ALIBABA CLOUD

阿里云

API 网关

API安全

文档版本：20220712

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

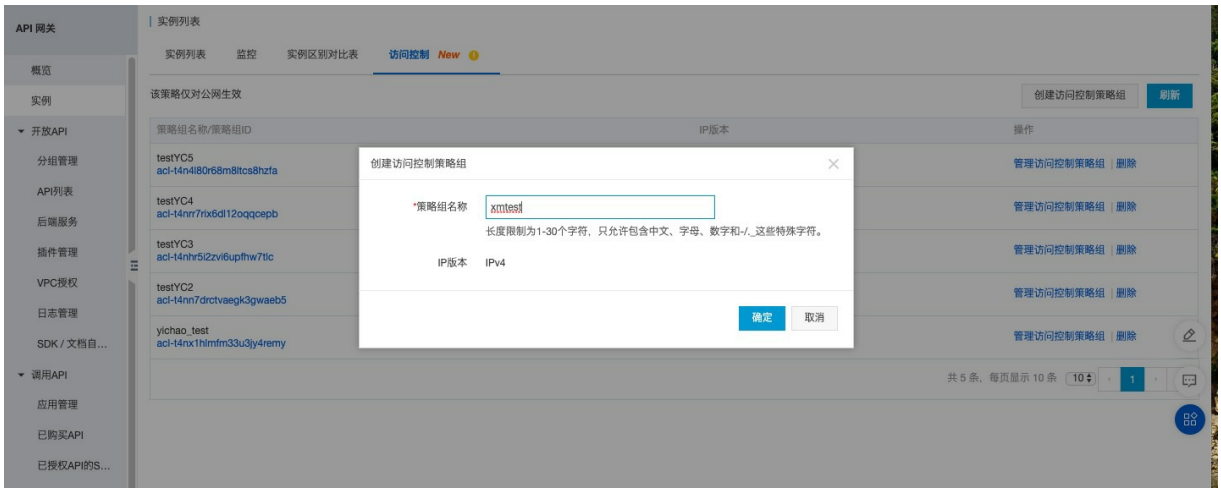
目录

1.实例级别访问控制

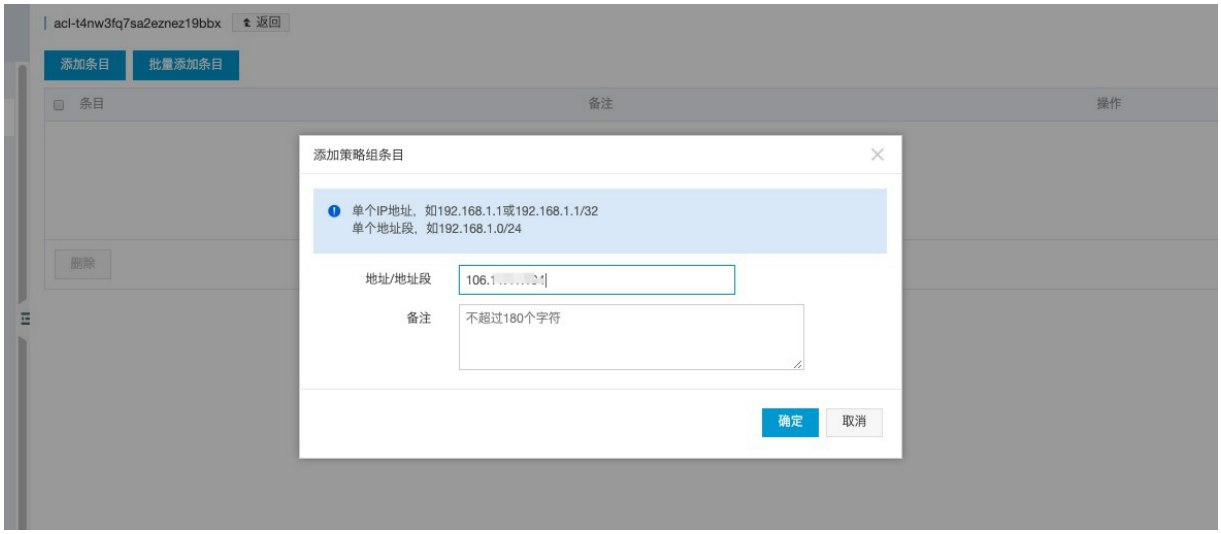
API网关提供实例级别的访问控制，您可以配置访问控制策略组应用于实例，仅专享实例支持配置实例级别访问控制策略，并且仅对公网生效。

1. 创建访问控制策略组

1.1 在API网关控制台中实例页面，单击访问控制，即进入访问控制策略组的创建和管理页面，单击 `创建访问控制策略组` 按钮，输入策略组名称，点击确认。



1.2 访问控制策略组创建成功后，点击管理访问控制策略组按钮，即可添加条目，目前支持单个添加条目以及批量添加条目，示例如下：



> **说明**
>
> - 每个region只能创建5个访问控制策略组。
> - 每个实例只能绑定一个访问控制策略组。
> - 批量添加策略组条目时最多可添加50个条目
> - 如果访问控制策略组的条目为空，黑白名单无法生效。

2. 专享实例设置黑白名单

在API网关控制台中实例页面，找到专享实例，点击设置黑白名单即可设置访问控制策略。选择黑名单或者是白名单，然后选择我们配置的策略组。阅读注意事项，没问题后点击确认，访问控制策略就生效了。

> **注意**
>
> 配置黑白名单后，实例下的所有API分组都会受到所选的访问控制策略的限制，请谨慎操作。

3. 常见问题

1、若配置了访问控制白名单，那么不在白名单中的客户端访问API网关会怎么样呢？

网关的接入层会直接拒绝访问，客户端请求时会有超时的相关报错。

> **注意**
>
> 如果需要使用API网关的调试功能，需要把调试页的ip添加至白名单，调试页的ip可以提交工单获取。

2、实例级别访问控制和ip访问控制插件有什么区别？

ip访问控制插件针对API级别进行访问控制。实例级别访问控制可以针对整个API网关专享实例进行访问控制，并且不会产生流量费。

2.支持 HTTPS

HTTPS在HTTP的基础上加入了SSL协议，对信息、数据加密，用来保证数据传输的安全。现如今被广泛使用。

API网关也支持使用HTTPS对您的API请求进行加密。可以控制到API级别，即您可以强制您的API只支持HTTP、HTTPS或者两者均支持。

如果您的API需要支持HTTPS，操作流程如下：

步骤1：准备

您需要准备如下材料：

- 一个自有可控域名。
- 为这个域名申请一个SSL证书
- 自定义上传证书，包含证书/私钥，均为 PEM 格式（注：API网关Tengine服务是基于Nginx，因此只支持Nginx能读取的证书，即PEM格式）。

SSL证书会包含两部分内容：XXXXX.key、XXXXX.pem，可以使用文本编辑器打开。示例如下：

KEY：

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA8GjIleJ7rlo86mtbwcDnUfqzTQAm4b3zZEo1aKsfAuwcvCud
....
-----END RSA PRIVATE KEY-----
```

PEM：

```
-----BEGIN CERTIFICATE-----
MIIFtDCCBJygAwIBAgIQRgWF1j00cozRl1pZ+ultKTANBgkqhkiG9w0BAQsFADBP
...
-----END CERTIFICATE-----
```

步骤2：绑定SSL证书

准备好以上材料，需要进行如下操作进行，登录API网关管理控制台【开放API】-【分组管理】，单击您需要绑定SSL证书的分组，查看分组详情。

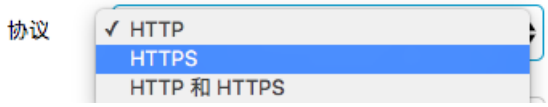
在绑定SSL证书，您首先需要您在API分组上绑定【独立域名】。

【独立域名】-添加SSL证书。

- 证书名称：用户自定义名称，以供后续识别。
- 证书内容：证书的完整内容，需要复制XXXXX.pem中的全部内容。
- 私钥：证书的私钥，需要复制XXXXX.key中的内容。点击【确定】后，完成SSL证书的绑定。

步骤3：API配置调整

绑定SSL证书后，您可以按API控制不同的访问方式，支持HTTP、HTTPS、HTTP和HTTPS三种，出于安全考虑，建议全部配置成HTTPS。



可以在【开放API】-【API列表】找到相应API，【API定义】-编辑-【请求基础定义】中进行修改。

API支持的协议包括：

- HTTP：只允许HTTP访问，不允许HTTPS
- HTTPS：只允许HTTPS访问，不允许HTTP
- HTTP和HTTPS：两者均可，调整后，API支持HTTPS协议配置完成。您的API将支持HTTPS访问。

3.HTTPS安全策略

分组HttpsPolicy的设置

用户可以再分组上调整API分组所支持的HTTPS安全策略，HTTPS安全策略仅对绑定了域名及证书的分组有效，目前API网关支持 `HTTPS1_1_TLS1_0` ， `HTTPS2_TLS1_0` ， `HTTPS2_TLS1_2` 安全策略，但不同Region支持的安全策略列表不同，在**控制台->分组详情页**可选择本Region支持的HTTPS安全策略。

HTTPS安全策略列表

HTTPS1_1_TLS1_0

- HTTP1.1协议。
- 支持TLS v1.0 , TLS v1.1 , TLS v1.2。
- 支持加密算法套件：ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-SHA:AES128-GCM-SHA256:AES128-SHA256:AES128-SHA:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:AES256-GCM-SHA384:AES256-SHA256:AES256-SHA:ECDHE-RSA-AES128-SHA256:!aNULL:!eNULL:!RC4:!EXPORT:!DES:!3DES:!MD5:!DSS:!PKS;

HTTPS2_TLS1_0

- HTTP2协议，**注意：http2协议会将所有的header转为小写。**
- 支持TLS v1.0 , TLS v1.1 , TLS v1.2。
- 支持加密算法套件：ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-SHA:AES128-GCM-SHA256:AES128-SHA256:AES128-SHA:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:AES256-GCM-SHA384:AES256-SHA256:AES256-SHA:ECDHE-RSA-AES128-SHA256:!aNULL:!eNULL:!RC4:!EXPORT:!DES:!3DES:!MD5:!DSS:!PKS;

HTTPS2_TLS1_2

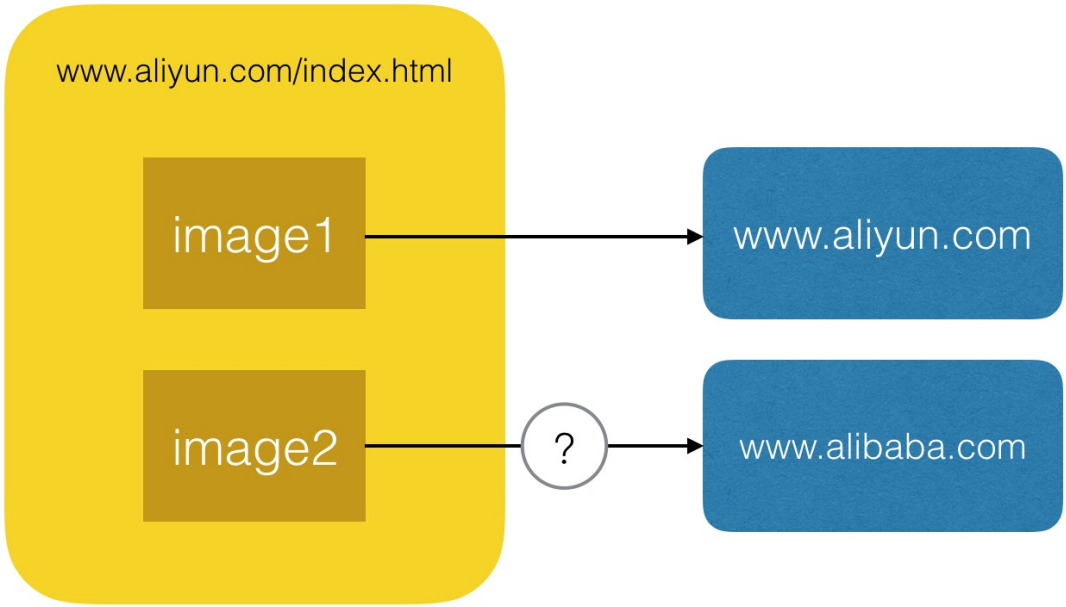
- HTTP2协议，**注意：http2协议会将所有的header转为小写。**
- 支持TLS v1.2，**注意：所有不支持TLS v1.2客户端将无法建立连接。**
- 支持加密算法套件：ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-RSA-AES256-SHA:!NULL:!aNULL:!MD5:!ADH:!RC4:!DH:!DHE:!3DES;

4.API网关实现跨域资源共享（CORS）

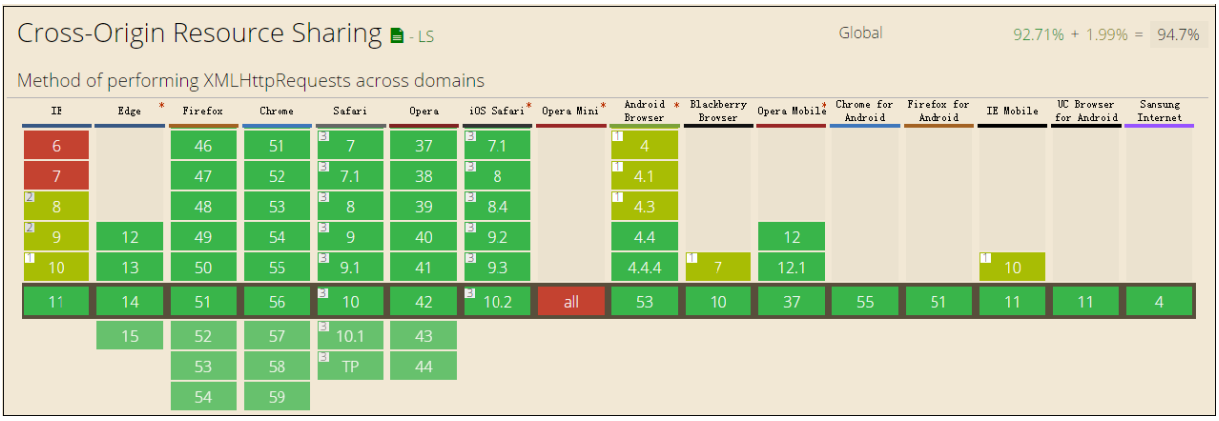
一、跨域带来的安全问题及浏览器的限制访问

当一个资源从与该资源本身所在的服务器不同的域或端口请求一个资源时，资源会发起一个跨域 HTTP 请求。比如，站点 http://www.aliyun.com 的某 HTML 页面通过img的src请求http://www.alibaba.com/image.jpg。网络上的许多页面都会加载来自不同域的CSS样式表，图像和脚本等资源。

出于安全原因，浏览器限制从页面脚本内发起的跨域请求，有些浏览器不会限制跨域请求的发起，但是会将结果拦截了。 这意味着使用这些API的Web应用程序只能加载同一个域下的资源，除非使用CORS机制（Cross-Origin Resource Sharing 跨源资源共享）获取目标服务器的授权来解决这个问题。



上图画的是典型的跨域场景，目前主流浏览器为了用户的安全，都会默认禁止跨域访问，但是主流浏览器都支持W3C推荐的一种跨域资源共享机制（CORS）。服务器端配合浏览器实现CORS机制，可以突破浏览器对跨域资源访问的限制，实现跨域资源请求。



二、跨域资源共享CORS介绍

2.1 两种验证模式

跨域资源共享CORS的验证机制分两种模式：简单请求和预先请求。

当请求**同时满足下面三个条件时**，CORS验证机制会使用简单模式进行处理。

1.请求方法是下列之一：

- GET
- HEAD
- POST

2.请求头中的Content-Type请求头的值是下列之一：

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

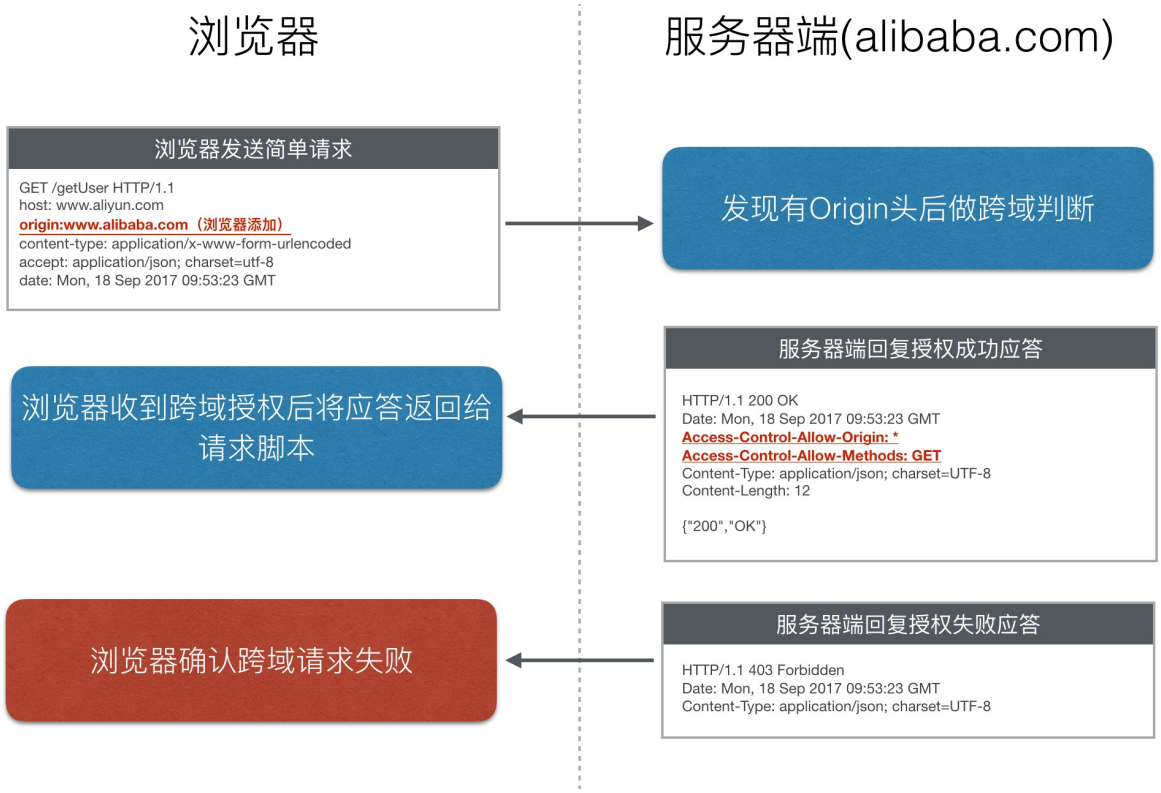
3.Fetch规范定义了CORS安全头的集合（跨域请求中自定义的头属于安全头的集合）该集合为：

- Accept
- Accept-Language
- Content-Language
- Content-Type （需要注意额外的限制）
- DPR
- Downlink
- Save-Data
- Viewport-Width
- Width

否则CORS验证机制会使用预先请求模式进行处理。

2.2 简单请求模式

简单请求模式，浏览器直接发送跨域请求，并在请求头中携带Origin的头，表明这是一个跨域的请求。 服务器端接到请求后，会根据自己的跨域规则，通过Access-Control-Allow-Origin和Access-Control-Allow-Methods响应头，来返回验证结果。



应答中携带了跨域头 Access-Control-Allow-Origin。使用 Origin 和 Access-Control-Allow-Origin 就能完成最简单的访问控制。本例中，服务端返回的 Access-Control-Allow-Origin: * 表明，该资源可以被任意外域访问。如果服务端仅允许来自 http://www.aliyun.com 的访问，该首部字段的内容如下：

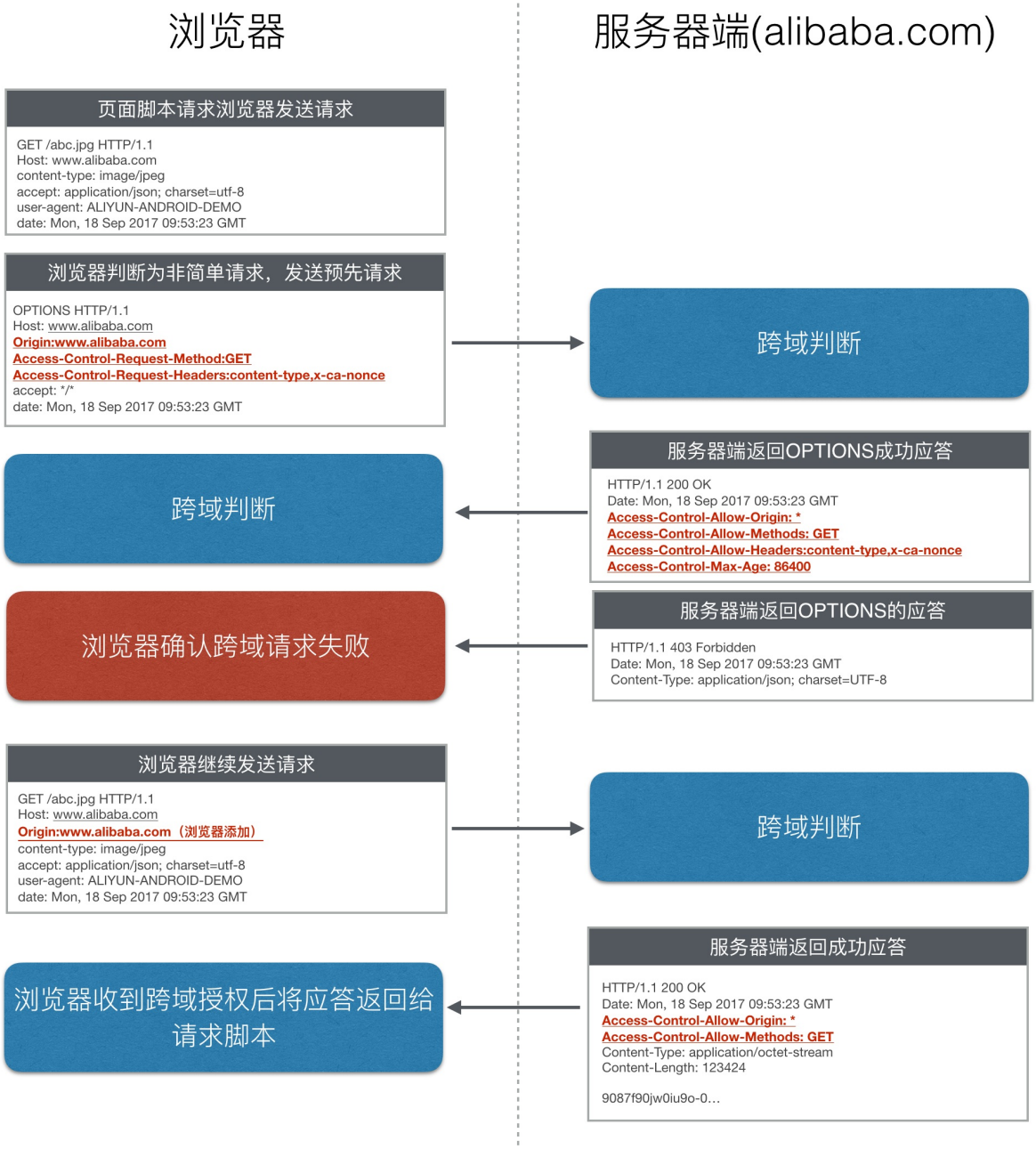
```
Access-Control-Allow-Origin: http://www.aliyun.com
```

现在，除了 http://www.aliyun.com，其它外域均不能访问该资源。

2.3 预先请求模式

浏览器在发现页面发出的请求非简单请求，并不会立即执行对应的请求代码，而是会触发预先请求模式。预先请求模式会先发送Preflighted requests（预先验证请求），Preflighted requests是一个OPTION请求，用于询问要被跨域访问的服务器，是否允许当前域名下的页面发送跨域的请求。在得到服务器的跨域授权后才能发送真正的HTTP请求。

OPTIONS请求头部中会包含以下头部：Origin、Access-Control-Request-Method、Access-Control-Request-Headers。 服务器收到OPTIONS请求后，设置Access-Control-Allow-Origin、Access-Control-Allow-Method、Access-Control-Allow-Headers、Access-Control-Max-Age头部与浏览器沟通来判断是否允许这个请求。 如果Preflighted requests验证通过，浏览器才会发送真正的跨域请求。



请求中的跨域头 Access-Control-Request-Method 告知服务器，实际请求将使用 GET 方法。 请求中的跨域头 Access-Control-Request-Headers 告知服务器，实际请求将携带两个自定义请求首部字段：x-ca-nonce 与 content-type。服务器据此决定该实际请求是否被允许。

应答中的跨域头 Access-Control-Allow-Methods 表明服务器允许客户端使用 GET 方法发起请求。值为逗号分割的列表。

应答中的跨域头 Access-Control-Allow-Headers 表明服务器允许请求中携带字段 x-ca-nonce 与 content-type。与 Access-Control-Allow-Methods 一样，Access-Control-Allow-Headers 的值为逗号分割的列表。

应答中的跨域头 Access-Control-Max-Age 表明该响应的有效时间为 86400 秒，也就是 24 小时。在有效时间内，浏览器无须为同一请求再次发起预检请求。请注意，浏览器自身维护了一个最大有效时间，如果该首部字段的值超过了最大有效时间，将不会生效。

三、在API网关实现CORS跨域资源共享

3.1实现简单请求模式

API网关默认所有API允许跨域访问，因此如果用户的API后端服务的应答中不做特殊返回，API网关会返回允许所有域跨域访问的相关头，下面是一个示例：

客户端的API请求

```
GET /simple HTTP/1.1
Host: www.alibaba.com
orgin: http://www.aliyun.com
content-type: application/x-www-form-urlencoded; charset=utf-8
accept: application/json; charset=utf-8
date: Mon, 18 Sep 2017 09:53:23 GMT
```

后端服务应答

```
HTTP/1.1 200 OK
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

API网关应答

```
HTTP/1.1 200 OK
Date: Mon, 18 Sep 2017 09:53:23 GMT
Access-Control-Allow-Origin: *
X-Ca-Request-Id: 104735BD-8968-458F-9929-DBFA43F324C6
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

从上面三个报文可以看出，API网关会对用户的后端服务应答做一定修改，增加一个跨域头：

```
Access-Control-Allow-Origin: *
```

这个跨域头的意思是，本API允许所有域的请求访问。

如果用户需要定制针对简单请求的应答的跨域头，只需要在后端服务应答中，增加Access-Control-Allow-Origin这个跨域头即可，后端服务应答中的头会默认覆盖掉API网关自己增加的头。下面是一个例子，这个例子中的API只允许http://www.aliyun.com 这一个域访问：

客户端的API请求

```
GET /simple HTTP/1.1
Host: www.alibaba.com
orgin: http://www.aliyun.com
content-type: application/x-www-form-urlencoded; charset=utf-8
accept: application/json; charset=utf-8
date: Mon, 18 Sep 2017 09:53:23 GMT
```

后端服务应答

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.aliyun.com
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

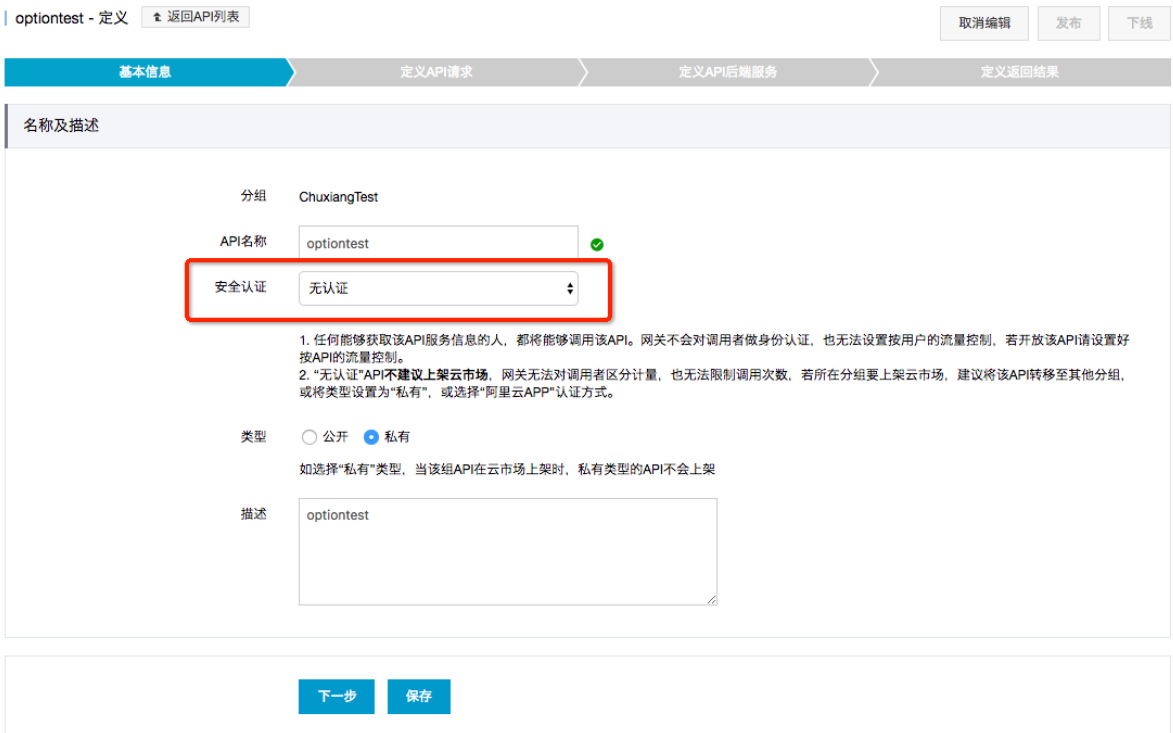
API网关应答

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.aliyun.com
X-Ca-Request-Id: 104735BD-8968-458F-9929-DBFA43F324C6
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

3.2 实现预先请求模式

API网关允许用户设置方法为OPTIONS的API，并且将后端服务的OPTIONS应答透传给客户端。新建方法为OPTIONS的API，定义的其他部分与正常API一样，有两点需要注意：

- 定义API认证方式时选择无认证；



- 定义API请求时，需要设置path为/，并且匹配所有子路径。选定方法为OPTIONS，API网关控制台会默认设置请求模式为透传模式，且不可修改，用户不需要定义请求参数；

用户可以在每个API分组下建立一个方法的OPTIONS的API，来定义这一组API绑定的域名的跨域资源策略。用户可以是用CURL方法来测试自己的跨域API应答情况，下面是针对一个定义好的OPTIONS的API访问的一个示例：

```
sudo curl -X OPTIONS -H "Access-Control-Request-Method:POST" -H "Access-Control-Request-Hea
ders:X-CUSTOM-HEADER" http://ec12ac094e734544be02c928366b7b26-cn-qingdao.alicloudapi.com/op
tinstest -i
HTTP/1.1 200 OK
Server: Tengine
Date: Sun, 02 Sep 2018 15:32:19 GMT
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,POST,PUT,DELETE,HEAD,OPTIONS,PATCH
Access-Control-Allow-Headers: X-CUSTOM-HEADER
Access-Control-Max-Age: 172800
X-Ca-Request-Id: 1016AC86-E345-405C-8049-A6C24078F65F
```

用户在实现方法为OPTIONS的API的时候需要注意的一点是：**API网关会对用户的后端服务应答做一定修改，增加四个跨域头（Access-Control-Allow-Origin、Access-Control-Allow-Methods、Access-Control-Allow-Headers、Access-Control-Max-Age），后端服务应答中，需要返回所有跨域头来覆盖API网关默认跨域头。**

下面是一个完整的预先请求模式的请求与应答示例。

客户端的方法为OPTIONS的API请求

```
OPTIONS /simple HTTP/1.1
Host: www.alibaba.com
orgin: http://www.aliyun.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
accept: application/json; charset=utf-8
date: Mon, 18 Sep 2017 09:53:23 GMT
```

后端服务应答

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.aliyun.com
Access-Control-Allow-Methods: GET,POST
Access-Control-Allow-Headers: X-CUSTOM-HEADER
Access-Control-Max-Age: 10000
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
```

API网关应答

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.aliyun.com
Access-Control-Allow-Methods: GET,POST
Access-Control-Allow-Headers: X-CUSTOM-HEADER
Access-Control-Max-Age: 10000
X-Ca-Request-Id: 104735BD-8968-458F-9929-DBFA43F324C6
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
```

客户端发送正常业务请求

```
GET /simple HTTP/1.1
Host: www.alibaba.com
orgin: http://www.aliyun.com
content-type: application/x-www-form-urlencoded; charset=utf-8
accept: application/json; charset=utf-8
date: Mon, 18 Sep 2017 09:53:23 GMT
```

后端服务应答

```
HTTP/1.1 200 OK
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

API网关应答

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,POST,PUT,DELETE,HEAD,OPTIONS,PATCH
Access-Control-Allow-Headers: X-Requested-With,X-Sequence,X-Ca-Key,X-Ca-Secret,X-Ca-Version
,X-Ca-Timestamp,X-Ca-Nonce,X-Ca-API-Key,X-Ca-Stage,X-Ca-Client-DeviceId,X-Ca-Client-AppId,X
-Ca-Signature,X-Ca-Signature-Headers,X-Forwarded-For,X-Ca-Date,X-Ca-Request-Mode,Authorizat
ion,Content-Type,Accept,Accept-Ranges,Cache-Control,Range,Content-MD5
Access-Control-Max-Age: 172800
X-Ca-Request-Id: 104735BD-8968-458F-9929-DBFA43F324C6
Date: Mon, 18 Sep 2017 09:53:23 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 12
{"200","OK"}
```

5.基于JWT的token认证

阿里云API网关在Json Web Token（JWT）这种结构化令牌的基础上实现了一套基于用户体系对用户的API进行授权访问的机制，满足用户个性化安全设置的需求。

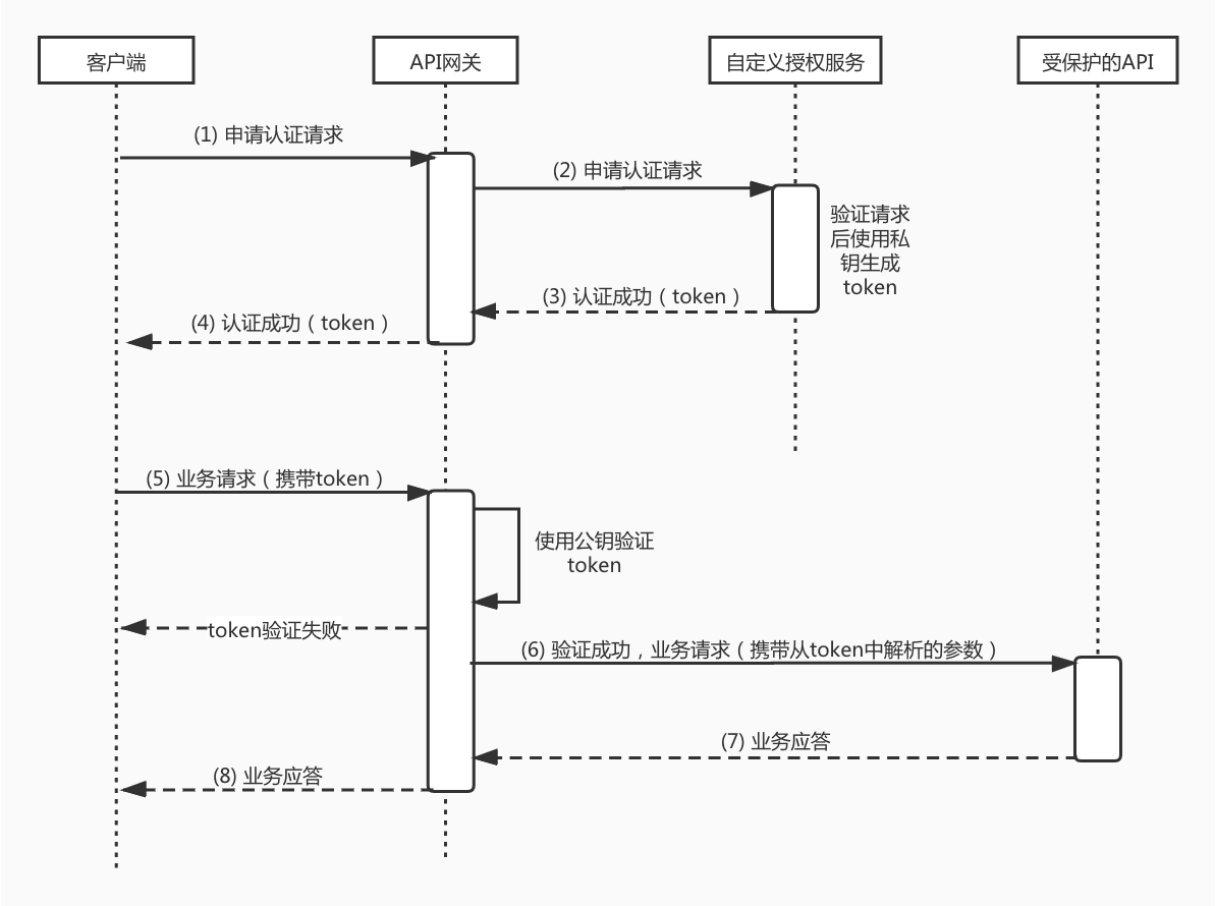
阿里云API网关在Json Web Token（JWT）这种结构化令牌的基础上实现了一套基于用户体系对用户的API进行授权访问的机制，满足用户个性化安全设置的需求。

一、基于token的认证

1.1 简介

很多对外开放的API需要识别请求者的身份，并据此判断所请求的资源是否可以返回给请求者。token就是一种用于身份验证的机制，基于这种机制，应用不需要在服务端保留用户的认证信息或者会话信息，可实现无状态、分布式的Web应用授权，为应用的扩展提供了便利。

1.2 流程描述



上图是API网关利用JWT实现认证的整个业务流程时序图，下面我们用文字来详细描述图中标注的步骤：

1. 客户端向API网关发起认证请求，请求中一般会携带终端用户的用户名和密码；
2. API网关将请求直接转发给后端服务；
3. 后端服务读取请求中的验证信息（比如用户名、密码）进行验证，验证通过后使用私钥生成标准的token，返回给API网关；
4. API网关将携带token的应答返回给客户端，客户端需要将这个token缓存到本地；

5. 客户端向API网关发送业务请求，请求中携带token；
6. API网关使用用户设定的公钥对请求中的token进行验证，验证通过后，将请求透传给后端服务；
7. 后端服务进行业务处理后应答；
8. API网关将业务应答返回给客户端。

在这个整个过程中，API网关利用token认证机制，实现了用户使用自己的用户体系对自己API进行授权的能力。下面我们就要介绍API网关实现token认证所使用的结构化令牌Json Web Toke(JWT)。

1.3 JWT

1.3.1 简介

Json Web Toke（JWT），是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准RFC7519。JWT一般可以用作独立的身份验证令牌，可以包含用户标识、用户角色和权限等信息，以便于从资源服务器获取资源，也可以增加一些额外的其它业务逻辑所必须的声明信息，特别适用于分布式站点的登录场景。

1.3.2 JWT的构成

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

如上面的例子所示，JWT就是一个字符串，由三部分构成：

- Header（头部）
- Payload（数据）
- Signature（签名）

Header

JWT的头部承载两个信息：

- 声明类型，这里是JWT
- 声明加密的算法

完整的头部就像下面这样的JSON：

```
{
  'typ': 'JWT',
  'alg': 'HS256'
}
```

然后将头部进行Base64编码（该编码是可以对称解码的），构成了第一部分。

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

Payload

载荷就是存放有效信息的地方。定义细节如下：

iss：令牌颁发者。表示该令牌由谁创建，该声明是一个字符串
sub： Subject Identifier，iss提供的终端用户的标识，在iss范围内唯一，最长为255个ASCII个字符，区分大小写
aud：Audience(s)，令牌的受众，分大小写的字符串数组
exp：Expiration time，令牌的过期时间戳。超过此时间的token会作废， 该声明是一个整数，是1970年1月1日以来的秒数
iat：令牌的颁发时间，该声明是一个整数，是1970年1月1日以来的秒数
jti：令牌的唯一标识，该声明的值在令牌颁发者创建的每一个令牌中都是唯一的，为了防止冲突，它通常是一个密码学随机值。这个值相当于向结构化令牌中加入了一个攻击者无法获得的随机熵组件，有利于防止令牌猜测攻击和重放攻击。

也可以新增用户系统需要使用的自定义字段，比如下面的例子添加了 `name` 用户昵称：

```
{
  "sub": "1234567890",
  "name": "John Doe"
}
```

然后将其进行Base64编码，得到Jwt的第二部分：

```
JTdCJTBBJTIwJTIwJTIyc3ViJTIyJTNBJTIwJTIyMTIzNDU2Nzg5MCUyMiUyQyUwQSUyMCUyMCUyMm5hbWUlMjIlM0ElMjAlMjJKb2huJTIwRG9lJTIyJTBBJTdE
```

Signature

这个部分需要Base64编码后的Header和Base64编码后的Payload使用 `.` 连接组成的字符串，然后通过Header中声明的加密方式进行加密（ `$secret` 表示用户的私钥），然后就构成了jwt的第三部分。

```
// javascript
var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);
var signature = HMACSHA256(encodedString, '$secret');
```

将这三部分用 `.` 连接成一个完整的字符串，就构成了 1.3.2 节最开始的JWT示例。

1.3.3 授权范围与时效

API网关会认为用户颁发的token有权利访问整个分组下的所有绑定JWT插件的API。如果需要更细力度的权限管理，还需要后端服务自己解开token进行权限认证。API网关会验证token中的exp字段，一旦这个字段过期了，API网关会认为这个token无效而将请求直接打回。过期时间这个值必须设置，并且过期时间一定要小于7天。

1.3.4 JWT的几个特点

1. JWT 默认是不加密，不能将秘密数据写入 JWT。
2. JWT 不仅可以用于认证，也可以用于交换信息。有效使用 JWT，可以降低服务器查询数据库的次数。JWT的安全特3. JWT 的最大缺点是，由于服务器不保存 session 状态，因此无法在使用过程中废止某个 token，或者更改 token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑。

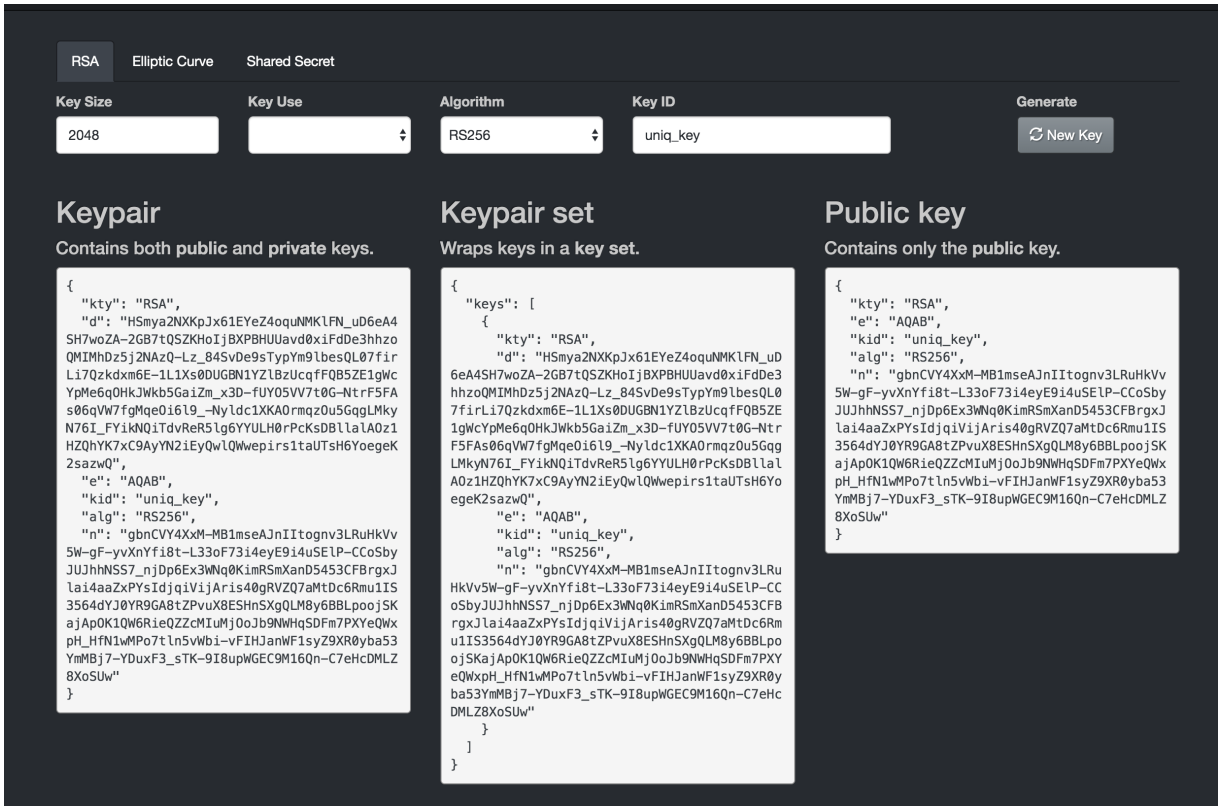
3. JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，JWT 的有效期应该设置得比较短。对于一些比较重要的权限，使用时应该再次对用户进行认证。
4. 为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用HTTPS 协议传输。

二、用户系统如何应用JWT插件保护API

2.1 生成一对JWK（JSON Web 密钥）

方法一、在线生成：

用户可以在这个站点https://mkjwk.org 生成用于token生成与验证的私钥与公钥， 私钥用于授权服务签发JWT，公钥配置到JWT插件中用于API网关对请求验签，目前API网关支持的密钥对的加密算法为RSA SHA256，密钥对的加密的位数为2048。



方法二、本地生成：

本文应用Java示例说明，其他语言用户也可以找到相关的工具生成密钥对。 新建一个Maven项目，加入如下依赖：

```
<dependency>
     <groupId>org.bitbucket.b_c</groupId>
    <artifactId>jose4j</artifactId>
    <version>0.7.0</version>
</dependency>
```

使用如下的代码生成一对RSA密钥：

```
RsaJsonWebKey rsaJsonWebKey = RsaJwkGenerator.generateJwk(2048);
rsaJsonWebKey.setKeyId("authServer");
final String publicKeyString = rsaJsonWebKey.toJson(JsonWebKey.OutputControlLevel.PUBLIC_ON
LY);
final String privateKeyString = rsaJsonWebKey.toJson(JsonWebKey.OutputControlLevel.INCLUDE_
PRIVATE);
```

2.2 使用JWK中的私钥实现颁发token 的认证服务

需要使用2.1节中在线生成的 `Keypair` JSON字符串（三个方框内的第一个）或者本地生成的 `privateKeyString` JSON字符串作为私钥来颁发token，用于授权可信的用户访问受保护的API，具体实现请参考本文第四节的示例。 向客户颁发token的形式由用户根据具体的业务场景决定，可以将颁发token的功能部署到生产环境，配置成普通API后由访问者通过用户名密码获得，也可以直接在本地环境生成token后，直接拷贝给指定用户使用。

2.3 将JWK中的公钥配置到JWT插件中

1. 登录API网关控制台。
2. 在左侧导航栏单击插件管理。
3. 在插件管理页面，单击右上角的 `创建插件` 。
4. 在创建插件页面，插件类型选择 `JWT鉴` ，下面是一个JWT鉴权插件的配置及说明，具体配置说明可以参考文档JWT认证插件。

```
---
parameter: X-Token          # 从指定的参数中获取JWT，对应API的参数
parameterLocation: header   # API为映射模式时可选，API为透传模式下必填，用于指定JWT的读取位置，仅支
持`query`,`header`
claimParameters:            # claims参数转换，网关会将jwt claims映射为后端参数
- claimName: aud            # claim名称,支持公共和私有
  parameterName: X-Aud      # 映射后参数名称
  location: header          # 映射后参数位置，支持`query,header,path,formData`
- claimName: userId         # claim名称,支持公共和私有
  parameterName: userId     # 映射后参数名称
  location: query           # 映射后的参数位置，支持`query,header,path,formData`
preventJtiReplay: false     # 是否开启针对`jti`的防重放检查，默认：false
#
# `Json Web Key`的`Public Key`， 即本文2.1节生成的公钥部分
jwk:
  kty: RSA
  e: AQAB
  use: sig
  alg: RS256
  n: qSVxcknOm0uCq5vGsOmaorPDzHUubBmZZ4UXj-9do7w9X1uKFXAnqfto4TepSNuYU2bA_-tzSLAGBsR-BqvT6w
9SjxakeiyQpVmexxnDw5WZwpWenUAcYrfSPEoNU-0hAQwFYgqZwJQMN8ptxkd0170PFauwACOx4Hfr-9FPGy8NCoIO4
MfLXzJ3mJ7xqgIZp3NIOGXz-GIAbCf13ii7kSStpYqN3L_zzpvXUAos1FJ9IPXRV84tIZpFVh2lmRh0h8ImK-vI42dw
lD_hOIzayL1Xno2R0T-d5AwTSdnep7g-Fwu8-sj4cCRWq3bd61Zs2QOJ8iustH0vSRMYdP5oYQ
```

2.4 JWT插件绑定API

在插件列表页找到刚刚创建好的JWT鉴权插件，单击 `绑定API` 按钮，在弹出框中添加指定分组和环境下的API到弹出框右侧API列表中，单击 `确定` ，绑定完成。

目前，控制台的API调试功能并没有支持JWT插件，建议用户通过Postman或者直接在系统命令行中应用 `curl` 命令测试绑定JWT插件的API。

三、API网关错误应答列表

Status	Code	Message	Description
400	I400JR	JWT required	未找到JWT参数
403	S403JI	Claim jti is required when preventJtiReplay:true	当在JWT授权插件中配置了防重放功能时，请求未提供有效的jti
403	S403JU	Claim jti in JWT is used	当在JWT授权插件中配置了防重放功能时，请求提供的jti已被使用
403	A403JT	Invalid JWT: ${Reason}	请求中提供的JWT非法

400	I400JD	JWT Deserialize Failed: ${Token}	请求中提供的JWT解析失
403	A403JK	No matching JWK, kid:${kid} not found	请求JWT中的kid没有匹配的JWK
403	A403JE	JWT is expired at ${Date}	请求中提供的JWT已过期
400	I400JP	Invalid JWT plugin config: ${JWT}	JWT授权插件配置错误

当出现非预期应答码是，请检查HTTP应答中的X-Ca-Error-Code头中获取ErrorCode，从X-Ca-Error-Message头中获取ErrorMessage当出现A403JT或I400JD错误码时，可访问jwt.io网站来检查自己的Token合法性与格式。

四、颁发token的认证服务示例代码

```
import java.security.PrivateKey;
import org.jose4j.json.JsonUtil;
import org.jose4j.jwk.RsaJsonWebKey;
import org.jose4j.jwk.RsaJwkGenerator;
import org.jose4j.jws.AlgorithmIdentifiers;
import org.jose4j.jws.JsonWebSignature;
import org.jose4j.jwt.JwtClaims;
import org.jose4j.jwt.NumericDate;
import org.jose4j.lang.JoseException;
public class GenerateJwtDemo {
    public static void main(String[] args) throws JoseException  {
          //使用在API网关设置的keyId
        String keyId = "uniq_key";
          //使用本文3.2节生成的Keypare
        String privateKeyJson = "{\n"
            + "  \"kty\": \"RSA\",\n"
            + "  \"d\": "
            +
            "\"O9MJSOgcjjiVMNJ4jmBAh0mRHF_TlaVva70Imghtlgwxl8BLfcf1S8ueN1PD7xV6Cnq8YenSKsfi
NOhC6yZ_fjW1syn5raWfj68eR7cjHWjLOvKjwVY33GBPNOvspNhVAFzeqfWneRTBbga53Agb6jjN0SUcZdJgnelzz5J
NdOGaLzhacjH6YPJKpbuzCQYPkWtoZHDqWTzCSb4mJ3n0NRTsWy7Pm8LwG_Fd3pACl7JIY38IanPQDLoighFfo-Lriv
5z3IdlhwbPnx0tk9sBwQBTRdZ8JkqqYkxUiB06phwr7mAnKEpQJ6HvhZBQ1cCnYZ_nIlrX9-I7qomrlE1UoQ\",\n"
            + "  \"e\": \"AQAB\",\n"
            + "  \"kid\": \"myJwtKey\",\n"
            + "  \"alg\": \"RS256\",\n"
            + "  \"n\": \"vCuB8MgwPZfziMSytEbBoOEwxsG7XI3MaVMoocziP4SjzU4IuWuE_DodbOHQwb_th
Uru57_Efe"
            +
            "--sfATHEa0Odv5ny3QbByqsvjyeHk6ZE4mSAV9BsHYa6GWAgEZtnDceeeDc0y76utXK2XHhC1Pysi2
KG8KAzqDa099Yh7s31AyoueoMnrYTmWfEyDsQL_OAIiwgXakkS5U8QyXmWicCwXntDzkIMh8MjfPskesyli0XQD1AmC
XVV3h2Opm1Amx0ggSOOiINUR5YRD6mKo49_cN-nrJWjtwSouqDdxHYP-4c7epuTcdS6kQHiQERBd1ejdpAxV4c0t0FH
F7MOy9kw\"\n"
            + "}";
```

```
        JwtClaims claims = new JwtClaims();
        claims.setGeneratedJwtId();
        claims.setIssuedAtToNow();
        //过期时间一定要设置，并且小于7天
        NumericDate date = NumericDate.now();
        date.addSeconds(120*60);
        claims.setExpirationTime(date);
        claims.setNotBeforeMinutesInThePast(1);
        claims.setSubject("YOUR_SUBJECT");
        claims.setAudience("YOUR_AUDIENCE");
        //添加自定义参数，所有值请都使用String类型
        claims.setClaim("userId", "1213234");
        claims.setClaim("email", "userEmail@youapp.com");
        JsonWebSignature jws = new JsonWebSignature();
        jws.setAlgorithmHeaderValue(AlgorithmIdentifiers.RSA_USING_SHA256);
          //必须设置
        jws.setKeyIdHeaderValue(keyId);
        jws.setPayload(claims.toJson());
        PrivateKey privateKey = new RsaJsonWebKey(JsonUtil.parseJson(privateKeyJson)).getPr
ivateKey();
        jws.setKey(privateKey);
        String jwtResult = jws.getCompactSerialization();
        System.out.println("Generate Json Web token , result is " + jwtResult);
    }
}
```

上述示例中有以下几个地方需要重点关注：

1. keyId需要三个环节都一致，且全局唯一：

- 使用2.1节生成密钥时填写的keyId;
- 在本文3.2.1节设置的keyId;
- 代码中的keyId，JsonWebSignature 对象的KeyIdHeaderValue的取值，该属性为必填

2. privateKeyJson 使用2.1节中在线生成的 `Keypair` JSON字符串（三个方框内的第一个）或者本地生成的 `privateKeyString` JSON字符串；
3. 过期时间一定要设置，并且小于7天；
4. 添加自定义参数，所有值请都使用String类型。

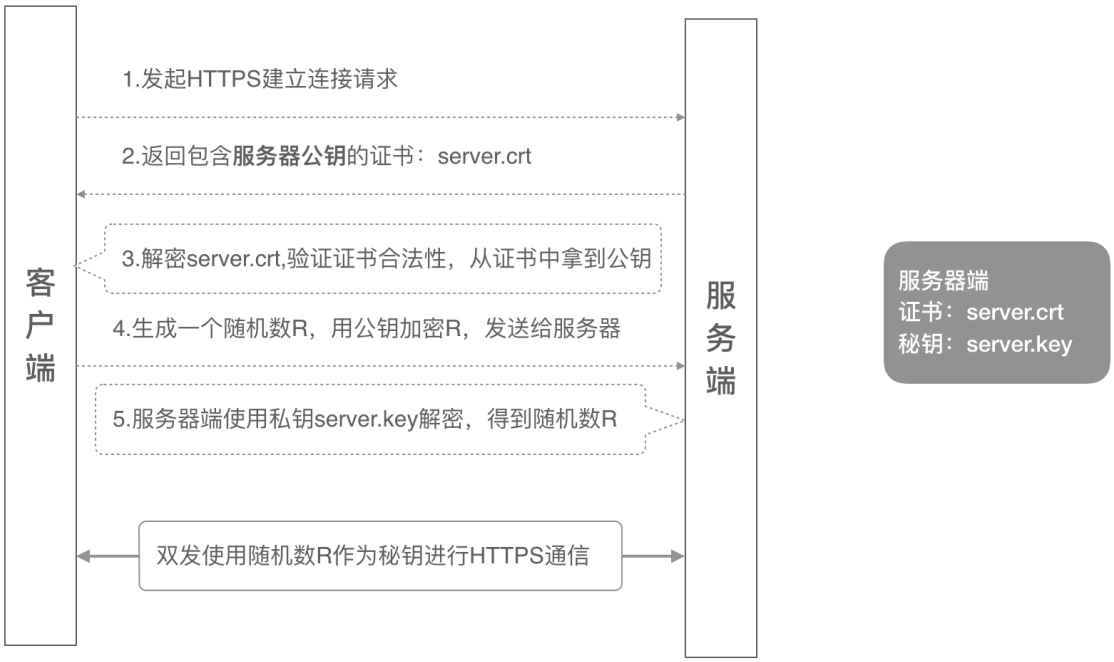
6.HTTPS双向认证（Mutual TLS authentication)

双向认证，顾名思义，客户端和服务器端都需要验证对方的身份，在建立HTTPS连接的过程中，握手的流程比单向认证多了几步。单向认证的过程，客户端从服务器端下载服务器端公钥证书进行验证，然后建立安全通信通道。双向通信流程，客户端除了需要从服务器端下载服务器的公钥证书进行验证外，还需要把客户端的公钥证书上传到服务器端给服务器端进行验证，等双方都认证通过了，才开始建立安全通信通道进行数据传输。

1. 原理

1.1 单向认证流程

单向认证流程中，服务器端保存着公钥证书和私钥两个文件，整个握手过程如下：



1. 客户端发起建立HTTPS连接请求，将SSL协议版本的信息发送给服务器端；
2. 服务器端将本机的公钥证书（server.crt）发送给客户端；
3. 客户端读取公钥证书（server.crt），取出了服务端公钥；
4. 客户端生成一个随机数（密钥R），用刚才得到的服务器公钥去加密这个随机数形成密文，发送给服务端；
5. 服务端用自己的私钥（server.key）去解密这个密文，得到了密钥R
6. 服务端和客户端在后续通讯过程中就使用这个密钥R进行通信了。

1.2 双向认证流程

1. 客户端发起建立HTTPS连接请求，将SSL协议版本的信息发送给服务端；
2. 服务器端将本机的公钥证书（server.crt）发送给客户端；
3. 客户端读取公钥证书（server.crt），取出了服务端公钥；
4. 客户端将客户端公钥证书（client.crt）发送给服务器端；
5. 服务器端使用根证书（root.crt）解密客户端公钥证书，拿到客户端公钥；
6. 客户端发送自己支持的加密方案给服务器端；
7. 服务器端根据自己和客户端的能力，选择一个双方都能接受的加密方案，使用客户端的公钥加密8. 后发送给客户端；
8. 客户端使用自己的私钥解密加密方案，生成一个随机数R，使用服务器公钥加密后传给服务器端；
9. 服务端用自己的私钥去解密这个密文，得到了密钥R
10. 服务端和客户端在后续通讯过程中就使用这个密钥R进行通信了。

2. 证书准备

从上一章内容中，我们可以总结出来，整个双向认证的流程需要六个证书文件：

- 服务器端公钥证书：server.crt
- 服务器端私钥文件：server.key
- 根证书：root.crt

- 客户端公钥证书：client.crt
- 客户端私钥文件：client.key
- 客户端集成证书（包括公钥和私钥，用于浏览器访问场景）：client.p12

所有的这些证书，我们都可以向证书机构去申请签发，一般需要收取一定的证书签发费用，此时我们需要选择大型的证书机构去购买。如果只是企业内部使用，不是给公众使用，也可以自行颁发自签名证书，具体的颁发办法请参见本文第四章。

3.在API网关配置HTTPS双向认证

在准备好上一章提到的六个证书文件后，就可以在API网关配置HTTPS的双向认证能力了，在配置之前，我们首先需要在API网关上拥有一个分组，并且在分组下绑定好了您的域名。本节介绍下将域名对应的服务器证书、根证书绑定到API网关来实现HTTPS双向认证的能力。

步骤1. 进入分组详情页面，找到要绑定的域名，点击域名对应的“选择证书”链接；

步骤2. 进入选择证书子页面，选择“手动添加证书”链接；

步骤3. 在手动添加证书页面，将第二章中说道的三个证书分别填写到本页面中：

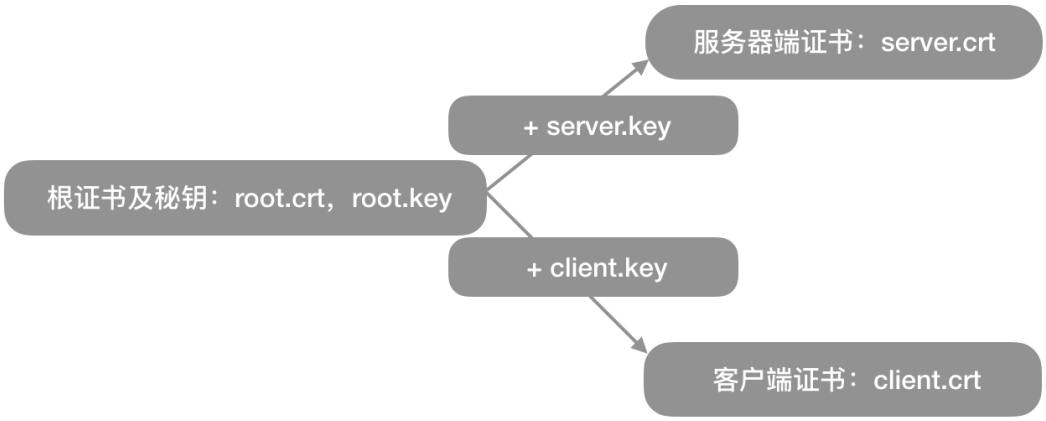
- 服务器端公钥证书（server.crt）的内容填写到“证书内容”文本框中；
- 服务器端私钥文件（server.key）的内容填写到“私钥”文本框中；

- 根证书（root.crt）的内容填写到“根证书”的文本框中；

通过以上三个步骤就可以在API网关完成配置HTTPS双向认证的配置。

4. 自签名证书

生成这一些列证书之前，我们需要先生成一个CA根证书，然后由这个CA根证书颁发服务器公钥证书和客户端公钥证书。为了验证根证书颁发验证客户端证书这个逻辑，我们使用根证书生成两套不同的客户端证书，然后同时用两个客户端证书来发送请求，看服务器端是否都能识别。下面是证书生成的内在逻辑示意图：



4.1生成自签名根证书

（1）创建根证书私钥：

```
openssl genrsa -out root.key 1024
```

（2）创建根证书请求文件：

```
openssl req -new -out root.csr -key root.key
```

后续参数请自行填写，下面是一个例子：

```
Country Name (2 letter code) [XX]:cn
State or Province Name (full name) []:bj
Locality Name (eg, city) [Default City]:bj
Organization Name (eg, company) [Default Company Ltd]:alibaba
Organizational Unit Name (eg, section) []:test
Common Name (eg, your name or your servers hostname) []:root
Email Address []:a.alibaba.com
A challenge password []:
An optional company name []:
```

（3）创建根证书：

```
openssl x509 -req -in root.csr -out root.crt -signkey root.key -CAcreateserial -days 3650
```

在创建证书请求文件的时候需要注意三点，下面生成服务器请求文件和客户端请求文件均要注意这三点： 根证书的Common Name填写root就可以，所有客户端和服务器端的证书这个字段需要填写域名，一定要注意的是，根证书的这个字段和客户端证书、服务器端证书不能一样； 其他所有字段的填写，根证书、服务器端证书、客户端证书需保持一致最后的密码可以直接回车跳过。

经过上面三个命令行，我们最终可以得到一个签名有效期为10年的根证书root.crt，后面我们可以用这个根证书去颁发服务器证书和客户端证书。

4.2 生成自签名服务器端证书

（1）生成服务器端证书私钥：

```
openssl genrsa -out server.key 1024
```

（2） 生成服务器证书请求文件，过程和注意事项参考根证书，本节不详述：

```
openssl req -new -out server.csr -key server.key
```

（3） 生成服务器端公钥证书

```
openssl x509 -req -in server.csr -out server.crt -signkey server.key -CA root.crt -CAkey ro
ot.key -CAcreateserial -days 3650
```

经过上面的三个命令，我们得到：

server.key：服务器端的密钥文件 server.crt：有效期十年的服务器端公钥证书，使用根证书和服务器端私钥文件一起生成

4.3 生成自签名客户端证书

（1）生成客户端证书密钥：

```
openssl genrsa -out client.key 1024
openssl genrsa -out client2.key 1024
```

（2） 生成客户端证书请求文件，过程和注意事项参考根证书，本节不详述：

```
openssl req -new -out client.csr -key client.key
openssl req -new -out client2.csr -key client2.key
```

（3） 生客户端证书

```
openssl x509 -req -in client.csr -out client.crt -signkey client.key -CA root.crt -CAkey ro
ot.key -CAcreateserial -days 3650
openssl x509 -req -in client2.csr -out client2.crt -signkey client2.key -CA root.crt -CAkey
root.key -CAcreateserial -days 3650
```

（4） 生客户端p12格式证书，需要输入一个密码，选一个好记的，比如123456

```
openssl pkcs12 -export -clcerts -in client.crt -inkey client.key -out client.p12
openssl pkcs12 -export -clcerts -in client2.crt -inkey client2.key -out client2.p12
```

重复使用上面的命令，我们得到两套客户端证书：

- client.key / client2.key：客户端的私钥文件

- client.crt / client2.key：有效期十年的客户端证书

使用根证书和客户端私钥一起生成 client.p12/client2.p12，这个证书文件包含客户端的公钥和私钥，主要用来给浏览器访问使用

5. 验证

使用curl加上证书路径，可以直接测试Nginx的HTTPS双向认证是否配置成功。下面我们测试三个用例：

- 使用client.crt / client.key这一套客户端证书来调用服务器端
- 使用client.crt2 / client2.key这一套客户端证书来调用服务器端
- 不使用证书来调用服务器端

下面是三个用例的测试结果：

5.1 带证书的成功调用：

```
#--cert指定客户端公钥证书的路径
#--key指定客户端私钥文件的路径
#-k 使用本参数不校验证书的合法性，因为我们用的是自签名证书
#可以使用-v来观察具体的SSL握手过程
curl --cert ./client.crt --key ./client.key https://integration-fred2.fredhuang.com -k -v
* Rebuilt URL to: https://47.93.XX.XX/
*   Trying 47.93.XX.XX...
* TCP_NODELAY set
* Connected to 47.93.XX.XX (47.93.XX.XX) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Request CERT (13):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Certificate (11):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS handshake, CERT verify (15):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: C=CN; ST=BJ; L=BJ; O=Alibaba; OU=Test; CN=integration-fred2.fredhuang.com; emai
lAddress=a@alibaba.com
*  start date: Nov  2 01:01:34 2019 GMT
*  expire date: Oct 30 01:01:34 2029 GMT
*  issuer: C=CN; ST=BJ; L=BJ; O=Alibaba; OU=Test; CN=root; emailAddress=a@alibaba.com
*  SSL certificate verify result: unable to get local issuer certificate (20), continuing a
nyway.
> GET / HTTP/1.1
> host:integration-fred2.fredhuang.com
> User-Agent: curl/7.54.0
> Accept: */*
```

```
>
< HTTP/1.1 200 OK
< Server: nginx/1.17.5
< Date: Sat, 02 Nov 2019 02:39:43 GMT
< Content-Type: text/html
< Content-Length: 612
< Last-Modified: Wed, 30 Oct 2019 11:29:45 GMT
< Connection: keep-alive
< ETag: "5db97429-264"
< Accept-Ranges: bytes
<
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
* Connection #0 to host 47.93.XX.XX left intact
```

使用client2.crt / client2.key这一套客户端证书来调用服务器端

```
curl --cert ./client2.crt --key ./client2.key https://integration-fred2.fredhuang.com -k
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

5.2 不带证书的调用

```
curl https://integration-fred2.fredhuang.com -k
<html>
<head><title>400 No required SSL certificate was sent</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
<center>No required SSL certificate was sent</center>
<hr><center>nginx/1.17.5</center>
</body>
</html>
```

三个用例都符合预期，从第一个测试日志中，我们可以看到，整个通信过程较长，客户端验证服务器端的证书，客户端也将自己的证书上传到服务器端进行验证。使用根证书颁发的两个客户端证书都可以正常发起双向HTTPS认证的调用。没有带客户端证书的调用会被服务器端拒绝服务。

6. 使用Java调用

由于使用的是自签名证书，使用ApacheHttpClient去调用的话，需要将服务器证书加入可信任证书库中，才能成功调用，也可以在代码中简单忽略证书。

```
cd $JAVA_HOME
sudo ./bin/keytool -import -alias ttt -keystore cacerts -file /Users/fred/temp/cert5/server
.crt
```

将服务器端公钥证书设置为可信证书后，使用以下代码可以直接发起带客户端证书的HTTPS请求：

```
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.ssl.SSLContexts;
import org.apache.http.util.EntityUtils;
import javax.net.ssl.SSLContext;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.security.KeyStore;
public class HttpClientWithClientCert {
    private final static String PFX_PATH = "/Users/fred/temp/cert5/client.p12";    //客户端证书路径
    private final static String PFX_PWD = "123456";    //客户端证书密码
    public static String sslRequestGet(String url) throws Exception {
        KeyStore keyStore = KeyStore.getInstance("PKCS12");
        InputStream instream = new FileInputStream(new File(PFX_PATH));
        try {
            keyStore.load(instream, PFX_PWD.toCharArray());
        } finally {
            instream.close();
        }
        SSLContext sslcontext = SSLContexts.custom().loadKeyMaterial(keyStore, PFX_PWD.toCharArray()).build();
        SSLConnectionSocketFactory sslsf = new SSLConnectionSocketFactory(sslcontext
                , new String[] { "TLSv1" }    // supportedProtocols ,这里可以按需要设置
                , null    // supportedCipherSuites
                , SSLConnectionSocketFactory.getDefaultHostnameVerifier());
        CloseableHttpClient httpclient = HttpClients.custom().setSSLSocketFactory(sslsf).build();
        try {
            HttpGet httpget = new HttpGet(url);
            //httpget.addHeader("host", "integration-fred2.fredhuang.com");// 设置一些header等
            CloseableHttpResponse response = httpclient.execute(httpget);
            try {
                HttpEntity entity = response.getEntity();
                String jsonStr = EntityUtils.toString(response.getEntity(), "UTF-8");//返回结果
                EntityUtils.consume(entity);
                return jsonStr;
            } finally {
                response.close();
            }
        } finally {
            httpclient.close();
        }
    }
    public static void main(String[] args) throws Exception {
        System.out.println(System.getProperty("java.home"));
        System.out.println(sslRequestGet("https://integration-fred2.fredhuang.com"));
    }
```

```
    }
}
```

7.WAF接入配置

本文主要介绍如何配置WAF，对API网关上发布的API进行增强安全防护。

1 概述

API网关的核心是为API提供认证、防篡改、防重放、参数验证、全链路签名、限流等诸多安全功能，因此针对恶意攻击者精心构造的攻击请求，进行应用层攻击（如OWASP TOP10常见Web攻击等）、暴力破解等情况，您可以考虑接入云盾Web应用防火墙（简称WAF），从而避免遭到入侵导致数据泄露，更好的保障您的业务安全。

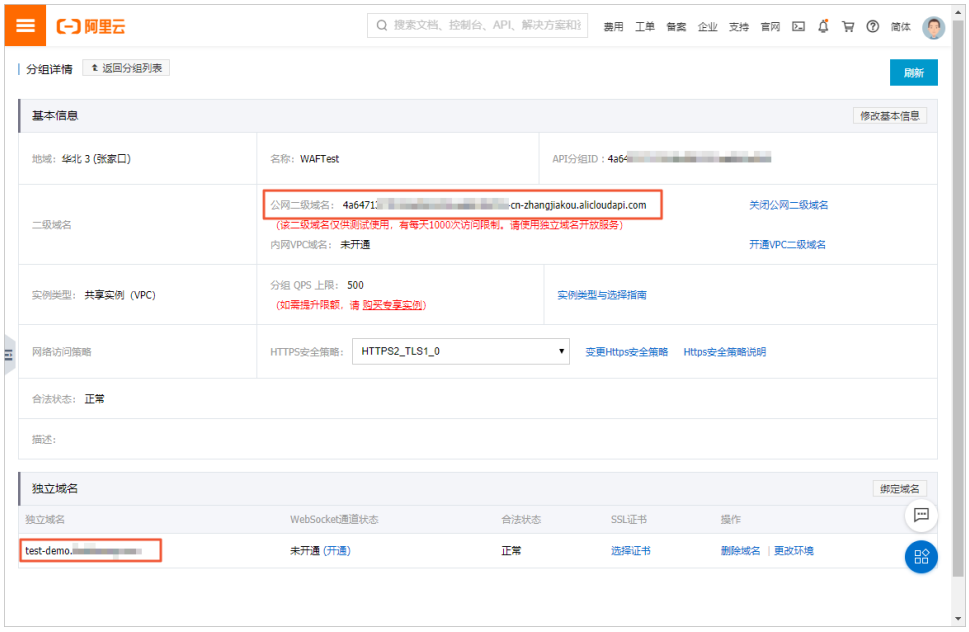
API网关和WAF完全兼容，您可以参考以下步骤为API接入WAF。

2 前提条件

- 开通Web应用防火墙
- 在API网关上已经发布了API

3 操作步骤

步骤1：在API分组上绑定您的域名，操作过程详见 分组的域名绑定。绑定成功后如下图所示：



> **注意**
>
> 由于后续步骤中还需配置WAF，建议当前阶段绑定域名时使用TXT解析。

步骤2：在WAF上添加网站。进入WAF控制台，在管理 - 网站配置菜单中添加站点。

主要的填写信息包括：

- 域名：填写您的域名，需要和 **步骤一** 中API网关分组上绑定的域名一致；
- 协议类型：需要和您在API网关在发布API的协议类型一致；
- 服务器地址：选择“其他地址”，填写API分组为您分配的公网二级域名。

点击下一步，按照WAF的提示，站点添加成功。之后为域名添加CNAME解析记录，将域名解析至WAF的CNAME地址，逐个完成业务流量的切换。