# API 网关

# 插件使用

腾讯云

# 文档目录

# 插件使用
# 插件概述

最近更新时间：2023-09-05 17:23:24

## 插件简介

插件是 API 网关提供的高级功能配置，您可以通过插件创建 IP 访问控制、条件路由、断路器、防重放等能力项，再将插件绑定到 API 上生效。
相较于传统配置项，插件有以下优势：

- 功能配置与 API 配置解耦，一个插件可以绑定到多个不同服务下的不同 API 上。
- 支持热更新，将插件绑定到 API 上后，无需发布服务即可生效。

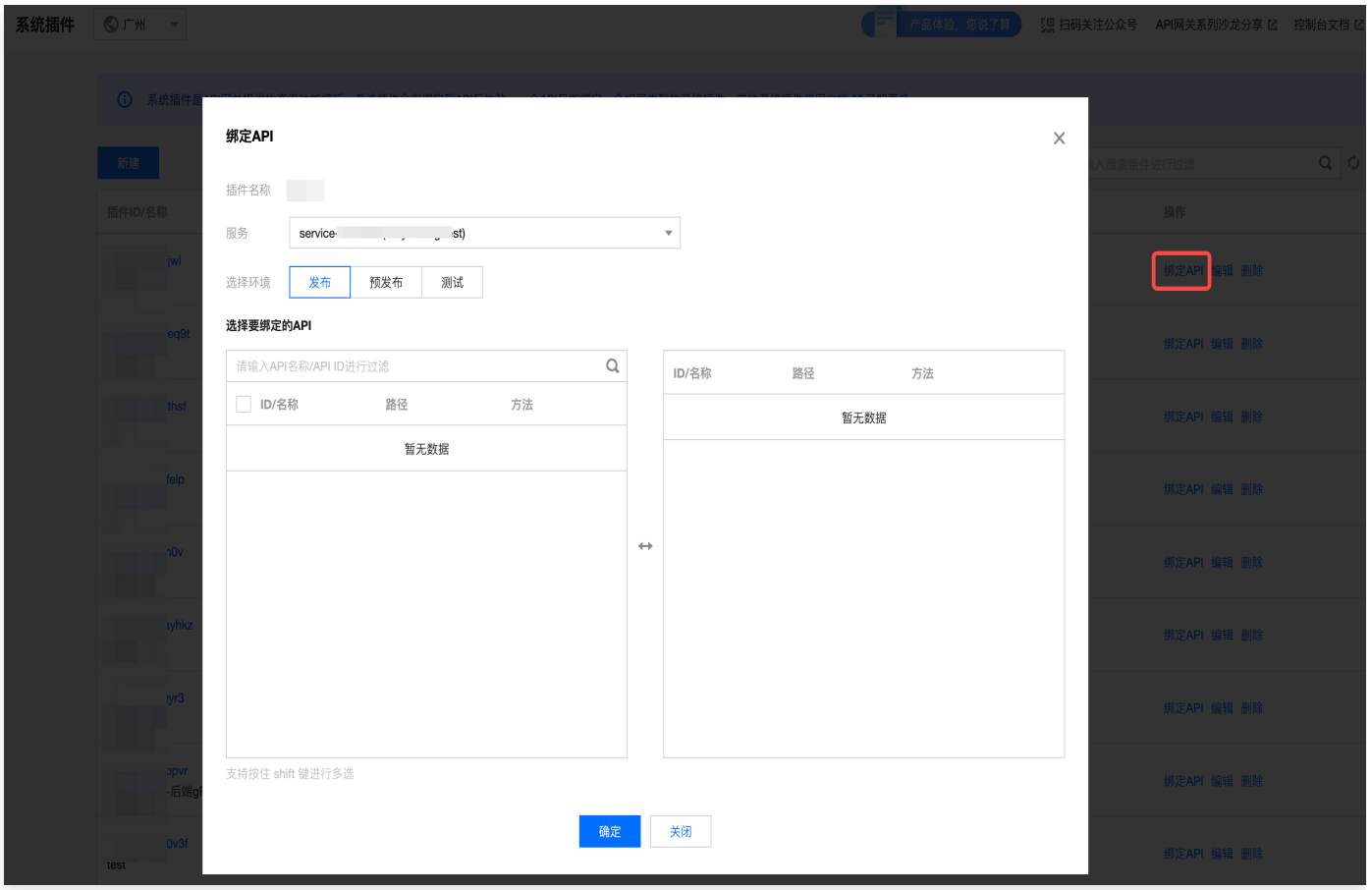
## 使用步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击 **插件 > 系统插件**，进入系统插件列表页。
3. 单击页面左上角的**新建**，新建一个插件，创建成功后回到系统插件列表。


### 步骤2：绑定 API 并生效

1. 在列表中选择刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

## 步骤3：查看 API 已绑定插件

1. 在左侧导航栏，单击**服务**，进入服务列表页。
2. 在服务列表中，单击目标服务的服务名，查看该服务。
3. 在 API 列表中，单击目标 API 的 API 名，查看该 API 详情页。
4. 在 API 详情页单击**插件管理** tab 页，即可查看当前 API 所绑定的插件信息。

## 插件规则

- 一个 API 只能绑定一个相同类型的插件。
- 插件具有地域属性，插件只能绑定到同一地域的 API，不支持跨地域绑定。
- API 只有发布到相应环境后，才能和插件进行绑定，未发布的 API 不支持绑定。
- 将 API 下线不影响和插件的绑定关系，重新发布到环境后插件仍然会生效。
- 插件支持热更新，所有的绑定、解绑操作无需重新发布服务即可生效。
- API 删除后，API 和插件的绑定关系也会一起删除。

## 已支持系统插件类型

- IP访问控制
- 基础流量控制
- 参数流量控制
- 跨域访问控制
- 条件路由
- 缓存
- 防重放

## 已支持自定义插件类型

- 自定义认证
- 自定义请求体
- 自定义响应体

### 自定义插件的服务类型

在创建自定义插件时，需要先选择一种**自定义服务类型**，此类型表示该自定义插件的后端服务类型，满足如下规则：

- 当自定义服务类型选择**云函数 SCF** 时，可支持该插件绑定到 共享实例服务上的 API。
- 当自定义服务类型选择**云函数 SCF**、**公网**、**内网 VPC** 时，可支持该插件绑定到专享实例服务上的 API。
- 共享实例与专享实例的规格差异，请参见 实例规格 。

# IP 访问控制

最近更新时间：2023-06-26 10:21:55

## 操作场景

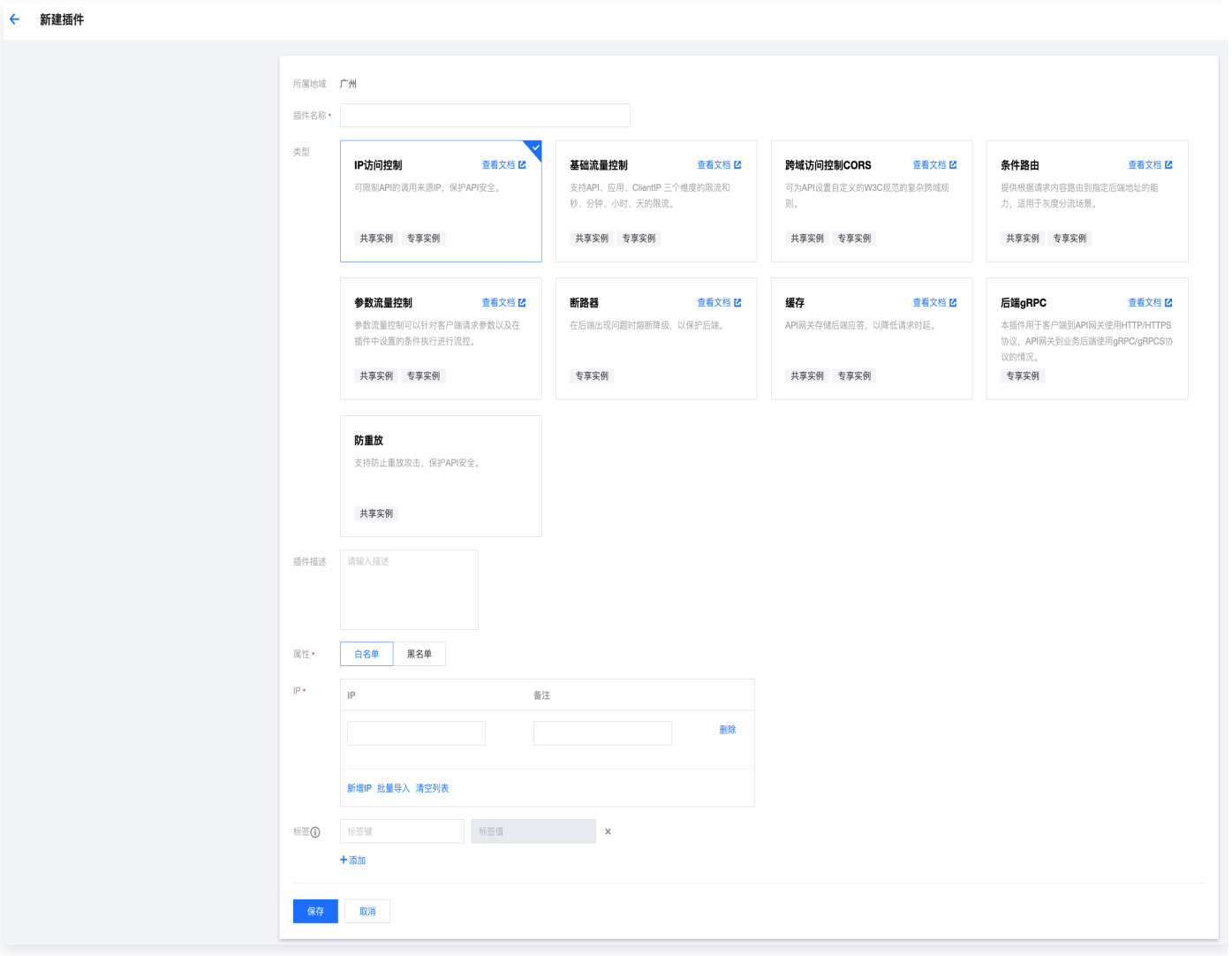
IP 访问控制是 API 网关提供的安全防护能力，主要用于限制 API 的调用来源 IP，您可以通过配置某个 API 的 IP 白名单/黑名单来允许/拒绝某个来源的 API 请求。

> **说明**
> 历史功能中的 IP 访问控制策略数据已经迁移到了 IP 访问控制插件中，请您前往 插件页面 管理。

## 操作步骤

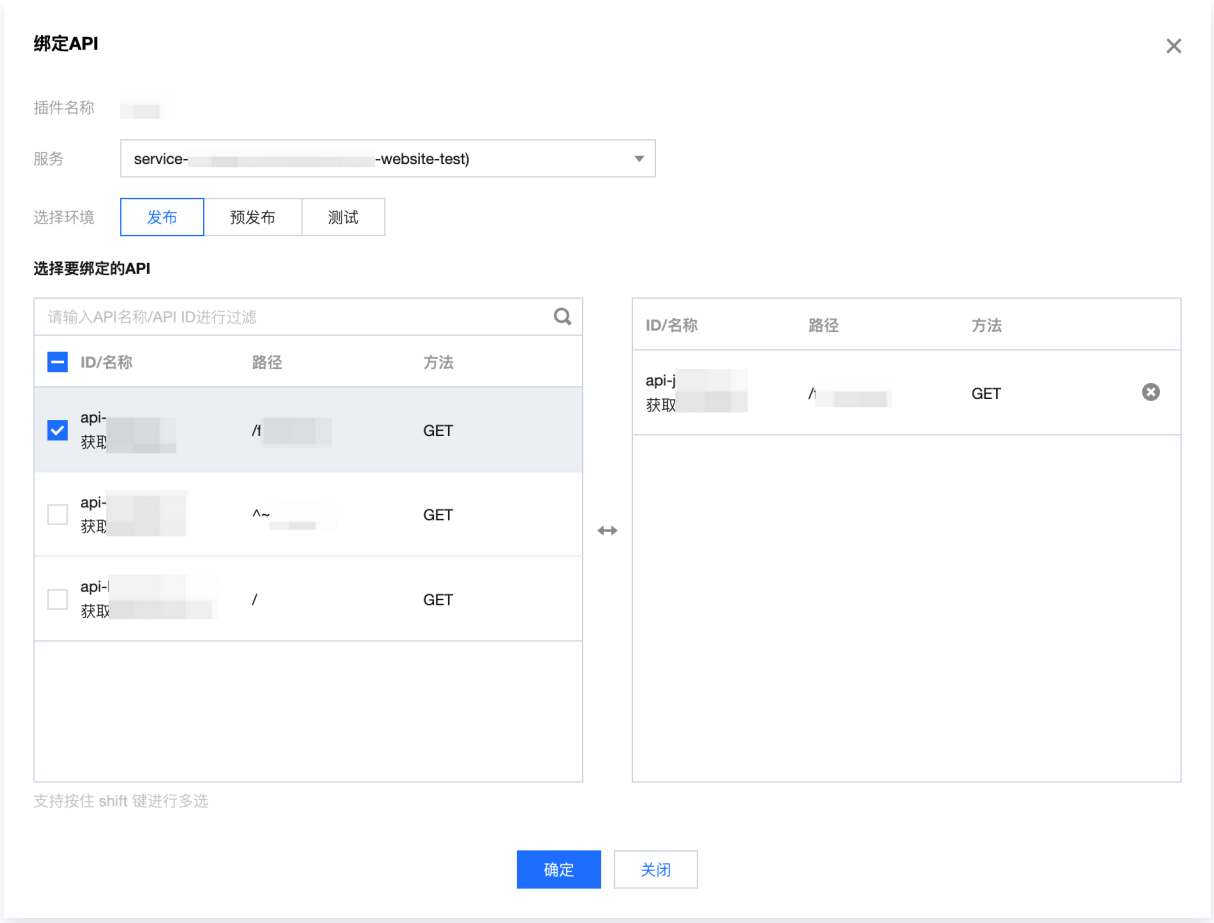
### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择 **IP 访问控制插件**。




## 步骤2：绑定 API 并生效

1. 在系统插件列表中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。
3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。




# PluginData

```
{
    "type":"white_list",     // IP访问控制类型，支持白名单模式（white_list）或黑名单模式（black_list）
    "blocks":"1.1.1.1\n1.1.1.0/24"     // IP地址段，用\n分隔
    "descriptions":{"1.1.1.1":"desc", "1.1.1.0/24":"desc"} //ip描述，本字段可省略
}
```

# 注意事项

- IP 访问控制插件支持黑名单/白名单方式。使用白名单时，不在白名单的请求将被 API 网关拒绝；使用黑名单时，黑名单中的 IP 请求将被 API 网关拒绝。
- IP 访问控制插件的 IP 支持填写 IP 或 CIDR，多个 IP 或 CIDR 间用英文分号分隔。
- IP 访问控制插件的IP支持增加描述信息，"descriptions"字段为可选项。

# 使用限制

专享实例中暂时不支持对内网客户端 IP 进行访问控制。

# 基础流量控制

最近更新时间：2023-06-26 10:23:45

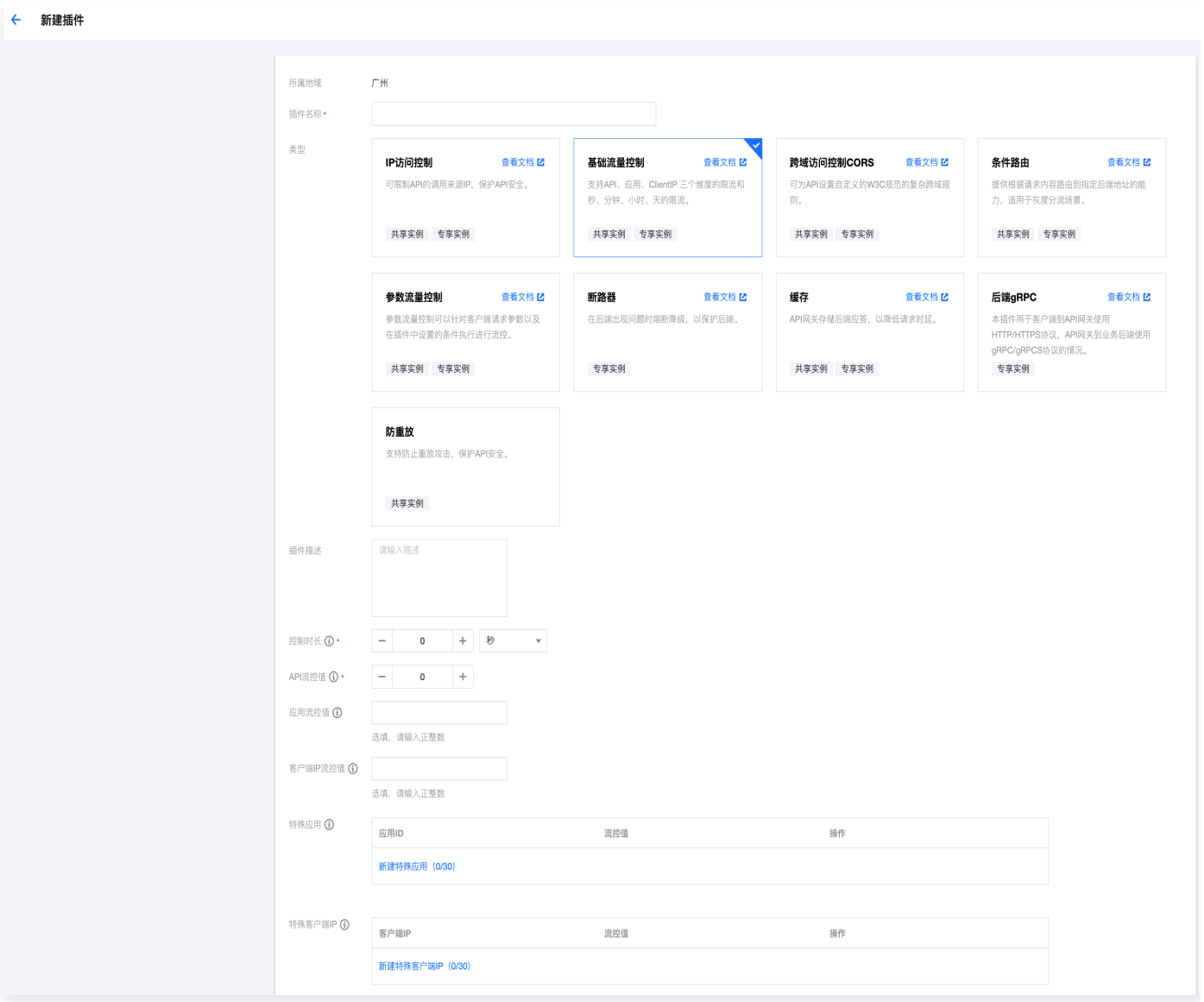
## 操作场景

基础流量控制插件是 API 网关提供的强大流控限制，支持API、应用、ClientIP 三个维度的限流和秒、分钟、小时、天的限流。您可创建基础流控插件并绑定到 API 生效，以保护您的后端服务。

## 操作步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择**基础流量控制**。

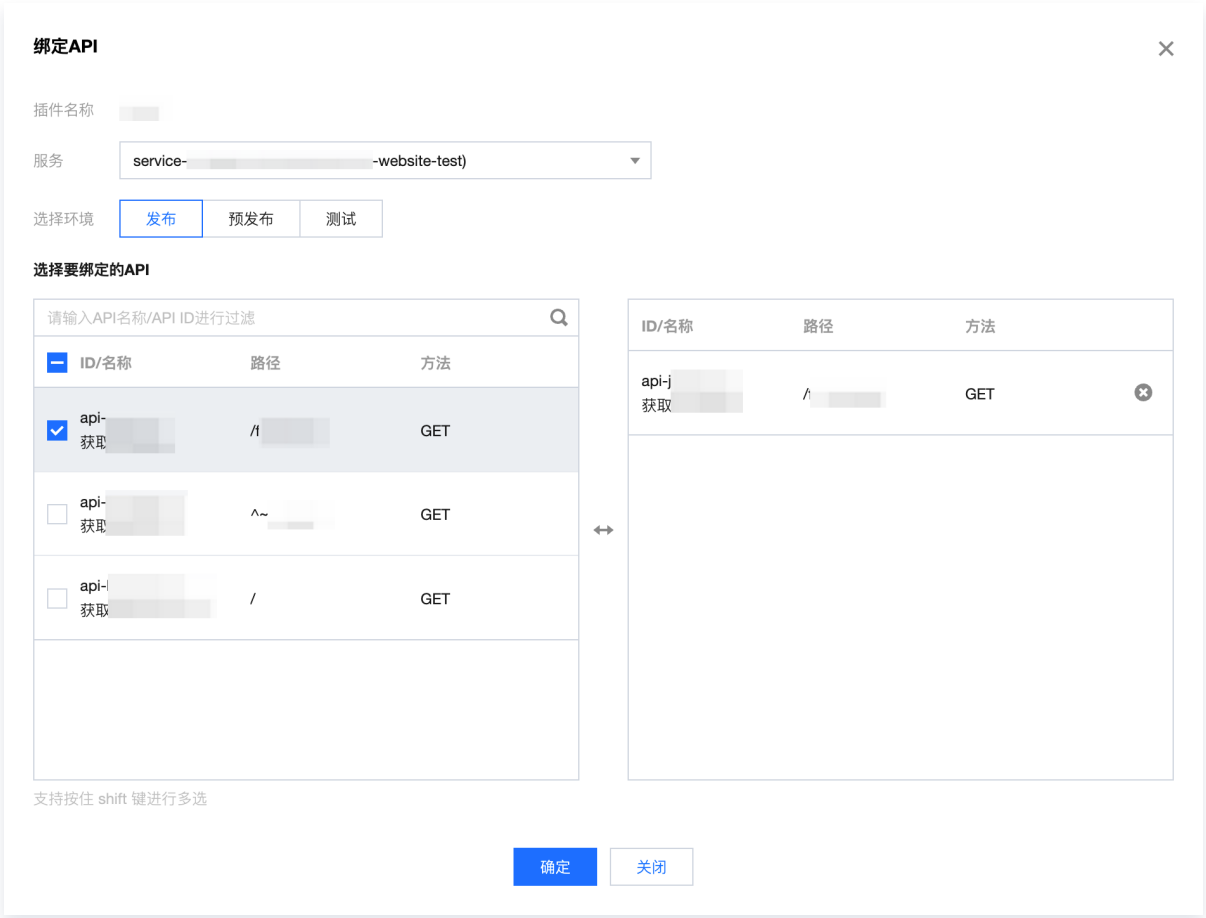



| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 控制时长 | 必填 | 流量控制的时长单位，支持设置秒、分钟、小时、天四种维度。与"流控值"配合使用，表示单位时间内的请求次数上限。 |
| API 流控值 | 必填 | API 流控值是指时长内一个 API 能够被访问的次数上限。 |
| 应用流控值 | 选填 | 对绑定了该 API 的所有应用生效，指时长内一个应用能够被访问的次数上限。 |
| 客户端 IP 流控值 | 选填 | 对绑定了该 API 的所有客户端 IP（ClientIP）生效，指时长内一个 ClientIP 能够被访问的次数上限。 |

| | | |
|---|---|---|
| 特殊应用 | 选填 | 最多可填写30个。对于特例，流控策略基础的 API 流量限制依然有效，您需要额外设定一个阈值作为该应用的流量限制值，同时流控策略基础的 App 流量限制和用户流量限制对该应用失效。 |
| 特殊客户端 IP | 选填 | 最多可填写30个。对于特例，流控策略基础的 API 流量限制依然有效，您需要额外设定一个阈值作为该 ClientIP 的流量限制值，同时流控策略基础的 App 流量限制和 ClientIP 流量限制对该应用失效。 |

## 步骤2：绑定 API 并生效

1. 在**系统插件列表**中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

# PluginData

```
{
    "expire_type":"hour", // 限流时间窗口单位，取值范围：[day, hour, minute, second]
    "expire":1, // 限流时间窗口大小
    "api_rate_limit":500, // API 流控值，需要为正整数
```

```
    "app_rate_limit":1, // 应用流控值，需要为正整数
    "ip_rate_limit":2, // 客户端 IP 流控值，需要为正整数
    "spec_app_rate_limits":[ // 特殊应用流控列表
        {
            "app_id":"app-3q914909", // 应用 ID
            "rate_limit":10 // 流控值，需要为正整数
        }
    ],
    "spec_ip_rate_limits":[ // 特殊客户端 IP 流控列表
        {
            "ip_key":"172.16.0.1", // 客户端 IP
            "rate_limit":10 // 流控值，需要为正整数
        }
    ]
}
```

## 注意事项

基础流控插件会受到服务流量控制、API 流控控制的影响；如果多种流控同时生效，最终流控值取最小限制。
例如：在基础流控插件中设置某 API 的流控为 500QPS，此时该 API 所属服务流控值为 100QPS，API 本身流控值为 50QPS，实际最终生效的流控值是 50QPS。

# 参数流量控制

最近更新时间：2024-11-14 10:55:01

## 操作场景

参数流控可以针对用户的请求参数以及用户在插件中设置的条件执行进行流控，参数流控配置支持如下特性：

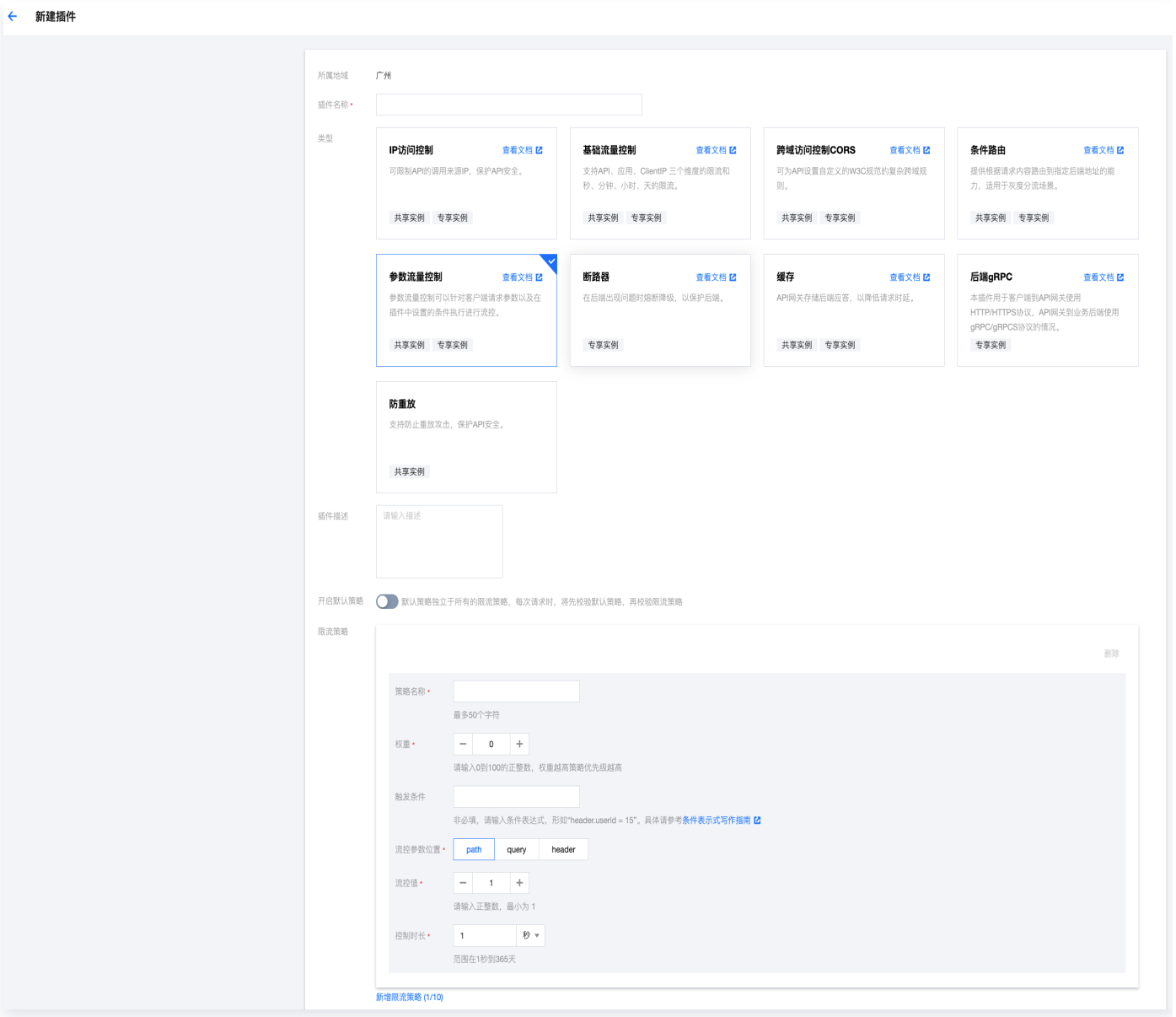
- 支持秒、分钟、小时、天的流控维度。
- 可以根据客户端请求参数、API 网关内置的系统参数设置条件，来执行不同的流控维度。
- 可以使用单个参数、或多个参数的组合来设置流控。

> **注意**
> 参数流控插件和 基础流控插件 同属于流控插件，一个 API 上只能绑定一个流控插件。因此如果在API已绑定流控插件时，再次绑定流控插件，则会自动替换成最新选定的插件。

## 操作步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择**参数流量控制**。



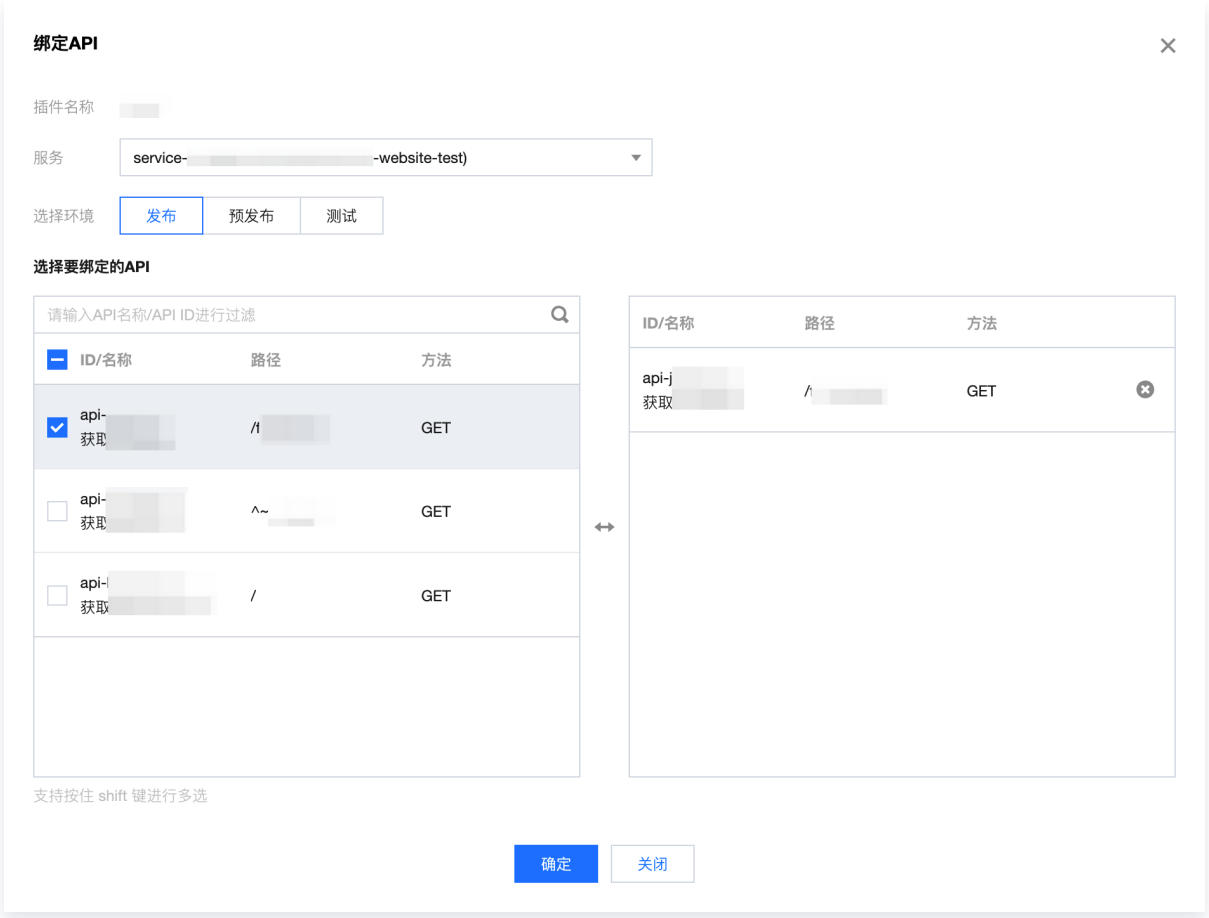

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 开启默认策略 | 必填 | 是否开启默认策略；默认策略是 API 级别的流控，独立于所有的流控策略；每次请求时，会先校验默认流控，再校验流控策略。 |
| 默认流控值 | 开启默认策略后必填 | 默认流控值是指时长内一个 API 能够被访问的次数上限，可输入正整数。 |
| 默认控制时长 | 开启默认策略后必填 | 支持秒、分钟、小时、天四个单位，与默认流控值搭配使用。 |
| 限流策略 | 必填 | 设置针对参数的限流策略，同一个参数限流插件内最多创建10条限流策略，限流策略的配置项详见下文。 |

同一个参数限流插件内最多创建10条限流策略，每条限流策略中都有以下配置项：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 策略名称 | 必填 | 当前策略的名称，最多50个字符，要求同一插件下不同策略间名称不能一样。 |
| 权重 | 必填 | 可输入0-100之间的正整数，权重越大优先级越高，要求同一插件下不同策略间权重不能一样。 |
| 触发条件 | 选填 | 如果设置了条件，只有当条件符合时，才会执行此条流控策略。<br>请输入条件表达式，条件表达式的写作规范详见本文 注意事项 章节。 |
| 流控参数位置 | 必填 | 仅支持填写一个流控参数，需要选择位置并填写参数名。例如：Header.ClientIP 表示，针对 Header 中每个 ClientIP 参数的取值分别进行流控。<br>流控参数位置支持 Header、Query、Path。Path 参数代表完整 API 路径，不需要填写参数名称。 |
| 流控参数名称 | 必填 | 仅支持填写一个流控参数，需要选择位置并填写参数名。例如：Header.ClientIP 表示，针对 Header 中每个 ClientIP 参数的取值分别进行流控。<br>流控参数名称需要和流控参数位置搭配使用。Path 参数代表完整 API 路径，不需要填写参数名称。 |
| 流控值 | 必填 | 本条策略中针对流控参数的流控值，请输入正整数，必须与控制时长搭配使用。 |
| 控制时长 | 必填 | 支持秒、分钟、小时、天四个单位，与流控值搭配使用。 |

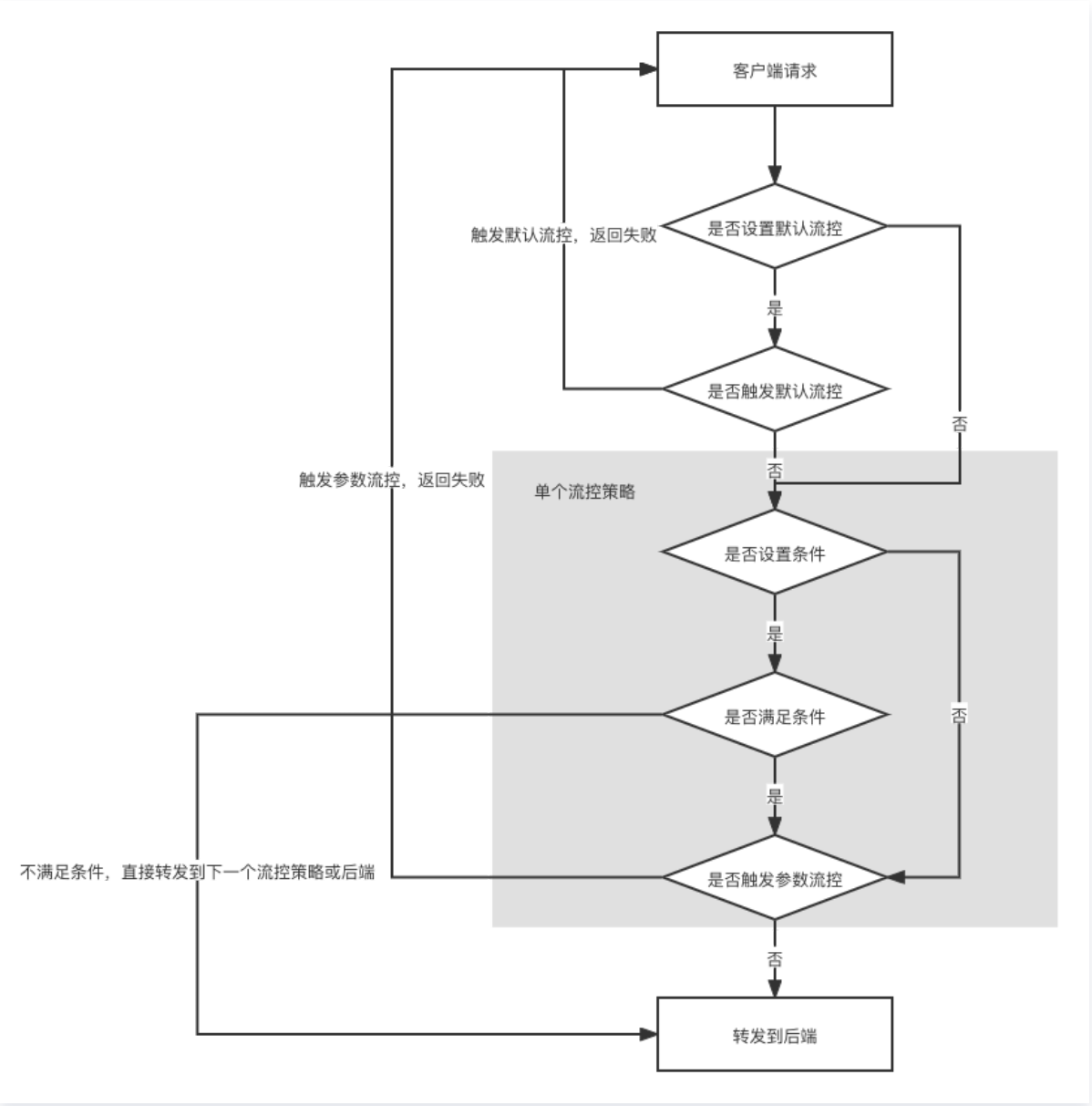
## 步骤2：绑定 API 并生效

1. 在**系统插件列表**中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

# 原理详解




- 默认策略是 API 级别的流控，独立于所有的限流策略。每次请求时，会先校验默认策略，再校验限流策略。
- 一个插件中同时配置多条流控策略时，API 网关将按策略权重由大到小的顺序依次校验请求是否满足条件；只要有一条流控策略不满足则触发流控，拒绝请求。
- 校验单条流控策略时，若配置了条件表达式将先校验条件，通过条件校验后再进行参数校验；未配置条件表达式时将直接根据参数校验。
- 如果在流控策略中设置 header.userid 参数每分钟流控10次，针对请求中该参数的每个不同取值，都会分别按每分钟十次进行限流。

## PluginData

```
{
    "default_window":60, // 限流时间窗口，单位秒，取值 0 时默认限流关闭
    "default_rate_limit":5, // 限流值，需要正整数
    "strategies":[ // 参数限流策略列表，至少一个策略，最多 10 个策略
```

```
        {
            "name":"a", // 策略名称
            "strategy_weight":0, // 策略执行优先级，数值高先执行，不允许重复
            "parameters":[ // 限流参数列表，暂时支持 1 个参数
                {
                    "type":"query", // 限流参数类型，取值范围:
[query,header,path]
                    "name":"a" // 限流参数名称，参数类型为 path 时，参数为整个
请求路径，不需要传 name
                }
            ],
            "rate_limit":1, // 策略限流值，需要正整数
            "window":1, // 策略限流时间窗口，单位秒
            "condition":"" // 策略触发条件，空字符串则为无条件执行
        }
    ]
}
```

## 注意事项

参数流控插件的条件表达式与条件路由插件的条件表达式完全一致，写作方法请参见 条件表达式写作指南 。

# 跨域访问控制 CORS

最近更新时间：2024-11-05 14:59:02

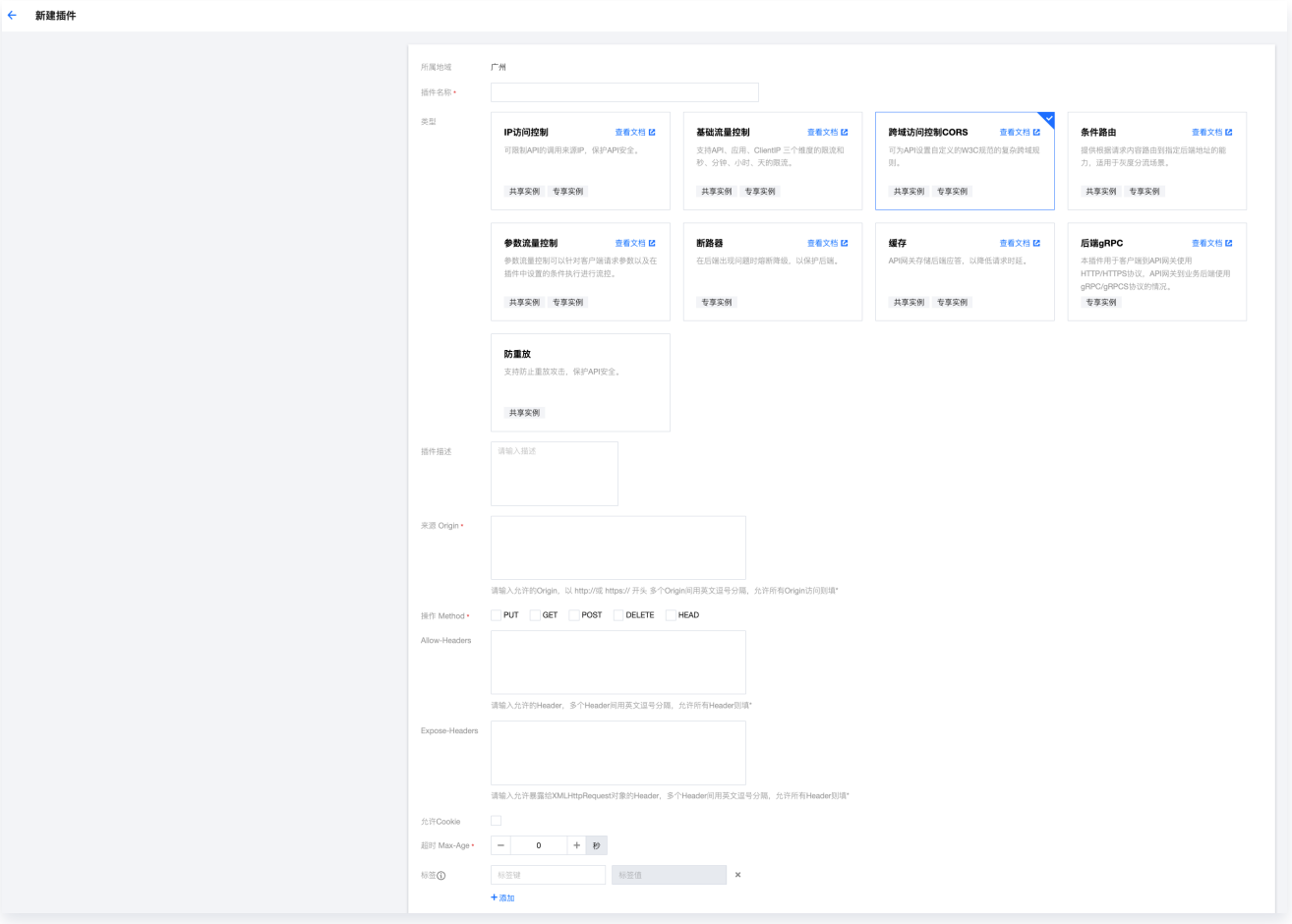
## 操作场景

跨域资源共享（Cross-Origin Resource Sharing，CORS）是 W3C 的标准。CORS 允许 Web 应用服务器进行跨域访问控制，从而使跨域数据传输得以安全进行。目前 API 网关支持对 CORS 规则的配置，从而根据需求允许或者拒绝相应的跨域请求。
当 API 网关的默认跨域配置不能满足您的需求时，您可通过跨域访问控制插件设置自定义的复杂跨域规则，并绑定到 API 生效。

## 操作步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择**跨域访问 CORS**。

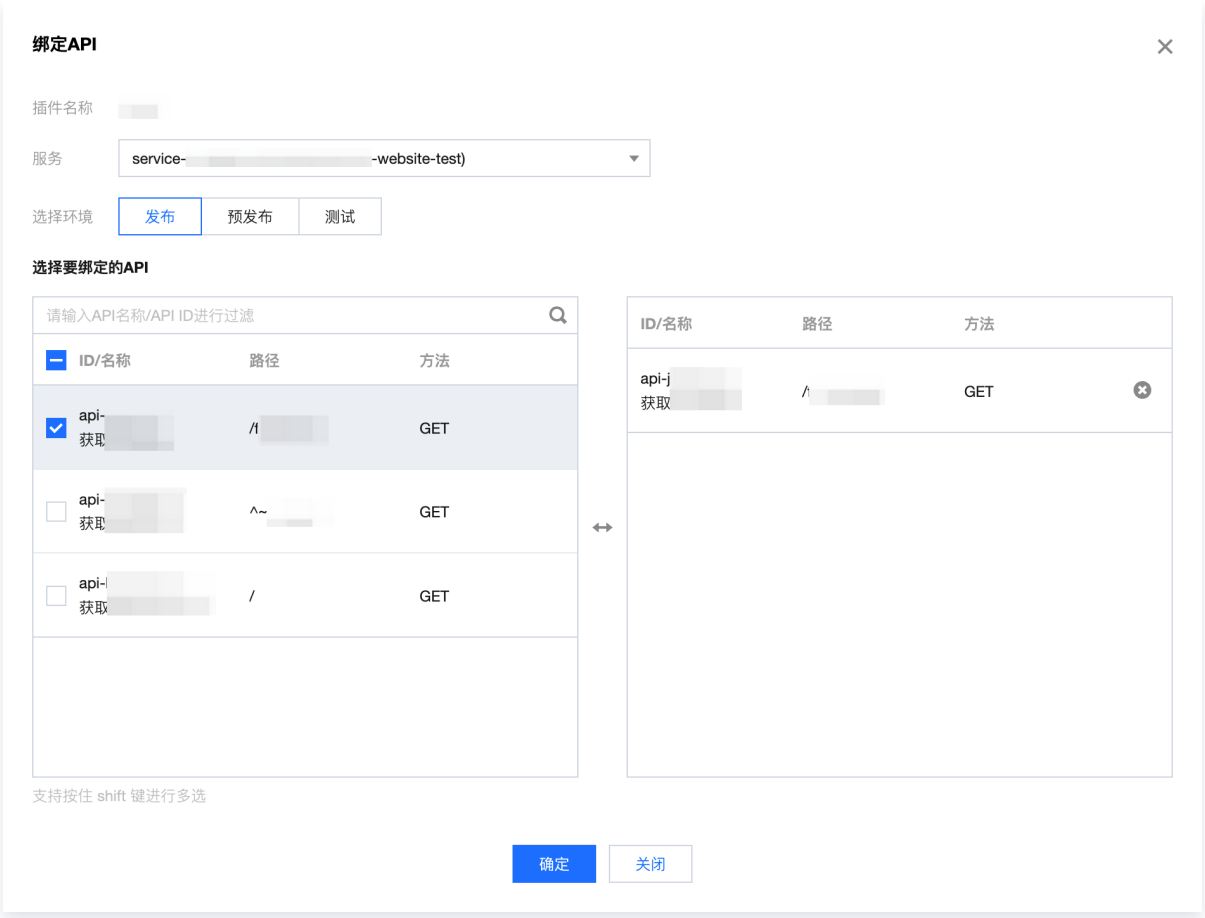



| 参数 | 是否必填 | 说明 |
| --- | --- | --- |

| | | |
|---|---|---|
| 来源 Origin | 必填 | • 允许跨域请求的来源。<br>• 可以同时指定多个来源，多个来源间用英文逗号分隔。<br>• 配置支持 `*` ，表示全部域名都允许。<br>• 注意不要遗漏协议名 http 或 https，若端口不是默认的80，还需要带上端口。<br>• 默认情况下，API 网关会根据请求来源的 Origin 进行跨域匹配。只要请求来源的 Origin 中包含了所配置的跨域 Origin，就会被认为是跨域请求。如果您希望进行完全匹配，可以使用正则表达式的 `^` 和符号来实现。例如，使用正则表达式：`^https://www.qq.com$` 可以确保只有完全匹配 https://www.qq.com 的请求才会被视为跨域请求。 |
| 操作 Method | 必填 | 支持 GET、PUT、POST、DELETE、HEAD。枚举允许一个或多个跨域请求方法。 |
| Allow-Headers | 选填 | 在发送 OPTIONS 请求时告知服务端，接下来的请求可以使用哪些自定义的 HTTP 请求头部。<br>可以同时指定多个 Headers，Header 间用英文逗号分隔。<br>配置支持 `*` ，表示全部 Header 都允许。<br>不填时代表空，即全部 Header 都禁止。 |
| Expose-Headers | 选填 | 允许暴露给XMLHttpRequest对象的头。<br>可以同时指定多个 Headers，Header 间用英文逗号分隔。<br>配置支持 `*` ，表示全部 Header 都允许。<br>不填时代表空，即全部 Header 都禁止。 |
| 允许Cookie | 选填 | 是否允许 Cookie。 |
| 超时 Max-Age | 必填 | 设置 OPTIONS 请求得到结果的有效期（秒）。数值必须为正整数，例如600。 |

## 步骤2：绑定 API 并生效

1. 在**系统插件列表**中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在**绑定API** 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

# PluginData

```
{
    "allow_origin":[ // 允许的ORIGIN，支持*，表示全部域名都允许
        "*"
    ],
    "allow_methods":[ // 允许的方法，支持GET、PUT、POST、DELETE、HEAD
        "PUT",
        "GET",
        "POST",
        "DELETE",
        "HEAD"
    ],
    "allow_headers":[ // 允许的请求头部，支持*，表示全部 Header 都允许
        "X-Api-ID"
    ],
    "expose_headers":[ // 允许暴露给XMLHttpRequest对象的头，支持*，表示全部
Header 都允许
        "X-Api-ID"
    ],
```

```
    "allow_credentials":true, // 是否允许Cookie
    "max_age":600 // 设置 OPTIONS 请求得到结果的有效期（秒）。数值必须为正整数，例如600秒
}
```

## 注意事项

目前 API 网关中有两个地方可以设置跨域访问控制规则：

- 创建 API – 前端配置 – 支持 CORS：在创建 API 时打开**支持 CORS** 配置项，开启后 API 网关将默认在响应头中添加 `Access-Control-Allow-Origin : *` 。
- 本文所描述的**跨域访问控制**插件请参见 操作步骤 。

**跨域访问控制**插件的优先级高于**支持 CORS** 配置项，当跨域访问控制插件绑定到某一 API 时，该 API 的**支持 CORS** 配置项将不生效。

# 条件路由

最近更新时间：2024-04-19 14:26:31

## 操作场景

条件路由插件可根据参数取值转发到不同后端。根据请求的参数取值与系统参数取值，按规则将不同的客户端请求转发到不同后端地址，可广泛应用于灰度发布、蓝绿发布、租户路由等场景。

## 操作步骤

### 步骤1：创建插件

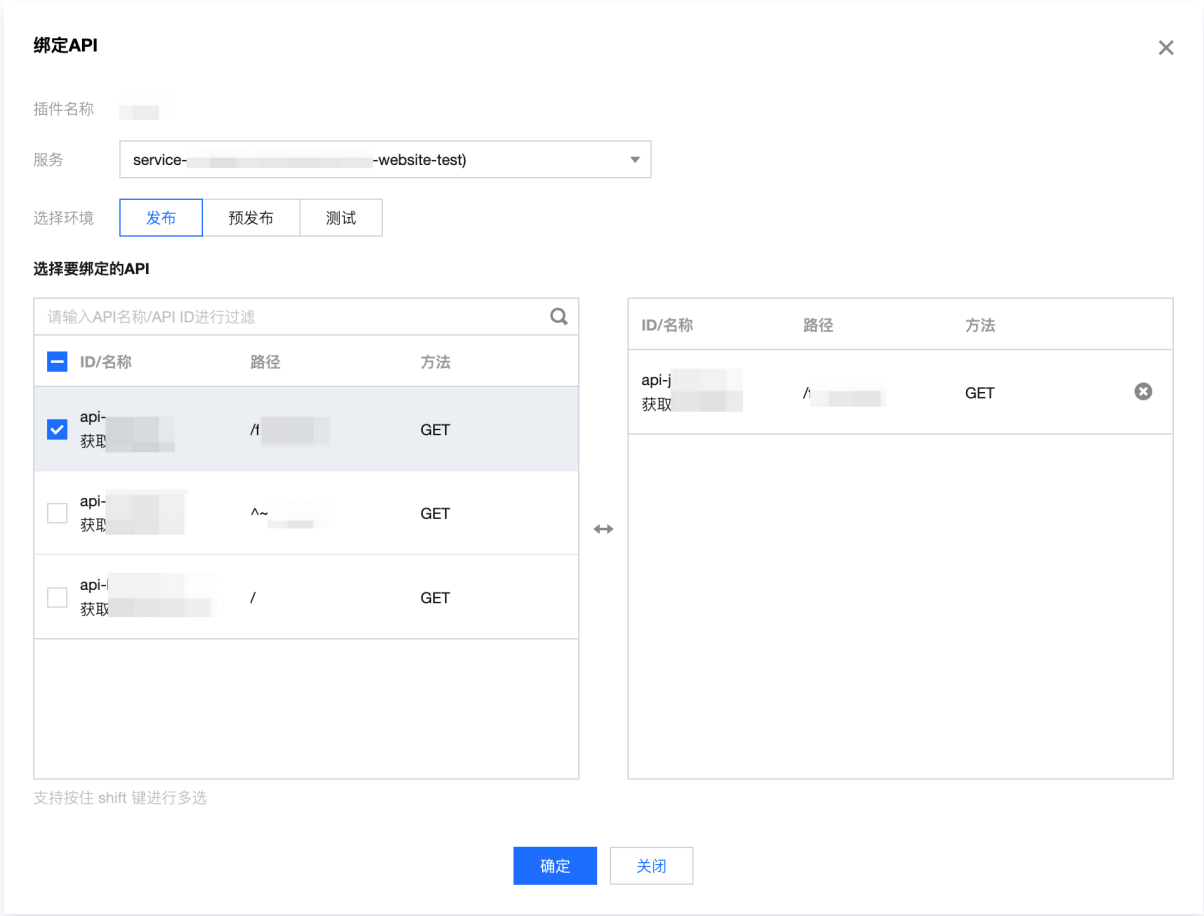
1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择**条件路由**。

   在条件路由插件中，您可最多同时创建10条路由策略，每条策略需要输入的内容如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 策略名称 | 必填 | 本条策略的名称，最多50个字符，要求同一插件下不同策略间名称不能一样。 |
| 权重 | 必填 | 策略匹配的优先级，可输入0-100之间的正整数，不填默认是0。<br>权重越大匹配优先级越高，权重相同时按创建时间由新到旧的顺序来排优先级。 |
| 触发条件 | 必填 | 用于判断客户端请求是否符合条件，请输入条件表达式，详见本文条件表达式相关说明。 |
| 后端类型 | 必填 | 支持公网 URL/IP、内网CLB、后端通道、云函数SCF、Mock、微服务平台 TSF。 |
| 后端配置 | 必填 | 当客户端请求满足条件时将被转发到的后端，请输入 YAML 格式的后端配置。 |


## 步骤2：绑定 API 并生效

1. 在**系统插件列表**中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。



3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

# 条件表达式写作指南

## 参数说明

条件表达式支持以下两种类型参数：

- **请求参数**，目前支持 Header、Path、Query 参数。
  - 您在 API 的前端配置中选择了 Header 位置、添加了参数名为 a 的参数，此时在插件中可以用 header.a 指代该参数。
  - Path 参数无参数名，因此使用 path 指代。例如：path='/test'，即请求路径为 /test 时满足条件。
  - Path 参数的参数值定义需以'/'开头，如'/test'。
- **系统参数**，目前可使用 **sysparam 参数形式**来引用当前请求的系统参数，系统参数不需要在 API 中被定义，可以直接引用。但如果您在 API 中定义了重名的参数，则取值会以您自定义的参数为准。系统参数推荐小驼峰写法，大小写不敏感。可用于条件路由插件的系统参数如下：
  - sysparam.clientIp：客户端 IP。
  - sysparam.httpScheme：请求的协议 HTTP、HTTPS。
  - sysparam.clientUa：客户端上传的 UserAgent 字段。

条件表达式支持以下常量类型：

| 常量类型 | 说明 | 示例 |
| --- | --- | --- |
| STRING | 字符串类型 | 支持单引号或双引号，如："Hello"、'hello' |
| INTEGER | 整数类型 | 例如：1001、-1 |
| NUMBER | 浮点数类型 | 例如：0.1、100.0 |
| BOOLEAN | 布尔类型 | 例如：true、false |

## 写作规则

1. 可以使用 and、or 来连接不同的表达式。
2. 可以用小括号 `(，)` 来指定条件判断的优先级。
3. `Random()` 作为内置函数，可以产生一个 0-1 的 NUMBER 浮点类型参数，用于随机的判断。
4. 如果表达式中使用了不存在的参数，例如 `param.unknown = 1`，则表达式的判断会返回 false。
5. 支持通过正则表达式函数 `regex()` 匹配参数值，例如 `regex(query.name,"colou?r")`。正则表达式字符串需要使用单引号或双引号。
6. 用 `exists()` 函数来指代是否存在，例如 `exists(header.Accept)`。
7. `==` 和 `=` 都可以用来判断“等于”关系。
8. 用 `not( )` 函数来判断“不等于”关系。例如 `not(sysparam.clientIp == "120.110.10.199")`。

### 条件表达式示例

- 5%的几率为真：
  **Random() < 0.05**
- 自定义 Header 参数中的 UserName 是 Admin 且来源 IP 是47.47.74.77：
  **header.UserName = 'Admin' and sysparam.clientIp = '47.47.74.77'**
- 当前请求的用户 Id（Header参数）是1001,1098,2011中的一个，且使用 HTTPS 协议请求：
  **sysparam.httpScheme = 'https' and (header.id = 1001 or header.id = 1098 or header.id = 2011)**

## 后端配置写作指南

支持填写对接公网 URL/IP、内网 CLB、后端通道、云函数 SCF、Mock、微服务平台 TSF 的后端配置：

- 后端配置必须是 YAML 格式的内容。
- 字段与 创建API 接口中后端配置的字段一一对应。
- 创建条件路由插件页已经列出了各后端的配置代码 Demo，您只需要修改参数值，即可完成配置。

## 公网 URL/IP

```
ServiceType: HTTP
ServiceConfig:
  Method: GET        # 后端服务方法
  Path: '/test'          # 后端服务路径
  Url: 'http://test.com'     # 后端服务URL
ReserveReqHost: true     #是否保留请求Host（默认为true）
ServiceTimeout: 15    # api后端超时时间（默认为15s）
```

## 内网CLB

```
ServiceType: HTTP
ServiceConfig:
  Method: GET        # 后端服务请求方法
  Path: /test        # 后端服务路径
  Url: 'http://test.com'         #后端服务URL
  UniqVpcId: vpc-xxxxx       # VPC 唯一ID
  Product: clb       # 标示后端为内网CLB资源（请保留）
ReserveReqHost: true     #是否保留请求Host（默认为true）
ServiceTimeout: 15    # api后端超时时间（默认为15s）
```

## 后端通道

```
ServiceConfig:
  Method: GET      # 后端服务请求方法
  Path: '/test'          # 后端服务路径
  Url: ''                #后端服务URL
  UniqVpcId: vpc-xxxxxx      # VPC 唯一ID
  UpstreamId: 'upstream-xxxxxx' # 后端通道唯一ID
  Product: upstream      # 标示后端为后端通道（请保留）
ReserveReqHost: true     #是否保留请求Host（默认为true）
ServiceType: HTTP
ServiceTimeout: 15    # api后端超时时间（默认为15s）
```

## 后端对接云函数 SCF

```
ServiceType: SCF
ServiceScfFunctionName: scftest      # SCF 函数名称
ServiceScfFunctionNamespace: mynamespace     # SCF 函数命名空间
ServiceScfFunctionQualifier: $LATEST        #示例表示以新发布版本号为准，也可填
写具体版本号，如版本为3则需要写为'3'
ServiceScfFunctionType: EVENT        # SCF 函数类型，事件函数（EVENT）web函数
（HTTP）
ServiceScfIsIntegratedResponse: false    # 是否开启响应集成
ServiceTimeout: 15    # api后端超时时间（默认为15s）
```

## 后端对接 Mock

```
ServiceType: MOCK
ServiceMockReturnMessage: hello mock from strategy   # 后端 Mock 返回信息
```

## 后端对接微服务平台 TSF

```
ServiceType: TSF
X-MicroService-Name: consumer-demo       # 微服务名称
X-NameSpace-Code: mytsf      # 微服务命名空间Code
MicroServices:
  - ClusterId: cls-xxxxxx            # 微服务集群ID
    MicroServiceName: consumer-demo          # 微服务名称
    NamespaceId: namespace-xxxxxx        # 微服务命名空间ID
ServiceConfig:
  Method: ANY    # 后端服务方法
```

```
  Path: '/'      # 后端服务路径
  Url: ''    # 后端服务URL
ServiceTsfHealthCheckConf:
  IsHealthCheck: true            # 是否开启健康检查
ServiceTsfLoadBalanceConf:
  IsLoadBalance: true        # 是否开启负载均衡
  Method: RoundRobinRule          # 负载均衡方式
  SessionStickRequired: false        # 是否开启会话保持
ServiceTimeout: 15    # api后端超时时间（默认为15s）
```

## pluginData

```
[
    {
        "strategy_name":"route-to-http", // 策略名，最多50个字符，只能包含 A-Za-z0-9 / % ~ _ \-.{}?&= ，同一插件下策略名不允许重复
        "strategy_weight":2, // 策略匹配的优先级，可输入0-100之间的正整数，不填默认是0
        "condition":"query.age<30 and query.need_verify=false or query.level>3", // 条件表达式
        "backend_type":"HTTP", // 路由转发后端类型，可选值：[MOCK, HTTP, SCF, VPC, UPSTREAM, TSF]
        "backend_config":{  // 后端配置
            "ServiceConfig":{
                "Method":"GET",
                "Path":"/v1/bpi/currentprice.json",
                "Url":"https://api.coindesk.com"
            },
            "ServiceType":"HTTP"
        }
    }
]
```

## 注意事项

当请求无法匹配到 API 绑定的条件路由插件中配置的路由策略时，该请求将被转发到 API 默认的后端配置上去。

# 缓存

最近更新时间：2023-06-26 10:42:44

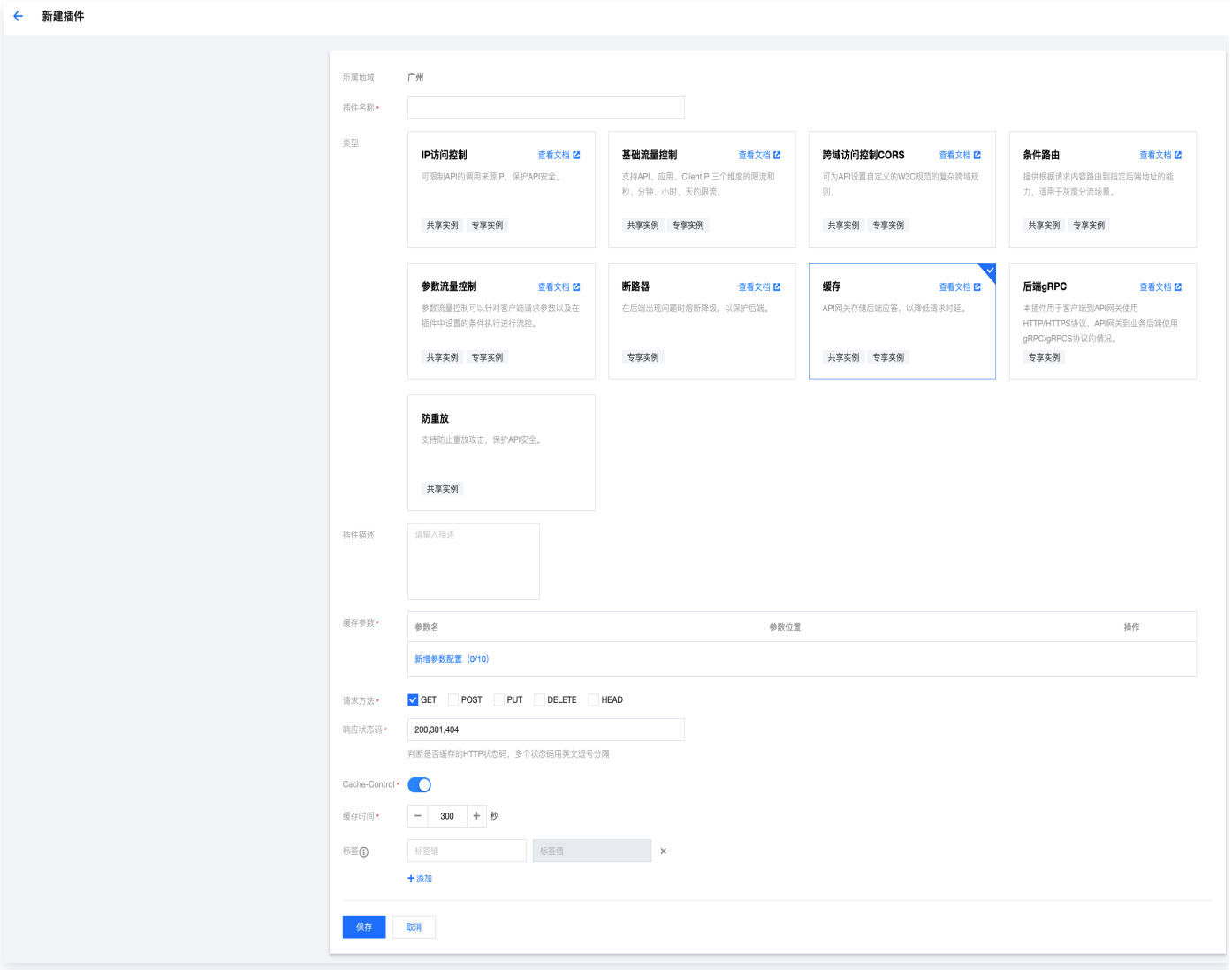
## 操作场景

通过配置缓存插件，API 网关可存储后端应答，当遇到相同请求参数的请求时，API 网关将直接返回缓存的应答，无需转发到后端服务，以此达到降低后端的负荷，减少时延，增加平滑度的目的。

## 操作步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件** ，进入系统插件列表页。
3. 单击列表左上角的**新建**，插件类型选择**缓存**。

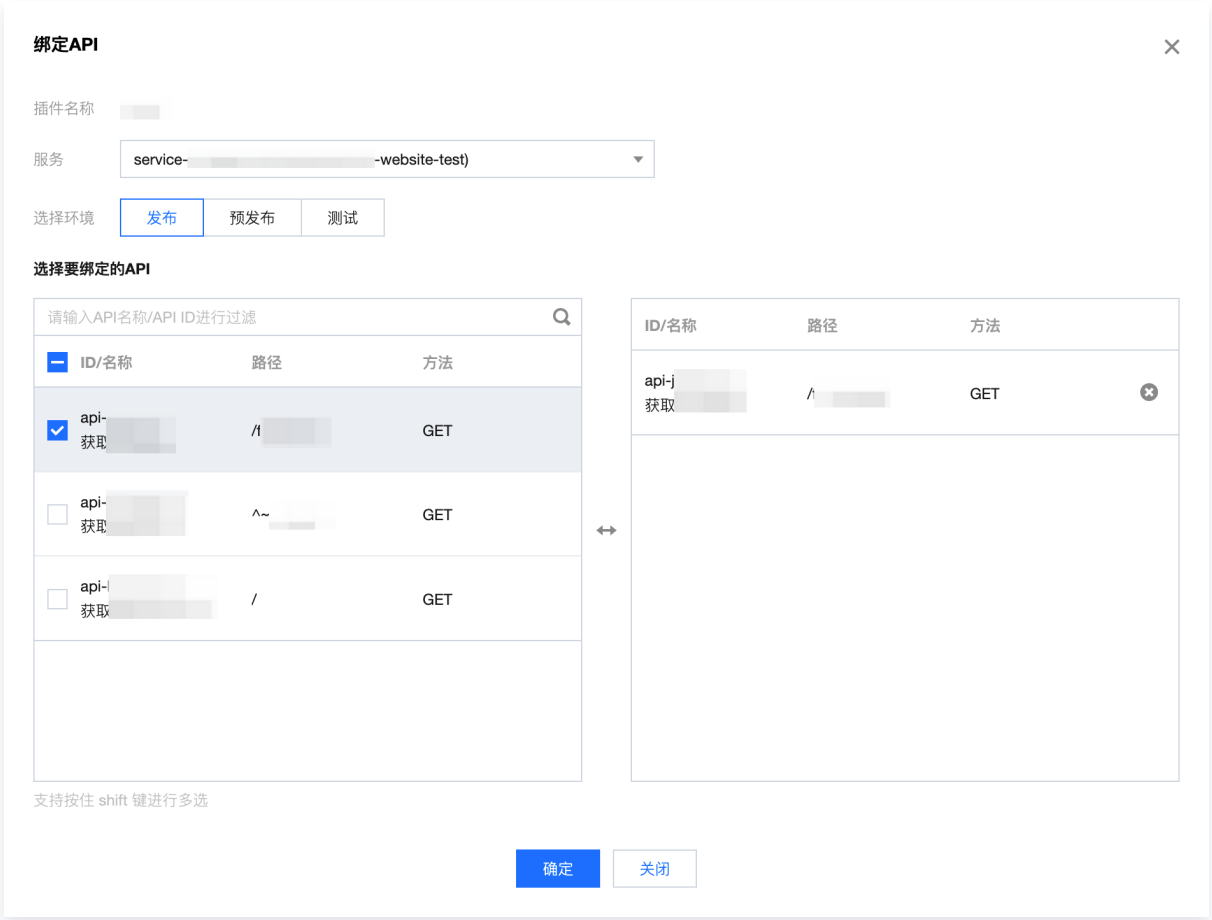


缓存插件的配置项如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 缓存参数 | 必填 | 根据参数或参数的组合区分缓存内容，参数位置支持选择 Header、Path、Query。 |
| 请求方法 | 必填 | 支持 GET、POST、PUT、DELETE、HEAD 方法，支持多选。 |
| 缓存状态码 | 必填 | 仅缓存插件中填写的状态码的响应，其他状态码不缓存，多个状态码间用英文逗号隔开。 |
| Cache-Control | 必填 | 通过 Cache-Control 请求头影响缓存策略，默认关闭。 |
| 缓存时间 | 必填 | 缓存有效的时间，可填写大于0到3600之间的正整数。 |

## 步骤2：绑定 API 并生效

1. 在**系统插件列表**中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。



3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

## PluginData

```
{
    "cache_key_params": [{ // 区分缓存的参数，parameter来源是api定义的参数，position取值为[header,query,path]
        "parameter": "param1",
        "position": "header"
    }, {
        "parameter": "param2",
        "position": "query"
    }, {
        "parameter": "param3",
        "position": "path"
    }],
    "cacheable_methods": ["GET", "POST"], // 可取缓存的http method，取值[GET、POST、PUT、DELETE、HEAD]
    "cacheable_response_codes": [200, 301, 404], // 可缓存的http返回码
    "cache_control": false, // 是否开启http标准的cache control语义，开启后，request和response的cache control都会生效，生效时会忽略自定义的ttl
    "ttl": 300 // 自定义缓存有效期，cache_control为false时生效。取值范围[1,3600]
}
```

## 注意事项

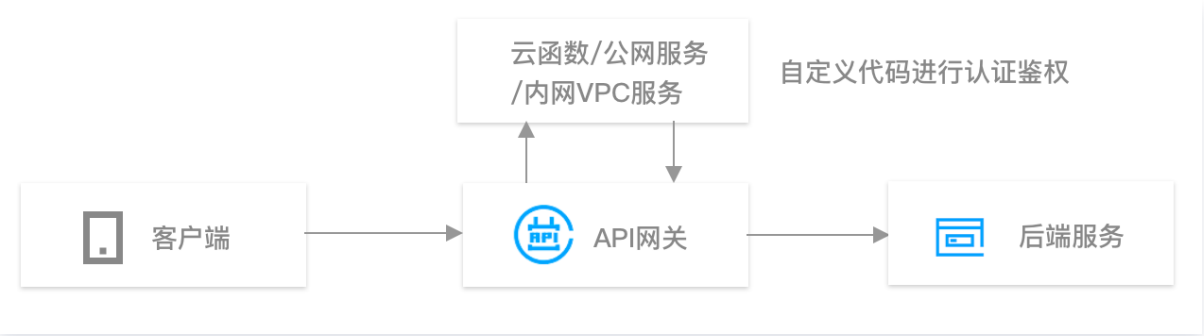
- API 网关做参数校验和命中缓存时，都是大小写不敏感的。
- 对于共享实例来讲，每个用户每个 Region 的缓存容量总限制为5M；每台专享实例的缓存容量总限制为1G。
- 当开启 `Cache-Control` 配置时，网关将遵守请求/响应头中的 Cache-Control 头的约定来处理缓存。在这种情况下，如果网关获取不到 Cache-Control 头，则默认进行缓存，并使用插件中配置的“缓存时间”字段作为缓存超期时间。

# 自定义认证

最近更新时间：2024-04-12 15:08:41

## 操作场景

如果 API 网关提供的认证鉴权方式不能满足您的需求，您可以使用自定义认证插件，通过您自定义的代码进行认证鉴权。
自定义认证插件作用在请求过程中，客户端请求 API 网关后，API 网关会将请求内容转发到认证函数中。您可以将认证函数部署在云函数上，公网、或内网 VPC 上，认证通过后请求才会被转发给业务后端，否则将拒绝请求。




## 前提条件

对于部署在云函数的认证服务，需开通 云函数 服务。
对于部署在公网/内网的服务，需要按照最下方的代码模板搭建对应的认证服务。

## 操作步骤

### 步骤1：创建认证函数

对于部署在公网或内网 VPC 的认证函数，可省略该步骤。

1. 登录 云函数控制台 。
2. 在左侧导航栏，单击**函数服务**，进入函数列表页。
3. 单击页面左上角的**新建**，选择模板创建，可搜索认证类函数模板，如图**模糊搜索自定义认证、选择标签为 Python2.7** 的版本。


4. 模板默认提供了三种参数认证的示例：Header、Query、Body。实际业务中，您可以根据需要选择不同的认证方式，也可以结合多个参数进行认证。

```python
# -*- coding: utf-8 -*-


def header_auth(event, response):
    header_parameters = event.get("headerParameters")  # 客户端原始请求的header内容，其中认证参数都被转成了小写
    if header_parameters is None or not isinstance(header_parameters, dict):
        return

    # 示例：通过 header-auth 进行认证
    if header_parameters.get("header-auth") == "apigw":
        response["api-auth"] = False # 未通过认证


def body_auth(event, response):
    body = event.get("body")   # 客户端请求的body内容

    # 示例：通过 body 进行认证
    if body is None:
        response["api-auth"] = False # 未通过认证


def query_auth(event, response):
    query_parameters = event.get("queryStringParameters")  # 客户端请求的query内容，其中认证参数都被转成了小写
    if query_parameters is None or not isinstance(query_parameters, dict):
        return

    # 示例：通过 query-auth 进行认证
    if query_parameters.get("query-auth") == "apigw":
        response["api-auth"] = False # 未通过认证
```

## 步骤2：创建自定义认证插件

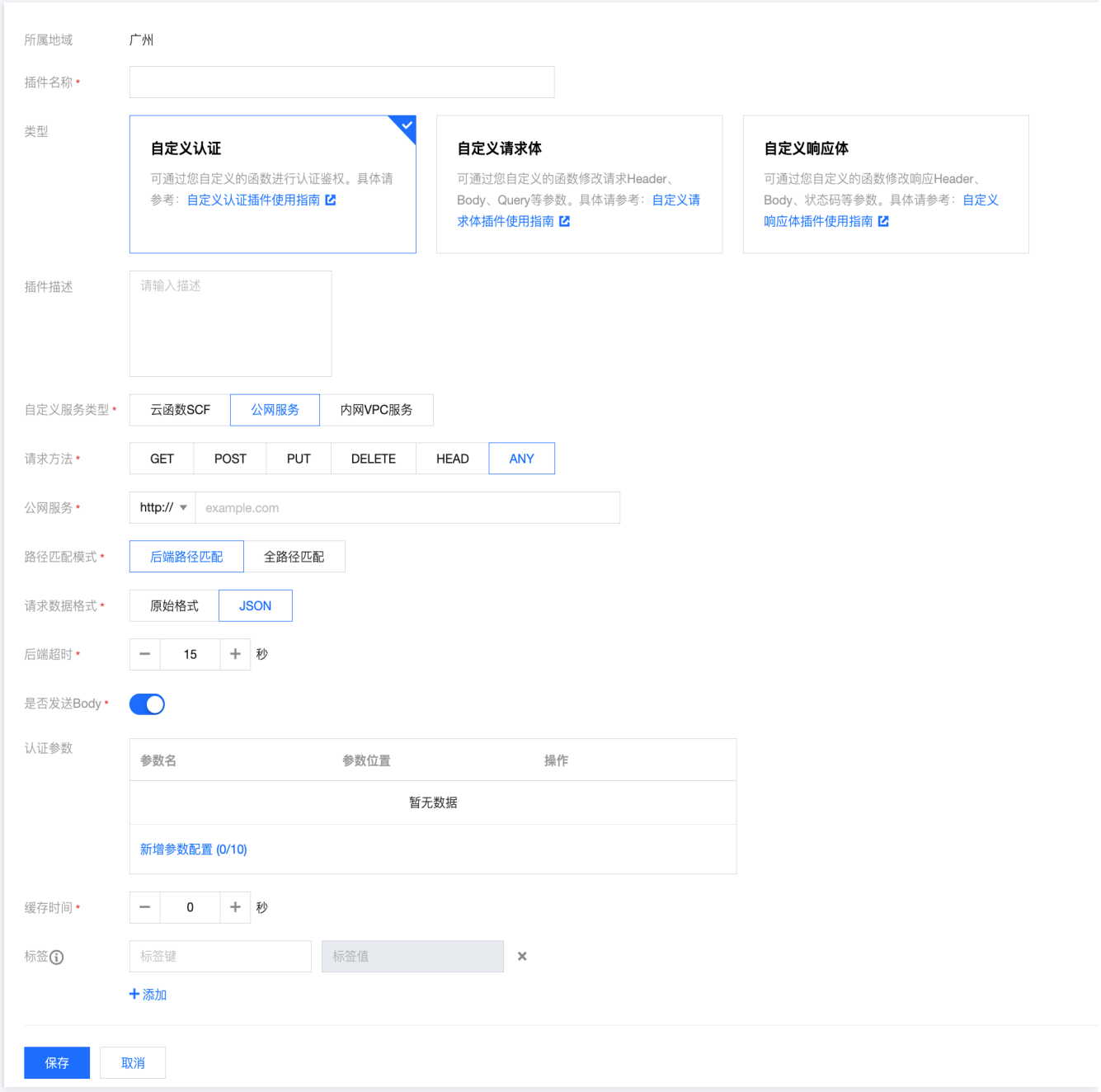
1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件** > **自定义插件**，进入自定义插件列表页。
3. 单击页面左上角的**新建**，新建一个自定义认证插件。
   - 对于部署在云函数的认证服务，创建自定义认证插件时需要填写的数据如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择函数 | 必填 | 选择认证函数所在的命名空间、名称和版本。 |
| 后端超时 | 必填 | 设置 API 网关转发到认证函数的后端超时时间，超时时间的最大限制为30分钟。在 API 网关调用认证函数，未在超时时间内获得响应时，API 网关将终止此次调用，并返回相应的错误信息。 |

| | | |
|---|---|---|
| 是否发送 Body | 必填 | • 当此选项的值为“是”时，会把客户端请求的 Header、Body、Query 都会发送给云函数。<br>• 当此选项的值为“否”时，不会发送请求 Body。 |
| 认证参数 | 选填 | 选填，但如果设置了参数则请求数据中为必填。当“缓存时间”不为0时，必须设置此参数。使用缓存时，此参数将作为搜索条件来查询认证结果。 |
| 缓存时间 | 必填 | 设置认证结果缓存的时间。值为0时代表不开启缓存，缓存时间最大支持3600秒。 |

- 对于部署在公网的认证服务，创建自定义认证插件时需要填写的数据如下：

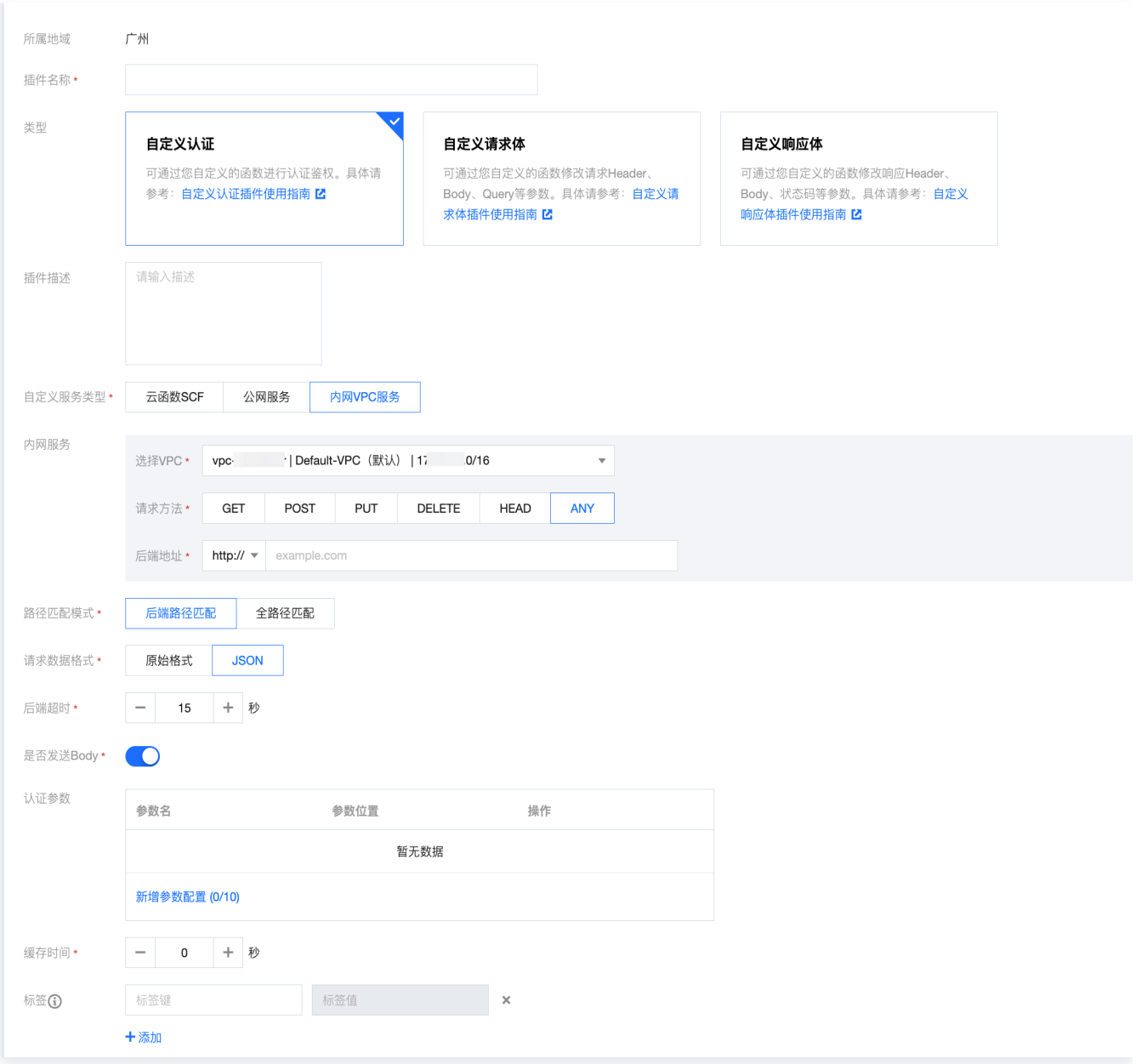
| 参数 | 是否必填 | 说明 |
|---|---|---|
| 请求方法 | 必填 | 请求自定义认证函数的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |
| 公网服务 | 必填 | 自定义认证服务访问地址，支持 HTTP 和 HTTPS 协议。 |
| 路径匹配模式 | 必填 | 支持后端路径匹配和全路径匹配两种方式。<br>• 后端路径匹配：直接使用配置的路径请求服务。<br>• 全路径匹配：使用去除请求路径的路径请求服务，如 API 路径配置为 /a/，请求路径为 /a/b，开启全路径匹配后，传输给服务的为 /b。 |
| 请求数据格式 | 必填 | APIGW会把客户端发送的请求按照不同的格式转发给公网/内网认证服务。<br>• 原始格式：按照请求原始的内容和格式转发给认证服务。<br>• JSON：将原始请求中的 Header Query Body Method PATH 参数转换成新的格式作为 Body 转发给公网/内网认证服务。 |
| 超时时间 | 必填 | 请求后端超时时间。默认15s。 |
| 是否发送 body | 必填 | 如果后端认证服务不需要通过 Body 做额外验证，可以不需要发送 Body，减少转发的性能损耗。 |
| 缓存时间 | 必填 | 相同的认证参数在缓存时间内只会请求一次后端。默认为0，表示不走缓存，每次都请求后端认证服务。 |
| 认证参数 | 选填 | 选填，但如果设置了参数则请求数据中为必填。当“缓存时间”不为0时，必须设置此参数。使用缓存时，此参数将作为搜索条件来查询认证结果。 |




- **对于部署在内网 VPC 的认证服务，创建自定义认证插件时需要填写的数据如下：**

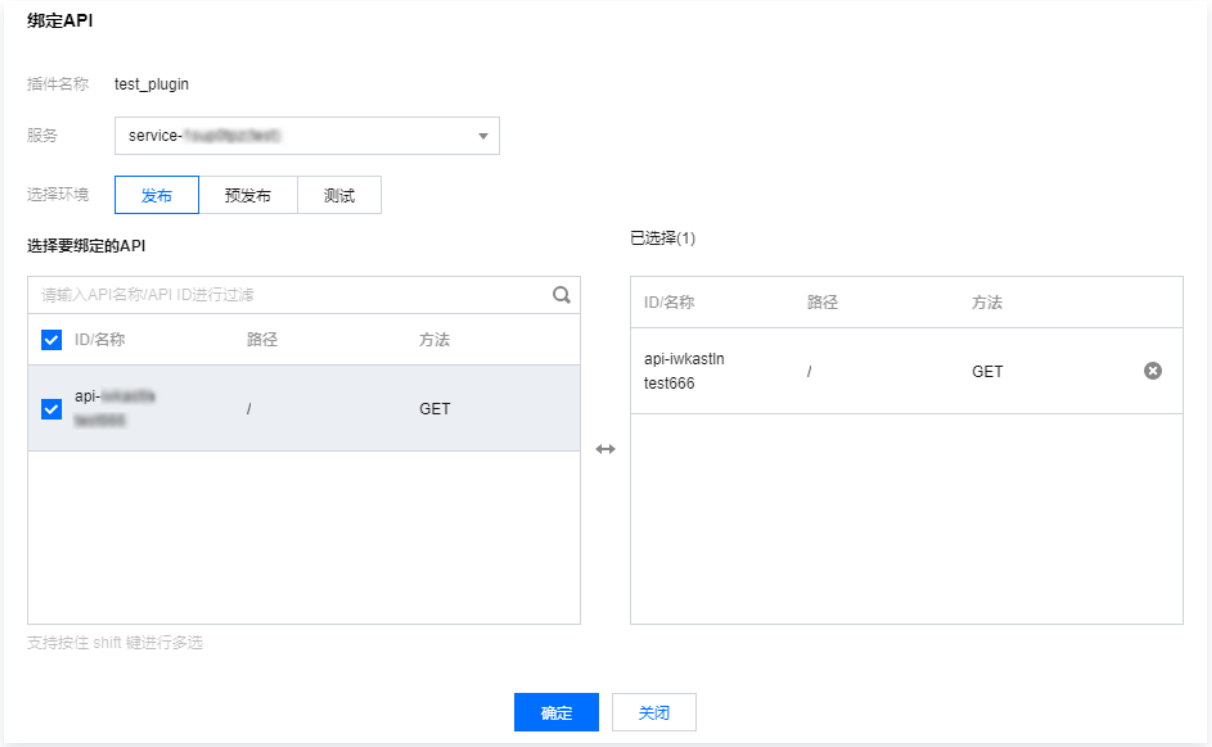
| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择 VPC | 必填 | 选择认证服务所属的 VPC。 |
| 请求方法 | 必填 | 请求自定义认证函数的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |

| | | |
|---|---|---|
| 后端地址 | 必填 | 自定义认证服务访问地址，支持 HTTP 和 HTTPS 协议。 |




## 步骤3：绑定 API

1. 在列表中选中刚刚创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

## 插件及请求响应示例

请在下方示例区切换页签，查看插件数据 pluginData、请求示例、响应示例。

**插件数据**

```
{
    "cache_time":10,    // 认证结果缓存时间，单位秒，合法值：0 ~ 3600 秒
    "endpoint_timeout":15, // 后端超时时间，单位秒，合法值：0 ~ 60 秒
    "func_name":"test_name", // 自定义函数名称
    "func_namespace":"test_namespace", // 自定义函数命名空间
    "func_qualifier":"$LATEST", // 自定义函数版本
    "is_send_body":true, // 是否将请求Body发送到函数
    "header_auth_parameters":[ // Header位置的认证参数，插件根据参数值来缓存认证结果
        "Header1"
    ],
    "query_auth_parameters":[ // Query位置的认证参数，插件根据参数值来缓存认证结果
        "Query1"
    ],
    "user_id":1253970226, // appid
    "version":"2021-12-26 17:17:49"
```

```
    // 插件版本，格式：yyyy-MM-dd HH:mm:ss，编辑插件时，传入新值会使得插件下的缓存结果失效
}
```

### 请求认证服务（JSON格式）

```
{
        "requestContext":  {    // 请求的信息
                "path":  "\/test",
                "httpMethod":  "POST",
                "sourceIp":  "113.108.77.64",
                "stage":  "release",
                "serviceId":  "service-i76yjnfu",
                "identity":  {}
                        },
        "path":  "\/test",          // 请求路径
        "httpMethod":  "POST",      // 请求方法
        "body":  "{\"data\":1}",    // 请求Body
        "queryStringParameters":  { // 请求的Query
                "query":  "1"},
        "headerParameters":  {      // 请求Header
                "accept":  "*\/*",
                "host":  "service-ixxxxx-xxxxx.gz.tencentapigw.cn",
                "header-auth":  "apigw",
                "content-length":  "10",
                "content-type":  "application\/x-www-form-urlencoded",
                "user-agent":  "curl\/8.1.2"
                        }
```

### 返回响应（application/JSON格式）

```
{
   "api-auth": true,                        // 认证结果 true:认证通过  false:认证失败
   "api-succ-headers": {"key":"value"},   // 认证成功后需要添加到转发到后端的header
   "api-response": "auth fail"              // 认证失败后返回自定义内容
}
```

## 注意事项

- 当用户开启缓存并配置了认证参数时，API 网关会进行参数校验。如果请求不传递该认证参数，API 网关将会报错“缺少 xxx 参数”。API 网关做参数校验和命中缓存时，都是大小写不敏感的。
- 每次将自定义插件绑定到一个网关 API 时，相当于为认证函数创建了一个该网关 API 的触发器。在 SCF 侧删除触发器，相当于把插件和 API 解绑。
- 自定义认证插件目前仅支持事件函数，不支持 Web 函数。
- 自定义认证插件可与 API 网关提供的认证方式共存，API 网关提供的认证方式优先级高于自定义认证，建议您将自定义认证插件绑定的 API 网关 API 设置为“免认证”。

## 自定义服务类型规则

自定义插件的自定义服务类型，支持绑定不同实例上的服务 API，规则请参见 插件概述-已支持自定义插件类型 。

## 示例代码：自定义认证服务

以下均以 Python2.7 为例，提供2种内网或公网的数据请求格式的示例。

**原始数据格式**

```
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlparse, parse_qs
import json

PORT_NUMBER = 8088

class MyHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        response = {'api-auth': True}
        # Header认证
        header_auth_key = self.headers.get("header-auth")
        if header_auth_key == "apigw":
            response['api-auth'] = False

        # Query认证
        # parsed_url = urlparse(self.path)
        # query_params = parse_qs(parsed_url.query)
        # query_auth_key = query_params.get("query-auth", [None])[0]
        #if query_auth_key == "apigw":
        #    response['api-auth'] = False

        self.send_response(200)
        self.send_header('Content-type', 'application/json')
```

```
        self.end_headers()
        self.wfile.write(json.dumps(response).encode('utf-8'))

try:
    server = HTTPServer(('', PORT_NUMBER), MyHandler)
    print(f'Started HTTP server on port {PORT_NUMBER}')

    # Wait forever for incoming HTTP requests
    server.serve_forever()

except KeyboardInterrupt:
    print('^C received, shutting down the web server')
    server.socket.close()
```

### JSON请求格式

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

PORT_NUMBER = 8088

class MyHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        response = {'api-auth': True}
        content_length = int(self.headers.get('Content-Length', 0))
        request_body = self.rfile.read(content_length).decode('utf-8')
        request_body = json.loads(request_body)

        # Header认证
        header_parameters = request_body.get('headerParameters', {})
        if header_parameters.get('header-auth') == "apigw":
            response['api-auth'] = False

        # Query认证
        # query_parameters = request_body.get('queryStringParameters',
{})
        # if query_parameters.get('query-auth') == "apigw":
        #     response['api-auth'] = False

        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
```

```
            self.wfile.write(json.dumps(response).encode('utf-8'))
try:
    server = HTTPServer(('', PORT_NUMBER), MyHandler)
    print(f'Started HTTP server on port {PORT_NUMBER}')

    # Wait forever for incoming HTTP requests
    server.serve_forever()

except KeyboardInterrupt:
    print('^C received, shutting down the web server')
    server.socket.close()
```

## 常见错误信息

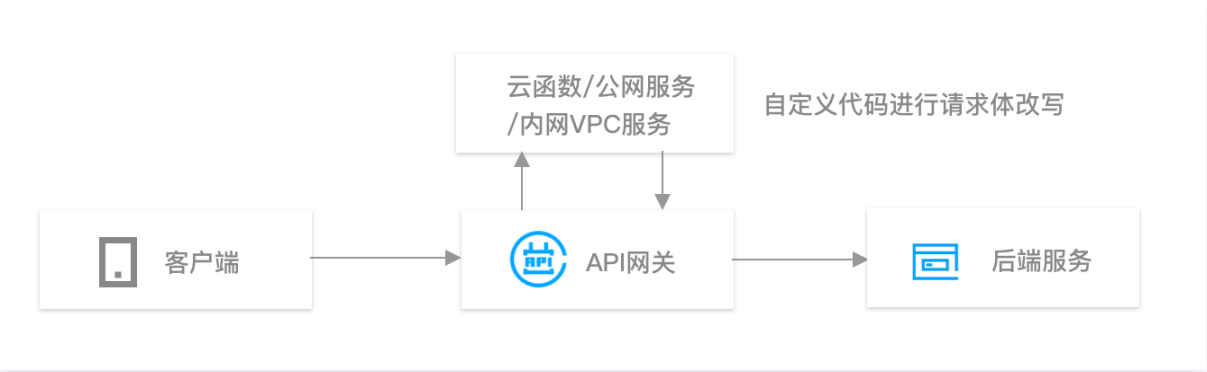
| 错误信息 | 含义 |
| --- | --- |
| Custom authentication failed: authentication information, incorrect | 认证失败，默认返回内容。 |
| Custom authentication failed: custom service request, request failed | 请求后端服务失败，需排查后端服务状态是否正常。 |
| Custom authentication failed: custom service response, invalid content | 认证服务返回的数据格式错误或者认证服务返回的数据做了 Gzip 压缩，当前不支持对认证服务响应做 Gzip 数据解析。 |

# 自定义请求体

最近更新时间：2023-09-19 15:49:31

## 操作场景

客户端发给业务后端的请求体中包含很多字段，如果您需要修改请求体内容，可以通过自定义请求体插件实现。自定义请求体插件作用在请求过程中， 请求体改写服务可部署在云函数、公网、或内网 VPC 上。客户端请求 API 网关后，API 网关会将请求内容转发到请求体改写服务中，请求体内容修改完成后，将修改后的请求体响应给 API 网关，API 网关再将修改后的请求体转发给业务后端。




## 前提条件

对于部署在云函数的认证服务，需开通 云函数 服务。

> **说明：**
> 自定义请求体插件目前仅支持**事件函数**，不支持 Web 函数。

## 操作步骤

### 步骤1：创建修改请求体函数

对于部署在公网或内网 VPC 的认证函数，可省略该步骤。

1. 登录 云函数控制台 。
2. 在左侧导航栏，单击**函数服务**，进入**函数列表**页面。
3. 单击页面左上角的**新建**，新建一个函数，内容为修改请求体。

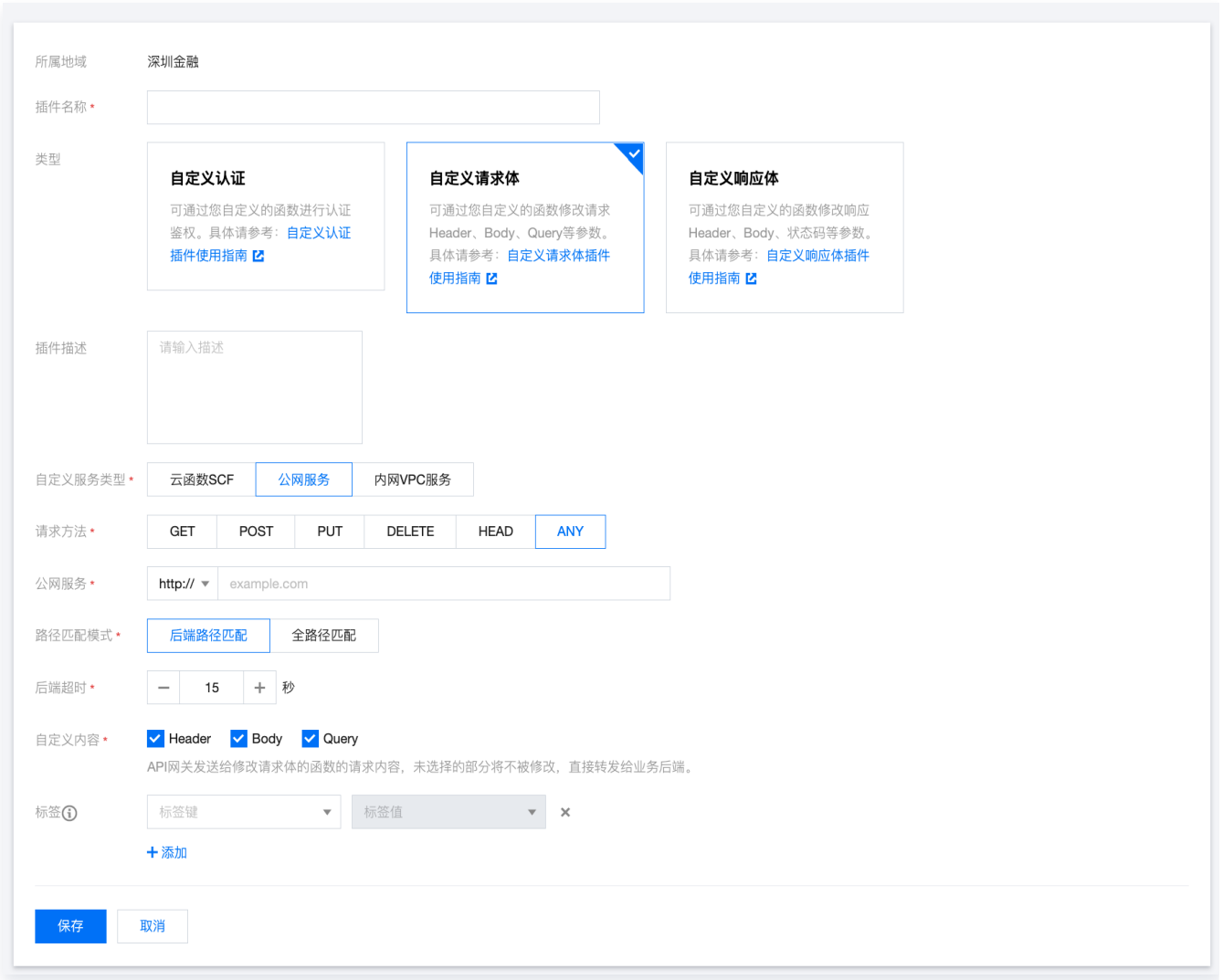
### 步骤2：创建自定义请求体插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件** > **自定义插件**，进入**自定义插件列表**页面。
3. 单击页面左上角的新建，新建一个自定义请求体插件。
   - 对于部署在云函数的认证服务，创建自定义请求体插件时需要填写的数据如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择函数 | 必填 | 选择修改请求体的函数所在的命名空间、名称和版本。 |
| 后端超时 | 必填 | 设置 API 网关转发到修改请求体的函数的后端超时时间，超时时间的最大限制为30分钟。在 API 网关调用修改请求体的函数，未在超时时间内获得响应时，API 网关将终止此次调用，并返回相应的错误信息。 |
| 自定义内容 | 必填 | 设置 API 网关发送给修改请求体的函数的请求内容，支持选择 Header、Body、Query。未选择的请求内容部分将不被修改，直接转发给业务后端。 |
| Base64 编码 | 必填 | 是否将请求内容 Base64 编码后再转发给修改请求体的函数，一般适用于请求内容是二进制的情况。 |

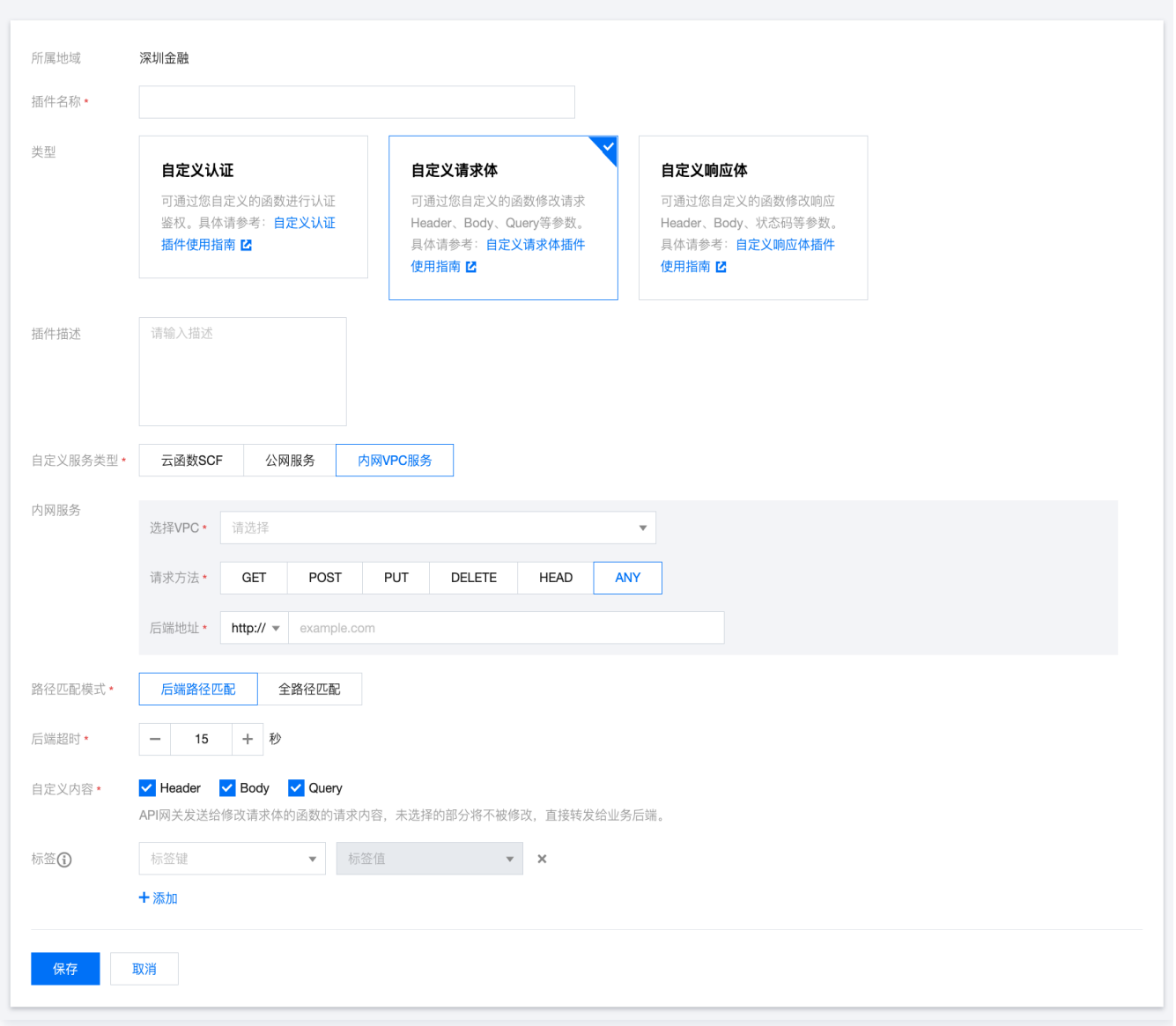
- 对于部署在公网的认证服务，创建自定义认证插件时需要填写的数据如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 请求方法 | 必填 | 请求自定义请求体函数的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |
| 公网服务 | 必填 | 请求体服务访问地址，支持 HTTP 和 HTTPS 协议。 |
| 路径匹配模式 | 必填 | 支持后端路径匹配和全路径匹配两种方式。<br>• 后端路径匹配：直接使用配置的路径请求服务。<br>• 全路径匹配：使用去除请求路径的路径请求服务，如 API 路径配置为 /a/，请求路径为 /a/b，开启全路径匹配后，传输给服务的为 /b。 |



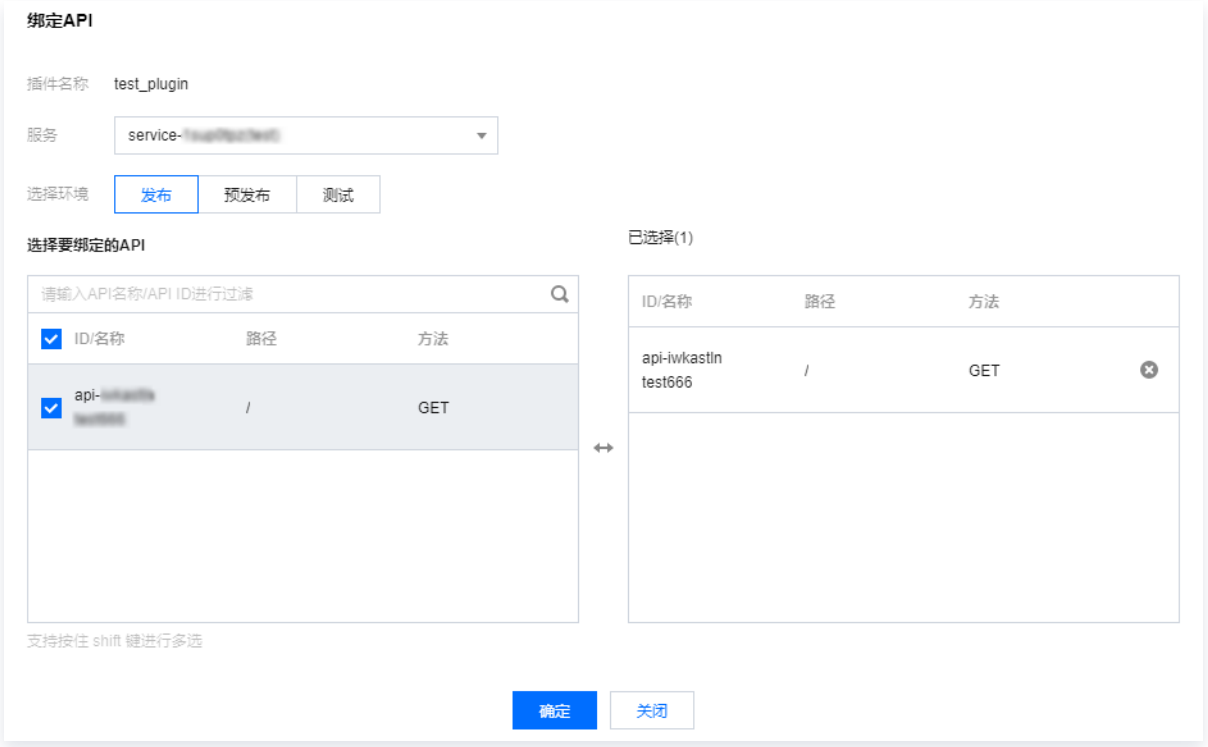

- **对于部署在内网 VPC 的认证服务，创建自定义认证插件时需要填写的数据如下：**

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择 VPC | 必填 | 选择自定义请求改写服务所属的 VPC。 |
| 请求方法 | 必填 | 请求自定义请求改写服务的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |
| 后端地址 | 必填 | 自定义请求体服务访问地址，支持 HTTP 和 HTTPS 协议。 |




## 步骤3：绑定 API

1. 在列表中选中 步骤2 创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

# 自定义请求体云函数的编写方法

## 返回值定义

API 网关自定义请求体插件需要接受自定义请求体云函数返回特定格式的 Response，具体格式如下：

```
{
  "replace_headers":{
    "header1":"header1-value",
    "header2":"header2-value"
  },
  "remove_headers":[
    "header3",
    "header4"
  ],
  "replace_body":"hello",
  "replace_querys":{
    "query1":"query1-value",
    "query2":"query2-value"
  },
  "remove_querys":[
    "query3",
    "query4"
  ]
```

```
}
```

## Python Demo

使用 Python 语言修改请求体的函数的编写方法请您参见 自定义请求体 Python Demo 。

## Java Demo

```
package com.example.demo;

import com.google.gson.JsonArray;
import com.google.gson.JsonObject;
import
com.qcloud.services.scf.runtime.events.APIGatewayProxyRequestEvent;

public class Demo {

    public String mainHandler(APIGatewayProxyRequestEvent request) {
        System.out.println("helloworld");
        System.out.println(request.getHttpMethod());
        JsonObject resp = new JsonObject();
        headerHandler(request, resp);
        headerQuery(request, resp);
        headerBody(request, resp);
        return resp.toString();
    }

    private void headerHandler(APIGatewayProxyRequestEvent request,
JsonObject resp) {
        JsonObject replace_headers = new JsonObject();
        JsonArray remove_headers = new JsonArray();
        // 示例：替换或新增 header1 header2
        replace_headers.addProperty("header1", "header1-value");
        replace_headers.addProperty("header2", "header2-value");

        // 示例：删除 header3
        remove_headers.add("header3");
        resp.add("replace_headers", replace_headers);
        resp.add("remove_headers", remove_headers);
    }

    private void headerQuery(APIGatewayProxyRequestEvent request,
JsonObject resp) {
```

```
        JsonObject replace_querys = new JsonObject();
        JsonArray remove_querys = new JsonArray();

        // 示例：替换或新增 query1 query2
        replace_querys.addProperty("query1", "query1-value");
        replace_querys.addProperty("query2", "query2-value");

        // 示例：删除 header3
        remove_querys.add("query3");
        resp.add("replace_querys", replace_querys);
        resp.add("remove_querys", remove_querys);
    }

    private void headerBody(APIGatewayProxyRequestEvent request,
JsonObject resp) {
        resp.addProperty("replace_body", "{'name':'Yagr'}");
    }
}
```

## PluginData

```
{
    "endpoint_timeout":15, // 后端超时时间，单位秒，合法值：0 ~ 60 秒
    "func_name":"test_name", // 自定义函数名称
    "func_namespace":"test_namespace", // 自定义函数命名空间
    "func_qualifier":"$LATEST", // 自定义函数版本
    "is_base64_encoded":true, // 是否将请求内容 Base64 编码后再转发给修改请求体
的函数
    "is_send_req_body":true, // 是否将请求的Body内容发送到函数
    "is_send_req_headers":true, // 是否将请求的Header内容发送到函数
    "is_send_req_querys":true, // // 是否将请求的Query内容发送到函数
    "user_id":1253970226 // appid
}
```

## 注意事项

每次将自定义插件绑定到一个网关 API 时，相当于为修改请求体的函数创建了一个该网关 API 的触发器。在 SCF 侧删除触发器，相当于把插件和 API 解绑。

## 自定义服务类型规则

自定义插件的自定义服务类型，支持绑定不同实例上的服务 API，规则请参见 插件概述-已支持自定义插件类型 。

## 示例代码

自定义请求体时，HTTP 服务的示例代码如下：

### Python 2.7

```
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlparse, parse_qs
import json

PORT_NUMBER = 8022


def handle_header(header, response):
    replace_headers = {}  # 需要替换或者新增的header
    remove_headers = []  # 需要删除的header

    header_parameters = header  # 客户端原始请求的header内容
    if header_parameters is None or not isinstance(header_parameters,
dict):
        print("Invalid event.headerParameters")
        return

    # 示例：替换或新增 header1
    replace_headers["header1"] = "header1"

    # 示例：删除 header2
    if header_parameters.get("header2") is not None:
        remove_headers.append("header2")

    # replace_headers 和 remove_headers 至少有一个非空
    if len(replace_headers) < 1 and len(remove_headers) < 1:
        print("Invalid custom request headers")
        return

    response["replace_headers"] = replace_headers
    response["remove_headers"] = remove_headers


def handle_body(body, response):
    replace_body = ""  # 替换之后的body内容
    # 示例：替换body
    if body is not None:
```

```
        replace_body = "hello world"

    # replace_body 必须为字符串类型
    if not isinstance(replace_body, str):
        print("Invalid custom request body")
        return

    response["replace_body"] = replace_body


def handle_query(query, response):
    replace_query = {}  # 需要替换或者新增的query
    remove_query = []  # 需要删除的query

    query_parameters = query
    if query_parameters is None or not isinstance(query_parameters,
dict):
        print("Invalid event.queryStringParameters")
        return

    # 示例：替换或新增 query1
    replace_query["query1"] = "query1"

    # 示例：删除原始请求中的 query2
    if query_parameters.get("query2") is not None:
        remove_query.append("query2")

    # replace_query 和 remove_query 至少有一个非空
    if len(replace_query) < 1 and len(remove_query) < 1:
        print("Invalid custom request query")
        return

    response["replace_query"] = replace_query
    response["remove_query"] = remove_query


class MyHandler(BaseHTTPRequestHandler):

    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
```

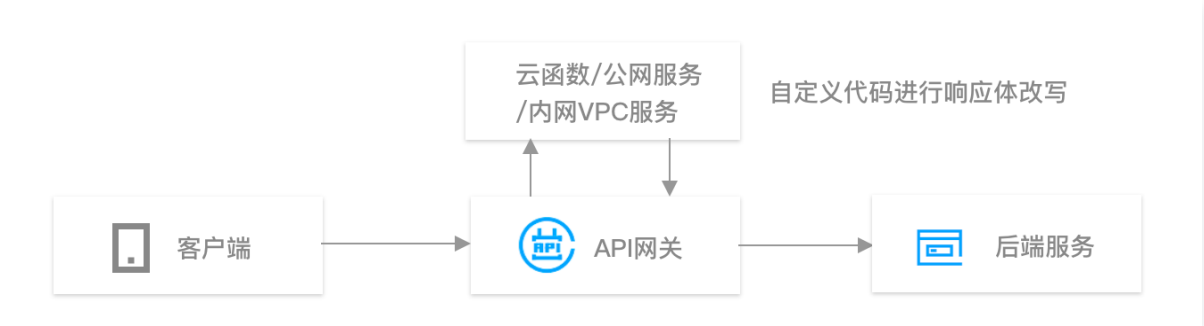
```
    def do_POST(self):
        response = {}
        try:
            request_body =
self.rfile.read(int(self.headers.getheader("Content-length")))
        except:
            request_body = "no body."

        # 重写header
        handle_header(self.headers.__dict__, response)

        # 重写query
        parsed_url = urlparse(self.path)
        request_query = parse_qs(parsed_url.query)
        handle_query(request_query, response)

        # 重写body
        handle_body(request_body, response)

        self._set_headers()
        self.wfile.write(json.dumps(response).encode('utf-8'))
        return


try:
    server = HTTPServer(('', PORT_NUMBER), MyHandler)
    print(f'Started HTTP server on port {PORT_NUMBER}')

    # Wait forever for incoming http requests
    server.serve_forever()

except KeyboardInterrupt:
    print('^C received, shutting down the web server')
    server.socket.close()
```

# 自定义响应体

最近更新时间：2023-11-10 10:05:52

## 操作场景

API 网关响应给客户端响应体中包含很多字段，如果您需要修改响应体内容，可以通过自定义响应体插件实现。自定义请求体插件作用在响应过程中， 响应内容改写服务可部署在云函数、公网、或内网 VPC 上。业务后端处理完请求报文后，会将响应体传递给 API 网关。API 网关接收到响应内容后，API 网关会将响应内容转发到响应体修改服务中，响应体内容修改完成后，将修改后的响应体响应给 API 网关，API 网关再将修改后的响应体转发给业务后端。




## 前提条件

目前已支持云函数、公网、或内网 VPC ，故满足三者任意一个，均可使用。

1. 已开通 云函数 服务。
2. 已存在公网服务。
3. 已开通 私有网络 服务。

> **说明：**
> 自定义请求体插件目前仅支持**事件函数**，不支持 Web 函数。

## 操作步骤

以下以云函数为例。

### 步骤1：创建修改响应体的函数

1. 登录 云函数控制台 。
2. 在左侧导航栏，单击**函数服务**，进入函数列表页。
3. 单击页面左上角的**新建**，新建一个函数，内容为修改响应体。

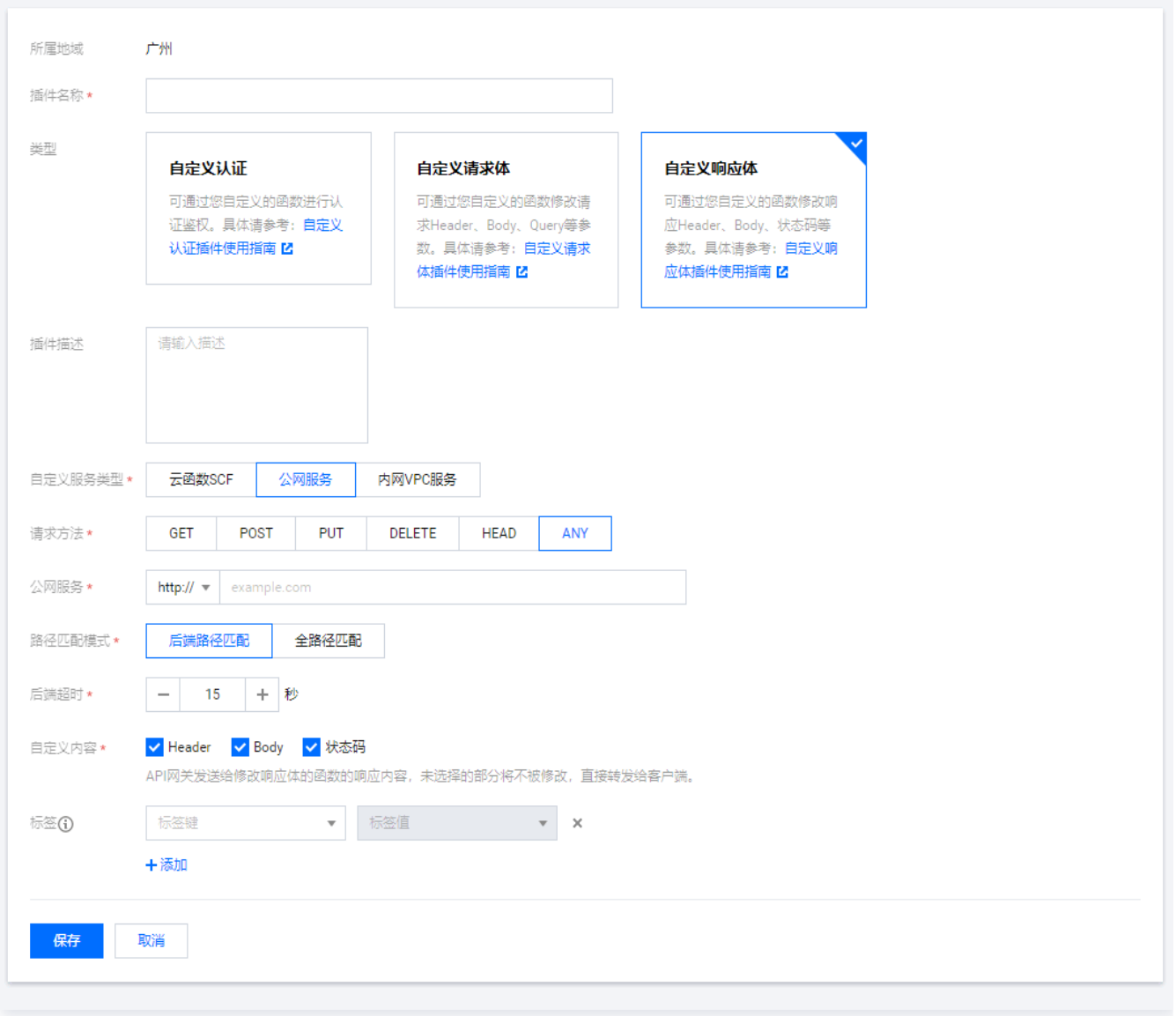
### 步骤2：创建自定义响应体插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件** > **自定义插件**，进入自定义插件列表页。
3. 单击页面左上角的**新建**，新建一个自定义响应体插件。
   - 对于部署在云函数的认证服务，创建自定义响应插件时需要填写的数据如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择函数 | 必填 | 选择修改响应体的函数所在的命名空间、名称和版本。 |
| Base64 编码 | 必填 | 业务后端响应内容传到云函数中时是否要经过 Base64 编码，一般适用于响应内容是二进制的情况。 |
| 后端超时 | 必填 | 设置 API 网关转发到修改响应体的函数的后端超时时间，超时时间的最大限制为30分钟。在 API 网关调用修改响应体的函数，未在超时时间内获得响应时，API 网关将终止此次调用，并返回相应的错误信息。 |
| 自定义内容 | 必填 | 设置 API 网关发送给修改响应体的函数的响应内容，支持选择 Header、Body、状态码。未选择的响应内容部分将不被修改，直接转发给客户端。 |

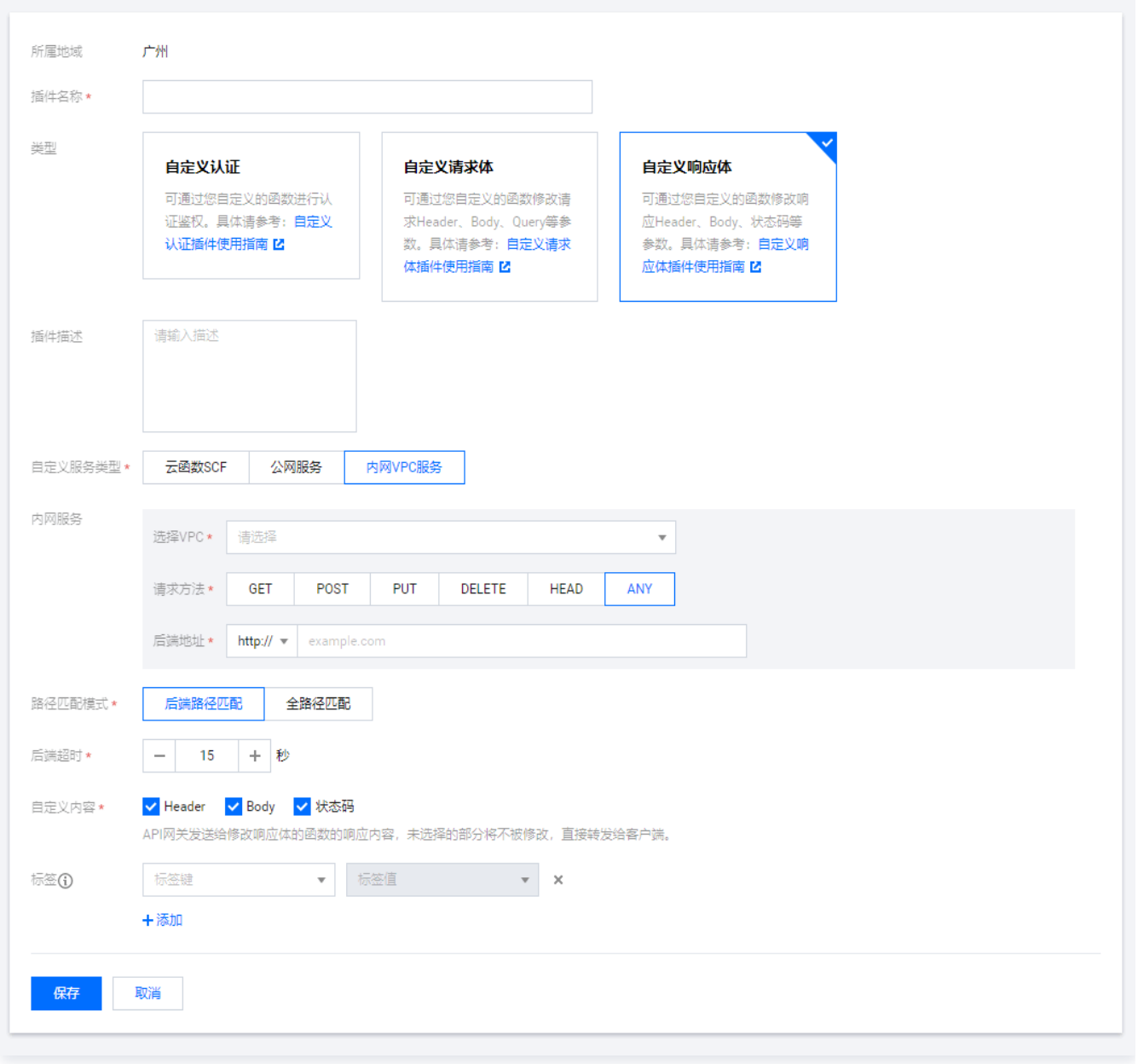
   - 对于部署在公网的认证服务，创建自定义响应插件时需要填写的数据如下：

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 请求方法 | 必填 | 请求自定义响应体函数的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |
| 公网服务 | 必填 | 自定义响应体改写服务访问地址，支持 HTTP 和 HTTPS 协议。 |
| 路径匹配模式 | 必填 | 支持后端路径匹配和全路径匹配两种方式。<br>• 后端路径匹配：直接使用配置的路径请求服务。<br>• 全路径匹配：使用去除请求路径的路径请求服务，如 API 路径配置为 /a/，请求路径为 /a/b，开启全路径匹配后，传输给服务的为 /b。 |



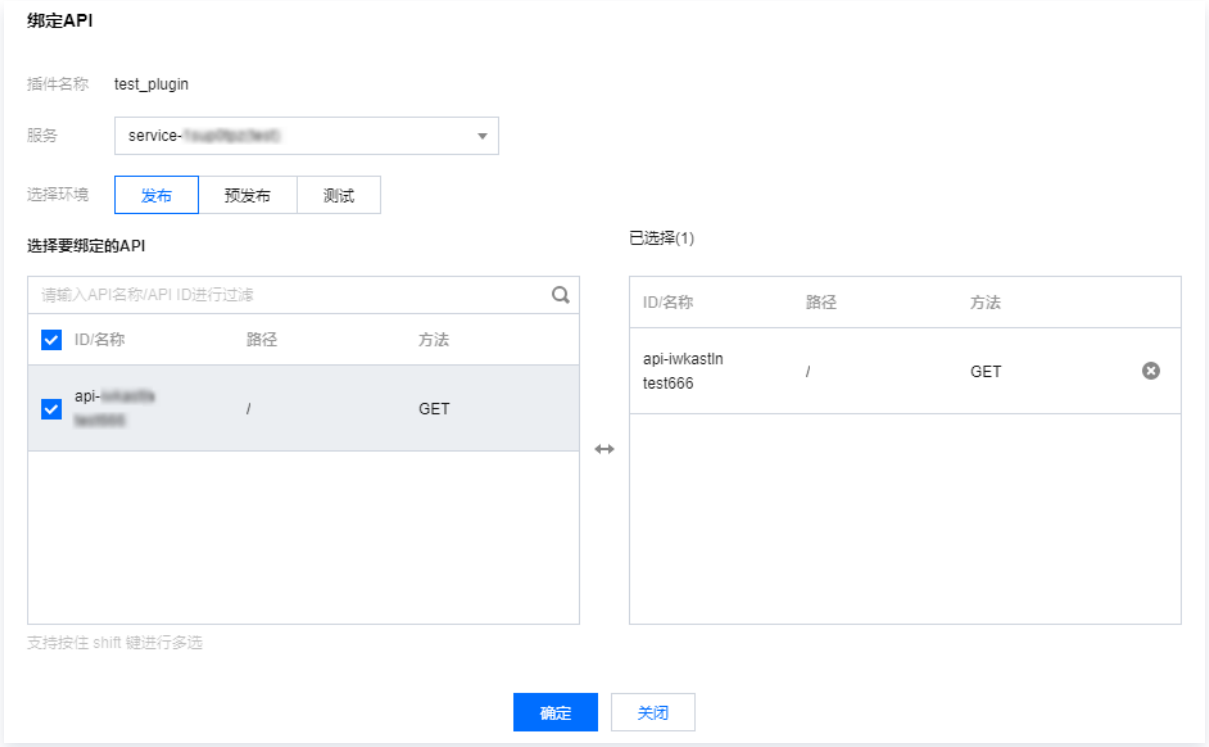

- **对于部署在内网 VPC 的认证服务，创建自定义响应插件时需要填写的数据如下：**

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 选择 VPC | 必填 | 选择响应体改写服务所属的 VPC。 |
| 请求方法 | 必填 | 请求响应体改写函数的方法，支持 GET、POST、PUT、DELETE、HEAD、ANY。 |
| 后端地址 | 必填 | 响应体改写服务访问地址，支持 HTTP 和 HTTPS 协议。 |




## 步骤3：绑定 API

1. 在插件列表中选中 步骤2 创建好的插件，单击操作列的**绑定 API**。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。




3. 单击**确定**，即可将插件绑定到 API，此时插件的配置已经对 API 生效。

## pluginData

```
{
    "endpoint_timeout":15, // 后端超时时间，单位秒，合法值：0 ~ 60 秒
    "func_name":"test_name", // 自定义函数名称
    "func_namespace":"test_namespace", // 自定义函数命名空间
    "func_qualifier":"$LATEST", // 自定义函数版本
    "is_base64_encoded":true, // 业务后端响应内容传到云函数中时是否要经过Base64
编码
    "is_custom_status":true, // 是否将响应的状态码内容发送到函数
    "is_custom_headers":true, // 是否将响应的Header内容发送到函数
    "is_custom_body":true, // 是否将响应的Body内容发送到函数
    "user_id":1253970226 // appid
}
```

## 注意事项

- 每次将自定义插件绑定到一个网关 API 时，相当于为修改响应体的函数创建了一个该网关 API 的触发器。在 SCF 侧删除触发器，相当于把插件和 API 解绑。
- 自定义响应体插件优先级低于所有作用在请求过程中的插件。
- 自定义响应体插件绑定到后端为 Mock、微服务平台 TSF 的 API 上时，将不生效。
- 自定义响应体插件不支持 HTTP2 协议。

- 自定义响应体插件不支持后端返回经 gzip 压缩后的响应体。

## 自定义服务类型规则

自定义插件的自定义服务类型，支持绑定不同实例上的服务 API，规则请参见 插件概述-已支持自定义插件类型 。

## 示例代码

自定义响应体时，HTTP 服务的示例代码如下：

Python 2.7

```
from http.server import BaseHTTPRequestHandler, HTTPServer
from urllib.parse import urlparse, parse_qs
import json

PORT_NUMBER = 8022


def handle_header(header, response):
    replace_headers = {}  # 需要替换或者新增的header
    remove_headers = []  # 需要删除的header

    # 示例：替换或新增 header1
    replace_headers["header1"] = "header1"

    # 示例：删除 X-Api-Serviceid
    remove_headers.append("X-Api-Serviceid")

    response["replace_headers"] = replace_headers
    response["remove_headers"] = remove_headers


def handle_body(body, header, response):
    response["replace_body"] = "replace_body"

    # 是否需要对replace_body 进行base64解码
    response["is_base64_encoded"] = header.get("is_base64_encoded")


def handle_status(header, response):
    status = header.get("status")

    # 示例：后端错误处理
```

```
    if status is not None and status != 200:
        response["replace_status"] = 200
        response["replace_body"] = "custom response, upstream_status is
" + str(status)
        response["is_base64_encoded"] = False


class MyHandler(BaseHTTPRequestHandler):

    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()

    def do_POST(self):
        response = {}
        try:
            request_body =
self.rfile.read(int(self.headers.getheader("Content-length")))
        except:
            request_body = "no body."

        # 重写header
        handle_header(self.headers.__dict__, response)

        # 重写status
        handle_status(self.headers.__dict__, response)

        # 重写body
        handle_body(request_body, self.headers.__dict__, response)

        self._set_headers()
        self.wfile.write(json.dumps(response).encode('utf-8'))
        return

    def do_GET(self):
        response = {
            'api-auth': True
        }

        # header认证
        header_auth_key = self.headers.get("header-auth")
        if header_auth_key == "apigw":
```

```
            response['api-auth'] = False

        # query认证
        parsed_url = urlparse(self.path)
        query_params = parse_qs(parsed_url.query)
        query_auth_key = query_params.get("query-auth", [None])[0]
        if query_auth_key == "apigw":
            response['api-auth'] = False

        self._set_headers()
        self.wfile.write(json.dumps(response).encode('utf-8'))
        return


try:
    server = HTTPServer(('', PORT_NUMBER), MyHandler)
    print(f'Started HTTP server on port {PORT_NUMBER}')

    # Wait forever for incoming http requests
    server.serve_forever()

except KeyboardInterrupt:
    print('^C received, shutting down the web server')
    server.socket.close()
```

# 防重放插件

最近更新时间：2023-06-20 09:43:42

## 操作场景

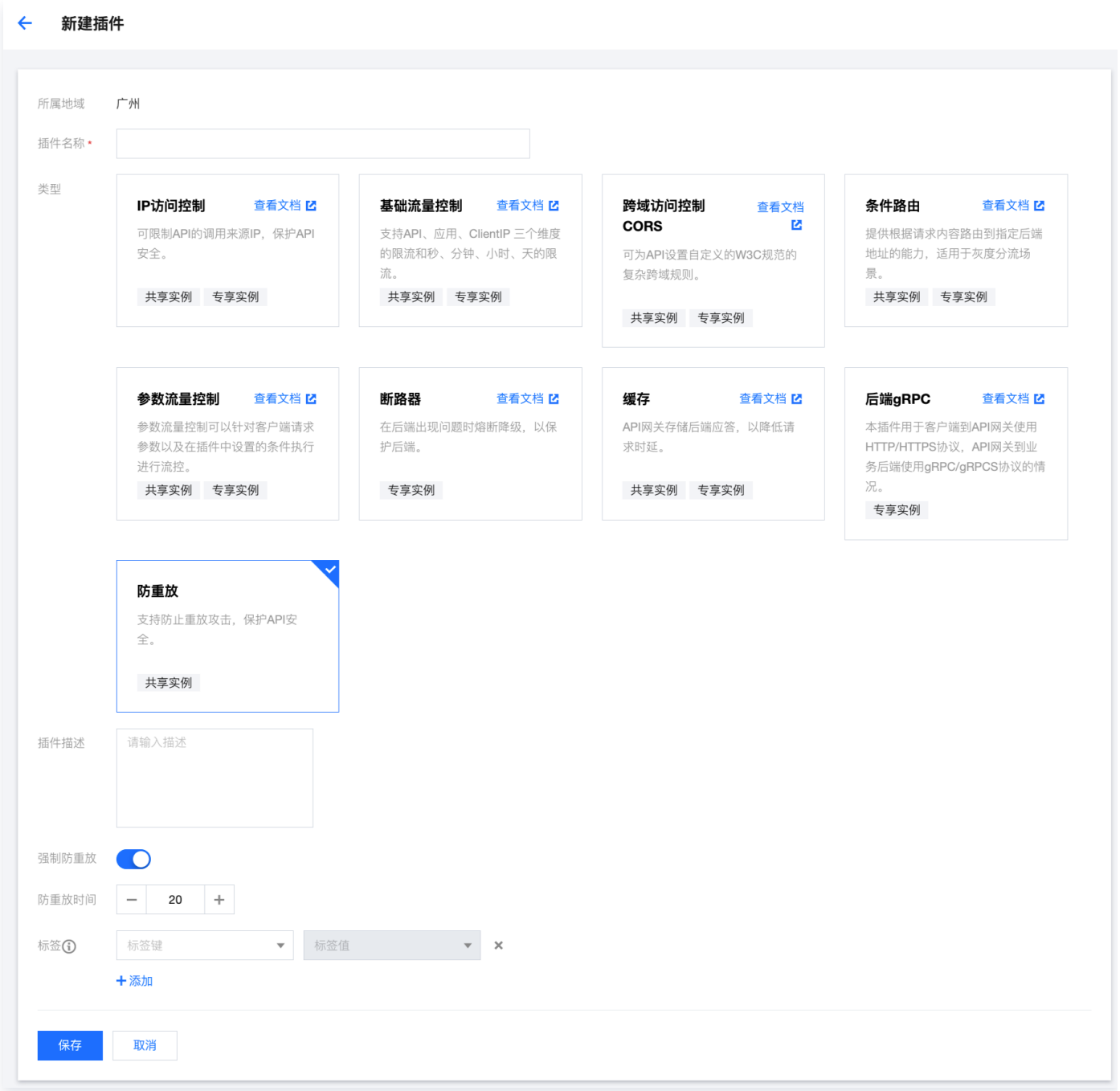
防重放插件是 API 网关提供的针对重放攻击进行 API 保护的能力。您可以创建防重放插件并绑定到 API 生效，以保护您的后端服务。

## 操作步骤

### 步骤1：创建插件

1. 登录 API 网关控制台 。
2. 在左侧导航栏，单击**插件 > 系统插件**，进入**系统插件列表**页面。
3. 单击页面左上角的**新建**，选择插件类型为**防重放**，新建一个防重放插件。

| 参数 | 是否必填 | 说明 |
| --- | --- | --- |
| 强制防重放 | 否 | 是否开启强制防重放，默认关闭，如需开启，请求头部须添加 x-apigw-nonce 字段。 |
| 防重放时间 | 否 | 防重放有效时间，默认900，单位s，范围 1-1800。 |




## 步骤2：绑定 API 并生效

1. 在列表中选中刚刚创建好的插件，单击操作列的绑定 API。
2. 在绑定 API 弹窗中选择服务和环境，并选择需要绑定插件的 API。

## PluginData

```
{
```

```
    "force_nonce":true, // 是否开启强制防重放，默认关闭，如需开启，请求头部必须添加x-apigw-nonce字段
    "nonce_ttl":1       //  Nonce的有效时间， 默认900，单位s，取值范围 1-1800
}
```