**[SpringCloud之Zuul网关原理及其配置](https://www.cnblogs.com/jing99/p/11696192.html)**

　　Zuul是spring cloud中的微服务网关。网关： 是一个网络整体系统中的前置门户入口。请求首先通过网关，进行路径的路由，定位到具体的服务节点上。

　　Zuul是一个微服务网关，首先是一个微服务。也是会在Eureka注册中心中进行服务的注册和发现。也是一个网关，请求应该通过Zuul来进行路由。

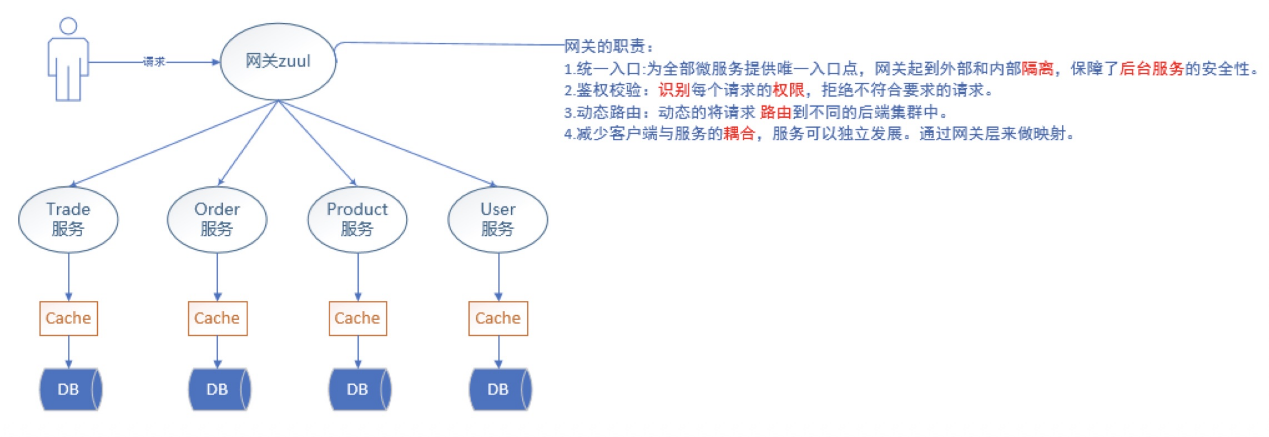
　　Zuul网关不是必要的。是推荐使用的。

　　使用Zuul，一般在微服务数量较多（多于10个）的时候推荐使用，对服务的管理有严格要求的时候推荐使用，当微服务权限要求严格的时候推荐使用。

**一、Zuul网关的作用**

　　网关有以下几个作用：

* 统一入口：为全部为服务提供一个唯一的入口，网关起到外部和内部隔离的作用，保障了后台服务的安全性。
* 鉴权校验：识别每个请求的权限，拒绝不符合要求的请求。
* 动态路由：动态的将请求路由到不同的后端集群中。
* 减少客户端与服务端的耦合：服务可以独立发展，通过网关层来做映射。



**二、Zuul网关的应用**

**1、网关访问方式**

　　通过zuul访问服务的，URL地址默认格式为：**http://zuulHostIp:port/要访问的服务名称/服务中的URL**

　　服务名称：properties配置文件中的spring.application.name。

　　服务的URL：就是对应的服务对外提供的URL路径监听。

**2、网关依赖注入**

[复制代码](javascript:void(0);)

<!-- spring cloud Eureka Client 启动器 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<!-- zuul网关的重试机制，不是使用ribbon内置的重试机制

是借助spring-retry组件实现的重试

开启zuul网关重试机制需要增加下述依赖

-->

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

</dependency>

[复制代码](javascript:void(0);)

**3、网关启动器**

[复制代码](javascript:void(0);)

/\*\*

\* @EnableZuulProxy - 开启Zuul网关。

\* 当前应用是一个Zuul微服务网关。**会在Eureka注册中心中注册当前服务。并发现其他的服务。**

\* Zuul需要的必要依赖是spring-cloud-starter-zuul。

\*/

@SpringBootApplication

**@EnableZuulProxy**

public class ZuulApplication {

public static void main(String[] args) {

SpringApplication.run(ZuulApplication.class, args);

}

}

[复制代码](javascript:void(0);)

**4、网关全局变量配置**

**4.1 URL路径匹配**

[复制代码](javascript:void(0);)

# URL pattern

# 使用路径方式匹配路由规则。

# 参数key结构： zuul.routes.customName.path=xxx

# 用于配置路径匹配规则。

# 其中customName自定义。通常使用要调用的服务名称，方便后期管理

# 可使用的通配符有： \* \*\* ?

**# ? 单个字符**

**# \* 任意多个字符，不包含多级路径**

**# \*\* 任意多个字符，包含多级路径**

zuul.routes.eureka-application-service.path=/api/\*\*

# 参数key结构： zuul.routes.customName.url=xxx

**# url用于配置符合path的请求路径路由到的服务地址。**

zuul.routes.eureka-application-service.url=http://127.0.0.1:8080/

[复制代码](javascript:void(0);)

**4.2 服务名称匹配**

[复制代码](javascript:void(0);)

# service id pattern 通过服务名称路由

# key结构 ： zuul.routes.customName.path=xxx

# 路径匹配规则

zuul.routes.eureka-application-service.path=/api/\*\*

# key结构 ： zuul.routes.customName.serviceId=xxx

**# serviceId用于配置符合path的请求路径路由到的服务名称。**

zuul.routes.eureka-application-service.serviceId=eureka-application-service

[复制代码](javascript:void(0);)

　　服务名称匹配也可以使用简化的配置：

# simple service id pattern 简化配置方案

**# 如果只配置path，不配置serviceId。则customName相当于服务名称。**

# 符合path的请求路径直接路由到customName对应的服务上。

zuul.routes.eureka-application-service.path=/api/\*\*

**4.3 路由排除配置**

[复制代码](javascript:void(0);)

# ignored service id pattern

**# 配置不被zuul管理的服务列表。多个服务名称使用逗号','分隔。**

# 配置的服务将不被zuul代理。

zuul.ignored-services=eureka-application-service

**# 此方式相当于给所有新发现的服务默认排除zuul网关访问方式，只有配置了路由网关的服务才可以通过zuul网关访问**

# 通配方式配置排除列表。

zuul.ignored-services=\*

# 使用服务名称匹配规则配置路由列表，相当于只对已配置的服务提供网关代理。

zuul.routes.eureka-application-service.path=/api/\*\*

**# 通配方式配置排除网关代理路径。所有符合ignored-patterns的请求路径都不被zuul网关代理。**

zuul.ignored-patterns=/\*\*/test/\*\*

zuul.routes.eureka-application-service.path=/api/\*\*

[复制代码](javascript:void(0);)

**4.4 路由前缀配置**

[复制代码](javascript:void(0);)

# prefix URL pattern 前缀路由匹配

**# 配置请求路径前缀，所有基于此前缀的请求都由zuul网关提供代理。**

zuul.prefix=/api

# 使用服务名称匹配方式配置请求路径规则。

**# 这里的配置将为：http://ip:port/api/appservice/\*\*的请求提供zuul网关代理，可以将要访问服务进行前缀分类。**

# 并将请求路由到服务eureka-application-service中。

zuul.routes.eureka-application-service.path=/appservice/\*\*

[复制代码](javascript:void(0);)

**5 Zuul网关配置总结**

　　网关配置方式有多种，**默认、URL、服务名称、排除|忽略、前缀**。

　　网关配置没有优劣好坏，应该在不同的情况下选择合适的配置方案。

**zuul网关其底层使用ribbon来实现请求的路由，并内置Hystrix，可选择性提供网关fallback逻辑。使用zuul的时候，并不推荐使用Feign作为application client端的开发实现。**毕竟**Feign技术是对ribbon的再封装，使用Feign本身会提高通讯消耗，降低通讯效率**，只在服务相互调用的时候使用Feign来简化代码开发就够了。而且**商业开发中，使用Ribbon+RestTemplate来开发的比例更高**。

**三、Zuul网关过滤器**

　　Zuul中提供了过滤器定义，可以用来过滤代理请求，提供额外功能逻辑。如：权限验证，日志记录等。

**Zuul提供的过滤器是一个父类。父类是ZuulFilter。通过父类中定义的抽象方法filterType，来决定当前的Filter种类是什么。**有前置过滤、路由后过滤、后置过滤、异常过滤。

* **前置过滤**：是请求进入Zuul之后，立刻执行的过滤逻辑。
* **路由后过滤**：是请求进入Zuul之后，并Zuul实现了请求路由后执行的过滤逻辑，路由后过滤，是在远程服务调用之前过滤的逻辑。
* **后置过滤**：远程服务调用结束后执行的过滤逻辑。
* **异常过滤**：是任意一个过滤器发生异常或远程服务调用无结果反馈的时候执行的过滤逻辑。无结果反馈，就是远程服务调用超时。

**3.1 过滤器实现方式**

继承父类ZuulFilter。在父类中提供了4个抽象方法，分别是：**filterType, filterOrder, shouldFilter, run**。其功能分别是：

* filterType：方法返回字符串数据，代表当前过滤器的类型。可选值有-pre, route, post, error。

pre - 前置过滤器，**在请求被路由前执行**，通常用于处理身份认证，日志记录等；

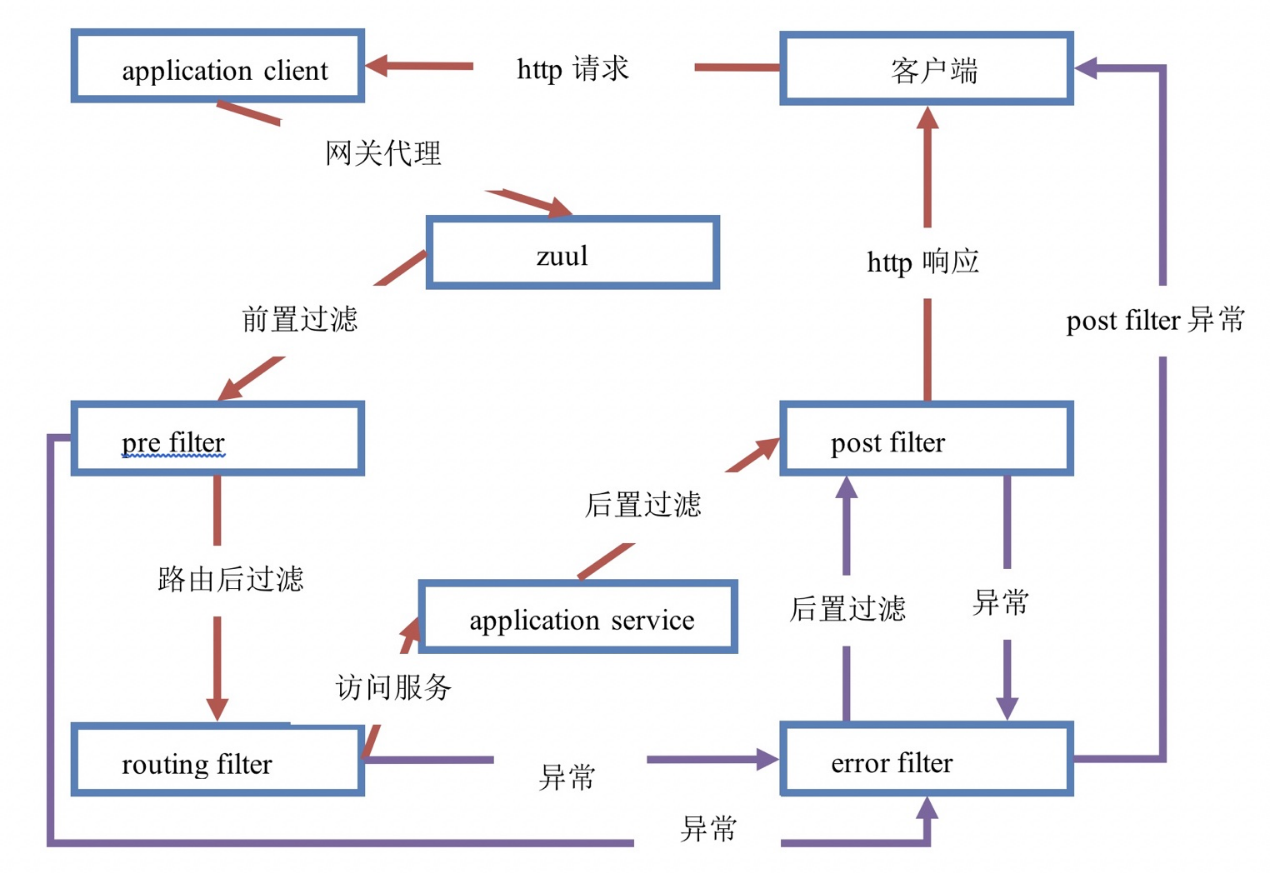
route - 在**路由执行后，服务调用前**被调用；

error - **任意一个filter发生异常**的时候执行或**远程服务调用没有反馈的时候执行（超时）**，通常用于处理异常；

post - 在**route或error执行后被调用**，一般用于收集服务信息，统计服务性能指标等，也可以对response结果做特殊处理。

* filterOrder：返回int数据，用于为**同filterType的多个过滤器定制执行顺序，返回值越小，执行顺序越优先**。
* shouldFilter：返回boolean数据，**代表当前filter是否生效**。
* run：具体的**过滤执行逻辑**。如pre类型的过滤器，可以通过对请求的验证来决定是否将请求路由到服务上；如post类型的过滤器，可以对服务响应结果做加工处理（如为每个响应增加footer数据）。

**3.2 过滤器的生命周期**



**3.3 代码示例**

[复制代码](javascript:void(0);)

/\*\*

\* Zuul过滤器，**必须继承ZuulFilter父类。**

\* 当前类型的对象**必须交由Spring容器管理。使用@Component注解描述。**

\* **继承父类后，必须实现父类中定义的4个抽象方法。**

\* **shouldFilter、 run、 filterType、 filterOrder**

\*/

@Component

public class LoggerFilter **extends ZuulFilter** {

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

/\*\*

\* 返回boolean类型。**代表当前filter是否生效。**

\* 默认值为false。

\* 返回true代表开启filter。

\*/

@Override

public boolean shouldFilter() {

**return true;**

}

/\*\*

\* run方法就是过滤器的具体逻辑。

\* return 可以返回任意的对象，**当前实现忽略**。（spring-cloud-zuul官方解释）

**\* 直接返回null即可。**

\*/

@Override

public Object run() throws ZuulException {

**// 通过zuul，获取请求上下文**

RequestContext rc = **RequestContext.getCurrentContext();**

HttpServletRequest request = **rc.getRequest()**;

logger.info("LogFilter1.....method={},url={}",

request.getMethod(),request.getRequestURL().toString());

// 可以记录日志、鉴权，给维护人员记录提供定位协助、统计性能

**return null;**

}

/\*\*

\* 过滤器的类型。可选值有：

**\* pre - 前置过滤**

**\* route - 路由后过滤**

**\* error - 异常过滤**

**\* post - 远程服务调用后过滤**

\*/

@Override

public String filterType() {

**return "pre";**

}

/\*\*

\* **同种类的过滤器的执行顺序。**

\* 按照返回值的**自然升序**执行。

\*/

@Override

public int filterOrder() {

**return 0;**

}

}

[复制代码](javascript:void(0);)

**四、Zuul网关的容错（与Hystrix的无缝结合）**

　　在spring cloud中，Zuul启动器中包含了Hystrix相关依赖，在Zuul网关工程中，默认是提供了Hystrix Dashboard服务监控数据的(hystrix.stream)，但是不会提供监控面板的界面展示。可以说，在spring cloud中，zuul和Hystrix是无缝结合的。

**4.1 Zuul中的服务降级处理**

　　在Edgware版本之前，Zuul提供了接口**ZuulFallbackProvider**用于实现fallback处理。从Edgware版本开始，Zuul提供了ZuulFallbackProvider的子接口**FallbackProvider**来提供fallback处理。  
　　Zuul的fallback容错处理逻辑，只针对timeout异常处理，当请求被Zuul路由后，**只要服务有返回（包括异常），都不会触发Zuul的fallback容错逻辑。**

　　因为对于Zuul网关来说，做请求路由分发的时候，结果由远程服务运算的。那么远程服务反馈了异常信息，Zuul网关不会处理异常，因为无法确定这个错误是否是应用真实想要反馈给客户端的。

**4.2 代码示例**

[复制代码](javascript:void(0);)

/\*\*

\* **如果需要在Zuul网关服务中增加容错处理fallback，需要实现接口ZuulFallbackProvider**

\* spring-cloud框架，**在Edgware版本(包括)之后，声明接口ZuulFallbackProvider过期失效，**

\* 提供了新的ZuulFallbackProvider的子接口 - **FallbackProvider**

\* 在老版本中提供的**ZuulFallbackProvider**中，定义了两个方法。

\* - **String getRoute()**

\* 当前的fallback容错处理逻辑处理的是哪一个服务。**可以使用通配符‘\*’代表为全部的服务提供容错处理。**

\* 如果只为某一个服务提供容错，返回对应服务的spring.application.name值。

\* - **ClientHttpResponse fallbackResponse()**

\* 当服务发生错误的时候，如何容错。

\* 新版本中提供的**FallbackProvider**提供了新的方法。

\* - **ClientHttpResponse fallbackResponse(Throwable cause)**

\* **如果使用新版本中定义的接口来做容错处理，容错处理逻辑，只运行子接口中定义的新方法。也就是有参方法。**

\* 是为远程服务发生异常的时候，通过异常的类型来运行不同的容错逻辑。

\*/

**@Component**

public class TestFallBbackProvider implements FallbackProvider {

/\*\*

\* return - 返回fallback处理哪一个服务。**返回的是服务的名称**

**\* 推荐 - 为指定的服务定义特性化的fallback逻辑。**

**\* 推荐 - 提供一个处理所有服务的fallback逻辑。**

**\* 好处 - 服务某个服务发生超时，那么指定的fallback逻辑执行。如果有新服务上线，未提供fallback逻辑，有一个通用的。**

\*/

@Override

public String **getRoute**() {

**return "eureka-application-service";**

}

/\*\*

\* fallback逻辑。在早期版本中使用。

\* Edgware版本之后，ZuulFallbackProvider接口过期，提供了新的子接口FallbackProvider

**\* 子接口中提供了方法ClientHttpResponse fallbackResponse(Throwable cause)。**

**\* 优先调用子接口新定义的fallback处理逻辑。**

\*/

@Override

public ClientHttpResponse fallbackResponse() {

System.out.println("ClientHttpResponse fallbackResponse()");

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("message", "服务正忙，请稍后重试");

result.add(data);

ObjectMapper mapper = new ObjectMapper();

String msg = "";

try {

msg = mapper.writeValueAsString(result);

} catch (JsonProcessingException e) {

msg = "";

}

return this.executeFallback(HttpStatus.OK, msg,

"application", "json", "utf-8");

}

/\*\*

\* fallback逻辑。**优先调用。**可以根据异常类型动态决定处理方式。

\*/

@Override

public ClientHttpResponse fallbackResponse(Throwable cause) {

System.out.println("ClientHttpResponse fallbackResponse(Throwable cause)");

if(cause instanceof NullPointerException){

List<Map<String, Object>> result = new ArrayList<>();

Map<String, Object> data = new HashMap<>();

data.put("message", "网关超时，请稍后重试");

result.add(data);

ObjectMapper mapper = new ObjectMapper();

String msg = "";

try {

msg = mapper.writeValueAsString(result);

} catch (JsonProcessingException e) {

msg = "";

}

return this.executeFallback(HttpStatus.GATEWAY\_TIMEOUT,

msg, "application", "json", "utf-8");

}else{

return this.fallbackResponse();

}

}

/\*\*

\* 具体处理过程。

\* @param **status 容错处理后的返回状态，如200正常GET请求结果，201正常POST请求结果，404资源找不到错误等。**

\* 使用spring提供的**枚举类型对象实现。HttpStatus**

\* @param **contentMsg 自定义的响应内容。就是反馈给客户端的数据。**

\* @param **mediaType** 响应类型，是**响应的主类型**， 如： application、text、media。

\* @param **subMediaType** 响应类型，是**响应的子类型**， 如： json、stream、html、plain、jpeg、png等。

\* @param **charsetName** 响应结果的字符集。这里只传递**字符集名称**，如： utf-8、gbk、big5等。

\* @return **ClientHttpResponse** 就是**响应的具体内容**。

\* **相当于一个HttpServletResponse。**

\*/

private final ClientHttpResponse executeFallback(**final HttpStatus status,**

**String contentMsg, String mediaType, String subMediaType, String charsetName**) {

return new ClientHttpResponse() {

/\*\*

\* 设置响应的头信息

\*/

@Override

public **HttpHeaders getHeaders**() {

HttpHeaders header = new HttpHeaders();

MediaType mt = new MediaType(mediaType, subMediaType, Charset.forName(charsetName));

header.setContentType(mt);

return header;

}

/\*\*

\* 设置响应体

\* zuul会将本方法返回的输入流数据读取，并通过**HttpServletResponse**的输出流输出到客户端。

\*/

@Override

public **InputStream getBody() throws IOException** {

String content = contentMsg;

return new ByteArrayInputStream(content.getBytes());

}

/\*\*

\* **ClientHttpResponse的fallback的状态码 返回String**

\*/

@Override

public String **getStatusText**() throws IOException {

return this.getStatusCode().**getReasonPhrase**();

}

/\*\*

\* **ClientHttpResponse的fallback的状态码 返回HttpStatus**

\*/

@Override

public HttpStatus **getStatusCode**() throws IOException {

return status;

}

/\*\*

\* **ClientHttpResponse的fallback的状态码 返回int**

\*/

@Override

public int **getRawStatusCode**() throws IOException {

return this.getStatusCode().value();

}

/\*\*

\* 回收资源方法

\* **用于回收当前fallback逻辑开启的资源对象的。**

\* **不要关闭getBody方法返回的那个输入流对象。**

\*/

@Override

public void close() {

}

};

}

}

[复制代码](javascript:void(0);)

**五、Zuul网关的限流保护**

　　Zuul网关组件也提供了限流保护。当请求并发达到阀值，自动触发限流保护，返回错误结果。只要提供error错误处理机制即可。

　　Zuul的限流保护需要额外依赖spring-cloud-zuul-ratelimit组件。

<dependency>

<groupId>com.marcosbarbero.cloud</groupId>

<artifactId>**spring-cloud-zuul-ratelimit**</artifactId>

<version>1.3.4.RELEASE</version>

</dependency>

**5.1 全局限流配置**

　　使用全局限流配置，zuul会对代理的所有服务提供限流保护。

[复制代码](javascript:void(0);)

# 开启限流保护

zuul.**ratelimit**.**enabled**=true

# 60s内请求超过3次，服务端就抛出异常，60s后可以恢复正常请求

zuul.ratelimit.default-policy.**limit**=3

zuul.ratelimit.default-policy.**refresh-interval**=60

# 针对IP进行限流，不影响其他IP

zuul.ratelimit.default-policy.**type**=**origin**

[复制代码](javascript:void(0);)

**5.2 局部限流配置**

　　使用局部限流配置，zuul仅针对配置的服务提供限流保护。

[复制代码](javascript:void(0);)

# 开启限流保护

zuul.ratelimit.enabled=true

# **hystrix-application-client服务**60s内请求超过3次，服务抛出异常。

zuul.ratelimit.policies.**hystrix-application-client**.limit=3

zuul.ratelimit.policies.**hystrix-application-client**.refresh-interval=60

# 针对IP限流。

zuul.ratelimit.policies.**hystrix-application-client**.type=origin

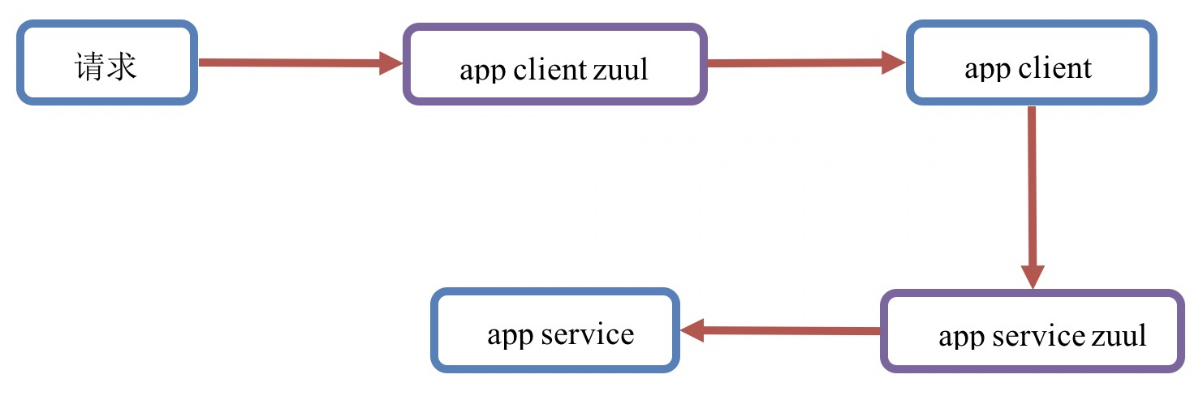
[复制代码](javascript:void(0);)

**5.3 限流参数简介**



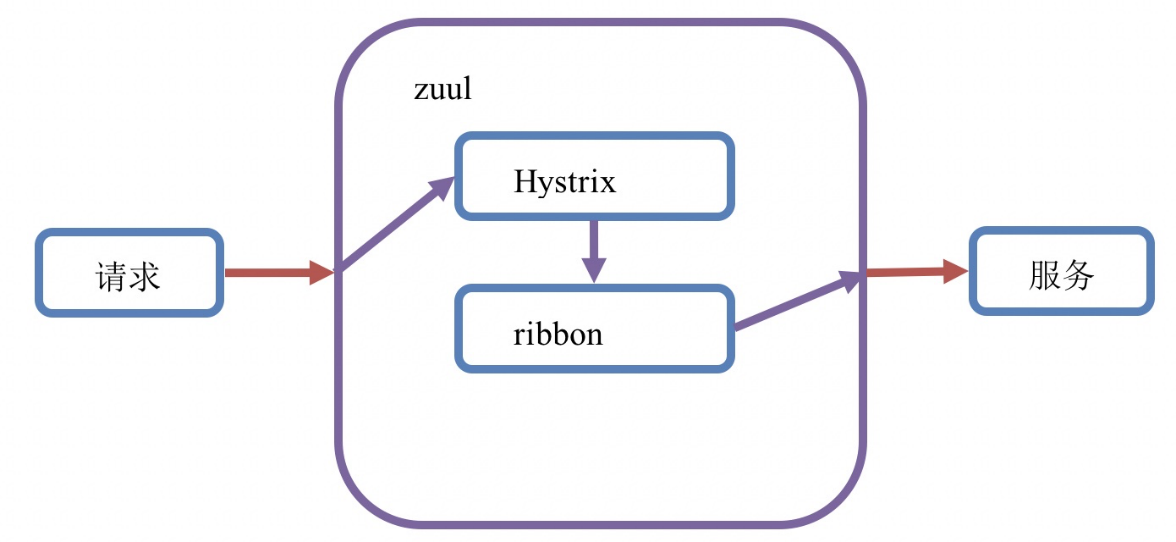
**六、Zuul网关性能调优：网关的两层超时调优**

　　使用Zuul的spring cloud微服务结构图：



　　从上图中可以看出。整体请求逻辑还是比较复杂的，在没有zuul网关的情况下，app client请求app service的时候，也有请求超时的可能。那么当增加了zuul网关的时候，请求超时的可能就更明显了。

　　当请求通过zuul网关路由到服务，并等待服务返回响应，这个过程中zuul也有超时控制。zuul的底层使用的是Hystrix+ribbon来实现请求路由。结构如下：



　　zuul中的Hystrix内部使用线程池隔离机制提供请求路由实现，其默认的超时时长为1000毫秒。ribbon底层默认超时时长为5000毫秒。**如果Hystrix超时，直接返回超时异常。如果ribbon超时，同时Hystrix未超时，ribbon会自动进行服务集群轮询重试，直到Hystrix超时为止。如果Hystrix超时时长小于ribbon超时时长，ribbon不会进行服务集群轮询重试。**

　　那么在zuul中可配置的超时时长就有两个位置：Hystrix和ribbon。具体配置如下：

[复制代码](javascript:void(0);)

**# 开启zuul网关重试**

zuul.**retryable**=true

# hystrix超时时间设置

# **线程池隔离**，默认超时时间1000ms

hystrix.command.default.execution.isolation.thread.**timeoutInMilliseconds**=8000

# ribbon超时时间设置：建议设置比Hystrix小

# **请求连接的超时时间**: 默认5000ms

ribbon.ConnectTimeout=5000

# **请求处理的超时时间**: 默认5000ms

ribbon.ReadTimeout=5000

# 重试次数：MaxAutoRetries表示访问服务集群下原节点（同路径访问）；MaxAutoRetriesNextServer表示访问服务集群下其余节点（换台服务器）

ribbon.MaxAutoRetries=1

ribbon.MaxAutoRetriesNextServer=1

# 开启重试

ribbon.OkToRetryOnAllOperations=true

[复制代码](javascript:void(0);)

　　Spring-cloud中的zuul网关重试机制是使用spring-retry实现的。工程必须依赖下述资源：

<dependency>

　　<groupId>org.springframework.retry</groupId>

　　<artifactId>**spring-retry**</artifactId>

</dependency>