



API 网关

开发指南

发布日期 2024-10-09

目录

1 使用场景	1
2 使用 APP 认证调用 API	2
2.1 认证前准备	2
2.2 APP 认证工作原理	3
2.3 Java	7
2.4 Go	18
2.5 Python	21
2.6 C#	25
2.7 JavaScript	27
2.8 PHP	34
2.9 C++	38
2.10 C	40
2.11 Android	43
2.12 curl	45
3 使用 IAM 认证调用 API	47
3.1 Token 认证	47
3.2 AK/SK 认证	48
4 创建用于前端自定义认证的函数	50
5 创建用于后端自定义认证的函数	55
6 对后端服务进行签名	59
6.1 签名前准备	59
6.2 Java	59
6.3 Python	65
6.4 C#	71
7 修订记录	76

1 使用场景

API网关主要涉及以下几个开发场景：

- API调用认证开发：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。
 - **使用APP认证（签名认证）调用API**：API调用者通过API网关提供Key、Secret和SDK，对API请求进行签名认证。
 - **使用IAM认证（Token认证）调用API**：API调用者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。
 - **使用IAM认证（AK/SK认证）调用API**：API调用者通过AK（Access Key ID）/SK（Secret Access Key）和API网关提供的SDK，对API请求进行签名处理。
- 自定义认证的函数开发：使用自定义认证时，在函数服务中编写一个函数，将其作为认证服务。
 - **创建用于前端自定义认证的函数**：API网关利用函数对收到的API请求进行安全认证。
 - **创建用于后端自定义认证的函数**：API后端服务利用函数对来自API网关转发的API请求进行安全认证。
- **后端服务签名校验开发**：若API在API网关上绑定了签名密钥，则API网关向该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API网关提供的SDK，并对请求中的签名信息进行校验。

2 使用 APP 认证调用 API

2.1 认证前准备

APP认证方式调用API，需要提前获取如下信息：

- 获取API的请求信息
在APIG实例控制台选择“API管理 > API列表”，单击API名称进入API详情，在“API运行”页签的API名称下方查看API的域名、请求路径和请求方法。
- API已发布到环境
在APIG实例控制台选择“API管理 > API列表”，单击API名称进入API详情，在“API运行”页签的“前端配置 > 前端定义”中查看已发布的环境。
- 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APIG实例控制台选择“API管理 > 凭据管理”，进入凭据详情，获取Key和Secret。

📖 说明

- AppKey/Key: APP访问密钥ID。与私有访问密钥关联的唯一标识符；访问密钥ID和私有访问密钥一起使用，对请求进行加密签名。
- AppSecret/Secret: 与访问密钥ID结合使用的密钥，对请求进行加密签名，可标识发送方，并防止请求被修改。
- 发送API请求时，需要将当前时间置于请求消息头的X-Sdk-Date，将签名信息置于请求消息头的Authorization。

⚠️ 注意

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

API网关除了校验时间格式外，还会校验该时间值与网关收到请求的时间差，如果时间差大于15分钟，API网关将拒绝请求。

2.2 APP 认证工作原理

1. **构造规范请求。**
将待发送的请求内容按照与API网关（即API管理）后台约定的规则组装，确保客户端签名、API网关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. API网关收到请求后，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API网关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://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/app1?b=2&a=1 HTTP/1.1  
Host: c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com  
X-Sdk-Date: 20191111T093443Z
```

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。

API网关除了校验时间格式外，还会校验该时间值与网关收到请求的时间差，如果时间差大于15分钟，API网关将拒绝请求。

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。
- 消息头名称要保持唯一性，出现多个相同消息头名称时，无法完成认证。

举例：

```
GET
/app1/
a=1&b=2
host:c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com
x-sdk-date:20191111T093443Z
```

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: c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com\n
Content-Type: application/json;charset=utf8\n
My-header1: a b c \n
X-Sdk-Date:20191111T093443Z\n
My-Header2: "a b c" \n
```

规范消息头为：

```
content-type:application/json;charset=utf8\n
host:c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com\n
my-header1:a b c\n
my-header2:"a b c"\n
x-sdk-date:20191111T093443Z\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API网关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API网关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:c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com
```

```
x-sdk-date:20191111T093443Z
```

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

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

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:c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com
x-sdk-date:20191111T093443Z

host;x-sdk-date
e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
```

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：

```
af71c5a7ef45310b8dc05ab15f7da50189ffa81a95cc284379ebaa5eb61155c0
```

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
20191111T093443Z
af71c5a7ef45310b8dc05ab15f7da50189ffa81a95cc284379ebaa5eb61155c0
```


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FWTh5tqu2Pb9ZGt8NI09XYZti2V1LTa8useKXMD8，则计算得到的signature为：

```
01cc37e53d821da93bb7239c5b6e1640b184a748f8c20e61987b491e00b15822
```

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=FM9RLCN*****NAXISK, SignedHeaders=host;x-sdk-date,  
Signature=01cc37e53d821da93bb7239c5b6e1640b184a748f8c20e61987b491e00b15822
```

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API网关，由API网关完成身份认证。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。

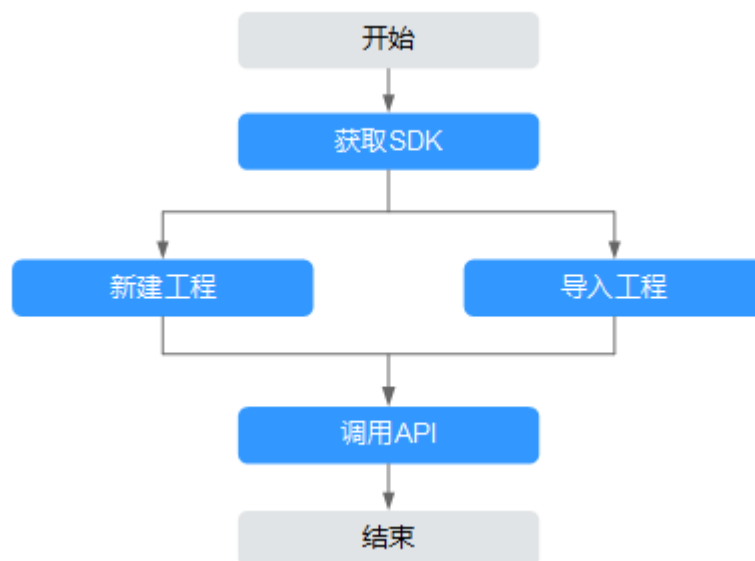
2.3 Java

操作场景

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

本章节以Eclipse 4.5.2版本为例介绍。

图 2-1 调用流程



前提条件

- 已获取API调用信息，具体参见[认证前准备](#)。
- 已安装Eclipse 3.6.0或以上版本，如果未安装，请至[Eclipse官方网站](#)下载。
- 已安装Java Development Kit 1.8.111或以上版本，如果未安装，请至[Oracle官方下载页面](#)下载。暂不支持Java Development Kit 17或以上版本。

获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-java-sdk.zip”压缩包，解压后目录结构如下：

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

导入工程

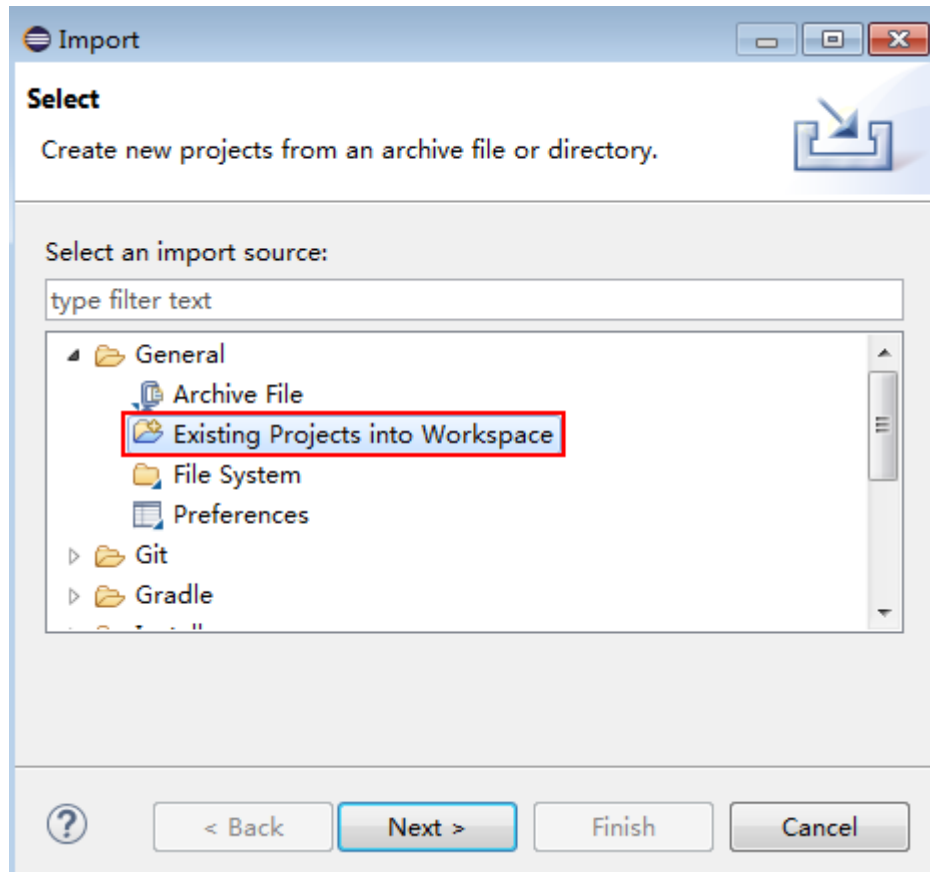
步骤1 打开Eclipse，在菜单栏选择“File > Import”。

弹出“Import”对话框。

步骤2 选择“General > Existing Projects into Workspace”，单击“Next”。

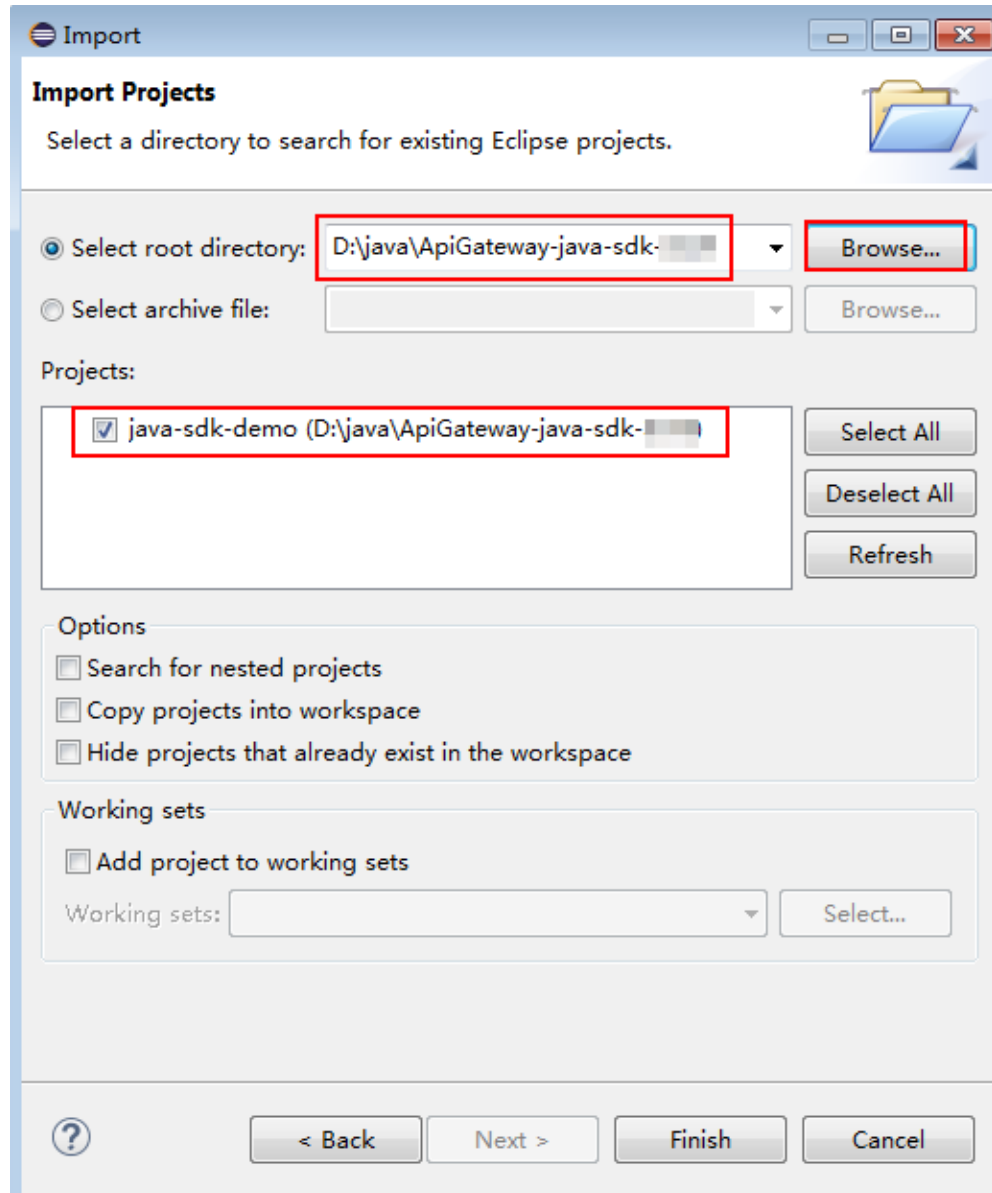
弹出“Import Projects”对话框。

图 2-2 Import



步骤3 单击“Browse”，在弹出的对话框中选择解压后的SDK路径。

图 2-3 选择 demo 工程



步骤4 单击“Finish”，完成工程导入。

“Main.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[调用API示例](#)。

----结束

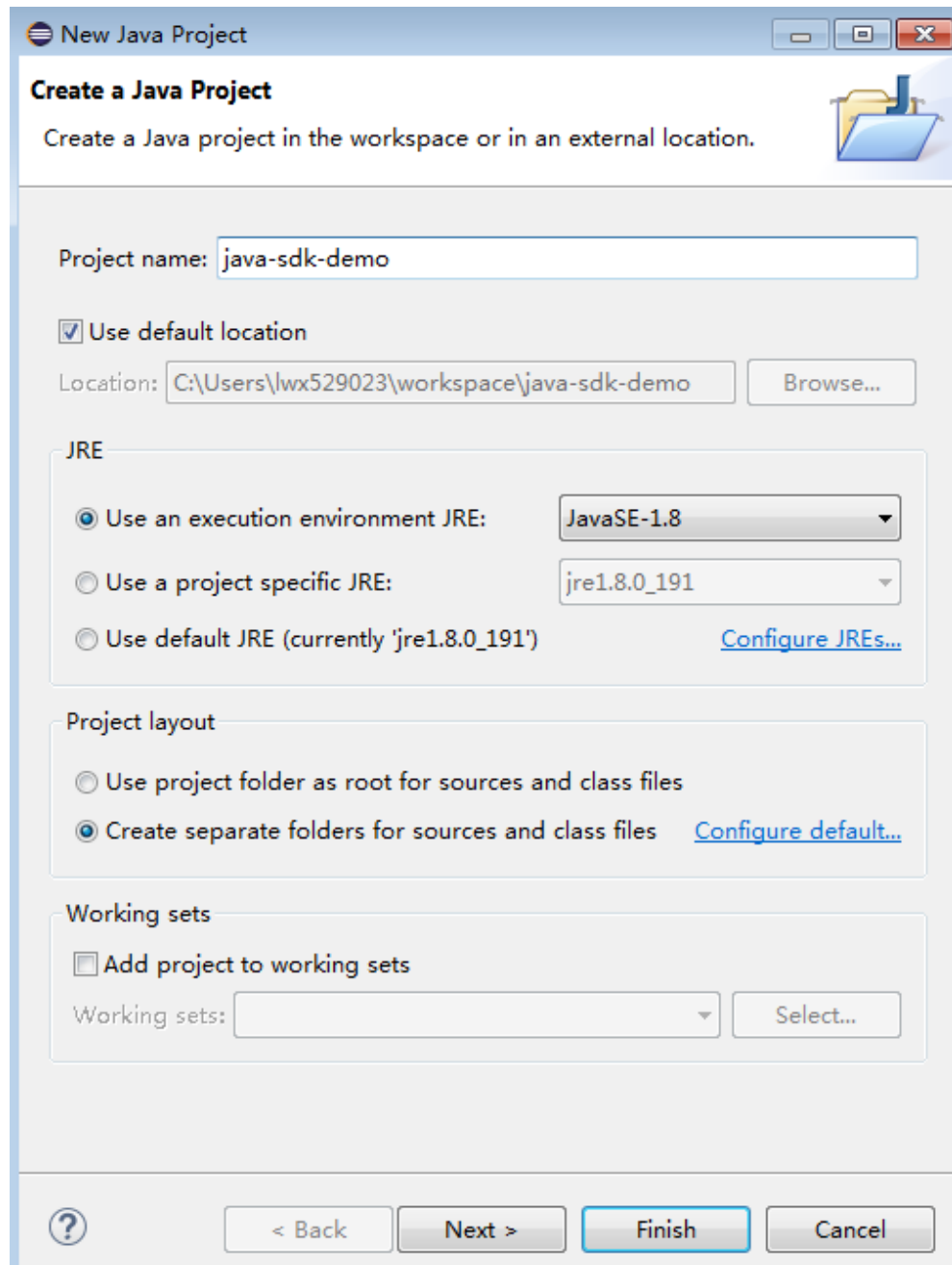
新建工程

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

弹出“New Java Project”对话框。

步骤2 自定义“Project name”，以“java-sdk-demo”为例，其他参数保持默认，单击“Finish”。

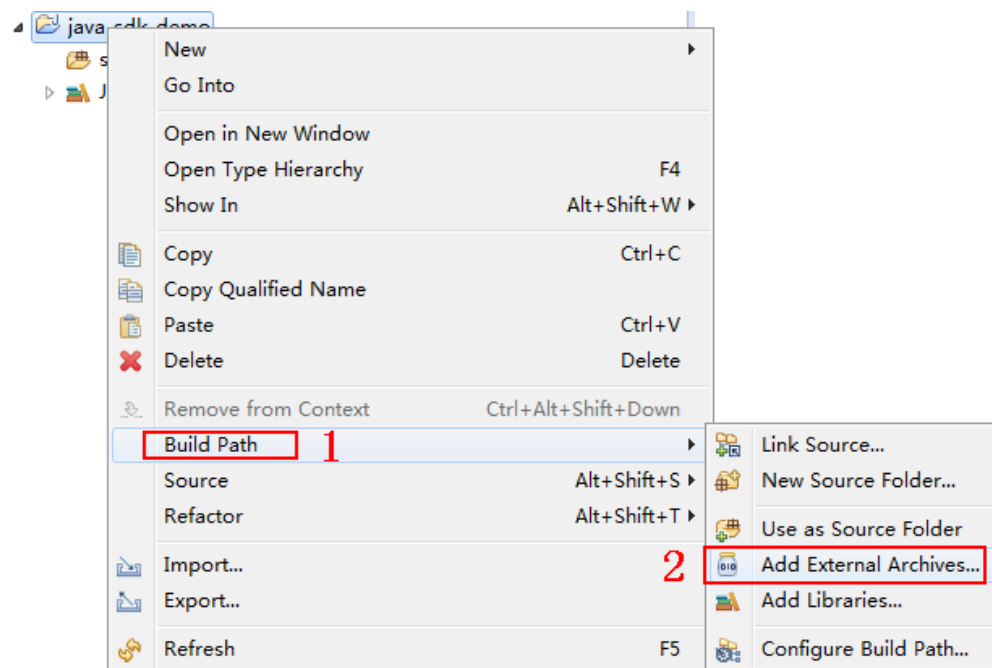
图 2-4 新建工程



步骤3 导入API Gateway Java SDK的“jar”文件。

1. 选择“java-sdk-demo”，单击鼠标右键，选择“Build Path > Add External Archives”。

图 2-5 导入 jar 文件



2. 选择SDK中“\libs”目录下所有以“jar”结尾的文件，单击“打开”。

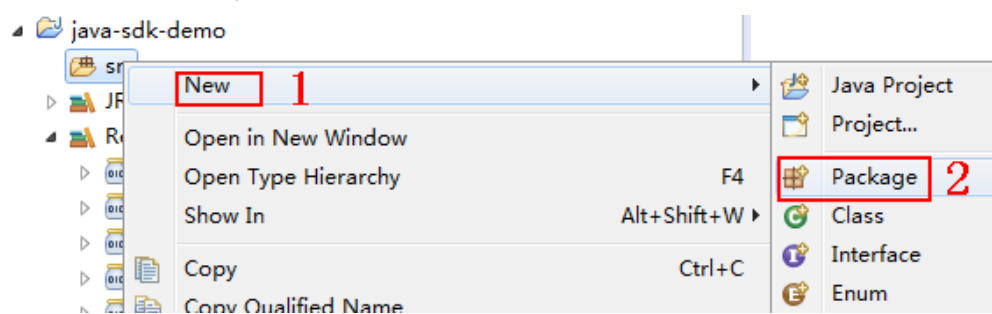
图 2-6 选择 jar 文件

commons-codec-1.11.jar	2019/3/15 10:23	Executable Jar File	328 KB
commons-logging-1.2.jar	2019/3/15 10:23	Executable Jar File	61 KB
httpclient-4.5.5.jar	2019/3/15 10:23	Executable Jar File	749 KB
httpcore-4.4.9.jar	2019/3/15 10:23	Executable Jar File	318 KB
java-sdk-core-3.0.9.jar	2019/3/15 10:23	Executable Jar File	101 KB
joda-time-2.9.9.jar	2019/3/15 10:23	Executable Jar File	620 KB
okhttp-3.11.0.jar	2019/3/15 10:23	Executable Jar File	404 KB
okio-1.15.0.jar	2019/3/15 10:23	Executable Jar File	87 KB

步骤4 新建“Package”及“Main”文件。

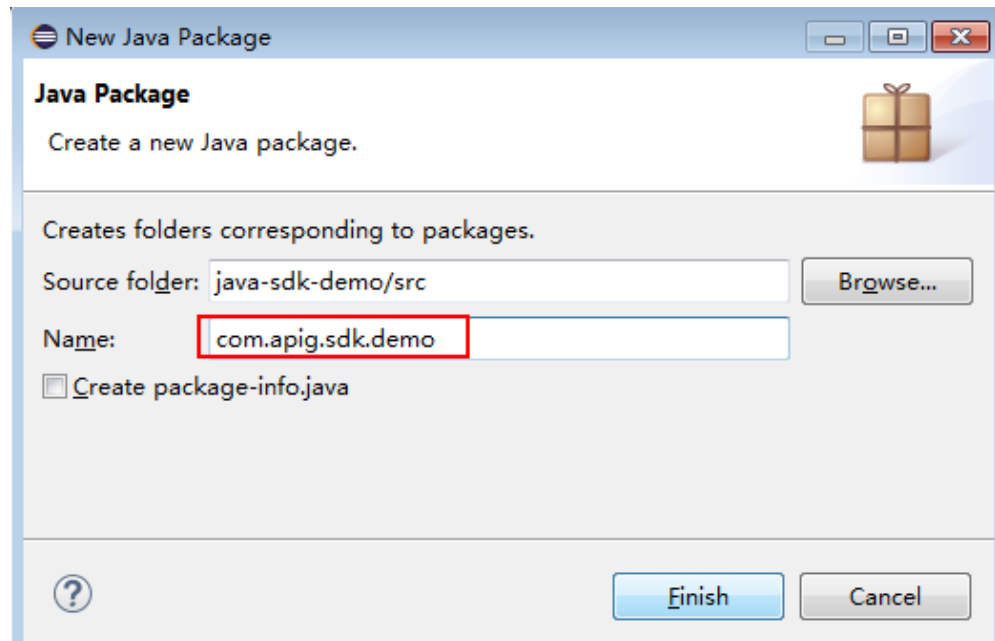
1. 选择“src”，单击鼠标右键，选择“New > Package”。

图 2-7 新建 Package



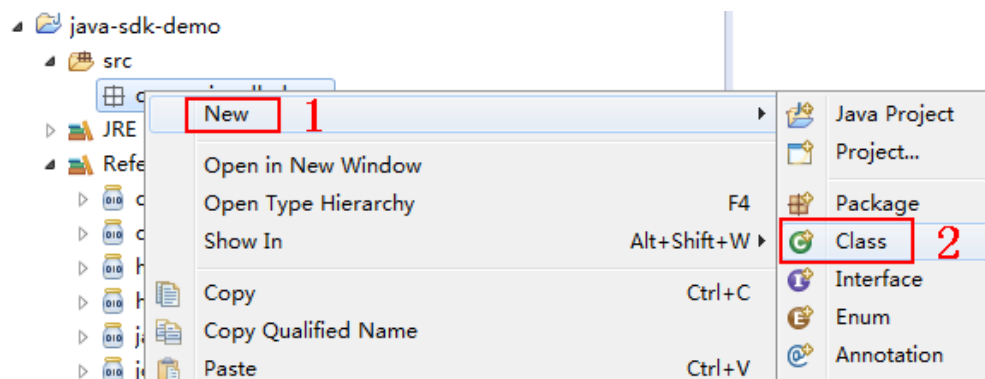
2. 在“Name”中输入“com.apig.sdk.demo”。

图 2-8 设置 Package 的名称



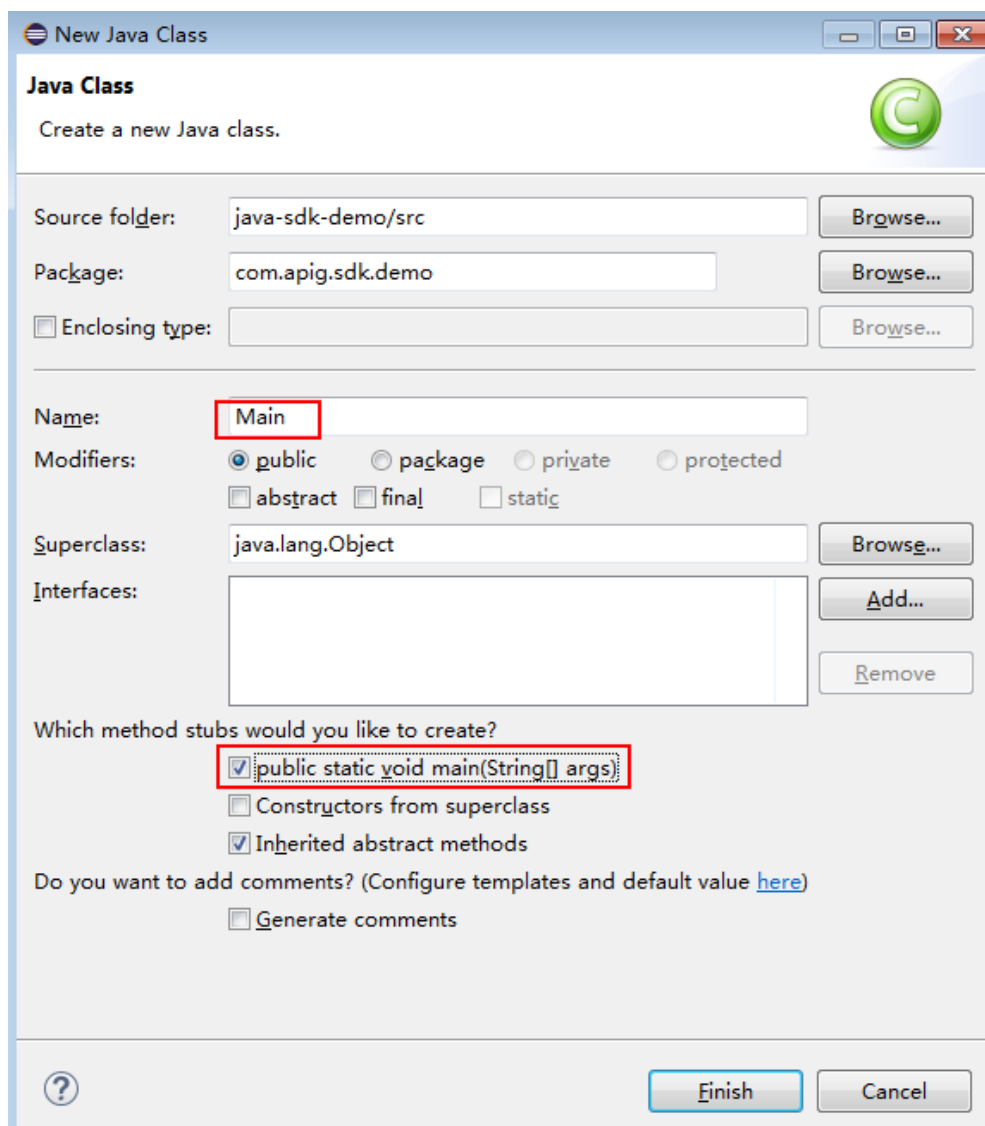
3. 单击“Finish”。
完成“Package”的创建。
4. 选择“com.apig.sdk.demo”，单击鼠标右键，选择“New > Class”。

图 2-9 新建 Class



5. 在“Name”中输入“Main”，勾选“public static void main(String[] args)”。

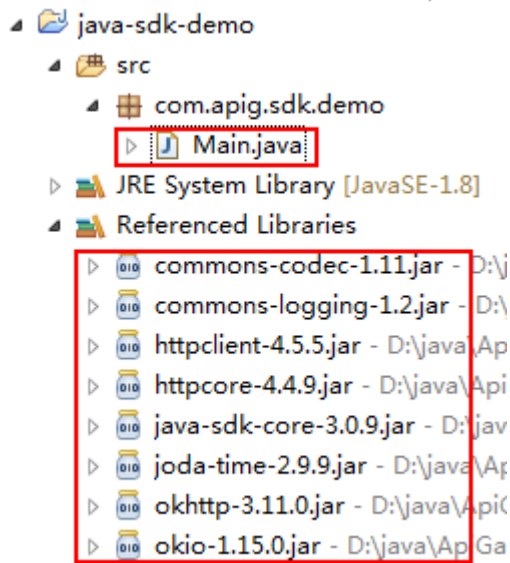
图 2-10 设置 Class 的配置



6. 单击“Finish”。
完成“Main”文件的创建。

步骤5 完成工程创建后，最终目录结构如下。

图 2-11 新建工程 Main 的目录结构



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

----结束

调用 API 示例

📖 说明

- 您需要在控制台创建一个API，可以选择Mock模式，并发布到环境上。创建及发布API的步骤请参见《API网关用户指南》。
- 示例API的后端为打桩的HTTP服务，此后端返回一个“200”响应码及“Congratulations, sdk demo is running”消息体。

步骤1 在“Main.java”中加入以下引用。

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

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

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

步骤2 创建request，过程中需要用到如下参数。

- AppKey和AppSecret：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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量，并将环境变量填入示例代码中。
 - a. 打开终端，输入以下命令打开环境变量配置文件。

vi ~/.bashrc

- b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

- c. 输入以下命令使配置文件生效。

source ~/.bashrc

- Method: 请求的方法名。根据API实际情况填写，示例代码使用“POST”作为样例。
- url: 请求的url，不包含QueryString及fragment部分。域名部分请使用绑定的域名，您自己绑定的独立域名。示例代码使用“http://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/java-sdk”作为样例。
- queryString: url携带参数的部分，根据API实际情况填写。支持的字符集为[0-9a-zA-Z./;[]\-=~#%^&_+: "]。示例代码使用“name=value”作为样例。
- header: 请求的头域，根据API实际情况填写。示例代码使用“Content-Type:text/plain”作为样例。如果API发布到非RELEASE环境时，需要增加自定义的环境名称，示例代码使用“x-stage:publish_env_name”作为样例。
- body: 请求的正文。根据API实际情况填写，示例代码使用“demo”作为样例。

样例代码如下：

```
Request request = new Request();  
try  
{  
    request.setKey(System.getenv("CLOUD_SDK_AK"));  
    request.setSecret(System.getenv("CLOUD_SDK_SK"));  
    request.setMethod("POST");  
    request.setUrl("http://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/java-  
sdk");  
    //url地址  
    request.addQueryStringParam("name", "value");  
    request.addHeader("Content-Type", "text/plain");  
    //request.addHeader("x-stage", "publish_env_name"); //如果API发布到非RELEASE环境，需要增加自  
定义的  
环境名称  
    request.setBody("demo");  
} catch (Exception e)  
{  
    e.printStackTrace();  
    return;  
}
```

步骤3 对请求进行签名、访问API并打印结果：

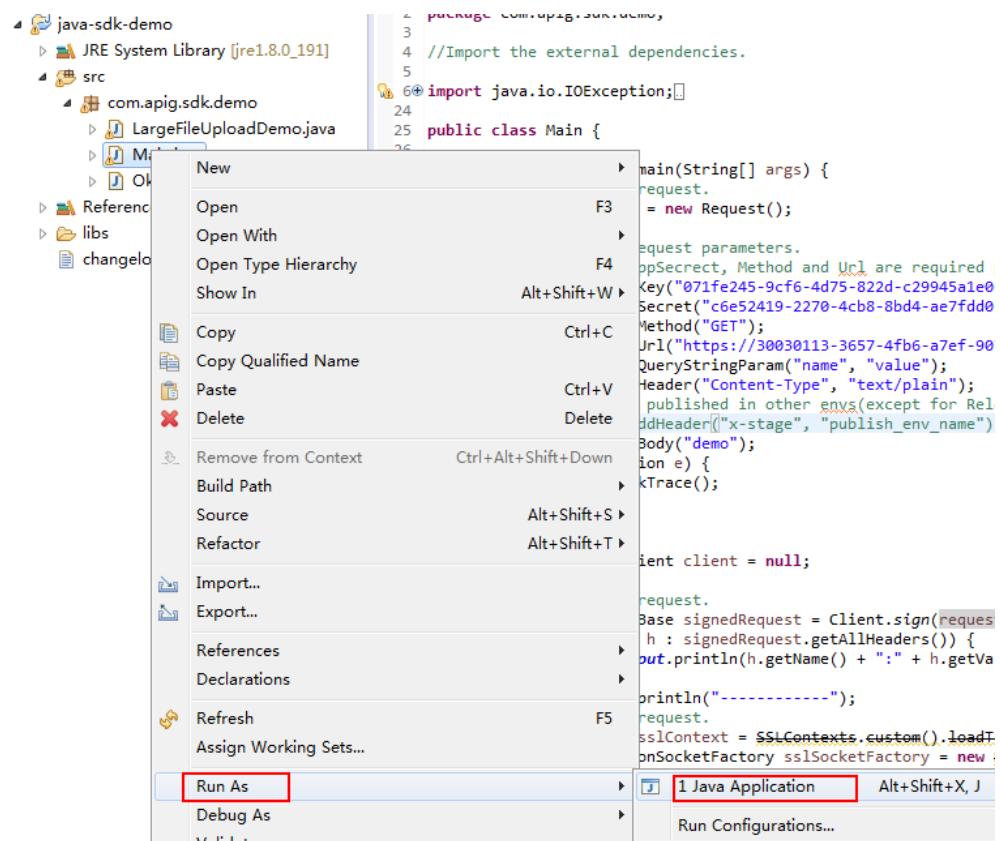
样例代码如下：

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

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

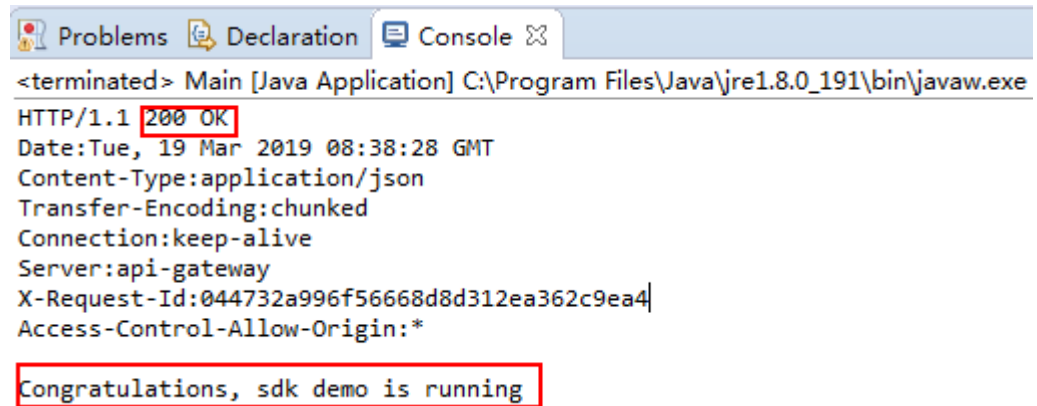
步骤4 选择“Main.java”，单击鼠标右键，选择“Run As > Java Application”，运行工程测试代码。

图 2-12 运行工程测试代码



步骤5 在“Console”页签，查看运行结果。

图 2-13 调用成功后的返回信息



```
Problems Declaration Console
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
HTTP/1.1 200 OK
Date: Tue, 19 Mar 2019 08:38:28 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Connection: keep-alive
Server: api-gateway
X-Request-Id: 044732a996f56668d8d312ea362c9ea4
Access-Control-Allow-Origin: *

Congratulations, sdk demo is running
```

----结束

2.4 Go

操作场景

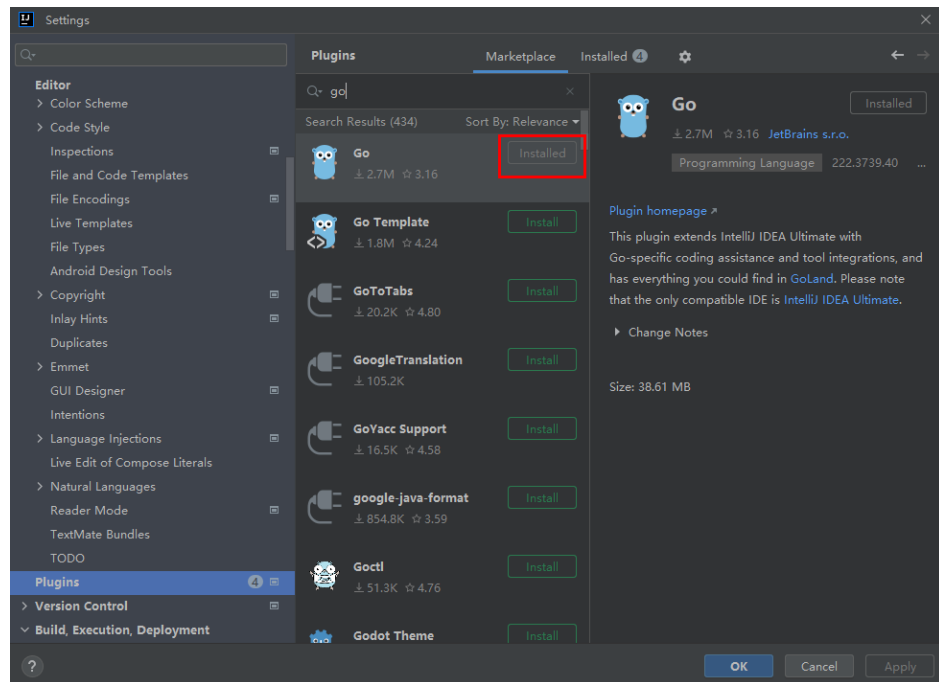
使用Go语言调用APP认证的API时，您需要先获取SDK，然后新建工程，最后参考调用API示例调用API。

本章节以IntelliJ IDEA 2022.2.1版本为例介绍。

前提条件

- 已获取API的调用信息，具体参见[认证前准备](#)。
- 已安装Go安装包1.14或以上版本，如果未安装，请至[Go官方下载页面](#)下载。
- 已安装IntelliJ IDEA 2022.2.1或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。
- 已在IntelliJ IDEA中安装Go插件，如果未安装，请按照[图2-14](#)所示安装。

图 2-14 安装 Go 插件



获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-go-sdk.zip”压缩包，解压后目录结构如下：

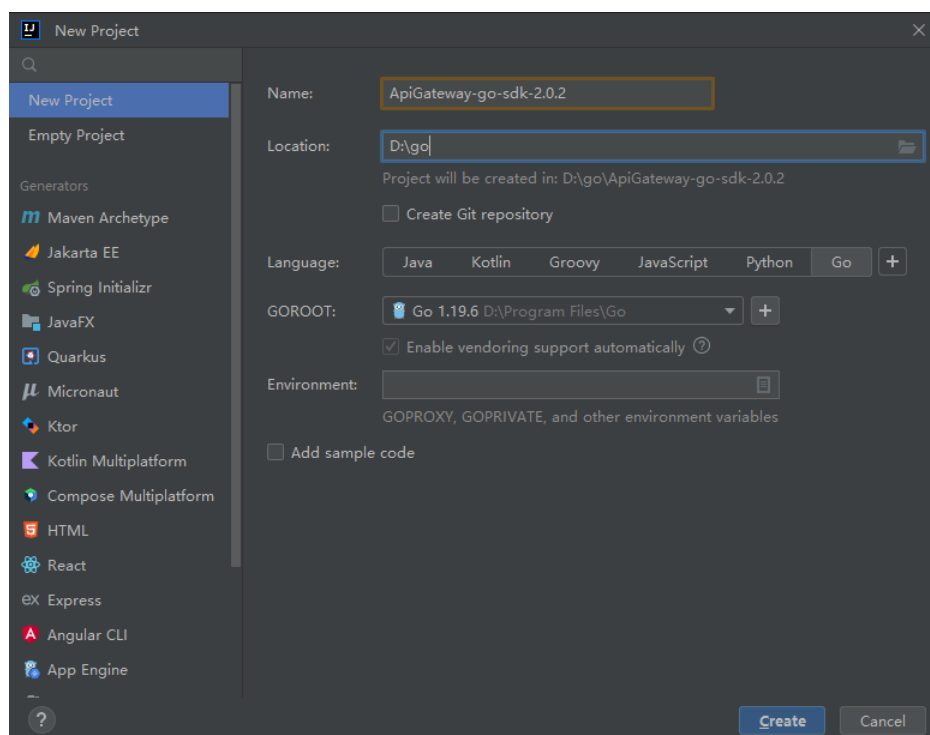
名称	说明
core\escape.go	SDK代码
core\signer.go	
demo.go	示例代码

新建工程

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

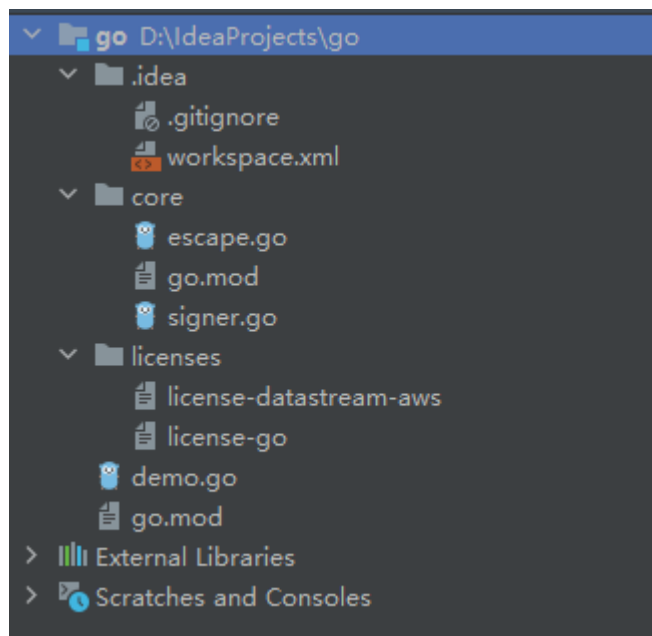
弹出“New Project”对话框，“Name”填写SDK压缩包内文件夹的名称，“Location”填写该文件夹的解压路径，“Language”选择“Go”，单击“Create”。

图 2-15 Go



步骤2 完成工程创建后，目录结构如下。

图 2-16 新建工程 go 的目录结构



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

----结束

调用 API 示例

步骤1 在工程中引入sdk (signer.go) 。

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

步骤2 生成一个新的Signer，输入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量。

a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个新的Signer，填入已设置的环境变量。

```
s := core.Signer{  
    Key: os.Getenv("CLOUD_SDK_AK"),  
    Secret: os.Getenv("CLOUD_SDK_SK"),  
}
```

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

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

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

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

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

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

步骤6 访问API，查看访问结果。

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

----结束

2.5 Python

操作场景

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

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

前提条件

- 已获取API的调用信息，具体请参见[认证前准备](#)。
- 已安装Python安装包2.7.9或3.X版本，如果未安装，请至[Python官方下载页面](#)下载。

Python安装完成后，在cmd/shell窗口中使用pip安装“requests”库。

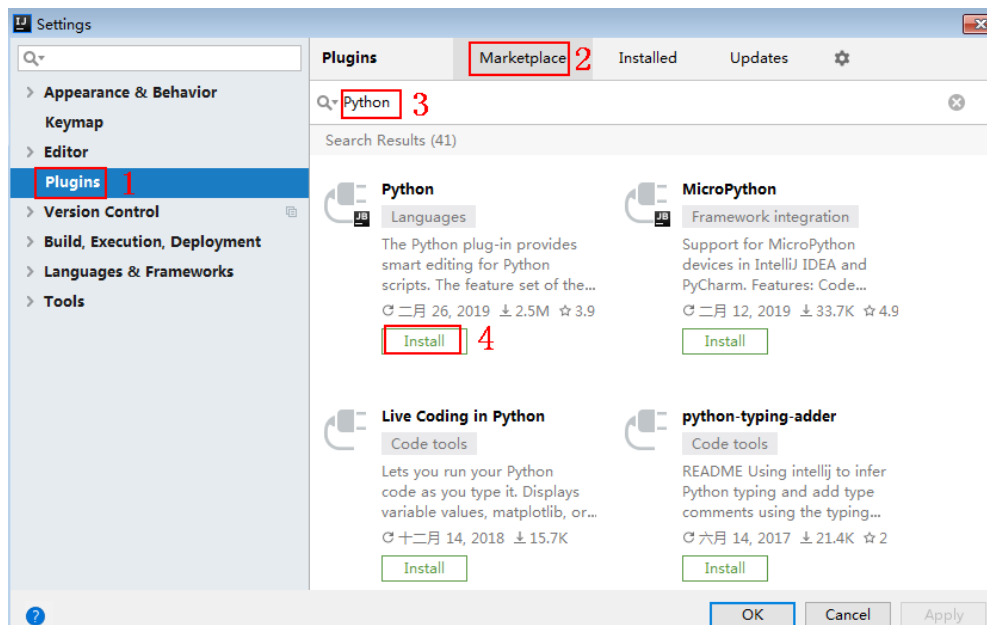
```
pip install requests
```

📖 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。

- 已安装IntelliJ IDEA 2018.3.5或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。
- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照[图2-17](#)所示安装。

图 2-17 安装 Python 插件



获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-python-sdk.zip”压缩包，解压后目录结构如下：

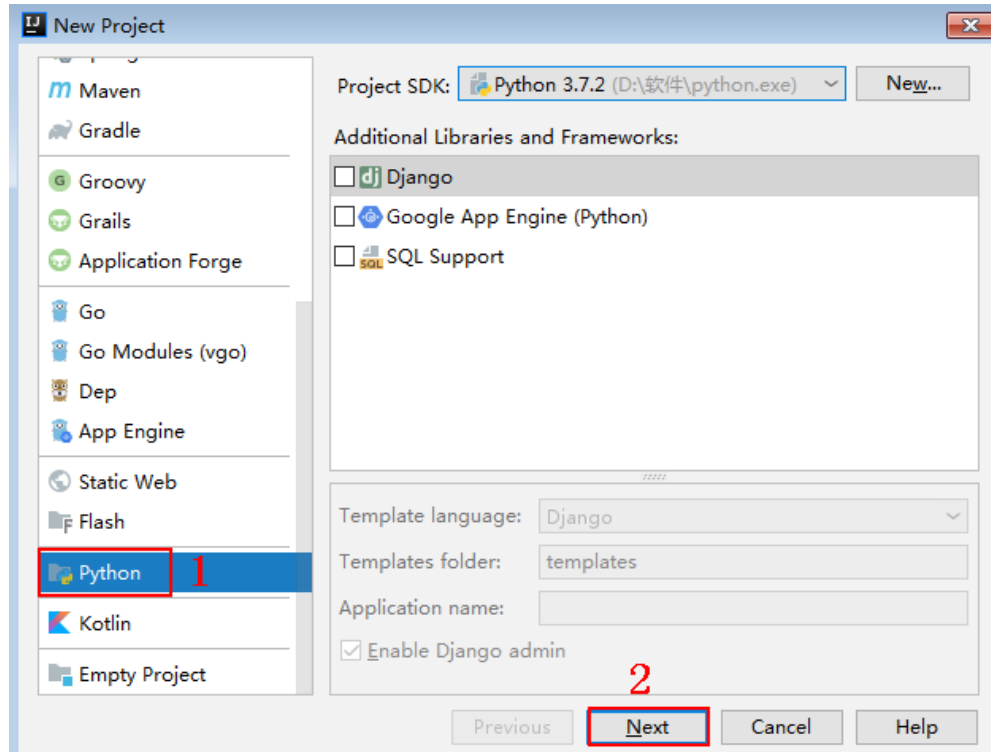
名称	说明
apig_sdk__init__.py	SDK代码
apig_sdk\signer.py	
main.py	示例代码
backend_signature.py	后端签名示例代码
licenses\license-requests	第三方库license文件

新建工程

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

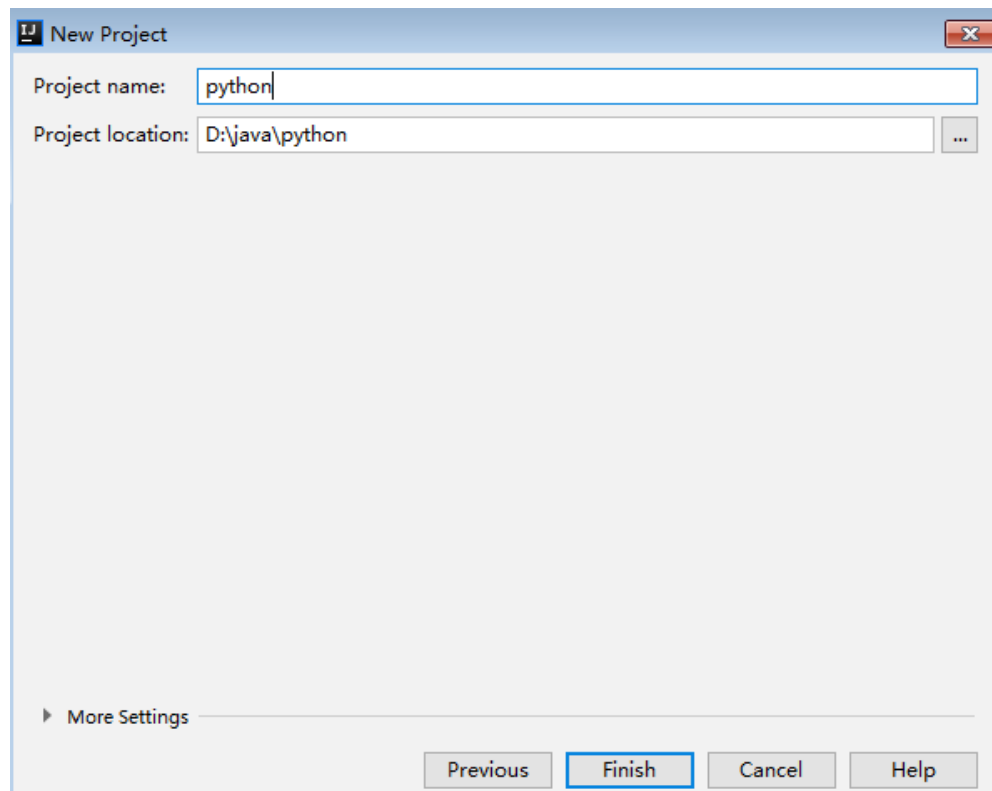
弹出“New Project”对话框，选择“Python”，单击“Next”。

图 2-18 Python



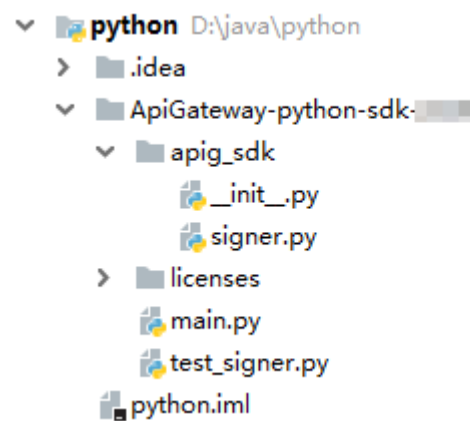
步骤2 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-19 选择解压后的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

图 2-20 新建工程 python 的目录结构



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

----结束

调用 API 示例

步骤1 在工程中引入apig_sdk。

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

步骤2 生成一个新的Signer，填入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量。

a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个新的Signer，填入已设置的环境变量。

```
sig = signer.Signer()  
sig.Key = os.getenv('CLOUD_SDK_AK')  
sig.Secret = os.getenv('CLOUD_SDK_SK')
```

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

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

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

📖 说明

X-Sdk-Date是一个必须参与签名的请求消息头参数。

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

步骤5 访问API，查看访问结果。

```
resp = requests.request(r.method, r.scheme + "://" + r.host + r.uri, headers=r.headers, data=r.body)  
print(resp.status_code, resp.reason)  
print(resp.content)
```

----结束

2.6 C#

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，具体参见[认证前准备](#)。
- 已安装Visual Studio 2019 version 16.8.4或以上版本，如果未安装，请至[Visual Studio官方网站](#)下载。

获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-csharp-sdk.zip”压缩包，解压后目录结构如下：

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

打开工程

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

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

调用 API 示例

步骤1 在工程中引入sdk。

```
using System;
using System.Net;
using System.IO;
using System.Net.Http;
using System.Threading;
using APIGATEWAY_SDK;
```

步骤2 生成一个新的Signer，填入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已[获取的AK/SK](#)设置为环境变量。
 - a. 打开终端，输入以下命令打开环境变量配置文件。
vi ~/.bashrc
 - b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"
export CLOUD_SDK_SK="已获取SK值"
```
 - c. 输入以下命令使配置文件生效。
source ~/.bashrc
2. 生成一个新的Signer，填入已设置的环境变量。

```
Signer signer = new Signer();
signer.Key = Environment.GetEnvironmentVariable("CLOUD_SDK_AK");
signer.Secret = Environment.GetEnvironmentVariable("CLOUD_SDK_SK");
```

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

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

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

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

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

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

步骤6 访问API，查看访问结果。

```
try
{
    var writer = new StreamWriter(req.GetRequestStream());
    writer.Write(r.body);
    writer.Flush();
    HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
    var reader = new StreamReader(resp.GetResponseStream());
    Console.WriteLine(reader.ReadToEnd());
}
catch (WebException e)
{
    HttpWebResponse resp = (HttpWebResponse)e.Response;
    if (resp != null)
    {
        Console.WriteLine((int)resp.StatusCode + " " + resp.StatusDescription);
        var reader = new StreamReader(resp.GetResponseStream());
        Console.WriteLine(reader.ReadToEnd());
    }
    else
    {
        Console.WriteLine(e.Message);
    }
}
Console.WriteLine("-----");
```

----结束

2.7 JavaScript

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**、**浏览器**等运行环境。

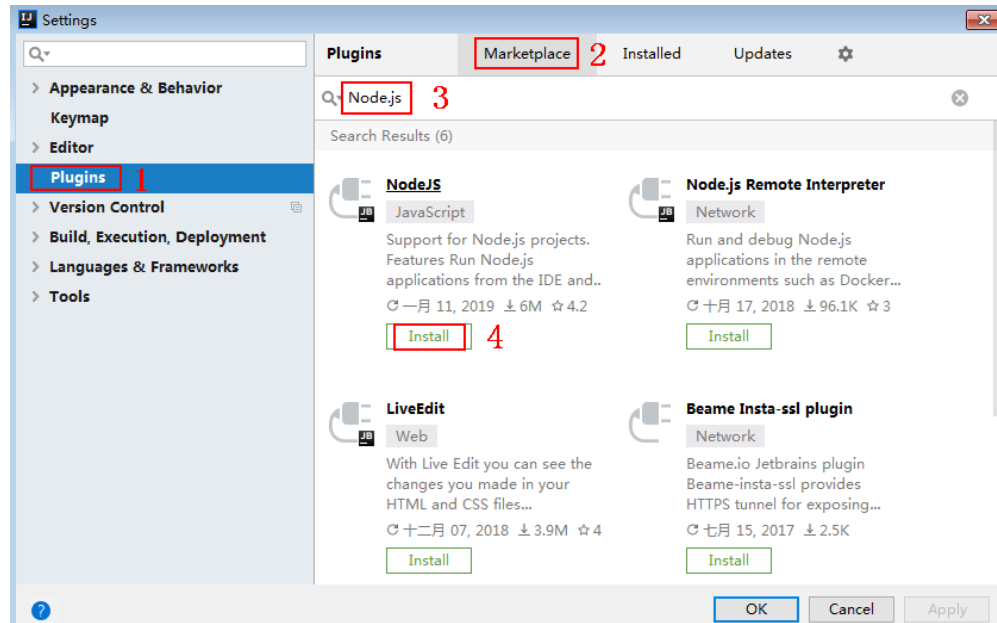
关于开发环境搭建，本章节以IntelliJ IDEA 2018.3.5版本、搭建Node.js环境为例。浏览器只提供代码示例说明。

准备环境

- 已获取API调用信息，具体参见[认证前准备](#)。
- 已安装Nodejs安装包15.10.0或以上版本，如果未安装，请至[Nodejs官方下载页面](#)下载。

- 已安装IntelliJ IDEA 2018.3.5或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。
- 已在IntelliJ IDEA中安装NodeJS插件，如果未安装，请按照图2-21所示安装。

图 2-21 安装 NodeJS 插件



获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-javascript-sdk.zip”压缩包，解压后目录结构如下：

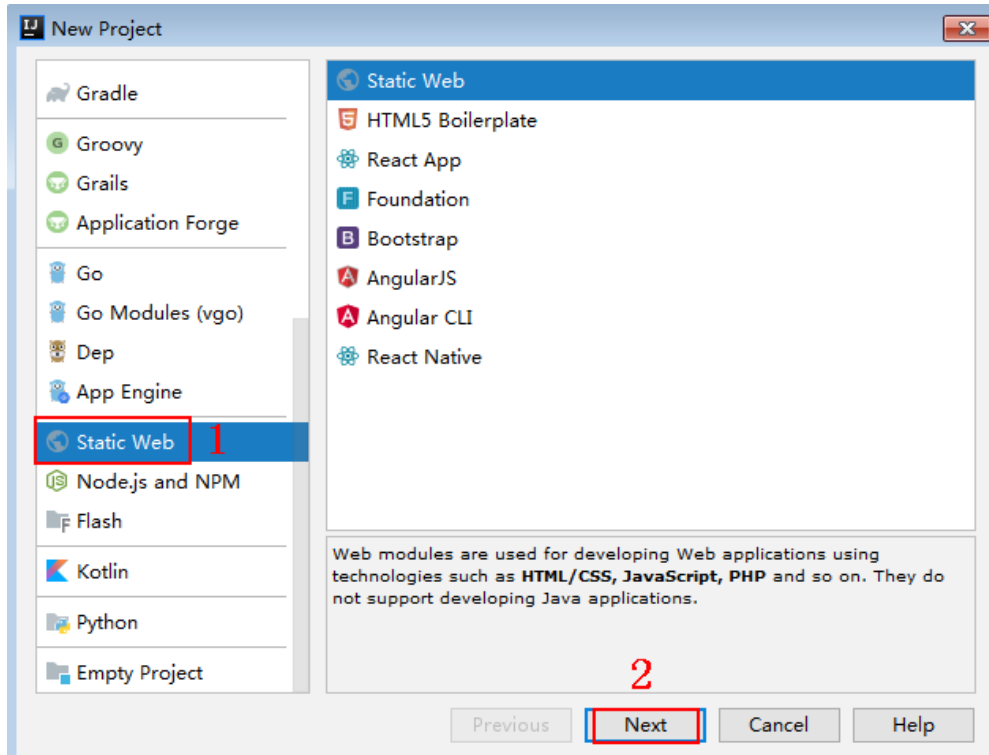
名称	说明
signer.js	SDK代码
node_demo.js	Nodejs示例代码
demo.html	浏览器示例代码
demo_require.html	浏览器示例代码（使用require加载）
test.js	测试用例
js\hmac-sha256.js	依赖库
licenses\license-crypto-js	第三方库license文件
licenses\license-node	

创建工程

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

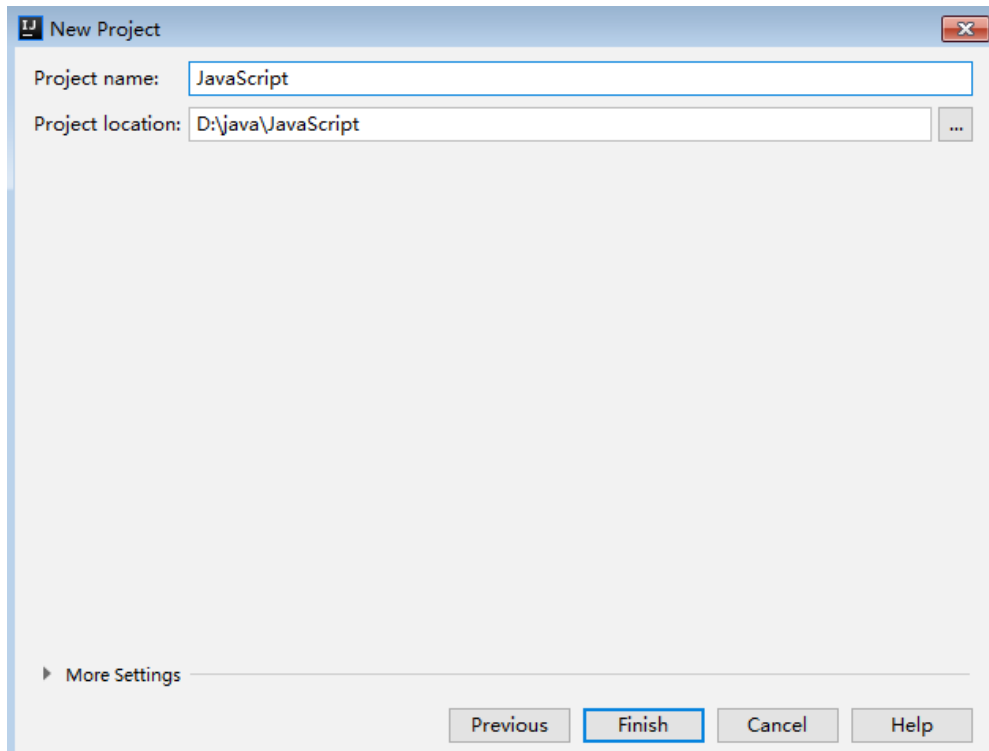
弹出“New Project”对话框。选择“Static Web”，单击“Next”。

图 2-22 Static Web



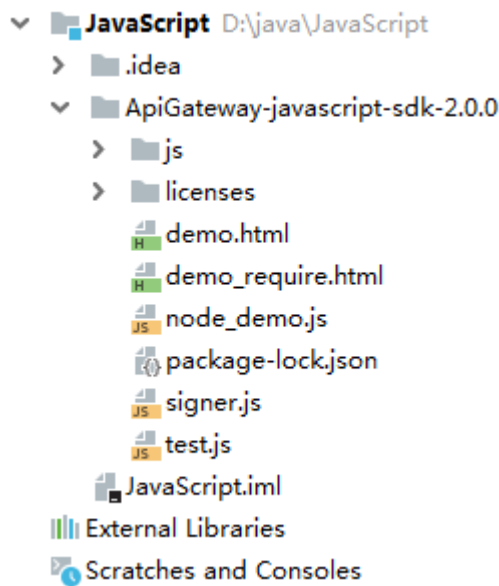
步骤2 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-23 选择解压后 JavaScript 的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

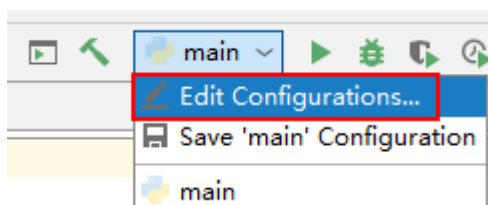
图 2-24 新建工程 JavaScript 的目录结构



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

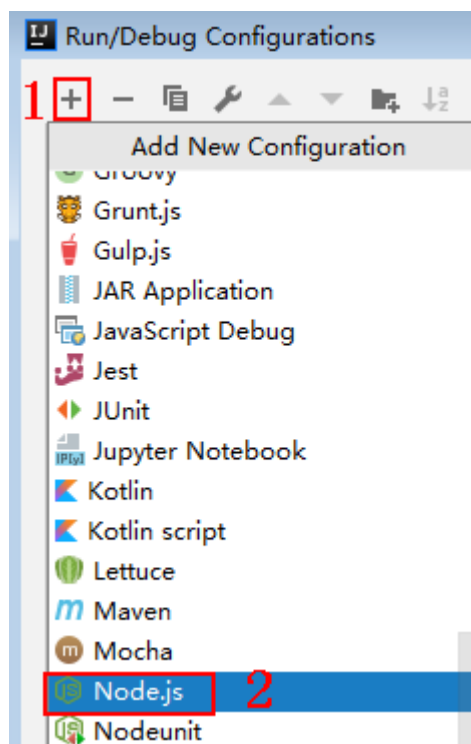
步骤4 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 2-25 Click Edit Configurations



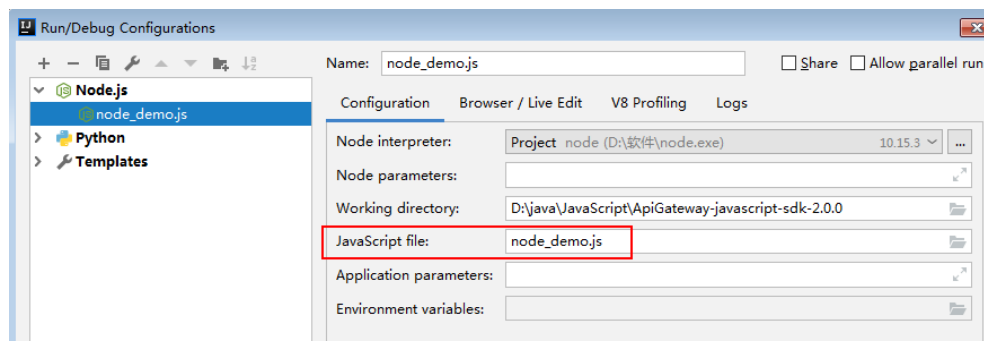
步骤5 单击“+”，选择“Node.js”。

图 2-26 选择 Node.js



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

图 2-27 选择 node_demo.js



----结束

调用 API (Node.js) 示例

步骤1 在工程中引入signer.js。

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

步骤2 生成一个新的Signer，填入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量。

a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

- b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

- c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个新的Signer，填入已设置的环境变量。

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

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

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

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

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

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

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

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

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

----结束

调用 API（浏览器）示例

- 步骤1** 由于浏览器限制从页面脚本内发起跨域请求，因此浏览器访问API前需要在API网关控制台上创建一个跨域共享策略，允许浏览器向跨域服务器发送XMLHttpRequest请求，并为跨域共享策略绑定API。

创建策略请参考《API网关用户指南》的“跨域共享策略说明”章节，绑定API请参考《API网关用户指南》的“创建策略并绑定API”章节。

图 2-28 创建跨域共享策略示例

创建策略

* 策略名称

* 策略类型 跨域资源共享

描述

策略内容 ☰ 🔒

* Allowed Origins ?

* Allowed Methods ? GET POST PUT

Allowed Headers ?

Exposed Headers ?

Maximum Age ?

Allow Credentials ?

步骤2 在html中引入signer.js及依赖。

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

步骤3 进行签名和访问。

1. 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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量。

- a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

- b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"
export CLOUD_SDK_SK="已获取SK值"
```

- c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 填入已设置的环境变量，进行签名和访问。

```
var sig = new signer.Signer()
sig.Key = getenv('CLOUD_SDK_AK')
sig.Secret = getenv('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
});
```

----结束

2.8 PHP

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。

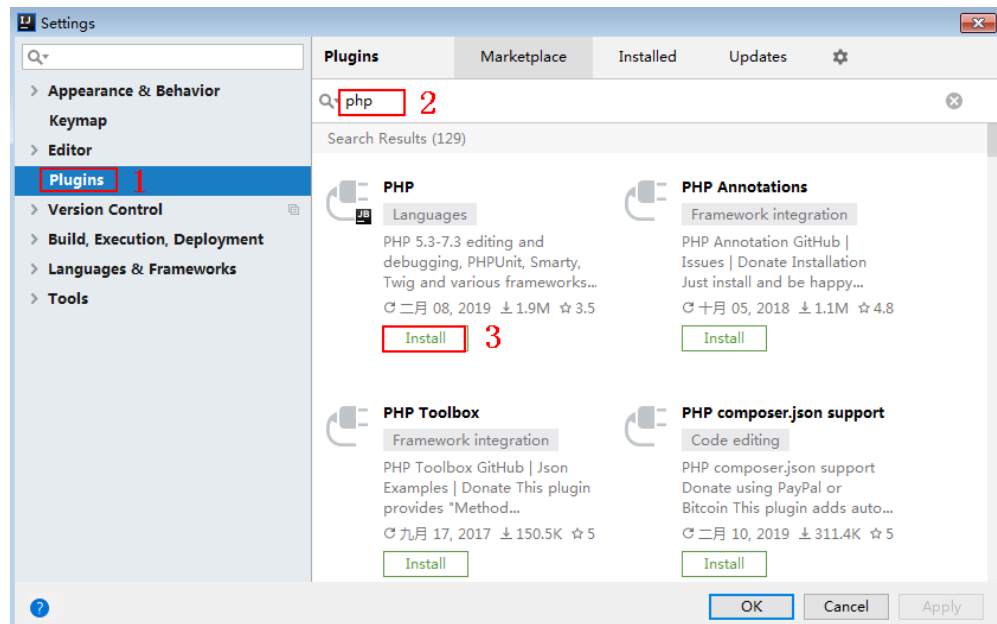
前提条件

- 已获取API的调用信息，具体参见[认证前准备](#)。
- 已安装IntelliJ IDEA 2018.3.5或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。
- 已安装PHP安装包8.0.3或以上版本，如果未安装，请至[PHP官方下载页面](#)下载。
- 将PHP安装目录中的“php.ini-production”文件复制到“C:\windows”，改名为“php.ini”，并在文件中增加如下内容。

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

- 已在IntelliJ IDEA中安装PHP插件，如果未安装，请按照[图2-29](#)所示安装。

图 2-29 安装 PHP 插件



获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-php-sdk.zip”压缩包，解压后目录结构如下：

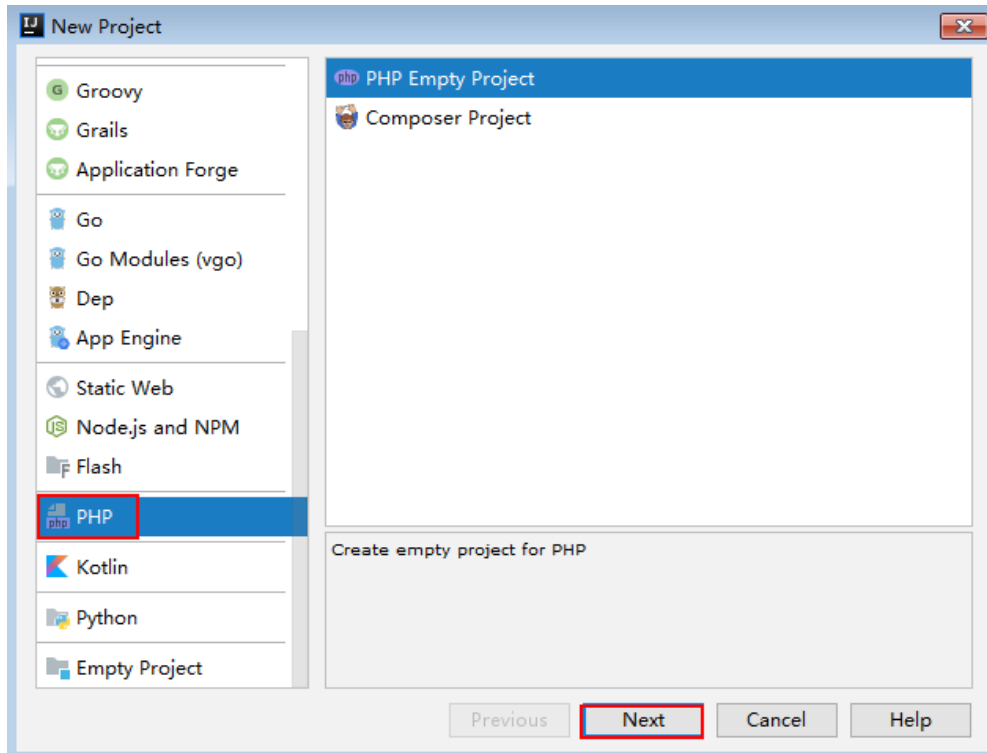
名称	说明
signer.php	SDK代码
index.php	示例代码

新建工程

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

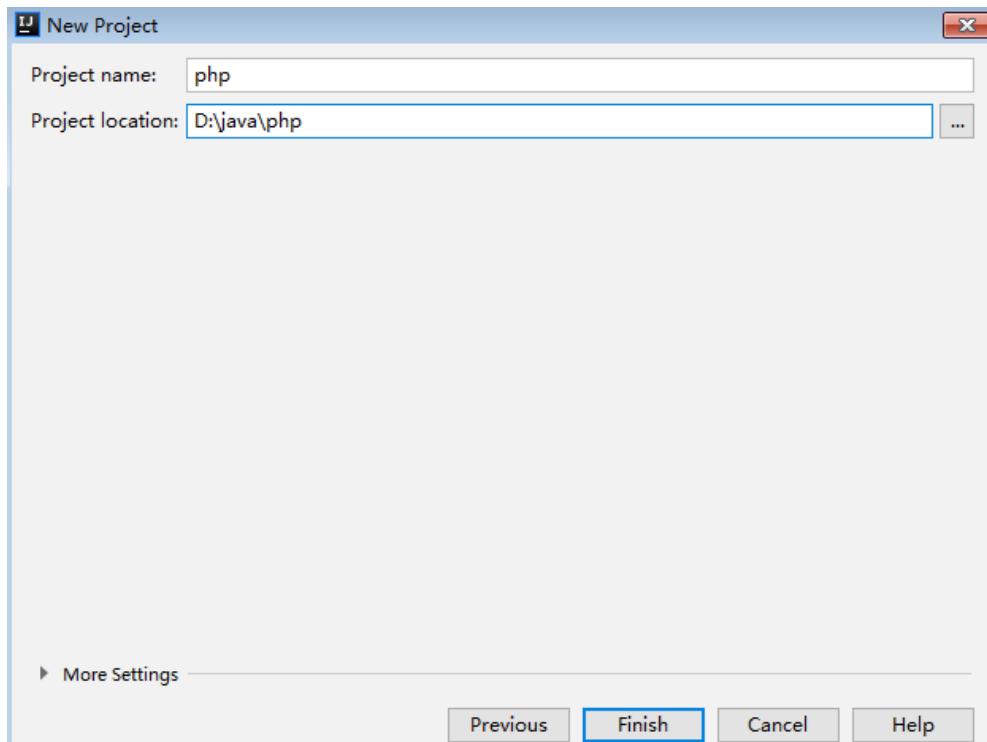
弹出“New Project”对话框，选择“PHP”，单击“Next”。

图 2-30 PHP



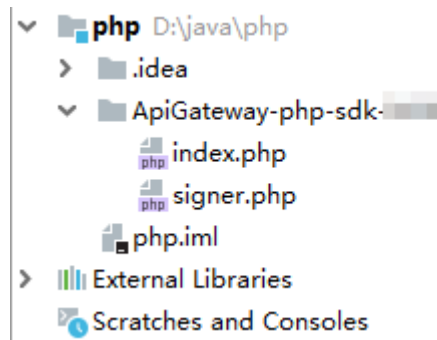
步骤2 单击“...”，在弹出的对话框中选择解压后的SDK路径，单击“Finish”。

图 2-31 选择解压后 php 的 SDK 路径



步骤3 完成工程创建后，目录结构如下。

图 2-32 新建工程 php 的目录结构



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

----结束

调用 API 示例

步骤1 在代码中引入sdk。

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

步骤2 生成一个新的Signer，填入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已[获取的AK/SK](#)设置为环境变量。

a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"
export CLOUD_SDK_SK="已获取SK值"
```

c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个新的Signer，填入已设置的环境变量。

```
$signer = new Signer();
$signer->Key = getenv('CLOUD_SDK_AK');
$signer->Secret = getenv('CLOUD_SDK_SK');
```

步骤3 生成一个新的Request，指定方法名、请求url和body（body根据实际的接口请求指定）。

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

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

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

步骤5 进行签名，执行此函数会生成一个\$curl上下文变量。

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

步骤6 访问API，查看访问结果。

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

```
echo $response;  
curl_close($curl);
```

----结束

2.9 C++

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

1. 已获取API的调用信息，具体参见[认证前准备](#)。
2. 安装openssl库。

```
apt-get install libssl-dev
```
3. 安装curl库。

```
apt-get install libcurl4-openssl-dev
```

获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-cpp-sdk.zip”压缩包，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	
signer.h	
Makefile	Makefile文件
main.cpp	示例代码

调用 API 示例

步骤1 在main.cpp中加入以下引用。

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


步骤2 生成一个新的Signer，填入AppKey和AppSecret。

1. 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。以Linux系统为例在本地将已[获取的AK/SK](#)设置为环境变量。

- a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

- b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

- c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个新的Signer，填入已设置的环境变量。

```
Signer signer(getenv("CLOUD_SDK_AK"), getenv("CLOUD_SDK_SK"));
```

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

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

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

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

步骤5 进行签名，执行此函数会将生成的签名头加入request变量中。

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

步骤6 使用curl库访问API，查看访问结果。

```
static size_t  
WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)  
{  
    size_t realsize = size * nmemb;  
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;  
  
    mem->memory = (char*)realloc(mem->memory, mem->size + realsize + 1);  
    if (mem->memory == NULL) {  
        /* out of memory! */  
        printf("not enough memory (realloc returned NULL)\n");  
        return 0;  
    }  
  
    memcpy(&(mem->memory[mem->size]), contents, realsize);  
    mem->size += realsize;  
    mem->memory[mem->size] = 0;  
  
    return realsize;  
}  
  
//send http request using curl library  
int perform_request(RequestParams* request)  
{  
    CURL *curl;  
    CURLcode res;  
    struct MemoryStruct resp_header;  
    resp_header.memory = (char*)malloc(1);  
    resp_header.size = 0;  
    struct MemoryStruct resp_body;  
    resp_body.memory = (char*)malloc(1);  
    resp_body.size = 0;  
  
    curl_global_init(CURL_GLOBAL_ALL);  
    curl = curl_easy_init();  
  
    curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, request->getMethod().c_str());
```

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

curl_global_cleanup();

return 0;
}
```

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

----结束

2.10 C

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

1. 已获取API的调用信息，具体参见[认证前准备](#)。
2. 安装openssl库。
apt-get install libssl-dev
3. 安装curl库。
apt-get install libcurl4-openssl-dev

获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-c-sdk.zip”压缩包，解压后目录结构如下：

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

调用 API 示例

步骤1 在main.c中加入以下引用。

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

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

1. 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。以Linux系统为例在本地将已[获取的AK/SK](#)设置为环境变量。

a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"
export CLOUD_SDK_SK="已获取SK值"
```

c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 生成一个sig_params_t类型的变量，填入已设置的环境变量。

```
sig_params_t params;
sig_params_init(&params);
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;
```

步骤3 指定方法名、域名、请求uri、查询字符串和body。

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

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

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

步骤5 进行签名，执行此函数会将生成的签名头加入request变量中。

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

步骤6 使用curl库访问API，查看访问结果。

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

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

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

    return realsize;
}

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

    curl_global_init(CURL_GLOBAL_ALL);
    curl = curl_easy_init();

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

```
free(resp_header.memory);
free(resp_body.memory);
curl_easy_cleanup(curl);

curl_global_cleanup();

//free signature params
sig_params_free(&params);
return 0;
}
```

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

----结束

2.11 Android

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，具体参见[认证前准备](#)。
- 已安装Android Studio 4.1.2或以上版本，如果未安装，请至[Android Studio官方网站](#)下载。

获取 SDK

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

获取“ApiGateway-android-sdk.zip”压缩包，解压后目录结构如下：

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

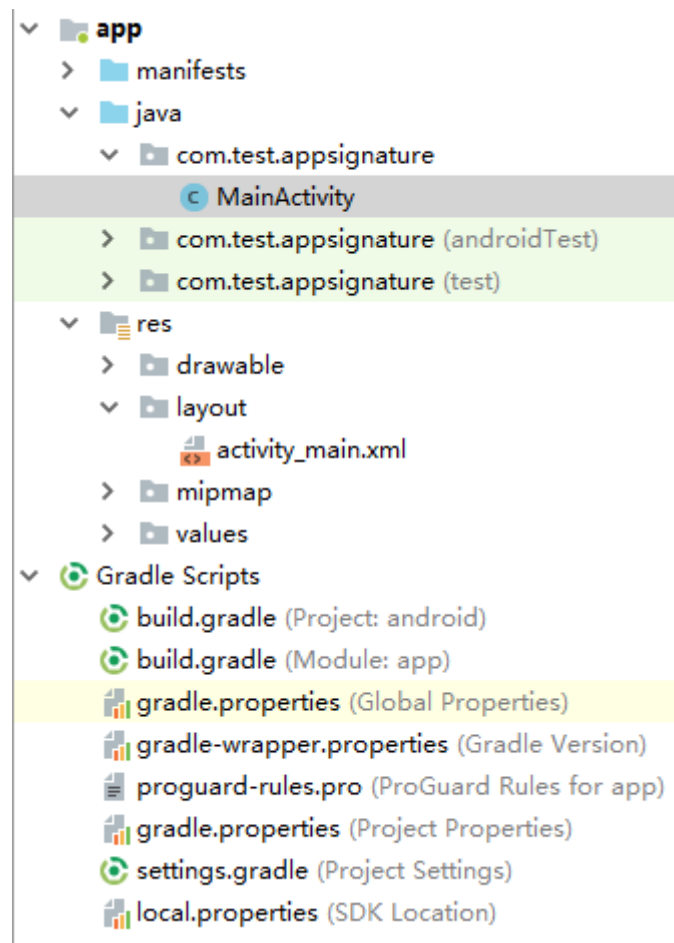
打开工程

步骤1 打开Android Studio，选择“File > Open”。

在弹出的对话框中选择解压后的SDK路径。

步骤2 打开工程后，目录结构如下。

图 2-33 工程目录结构



----结束

调用 API 示例

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

- java-sdk-core-x.x.x.jar
- commons-logging-1.2.jar
- joda-time-2.10.jar

步骤2 在“build.gradle”文件中加入okhttp库的依赖。

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

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

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

1. 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。以Linux系统为例在本地将已获取的AK/SK设置为环境变量。

- a. 打开终端，输入以下命令打开环境变量配置文件。

```
vi ~/.bashrc
```

- b. 设置环境变量，保存文件并退出编辑器。

```
export CLOUD_SDK_AK="已获取AK值"  
export CLOUD_SDK_SK="已获取SK值"
```

- c. 输入以下命令使配置文件生效。

```
source ~/.bashrc
```

2. 创建request，填入已设置的环境变量，并指定域名、方法名、请求uri和body。

```
Request request = new Request();  
try {  
    request.setKey(System.getenv("CLOUD_SDK_AK"));  
    request.setSecret(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;  
}
```

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

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

----结束

2.12 curl

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，具体参见[认证前准备](#)。

调用 API 示例

- 步骤1** 使用JavaScript SDK生成curl命令。

请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。

在浏览器中打开demo.html，页面如下图所示。

Apigateway Signature Test

Key

Secret

Method Url

Headers

Body


```
curl -X GET "http://30030113-3657-4fb6-a7ef-90764239b038.apigw.exampleRegion.com/" -H "X-Sdk-Date: 20190731T065514Z" -H "host: 30030113-3657-4fb6-a7ef-9076423
```

Note: accessing the API from browser requires [support for CORS](#)

200
Congratulations, sdk demo is running

步骤2 填入Key、Secret、方法名、请求协议、域名和url。

```
Key=4f5f****100c  
Secret=*****  
Method=POST  
Url=https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com
```

步骤3 填入json格式的Query和Headers，填入Body。

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

```
$ curl -X POST "https://c967a237-cd6c-470e-906f-a8655461897e.apigw.exampleRegion.com/" -H "X-Sdk-Date: 20180530T115847Z" -H "Authorization: SDK-HMAC-SHA256 Access=071fe245-9cf6-4d75-822d-c29945a1e06a, SignedHeaders=host;x-sdk-date, Signature=9e5314bd156d517*****dd3e5765fdde4" -d ""  
Congratulations, sdk demo is running
```

📖 说明

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

----结束

3 使用 IAM 认证调用 API

3.1 Token 认证

操作场景

当您使用Token认证方式调用API时，需要获取用户Token并在调用API时将Token值设置到调用请求的“X-Auth-Token”头域中。

📖 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 API 示例

1. 获取Token，并设置成环境变量，Token用于后续调用其他接口鉴权。

- a. 获取用户Token，请求内容示例如下：

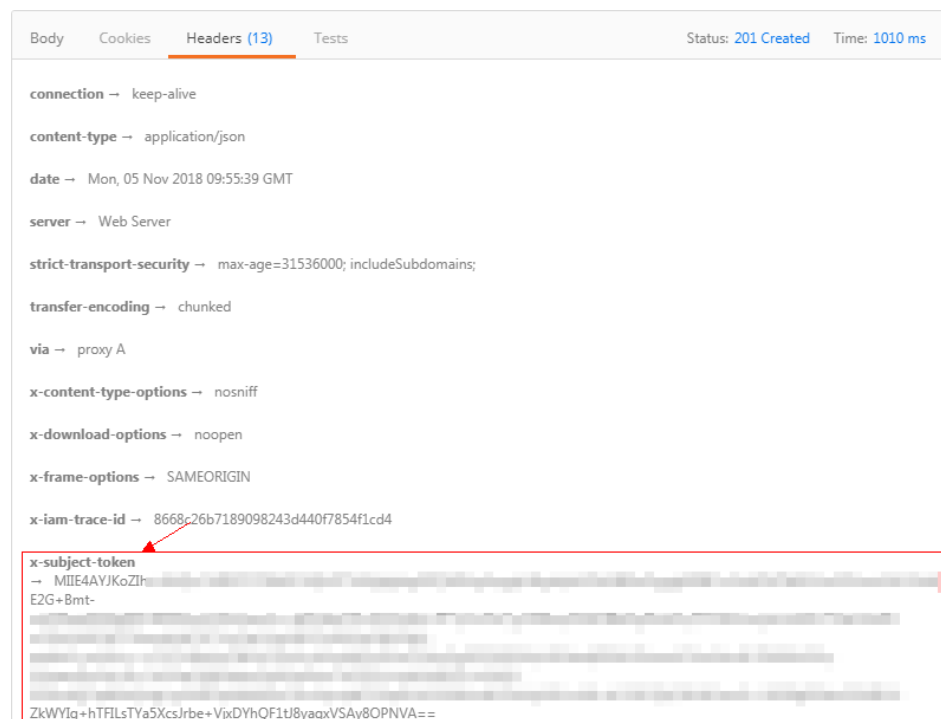
```
curl -X POST https://{iam_endpoint}/v3/auth/tokens -H 'content-type: application/json' -d '{
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "name": "{user_name}",
          "domain": {
            "name": "{user_name}"
          }
        },
        "password": "{password}"
      }
    },
    "scope": {
      "project": {
        "id": "{project_id}"
      }
    }
  }
}' -vk
```

上述命令中，部分参数请参见以下说明进行修改，具体请参考《统一身份认证服务API参考》的获取用户Token接口：

- `{iam_endpoint}`请参见[地区和终端节点](#)获取。
- `{user_name}`和`{password}`分别替换为连接IAM服务器的用户名和密码。
- `{project_id}`为项目ID。在管理控制台，将鼠标移至用户名，在下拉列表中单击“我的凭证”，查看“项目ID”。

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，如下图所示：

图 3-1 从返回消息的 Header 中获取 X-Subject-Token



- b. 使用如下命令将token设置为环境变量，方便后续事项。

```
export Token={X-Subject-Token}
```

X-Subject-Token即为1.a获取到的token，命令示例如下：

```
export Token=MIIDkg*****BZQMEAgEwg
```

2. 调用API，请参考[认证前准备](#)获取域名、请求方法和URL，并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。参数请根据实际情况填写。

```
curl -X 请求方法 域名+URL -H "x-auth-token: $Token" -vk
```

3.2 AK/SK 认证

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 3-2 credential.csv 文件内容

	A	B	C
1	User Name	Access Key Id	Secret Access Key
2	QWSDSDJWQ7WAG7WEWEDSD	zuIJJ'J' _ GBN77' .. :.....qXbhSzsdGwf4

AK/SK生成步骤：

1. 登录控制台。
2. 将鼠标移至用户名，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. 按照界面提示输入验证码或登录密码，单击“确定”，下载密钥，请妥善保管。

生成签名

生成签名的方式和APP认证相同，用AK代替APP认证中的AppKey，SK替换APP认证中的AppSecret，即可完成签名和请求。您可使用[Java](#)、[Go](#)、[Python](#)、[C#](#)、[JavaScript](#)、[PHP](#)、[C++](#)、[C](#)、[Android](#)进行签名和访问。

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。

API网关（即API管理）除了校验时间格式外，还会校验该时间值与网关收到请求的时间差，如果时间差大于15分钟，API网关将拒绝请求。

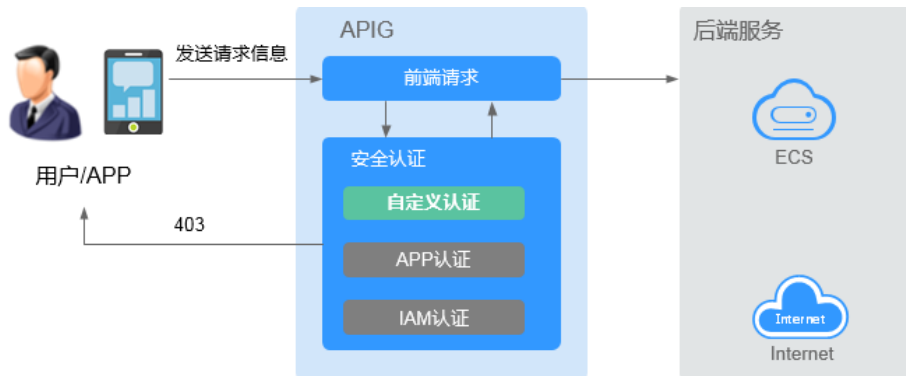
4 创建用于前端自定义认证的函数

操作场景

如果您想要使用自己的认证系统对API的访问进行认证鉴权，您可以在API管理中创建一个前端自定义认证来实现此功能。在使用前端自定义认证对前端请求进行认证鉴权前，您需要先在FunctionGraph创建一个函数，通过函数定义您所需的认证信息。函数创建完后，作为自定义认证的后端函数，对API网关中的API进行认证鉴权。

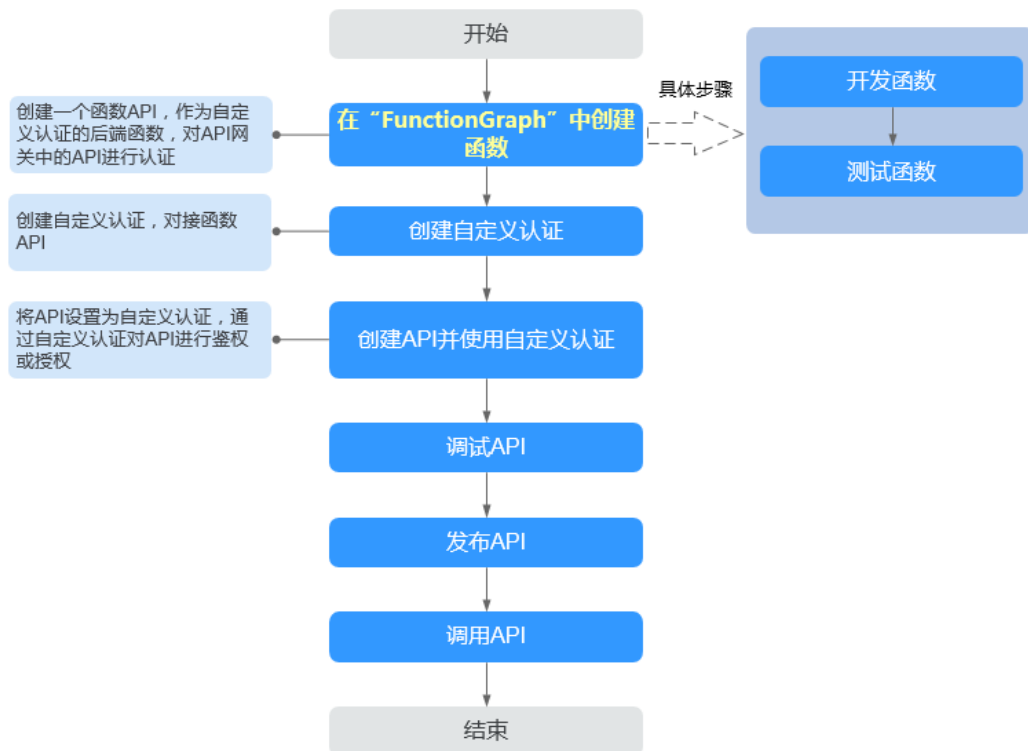
本章节介绍如何将校验函数封装成一个“自定义认证”，以及封装成自定义认证过程中的操作注意事项。

图 4-1 前端自定义认证示意图



使用自定义认证调用API的流程如下图所示：

图 4-2 自定义认证调用 API



说明

自定义认证依赖函数服务。如果当前Region没有上线函数服务，则不支持使用自定义认证。

操作步骤

步骤1 在FunctionGraph中开发函数。

下面以python2.7语言为例，函数代码需要满足如下条件：

- 函数代码支持三种请求参数定义，格式为：
 - Header中的请求参数：event["headers"]["参数名"]
 - Query中的请求参数：event["queryStringParameters"]["参数名"]
 - 您自定义的用户数据：event["user_data"]
- 函数代码获取的三种请求参数与API网关自定义认证中的参数关系如下所示：
 - Header中的请求参数：对应自定义认证中参数位置为Header的身份来源，其参数值在您调用使用该前端自定义认证的API时传入
 - Query中的请求参数：对应自定义认证中参数位置为Query的身份来源，其参数值在您调用使用该前端自定义认证的API时传入
 - 您自定义的用户数据：对应自定义认证中的用户数据，其参数值在您创建自定义认证时输入

- 函数的返回值不能大于1M，必须满足如下格式：

```
{
  "statusCode":200,
  "body": "{\"status\": \"allow\", \"context\": {\"user\": \"abc\"}}"
}
```

其中，body字段的内容为字符串格式，json解码之后为：

```
{
  "status": "allow/deny",
  "context": {
    "user": "abc"
  }
}
```

- “statusCode” 字段为必选，函数服务正常且自定义认证函数代码符合规范时，statusCode的值则为自定义认证函数的响应码。
 - 调用自定义认证的API，当自定义认证函数的响应码为非200时，API网关认为函数服务异常，并返回错误码“500”，错误信息为“Internal server error”。
 - 调用自定义认证的API，如果自定义认证开启宽松模式，当自定义认证函数连接失败、返回“500”或者“503”时，自定义认证不会校验body字段里面的status字段，直接返回调用成功，同时从函数代码中获取到的context字段也为空。
- “status” 字段为必选，用于标识认证结果。只支持“allow”或“deny”，“allow”表示认证成功，“deny”表示认证失败。
- “context” 字段为可选，支持字符串类型键值对，键值不支持JSON对象或数组。

context中的数据为您自定义的字段，认证通过后作为认证参数映射到API网关后端参数中，其中context中的参数名称与系统参数名称必须完全一致，且区分大小写，context中的参数名称必须以英文字母开头，支持英文大小写字母、数字、下划线和中划线，且长度为1~32个字符。

前端认证通过后，context中的user的值abc映射到后端服务Header位置的test参数中。

Header中的请求参数定义代码示例：

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event["headers"].get("test")=="abc":
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status": "allow",
                "context": {
                    "user": "abcd"
                }
            })
        }
    else:
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status": "deny",
            })
        }
    return json.dumps(resp)
```

Query中的请求参数定义代码示例：

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event["queryStringParameters"].get("test")=="abc":
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status": "allow",
            })
        }
```

```
        "context":{
            "user":"abcd"
        }
    })
}
else:
    resp = {
        'statusCode': 200,
        'body': json.dumps({
            "status":"deny",
        })
    }
return json.dumps(resp)
```

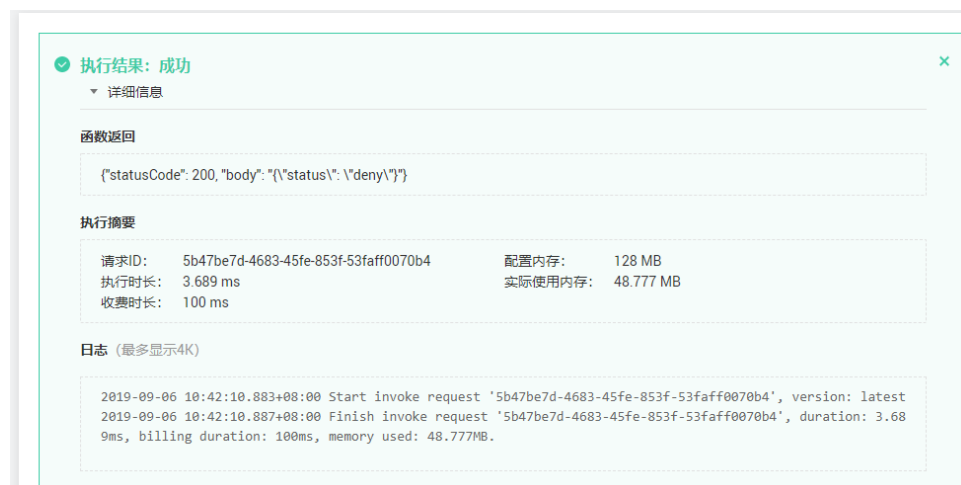
用户数据定义代码示例:

```
# -*- coding:utf-8 -*-
import json
def handler(event, context):
    if event.get("user_data")=='abc':
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"allow",
                "context":{
                    "user":"abcd"
                }
            })
        }
    else:
        resp = {
            'statusCode': 200,
            'body': json.dumps({
                "status":"deny",
            })
        }
    return json.dumps(resp)
```

步骤2 测试函数。在测试事件的“事件模板”中选择“apig-event-template”，根据实际情况修改后保存测试模板，单击“测试”。

执行结果为“成功”时，表示测试成功。

接下来您需要进入API网关界面创建前端自定义认证。



----结束

后续操作

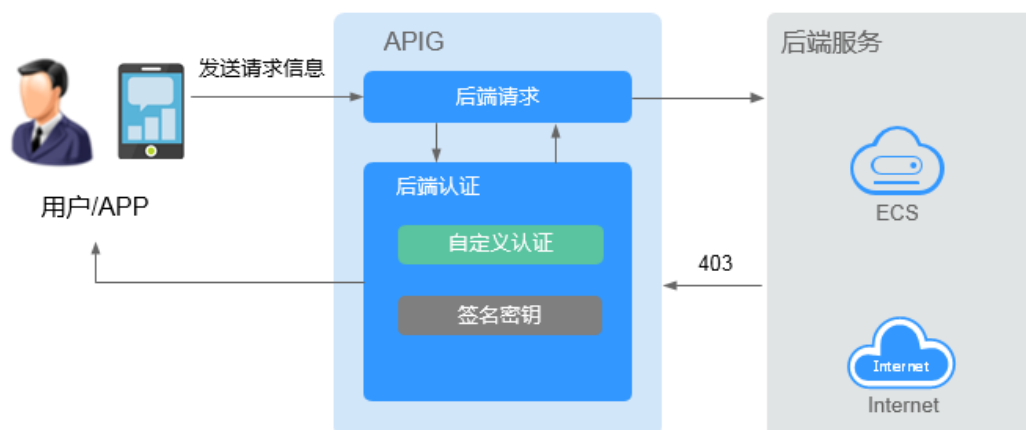
在自定义认证中已经创建完成用于前端自定义认证的Function API，下一步您需要进入API网关中创建前端自定义认证，具体操作请参见《API网关用户指南》的“创建自定义认证”章节。

5 创建用于后端自定义认证的函数

操作场景

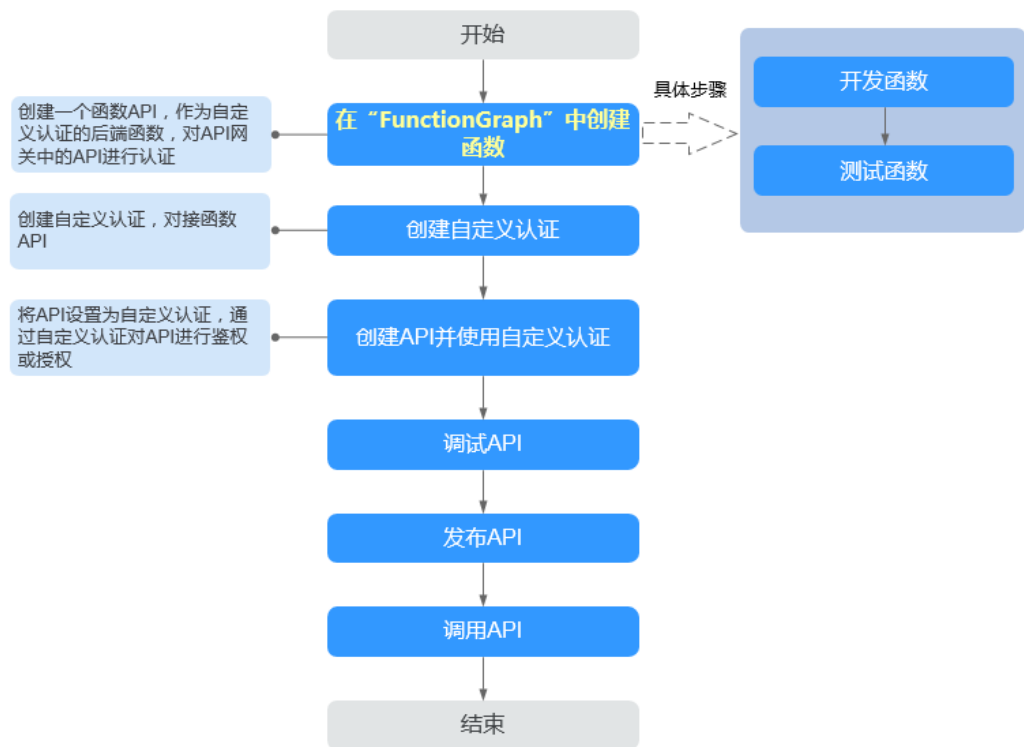
如果您需要使用一种认证机制对接多个不同的外部认证系统，实现对于后端服务的保护，您可以通过API网关中的后端自定义认证实现此功能。在使用后端自定义认证对后端请求进行认证授权前，您需要先在FunctionGraph创建一个函数，通过函数定义您所需的认证信息。函数作为自定义认证的后端函数，对API网关中的API进行认证授权。

图 5-1 后端自定义认证示意图



使用自定义认证调用API的流程如下图所示：

图 5-2 使用自定义认证调用 API



说明

自定义认证依赖函数服务。如果当前Region没有上线函数服务，则不支持使用自定义认证。

操作步骤

步骤1 在FunctionGraph中开发函数。

下面以python2.7为例，函数代码需要满足如下条件：

- 函数代码只支持您自定义的用户数据，且它的格式为：`event["user_data"]`。
- 函数代码获取的请求参数与API网关自定义认证中的参数关系为：函数请求参数中的自定义用户数据对应API网关自定义认证中的用户数据，参数值在您创建API网关自定义认证时输入，用户数据格式不限制，您可以自行指定。
- 函数的返回值不能大于1M，必须满足如下格式：

```
{
  "statusCode":200,
  "body": "{\"status\": \"allow\", \"context\": {\"user\": \"abc\"}}"
```

其中，body字段的内容为字符串格式，json解码之后为：

```
{
  "status": "allow/deny",
  "context": {
    "user": "abc"
  }
}
```

- “statusCode”字段为必选，函数服务正常且自定义认证函数代码符合规范时，statusCode的值则为自定义认证函数的响应码。

- 调用自定义认证的API，当自定义认证函数的响应码为非200时，API网关认为函数服务异常，并返回错误码“500”，错误信息为“Internal server error”。
- 调用自定义认证的API，如果自定义认证开启宽松模式，当自定义认证函数连接失败、返回“500”或者“503”时，自定义认证不会校验body字段里面的status字段，直接返回调用成功，同时从函数代码中获取到的context字段也为空。
- “status”字段为必选，用于标识认证结果。只支持“allow”或“deny”，“allow”表示认证成功，“deny”表示认证失败。
- “context”字段为可选，支持字符串类型键值对，键值不支持JSON对象或数组。

context中的数据为您自定义的字段，认证通过后作为认证参数映射到API网关后端参数中，其中context中的参数名称与系统参数名称必须完全一致，且区分大小写。context中的参数名称必须以英文字母开头，支持英文大小写字母、数字、下划线和中划线，且长度为1 ~ 32个字符。

后端认证通过后，context中的user的值abc映射到后端服务Header位置的test参数中，并将其传递给API的后端服务。

用户数据定义代码示例：

```
# -*- coding:utf-8 -*-
import json
import base64
def handler(event, context):
    exampleuserdata=base64.b64encode(event["user_data"])
    resp = {
        'statusCode': 200,
        'body': json.dumps({
            "status":"allow",
            "context":{
                "user":exampleuserdata
            }
        })
    }
    return json.dumps(resp)
```

步骤2 测试函数。在测试事件的“事件模板”中选择“空白模板”，内容为：

```
{"user_data": "123"}
```

根据实际情况修改后保存测试模板，单击“测试”。

执行结果为“成功”时，表示测试成功。

接下来您需要进入API网关界面创建后端自定义认证。



----结束

后续操作

在自定义认证中已经创建完成用于后端自定义认证的Function API，下一步您需要进入API网关中创建后端自定义认证，具体操作请参见《API网关用户指南》的“创建自定义认证”章节。

6 对后端服务进行签名

6.1 签名前准备

获取签名密钥信息：

在APIG实例控制台选择“API管理 > API策略 > 策略管理”，单击签名密钥名称进入详情页面，在“策略内容”区域查看Key和Secret。

6.2 Java

操作场景

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

本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

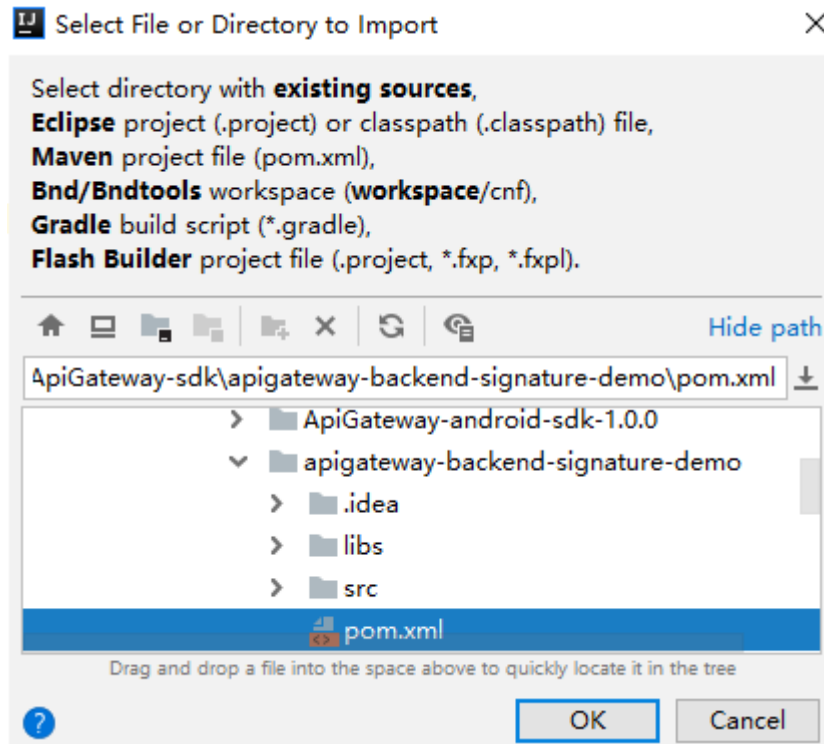
前提条件

- 已在API网关控制台创建签名密钥，并绑定API，具体请参见《API网关用户指南》的“签名密钥策略说明”章节。
- 已获取签名密钥的Key和Secret，请参见[签名前准备](#)获取。
- 进入API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。
- 已安装IntelliJ IDEA 2018.3.5或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。
- 已安装Java Development Kit 1.8.111或以上版本，如果未安装，请至[Oracle官方下载页面](#)下载。暂不支持Java Development Kit 17或以上版本。

导入工程

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

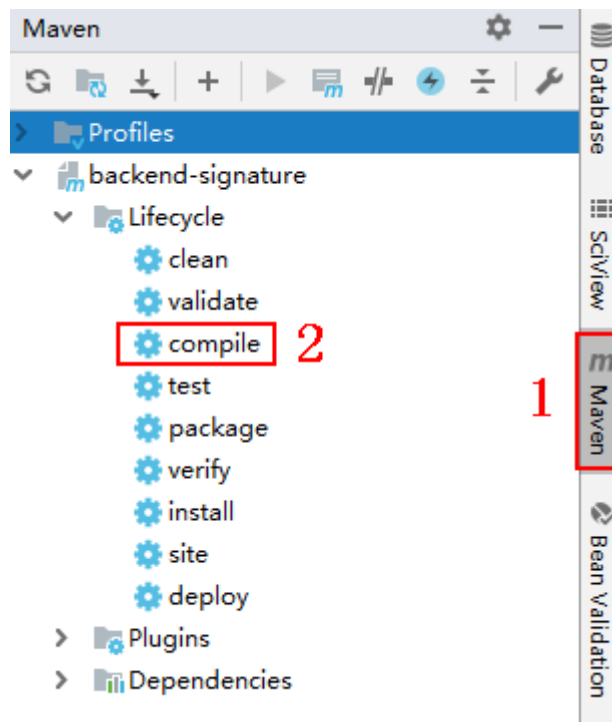
图 6-1 Select File or Directory to Import



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

步骤3 在右侧Maven页签，双击“compile”进行编译。

图 6-2 编译工程

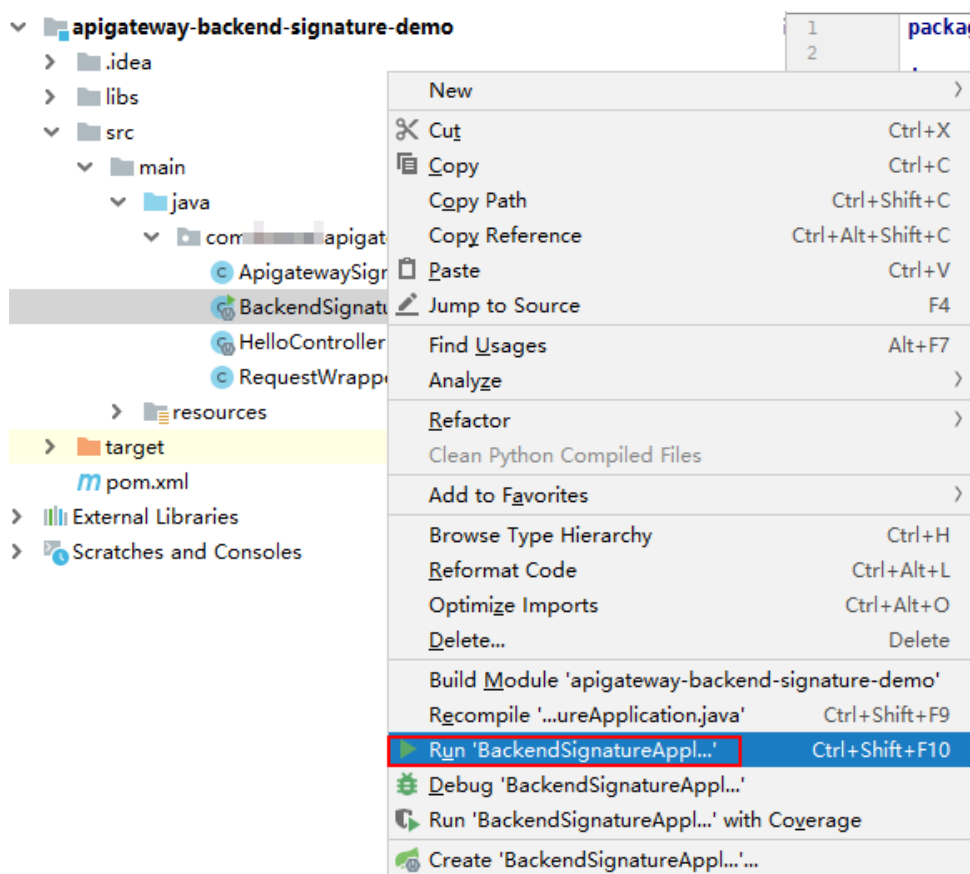


返回“BUILD SUCCESS”，表示编译成功。

```
Run: m backend-signature [compile] x
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ backend-signature ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ backend-signature ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 2.688 s
[INFO] Finished at: 2019-03-11T18:41:09+08:00
[INFO] Final Memory: 21M/309M
[INFO]
Process finished with exit code 0
```

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

图 6-3 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[校验后端签名示例](#)。

----结束

校验后端签名示例

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API网关（即API管理）的请求做签名校验。

📖 说明

API绑定签名密钥后，发给后端的请求中才会添加签名信息。

步骤1 编写一个Controller，匹配所有路径和方法，返回体为“Hello World!”。

```
// HelloController.java

@RestController
@EnableAutoConfiguration
public class HelloController {

    @RequestMapping("/")
    private String index() {
        return "Hello World!";
    }
}
```

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

```
// ApigatewaySignatureFilter.java

@Component
@WebFilter(filterName = "ApigatewaySignatureFilter", urlPatterns = "/*")
public class ApigatewaySignatureFilter implements Filter {
    private static Map<String, String> secrets = new HashMap<>();
    static {

        secrets.put(System.getenv("SDK_AK1"), System.getenv("SDK_SK1"));
        secrets.put(System.getenv("SDK_AK2"), System.getenv("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
        ...
    }
}
```

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

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

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

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

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

Matcher m = authorizationPattern.matcher(authorization);
if (!m.find()) {
```



```
response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Authorization format incorrect.");
return;
}
String signingKey = m.group(1);
String signingSecret = secrets.get(signingKey);
if (signingSecret == null) {
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Signing key not found.");
    return;
}
String[] signedHeaders = m.group(2).split(";");
```

例如，Authorization头为：

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

则解析的结果为：

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

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

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

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

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

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

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

```
    }  
    if (signedHeaders[i].toLowerCase().equals("x-sdk-date")) {  
        dateHeader = headerValue;  
    }  
}  
}  
  
if (needbody) {  
    apiRequest.setContent(new ByteArrayInputStream(request.getBody())); //set body  
}
```

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

```
private static final DateTimeFormatter timeFormatter =  
    DateTimeFormat.forPattern("yyyyMMdd'T'HHmmss'Z").withZoneUTC();  
  
...  
  
if (dateHeader == null) {  
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Header x-sdk-date not found.");  
    return;  
}  
long date = timeFormatter.parseMillis(dateHeader);  
long duration = Math.abs(DateTime.now().getMillis() - date);  
if (duration > 15 * 60 * 1000) {  
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Signature expired.");  
    return;  
}
```

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

```
apiRequest.addHeader("authorization", authorization);  
apiRequest.setKey(signingKey);  
apiRequest.setSecret(signingSecret);  
Signer signer = new Signer();  
boolean verify = signer.verify(apiRequest);  
if (verify) {  
    chain.doFilter(request, response);  
} else {  
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Verify authroization failed.");  
}
```

步骤9 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。

Apigateway Signature Test

Key

Secret

Method Scheme Host Url

Query

Headers

Body


```
curl -X POST "http://localhost:8080/test?xxx=yyy" -H "aaa: bbb" -H "X-Sdk-Date: 20190307T122402Z" -H "host: localhost:8080" -H "Authorization: SDK-HMAC-SHA256 Access=signatur
```

----结束

6.3 Python

操作场景

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

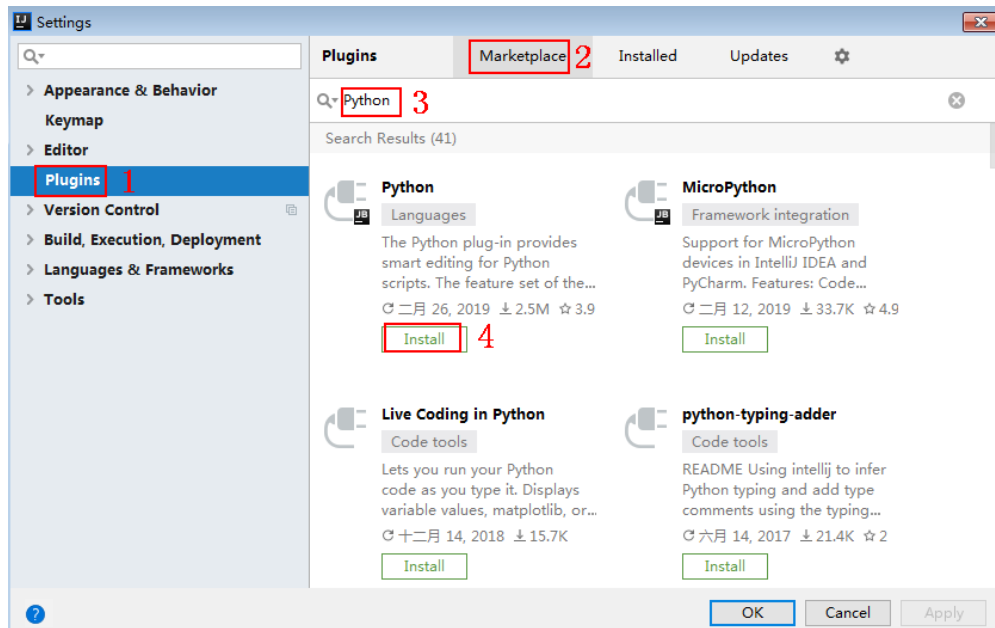
本章节以IntelliJ IDEA 2018.3.5版本为例介绍。

准备环境

- 已在API网关控制台创建签名密钥，并绑定API，具体请参见《API网关用户指南》的“签名密钥策略说明”章节。
- 已获取签名密钥的Key和Secret，请参见[签名前准备](#)获取。
- 请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。
- 已安装Python安装包2.7.9或3.X版本，如果未安装，请至[Python官方下载页面](#)下载。
- 已安装IntelliJ IDEA 2018.3.5或以上版本，如果未安装，请至[IntelliJ IDEA官方网站](#)下载。

- 已在IntelliJ IDEA中安装Python插件，如果未安装，请按照图6-4所示安装。

图 6-4 安装 Python 插件

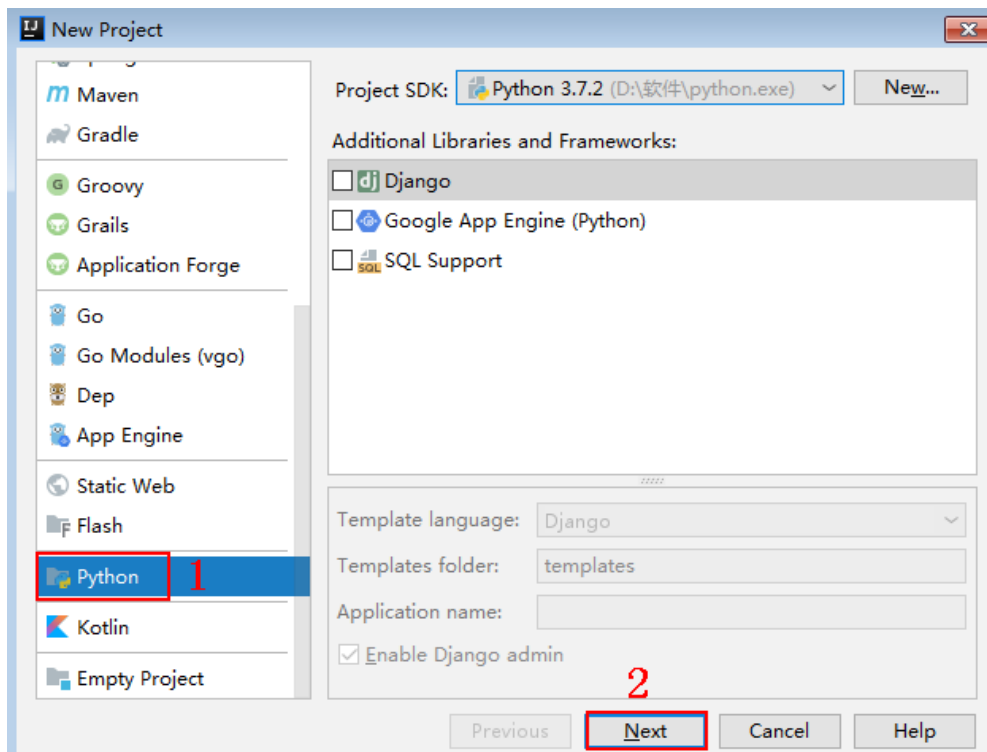


导入工程

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

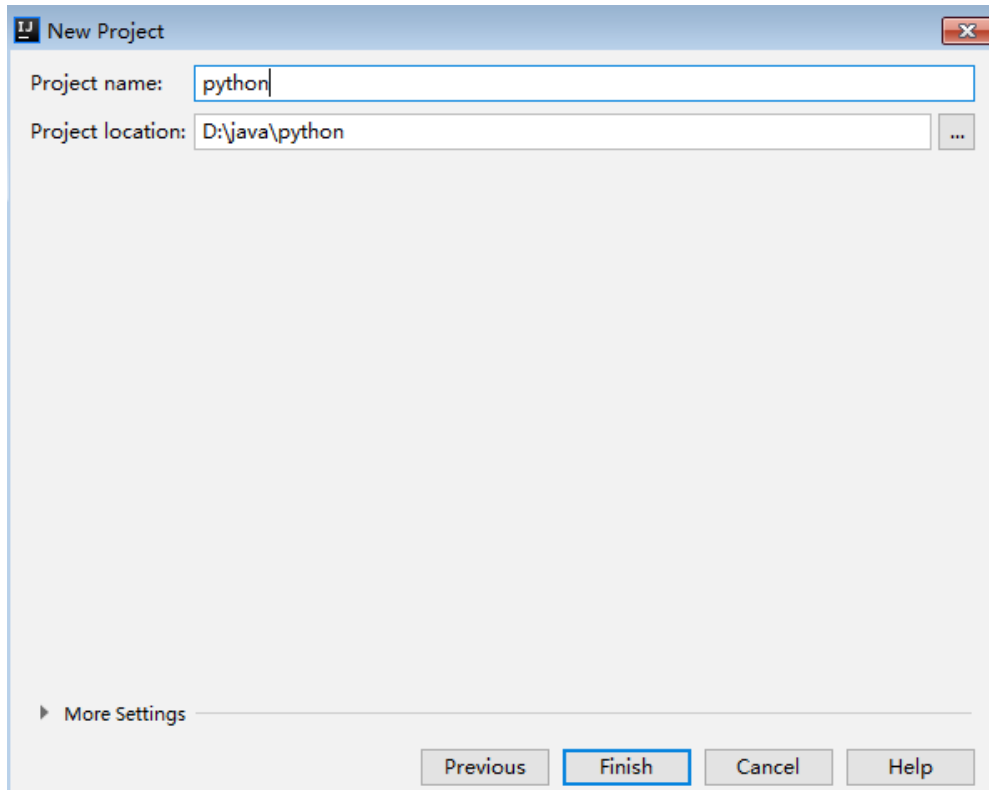
弹出“New Project”对话框，选择“Python”，单击“Next”。

图 6-5 New Python



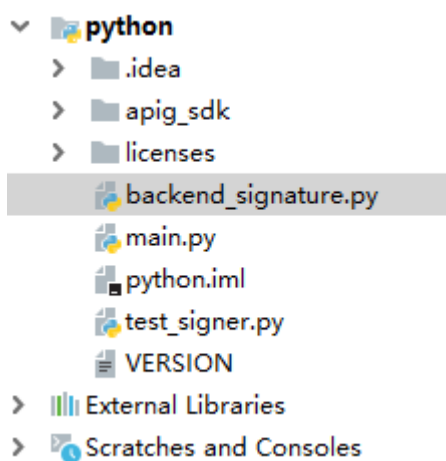
步骤2 再次单击“Next”，弹出以下对话框。单击“...”，在弹出的对话框中选择解压后的 SDK 路径，单击“Finish”。

图 6-6 选择解压后 python 的 SDK 路径



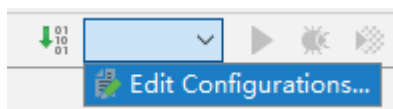
步骤3 完成工程创建后，目录结构如下。

图 6-7 目录结构



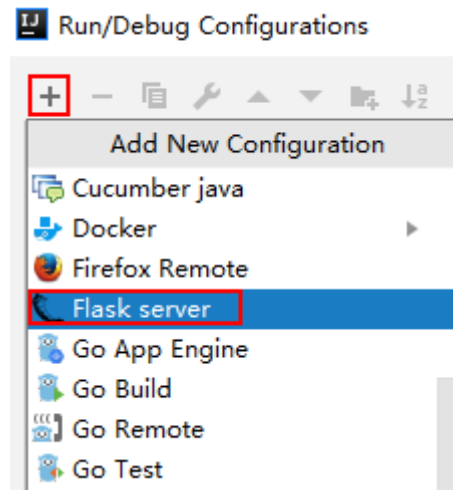
步骤4 单击“Edit Configurations”，弹出“Run/Debug Configurations”对话框。

图 6-8 Edit Configurations

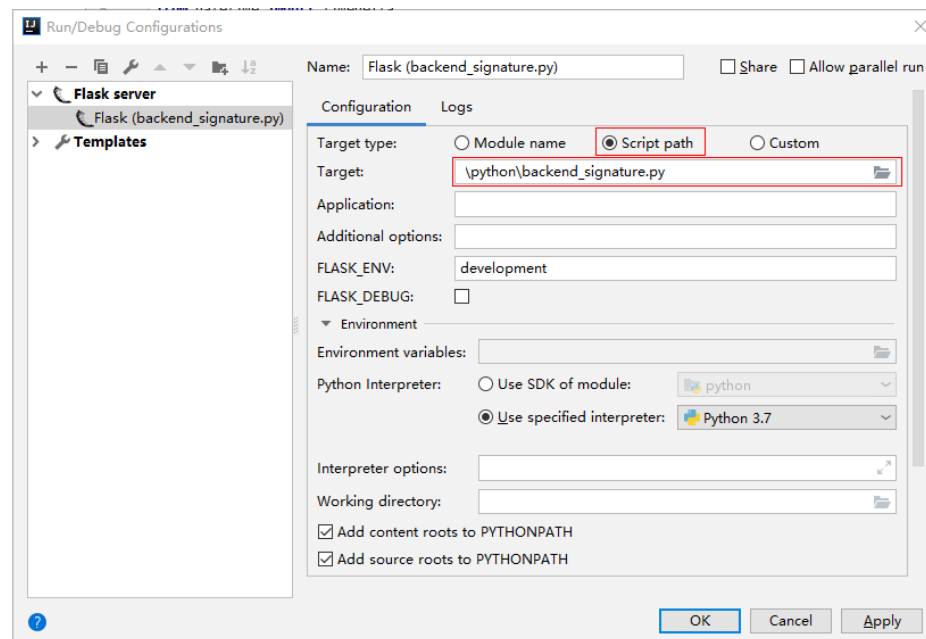


步骤5 单击“+”，选择“Flask server”。

图 6-9 选择 Flask server



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



----结束

校验后端签名示例

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API网关（即API管理）的请求做签名校验。

说明

API绑定签名密钥后，发给后端的请求中才会添加签名信息。

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

```
app = Flask(__name__)

@app.route("/<id>", methods=['GET', 'POST', 'PUT', 'DELETE'])
@requires_apigateway_signature()
def hello(id):
    return "Hello World!"
```

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

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

        secrets = {

            os.getenv('SDK_AK1'): os.getenv('SDK_SK1'),
            os.getenv('SDK_AK2'): os.getenv('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
```

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

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

例如，`Authorization`头为：

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

则解析的结果为：

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

步骤4 通过`key`找到`secret`，如果不存在`key`，则返回认证失败。

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

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

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

```
r = signer.HttpRequest()
r.method = request.method
r.uri = request.path
r.query = {}
for k in request.query_string.decode('utf-8').split('&'):
```

```
spl = k.split("=", 1)
if len(spl) < 2:
    r.query[spl[0]] = ""
else:
    r.query[spl[0]] = spl[1]
r.headers = {}
needbody = True
dateHeader = None
for k in signedHeaders:
    if k not in request.headers:
        return 'Signed header ' + k + ' not found', 401
    v = request.headers[k]
    if k.lower() == 'x-sdk-content-sha256' and v == 'UNSIGNED-PAYLOAD':
        needbody = False
    if k.lower() == 'x-sdk-date':
        dateHeader = v
    r.headers[k] = v
if needbody:
    r.body = request.get_data()
```

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

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

步骤7 调用verify方法校验请求签名。判断校验是否通过。

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

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

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

如果使用错误的Key和Secret访问，服务器返回401认证不通过。

Apigateway Signature Test

Key

Secret

Method Scheme Host Url

Query

Headers

Body


```
curl -X POST "http://localhost:8080/test?xxx=yyy" -H "aaa: bbb" -H "X-Sdk-Date: 20190307T122402Z" -H "host: localhost:8080" -H "Authorization: SDK-HMAC-SHA256 Access=signatur
```

----结束

6.4 C#

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。

准备环境

- 已获取签名密钥的Key和Secret，请参见[签名前准备](#)获取。
- 已在API网关控制台创建签名密钥，并绑定API，具体请参见《API网关用户指南》的“签名密钥策略说明”章节。
- 请登录API网关控制台，参考《API网关用户指南》的“SDK”章节，进入SDK页面并下载SDK。
- 已安装Visual Studio 2019 version 16.8.4或以上版本，如果未安装，请至[Visual Studio官方网站](#)下载。

打开工程

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

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

校验后端签名示例

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 API 网关 (即 API 管理) 的请求做签名校验。

📖 说明

API 绑定签名密钥后, 发给后端的请求中才会添加签名信息。

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

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

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

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

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

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

```
// ApigatewaySignatureFilter.cs
namespace backend_signature.Filters
{
    public class ApigatewaySignatureFilter : Attribute, IAuthorizationFilter
    {
        private Dictionary<string, string> secrets = new Dictionary<string, string>
        {
        }
    }
}
```

```
        {"signature_key1", "signature_secret1" },
        {"signature_key2", "signature_secret2" },

        {Environment.GetEnvironmentVariable(" SDK_AK1"),
Environment.GetEnvironmentVariable(" SDK_SK1" )},
        {Environment.GetEnvironmentVariable(" SDK_AK2"),
Environment.GetEnvironmentVariable(" SDK_SK2" )},
    };

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

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

```
private Regex authorizationPattern = new Regex("SDK-HMAC-SHA256\\s+Access=([^,]+),\\s?SignedHeaders=([^,]+),\\s?Signature=(\\w+)");
...
string authorization = request.Headers["Authorization"];
if (authorization == null)
{
    context.Result = new UnauthorizedResult();
    return;
}
var matches = authorizationPattern.Matches(authorization);
if (matches.Count == 0)
{
    context.Result = new UnauthorizedResult();
    return;
}
var groups = matches[0].Groups;
string key = groups[1].Value;
string[] signedHeaders = groups[2].Value.Split(';');
```

例如，Authorization头为：

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

则解析的结果为：

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

步骤4 通过key找到secret，如果不存在key，则返回认证失败。

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

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

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

```
HttpRequest sdkRequest = new HttpRequest();
sdkRequest.method = request.Method;
sdkRequest.host = request.Host.Value;
sdkRequest.uri = request.Path;
```

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

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

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

步骤7 调用verify方法校验请求签名。判断校验是否通过。

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

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

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

如果使用错误的Key和Secret访问，服务器返回401认证不通过。

Apigateway Signature Test

Key

Secret

Method Scheme Host Url

Query

Headers

Body


```
curl -X POST "http://localhost:8080/test?xxx=yyy" -H "aaa: bbb" -H "X-Sdk-Date: 20190307T122402Z" -H "host: localhost:8080" -H "Authorization: SDK-HMAC-SHA256 Access=signatur
```

----结束

7 修订记录

表 7-1 文档修订记录

发布日期	修订记录
2024-10-09	本次变更： <ul style="list-style-type: none">• 下线“概述”章节。• 下线“导入导出API”章节，如需参考导入导出相关资料请参考《API网关用户指南》中的“导入导出API”章节。• 新增签名前准备章节。• “如何选择认证方式”章节更名使用场景并优化。
2023-05-30	本次变更： Java 章节中的前提条件更新。
2023-04-30	本次变更： 认证前准备 适配新版UI内容。
2022-12-30	本次变更： Java 章节更新JAVA SDK包获取地址及相关内容。
2020-11-10	第一次正式发布。