# 网关

Spring Cloud Zuul大家可以理解为一个集网关（路由）、负载均衡、校验过滤、结合服务治理框架、请求转发时熔断机制、服务聚合等 一系列功能。我们可以将Zuul当成一个门面，所有外部请求都经过Zuul的转发到具体的服务实例，减少了每个服务之间互相鉴权代码冗余问题，统一交给Zuul进行鉴权，在此基础上集成上边说的高级功能。路由功能相当于反向代理。

## 路由

### 传统路由实现方式

zuul:

routes:

traditional-url: #传统的路由配置,此名称可以自定义

path: /tr-url/\*\* #映射的url

url: http://localhost:9001/ #被映射的url

### 面向服务的路由

面向服务的路由需要指定注册中心的地址

zuul:

routes:

orient-service-url: #面向服务的路由配置,此名称可以自定义

path: /os-url/\*\*

service-id: feign-customer #服务名

eureka:

client:

serviceUrl:

defaultZone: <http://peer1:1111/eureka/>

如果有一个可以同时满足多个path的匹配的情况，此时匹配结果取决于路由规则的定义顺序

## 请求过滤

### 过滤器

String filterType();

int filterOrder();

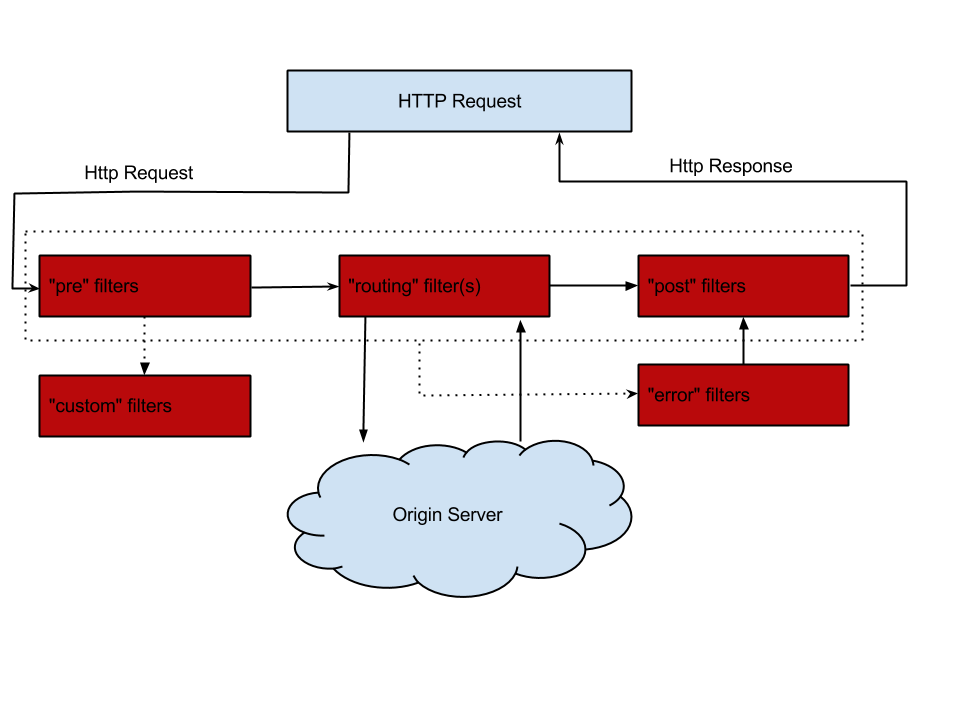
boolean shouldFilter();

Object run();

这里分别进行解释：

* filterType：该方法需要返回一个字符串来代表过滤器的类型，而这个类型就是Zuul中的4种不同生命周期的过滤器类型，如下
  + pre：在请求到达路由前被调用
  + route：在路由请求时被调用
  + error： 处理请求时发生的错误时被调用。
  + post：在route和error过滤器之后被调用，最后调用。
* filterOrder：通过int值定义过滤器执行顺序，数值越小优先级越高。
* shouldFilter：返回布尔值来判断该过滤器是否执行。
* run：过滤器的具体逻辑。可以在此确定是否拦截当前请求等。

### 过滤器生命周期



**首先HTTP请求到达Zuul，最先来到pre过滤器，在这里会去映射url patern到目标地址上**，**然后将请求与找到的地址交给route类型的过滤器进行求转发，请求服务实例获取响应，通过post类型过滤器对处理结果进行加工与转换等操作返回。error类型的过滤器比较特殊，在这整个请求过程中只要有异常才会触发，将异常结果交给post类型过滤器加工返回**

### 禁用过滤器

说到禁用过滤器，第一想到的是自定义的过滤器中shouldFilter返回false，实际应用中，这样还需要重新编译代码。

Zuul贴心地提供了一个参数用来禁用指定过滤器，zuul.<过滤器名>.<过滤器类型>.disable=true

之前我们的实验必须headers中有accessToken才能通过AccessFilter，现在我们禁用一下试试

zuul:

AccessFilter:

pre:

disable: true #禁用名为AccessFilter的过滤器

### 例子

定义一个简单的Zuul过滤器，实现检查HttpServletRequest中是否有accessToken参数，如果有就进行路由，没有则拒绝访问，还回 401 Unauthorized 错误

package com.cnblogs.hellxz.filter;

import com.netflix.zuul.ZuulFilter;

import com.netflix.zuul.context.RequestContext;

import org.apache.commons.lang.StringUtils;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;

/\*\*

\* 访问过滤器

\*/

@Component

public class AccessFilter extends ZuulFilter {

private final Logger logger = LoggerFactory.getLogger(AccessFilter.class);

/\*\*

\* 过滤器类型选择：

\* pre 为路由前

\* route 为路由过程中

\* post 为路由过程后

\* error 为出现错误的时候

\* 同时也支持static ，返回静态的响应，详情见StaticResponseFilter的实现

\* 以上类型在会创建或添加或运行在FilterProcessor.runFilters(type)

\*/

@Override

public String filterType() {

return "pre"; //ZuulFilter源码中注释"pre"为在路由前过滤

}

/\*\*

\* 用来过滤器排序执行的

\* @return 排序的序号

\*/

@Override

public int filterOrder() {

return 0;

}

/\*\*

\* 是否通过这个过滤器，默认为true，改成false则不启用

\*/

@Override

public boolean shouldFilter() {

return false; //返回true表示执行这个过滤器

}

/\*\*

\* 过滤器的逻辑

\*/

@Override

public Object run() {

//获取当前请求上下文

RequestContext ctx = RequestContext.getCurrentContext();

//取出当前请求

HttpServletRequest request = ctx.getRequest();

logger.info("进入访问过滤器，访问的url:{}，访问的方法：{}",request.getRequestURL(),request.getMethod());

//从headers中取出key为accessToken值

String accessToken = request.getHeader("accessToken");//这里我把token写进headers中了

//这里简单校验下如果headers中没有这个accessToken或者该值为空的情况

//那么就拦截不入放行，返回401状态码

if(StringUtils.isEmpty(accessToken)) {

logger.info("当前请求没有accessToken");

//使用Zuul来过滤这次请求

ctx.setSendZuulResponse(false);

ctx.setResponseStatusCode(401);

return null;

}

logger.info("请求通过过滤器");

return null;

}

}

## Hystrix和Ribbon

上文有讲过，Zuul中包含了Hystrix和Ribbon的依赖，所以Zuul拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡，需要注意的是传统路由也就是使用path与url映射关系来配置路由规则的时候，对于路由转发的请求不会使用HystrixCommand来包装，所以没有线程隔离和断路器的保护，并且也不会有负载均衡的能力。所以我们在使用Zuul的时候推荐使用path与serviceId的组合来进行配置。

### 设置Hystrix超时时间

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds

来设置API网关中路由转发请求的命令执行时间超过配置值后，Hystrix会将该执行命令标记为TIMEOUT并抛出异常，Zuul会对该异常进行处理并返回如下JSON信息给外部调用方

### 设置Ribbon连接超时时间

ribbon.ConnectTimeout

参数创建请求连接的超时时间

### 设置Ribbon的请求转发超时时间

ribbon.ReadTimeout

来设置请求转发超时时间

### 关闭重试配置

* 全局配置zuul.retryable=false
* 针对路由配置zuul.routes.<路由名>.retryable=false

## 动态路由

<https://cloud.tencent.com/developer/article/1033922>

ZuulServerAutoConfiguration

# OAUTH2.0

## ClientDetailsService

org.springframework.security.oauth2.provider.ClientDetailsService#loadClientByClientId方法没有做缓存处理，在请求获取token时，发现一次token请求，这个方法要被调用4次，因此跟踪一下都是哪里调用这个方法的。

请求先经过一系列过滤器，里面有个BasicAuthenticationFilter，主要用来做安全认证

org.springframework.web.filter.OncePerRequestFilter

过滤器基类，旨在确保在任何servlet容器上每个请求分派单个执行。

提供一个带有HttpServletRequest和HttpServletResponse参数的doFilterInternal方法。

org.springframework.security.web.authentication.www.BasicAuthenticationFilter#doFilterInternal

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, ServletException {

final boolean debug = this.logger.isDebugEnabled();

try {

//将请求转为鉴定token需要的对象

/\*\*

\* 1.判断请求头是否有Authorization属性，没有返回null

\* 2.判断Authorization属性值是否以Basic忽略大小写开头，不是的话返回null

\* 3.截取Authorization属性值里面的base64加密后的值，进行解码出用户名和密码

\*/

UsernamePasswordAuthenticationToken authRequest = authenticationConverter.convert(request);

if (authRequest == null) {

chain.doFilter(request, response);

return;

}

/\*\*

\* 获取认证用户名

\*/

String username = authRequest.getName();

if (debug) {

this.logger.debug("Basic Authentication Authorization header found for user '" + username + "'");

}

/\*\*

\* 判断是否需要身份鉴定，true-需要鉴定

\* 1.从SecurityContext获取Authentication鉴权实例对象，如果为空或者未登陆，返回true

\* 2.如果是UsernamePasswordAuthenticationToken类型的实例对象，判断username和对象上的username是否一致，不一致返回true

\* 3.如果是AnonymousAuthenticationToken类型的实例对象，返回true

\* 4.其他情况返回false

\*/

if (authenticationIsRequired(username)) {

//从身份鉴定管理器获取身份鉴定对象，并放入上下文

/\*\*

\* 1.遍历找到UsernamePasswordAuthenticationToken的身份鉴定提供者AuthenticationProvider对象

\* 2.AbstractUserDetailsAuthenticationProvider#authenticate进行认证

\* 2-1. retrieveUser方法检索用户，调用了UserDedtailsService.loadUserByUsername,

\* 实际实现类ClientDetailsUserDetailsService里面调用了

\* ClientDetailsService.loadClientByClientId第1次

\*/

Authentication authResult = this.authenticationManager.authenticate(authRequest);

if (debug) {

this.logger.debug("Authentication success: " + authResult);

}

SecurityContextHolder.getContext().setAuthentication(authResult);

this.rememberMeServices.loginSuccess(request, response, authResult);

onSuccessfulAuthentication(request, response, authResult);

}

} catch (AuthenticationException failed) {

SecurityContextHolder.clearContext();

if (debug) {

this.logger.debug("Authentication request for failed: " + failed);

}

this.rememberMeServices.loginFail(request, response);

onUnsuccessfulAuthentication(request, response, failed);

if (this.ignoreFailure) {

chain.doFilter(request, response);

} else {

this.authenticationEntryPoint.commence(request, response, failed);

}

return;

}

chain.doFilter(request, response);

}

完成过滤器逻辑后，进入到org.springframework.security.oauth2.provider.endpoint.TokenEndpoint#postAccessToken

## UserDetailsService

## RedisTokenStore

##### **access:[AccessToken]（对应对象：OAuth2AccessToken）**

{ "access\_token": "x8U6xmAK0MeFDEJ0", "token\_type": "bearer", "refresh\_token": "0qLDRZE70MeFDEI!", "expires\_in": 29658, "scope": "server" }

##### **auth\_to\_access:[这里不再是AccessToken]**

key是将username、client\_id、scope三个值加密后的值，我们再看auth\_to\_access:存储的内容如下：

{ "access\_token": "x8U6xmAK0MeFDEJ0", "token\_type": "bearer", "refresh\_token": "0qLDRZE70MeFDEI!", "expires\_in": 24126, "scope": "server" }

跟access:的内容一模一样，那我们就知道了，代码内部实现可以通过username、client\_id、scope 3个字段获取AccessToken。

##### **auth:[accessToken] (对应对象:OAuth2Authentication)**

主要包含当前登录用户的信息，以及用户附带的角色和和权限信息、生成Token时的授权方式等信息。

{ "authorities": [{ "authority": "ROLE\_USER" }, { "authority": "USER\_RETRIEVE" } ], "details": null, "authenticated": true, "userAuthentication": { "authorities": [{ "authority": "ROLE\_USER" }, { "authority": "USER\_RETRIEVE" } ], "details": { "client\_secret": "app", "grant\_type": "password", "client\_id": "app", "username": "wuji" }, "authenticated": true, "principal": { "password": null, "username": "wuji", "authorities": [{ "authority": "ROLE\_USER" }, { "authority": "USER\_RETRIEVE" } ], "accountNonExpired": true, "accountNonLocked": true, "credentialsNonExpired": true, "enabled": true }, "credentials": null, "name": "wuji" }, "credentials": "", "principal": { "password": null, "username": "wuji", "authorities": [{ "authority": "ROLE\_USER" }, { "authority": "USER\_RETRIEVE" } ], "accountNonExpired": true, "accountNonLocked": true, "credentialsNonExpired": true, "enabled": true }, "oauth2Request": { "clientId": "app", "scope": ["server"], "requestParameters": { "grant\_type": "password", "client\_id": "app", "username": "wuji" }, "resourceIds": [], "authorities": [], "approved": true, "refresh": false, "redirectUri": null, "responseTypes": [], "extensions": {}, "grantType": "password", "refreshTokenRequest": null }, "clientOnly": false, "name": "wuji"}

##### **access\_to\_refresh:[accessToken]**

access\_to\_refresh:存储的内容很简单，在通过password等授权方式获取token时的refreshToken

{

//refresh\_token

grz0Xlzi0MeQwkx9

}

##### **refresh\_to\_access:[refreshToken]**

拿refreshToken去刷新accessToken时，会将新生成的accessToken放到refresh\_to\_access:

##### **refresh:[refreshToken]**

{ //refresh\_toekn

"value": "grz0Xlzi0MeQwkx9",

//过期时间戳（mills）

"expiration": 1557821765322

}

拿refreshToken去刷新accessToken时, 会先拿到这个KEY的信息，判断请求方的refresh token是否有效，无效的不能刷新access token。

##### **refresh\_auth:[refreshToken] (对应对象:OAuth2Authentication)**

refresh\_auth:存的内容与auth:类似

##### **client\_id\_to\_access:[client\_id]**

[{ "access\_token": "TesxUOBt0MeRDDxA", "token\_type": "bearer", "refresh\_token": "grz0Xlzi0MeQwkx9", "expires\_in": 41714, "scope": "server"}]

这个key将对应client\_id的AccessToken对象存储了起来，因为不同的username、scope使用同一个client\_id去请求获取token，所以这是一个list。

##### **uname\_to\_access:[client\_id:username]**

[{ "access\_token": "TesxUOBt0MeRDDxA", "token\_type": "bearer", "refresh\_token": "grz0Xlzi0MeQwkx9", "expires\_in": 41714, "scope": "server"}]这是一个list