## 17. Declarative REST Client: Feign

[Feign](https://github.com/Netflix/feign)是一个声明式的Web服务客户端。它让使用Web服务更简单。要使用Feign，需要创建一个接口并添加注解。它提供插拔式的Feign注解和JAX-RS注解支持。Feign还支持可插拔编码器和解码器。Spring Cloud增加了对Spring MVC注解的支持，以及在Spring Web项目中默认使用同样的HttpMessageConverters。Spring Cloud集成Ribbon和Eureka以在使用Feign时提供负载均衡的http客户端。

## 17.1 How to Include Feign

要在您的项目中使用Feign，需要引入group org.springframework.cloud artifact ID spring-cloud-starter-feign。

*@Configuration*

*@ComponentScan*

*@EnableAutoConfiguration*

*@EnableFeignClients*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

**StoreClient.java.**

*@FeignClient("stores")*

**public** **interface** StoreClient {

*@RequestMapping(method = RequestMethod.GET, value = "/stores")*

List<Store> getStores();

*@RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes = "application/json")*

Store update(*@PathVariable("storeId")* Long storeId, Store store);

}

@FeignClient注解的value（上文中的“stores”）是客户端名字（微服务application name），这个名字被用于创建Ribbon负载平衡器（[有关Ribbon支持的详细信息，](https://springcloud.cc/spring-cloud-dalston.html#spring-cloud-ribbon)请参阅[下文](https://springcloud.cc/spring-cloud-dalston.html#spring-cloud-ribbon)））。您还可以使用注解的url属性（值可为绝对路径或主机名 http://ip:port/api）指定URL。这些接口在Application context中的实例化的bean名称是该接口的完全限定名称。要指定您自己的别名值，您可以使用@FeignClient注解的qualifier属性。

Ribbon客户端将会寻找“stores”服务的物理地址。如果您的应用程序是Eureka客户端，那么它将在Eureka服务注册中解析该服务。如果您不想使用Eureka，你可以配置 stores的服务列表（细节参考16.6）。

## 17.2 Overriding Feign Defaults

Spring Cloud的Feign支持的核心概念是声明式客户端。每个feign客户端都是feign全体组件的一部分，它们一起工作以根据需要联系远程服务器，并且该集合有一个名字用在@FeignClient注解中。Spring cloud 每一个使用 FeignClientsConfiguration 生成的声明式客户端创建一个application context。这里面包含 一个 feign.Decoder, 一个 feign.Encoder 和一个 feign.Contract.

Spring cloud提供通过@FeignClient.添加添额外的配置的方法让你完全控制feign client。例如

*@FeignClient(name = "stores", configuration = FooConfiguration.class)*

**public** **interface** StoreClient {

*//..*

}

在这种情况下，client由FeignClientsConfiguration中的组件和FooConfiguration中的组件共同组成（后者会覆盖先者）。

* 警告：FooConfiguration不需要使用@Configuration注解。如果加上了，需要将它从@ComponentScan注解中排除否则它将会作为的 feign.Decoder, feign.Encoder, feign.Contract 等等组件的默认来源，如果加上了@Configuration注解，你可以将它放在一个分离的，非重叠性的 @ComponentScan 注解或者@SpringBootApplication 注解扫描包中，或者在@ComponentScan中显示的排除掉

注意：该 serviceId 已经过时，建议使用 name 属性

警告：以前，使用 url 属性，则 name 不是必须的，但现在是必须的.

name 和 url 属性都支持占位符。

*@FeignClient(name = "${feign.name}", url = "${feign.url}")*

**public** **interface** StoreClient {

*//..*

}

Spring Cloud Netfix 默认给 feign 提供下列的beans（BeanType beanName: ClassName）

Decoder feignDecoder: ResponseEntityDecoder (包装了SpringDecoder)  
*Encoder feignEncoder: SpringEncoder*  
Logger feignLogger: Slf4jLogger  
*Contrac feignContract: SpringMvcContract //因此可以试用spring mvc注解*  
Feign.Builder feignBuilder: HystrixFeign.Builder  
*Client feignClient: 如果开启了Ribbon使用LoadBalancerFeignClient, 否则使用默认的 feign Client.*

可以通过将feign.okhttp.enabled或feign.httpclient.enabled设置为true，并将它们放在类路径上来使用OkHttpClient和ApacheHttpClient feign客户端。你可以提供一个ClosableHttpClient 来自定义http client，当你使用apache or ok时。

Spring Cloud Netfix 默认没有给feign装配下列的beans，但是在创建feign 客户端时会在application context中查找并使用这些类型的bean:

* Logger.Level
* Retryer
* ErrorDecoder
* Request.Options
* Collection<RequestInterceptor>
* SetterFactory

创建这些类型的一个bean并放在@FeignClient配置中(如上FooConfiguration),允许你覆盖所描述的每一个bean. 例子:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

**public** Contract feignContract() {

**return** **new** feign.Contract.Default();

}

*@Bean*

**public** BasicAuthRequestInterceptor basicAuthRequestInterceptor() {

**return** **new** BasicAuthRequestInterceptor("user", "password");

}

}

这将SpringMvcContract替换为feign.Contract.Default，并将一个RequestInterceptor添加到RequestInterceptor的集合中。

@FeignClient 也可以在配置文件中设置

application.yml

feign:

client:

config:

feignName:

connectTimeout: 5000

readTimeout: 5000

loggerLevel: full

errorDecoder: com.example.SimpleErrorDecoder

retryer: com.example.SimpleRetryer

requestInterceptors:

- com.example.FooRequestInterceptor

- com.example.BarRequestInterceptor

decode404: **false**

可以在@EnableFeignClients属性defaultConfiguration中以与上述相似的方式指定默认配置。不同之处在于，此配置将适用于feign客户端。

如果你选择使用配置文件设置所有的@FeignClient，你可以使用 “default” feign name（可以针对某一个feignclient设置）

application.yml

feign:

client:

config:

default:

connectTimeout: 5000

readTimeout: 5000

loggerLevel: basic

如果你同时创建了@Configuration bean和配置文件，配置文件优先级搞，它将会覆盖@Configuration bean属性。如果你要改变优先级策略，你可以设置feign.client.default-to-properties为 false.

注意: 如果你需要在 RequestInterceptor 中使用 ThreadLocal 去绑定变量，你需要设置对应的hystrix线程隔离策略为“SEMAPHORE”或取消Hystrix在Feign中的使用.

application.yml

*# To disable Hystrix in Feign*

feign:

hystrix:

enabled: **false**

*# To set thread isolation to SEMAPHORE*

hystrix:

command:

default:

execution:

isolation:

strategy: SEMAPHORE

## 17.3 Creating Feign Clients Manually

在一些情况下可能需要自定义Feign clients但是不能用以上的方法。所以你可以使用[Feign Builder API](https://github.com/OpenFeign/feign/#basics)创建clients。下面是一个例子，创建了两个相同接口的client但是用配置了不同的拦截器。

*@Import(FeignClientsConfiguration.class)*

**class** FooController {

**private** FooClient fooClient;

**private** FooClient adminClient;

*@Autowired*

**public** FooController(

Decoder decoder, Encoder encoder, Client client) {

**this**.fooClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.requestInterceptor(**new** BasicAuthRequestInterceptor("user", "user"))

.target(FooClient.**class**, "http://PROD-SVC");

**this**.adminClient = Feign.builder().client(client)

.encoder(encoder)

.decoder(decoder)

.requestInterceptor(**new** BasicAuthRequestInterceptor("admin", "admin"))

.target(FooClient.**class**, "http://PROD-SVC");

}

}

|  |  |
| --- | --- |
| 注意 | 在上面的例子中，FeignClientsConfiguration.class是Spring Cloud Netflix提供的默认配置。 |
| 注意 | PROD-SVC是客户端将要求的服务的名称。 |

## 17.4 Feign Hystrix Support

如果Hystrix在classpath中并设置feign.hystrix.enabled=true, Feign用熔断器包装所有方法。返回一个 com.netflix.hystrix.HystrixCommand。这允许你以响应模式使用（使用.toObservable()或者.observer()）或者 异步调用（.queue()）。

要在一个客户端上禁用 Hystrix 支持，创建一个 Feign.Builder 并将scope 设置为”prototype”,例如:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

*@Scope("prototype")*

**public** Feign.Builder feignBuilder() {

**return** Feign.builder();

}

}

注意:在Spring Cloud Dalston版本之前，只要Hystrix 存在你项目的classpath里面，Feign默认会给所有的方法都加上熔断器(circuit breaker),这个默认的行为在Spring Cloud Dalston版本中变为可选择的方法

## 17.5 Feign Hystrix Fallbacks

Hystrix支持服务降级：当执行方法出错或者断点为开启状态时一段默认路径的代码将会被执行，可以通过在@FeignClient注解上设置fallback属性为对应的实现类上来开启回调功能，需要声明这个实现类为spring bean.

*@FeignClient(name = "hello", fallback = HystrixClientFallback.class)*

**protected** **interface** HystrixClient {

*@RequestMapping(method = RequestMethod.GET, value = "/hello")*

Hello iFailSometimes();

}

**static** **class** HystrixClientFallback **implements** HystrixClient {

*@Override*

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback");

}

}

如果你想获取造成服务降级的原因，你可以使用@FeignClient注解的fallbackFactory属性：

*@FeignClient(name = "hello", fallbackFactory = HystrixClientFallbackFactory.class)*

**protected** **interface** HystrixClient {

*@RequestMapping(method = RequestMethod.GET, value = "/hello")*

Hello iFailSometimes();

}

*@Component*

**static** **class** HystrixClientFallbackFactory **implements** FallbackFactory<HystrixClient> {

*@Override*

**public** HystrixClient create(Throwable cause) {

**return** **new** HystrixClient() {

*@Override*

**public** Hello iFailSometimes() {

**return** **new** Hello("fallback; reason was: " + cause.getMessage());

}

};

}

}

注意: Feign fallbacks有一个限制。Fallbacks目前不支持返回 com.netflix.hystrix.HystrixCommand 和 rx.Observable 类的方法

## 17.6 Feign and @Primary

当使用Feign与Hystrix服务降级时，在同一类型的ApplicationContext中有多个bean。这将导致@Autowired无法起作用，因为不只有一个bean，或者有多个bean却没有标记其中一个为主。要解决这个问题，Spring Cloud Netflix将所有Feign实例标记为@Primary，所以Spring Framework将知道要注入哪个bean。在某些情况下并不需要。要关闭此行为，将@FeignClient的primary属性设置为false。

*@FeignClient(name = "hello", primary = false)*

**public** **interface** HelloClient {

*// methods here*

}

## 17.7 Feign Inheritance Support

Feign 通过单独继承接口支持apis模板，这可以使得将共同的操作方法都放在基础接口里面：

**UserService.java.**

**public** **interface** UserService {

*@RequestMapping(method = RequestMethod.GET, value ="/users/{id}")*

User getUser(*@PathVariable("id")* **long** id);

}

**UserResource.java.**

*@RestController*

**public** **class** UserResource **implements** UserService {

}

**UserClient.java.**

**package** project.user;

*@FeignClient("users")*

**public** **interface** UserClient **extends** UserService {

}

注意：通常不建议在服务器和客户端之间共享接口。它引入了紧耦合，并且实际上并不能与Spring MVC配合工作（方法参数映射不被继承）。（意思是：弃掉接口，服务器端与feign client各自定义，服务器直接实现逻辑，feign client可以只定义需要的接口）

## 17.8 Feign request/response compression

您可以考虑为Feign请求启用请求或响应GZIP压缩。您可以通过启用其中一个属性来执行此操作：

feign.compression.request.enabled=true

feign.compression.response.enabled=true

Feign请求压缩为您提供与您为Web服务器设置的设置相似的设置：

feign.compression.request.enabled=true

feign.compression.request.mime-types=text/xml,application/xml,application/json

feign.compression.request.min-request-size=2048

这些属性允许你选择要压缩的 MIME-TYPE 和最小的请求长度。

## 17.9 Feign logging

每个Feign client都创建了一个logger。默认的logger的名字是创建Feign client的接口的完整类名。Feign日志只响应 DEBUG 级别。

**application.yml.**

logging.level.project.user.UserClient: DEBUG

这个 Logger.Level 对象可以在每个客户端中进行配置，不同的等级对应不同的日志量:

NONE, 不记录(默认)

BASIC, 仅记录请求的方法和地址以及响应的状态码和执行时间

HEADERS, 单独的记录请求和响应头的信息

FULL, 记录请求和响应的请求头，请求体和元数据

下面演示设置Logger.Level 为 FULL 类型:

*@Configuration*

**public** **class** FooConfiguration {

*@Bean*

Logger.Level feignLoggerLevel() {

**return** Logger.Level.FULL;

}

}

## 18. External Configuration: Archaius

Archaius是Netflix client side类库。它的类库可以被所有的Netflix OSS组件使用。Archaius是 [Apache Commons Configuration](http://commons.apache.org/proper/commons-configuration) 的扩展项目。它允许通过轮询或者推送把配置变更推送到client来更新配置。Archaius使用Dynamic<Type>Property相关类来来处理属性。

**Archaius Example.**

**class** ArchaiusTest {

DynamicStringProperty myprop = DynamicPropertyFactory

.getInstance()

.getStringProperty("my.prop");

**void** doSomething() {

OtherClass.someMethod(myprop.get());

}

}

Archaius具有自己的一组配置文件和加载优先级。Spring应用程序一般不应直接使用Archaius，但本身仍然有配置Netflix工具的需求。Spring Cloud具有Spring Environment Bridge，所以Archaius可以从Spring environment读取属性。这允许Spring Boot项目使用标准配置工具链，同时允许他们在配置文件中配置Netflix工具。

## 19. Router and Filter: Zuul

路由在微服务体系结构的一个组成部分。例如，/可以映射到您的Web应用程序，/api/users映射到用户服务，并将/api/shop映射到商店服务。[Zuul](https://github.com/Netflix/zuul)是Netflix的基于JVM的路由器和服务器端负载均衡器

[Netflix使用Zuul](http://www.slideshare.net/MikeyCohen1/edge-architecture-ieee-international-conference-on-cloud-engineering-32240146/27)进行以下操作：

认证

洞察

压力测试

金丝雀测试

动态路由

服务迁移

负载脱落

安全

静态响应处理

主动/主动流量管理

Zuul的规则和过滤器允许使用各种基于JVM的语言，支持基于Java和Groovy。

注意：配置属性zuul.max.host.connections已被两个新属性zuul.host.maxTotalConnections和zuul.host.maxPerRouteConnections替换，分别默认为200和20。

注意：默认所有路由的 Hystrix 隔离模式为 “SEMAPHORE”, 可以修改 zuul.ribbonIsolationStrategy 为 “THREAD” 将隔离级别改为 “THREAD” 模式。

## 19.1 How to Include Zuul

<dependency>

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

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

</dependency>

## 19.2 Embedded Zuul Reverse Proxy

Spring Cloud已经创建了一个嵌入式Zuul代理，以简化前端UI应用调用一个或多个后端服务。这个功能对于代理前端需要访问的后端服务非常有用，避免了所有后端服务需要关心管理CORS和认证的问题.

在Spring Boot主函数上通过注解 @EnableZuulProxy 来开启, 这样可以让本地的请求转发到适当的服务. 按照约定, 一个ID为”users”的服务会收到 /users 请求路径的代理请求(前缀会被剥离). Zuul使用Ribbon定位服务注册中的实例, 并且所有的请求都在hystrix的command中执行, 所以失败信息将会展现在Hystrix metrics中, 并且一旦断路器打开, 代理请求将不会尝试去链接服务.

注意:  Zuul 的starter 包里面不包含注册中心的客户端，所以如果你想基于service IDs去做路由还需要将注册中心对应的客户端的包放在项目的classpath里(比如说Eureka)。

去忽略一个自动添加的服务，可以在服务ID表达式列表中设置 zuul.ignored-services。如果一个服务匹配到了要忽略的列表, 但是它也明确的配置在路由列表中, 将不会被忽略, 例如:

application.yml

zuul:

ignoredServices: '\*'

routes:

users: /myusers/\*\*

在这个例子中，所有的服务都会被忽略，除了“users”。

增加或改变代理路由规则, 你可以添加类似下面的外部配置:

application.yml

zuul:

routes:

users: /myusers/\*\*

这表示，HTTP调用 “/myusers” 会转发到 “user” 服务（例如：”/myusers/101”转发到”/101”）

要获得对路由的更细粒度的控制，您可以独立地指定路径和serviceId：

application.yml

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users\_service

这意味着对“/myusers”的http调用转发到“users\_service”服务。路由必须有一个“路径”，可以指定为ant-style pattern，所以“/ myusers / \*”只匹配一个级别，但“/ myusers / \*\*”分层匹配。

后端的配置既可以是”serviceId”(对于服务发现中的服务), 或者可以是”url”(物理地址), 例如:

application.yml

zuul:

routes:

users:

path: /myusers/\*\*

url: http://example.com/users\_service

上面这种配置方式不会执行HystrixCommand，也会使用Ribbon对多个URL进行负载均衡。可以通过制定server列表并禁用eureka

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

serviceId: users

ribbon:

eureka:

enabled: **false**

users:

ribbon:

listOfServers: example.com,google.com

你可以使用regexmapper提供serviceId和routes之间的绑定. 它使用正则表达式组来从serviceId提取变量, 然后注入到路由表达式中.

**ApplicationConfiguration.java.**

*@Bean*

**public** PatternServiceRouteMapper serviceRouteMapper() {

**return** **new** PatternServiceRouteMapper(

"(?<name>^.+)-(?<version>v.+$)",

"${version}/${name}");

}

这表示serviceId “myusers-v1” 将会被映射到 “/v1/myusers/**“**.任何正则表达式都可以，但是所有的命名组都必须在servicePattern和routePattern中存在。如果servicePattern没有匹配到一个serviceId，默认的行为会被启用。在上面的例子中，serviceId” users”将会映射到”/myusers/\*\*“(路由时不监测版本号)这个特性默认是禁用的，而且只能用于有注册中心的zuul中。

给所有映射添加前缀，可以设置 zuul.prefix 一个值，比如/api。这个前缀默认在请求跳转之前会删除（通过 zuul.stripPrefix=false 可以关闭这个功能）。你也可以在单个服务中关闭这个功能, 例如:

application.yml

zuul:

routes:

users:

path: /myusers/\*\*

stripPrefix: false

注意：zuul.stripPrefix只适用于使用了zuul.prefix配置情况下。在一个定义好了的 route’s path中不会有任何影响。

在本示例中，对“/ myusers / 101”的请求将转发到“users”服务上的“/ myusers / 101”。

zuul.routes 实际上绑定到类型为 ZuulProperties 的对象上. 如果你查看这个对象你会发现一个叫”retryable”的字段, 设置为”true”会使Ribbon客户端自动在失败时重试(如果需要，可以使用Ribbon client configuration修改重试操作的参数)。

每个请求header里面默认会附带一个 “X-Forwarded-Host”属性, 可以通过设置 zuul.addProxyHeaders=false 来关闭它. 虽然路径的前缀默认会被删除的,但是请求后端的时候header里面还是会附带一个”X-Forwarded-Prefix”属性。

如果设置了默认路由（“/”），则@EnableZuulProxy的应用程序可以作为独立服务器，例如zuul.route.home: /，会将所有请求（即“/ \*\*”）路由到“home”服务。

如果需要更细粒度的忽略配置，你可以指定特殊的表达式来配置忽略规则.这些表达式从route location的开始进行匹配，意味着前缀应该包括在匹配表达式中才能保证匹配到. 忽略表达式影响所有服务和取代任何路由的特殊配置.

**application.yml.**

zuul:

ignoredPatterns: /\*\*/admin/\*\*

routes:

users: /myusers/\*\*

这个的意思是所有请求, 比如”/myusers/101”的请求会跳转到”users”服务的”/101”, 但包含”/admin/”的请求将不被处理.

警告：yaml文件可以保留配置文件中路由顺序，properties文件会丢失顺序。

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

legacy:

path: /\*\*

如果要使用properties文件，则legacy路径可能会排在users路径前面，从而使users路径不可达。

## 19.3 Zuul Http Client

默认的zull的Http clietn现在是Apach HTTP Client,替代了已过期的Ribbon RestClient。想使用RestClient或使用okhttp3.OKHttpClient,可以设置ribbon.restclient.enable=true或者ribbon.okhttp.enable=true。可以自定义Http client。

## 19.4 Cookies and Sensitive Headers

在同一个系统的多个服务之间中分享headers是可以的，但是你可能不想把一些敏感headers泄露到下游服务器。你可以在路由配置中指定一批忽略的headers列表。Cookies扮演了一个特殊的角色, 因为它们在浏览器中具有明确的语义, 而且它们总是被认为是敏感的. If the consumer of your proxy is a browser, then cookies for downstream services also cause problems for the user because they all get jumbled up (all downstream services look like they come from the same place).

如果你对于你的服务设计很细心，比如，如果只有一个下游的服务设置了cookies，你可能会让它从后端服务一直追溯到前端调用者，同样的，如果你的代理设置了cookies而且所有你的后端服务都是同一系统的一部分，它可以很自然的共享（比如使用spring session去联系一些共享状态）。除此之外，任何下游服务设置的cookies可以能不会对前端调用者有用。所以建议你将”Set-Cookie”和”Cookie”(至少一个)放入路由中不属于同一个域里面的服务的敏感头信息设置里面,即使对于路由服务来说它和后端服务也是处于同一个域里面,仔细想想这样的做法对前面使得cookies一直在流动的做法有什么意义。

敏感头是一个用逗号隔开的列表表示,例如:

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders: Cookie,Set-Cookie,Authorization

url: https://downstream

注意：这是sensitiveHeaders的默认值，因此您不需要设置它，除非您希望它不同。

sensitiveHeaders是一个黑名单，默认值不为空，所以要使Zuul发送所有headers（“被忽略”除外），您必须将其显式设置为空列表。如果您想将pass cookie or authorization headers 传递到后端,例：

**application.yml.**

zuul:

routes:

users:

path: /myusers/\*\*

sensitiveHeaders:

url: https://downstream

zuul.sensitiveHeaders设置全局敏感头。如果在路由上设置sensitiveHeaders，则将覆盖全局sensitiveHeaders设置。

## 19.5 Ignored Headers

除了对每个路由设置敏感头之外,你可以为 zuul.ignoredHeaders设置一个调用下游服务时丢弃的值 (包括请求和响应).默认这个值是空的,如果你项目中没有使用Spring Security, 否则Spring Security会在请求头设置一组”security”头 (例如caching相关 Pragma,Cache-Control,X-Frame-Options,X-Content-Type-Options,X-XSS-Protection,Expires) 。假设Spring Security 在类路径上，为了不丢弃这些 security headers，您可以将zuul.ignoreSecurityHeaders设置为false。这个设置在下面这种情况下非常有用：如果您禁用Spring Security 的HTTP Security response headers，并希望由下游服务设置securit值.

（事实上sensitive-headers会被添加到ignored-headers中。）

## 19.6 Management Endpoints

当你同时在项目中使用了 @EnableZuulProxy 注解和 Spring Boot Actuator 时默认情况下你会开启两个额外的端点

* Routes
* Filters

### 19.6.1 Routes Endpoint

**GET /routes.**

**{**

/stores/\*\*: "http://localhost:8081"

**}**

**GET /routes?format=details.**

**{**

"/stores/\*\*": **{**

"id": "stores"**,**

"fullPath": "/stores/\*\*"**,**

"location": "http://localhost:8081"**,**

"path": "/\*\*"**,**

"prefix": "/stores"**,**

"retryable": **false,**

"customSensitiveHeaders": **false,**

"prefixStripped": **true**

**}**

**}**

注意：虽然当服务目录里面路由信息改变时,它是会自动刷新的,但是POST访问”/routes”端点会使这个改变立即生效.

### 19.6.2 Filters Endpoint

**GET /** filters 

## 19.7 Strangulation Patterns and Local Forwards

一个常见的迁移旧应用或者旧接口的方式，就是逐步的替换它的实现。Zuul代理是一个有用的工具，因为您可以使用它来处理来自旧端点的客户端的所有流量，但将一些请求重定向到新端点。

**application.yml.**

zuul:

routes:

first:

path: /first/\*\*

url: http://first.example.com

second:

path: /second/\*\*

url: forward:/second

third:

path: /third/\*\*

url: forward:/3rd

legacy:

path: /\*\*

url: http://legacy.example.com

在这个例子展示了如何扼杀“legacy”app,起先所有的路由/\*\*都映射到它上面。现在，“/first/\*\*” 路由到一个指向外部URL的新服务上，“/second/\*\*” 被转发到本地处理，例如被一个正常的@RequestMapping处理。/third/\*\*中的路径也被转发，且具有不同的前缀（即/third/foo转发到/3rd/foo）。

注意：忽略表达式并不是完全的忽略请求, 只是配置这个代理不处理这些请求。

## 19.8 Uploading Files through Zuul

如果你使用 @EnableZuulProxy , 你可以使用代理路径上传文件, 对于小文件可以正常使用. 对于大文件有可选的路径”/zuul/*“绕过Spring DispatcherServlet (避免处理multipart). 比如对于 zuul.routes.customers=/customers/\** , 你可以使用 “/zuul/customers/\*” 去上传大文件. Servlet路径通过 zuul.servletPath 指定. 如果使用Ribbon负载均衡器的代理路由, 在 处理非常大的文件时, 仍然需要提高超时配置. 比如:

**application.yml.**

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000

ribbon:

ConnectTimeout: 3000

ReadTimeout: 60000

请注意，要使用大型文件进行流式传输，您需要在请求中使用分块编码（某些浏览器默认情况下不会执行）。例如在命令行

curl -v -H "Transfer-Encoding: chunked" \

-F "file=@mylarge.iso" localhost:9999/zuul/simple/file

## 19.9 Query String Encoding

当处理请求时，查询参数将被解码以遍在Zuul的filters中进行处理修改。然后在路由过滤器中构建后端请求时重新编码。如果使用Javascript的encodeURIComponent()方法编码，结果可能与原始输入不同，大多数情况下这样做并不会造成什么问题，但是有些web服务器对请求的参数是严格需要编码的。

**application.yml.**

zuul:

forceOriginalQueryStringEncoding: **true**

**注意：**这个特殊的标志只在SimpleHostRoutingFilter 启用的时候才有用，您可以使用RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)轻松覆盖查询参数，因为查询字符串现在直接在原始的HttpServletRequest上获取。

## 19.10 Plain Embedded Zuul

运行zuul服务，不使用代理。- 暂不翻译

## 19.11 Disable Zuul Filters

Spring Cloud的Zuul在代理和服务器模式下默认启用了多个ZuulFilter bean。有关启用的可能过滤器，请参阅[zuul过滤器包](https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters)（https://github.com/spring-cloud/spring-cloud-netflix/tree/master/spring-cloud-netflix-zuul/src/main/java/org/springframework/cloud/netflix/zuul/filters）。如果要禁用它，只需设置zuul.<SimpleClassName>.<filterType>.disable=true。按照惯例，filters之后的包是Zuul过滤器类型。例如，禁用org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter设置zuul.SendResponseFilter.post.disable=true。

## 19.12 Providing Hystrix Fallbacks For Routes

当zuul的某个路由熔断时你可以创建一个ZuulFallbackProvider类型的bean来提供熔断响应，在这个bean里面你需要指定路由ID，并返回一个ClientHttpResponse返回响应，下面是一个简单的ZuulFallbackProvider实现类:

**class** MyFallbackProvider **implements** ZuulFallbackProvider {

*@Override*

**public** String getRoute() {

**return** "customers";

}

*@Override*

**public** ClientHttpResponse fallbackResponse() {

**return** **new** ClientHttpResponse() {

*@Override*

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

**return** HttpStatus.OK;

}

*@Override*

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

**return** 200;

}

*@Override*

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

**return** "OK";

}

*@Override*

**public** **void** close() {

}

*@Override*

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

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

*@Override*

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

对应的路由信息配置如下:

zuul:

routes:

customers: /customers/\*\*

如果您希望为所有路由提供默认的回退，您可以创建一个类型为ZuulFallbackProvider的bean，并且getRoute方法返回\*或null。

**class** MyFallbackProvider **implements** ZuulFallbackProvider {

*@Override*

**public** String getRoute() {

**return** "\*";

}

*@Override*

**public** ClientHttpResponse fallbackResponse() {

**return** **new** ClientHttpResponse() {

*@Override*

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

**return** HttpStatus.OK;

}

*@Override*

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

**return** 200;

}

*@Override*

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

**return** "OK";

}

*@Override*

**public** **void** close() {

}

*@Override*

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

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

*@Override*

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

可以使用FallbackProvider来根据失败原因来返回熔断响应

**class** MyFallbackProvider **implements** FallbackProvider {

*@Override*

**public** String getRoute() {

**return** "\*";

}

*@Override*

**public** ClientHttpResponse fallbackResponse(**final** Throwable cause) {

**if** (cause **instanceof** HystrixTimeoutException) {

**return** response(HttpStatus.GATEWAY\_TIMEOUT);

} **else** {

**return** fallbackResponse();

}

}

*@Override*

**public** ClientHttpResponse fallbackResponse() {

**return** response(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

**private** ClientHttpResponse response(**final** HttpStatus status) {

**return** **new** ClientHttpResponse() {

*@Override*

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

**return** status;

}

*@Override*

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

**return** status.value();

}

*@Override*

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

**return** status.getReasonPhrase();

}

*@Override*

**public** **void** close() {

}

*@Override*

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

**return** **new** ByteArrayInputStream("fallback".getBytes());

}

*@Override*

**public** HttpHeaders getHeaders() {

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

**return** headers;

}

};

}

}

## 19.13 Zuul Timeouts

### 19.13.1 Service Discovery Configuration

如果zuul使用了服务发现，那么需要关心两个超时，hystrix timeout 和 ribbon timeout，默认情况下，zuul会自动计算hystrix timeout除非你设置了hystrix timeout。

hystrix timeout计算公式

(ribbon.ConnectTimeout + ribbon.ReadTimeout) \* (ribbon.MaxAutoRetries + 1) \* (ribbon.MaxAutoRetriesNextServer + 1)

如果你按照以下方式设置

**application.yml.**

ribbon:

ReadTimeout:100

ConnectTimeout:500

MaxAutoRetries:1

MaxAutoRetriesNextServer:1

那么所有路由的hystrix timeout为2400ms

注意：可以通过 service.ribbon.\* 配置一个路由的hystrix timeout

注意：如果没有配置以上属性，那么hystrix timeout默认为4000ms

Hystrix配置项：

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

hystrix.command.commandKey.execution.isolation.thread.timeoutInMilliseconds commandKey为路由id

### 19.13.2 URL Configuration

如果通过指定url配置路由，那么需要配置 zuul.host.connect-timeout-millis and zuul.host.socket-timeout-millis

## 19.14 Rewriting Location header

如果Zuul在一个web应用前面，那么你需要重写Location头部当你的web应用通过HTTP状态码3XX重定向。否则，浏览器会重定向到web应用的URL而不是Zuul的URL。

可以通过配置一个LocationRewriteFilter类型的Zuul过滤器来重写Location头部到Zuul的URL。它还恢复了删除的全局前缀和特定于路由的前缀。如下：

**import** org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;

...

*@Configuration*

*@EnableZuulProxy*

**public** **class** ZuulConfig {

*@Bean*

**public** LocationRewriteFilter locationRewriteFilter() {

**return** **new** LocationRewriteFilter();

}

}

注意：要非常小心使用这个过滤器，因为它会作用于所有响应码为3XX的Location头部，这可能在某些场合不适合。比如要重定向到一个外部地址。

## 19.15 Zuul Developer Guide

有关Zuul如何工作的一般概述，请参阅[Zuul Wiki](https://github.com/Netflix/zuul/wiki/How-it-Works)。https://github.com/Netflix/zuul/wiki/How-it-Works

### 19.15.1 The Zuul Servlet

Zuul是作为Servlet实现的。 对于一般情况，Zuul嵌入到Spring Dispatch机制中。 这让Spring MVC可以控制路由。 在这种情况下，Zuul被配置为缓冲请求。 如果需要在没有缓冲请求的情况下通过Zuul（例如，对于大型文件上载），Servlet也可以绕过Spring Dispatcher被访问， 默认地址为/zuul。 可以使用zuul.servlet-path属性更改此路径。

### 19.15.2 Zuul RequestContext

为了在过滤器之间传递信息，Zuul使用[RequestContext](https://github.com/Netflix/zuul/blob/1.x/zuul-core/src/main/java/com/netflix/zuul/context/RequestContext.java)。 它的数据保存在特定于每个请求的ThreadLocal中。 有关路由目的地，错误以及HttpServletRequest和HttpServletResponse的信息都存储在那里。 RequestContext集成于 ConcurrentHashMap，因此任何东西都可以存储在上下文中。 [FilterConstants](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/zuul/filters/support/FilterConstants.java)包含Spring Cloud Netflix安装的过滤器使用的key（[稍后](http://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/2.0.1.RELEASE/single/spring-cloud-netflix.html#zuul-developer-guide-enable-filters)将详细介绍)。

### 19.15.3 @EnableZuulProxy vs. @EnableZuulServer

Spring Cloud Netflix 在你以不同的注解使用Zuul时会自动安装一些filter。 @EnableZuulProxy是@EnableZuulServer的超集。 换句话说，@EnableZuulProxy包含@EnableZuulServer安装的所有过滤器。 “代理”中的其他过滤器启用路由功能。 如果你想要一个“空白”Zuul，你应该使用@EnableZuulServer。

### 19.15.4 @EnableZuulServer Filters

SimpleRouteLocator 类用来从Spring Boot中加载路由配置信息。

安装了以下过滤器（与普通的Spring Bean一样）：

预过滤器：

ServletDetectionFilter: 检测请求是否通过Spring Dispatcher。

通过FilterConstants.IS\_DISPATCHER\_SERVLET\_REQUEST\_KEY设置

FormBodyWrapperFilter: 解析表单数据并为下游请求重新编码。

DebugFilter: 检测请求参数中是否含有Debug，如果有的话，则将RequestContext.setDebugRouting()和RequestContext.setDebugRequest()设置为true。

路由过滤器：

SendForwardFilter: 使用Servlet RequestDispatcher转发请求。 转发位置存储在RequestContext属性FilterConstants.FORWARD\_TO\_KEY中。 这对于转发到当前应用程序中的端点很有用。

后置过滤器：

SendResponseFilter: 将代理请求的响应写入当前响应。

错误过滤器：

SendErrorFilter: 如果RequestContext.getThrowable()不为null，则转发到/error（默认情况下）。 您可以通过设置error.path属性来更改默认转发路径。

### 19.15.5 @EnableZuulProxy Filters

DiscoveryClientRouteLocator 类用来从注册中心(例如Eureka)加载路由配置信息，和从配置文件加载一样。每个serviceId都是一个路由，每当services增加时，路由将会刷新。

除了前面描述的过滤器之外，还安装了以下过滤器（与普通的Spring Bean一样）：

预过滤器：

PreDecorationFilter: 根据提供的RouteLocator确定路由的方式和路由的目的地。 它还为下游请求设置各种与代理相关的标头。

路由过滤器：

RibbonRoutingFilter: 使用Ribbon，Hystrix和可插拔HTTP客户端发送请求。 服务ID位于RequestContext属性FilterConstants.SERVICE\_ID\_KEY中。 此过滤器可以使用不同的HTTP客户端：

Apache HttpClient: 默认客户端。

Squareup OkHttpClient v3: 通过在类路径上添加com.squareup.okhttp3:okhttp库并设置ribbon.okhttp.enabled = true来启用。

Netflix Ribbon HTTP client: 通过设置ribbon.restclient.enabled = true启用。 此客户端具有限制，包括它不支持PATCH方法，但它也具有内置重试。

SimpleHostRoutingFilter: 这个filter通过Apache Client给预先定义的URLs发送请求，可以通过RequestContext.getRouteHost()方法获取URLs。。

### 19.15.6 Custom Zuul Filter examples

自定义zuul filter参考<https://github.com/spring-cloud-samples/sample-zuul-filters>，其中一些有操作 “请求/响应” 体的例子。

### 19.15.7 How to Write a Pre Filter

暂不翻译

如果你想要如路由到一个绝对地址，你可以调用 ctx.setRouteHost(url)方法。

如果要修改路由过滤器将转发的路径，请设置REQUEST\_URI\_KEY

### 19.15.8 How to Write a Route Filter

暂不翻译

### 19.15.9 How to Write a Post Filter

暂不翻译

### 19.15.10 How Zuul Errors Work

如果在Zuul过滤器生命周期的任何部分抛出异常，则会执行错误过滤器。SendErrorFilter只有RequestContext.getThrowable()不是null才会运行。然后在请求中设置特定的javax.servlet.error.\*属性，并将请求转发到Spring Boot错误页面。

### 19.15.11 Zuul Eager Application Context Loading（饿汉加载）

Zuul 内部使用Ribbon执行远程调用，Spring Cloud第一次调用的时候Ribbon clients默认执行的是懒加载机制，这一默认的行为可以通过下列的配置改变，改变后可以导致Ribbon在应用启动的时候就执行饿加载。

**application.yml.**

zuul:

ribbon:

eager-load:

enabled: true

## 20. Polyglot support with Sidecar

暂不翻译

可以使用Sidecar整合非JVM微服务，比如C++、Python、PHP等语言写的。其他非JVM微服务可操作Eureka的REST端点，从而实现注册与发现。事实上，也可以使用sidecar更加方便整合非JVM微服务

## 21. RxJava with Spring MVC

Spring Cloud Netflix引入[RxJava](https://github.com/ReactiveX/RxJava)。一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库。

Spring Cloud Netflix提供并支持从Spring MVC Controllers返回rx.Single对象. 它还支持返回 rx.Observable 对象，这个对象支持 Server-sent events (SSE)

以下是使用rx.Single的一些示例：

*@RequestMapping(method = RequestMethod.GET, value = "/single")*

**public** Single<String> single() {

**return** Single.just("single value");

}

*@RequestMapping(method = RequestMethod.GET, value = "/singleWithResponse")*

**public** ResponseEntity<Single<String>> singleWithResponse() {

**return** **new** ResponseEntity<>(Single.just("single value"),

HttpStatus.NOT\_FOUND);

}

*@RequestMapping(method = RequestMethod.GET, value = "/singleCreatedWithResponse")*

**public** Single<ResponseEntity<String>> singleOuterWithResponse() {

**return** Single.just(**new** ResponseEntity<>("single value", HttpStatus.CREATED));

}

*@RequestMapping(method = RequestMethod.GET, value = "/throw")*

**public** Single<Object> error() {

**return** Single.error(**new** RuntimeException("Unexpected"));

}

如果你使用 Observable, 而不Single, 你可以使用.toSingle() 或 .toList().toSingle(). 下面是些例子:

*@RequestMapping(method = RequestMethod.GET, value = "/single")*

**public** Single<String> single() {

**return** Observable.just("single value").toSingle();

}

*@RequestMapping(method = RequestMethod.GET, value = "/multiple")*

**public** Single<List<String>> multiple() {

**return** Observable.just("multiple", "values").toList().toSingle();

}

*@RequestMapping(method = RequestMethod.GET, value = "/responseWithObservable")*

**public** ResponseEntity<Single<String>> responseWithObservable() {

Observable<String> observable = Observable.just("single value");

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(APPLICATION\_JSON\_UTF8);

**return** **new** ResponseEntity<>(observable.toSingle(), headers, HttpStatus.CREATED);

}

*@RequestMapping(method = RequestMethod.GET, value = "/timeout")*

**public** Observable<String> timeout() {

**return** Observable.timer(1, TimeUnit.MINUTES).map(**new** Func1<Long, String>() {

*@Override*

**public** String call(Long aLong) {

**return** "single value";

}

});

}

如果你有一个流端点和客户端,可以试用sse。使用 RxResponse.sse()将rx.Observable转换到Spring 的SseEmitter. 以下是一些例子:

*@RequestMapping(method = RequestMethod.GET, value = "/sse")*

**public** SseEmitter single() {

**return** RxResponse.sse(Observable.just("single value"));

}

*@RequestMapping(method = RequestMethod.GET, value = "/messages")*

**public** SseEmitter messages() {

**return** RxResponse.sse(Observable.just("message 1", "message 2", "message 3"));

}

*@RequestMapping(method = RequestMethod.GET, value = "/events")*

**public** SseEmitter event() {

**return** RxResponse.sse(APPLICATION\_JSON\_UTF8,

Observable.just(**new** EventDto("Spring io", getDate(2016, 5, 19)),

**new** EventDto("SpringOnePlatform", getDate(2016, 8, 1))));

}

## 22. Metrics: Spectator, Servo, and Atlas