

TESLA使用文档（PDF版）

- TESLA介绍
 - 系统简介
- 管理控制台简介
- 后端服务发布到TESLA
 - 1、发布说明
 - 2、后端order-service服务介绍
 - 3、配置到TESLA
- 插件详细介绍
 - API配置-请求头配置
 - API配置-跨域配置
 - API配置-路由信息
 - API配置-鉴权信息
 - API配置-限流配置
 - Endpoint配置-Groovy脚本插件
 - Endpoint配置-Mock插件
 - Endpoint配置-Rpc路由转换插件
 - Endpoint配置-Url重写插件
 - Endpoint配置-创建token插件
 - Endpoint配置-执行上传jar包插件
 - Endpoint配置-查询聚合插件
 - Endpoint配置-消除鉴权插件
 - Endpoint配置-签名校验插件
 - Endpoint配置-缓存结果插件
 - Endpoint配置-请求报文转换插件
- 调用方接入管理
 - 接入简介
 - 接入列表
 - 新增接入系统
- 灰度规则配置
 - 灰度简介
 - 配置介绍
 - 配置示例

TESLA介绍

系统简介

tesla为公司内部系统提供统一的，安全的入口服务，并通过多种可配置的插件灵活的实现自己需要的功能。

tesla由tesla-ops(tesla管理控制台)和tesla-gateway(tesla网关)两个服务组成，开发和使用人员在tesla-ops上进行服务发布和调用方接入配置。

- 服务发布：通过在tesla-ops的**API管理**菜单处进行配置，可将自己的服务通过tesla-gateway发布出去，其他系统可通过tesla-gateway访问到后端服务。
- 接入配置：外部系统想通过tesla-gateway调用已发布的服务，可通过tesla-ops的**接入管理**菜单处进行配置，设置访问权限等功能。

tesla还与summerframework结合，通过gateway为入口，实现了整个调用链的灰度节点选择，summerframework文档见.....。

本文档主要介绍tesla-ops的配置使用，了解并理解tesla-ops的配置后，就很容易明白如果通过系统调用tesla-gateway。

可使用docker一键启动整个演示环境，具体启动方式请参考github。

管理控制台简介

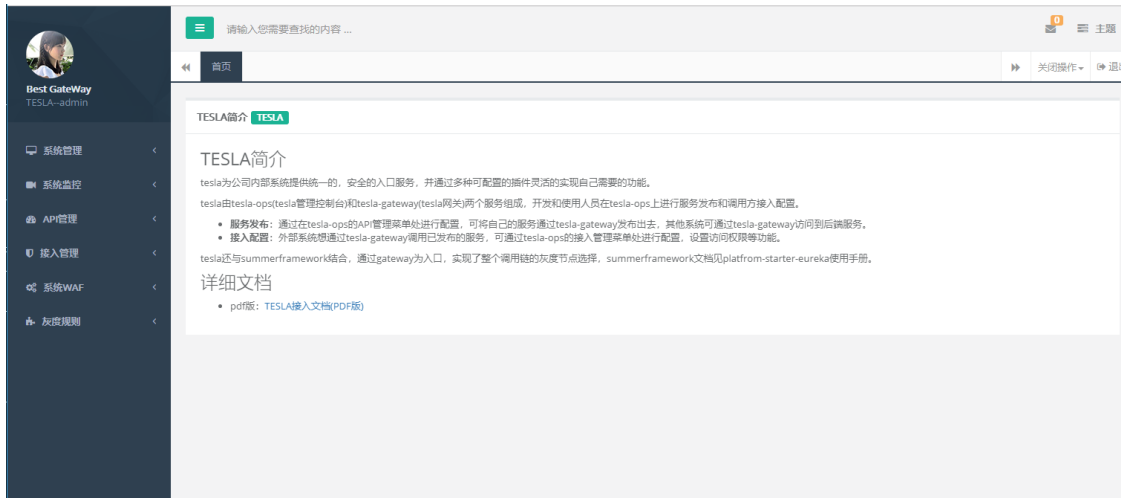
通过访问对应环境的tesla-ops地址访问管理控制台，如dev环境：<https://localhost:8080.com>



可看到上面的登录页面，ops是一个类似多租户的控制台，可以管理多个不同的gateway，可通过下拉框选择不同的gateway。

用户登录可使用 admin/Password@1 账户登录，登录成功后会进入如下页面，当前开放给普通用户主要有如图所示的三块配置：

- **API管理**：用于后端服务将服务配置发布在gateway，通过gateway暴露出去。
- **接入管理**：外部系统希望访问gateway上发布的服务时需要在此处申请一个接入key来获得访问权限。
- **灰度规则**：与summerframework结合，实现全调用链路的灰度节点选择，此处为配置规则的操作页面。



后端服务发布到TESLA

1、发布说明

本文将通过一个配置示例说明后端服务发布到TESLA的流程，并介绍各个插件的功能，演示环境可通过docker一键启动，可以点开配置配合了解。

2、后端order-service服务介绍

该后端服务可见源码tesla\tesla-sample\tesla-backend-sample项目。

通过docker命令启动后有一个该服务的容器，暴露在本地8902端口，该服务提供三种接口，可自行点开下面url查看效果，注意不要修改参数，目前只预定义了三个cusid：

查询客户名称

```
GET: http://localhost:8902/order-service/queryName?cusid=1
GET: http://localhost:8902/order-service/queryName?cusid=2
GET: http://localhost:8902/order-service/queryName?cusid=3
```

POST: <http://localhost:8902/order-service/queryName/1>
POST: <http://localhost:8902/order-service/queryName/2>
POST: <http://localhost:8902/order-service/queryName/3>

查询客户所在省份

GET/POST: <http://localhost:8902/order-service/queryProvinces?cusid=1>
GET/POST: <http://localhost:8902/order-service/queryProvinces?cusid=2>
GET/POST: <http://localhost:8902/order-service/queryProvinces?cusid=3>

查询客户订单

GET/POST: <http://localhost:8902/order-service/queryOrder?cusid=1>
GET/POST: <http://localhost:8902/order-service/queryOrder?cusid=2>
GET/POST: <http://localhost:8902/order-service/queryOrder?cusid=3>

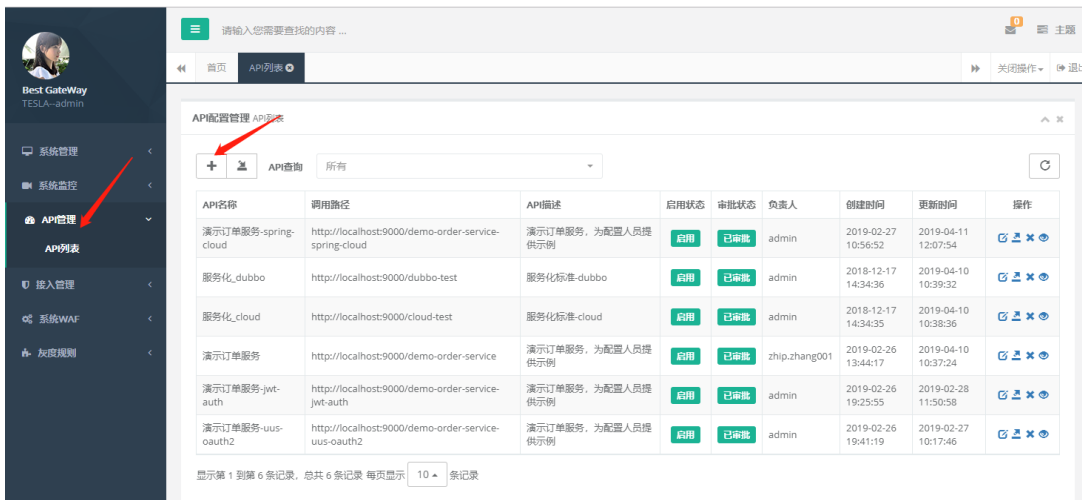
同时，该order-service还注册到了dev环境的eureka，请在<http://eureka.springcloud.cn/>搜索TESLA-BACKEND-SAMPLE服务即可看到。

3、配置到TESLA

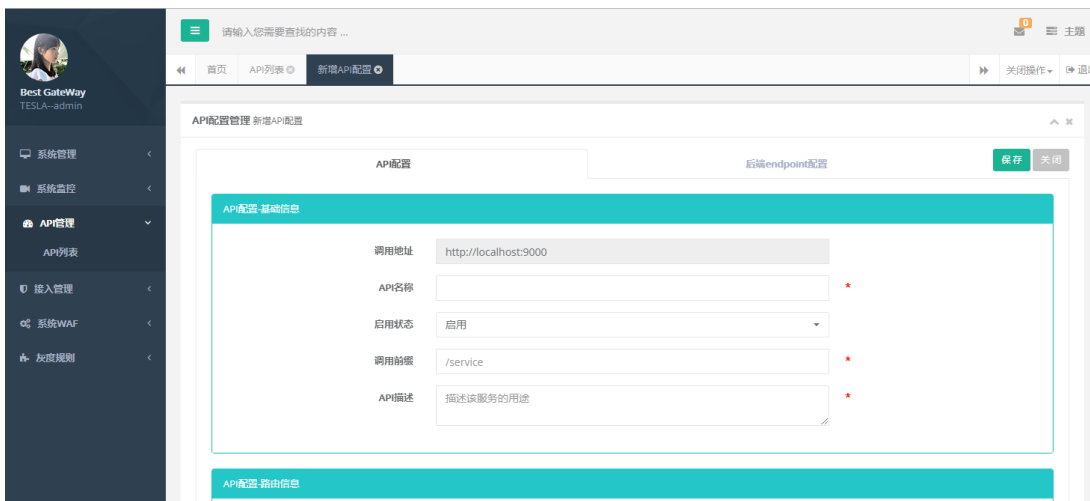
3.1、简单配置

将一个服务通过Tesla发布出去，最简单的情况下只需要配置一个**基础信息**，**路由信息**和一个**endpoint**。

点击下方按钮，进入新增API配置页面：



新增API配置页面如下图：



主要分为API配置和后端Endpoint配置两大块，Endpoint可以理解为后端服务的一个接口，在TESLA中，一个Endpoint可以精确针对后端服务的一个接口，也可以通过通配符模糊匹配后端的一组真实接口，API是一组Endpoint的聚合，可以认为是后端的一个服务，API的配置会适用于该API中的所有Endpoint。

下面我们进行配置来讲order-service发布出去。

3.1.1、配置API配置-基础信息

API配置-基础信息

调用地址	http://localhost:9000/demo-order-service	
API名称	演示订单服务	*
启用状态	启用	
调用前缀	/demo-order-service	*
API描述	演示订单服务，为配置人员提供示例	*

此处我们填写了API名称，调用前缀，API描述。

- **API名称**：展示用的名称，简单描述下该服务。
- **调用前缀**：调用前缀指的是该服务在TESLA上绑定的path前缀，如我图中配置的/demo-order-service，那么当请求url的path以/demo-order-service开头时，就会路由到该API。
- **API描述**：此处可用简短的语句描述该服务的功能，并填上对应负责人的信息。

3.1.2、配置API配置-路由信息

API配置-路由信息

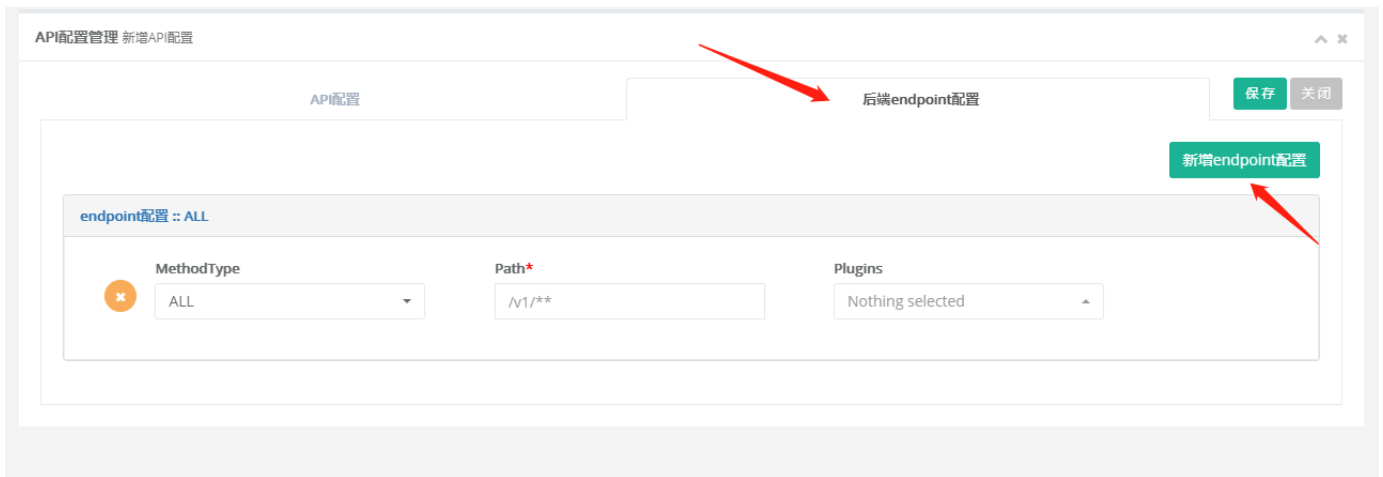
路由类型	直接路由	
目标地址	tesla-backend-sample:8902	*
	<small>eg. backend-server-host:8080 默认端口80或443可不填端口</small>	
地址前缀	/order-service	
HTTPS	否	
自签证书	选择文件	
	<small>如果服务器的证书为自签证书，请上传crt格式证书</small>	

此处路由类型选择了直接路由，并在下面填写了目标地址和地址前缀。

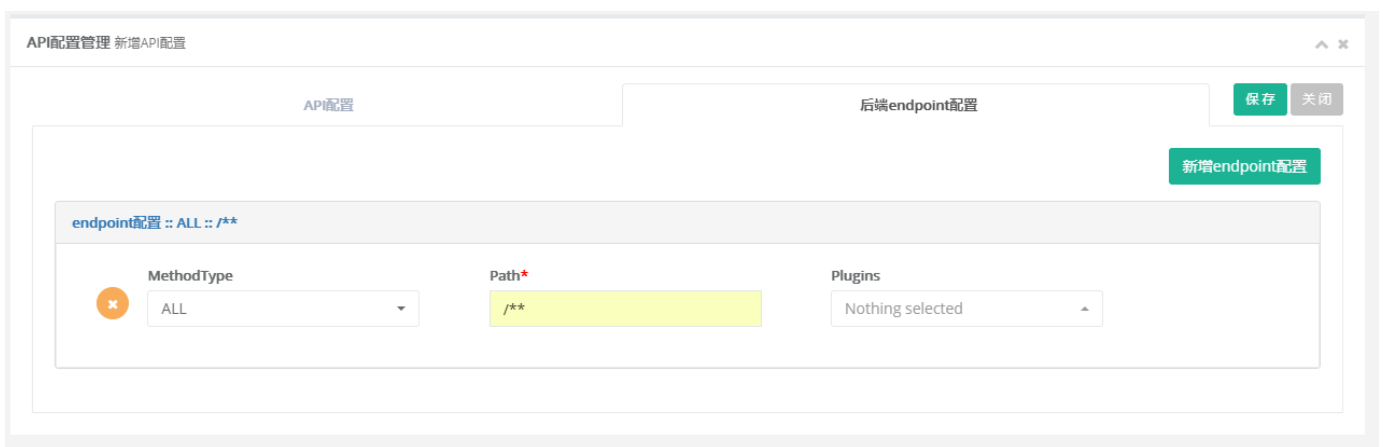
- **目标地址**：填写后端服务的访问地址，注意没有http://|https://，也没有path，只有ip:port或域名，此处填写的是：tesla-backend-sample:8902
- **调用前缀**：tesla将请求转发给后端服务时会统一拼上这个前缀，如此处因为后端的order-service统一是以/order-service开始的，所以在此处配置了调用前缀为/order-service。

其他路由信息的配置详细介绍请见API配置-路由信息。

3.1.3、配置后端Endpoint

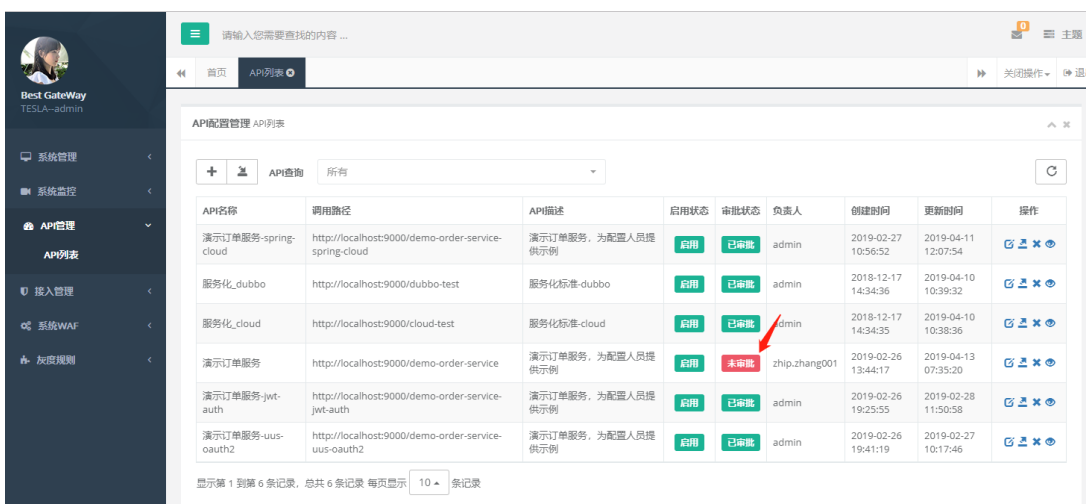


点击后端endpoint配置选项卡后点击右侧新增endpoint配置，下面会出现一个配置卡片，此处我们选择methodType为ALL，Path为/**，不选择任何Plugins，如下图




3.1.4、管理员审批

点击保存后，会跳回到列表页面，如下图



我们刚才配置的服务已经出现在列表中，但是此时的审批状态是未审批，需要联系管理员进行检查审批，可使用管理员账号进行审批。

审批完成后，如下图

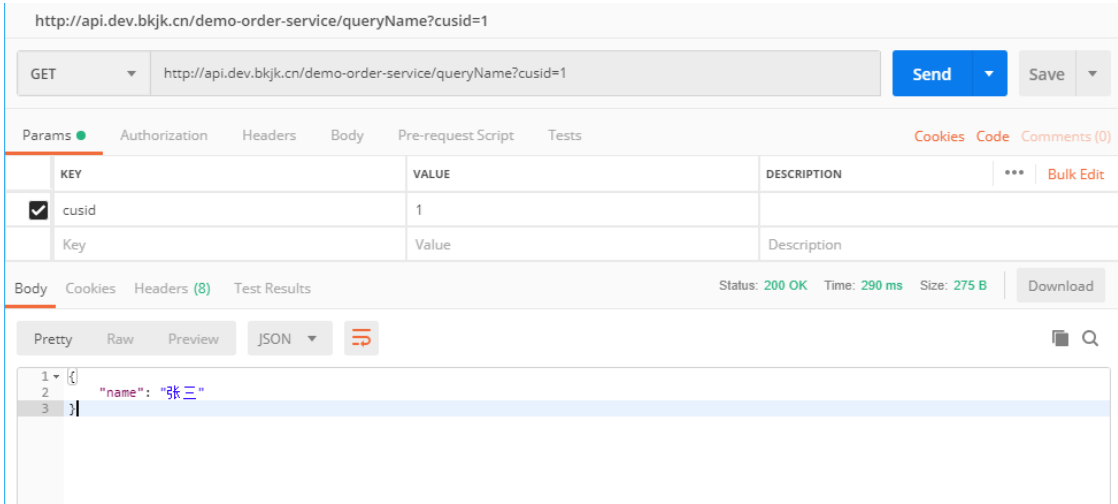
服务化_cloud	http://localhost:9000/cloud-test	服务化标准-cloud	启用	已审批	admin	2018-12-17 14:34:35	2019-04-10 10:38:36	   
演示订单服务	http://localhost:9000/demo-order-service	演示订单服务，为配置人员提供示例	启用	已审批	zhip.zhang001	2019-02-26 13:44:17	2019-04-13 07:35:20	   

调用路径就是当前环境下tesla-gateway的访问路径。

3.1.5、调用测试

此时我们通过Postman访问 GET：http://localhost:9000/demo-order-service/queryName?cusid=1，如下图，

可以看到实际请求结果与 GET：http://localhost:8902/order-service/queryName?cusid=1结果一致。



转发关系如下：

查询客户名称

GET: http://localhost:9000/demo-order-service/queryName?cusid=1 --> http://localhost:8902/order-service/queryName?cusid=1
GET: http://localhost:9000/demo-order-service/queryName?cusid=2 --> http://localhost:8902/order-service/queryName?cusid=2
GET: http://localhost:9000/demo-order-service/queryName?cusid=3 --> http://localhost:8902/order-service/queryName?cusid=3
POST: http://localhost:9000/demo-order-service/queryName/1 --> http://localhost:8902/order-service/queryName/1
POST: http://localhost:9000/demo-order-service/queryName/2 --> http://localhost:8902/order-service/queryName/2
POST: http://localhost:9000/demo-order-service/queryName/3 --> http://localhost:8902/order-service/queryName/3

查询客户所在省份

GET/POST: http://localhost:9000/demo-order-service/queryProvinces?cusid=1 --> http://localhost:8902/order-service/queryProvinces?cusid=1
GET/POST: http://localhost:9000/demo-order-service/queryProvinces?cusid=2 --> http://localhost:8902/order-service/queryProvinces?cusid=2
GET/POST: http://localhost:9000/demo-order-service/queryProvinces?cusid=3 --> http://localhost:8902/order-service/queryProvinces?cusid=3

查询客户订单

GET/POST: http://localhost:9000/demo-order-service/queryOrder?cusid=1 --> http://localhost:8902/order-service/queryOrder?cusid=1
GET/POST: http://localhost:9000/demo-order-service/queryOrder?cusid=2 --> http://localhost:8902/order-service/queryOrder?cusid=2
GET/POST: http://localhost:9000/demo-order-service/queryOrder?cusid=3 --> http://localhost:8902/order-service/queryOrder?cusid=3

3.2、进阶API配置

通过上述的配置之后，后端服务order-service已经发布在了tesla上，但是此时tesla只是提供了路由转发的功能，下面将介绍tesla提供的针对API层面的插件，这些插件作用于整个API。

3.2.1、权限保护

当后端应用需要权限保护时，请参考API配置-鉴权信息。

3.2.2、限流保护

当后端应用需要限流保护时，请参考API配置-限流配置。

3.2.3、请求头自定义

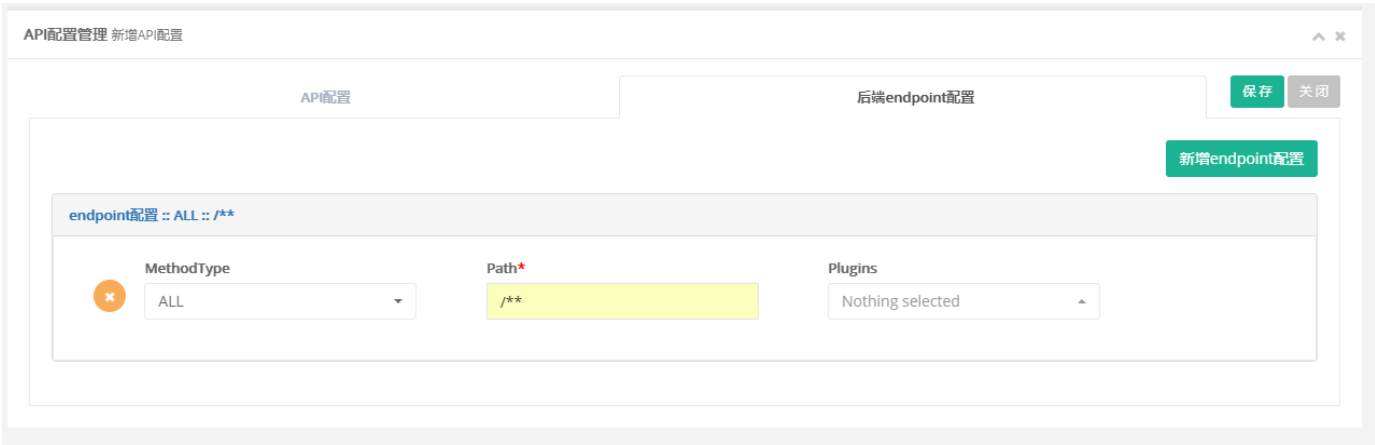
当有些请求头不需要打给后端服务，或者请求后端服务时需要额外添加一些固定的请求头，而调用方又不方便做的时候，可以使用该插件，请参考API配置-请求头配置。

3.2.4、跨域支持

当调用方跨域调用服务时，可能会出现跨域的问题，此时，可以动态的在TESLA上配置跨域，即时生效，无需修改代码，请参考API配置-跨域配置。

3.3、进阶Endpoint配置

上面我们只配置了一个通配符类型的Endpoint，如下图，其中MethodType选择了ALL，表示会匹配到所有method，Path配置为了/**，表示会匹配到所有Path。

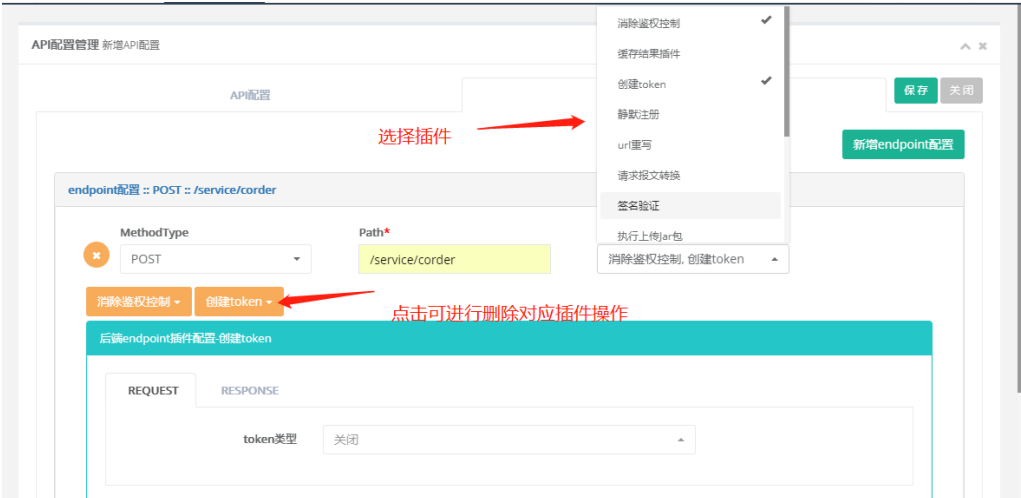


- **MethodType:** 可选择ALL或精确的method如POST.GET等，当请求的Path一致时，精确的MethodType匹配优先级高于ALL。
- **Path:** 支持精确匹配和通配符匹配，支持 *和**两种通配符，*匹配0个或多个字符，**匹配0个或多个目录，**只能放到path最后，只允许存在一个(**匹配多目录时不放在后面有歧义)，越精确匹配优先级越高。

默认情况下，path既为请求tesla的path，同时也是tesla转发给后端服务的path，如果希望打向后端的path可以差异化定制，则需要用到url重写插件。

3.3.1、Plugins（插件配置）

配置插件的基本操作如下图所示：



下面对每个插件结合demo-order-service进行详细介绍：

- Endpoint配置-AccessToken校验插件
- Endpoint配置-Groovy脚本插件
- Endpoint配置-Mock插件
- Endpoint配置-Url重写插件
- Endpoint配置-查询聚合插件
- Endpoint配置-执行上传jar包插件
- Endpoint配置-消除鉴权插件
- Endpoint配置-缓存结果插件
- Endpoint配置-Rpc路由转换插件
- Endpoint配置-创建token插件
- Endpoint配置-签名校验插件
- Endpoint配置-请求报文转换插件
- Endpoint配置-静默注册插件

插件详细介绍

API配置-请求头配置

功能介绍

- 该插件可以根据配置动态的在request和response上添加或删除指定的请求头。

配置介绍

该插件配置页面如下：

API配置-请求头配置

REQUESTRESPONSE

删除headerheader添加

添加headerkeyvalue添加

配置演示

以demo-order-service为例，在请求头配置处做了如下配置：

API配置-请求头配置

REQUESTRESPONSE

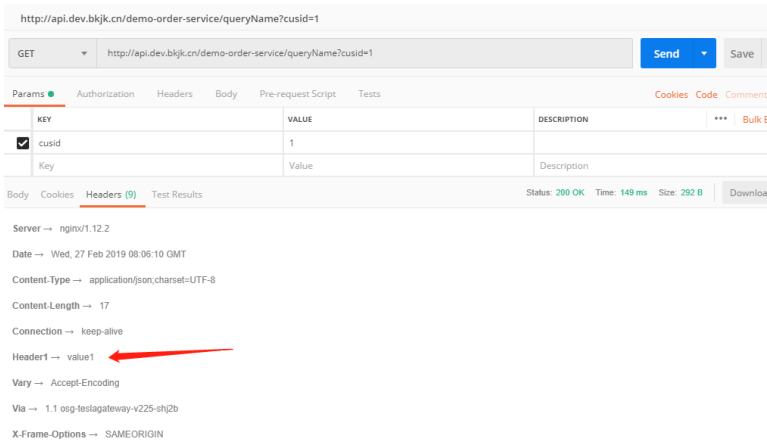
删除headerheader添加

添加headerkeyvalue添加

header1 :: value1

如图，我们在response处增加了一个增加header的配置，该配置对整个API都生效，我使用下面请求做演示，可以看到同样的请求，响应头里多出了上面的配置：

GET: <http://localhost:9000/demo-order-service/queryName?cusid=1>



此处只演示了response 添加header，通过上面的配置，可以在request和response中添加和删除header。

API配置-跨域配置

功能介绍

- 当通过页面访问不同源的后端服务出现跨域问题时，一般的做法是修改后端服务的代码或配置来支持跨域。
- 当您的后端服务是通过TESLA发布出去的时，在碰到跨域问题时可直接使用该插件进行配置，来解决跨域问题，即时动态生效，免去改后端代码再发布的烦恼。

配置介绍

该插件配置页面如下：

API配置 跨域配置

跨域支持

开启

允许跨域源

允许的请求源ip，多个可用.分割，全部允许请填写*

① 对应 Access-Control-Allow-Origin 属性

允许cookie跨域

是

① 对应 Access-Control-Allow-Credentials 属性

允许跨域方法

Nothing selected

① 对应 Access-Control-Allow-Methods 属性

允许跨域请求头

允许跨域header，多个可用.分割，全部允许请填写*

① Access-Control-Allow-Headers 属性

上面配置项都有对应的跨域参数说明，如不清楚含义，请自行百度含义。

API配置-路由信息

功能介绍

API配置-路由信息

路由类型	springCloud
服务ID	直接路由
服务Group	springCloud
服务Version	Dubbo
地址前缀	gRpc

该部分如上图所示，目前共支持四种路由方式

直接路由

API配置-路由信息

路由类型	直接路由
目标地址	
地址前缀	
HTTPS	否
自签证书	选择文件

目标地址必填，填写后端服务的服务地址，ip:port或者域名，注意不带http://或https://，此处不可填写path。

地址前缀，指的是后端服务的统一前缀，后面匹配到的endpoint像后端路由时都会加上该前缀，可不填。

如果后端服务是https，则选择HTTPS为是，支持自签证书。

SpringCloud

API配置-路由信息

路由类型	springCloud	
服务ID	SpringCloud服务ID	*
服务Group	SpringCloud服务组别	
服务Version	SpringCloud服务版本	
地址前缀		

tesla-gateway启动时会配置一个eureka地址，spring cloud的路由方式就会从对应的eureka上查找服务。

服务ID必填，此处为后端服务注册到Eureka上的服务名称。

服务Group对应spring.application.group属性，非必填。

服务Version对应spring.application.version属性，非必填。

我们通过order-service服务进行一下配置演示，order-service目前已经注册到了dev环境对应的eureka，我们从**演示订单服务**复制一个新服务**演示订单服务-spring-cloud**并对路由信息进行如下配置：

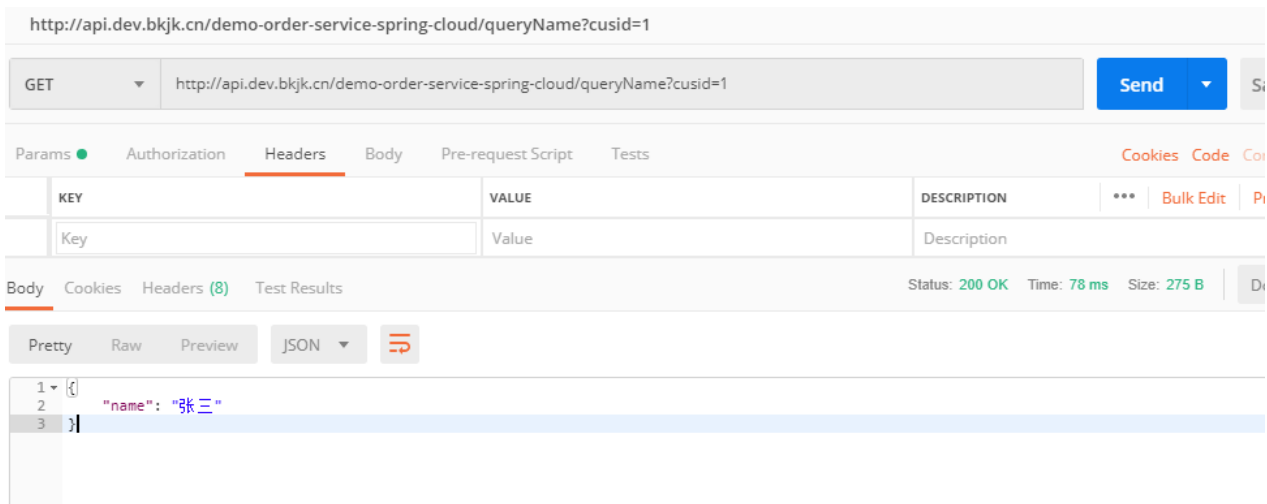
API配置-路由信息

路由类型	springCloud	
服务ID	TESLA-BACKEND-SAMPLE	*
服务Group	SpringCloud服务组别	
服务Version	SpringCloud服务版本	
地址前缀	/order-service	

该服务的调用前缀配置为/demo-order-service-jwt-auth，请求映射关系如下：

GET: http://localhost:9000/demo-order-service-spring-cloud/queryName?cusid=1-->http://localhost:8902/order-service/queryName?cusid=1

访问正常，结果如下：



Dubbo

dubbo路由主要依赖endpoint级别的rpc路由插件完成，请看如下文档：[Endpoint配置-Rpc路由转换插件](#)。

GRpc

grpc路由主要依赖endpoint级别的rpc路由插件完成，请看如下文档：[Endpoint配置-Rpc路由转换插件](#)。

API配置-鉴权信息

功能介绍

鉴权信息的初始配置页面如下图，默认情况是开放状态，此时访问该API下的所有接口都不会进行任何的认证。

API配置-鉴权信息

鉴权类型

开放

标准-OAUTH2

API配置-鉴权信息

鉴权类型	标准-OAUTH2	
认证服务器地址	<input type="text"/>	*
① 最终会调用认证服务器的 /oauth/check_token 接口进行验证		
tokenHeader	<input type="text"/>	*
① 从哪个header中取token，例如 Authorization		

如果不希望接入UUS，又希望有OAUTH2的权限验证动作，可以使用标准-OAUTH2类型，此处需要填写OAUTH2服务器地址（需要服务方有自己的OAUTH2服务），

tesla会从指定Header中取出token，调用oauth2服务器的/oauth/check_token接口进行验证，oauth2服务器代码可以参考 `tesla\tesla-auth\tesla-oauth2-server` 项目

JWT

API配置-鉴权信息

鉴权类型	JWT	
issuer	可填为BKJK	*
验证秘钥	secretKey	*
是否解析claims	解析	
claimsHeader	解析到的claims放入哪个header	

使用该鉴权类型需要了解使用tesla-auth项目下的jwt工具包，可参考项目下test代码。

使用该鉴权类型用户需要首先有一个途径来通过JWT工具包来生成一个有效的jwt token，然后后续通过tesla的请求中将这个token放到header的**Authorization**中。

tesla收到请求后，会从**Authorization**中获取jwt token，并使用JWT工具包中验证token的方法来验证该token的有效性。

- **issuer**：请参考JWT工具包，生成token是需要传入这个值，如无特殊需要可直接填写为 BKJK。
- **验证秘钥**：请参考JWT工具包，与生成jwt token时的秘钥一致，参考值 1FihRrMitxjiEVC11CytWdthUyWytD+7。
- **是否解析claims**：jwtToken，包含有加密的用户信息，如果后端系统需要用户信息，可以此处选择解析，并填写claimsHeader，tesla会解析用户信息，并放入指定的header中，后端系统可以直接使用。

JWT-配置演示

我在演示订单服务的基础上复制了一个新的API配置--演示订单服务-jwt-auth，在API配置-鉴权信息处进行了如下配置：

API配置-鉴权信息

鉴权类型

JWT

issuer

BKJK

*

验证密钥

1FihRrMitxjiEVC1ICytWdthUyWytD+7

*

是否解析claims

解析

claimsHeader

X-BK-Jwt-Claims

该服务的调用前缀配置为/demo-order-service-jwt-auth，请求映射关系如下：

GET: http://localhost:9000/demo-order-service-jwt-auth/queryName?cusid=1-->http://localhost:8902/order-service/queryName?cusid=1

但此时直接访问http://localhost:9000/demo-order-service-jwt-auth/queryName?cusid=1是访问不通的，因为此时整个服务受鉴权保护，需要校验JWT TOKEN，访问结果截图如下：

http://api.dev.bkjk.cn/demo-order-service-jwt-auth/queryName?cusid=1

GET http://api.dev.bkjk.cn/demo-order-service-jwt-auth/queryName?... Send Save

Params Authorization Headers Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
Key	Value	Description			

Body Cookies Headers (8) Test Results Status: 403 Forbidden Time: 313 ms Size: 344 B Download

Pretty Raw Preview Auto

```
1 { "code": "OSG003", "message": "There are at least one header 'Authorization' expected" }
```

我们需要使用JWT工具包生成jwt token，将token放入Authorization Header中，再进行如下图的访问，才可访问成功：

http://api.dev.bkjk.cn/demo-order-service-jwt-auth/queryName?cusid=1

GET http://api.dev.bkjk.cn/demo-order-service-jwt-auth/queryName?cusid=1 Send Save

Params Authorization Headers (1) Body Pre-request Script Tests Cookies Code Comments

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJrZV9p...				
Key	Value	Description			

Body Cookies Headers (8) Test Results Status: 200 OK Time: 1214 ms Size: 275 B Download

Pretty Raw Preview JSON

```
1 {  
2   "name": "张三"  
3 }
```

API配置-限流配置

功能介绍

- 该插件使用令牌桶的方式来对请求进行限流。

配置介绍

该插件配置页面如下：

API配置-限流配置

启用限流

禁用

单位时间频率

单位时间频率

单位时间(秒)

单位时间

默认是不启用限流，若要开启限流，请选择**启用限流**为启用，并填入**单位时间频率**和**单位时间(秒)**。

配置演示

以demo-order-service为例，在启用限流插件处做了如下配置：

API配置-限流配置

启用限流

启用

单位时间频率

2

单位时间(秒)

1

该插件对整个API都生效，该配置表示1秒最多能请求两次，我使用下面请求做演示，当请求超过1秒两次后，会出现下面结果：

GET: <http://localhost:9000/demo-order-service/queryName?cusid=1>

http://api.dev.bkjk.cn/demo-order-service/queryName?cusid=1

GET

http://api.dev.bkjk.cn/demo-order-service/queryName?cusid=1

Send

Save

Params

Authorization

Headers

Body

Pre-request Script

Tests

Cookies

Code

Comments

	KEY	VALUE	DESCRIPTION	...	Bulk Ed
<input checked="" type="checkbox"/>	cusid	1			
	Key	Value	Description		

Body

Cookies

Headers (8)

Test Results

Status: 429 Too Many Requests Time: 44 ms Size: 392 B Download

Pretty

Raw

Preview

Auto

1

rate rule : 2_1 , rateId : service_549949827350265856 , uri : /demo-order-service/queryName RateLimiter Filter has limited

出现该响应即表示请求频率超过配置的频率，该请求被限流插件拦截掉了。

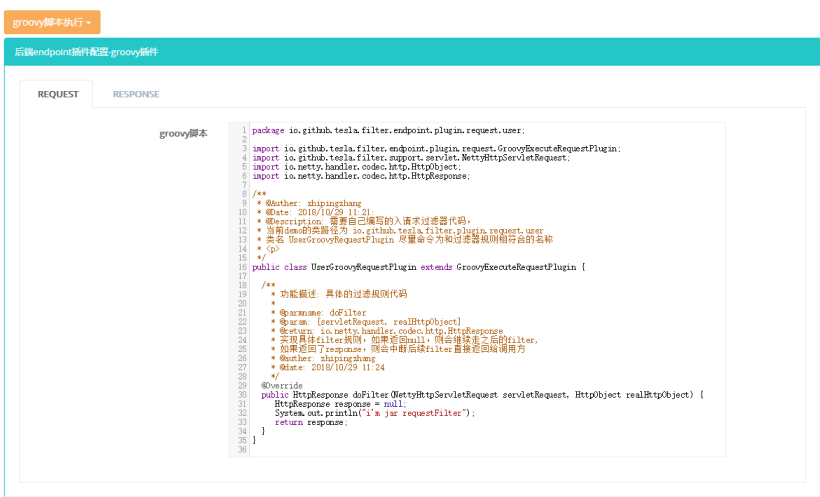
Endpoint配置-Groovy脚本插件

功能介绍

- 如果有特殊化的功能，当前插件体系无法满足时，可以使用该插件自己编写逻辑。
- 该插件是用GroovyClassLoader来编译运行java代码，所以该插件中配置的代码实际是java代码。
- 该插件只适用于编写简单逻辑，如需要引用tesla本身没有的jar包，请使用执行上传jar包插件。

配置介绍

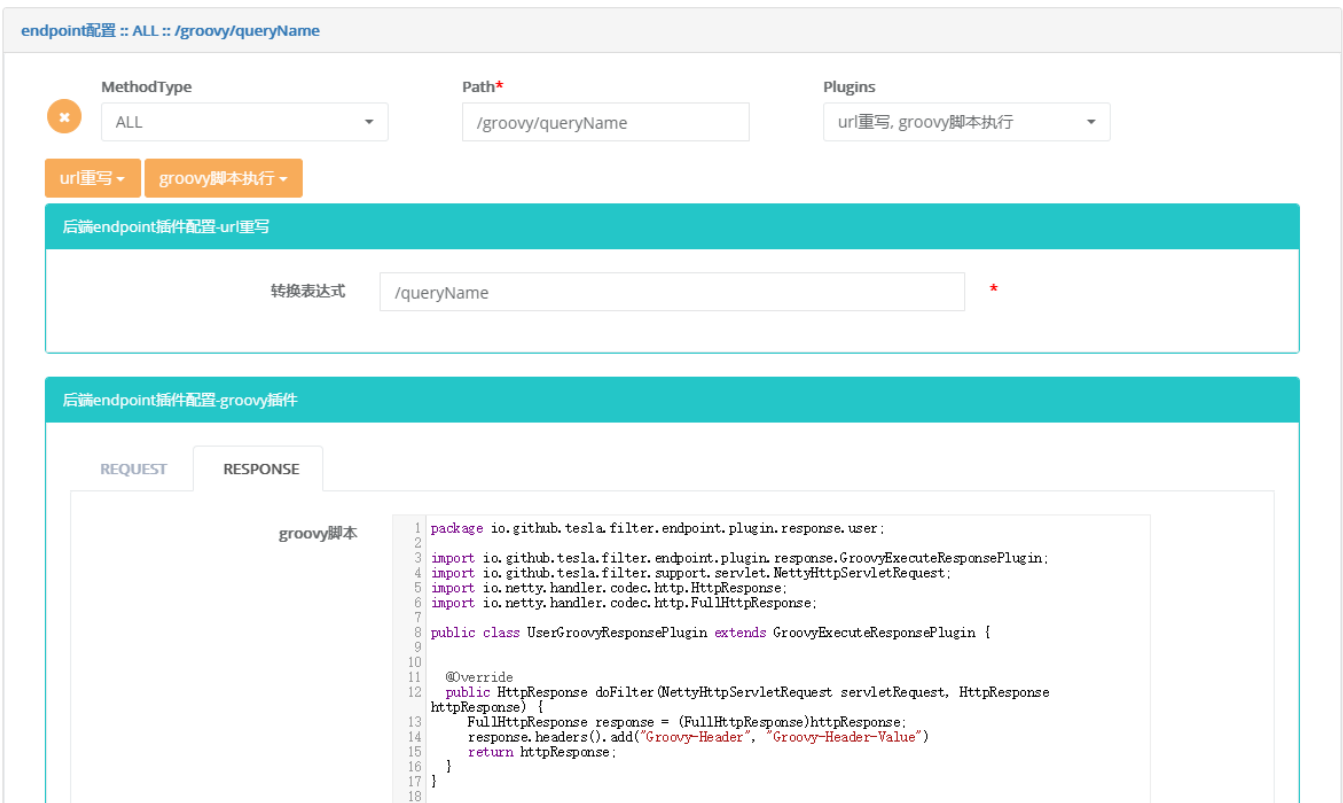
该插件配置页面如下：



可配置request和response时执行的脚本，此处给出了代码示例，可在示例代码基础上修改完善自己需要的逻辑。

配置演示

以demo-order-service为例，我想要在/queryName上加入groovy脚本执行插件，此时我新增一个Endpoint配置，如下图：



我在groovy执行脚本response部分在示例的基础上修改代码，主要修改类名，doFilter方法体的逻辑，要注意import，如果方法体里引了其他类，注意import进来，代码如下：

```
package io.github.tesla.filter.endpoint.plugin.response.user;

import io.github.tesla.filter.endpoint.plugin.response.
GroovyExecuteResponsePlugin;
import io.github.tesla.filter.support.servlet.NettyHttpRequest;
import io.netty.handler.codec.http.HttpResponse;
import io.netty.handler.codec.http.FullHttpResponse;

public class UserGroovyResponsePlugin extends GroovyExecuteResponsePlugin {

    @Override
    public HttpResponse doFilter(NettyHttpRequest servletRequest,
        HttpResponse httpResponse) {
        FullHttpResponse response = (FullHttpResponse)httpResponse;
        response.headers().add("Groovy-Header", "Groovy-Header-Value")
        return httpResponse;
    }
}
```

这段脚本实际就是java代码，大家都可以看明白，作用就是给response添加了一个Header。

这里我为了不影响原/queryName的效果，还使用了url重写插件，按上图配置，请求映射关系如下：

```
GET http://localhost:9000/demo-order-service/groovy/queryName?cusid=1 -->
http://localhost:8902/order-service/queryName?cusid=1
```

此时我们访问<http://localhost:9000/demo-order-service/groovy/queryName?cusid=1>，可以看到下图中的箭头所指，该groovy脚本已生效。

http://api.dev.bjkc.cn/demo-order-service/groovy/queryName?cusid=1

GET http://api.dev.bjkc.cn/demo-order-service/groovy/queryName?cusid=1 Send Save

Params Authorization Headers Body Pre-request Script Tests Cookies Code Comments

KEY	VALUE	DESCRIPTION	...	Bulk
<input checked="" type="checkbox"/> cusid	1			
Key	Value	Description		

Body Cookies Headers (10) Test Results Status: 200 OK Time: 2215 ms Size: 328 B Download

Server → nginx/1.12.2

Date → Wed, 27 Feb 2019 08:23:16 GMT

Content-Type → application/json; charset=UTF-8

Content-Length → 17

Connection → keep-alive

Groovy-Header → Groovy-Header-Value

Header1 → value1

Vary → Accept-Encoding

Via → 1.1 osg-testgateway-v225-shj2b

X-Frame-Options → SAMEORIGIN

Endpoint配置-Mock插件

功能介绍

- 配置该插件后，请求不会再转发给后端服务，根据配置好的freemarker模板直接响应给调用方。
- 适用于开发环境联调，或者后端服务出现问题时使用。

配置介绍

该插件配置页面如下：

后端endpoint插件配置-mock插件

启用标志

启用

★

① 启用后，请求将不会打到后端，将会使用下方配置的模板构造返回结果

mock返回脚本

1 <!-- 下面为转换模板demo，freemarker脚本，body即为整个requestBody -->
2 {
3 "code": 200,
4 "name": "\${body.name}"
5 }
6

① 脚本可直接定义为静态文本，也可使用freemarker脚本， requestBody在脚本中用body表示，如requestBody为{"name":"bkjk"}, 则在模板中使用\${body.name}取到该值

测试输入

1

测试输出

1

转换测试

mock返回脚本，可以配置成固定的响应体，也可以使用jsonpath抽取request body中的内容来构建响应体。

下方的输入框可以用于测试响应是否符合预期。

配置演示

以demo-order-service为例，我想要在/queryName上加入Mock插件，此时我新增一个Endpoint配置，注意此处配置的是POST，因为需要请求体有值，如下图：

endpoint配置 :: POST :: /mock/queryName

MethodType

×

POST

Path★

/mock/queryName

Plugins

url重写, mock插件

url重写 mock插件

后端endpoint插件配置-url重写

转换表达式

/queryName

★

后端endpoint插件配置-mock插件

启用标志

启用

★

① 启用后，请求将不会打到后端，将会使用下方配置的模板构造返回结果

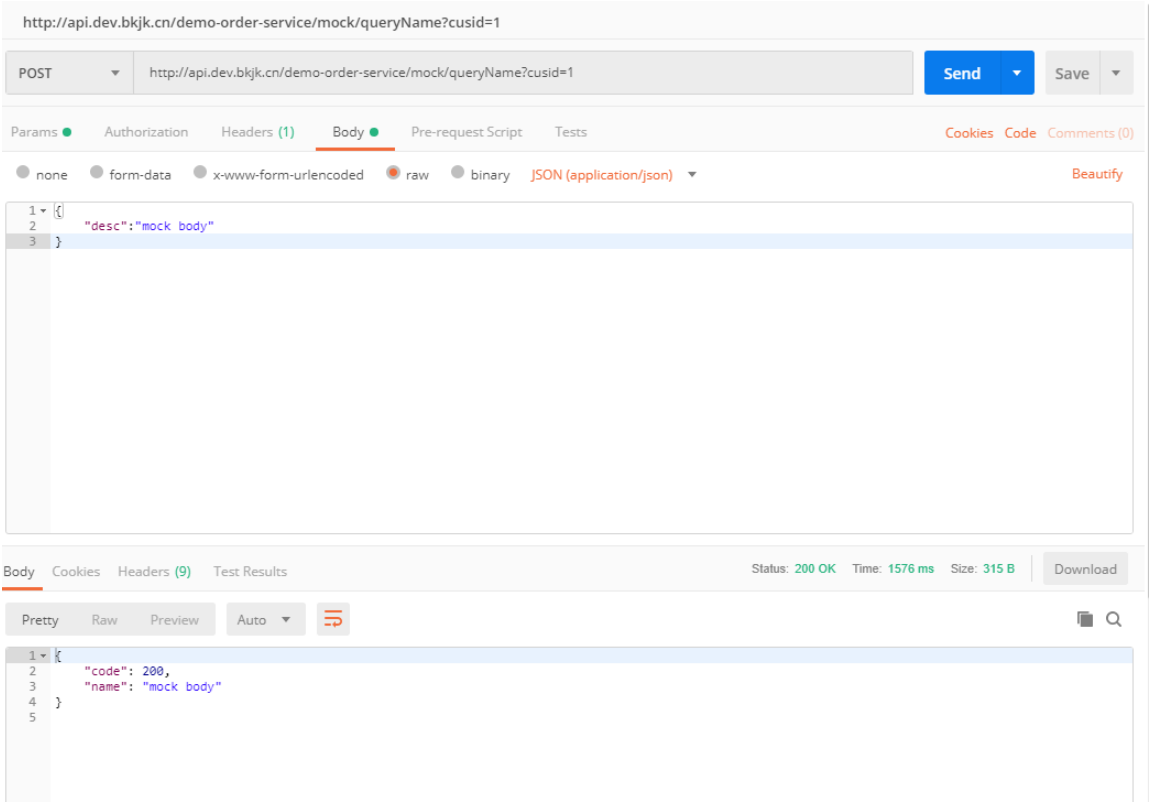
Mock插件配置如下：



这里我为了不影响原/queryName的效果，还使用了url重写插件，按上图配置，请求映射关系如下：

POST http://localhost:9000/demo-order-service/mock/queryName?cusid=1 -->
http://localhost:8902/order-service/queryName?cusid=1

此时我们用POST方式访问http://localhost:9000/demo-order-service/mock/queryName?cusid=1，并在请求体中赋值，会得到如下响应



可以看到，返回的结果是我们刚才配置在mock插件中的内容，并且提取了请求体的内容。

Endpoint配置-Rpc路由转换插件

功能介绍

- 该插件是为gRpc和dubbo路由方式服务的。
- 用于配置调用的方法信息，将http转换为rpc调用所需要的参数。

配置介绍

该插件配置页面如下：

Rpc路由配置

后端endpoint插件配置-rpc路由转换

rpc类型

Dubbo

服务名称

服务名称

服务group

服务group

服务version

服务version

方法名称

方法名称

参数转换模板

1

{

2

{

3

"type": "io.github.tesla.dubbo.pojo.UserRequest",

4

"expression": "\${jsonStr}"

5

}

6

}

7

}

rpc类型可选择dubbo和grpc。

dubbo

tesla-gateway启动时会配置一个zookeeper地址，配置dubbo服务必须注册在这个zookeeper上，

服务名称，group，version都是注册到zookeeper的属性，方法名称即为调用该服务的哪个方法。

参数转换之前需要使用者了解dubbo泛化调用，参考文档：<http://dubbo.apache.org/zh-cn/docs/user/demos/generic-reference.html>。

参数转换模板就是为了抽取http的body，所以要求Content-Type: application/json，整个body会被转为别名 jsonStr，模板示例中的

```
[
  {
    "type": "io.github.tesla.dubbo.pojo.UserRequest",
    "expression": "${jsonStr}"
  }
]
```

表示调用方法只有一个参数，参数类型是io.github.tesla.dubbo.pojo.UserRequest，参数是body，此时就要求body是可以转换成io.github.tesla.dubbo.pojo.UserRequest的。

也可以参考dubbo泛化调用的demo，假定参数只是一个string时，要抽取body中的name时，则参数转换模板应该这么配：

```
[
  {
    "type": "java.lang.String",
    "expression": "${jsonStr.name}"
  }
]
```

GRpc

配置界面如下：

Rpc路由配置 ▾

后端endpoint插件配置:rpc路由转换

rpc类型	gRPC ▾	
服务名称	服务名称	*
服务group	服务group	
服务version	服务version	
方法名称	方法名称	*
grpc定义文件	选择文件	*

📎 上传grpc定义文件

与duubo配置不同处在于，grpc需要用户上传一个包含proto files的zip压缩包。

Endpoint配置-Url重写插件

功能介绍

- 该插件可对tesla请求到后端的url进行差异化配置。
- 只有路由方式不为RPC时才可选并生效，用于当默认mapping不适用时的特殊化配置，支持直接转发和占位符转发。
- parameter不受插件影响。

配置介绍

该插件配置页面如下：

endpoint配置 :: ALL

✕

MethodType

ALL ▾

Path*

/v1/**

Plugins

url重写 ▾

url重写 ▾

后端endpoint插件配置:url重写

转换表达式	对path进行特殊化转换，支持占位符，如/service/{1}	*
-------	----------------------------------	---

当endpoint的Path配置为service/v1，默认发向后端的液位service/v1，如果后端服务实际为service/v2，可在该插件中转换表达式配置为service/v2。当endpoint的Path配置为*/v1，可在插件中配置#{1}/v2，则当收到service/v1时，会转发给service/v2，**通配符时同理。

配置演示

直接改写

如下图所示，此时转发关系为：

```
GET http://localhost:9000/demo-order-service/changeurl/queryName?cusid=1 --> http://localhost:8902/order-service/queryName?cusid=1
```

endpoint配置 :: ALL :: /changeurl/queryName

MethodType

ALL

Path*

/changeurl/queryName

Plugins

url重写

url重写

后端endpoint插件配置-url重写

转换表达式

/queryName

*

通配符改写

可以使用通配符来抽取原请求path的内容，如下图所示，此时转发关系为：

```
GET http://localhost:9000/demo-order-service/changeurl/queryName?cusid=1 --> http://localhost:8902/order-service/queryName?cusid=1
```

endpoint配置 :: ALL :: /changeurl/*

MethodType

ALL

Path*

/changeurl/*

Plugins

url重写

url重写

后端endpoint插件配置-url重写

转换表达式

/#{1}

*

Endpoint配置—创建token插件

功能介绍

- 外部系统通过tesla调用认证的接口后，由tesla通过该插件使用认证成功后的信息封装jwtToken，返回给调用方，后续的鉴权验证就使用该token。
- 该插件应该用于认证授权类型的接口。
- 该插件目前以不再提供给新接入方使用。

配置介绍

该插件配置页面如下：

endpoint配置 :: ALL

MethodType

ALL

Path*

/v1/**

Plugins

创建token

创建token

后端endpoint插件配置-创建token

REQUEST

RESPONSE

token类型

关闭

关闭

oauth

jwt

request中生成token已不建议使用，下面只说明response中生成token，token类型中可以选择token类型。

jwt:

后端endpoint插件配置-创建token

REQUEST

RESPONSE

token类型

jwt

expiresHeader

从哪个header中取超时时间

*

issuer

可填为BKJK

*

claimsHeader

从哪个header中取claims

*

secretKey

生成token使用的秘钥

*

tokenHeader

生成的token放入哪个header

*

tesla会从该后端服务的响应中的该claimsHeader中获取claims信息，再配合issuer，secretKey，expires值来生成jwt token，该token会被放入tokenHeader中返回给调用方。

Endpoint配置-执行上传jar包插件

功能介绍

- 如果有特殊化的功能，当前插件体系无法满足时，可以使用该插件自己编写逻辑。
- 该插件相比较于groovy插件，优势在于可引入自己需要的其他jar包，实现复杂逻辑，甚至是与外部系统或组件交互。

配置介绍

该插件配置页面如下：

执行上传jar包

后端endpoint插件配置 jar包插件

REQUEST

RESPONSE

上传jar包

选择文件

请上传对于规则jar包，具体打包项目请参考<https://code.bkjk-inc.com/projects/OSG/repos/tesla/browse/tesla-jarfilter>

主要介绍如何生成Jar包，请参考tesla\tesla-sample\tesla-jarfilter项目。

将该项目clone到本地，可看到代码目录结构如下图所示：

```
tesla-jarfilter
├── lib
├── src
│   └── main
│       └── java
│           └── io
│               └── github
│                   └── tesla
│                       └── filter
│                           └── plugin
│                               ├── request
│                               │   └── myapp
│                               │       └── MyRequestPlugin.java
│                               └── response
│                                   └── myapp
│                                       └── MyResponsePlugin.java
```

代码分别给出了request和response的示例

```
/**
 * @Description: demo io.github.tesla.filter.plugin.request.myapp
 * myapp, MyRequestPlugin
 * <p>
 * myappplugindemo
 */
public class MyRequestPlugin extends JarExecuteRequestPlugin {

    /**
     * :
     *
     * @paramname: doFilter
     * @param: [servletRequest, realHttpRequest]
     * @return: io.netty.handler.codec.http.HttpResponse
     * filternullfilter,responsefilter
     */
    @Override
    public HttpResponse doFilter(NettyHttpServletRequest servletRequest,
                                HttpRequest realHttpRequest) {
        HttpResponse response = null;
        System.out.println("i'm jar requestFilter");
    }
}
```

```

        return response;
    }
}

/**
 * @Description: demo io.github.tesla.filter.plugin.response.myapp
 * myapp, MyResponsePlugin
 * <p>
 * myappplugindemo
 */
public class MyResponsePlugin extends JarExecuteResponsePlugin {
    /**
     * :
     * @paramname: doFilter
     * @param: [servletRequest, httpResponse]
     * @return: io.netty.handler.codec.http.HttpResponse
     * filternullfilter,responsefilter
     */
    @Override
    public HttpResponse doFilter(NettyHttpRequest servletRequest,
        HttpResponse httpResponse) {
        HttpResponse response = null;
        // load jar and exec
        System.out.println("i'm jar requestFilter");
        return response;
    }
}

```

实现自己的逻辑后，请注意包名myapp一定要进行修改，之后就可以打包，得到jar包就可以在插件配置处进行上传使用。

配置演示

我实现了一个在response中添加header的功能，并打了Jar包，源码和Jar包请见附件，主要改造为删除了demo代码，因为我不需要，并新建了以下类AddHeaderResponsePlugin，代码如下：

```

package io.github.tesla.filter.plugin.response.addheader;

import io.github.tesla.filter.endpoint.plugin.response.
JarExecuteResponsePlugin;
import io.github.tesla.filter.support.servlet.NettyHttpRequest;
import io.netty.handler.codec.http.HttpResponse;

import java.util.Date;

public class AddHeaderResponsePlugin extends JarExecuteResponsePlugin {
    @Override

```

```

    public HttpServletResponse doFilter(NettyHttpServletRequest servletRequest,
    HttpServletResponse httpResponse) {
        HttpServletResponse response = null;
        httpResponse.headers().add("Jar-Execute-Header",new Date());
        return response;
    }
}

```

然后打包，拿到如下Jar包：

tesla-jarfilter-1.0.0-jar-with-dependencies.jar

我们在demo-order-service中新建如下endpoint，并使用Jar包执行插件，配置如下，请注意箭头处：

endpoint配置 :: ALL :: /jar/queryName

✕

MethodType

ALL

Path*

/jar/queryName

Plugins

url重写, 执行上传jar包

url重写

执行上传jar包

后端endpoint插件配置-url重写

转换表达式

/queryName

*

后端endpoint插件配置-jar包插件

REQUEST

RESPONSE

上传jar包

选择文件

tesla-jarfilter-1.0.0-jar-with-dependencies.jar

ⓘ 请上传对于规则jar包，具体打包项目请参考<https://code.bkjk-inc.com/projects/OSG/repos/tesla/browse/tesla-jarfilter>

请求映射关系如下：

```

GET http://localhost:9000/demo-order-service/jar/queryName?cusid=1 -->
http://localhost:8902/order-service/queryName?cusid=1

```

请求结果如下，可以看到响应header里包含了Jar-Execute-Header，说明jar包中的代码已执行，符合预期

http://api.dev.bkjk.cn/demo-order-service/jar/queryName?cusid=2

GET http://api.dev.bkjk.cn/demo-order-service/jar/queryName?cusid=2 Send

Params Authorization Headers Body Pre-request Script Tests Cookies Code

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> cusid	2	
Key	Value	Description

Body Cookies Headers (10) Test Results Status: 200 OK Time: 195 ms Size: 343 B

Server → nginx/1.12.2

Date → Thu, 28 Feb 2019 10:53:44 GMT

Content-Type → application/json; charset=UTF-8

Content-Length → 17

Connection → keep-alive

Header1 → value1

Jar-Execute-Header → Thu, 28 Feb 2019 10:53:44 GMT

Vary → Accept-Encoding

Via → 1.1 osg-teslagateway-v229-pjh44

X-Frame-Options → SAMEORIGIN

Endpoint配置-查询聚合插件

功能介绍

- 该插件可实现调用方一次查询，聚合多个后端接口数据的效果。
- 后端服务需要已经发布在tesla上。
- 后端接口返回数据必须为json类型，且顶层结构必须为对象，不支持顶层为list。

配置介绍

该插件配置页面如下：

endpoint配置 :: ALL ::

MethodType Path* Plugins

ALL /v1/** 查询聚合插件

查询聚合插件

后端endpoint插件配置-聚合查询

添加查询配置 该插件可实现调用方一次查询，聚合多个后端接口数据的效果 [了解更多](#)。

后端服务 请求路径

房产金融套 支持占位符，如/service/{1}

选择该插件后，可通过[添加查询配置](#)按钮添加一个后端查询配置。

每一个后端查询配置需要配置两项：

- 后端服务：请求后端服务的路由信息，从已有服务中选择

- 请求路径：具体请求的路径

重点举例说明请求路径的配置，请求路径支持从该endpoint中path取参，

例如endpoint中path配置为 /service/*，

下方的某个后端请求路径配置为 /service/{1}?name={name1}，

其中 /service/{1} 的 {1} 会取到 /service/* 第一个 * 匹配的值，

?name={name1} 中的 {name1} 会取到 param 中 key 为 name1 的值，
当请求 tesla 的真实请求为 /service/man?name1=zhang 时，打给后端服务的请求即为 /service/man?name=zhang

配置演示

我们以 demo-order-service 服务为例做演示，我希望调用一次 tesla 就可以得到 /queryName, /queryProvinces, /queryOrder 三个请求的结果聚合。

我们先分别直接请求上述三个接口，看下结果：

```
GEThttp://localhost:8902/order-service/queryName?cusid=1
Response:
{
    "name": ""
}

GEThttp://localhost:8902/order-service/queryProvinces?cusid=1
Response:
{
    "provinces": ""
}

GEThttp://localhost:8902/order-service/queryOrder?cusid=1
Response:
{
    "orderType": "",
    "orderTime": "Thu Feb 28 11:22:02 CST 2019"
}
```

现在我在 demo-order-service 上，新增一个 Endpoint 配置，并使用查询聚合插件，如下图：

endpoint配置 :: ALL :: /queryMerge

MethodType: ALL Path: /queryMerge Plugins: 查询聚合插件

查询聚合插件

后端endpoint插件配置-聚合查询

添加查询配置 该插件可实现调用方一次查询，聚合多个后端接口数据的效果 [了解更多](#)。

后端服务	*	请求路径	*
演示订单服务		/order-service/queryName	
后端服务	*	请求路径	*
演示订单服务		/order-service/queryProvinces	
后端服务	*	请求路径	*
演示订单服务		/order-service/queryOrder	

请注意，我上面的配法未关注param，param会被原封不动的转发给后端，而且path也填写了后端服务完整的path，包含了/order-service，此时我发起

```
GEThttp://localhost:9000/demo-order-service/queryMerge?cusid=1
```

此时实际会向后端打出三个请求，?cusid=1 这部分被原封不动的转了后端：

```
GEThttp://localhost:8902/order-service/queryName?cusid=1
GEThttp://localhost:8902/order-service/queryProvinces?cusid=1
GEThttp://localhost:8902/order-service/queryOrder?cusid=1
```

此时我得到如下响应，大家可以调用试一下

```
{
  "orderType": "",
  "provinces": "",
  "orderTime": "Thu Feb 28 11:32:45 CST 2019",
  "name": ""
}
```

上述例子是后端服务三个请求需要的param一样的情况，但是大多数时候都是不一致的，如果我需要对打向后端服务的param做修改应该如何操作呢？下面我们通过另一个例子来说明

再配置一个endpoint，如下图

endpoint配置 :: ALL :: /queryMerge2

MethodType

ALL

Path*

/queryMerge2

Plugins

查询聚合插件

查询聚合插件

后端endpoint插件配置 聚合查询

添加查询配置

该插件可实现调用方一次查询，聚合多个后端接口数据的效果 [了解更多](#)。

后端服务

演示订单服务

请求路径

/order-service/queryName?cusid=#{idv1}

后端服务

演示订单服务

请求路径

/order-service/queryProvinces?cusid=#{idv1}

后端服务

演示订单服务

请求路径

/order-service/queryOrder?cusid=#{idv1}

结合上面的说明，此处我从原请求中抽取了paramidv1，并放到了后端请求中，此时调用

```
GEThttp://localhost:9000/demo-order-service/queryMerge2?idv1=1
```

也会得到如下正确聚合结果：

```
{
  "orderType": "",
  "provinces": "",
  "orderTime": "Thu Feb 28 11:42:04 CST 2019",
  "name": ""
}
```

Endpoint配置-消除鉴权插件

功能介绍

- 该插件用于当API的鉴权类型不是开放时，但个别endpoint不希望鉴权时使用，如/login，/logout等。
- 该插件只需要选择，不需要配置属性。

配置演示

以演示订单服务-jwt-auth为例，此API我们配置了jwt-auth，此时整个API受权限校验保护，我们访问如下请求，不放入正确的token会失败

```
GET http://localhost:9000/demo-order-service-jwt-auth/queryOrder?cusid=1 --> http://localhost:8902/order-service/queryOrder?cusid=1

http code : 403
http response:
    {"code":"tesla003","message":"There are at least one header 'Authorization' expected"}
```

我们添加一个endpoint，并配置上消除鉴权插件，如下图：

The screenshot shows the 'endpoint配置' (Endpoint Configuration) interface for the path `/noauth/*`. The 'MethodType' is set to 'ALL'. The 'Path' is `/noauth/*`. The 'Plugins' section shows '消除鉴权控制, url重写' (Remove authentication control, URL rewriting). Below this, there is a section for '后端endpoint插件配置-url重写' (Backend endpoint plugin configuration - URL rewriting) with a '转换表达式' (Transformation expression) field containing `/#{1}`.

此时再进行如下访问，可绕过鉴权校验，访问成功：

```
GET http://localhost:9000/demo-order-service-jwt-auth/noauth/queryOrder?cusid=1 --> http://localhost:8902/order-service/queryOrder?cusid=1

http code : 200
http response:
    {
      "orderType": "",
      "orderTime": "Thu Feb 28 11:51:26 CST 2019"
    }
```

Endpoint配置-签名校验插件

功能介绍

- 关于签名的使用请参考 `tesla\tesla-auth\tesla-auth-sdk\tesla-auth-sdk-jwt` 项目 `diam`。

- 调用方使用签名工具对请求进行签名，tesla会根据验证签名的有效性，防止数据篡改。

配置介绍

该插件配置页面如下：

endpoint配置 :: ALL

MethodType

ALL

Path*

/v1/*

Plugins

签名验证

签名验证

后端endpoint插件配置-签名校验

signHeader

从哪个header取签名

默认可填为：X-BK-Signature

默认AccessKey

当找不到对应的调用方时使用的Acces

默认可填为：eylhbGciOjllUzl1NiIsIngtc3MiOEy 如不填写则不会校验

默认SecretKey

当找不到对应的调用方时使用的Secre

默认可填为：EyMDkzMylsImZcyI6lIRhby1ZYW5nli 如不填写则不会校验

添加验签配置

signHeader：配置tesla从哪个请求头中获取签名进行验证。

默认AccessKey：解密时使用的AccessKey，与加密时的一致。

默认SecretKey：解密时使用的SecretKey，与加密时的一致。

该行配置为默认验签配置，如果该接口有可能接收不同系统的加签数据，且不同系统加签时使用的秘钥不一致，

则可点击添加加签配置，点击后如下图：

后端endpoint插件配置-签名校验

signHeader

从哪个header取签名

默认可填为：X-BK-Signature

默认AccessKey

当找不到对应的调用方时使用的Acces

默认可填为：eylhbGciOjllUzl1NiIsIngtc3MiOEy 如不填写则不会校验

默认SecretKey

当找不到对应的调用方时使用的Secre

默认可填为：EyMDkzMylsImZcyI6lIRhby1ZYW5nli 如不填写则不会校验

添加验签配置

调用方

信用

AccessKey

SecretKey

此处可选择调用方，该列表即为接入列表中的数据，可为选定的调用方配置不同的秘钥。

配置演示

在demo-order-service下新增endpoint配置，如下图：

endpoint配置 :: ALL :: /signVerify/queryName

✖

MethodType

ALL

Path*

/signVerify/queryName

Plugins

url重写, 签名验证

url重写

签名验证

后端endpoint插件配置 url重写

转换表达式

/queryName

*

后端endpoint插件配置 签名校验

signHeader

X-BK-Signature

默认可填为: X-BK-Signature

默认AccessKey

eyJhbGciOiJIUzI1NiIsInR5cCI6ImlzcyI6IHRhby1ZYW5nIl

默认可填为: eyJhbGciOiJIUzI1NiIsInR5cCI6ImlzcyI6IHRhby1ZYW5nIl 如不填写则不会校验

默认SecretKey

EyMDkzMyIsImZyI6IHRhby1ZYW5nIl

默认可填为: EyMDkzMyIsImZyI6IHRhby1ZYW5nIl 如不填写则不会校验

添加验证配置

请求映射关系如下:

```
GET http://localhost:9000/demo-order-service/signVerify/queryName?cusid=1
--> http://localhost:8902/order-service/queryName?cusid=1
```

此时当请求不在header中放入签名的话，调用就会失败

需要使用签名工具包的正确的对报文加签，获得签名并放入X-BK-Signature中，才可通过验证，正常访问。

Endpoint配置-缓存结果插件

功能介绍

- 该插件可根据配置缓存请求结果。
- 缓存以url和http method为key,仅建议GET请求时,且http header中无自定义信息时使用。

配置介绍

该插件配置页面如下:

endpoint配置 :: ALL ::

MethodType Path* Plugins

ALL /v1/** 缓存结果插件

缓存结果插件

后端endpoint插件配置-缓存结果插件

缓存超时时间(秒)

0表示无失效限制
缓存以url和http method为key,仅建议GET请求时,且http header中无自定义信息时使用

可配置缓存失效时间, 0表示永不失效。

配置演示

我们用order-service的/queryOrder接口来做演示, 该接口返回值如下, 会返回一个当前的时间:

```
{
  "orderType": "",
  "orderTime": "Wed Feb 27 17:24:28 CST 2019"
}
```

在demo-order-service上, 新增一个Endpoint配置, 如下图:

endpoint配置 :: ALL :: /cache/queryOrder

MethodType Path* Plugins

ALL /cache/queryOrder 缓存结果插件

缓存结果插件

后端endpoint插件配置-缓存结果插件

缓存超时时间(秒)

60

0表示无失效限制
缓存以url和http method为key,仅建议GET请求时,且http header中无自定义信息时使用

如图所示, 我配置了60S的超时时间, 这里我为了不影响原/queryName的效果, 还使用了url重写插件, 按上图配置, 请求映射关系如下:

```
GET http://localhost:9000/demo-order-service/cache/queryOrder?cusid=1 -->
http://localhost:8902/order-service/queryOrder?cusid=1
```

第一次访问<http://localhost:9000/demo-order-service/cache/queryOrder?cusid=1>，返回结果如下：

```
{
  "orderType": "",
  "orderTime": "Wed Feb 27 17:38:44 CST 2019"
}
```

60s内我们进行第二次访问，结果如下，可以看到，orderTime是一样的，说明这是被缓存的结果：

```
{
  "orderType": "",
  "orderTime": "Wed Feb 27 17:38:44 CST 2019"
}
```

60s之后我们再进行第三次访问，结果如下，可以看到orderTime已经更新，说明缓存已失效，又一次请求了后端服务：

```
{
  "orderType": "",
  "orderTime": "Wed Feb 27 17:42:10 CST 2019"
}
```

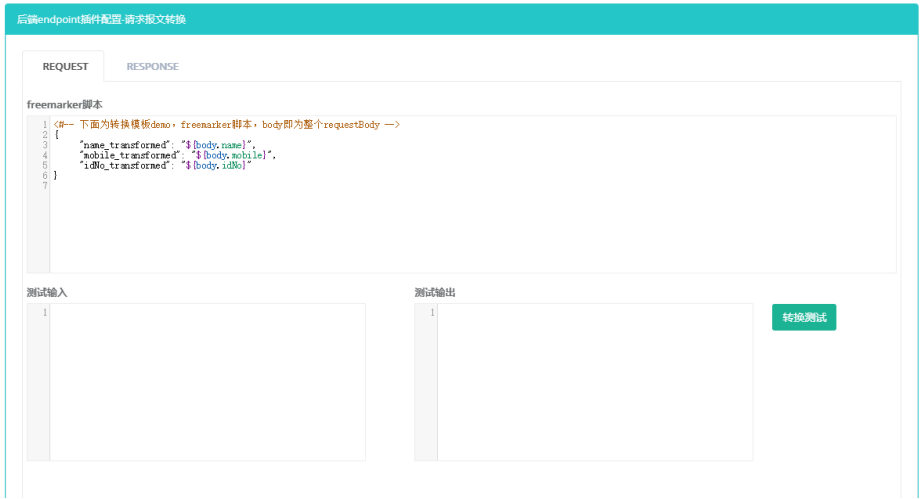
Endpoint配置-请求报文转换插件

功能介绍

- 该插件根据配置的freemarker脚本对请求体和响应体的报文进行转换。

配置介绍

该插件配置页面如下：



freemarker脚本为配置项，如果所示，默认给出了一段freemarker示例脚本，原请求体和响应体的数据都可以使用`${body.}`的jsonpath的形式进行抽取。

配置好freemarker脚本后，可以在下方进行测试，看转换后的报文体是否符合预期。

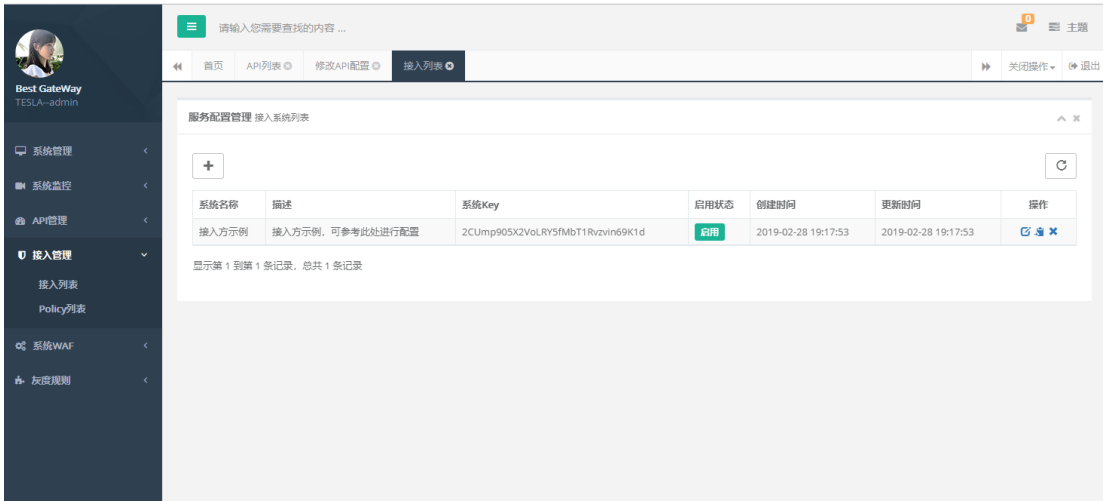
调用方接入管理

接入简介

接入管理是为了控制调用方通过tesla访问后端服务的访问权限、限流，还与一些插件结合用于识别不同的调用方从而进行不同的动作。

接入列表

点击左侧菜单 接入管理--接入列表，会显示如下页面，点击页面左上角+号进入新增接入系统配置页面。



新增接入系统

新增接入系统配置页面如下，配置项包含基础信息，Policy配置，限流配置，请求配置，权限配置。

基础信息

应用名称

*

Key

随机生成

*

接入方需将该key放入header的X-Tesla-AccessKey中，方可被OSG识别

是否启用

启用

▼

Key描述

*

Policy配置

Policy

Nothing selected

▼

基础信息

基础信息主要说明的是Key，该值可通过随机生成按钮生成随机字符串，也可手动填写，要求唯一，接入方需将该key放入header的X-Tesla-AccessKey中，方可被tesla识别，如果调用时不包含该Header，或key值错误，则会被tesla拒绝请求。

Policy配置

Policy是一组预配置好的规则，可在接入管理Policy列表处配置，选择某一个Policy后，下面的限流配置，请求配置，权限配置会被Policy中的配置覆盖。

权限配置

此处可选择该调用方允许访问的服务，可多选，如果不想进行访问权限控制，则此处不需要选择。

配置演示

可参考dev环境的[接入方示例](#)配置，如下图，也可登入系统查看

基础信息

应用名称

接入方示例

*

Key

2CUmp905X2VoLRY5fMbT1Rvzvin69K1d

随机生成

*

接入方需将该key放入header的X-Tesla-AccessKey中，方可被OSG识别

是否启用

启用

▼

Key描述

接入方示例，可参考此处进行配置

*

Policy配置

Policy

Nothing selected

▼

限流配置

限流开关

启用

单位时间频率

100

单位时间(秒)

10

请求配置

最大请求数

36000

配置为-1表示不限制

复位间隔时间

1

复位时间单位

day

权限配置

允许访问服务

演示订单服务-uus-oauth2, 演示订单服务-jwt-auth, 演示订单服务, 演示订单服务-

不选择, 则表示所有服务都可访问

保存

关闭

灰度规则配置

灰度简介

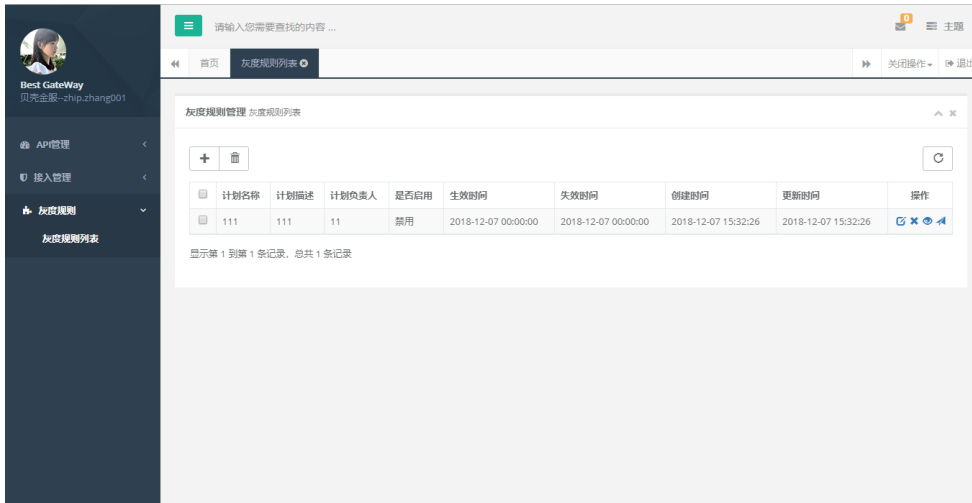
灰度发布（又名金丝雀发布）是指在黑与白之间，能够平滑过渡的一种发布方式。在其上可以进行A/B testing，即让一部分用户继续用产品特性A，一部分用户开始用产品特性B，如果用户对B没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到B上面来。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题，以保证其影响度。

tesla与summerframework结合，通过gateway为入口，实现了整个调用链的灰度节点选择，可以通过配置规则来通过header，param，body中的内容来选择将特殊的请求打向不同的节点。

要求后端服务接入platfrom-starter-eureka，文档请参考platfrom-starter-eureka使用手册。

配置介绍

下图为灰度规则列表页面



点击左上角+号进入灰度规则新增页面，如下

添加灰度计划

计划名称:

请输入计划名称

计划描述:

请输入计划描述

计划负责人:

请输入计划负责人

有效时间:

请选择起始日期

请选择结束日期

保存计划基本信息

新增灰度规则

关闭

计划名称，计划描述，计划负责人都是描述性字段，有效时间需注意，只有当前时间在该段时间内灰度策略才会生效。

上面都是基本信息，我们先保存计划基本信息，之后点击新增灰度规则按钮，下方出现新的选项卡：

保存计划基本信息

新增灰度规则

关闭

0-灰度规则

灰度规则

保存规则

删除

服务消费方:

TESLA-GATEWAY

服务提供方:

TESLA-GATEWAY

以下灰度策略中，当选择为请求头或请求参数时，key请填写header或param的key，当选择为请求体时，仅支持json格式的请求体，key请填写jsonpath，透传选项仅对请求头生效

灰度策略:

请求:

如果含有:

请输入key

请输入value

透传

路由结果:

节点中

GROUP:

请输入GROUP

节点中

VERSION:

请输入VERSION

节点中

bljkggray:

请输入bljkggray

每一个灰度规则的选项卡代表一条具体的灰度规则，假设有一个请求，整个请求链路如下：

APP----->TESLA----->A----->B

其中当A和B都有灰度节点时，我们需要在此处配置两条灰度规则，分别是TESLA----->A和A----->B。

配置示例

我们以demo-order-service为例，假设共有两个实例，其中1个版本号是1.0.0为正常节点，另外一个版本号为1.0.1位灰度节点，当检测到request header中有gray=true时，请求要打向灰度节点，需要如下配置：

0-灰度规则

① 灰度规则

服务消费方：TESLA-GATEWAY

服务提供方：DEMO-ORDER-SERVICE

ⓘ 以下灰度策略中，当选择为请求头或请求参数时，key请填写header或param的key，当选择为请求体时，仅支持json格式的请求体，key请填写jsonpath，透传选项仅对请求头生效

灰度策略：请求头

如果含有：graytrue

透传

路由结果：

节点中GROUP：请输入GROUP

节点中VERSION：1.0.1

节点中bkjkgray：请输入bkjkgray

保存规则

删除

之后我们点击保存规则就可生效，tesla在选择order-service节点的时候会根据这个规则来选择。

注意点：

- 透传只针对请求头，指的是用于判断的请求头是否传递给下一个系统。
- 多个条件之间是且的关系，比如既配置了请求头A=B，又配置了请求参数C=D，则A=B&&C=D时才会走灰度节点。
- 请求体的key是jsonpath，只支持Content-Type：application/json类型。