消息生产者需要实现两个回调函数，其中local_execute回调函数在发送完半事务消息后被调用，即上图中的第3阶段，check_callback回调函数在收到回查时调用，即上图中的第6阶段。两个回调函数均可返回3种事务状态：

- TransactionStatus.COMMIT：提交事务，允许消费者消费该消息。
- TransactionStatus.ROLLBACK：回滚事务，消息将被丢弃不允许消费。
- TransactionStatus.UNKNOWN：无法判断状态，期待服务端向生产者再次回查该消息的状态。

订阅事务消息

订阅事务消息的代码与**订阅普通消息**相同。

4.5.4 发送定时消息

操作场景

RocketMQ支持**任意时间**的定时消息，最大推迟时间可达到1年。同时也支持定时消息的取消。

定时消息即生产者生产消息到RocketMQ后，消息不会立即被消费，而是延迟到设定的时间点后才会发送给消费者进行消费。

定时消息适用于以下场景：

- 消息对应的业务逻辑有时间窗口要求，如电商交易中超时未支付关闭订单的场景。在订单创建时发送一条定时消息，5分钟以后投递给消费者，消费者收到此消息后需要判断对应订单是否完成支付，如果未完成支付，则关闭订单。如果已完成，则忽略。
- 通过消息触发定时任务的场景，如在某些固定时间点向用户发送提醒消息。

注意事项

- 定时消息的最大延迟时间为1年，延迟超过1年的消息将会发送失败。
- 定时消息的定时时间如果被设置成当前时间戳之前的某个时刻，消息将立刻投递给消费者。
- 在理想情况下，定时消息设定的时间与实际发送时间的误差在秒级以内。但在定时消息投递压力过大时，会触发定时消息投递流控机制，精度会变差。
- 在秒级的精度内，不保证消息投递的顺序性。即如果两条定时消息的定时时间差距在秒级以内，它们投递的顺序与它们发送的顺序无法确保一致。
- 无法确保定时消息仅投递一次，定时消息可能会重复投递。
- 定时消息的定时时间是服务端开始向消费端投递的时间。如果消费者当前有消息堆积，那么定时消息会排在堆积消息后面，将不能严格按照配置的时间进行投递。
- 由于客户端和服务端可能存在时间差，消息的实际投递时间与客户端设置的投递时间之间可能存在偏差，以服务端时间为准。
- 设置定时消息的投递时间后，依然受2天的消息保存时长限制。例如，设置定时消息5天后才能被消费，如果第5天后一直没被消费，那么这条消息将在第7天被删除。
- 定时消息将占用普通消息约3倍的存储空间，大量使用定时消息时需要注意存储空间占用。

前提条件

- 已获取MQS连接信息，具体请参见**开发准备**。
- 已安装开发工具和Go开发语言环境，具体请参见**开发准备**。

引入 RocketMQ 客户端

在“go.mod”中增加以下代码，添加依赖。

```
module rocketmq-example-go
go 1.13
require (
    github.com/apache/rocketmq-client-go/v2 v2.1.0
)
```

发送定时消息

发送定时消息的示例代码如下。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。

```
import time

from rocketmq.client import Producer, Message

topic = 'TopicName'    //修改为实际的Topic名称
gid = 'ProducerGroupName'    //修改为实际的生产组名称
name_srv = '192.168.0.1:8100'    //修改为实际的元数据连接地址

def create_message():
    msg = Message(topic)
    msg.set_keys('XXX')
    msg.set_tags('XXX')
    msg.set_property('property', 'test')
    msg.set_body('message body')
    return msg

def send_delay_message():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.start()
    msg = create_message()
    msg.set_property('__STARTDELIVERTIME', str(int(round((time.time() + 3) * 1000))))
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()

if __name__ == '__main__':
    send_delay_message()
```

4.5.5 使用 ACL 权限访问

操作场景

若ROMA Connect实例开启了MQS的ACL访问控制，客户端向Topic生产和消费消息时，需要使用授权集成应用的Key和Secret进行安全认证。

前提条件

已获取Topic所授权集成应用的Key和Secret，具体请参见**开发准备**。

生产者增加认证信息

生产消息的函数中增加如下加粗代码，示例如下。

```
def send_message_sync():
    producer = Producer(gid)
    producer.set_name_server_address(name_srv)
    producer.set_session_credentials("ACL_ACCESS_KEY", "ACL_SECRET_KEY", "")
    producer.start()
    msg = create_message()
    ret = producer.send_sync(msg)
    print('send message status: ' + str(ret.status) + ' msgId: ' + ret.msg_id)
    producer.shutdown()
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

消费者增加认证信息

消费消息的函数中增加如下加粗代码，示例如下。

```
def start_consume_message():
    consumer = PushConsumer('ConsumerGroupName')
    consumer.set_name_server_address('192.168.0.1:8100')
    consumer.set_session_credentials("ACL_ACCESS_KEY", "ACL_SECRET_KEY", "")
    consumer.subscribe('TopicName', callback)
    print('start consume message')
    consumer.start()
    while True:
        time.sleep(3600)
```

ACL_ACCESS_KEY和ACL_SECRET_KEY为Topic所授权集成应用的Key和Secret。

5 设备集成开发指导

5.1 开发说明

5.1.1 开发场景

场景说明

ROMA Connect的设备集成LINK支持设备通过MQTT协议接入并上报数据。设备需开发并集成MQTT客户端，并在集成过程中写入设备的接入信息，开发完成后设备上电并联网，即可接入ROMA Connect。

5.1.2 开发规范

- **开发工具版本要求**：
  - IntelliJ IDEA：2018.3.5或以上版本。
  - Eclipse：3.6.0或以上版本。
- **开发语言环境版本要求**：

  1.8.111或以上版本的Java Development Kit。
- **设备开发要求：**
  - 通过MQTT协议接入设备时，当前仅支持MQTT协议中的QoS0和QoS1。
  - 为防止因网络不稳定或实例升级等原因导致设备连接中断，建议在设备开发过程中增加自动重连机制。如果使用ROMA Connect提供的设备demo，则默认已开启重连机制。如果使用开源MQTT客户端，则需要根据开源代码配置重连机制。开启自动重连后，如果连接丢失，客户端会一直自动重连服务器，直到连接成功为止。

5.2 开发准备

获取设备的接入信息

在ROMA Connect实例控制台选择“设备集成 LINK > 设备管理”，在页面中可查看MQTT连接地址/MQTTS连接地址，在“设备”页签的设备列表中，可查看要接入设备的“客户端ID”、“用户名”和“密码”信息。

准备开发环境

- 安装开发工具

  安装2018.3.5或以上版本的IntelliJ IDEA，安装包请至**IntelliJ IDEA官方网站**下载。
- 安装开发语言环境

  安装1.8.111或以上版本的Java Development Kit，安装包请至**Oracle官方下载页面**下载。

5.3 设备集成开发

操作场景

本文提供了通过设备集成实现设备的接入、消息收发功能的操作指导，主要分为配置设备连接信息、收发消息两部分。

须知

设备集成支持标准的MQTT协议，您可以使用开源**Eclipse paho MQTT Client**与LINK进行对接，本例中Demo使用Java版本的SDK

前提条件

- 已获取设备的接入信息，具体请参见**开发准备**。
- 已安装开发工具和Java开发语言环境，具体请参见**开发准备**。
- 下载LINK Demo：

  在ROMA Connect实例控制台的“设备集成 LINK > 设备管理”页面，单击“下载Demo”，下载LINK Demo。

  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。

配置设备连接信息

1. 解压Demo压缩包，在“src”目录的底层路径下找到DeviceConnectDemo.java文件。
2. 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

Demo工程中依赖的软件包在lib目录下，使用Demo时需要将demo的lib目录设置为当前工程的lib目录。

```
// 设备连接地址，格式为：tcp://ip:port
final String host = "";
// 设备客户端ID
final String clientId = "";
// 示例仅用作测试或示意，设备认证的用户名为敏感信息，请勿硬编码使用
final String userName = "";
// 示例仅用作测试或示意，设备认证的密码为敏感信息，请勿硬编码使用
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LINK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)
     {
        // 示例仅用作测试或示意，API认证的appKey为敏感信息，请勿硬编码使用
        String appKey = "";
        // 示例仅用作测试或示意，API认证的appSecret为敏感信息，请勿硬编码使用
        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 LINK > 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LINK发送一条消息，代码运行后，LINK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”。

5.4 MQTT 协议 Topic 规范

5.4.1 使用前必读

- 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。

5.4.2 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。

5.4.3 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
}
```

5.4.4 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
}
```

5.4.5 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
}
```

5.4.6 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
}
```

5.4.7 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"
}
```

5.4.8 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"
}
```

5.4.9 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 5-1 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"
      }
   ]
}
```

5.4.10 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"
}
```

5.4.11 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"
}
```

5.4.12 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"
        }]
    }]
}
```