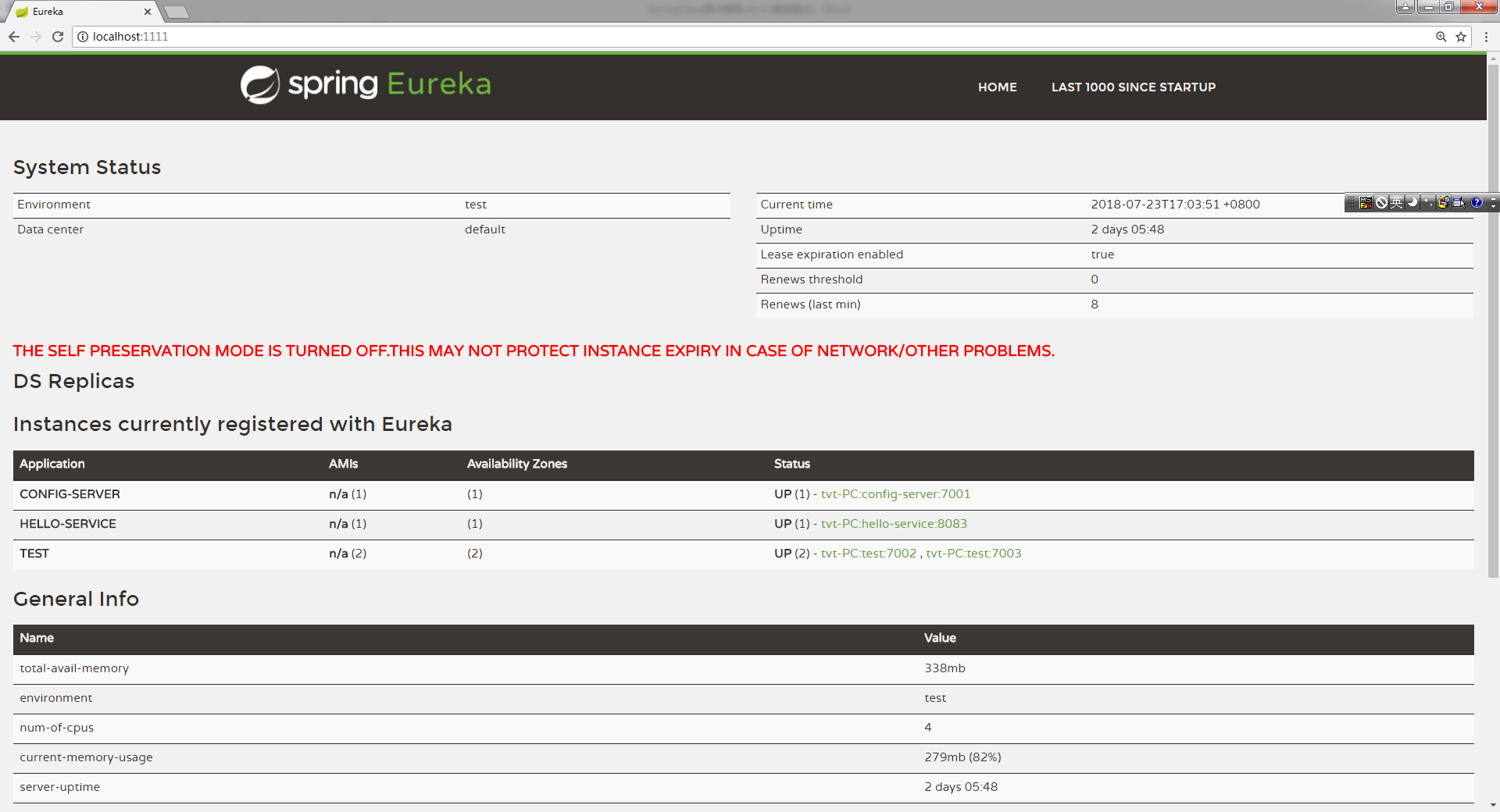


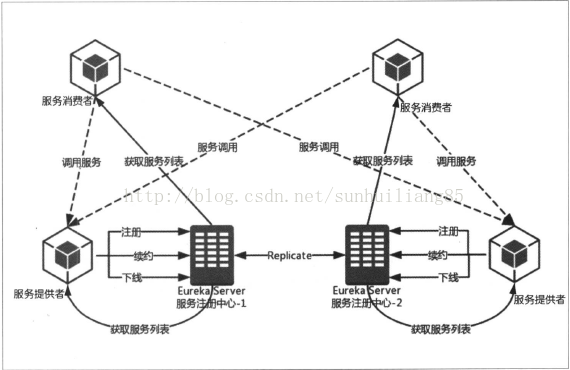
# Spring Cloud Eureka

## 概述

Spring Cloud Eureka是Spring Cloud Netflix微服务套件中的一部分，它基于Netflix Eureka做了二次封装。主要负责完成微服务架构中的服务治理功能。服务治理可以说是微服务架构中的核心和基础的模块，它主要用来实现各个微服务的自动化注册与发现，如下图：



## 服务治理



### 服务提供者

* **服务注册**

服务提供者在启动的时候会通过REST请求的方式将自己注册到Eureka Server上，同时带上自身服务的一些元数据信息。Eureka Server接收到这个Rest请求之后，将元数据信息存储在一个双层结构的Map中，其中第一层的key是服务名。第二层的key 是具体服务的实例名。

在服务注册时，需要确认一下eureka.client.register-with-eureka=true参数是否正确，该值默认为true。若设置为fasle将不会启动注册操作。

* **服务续约**

在注册服务之后，服务提供者会维护一个心跳用来持续高速Eureka Server，“我还在持续提供服务”，否则Eureka Server的剔除任务会将该服务实例从服务列表中排除出去。我们称之为服务续约。

下面是服务续约的两个重要属性：

（1）eureka.instance.lease-renewal-interval-in-seconds

leaseRenewalIntervalInSeconds，表示eureka client发送心跳给server端的频率默认30秒

（2）eureka.instance.lease-expiration-duration-in-seconds

服务失效时间，默认为90秒

* **服务下线**

当服务实例进行正常的关闭操作时，它会触发一个服务下线的Rest请求给Eureka Server.注册中心接收到请求后，将该服务状态置为下线(DOWN)。

### 服务消费者

* **获取服务**

消费者服务启动时，会发送一个Rest请求给服务注册中心，来获取上面注册的服务清单。为了性能考虑，Eureka Server会维护一份只读的服务注册清单来返回给客户端，同时该缓存清单默认会每隔30秒更新一次。

下面是获取服务的两个重要的属性：

（1）eureka.client.fetch-registry

是否需要去检索寻找服务，默认是true

（2）eureka.client.registry-fetch-interval-seconds

表示eureka client间隔多久去拉取服务注册信息，默认为30秒，对于api-gateway，如果要迅速获取服务注册状态，可以缩小该值，比如5秒

* **服务调用**

服务消费者在获取服务清单后，通过服务名可以获取具体提供服务的实例名和该实例的元数据信息。因为有这些服务实例的详细信息，所以客户端可以根据自己的需要决定具体调用哪个实例，在Ribbon中会默认采用轮询的方式进行调用，从而实现客户端的负载均衡。

### 服务注册中心

* **服务同步**

从eureka服务治理体系架构图中可以看到，不同的服务提供者可以注册在不同的服务注册中心上，它们的信息被不同的服务注册中心维护。

此时，由于多个服务注册中心互相注册为服务，当服务提供者发送注册请求到一个服务注册中心时，它会将该请求转发给集群中相连的其他注册中心，从而实现服务注册中心之间的服务同步。通过服务同步，提供者的服务信息就可以通过集群中的任意一个服务注册中心获得。

* **失效剔除**

有些时候，我们的服务实例并不一定会正常下线，可能由于内存溢出、网络故障等原因使服务不能正常运作。而服务注册中心并未收到“服务下线”的请求，为了从服务列表中将这些无法提供服务的实例剔除，Eureka Server在启动的时候会创建一个定时任务，默认每隔一段时间（默认为60秒）将当前清单中超时（默认为90秒）没有续约的服务剔除出去。

* **自我保护**

服务注册到Eureka Server后，会维护一个心跳连接，告诉Eureka Server自己还活着。Eureka Server在运行期间会统计心跳失败的比例在15分钟以之内是否低于85%，如果出现低于的情况，Eureka Server会将当前实例注册信息保护起来，让这些实例不会过期。这样做会使客户端很容易拿到实际已经不存在的服务实例，会出现调用失败的情况。因此客户端要有容错机制，比如请求重试、断路器。

以下是自我保护相关的属性：

eureka.server.enableSelfPreservation=true. 可以设置改参数值为false，以确保注册中心将不可用的实例删除

## 引用的依赖包及版本

* **服务注册中心**

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-server</**artifactId**>  
</**dependency**>

* **服务提供者**

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
</**dependency**>

## 代码样例

* **注册中心**

主类添加@EnableEurekaServer注解

@EnableEurekaServer

@SpringBootApplication

public class EurakaserverApplication {

public static void main(String[] args) {

SpringApplication.run(EurakaserverApplication.class, args);

}

}

配置文件application.properties

server.port=1111

eureka.instance.hostname=localhost

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/

* **服务提供方**

主类添加@EnableEurekaClient注解

@EnableEurekaClient

@SpringBootApplication

public class EurakaclientApplication {

public static void main(String[] args) {

SpringApplication.run(EurakaclientApplication.class, args);

}

}

控制器注入DiscoveryClient对象

@RestController

public class HelloController {

@Autowired

private DiscoveryClient client;

@RequestMapping("/hello")

public String index(HttpServletRequest httpServletRequest)

{

System.out.println(httpServletRequest.getRequestURI());

return "Hello World";

}

}

配置文件

**server.port**=**8083**

**spring.application.name**=**hello-service  
eureka.client.serviceUrl.defaultZone**=**http://localhost:1111/eureka/**

## 问题

### 关于服务健康检查的问题：

**问：**经Eureka Server服务注册中心仅通过心跳超时来剔除服务实例节点，如果开启了客户端的服务健康检查 eureka.client.healthcheck.enabled=true，实现自定义健康检查器HealthIndicator，并设置状态为DOWN，此时Eureka Server的Web界面上显示此服务实例的状态为Down，但不会剔除，此时调用此服务时，是否可调用到此服务实例？

**答：**通过Eureka去调用时提示找不到实例，无论是通过Ribbon还是Feign方式，而直接访问服务URL地址可以调用到实例。

### getLocalServiceInstance函数已经覆盖问题

DiscoveryClient类的getLocalServiceInstance函数已经覆盖，可以通过getInstances函数获取服务的对应实例例表信息。

# Spring Cloud Ribbon

## 概述

Spring Cloud Ribbon是一个基于http和tcp的客户端负载均工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以轻松将面向服务的REST模板请求自动转换成客户端负载均衡的服务调用。微服务间的调用，API网关的请求转发内容，实际上都是通过Ribbon来实现。

客户端负载均衡和服务端负载均衡最大的不同点在于服务清单所存储的位置。在客户端负载均衡中，所有客户端节点都维护着自己要访问的服务端清单，而这些清单来自于服务注册中心。

## 负载均衡器

通过继承AbstractLoadBalancerRule抽象类来具体实现负载均衡策略。

* **RandomRule**

通过随机服务实例的数量来产生一个随机数，通过索引获取该服务实例

* **RoundRobinRule**

按照线性轮询的方式依次选择每个服务实例的功能

* **RetryRule**

该策略实现了一个具备重试机制的实例选择功能。其内部还定义了一个IRule对象，默认使用RoundRobinRule实例。在choose方法中则实现了对内部定义的策略进行反复尝试的策略，若期间能够选择到具体的服务实例就返回，若选择不到就根据设置的尝试结束时间为阀值（maxRetryMillis参数定义的值+choose方法开始执行时的时间戳），当超过该值后就返回null

* **WeightedResponseTimeRule**

该策略是对RoundRobinRule的扩展，增加了根据实例的运行情况来计数权重，并根据权重来挑选实例。该策略实例化的时候在内部创建了一个定时任务，每过30s便去统计一下各个实例的权重。

* **ClientConfigEnableRoundRobinRule**

该策略较为特殊，一般不直接使用它。该策略内部定义了一个RoundRobinRule策略，choose函数的实现也是使用了RoundRobinRule的线下轮询机制。一般使用方法：继承该策略，默认的choose方法实现了线性轮询机制，在子类中做一些高级策略时通常可能会存在一些无法实施的情况，那么就可以用父类的实现作为备选。

* **BestAvailableRule**

该策略继承ClientConfigEnableRoundRobinRule，在实现中它注入了负载均衡器的统计对象LoadBalancerStats，同时在具体的choose算法中利用LoadBalancerStats保存的实例统计信息来满足要求的实例。它通过遍历负载均衡器中维护的所有服务实例，会过滤掉故障的实例，并找出并发请求数最少的一个，所以该策略的特性是可选出最空闲的实例。

* **PredicateBasedRule**

抽象策略，继承了ClientConfigEnableRoundRobinRule，基于Predicate实现的策略，Predicate是Google Guava Collection工具对集合进行过滤的条件接口，策略：先过滤清单，再轮询选择

* **AvailableFilteringRule**

继承自PredicationBasedRule，该策略先以线性方式选择一个实例，接着用过滤条件判断该实例是否满足要求，若满足就直接使用该实例，若不满足选择下一个实例，当过程重10次还没找到合适的实例，就采用父类的方案。

* **ZoneAvoidanceRule**

继承自PredicationBasedRule

使用主过滤条件对所有实例过滤并返回过滤后的实例清单

依次使用次过滤条件列表中的过滤条件对主过滤条件的结果进行过滤

每次过滤之后需要判断下面两个条件，只有一个符合就不再进行过滤，将当前结果返回供线性轮询算法选择：

过滤后的实例总数>=最小过滤实例数，默认为1

过滤后的实例比例〉最小过滤百分比，默认为0

## 重试

由于spring cloud eureka在可用性与一致性上的取舍，所以我们在实现服务调用的时候通常会加入一些重试机制。spring cloud 整合了spring retry来增强RestTemplate的重试能力，只需通过简单的配置，原来那些通过RestTemplate实现的服务访问就会自动根据配置来实现重试策略。

spring.cloud.loadbalancer.retry.enabled=true

#开启重试机制

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000

#断路器的超时时间需要大于Ribbon的超时时间，不然不会触发重试

ribbon.ConnectTimeout=250

#请求连接的超时时间

ribbon.ReadTimeout=1000

#请求处理的超时时间

ribbon.OkToRetryOnAllOperations=true

#对所有操作请求都进行重试

ribbon.MaxAutoRetriesNextServer=2

#切换实例的重试次数

ribbon.MaxAutoRetries=1

#对当前实例的重试次数

## 代码样例

* **引入的包：**

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-ribbon</**artifactId**>  
</**dependency**>

* **主类添加注解**

通过@EnableDiscoveryClient注解让该应用注册为Eureka客户端应用，以获得服务发现能力。

同时，在该主类中创建RestTemplate 的实例,并通过@LoadBalanced注解开启客户端负载均衡

Application代码如下：

@EnableDiscoveryClient  
@SpringBootApplication  
**public class** RibbonConsumerApplication {  
  
 @Bean  
 @LoadBalanced  
 RestTemplate restTemplate()  
 {  
 **return new** RestTemplate();  
 }  
  
 **public static void** main(String[] args)  
 {  
 SpringApplication.*run*(RibbonConsumerApplication.**class**, args);  
 }  
}

* **Controller代码如下：**

@RestController  
**public class** ConsumerController {  
  
 @Autowired  
 RestTemplate **restTemplate**;  
  
 @RequestMapping(**"/ribbonConsumer"**)  
 **public** String ribbonConsumer()  
 {  
 *return restTemplate.getForEntity("http://HELLO-SERVICE/hello",String.class).getBody();* }  
}

* **配置文件application.properties**

**spring.application.name**=**ribbon-consumer  
eureka.client.serviceUrl.defaultZone**=**http://localhost:1111/eureka/**

# Spring Cloud Feign

## 概述

在Ribbon中，通过RestTemplate已经实现了对HTTP请求的封装处理，完成对服务的调用。Feign在此基础上作了进一步封装，我们只需创建一个接口并用注解的方式来配置它，即可完成对服务提供方的接口绑定，简化了在使用Ribbon时自行封装服务调用客户端的开发量。

## 代码样例

* **服务提供方**

@RestController  
public class HelloController {  
  
 @Autowired  
 private DiscoveryClient client;  
  
  
 @RequestMapping("/up")  
 public String up(@RequestParam("up") Boolean up) {  
 myHealthChecker.setUp(up);  
  
 return up.toString();  
 }  
  
 @RequestMapping("/hello")  
 public String hello(HttpServletRequest httpServletRequest) throws Exception  
   
 return "Hello World";  
 }  
  
 @RequestMapping("/hello1")  
 public String hello(@RequestParam String name)  
 {  
 return "Hello "+name;  
 }  
  
 @RequestMapping("/hello2")  
 public User hello(@RequestHeader String name,@RequestHeader Integer age)  
 {  
 return new User(name,age);  
 }  
  
 @RequestMapping("/hello3")  
 public String hello(@RequestBody User user)  
 {  
 return "Hello " + user.getName() +", " + user.getAge();  
 }  
}

* **消费方引入包**

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-openfeign</**artifactId**>  
</**dependency**>

* **消费方主类添加@EnableFeignClients开启Feign支持**

@EnableFeignClients  
@EnableDiscoveryClient  
@SpringBootApplication  
**public class** FeignConsumerApplication  
{  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(FeignConsumerApplication.**class**, args);  
 }  
}

* **定义HelloService接口**

@Service  
@FeignClient(name=**"HELLO-SERVICE"**,fallback = HelloServiceFallback.**class**)  
**public interface** HelloService  
{  
 @RequestMapping(**"/hello"**)  
 String hello();  
  
 @RequestMapping(value=**"/hello1"**,method=RequestMethod.***GET***)  
 String hello(@RequestParam(**"name"**) String name);  
  
 @RequestMapping(value=**"/hello2"**,method=RequestMethod.***GET***)  
 User hello(@RequestHeader(**"name"**) String name,@RequestHeader(**"age"**) Integer age );  
  
 @RequestMapping(value=**"/hello3"**,method=RequestMethod.***POST***)  
 String hello(@RequestBody User user);  
  
}

* 创建Controller来实现对客户端的调用

@RestController

public class ConsumerControl {

@Autowired

HelloService helloService;

@Autowired

RefactorHelloService refactorHelloService;

@RequestMapping(value="/fegin-consumer",method=RequestMethod.GET)

public String helloConsummer()

{

return helloService.hello();

}

@RequestMapping(value="/fegin-consumer2",method=RequestMethod.GET)

public String helloConsummer2()

{

StringBuffer sb=new StringBuffer();

sb.append(helloService.hello()).append("\n");

sb.append(helloService.hello("")).append("\n");

sb.append(helloService.hello("DIDI",30)).append("\n");

sb.append(helloService.hello(new User("DIDI",30))).append("\n");

return sb.toString();

}

}

* 配置文件

**spring.application.name**=**feign-consumer  
eureka.client.serviceUrl.defaultZone**=**http://localhost:1111/eureka/**

## Ribbon配置

全局配置

**ribbon.ConnectTimeout**=**500  
ribbon.ReadTimeout**=**5000**

指定服务配置

**HELLO-SERVICE.ribbon.ConnectTimeout**=**500  
HELLO-SERVICE.ribbon.ReadTimeout**=**2000  
HELLO-SERVICE.ribbon.OkToRetryOnAllOperations**=**true  
HELLO-SERVICE.ribbon.MaxAutoRetiresNextServer**=**2  
HELLO-SERVICE.ribbon.MaxAutoRetries**=**1**

## Hystrix配置

全局配置

通过前缀hystrix.command.default进行设置，如全局超时

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000

通过feign.hystrix.enabled=false可关闭hystrix功能

指定命令配置

采用hystrix.command.<commandKey>作前缀

hystrix.command.hello.execution.isolation.thread.timeoutInMilliseconds=5000

# Spring Cloud Zuul

## 概述

API网关是一个更为智能的应用服务器，他的定义类似于面向对象设计模式中的Façade模式，它的存在就像是整个微服务架构系统的门面，所有的外部客户端访问都需要经过他来进行调度和过滤。它除了要实现请求路由、负载均衡、校验过滤等功能外，还需要更多的能力，比如请求转发时的熔断机制等。引入API具体原因如下：

* 作为系统统一入口，屏蔽了系统内部各个微服务的细节
* 与服务治理框架结合，实现自动化的实例维护及负载均衡和路由转发
* 实现接口权限校验与服务业务逻辑的解耦
* 通过过滤器，在各生命周期中去校验请求的内容

## 代码样例

* 构建网关

引入依赖包

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
</**dependency**>

创建应用主类

使用@EnableZuulProxy注解开启Zuul的API网关服务功能

@EnableZuulProxy  
@SpringBootApplication  
**public class** ApigatewayApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ApigatewayApplication.**class**, args);  
 }}

配置文件

**server.port**=**5555**

**pring.application.name**=**api-gateway  
eureka.client.serviceUrl.defaultZone**=**http://localhost:1111/eureka/**

* 请求路由

zuul.routes.user-service.path=/user-service/\*\*

zuul.routes.user-service.url=http://localhost:8080

* 请求过滤

建立过滤器

**public class** AccessFilter **extends** ZuulFilter {  
  
 **private static** Logger *log*=LoggerFactory.*getLogger*(AccessFilter.**class**);  
  
 @Override  
 **public** String filterType()  
 {  
 **return "pre"**;  
 }  
  
 @Override  
 **public int** filterOrder()  
 {  
 **return** 0;  
 }  
  
 @Override  
 **public boolean** shouldFilter()  
 {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run()  
 {  
 RequestContext ctx=RequestContext.*getCurrentContext*();  
 HttpServletRequest request=ctx.getRequest();  
  
 *log*.info(**"send{} request to {}"** ,request.getMethod());  
 Object accessToken=request.getParameter(**"accessToken"**);  
 **if**(accessToken==**null**)  
 {  
 *log*.warn(**"access token is empty"**);  
 ctx.setSendZuulResponse(**false**);  
 ctx.setResponseStatusCode(401);  
 **return null**;  
 }  
  
 *log*.info(**"access token ok"**);  
 **return null**;  
 }  
}

启动过滤器

@EnableZuulProxy  
@SpringBootApplication  
**public class** ApigatewayApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ApigatewayApplication.**class**, args);  
 }  
  
   
 @Bean  
 **public** AccessFilter accessFilter()  
 {  
 **return new** AccessFilter();  
 }  
   
}

访问<http://localhost:5555/user-service/hello返回401>错误

<http://localhost:5555/user-service/hello&accessToken=token>返回helloWorld

## 路由详解

传统路由配置

zuul.routes.user-service.path=/user-service/\*\*

zuul.routes.user-service.url=http://localhost:8080

服务路由配置

zuul.routes.user-service.path=/user-service/\*\*

zuul.routes.user-service.serviceId=user-service

服务路由的默认规则

zuul.routes.user-service.path=/user-service/\*\*

zuul.routes.user-service.serviceId=user-service

当zuul构建的API网关服务引入Eureka后，它为Eureka中的每个服务都自动创建一个默认路由规则，这些默认规则的path会使用serviceId配置的服务名作为请求前缀。

## Hystrix和Robbin支持

Zuul自带了线程隔离和断路器的自我保护功能，以及对服务调用客户端负载均衡功能。但是需要注意，当使用path和url的映射关系来配置路由规则时，由于路由转发的请求不会采用HystrixCommand来包装，所以这类路由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。因此尽量使path和serviceId的组合来进行配置，这样不仅能保证API网关的健壮和稳定，也能用到Ribbon的客户端负载均衡功能。

## 过滤器

过滤器负责对请求的处理过程进行干预，是实现请求路由、请求校验等功能的基础。

过滤器包含4个基本特征：过滤类型、执行顺序、执行条件、具体操作

实际上它就是ZuulFilter接口中定义的4个抽象方法

String filterType()

Int filterOrder()

Boolean shouldFilter();

Object run();

它们各自的含义和与功能总结如下

1. filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：

pre：可以在请求被路由之前调用

route：在路由请求时候被调用

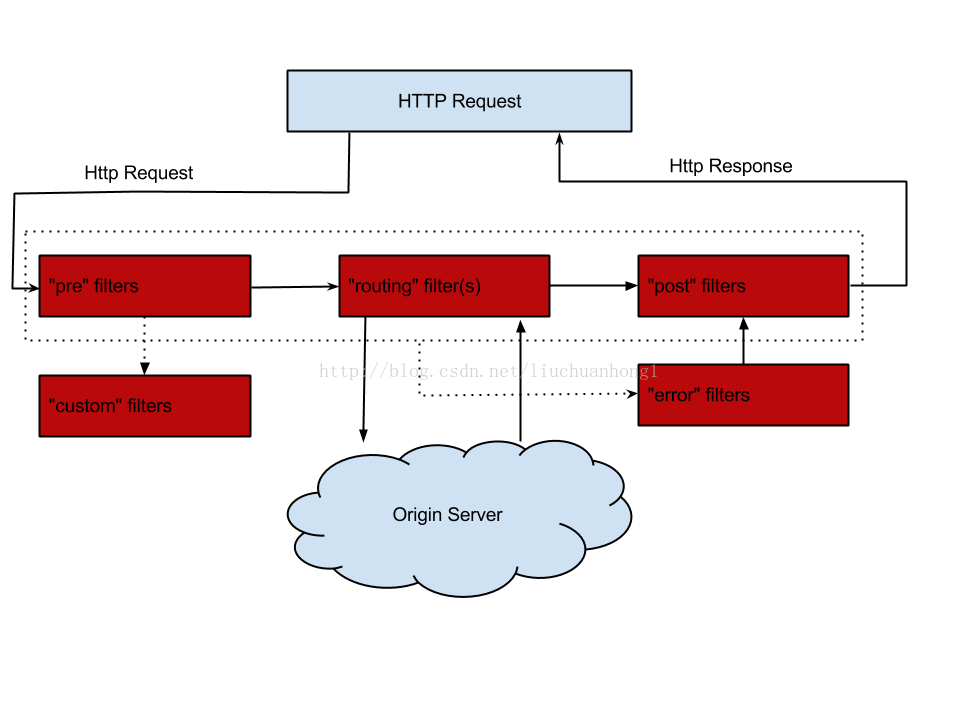
post：在route和error过滤器之后被调用

error：处理请求时发生错误时被调用

Zuul的主要请求生命周期包括“pre”，“route”和“post”等阶段。对于每个请求，都会运行具有这些类型的所有过滤器。

1. filterOrder：通过int值来定义过滤器的执行顺序
2. shouldFilter：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效
3. run：过滤器的具体逻辑。需要注意，这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回

的错误码



## 动态加载

在微服务架构中，API网关服务负担着外部访问统一入口的重任，它同其它应用不同，任何关闭应用和重启应用的操作都会是系统对外服务停止，对于很多7\*24小时服务系统，这样的情况是绝对不允许的。所以，它必须具备动态更新内部逻辑的能力。

通过Zuul实现的API网关当然也具备了动态路由和动态过滤得能力。可以在不重启API网关的前提下，为其动态修改路由规则和添加或删除过滤器。

# Spring Cloud Config

## 概述

Spring Cloud Config用来为分布式系统中的基础设施和为服务应用提供集中化的外部支持，他分为客户端和服务端两个部分。其中服务端也称为分布式配置中心，它是一个独立的为服务应用，用来连接配置仓库并为客户应用提供获取配置信息，加密/解密信息等访问接口；客户端则是各个微服务应用，它们通过制定的配置中心来管理应用资源与业务相关的配置内容，并在启动时从配置中心获取和加载配置信息。

Spring Cloud Config配置中心默认采用Git来存储配置信息，同时也提供对其它存储方式的支持，如SVN仓库、本地化文件系统。

## 基础架构



* 远程GIT仓库：用来存储配置文件的地方，钟对应用的多环境配置文件appliation-{profile}.properties
* Config Server：分布式配置中心，在该工程中指定了所要连接的Git仓库位置及帐户、密码等连接信息
* 本地Git仓库：在ConfigServer的文件系统中，每次客户端请求获取配置信息时，ConfigServer从Git仓库中获取最新配置到本地，然后再本地Git仓库中读取并返回。当远程仓库无法获取时，直接将本地内容返回。
* ServerA、ServerB:具体的微服务应用，它们指定了Config Server的地址，从而实现从外部化获取应用自己要用的配置信息。这些引用在启动时，会向Config Server请求获取配置信息来进行加载。

客户端应用从配置管理中获取配置信息遵从下面的执行流程：

1. 应用启动时，根据配置文件中配置的应用名{application}、环境名{profile}、分支名{label}向Config Server请求获取配置信息。
2. Config Server根据自己维护的Git仓库信息和客户端传递过来的配置定位信息去查找配置信息
3. 通过git clone命令找到的配置信息下载到Config Server的文件系统中
4. Config Server创建Spring的AppplicationContext实例，并从Git本地仓库中加在配置文件，最后将这些配置内容读取出来返回给客户端应用。
5. 客户端应用在获得外部配置文件后加载到客户端的ApplicationContext实例，该配置内容的优先级高于客户端Jar包内部的配置内容，所以在Jar包中重复的内容将不再被加载。

## 代码样例

* 构建配置中心

创建一个基础的Spring Boot 工程，在pom.xml中引入以下依赖：

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-config-server</**artifactId**>  
</**dependency**>

<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>${spring-cloud.version}</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

创建Spring Boot的程序主类，并添加@EnableConfigServer注解，开启Spring Cloud Config的服务端功能

@EnableConfigServer  
@SpringBootApplication  
**public class** ConfigserverApplication {  
  
 **public static void** main(String[] args)  
 {  
 SpringApplication.*run*(ConfigserverApplication.**class**, args);  
 }  
}

在application.properties配置文件中添加服务基本信息及Git仓库相关信息

**spring.application.name**=**config-server  
  
server.port**=**7001  
eureka.client.serviceUrl.defaultZone**=**http://localhost:1111/eureka/   
  
spring.cloud.config.server.git.uri**=**https://github.com/yanjj98/default-config  
spring.cloud.config.label**=**master  
spring.cloud.config.server.git.search-paths**=**/\*\*  
spring.cloud.config.server.git.username**=**user  
spring.cloud.config.server.git.password**=**password**

在Git仓库中新建4个配置文件

test.properties

test-dev.properties

test-test.properties

test-prod.properties

在这4个配置文件中均设置一个from属性，并为每个配置文件分别设置不同的值，如下所示：

from=git-default-1.0

from=git-dev-1.0

from=git-test-1.0

from=git-prod-1.0

配置规则详解

访问配置信息的URL与具体配置文件的映射关系如下所示：

/{application}/{profile}[{label}]/

/{application}-{profile}.yml

/{label}/{application}-{plrofile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

通过访问<http://localhost:7001/test/prod/master>可以访问对应配置文件中的内容

* 客户端配置映射

创建Spring Boot应用，命名为config-client,在pom.xml中引入下述依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
</**dependency**>

<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>${spring-cloud.version}</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

创建Spring Boot的应用主类

@EnableDiscoveryClient  
@SpringBootApplication  
**public class** ConfigclientApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ConfigclientApplication.**class**, args);  
 }  
}

创建bootstrap.proterties配置文件

**spring.application.name**=**test  
spring.cloud.config.profile**=**prod  
spring.cloud.config.label**=**master  
spring.cloud.config.discovery.enabled**=**true  
spring.cloud.config.discovery.service-id**=**CONFIG-SERVER  
  
server.port**=**7002**

spring.application.name:对应配置文件规则中的{application}部分

spring.cloud.config.profile:对应配置文件规则中的{profile}部分

spring.cloud.config.label:对应配置文件规则中的{label}部分

spring.cloud.config.uri:配置中心config-server的地址

创建一个RESTful借口来返回配置中心的from属性，通过@value(“${from}”)绑定配置服务中配置的from属性

@RefreshScope  
@RestController  
**public class** TestController  
{  
 @Value(**"${from}"**)  
 **private** String **from**;  
  
 @RequestMapping(**"/from"**)  
 **public** String from()  
 {  
 **return this**.**from**;  
 }  
}

启动应用访问<http://localhost:7002/from可以看到返回如下内容git-dev-1.0>

## 高可用配置中心

* 传统模式：不需要为服务端做任何额外的配置，只需要遵守一个配置规则，将所有的configServer都指向一个Git仓库，这样所有的配置内容就通过统一的共享文件系统来维护。而客户端在指定config Server时，只需要指定负载均衡设备地址即可



* 服务模式：将config Server作为一个普通的为服务应用纳入eureka治理体系中。这样位服务应用就可以通过配置中心的服务名来获取配置信息，这种方式比传统的实现模式更加有利于维护，因为对于服务端的负载均衡配置和客户端的配置中心指定都通过服务治理机制一并解决了，既实现了高可用，也实现了自维护。

## 动态刷新配置

将config-server和config-client都启动起来，并访问客户端提供的REST接口<http://localhost:7002/from>来获取配置信息，获得返回内容为git-dev-1.0。接着我们尝试用git工具修改当前配置内容，比如将from=git-dev-1.0修改为from=git-dev-2.0,再访问<http://localhost:7002/from,可以看到其返回内容还是git-dev-1.0>

接下来可以做一个改造实现配置信息的动态刷新

在config-client的pom.xml中新增actuator监控模块，。

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>

actuator模块包含了/actuator/refresh端点的实现,该端点将用于实现客户端应用配置信息的重新获取与刷新，但是Spring boot新版本端点信息默认是不开放的，需要在配置文件中进行配置，开放端点信息,在bootstrap.properties文件中添加如下配置：

**management.endpoints.web.exposure.include**=**\***

重新启动config-client,访问一次<http://localhost:7002/from>,可以看到当前的配置值。

修改git仓库中from的值

再访问<http://localhost:7002/from>,可以看到配置值没有改变

通过post发送请求到<http://localhost:7002/actuator/refresh>,可以看到返回内容如下，代表from参数的配置内容被更新了：

[

“from

]

再访问<http://localhost:7002/from>，可以看到配置值已经更新了。