**私有云微服务开发**

**使用手册**

**目录**

[1、服务治理Spring Cloud Eureka 3](#_Toc499133396)

[1.1、服务注册中心 3](#_Toc499133397)

[1.2、服务注册、发现与消费 8](#_Toc499133398)

[2、客户端负载均衡spring cloud ribbon 12](#_Toc499133399)

[2.1 spring cloud ribbon调用方式 12](#_Toc499133400)

[2.2 Ribbon自动化配置 17](#_Toc499133401)

[3 服务容错保护 Spring Cloud Hystrix 19](#_Toc499133402)

[3.1 Hystrix介绍 19](#_Toc499133403)

[3.2 Hystrix配置 20](#_Toc499133404)

[3.2.1 HystrixCommand命令属性 21](#_Toc499133405)

[3.2.1.1 Exceution配置 21](#_Toc499133406)

[3.2.1.2 Fallback配置 25](#_Toc499133407)

[3.2.1.3 Circuit配置 26](#_Toc499133408)

[3.2.1.4 Metric配置 30](#_Toc499133409)

[3.2.1.5 Request配置 35](#_Toc499133410)

[3.2.2 HystrixCollaper 属性 36](#_Toc499133411)

[3.2.3线程池threadPool属性 38](#_Toc499133412)

[3.3 Hystrix监控 42](#_Toc499133413)

[3.3.1 构建简单的监控服务 42](#_Toc499133414)

[3.3.2 Turbine 集群监控 44](#_Toc499133415)

[3.3.3 Turbine 与消息代理的结合 46](#_Toc499133416)

[4 Spring Cloud Feign声明式服务调用 46](#_Toc499133417)

[4.1 简单实例开发 46](#_Toc499133418)

[4.2 Feign继承特性 48](#_Toc499133419)

[4.3 Feign配置 48](#_Toc499133420)

[5 Spring cloud Zuul 服务API网关 56](#_Toc499133421)

[5.1 构建Simple zuul 56](#_Toc499133422)

[5.2 路由配置 58](#_Toc499133423)

[5.2.1 传统路由配置 58](#_Toc499133424)

[5.2.2 面向服务的路由配置 59](#_Toc499133425)

[5.3 过滤器配 62](#_Toc499133426)

[5.3.1 禁用Zuul过滤器 63](#_Toc499133427)

[5.3.2 核心过滤器 63](#_Toc499133428)

[5.4 异常处理 66](#_Toc499133429)

[5.5 动态加载 67](#_Toc499133430)

[6分布式配置中心Spring Cloud Config 68](#_Toc499133431)

[6.1 构建配置中心 68](#_Toc499133432)

[6.2 配置中心服务端 71](#_Toc499133433)

[6.3 配置中心客户端 75](#_Toc499133434)

[6.3.1 URI 指定配置中心 75](#_Toc499133435)

[6.3.2 服务化配置中心 76](#_Toc499133436)

[6.3.2 动态刷新配置 76](#_Toc499133437)

[6.3.3 失败快速响应与重试 77](#_Toc499133438)

[7 消息总线 Spring Cloud Bus 77](#_Toc499133439)

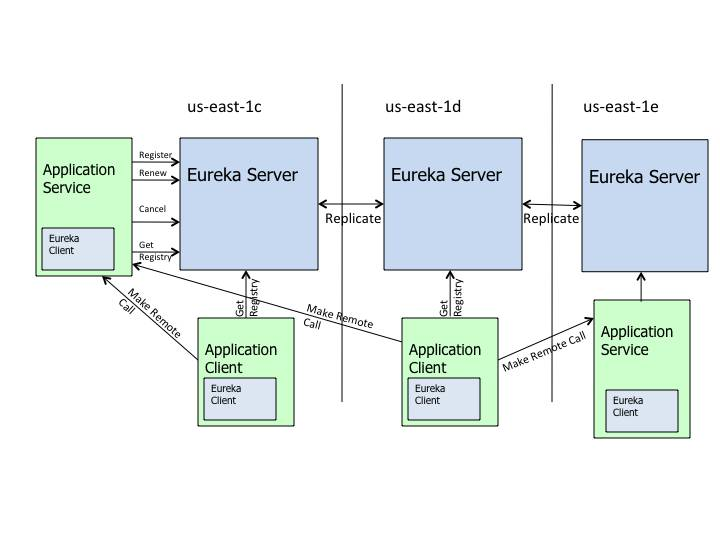
[8参数实例 80](#_Toc499133440)

环境Java:8，spring-boot-starter-parent:1.5.7、spring cloud-version:Dalston.SR4

## 1、服务治理Spring Cloud Eureka

本章主要是服务治理的核心组件：构建注册中心，服务注册、发现、消费

Eureka架构图：



### 1.1、服务注册中心

本小节主要在说明如何构建服务注册中心

1)首先，介绍spring cloud Eureka构建注册中心的常用注解

|  |  |  |
| --- | --- | --- |
| 序号 | 注解API | 释义 |
| 1 | EnableEurekaServer | 服务诠释为EurekaServer特性的注册中心(默认) |

2)构建注册中心主要pom依赖项:

<dependency>

<!--增加eureka-server的依赖-->  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka-server</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
</dependency>

3)主要配置参数说明及Eureka server集群实例

配置文件 applicaiton-{profile}.yml 或者使用application-{profile}.properties

Eureka-server1配置：

spring:

profiles:

active: dev

application:

name: eureka-server

management:

security:

enabled: false

eureka:

client:

# register-with-eureka: false (standone server)

# fetch-registry: false

service-url:

defaultZone: http://eureka-server2:8083/eureka/

instance:

hostname: eureka-server1

server:

debug: true

# enable-self-preservation: false

# prefer-ip-address: true

Eureka-server2配置：

server:

port: 8083

eureka:

client:

serviceUrl:

defaultZone: http://eureka-server1:8082/eureka/

instance:

hostname: eureka-server2

server:

# enable-self-preservation: false

# prefer-ip-address: true

spring:

application:

name: eureka-server

management:

security:

enabled: false

debug: true

关于动态配置更新，后续会说的。

对于更加详尽的参数含义参考类EurekaCLientConfigBean参数均以eureka.client为前缀和EurekaInstanceConfigBean参数均以eureka.instance为前缀

|  |  |  |
| --- | --- | --- |
| EurekaCLientConfigBean | | |
| 参数名 | 说明 | 默认值 |
| enabled | 启用eureka client | true |
| registryFetchIntervalSeconds | 从eureka服务端获取注册信息间隔秒 | 30 |
| instanceInfoReplicationIntervalSeconds | 更新副本实例信息到eureka服务端的间隔时间秒 | 30 |
| initialInstanceInfoReplicationIntervalSeconds | 初始化副本实例信息到eureka服务端的间隔时间单位秒 | 40 |
| eurekaServiceUrlPollIntervalSeconds | 更新eureka服务端地址变更时间间隔单位秒 | 300 |
| eurekaServerReadTimeoutSeconds | 读取eureka服务端超时时间 | 8 |
| eurekaServerConnectTimeoutSeconds | 连接eureka服务端超时时间 | 5 |
| eurekaServerTotalConnections | Eureka客户端连接到所有服务端总的连接数 | 200 |
| eurekaServerTotalConnectionsPerHost | Eureka客户端连接到服务端总的连接数 | 50 |
| eurekaConnectionIdleTimeoutSeconds | Eureka服务端空闲连接关闭时间 | 30 |
| region | 一个region可以有多个zone |  |
| zone |  |
| heartbeatExecutorThreadPoolSize | 心跳连接池初始化大小 | 2 |
| useDnsForFetchingServiceUrls |  |  |
| registerWithEureka |  |  |
| preferSameZoneEureka |  |  |
| filterOnlyUpInstances |  |  |

|  |  |  |
| --- | --- | --- |
| EurekaInstanceConfigBean | | |
| 参数名 | 说明 | 默认值 |
| preferIpAddress | 是否优先使用ip作为标识 | false |
| leaseRenewalIntervalInSeconds | 续约间隔单位秒 | 30 |
| leaseExpirationDurationInSeconds | 服务失效剔除时间 | 90 |

启动类

@EnableEurekaServer // 启用eureka server特性

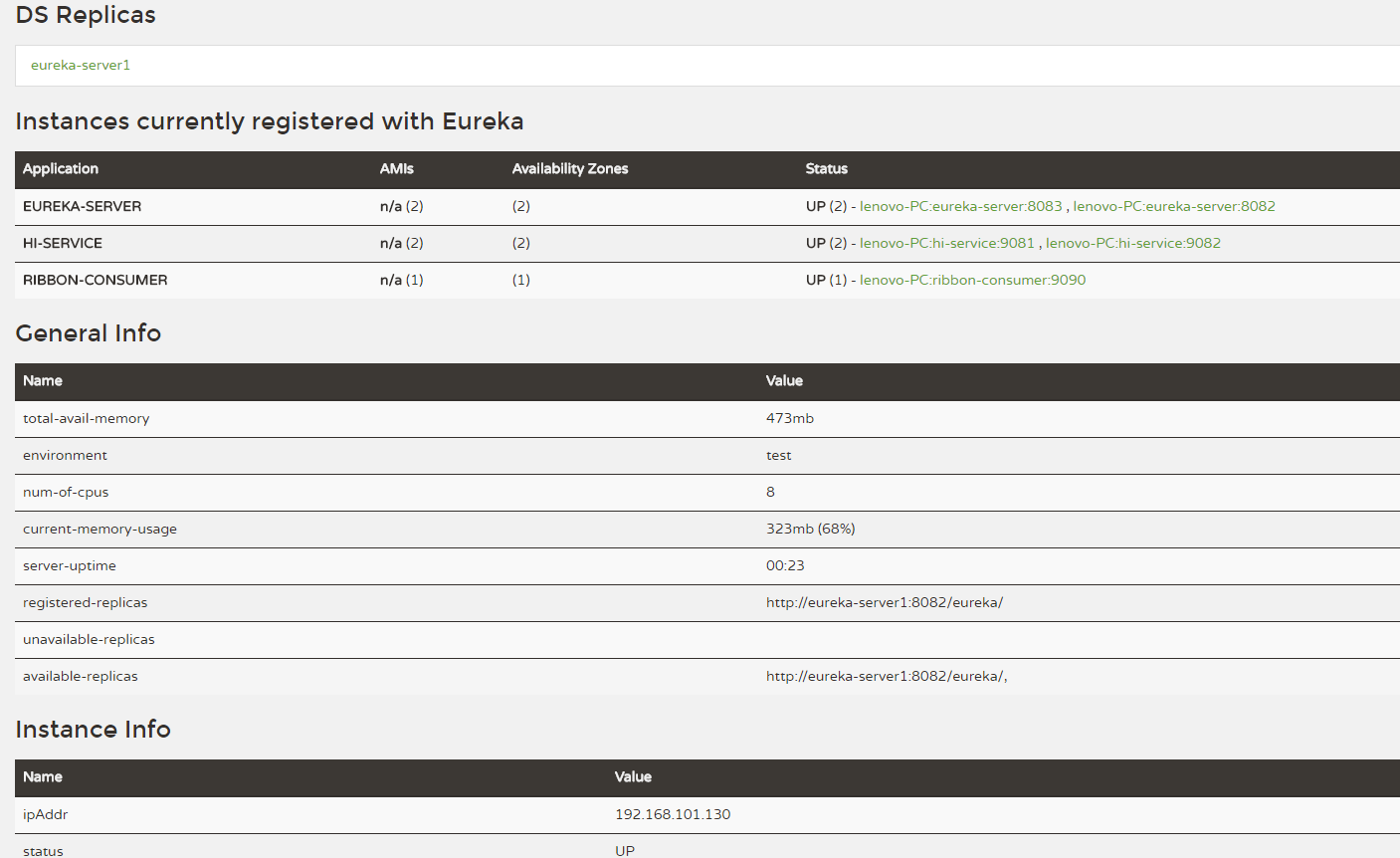
@SpringBootApplication // spring boot特性

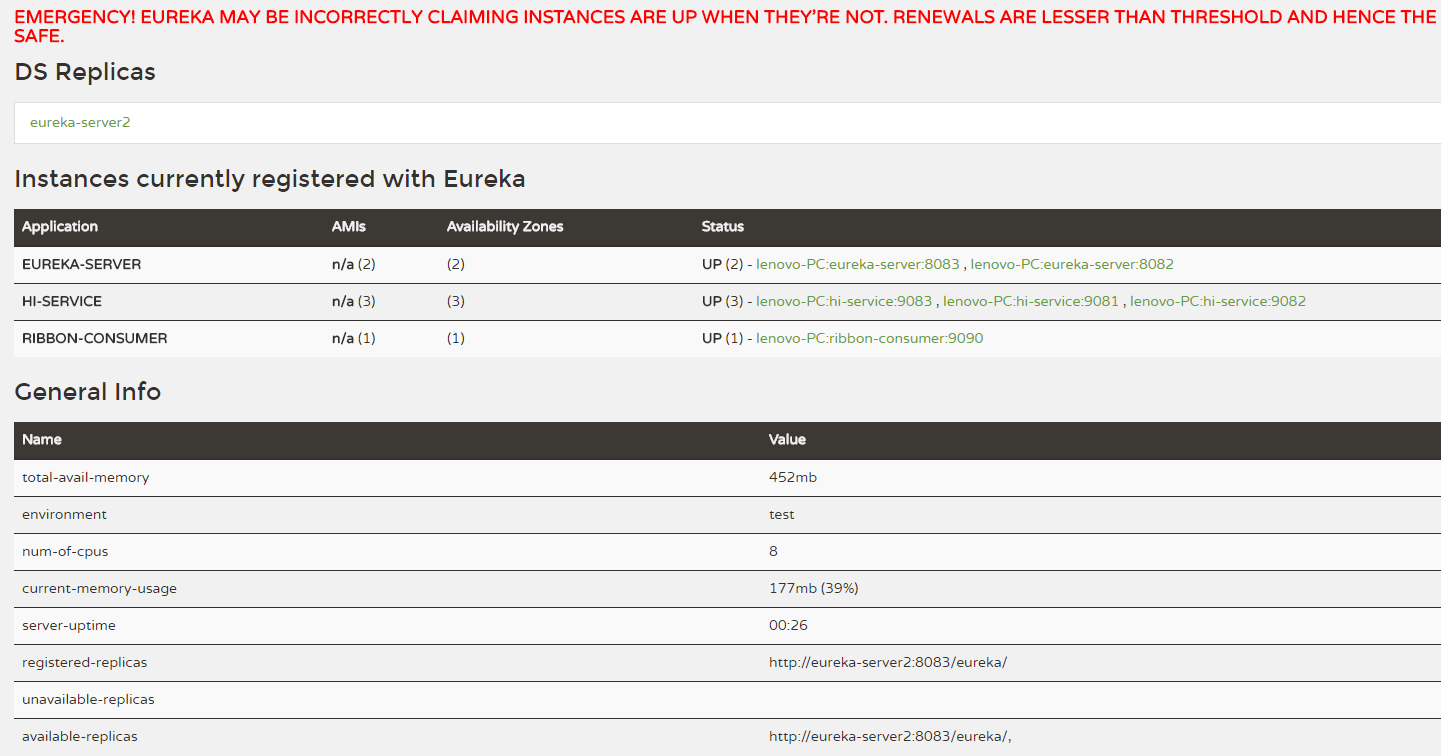
public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

} 



### 1.2、服务注册、发现与消费

通过上一节的操作完成注册中心的构建，这里可以在服务治理框架下，向注册中心注册服务、发现，当然要是不在服务治理框架下，服务的消费需要制定服务的消费地址，后续会说道如何去发现消费。

Eureka客户端主要处理服务注册与发现，通过注解与配置的方式来完成功能的嵌入。相关主要配置及API如下：

|  |  |  |
| --- | --- | --- |
| 序号 | 注解API | 释义 |
| 1 | EnableDiscoveryClient | 服务注册与发现 |
| 2 | EnableEurekaClient | 服务注册与发现（只对Eureka有效） |
| 3 | LoadBalanced | 负载均衡器 |
| 4 | RestTemplate | 访问http服务器，遵循restful规则 |
|  |  |  |

服务注册、发现与消费新增spring cloud eureak模块依赖和ribbon依赖具体依赖如下:

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

服务注册配置：

服务实例1：

spring:

profiles:

active: dev

application:

name: hi-service

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8082/eureka/,http://localhost:8083/eureka/

healthcheck:

enabled: true

prefer-same-zone-eureka: true

# region:

# defaultZone: http://192.168.101.130:8082/eureka/,http://192.168.101.130:8083/eureka/

instance:

metadata-map:

zone: zone2

# prefer-ip-address: true

management:

security:

enabled: false

debug: true

服务实例2:

server:

port: 9081

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8082/eureka/,http://localhost:8083/eureka/

healthcheck:

enabled: true

prefer-same-zone-eureka: true

instance:

metadata-map:

zone: zone1

spring:

application:

name: hi-service

management:

security:

enabled: false

debug: true

服务消费配置:

spring.application.name=ribbon-consumer

server.port=9090

eureka.client.service-url.defaultZone=http://localhost:8082/eureka/,http://localhost:8083/eureka/

#eureka.client.use-dns-for-fetching-service-urls=true

#eureka.client.eureka-server-d-n-s-name=eureka-server1

#eureka.client.eureka-server-port=8083

#eureka.client.eureka-server-u-r-l-context=eureka

eureka.client.region=us-east-1

eureka.client.availability-zones.us-east-1=zone1,zone2

eureka.client.prefer-same-zone-eureka=true ##区域感知故障

#eureka.instance.prefer-ip-address=true

通过以上看出都引入了spring-boot-starter-actuator模块，该模块实现了对应用系统的监控指标采集，也可以扩展自身系统的个性化监控，这里不做详述，后续会对该模块做出说明。

## 2、客户端负载均衡spring cloud ribbon

Spring cloud ribbon是通过spring cloud封装基于netflix ribbon实现的，是一个基于http和tcp的客户端负载均衡工具。这一节主要介绍如何通过ribbon来做客户端负载均衡。它会自动创建相关ribbon配置，如eureka和consul这里主要说eureka一起依赖使用。

### 2.1 spring cloud ribbon调用方式

1.主要通过以下api来完成客户端负载均衡消费调用:

RestTemplate 会使用ribbon的自动化配置,这里需要注意的是默认配置中失败是不可重试的，需要开启配置spring.cloud.loadbalancer.retry.enabled=true,通常与熔断命令配置，MaxAutoRetries、MaxAutoRetriesNextServer、OkToRetryOnAllOperation属性配合使用

LoadBalanced 开启客户端负载均衡

RibbonClient 一个Ribbon客户声明的配置。添加注释Configuration的任何配置然后注入springclientfactory访问客户端,对服务访问的某一个客户端坐到更详细的配置。参数：

name: ribbon客户端的名字，唯一标识一组客户资源含负载均衡

value: name的同义词

configuration: 自定义@Configuration配置为ribbon客户端。可用重写@Bean去定义组织客户端的行为，比如ILoadBalancer，ServerListFilter，IRule，默认采用RibbonClientConfiguration

Spring cloud netflix提供了一系列默认beans为ribbon(BeanType beanname: className):

\* `IClientConfig` ribbonClientConfig: `DefaultClientConfigImpl`

\* `IRule` ribbonRule: `ZoneAvoidanceRule`

\* `IPing` ribbonPing: `DummyPing`

\* `ServerList<Server>` ribbonServerList: `ConfigurationBasedServerList`

\*`ServerListFilter<Server>` ribbonServerListFilter: `ZonePreferenceServerListFilter`

\* `ILoadBalancer` ribbonLoadBalancer: `ZoneAwareLoadBalancer`

\* `ServerListUpdater` ribbonServerListUpdater: `PollingServerListUpdater`

在使用RibbonClient时，指定configuration时，你可以重写上面这样写beans如：

@Configuration

static class LocalBadClientConfiguration {

public LocalBadClientConfiguration() {

}

@Bean

public IRule loadBalancerRule() {

return new AvailabilityFilteringRule();

}

@Bean

public ILoadBalancer ribbonLoadBalancer(IClientConfig config,

ServerList<Server> serverList, IRule rule, IPing ping) {

return null;

}

RibbonClients可以添加多个RibbonClient配置

2.除了通过注解方式定制服务调用，还可以通过配置来定制格式这也是目前推荐的做法：

<clientName>.<nameSpace>.<propertyName>=<value>

nameSpace默认值是ribbon

ribbon.ReadTimeout=1000表示对所有实例都采用读超时时间为1秒即全局配置

C.ribbon.ReadTimeout=1000表示仅对c实例读超时时间1秒指定客户端配置会覆盖全局配置。

常用动态配置属性propertyName含义见:

com.netflix.client.config.CommonClientConfigKey类配置内容

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 类型 | 描述 |
| 1 | NFLoadBalancerClassName | String | 配置ILoadBalancer接口实现常用负载均衡器有:  com.netflix.loadbalancer.DynamicServerListLoadBalancer、com.netflix.loadbalancer.DynamicServerList.ZoneAwareLoadBalancer  com.netflix.loadbalancer.BaseLoadBalancer |
| 2 | NFLoadBalancerPingClassName | String | 配置IPing接口实现 |
| 3 | NFLoadBalancerRuleClassName | String | 配置IRule接口实现  常用规则:  com.netflix.loadbalancer.ZoneAvoidanceRule、  com.netflix.loadbalancer.AvailabilityFilteringRule、  com.netflix.loadbalancer.WeightedResponseTimeRule、  com.netflix.loadbalancer.RoundRobinRule、  com.netflix.loadbalancer.RandomRule、  com.netflix.loadbalancer.BestAvailableRule |
| 4 | NIWSServerListClassName | String | 配置ServerList接口实现 |
| 5 | NIWSServerListFilterClassName | String | 配置ServerListFilter接口实现 |
| 6 | listOfServers | String | 服务实例服务器，在没有服务治理引入时采用用逗号分隔 |
|  | ReadTimeout | Integer |  |
|  | MaxAutoRetries | Integer |  |
|  | MaxAutoRetriesNextServer | Integer |  |
|  | OkToRetryOnAllOperations | Boolean |  |
|  | RequestSpecificRetryOn | Boolean |  |
|  | ReceiveBufferSize | Integer |  |
|  | EnablePrimeConnections | Boolean |  |
|  | MaxRetriesPerServerPrimeConnection | Integer |  |
|  | MaxTotalTimeToPrimeConnections | Integer |  |
|  | PoolMaxThreads | Integer |  |
|  | PoolMinThreads | Integer |  |
|  | EnableConnectionPool | Boolean |  |
|  | MaxConnectionsPerHost | Integer |  |
|  | MaxTotalConnections | Integer |  |
|  | ConnectTimeout | Integer |  |
|  | BackoffTimeout | Integer |  |
|  | ReadTimeout | Integer |  |
|  | SendBufferSize | Integer |  |
|  | FollowRedirects | Boolean |  |
|  | ConnectionPoolCleanerTaskEnabled | Boolean |  |
|  | ConnIdleEvictTimeMilliSeconds | Integer |  |
|  | ConnectionCleanerRepeatInterval | Integer |  |
|  | EnableGZIPContentEncodingFilter | Boolean |  |
|  | ProxyHost | String |  |
|  | ProxyPort | Integer |  |
|  | KeyStore | String |  |
|  | KeyStorePassword | String |  |
|  | TrustStore | String |  |
|  | ServerListRefreshInterval | Integer |  |

参数属性参考类:com.netflix.client.config.DefaultClientConfigImpl

通过spring cloud ribbon，可以很容易的实现客户端负载简单步骤如下:

1. 服务提供者将自身服务注册到服务注册中心
2. 服务消费者直接通过调用@LoadBalanced注解修饰的RestTemplate来实现服务的接口调用。

在上一章节中有说道怎么使用，这里主要说明记着请求类型的处理。

|  |  |
| --- | --- |
| 请求类型 | 方法 |
| Get请求 | getForEntity |
| getForObject |
| POST请求 | postForObject |
| postForEntity |
|  |
|  |
| PUT请求 | put |
|  |
| DELETE请求 | delete |
|  |

### 2.2 Ribbon自动化配置

Ribbon提供了一系列接口来对不同策略的实现，而spring cloud ribbon则自动化配置帮助解决如何引入这些。

1.IClientConfig：定义了客户端配置可以被不同api使用去初始化客户端或者复制，默认实现DefaultClientConfigImpl。

2.IRule: 定义了一种规则为负载均衡策略默认才有ZoneAvoidanceRule

3.IPing: 定义如何去检测服务是否存活的接口，默认才有NoOpPing该实现默认认为所以服务是可用的，无任何意义。

4.ServerList：定义了一组方法维护服务列表，默认才用ConfigurationBasedServerList

5.ServerListFilter：这个接口允许过滤的配置或动态获得获取一定条件的服务器列表。

6.ILoadBalancer：负载均衡器接口默认采用ZoneAwareLoadBalancer具备区域感知力，会根据度量标准来选定可用区，负载都是在最新的统计分析下进行的。

这是在没有引入spring cloud eureka时默认的配置，在加入eureka 和 ribbon时，可用根据需求自定义配置，正如本节开始说的那样。

Spring cloud eureka与spring cloud ribbon同时依赖，会有eureka完成对ribbon的自动化配置。这个时候一下参数会自动被eureka的实现会被覆盖：

###服务清单列表交给服务治理来维护,特别说明

@Bean

@ConditionalOnMissingBean

public ServerList<?> ribbonServerList(IClientConfig config, Provider<EurekaClient> eurekaClientProvider) {

if (this.propertiesFactory.isSet(ServerList.class, serviceId)) {

return this.propertiesFactory.get(ServerList.class, config, serviceId);

}

DiscoveryEnabledNIWSServerList discoveryServerList = new DiscoveryEnabledNIWSServerList(

config, eurekaClientProvider);

DomainExtractingServerList serverList = new DomainExtractingServerList(

discoveryServerList, config, this.approximateZoneFromHostname);

return serverList;

}

上述这段代码DomainExtractingServerList为了让服务维护根据通用使用物理元数据来进行负载。

ribbon.NIWSServerListClassName=com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList

###实例检查任务交给服务治理来维护

ribbon.NFLoadBalancerPingClassName=com.netflix.niws.loadbalancer.NIWSDiscoveryPing

以上是spring cloud eureka与spring cloud ribbon的结合使用，如果不想eureka管理ribbon，可以禁用eureka，使用ribbon.eureka.enable=false配置.

## 3 服务容错保护 Spring Cloud Hystrix

Spring Cloud Hystrix实现了对断路器、线程隔离等一系列保护措施，它也是基于Netflix开源框架Hystrix实现的，目标在于通过控制远程访问节点，对延迟和故障提供容错能力。Hystrix具有服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并和监控等功能。

### 3.1 Hystrix介绍

1.这里我们结合前面提供的实例，来继续加入容错能力的消费端Hystrix。

这里我需要新引入POM依赖，让Hystrix能够找着依赖

<dependency>

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

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

</dependency>

1. 相关注解API说明

|  |  |  |
| --- | --- | --- |
| 序号 | API注解 | 说明 |
| 1 | @SpringCloudApplication | 一个标准的spring cloud应用会开启服务注册与发现、熔断机制 |
| 2 | @EnableCircuitBreaker | 启用熔断 |
| 3 | @EnableHystrix | 启用熔断仅对eureka启用 |
|  | @HystrixCommand | 熔断命令 |
| 4 | @HystrixCollapser | 此注释用于将一些命令合并到单个后端依赖调用中，必须和HystrixCommand一起使用 |
| 5 | @HystrixProperty | 为HystrixCommand命令指定属性格式：name=value |
| 6 | @DefaultProperties | 指定HystrixCommand默认属性比如指定Hystrix线程池属性见类HystrixThreadPoolProperties:corePoolSize、maximumPoolSize、 |
| 7 | com.netflix.hystrix.HystrixCommand | 用于包装代码在执行时有潜在风险的功能（如远程调用服务-通过网络）故障和延迟性，统计数据和性能指标的采集、断路器和舱壁的功能。这个命令是阻塞命令但提供了一个observe()观察，可以自定义继承此类来创建命令 |
| 8 | com.netflix.hystrix.contrib.javanica.command.AsyncResult | 封装命令异步调用逻辑 |
| 9 | com.netflix.hystrix.contrib.javanica.annotation.ObservableExecutionMode | Hystrix Command运行模式:  LAZY、EAGER |
|  | com.netflix.hystrix.HystrixCollapser | 请求命令合并，通常与7结合使用 |
|  | @CacheReuslt | 标记请求命令需要被缓存，需要和@HystrixCommand同步使用，属性cacheKeyMethod |
|  | @CacheRemove | 让请求命令的缓存失效，失效的缓存根据定义的chachekey来决定，属性：commandKey,cacheKeyMethod |
|  | @CacheKey | 请求命令的参数上标记，使其作为缓存Key的值，如果没有标注则会使用所有参数。如果同时使用了CacheResult和CacheRemove注解的cacheKeyMethod那么该CacheKey不会起作用 |

Hystrix提供的命令处理方式处理上述说过继承、注解来完成熔断以外，还提供更加丰富的、灵活的配置方式。

### 3.2 Hystrix配置

Hystrix提供了非常丰富、灵活的命令配置方式。这里主要介绍各自属性用途及含义。

在HystrixPropertiesStrategy提供了非常丰富的属性，如请求指令HystrixCommand、线程池属性HystrixThreadPoolProperties、批量请求指令HystrixCollapserProperties、定时线程属性HystrixTimerThreadPoolProperties，会在接下来几节详述。

他们有则一定的优先级：

1.全局默认值：在没有设置下面三个级别的属性时，属性都是默认的，都是在代码中作的默认定义。

2.全局配置属性值；通过配置文件定义全局属性值，在应用启动时或者在于spring cloud config和spring cloud bus结合实现动态刷新配置功能下，可以实现对全局配置的默认覆盖，以及运行是对全局配置属性的动态调整。

3.实例默认值：通过实例代码为实例定义属性的默认值，会覆盖相应的全局配置。

4.实例配置属性值：通过配置文件来指定的实例进行属性配置，以覆盖前面3个级别的属性配置，也可以与spring cloud config和spring cloud bus结合实现对具体实例动态刷新配置功能。

特别说明在进行采集指标参数配置时，timeInMillisecond必须能被numBuckets参数整除，不然会抛出异常IllegalArgumentException---The timeInMilliseconds must divide equally into numberOfBuckets

#### 3.2.1 HystrixCommand命令属性

请求指令HystrixCommand有5类不同类型的配置。

##### 3.2.1.1 Exceution配置

下面这些属性控制着请求指令执行run

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| execution.isolation.strategy | 配置请求指令执行的隔离策略2种:  THREAD是线程池隔离策略  , SEMAPHORE是信号了隔离策略 | 全局默认 | THREAD |
| 全局配置属性 | hystrix.command.default.execution.isolation.strategy |
| 实例默认配置 | HystrixCommandProperties.Setter()..withExecutionIsolationStrategy(ExecutionIsolationStrategy.THREAD);也可以配置注解@HystrixProperty(name = "execution.isolation.strategy", value = "THREAD") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.isolation.strategy |
| execution.isolation.thread.timeoutInMilliseconds | 请求指令执行超时时间单位毫秒，超时后，会进入降级逻辑 | 全局默认 | 1000毫秒 |
| 全局配置属性 | hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds |
| 实例默认配置 | HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(int value),也可以通过注解@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.isolation.thread.timeoutInMilliseconds |
| execution.timeout.enabled | 请求指令是否启用超时 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.execution.timeout.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withExecutionTimeoutEnabled(boolean enabled)，也可以通过注解@HystrixProperty(name = "execution.timeout.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.timeout.enabled |
| execution.isolation.thread.interruptOnTimeout | 请求指令执行超时是否需要中断指令 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.execution.isolation.thread.interruptOnTimeout |
| 实例默认配置 | HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnTimeout(boolean v)，也可通过注解@HystrixProperty(name = "execution.isolation.thread.interruptOnTimeout", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnTimeout |
| execution.isolation.thread.interruptOnFutureCancel | 请求指令在执行取消时是否要中断 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.execution.isolation.thread.interruptOnFutureCancel |
| 实例默认配置 | HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnFutureCancel(boolean value)，也可以注解@HystrixProperty(name = "execution.isolation.thread.interruptOnFutureCancel", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnFutureCancel |
| execution.isolation.semaphore.maxConcurrentRequests | 当请求指令执行信号量策略的时候，该属性用来表示信号量的大小(并发请求数)，当值达到最大时，后续请求会被拒绝 | 全局默认 | 10 |
| 全局配置属性 | hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests |
| 实例默认配置 | HystrixCommandProperties.Setter().withExecutionIsolationSemaphoreMaxConcurrentRequests(int value),也可以注解@HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests", value = "1") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.execution.isolation.semaphore.maxConcurrentRequests |

##### 3.2.1.2 Fallback配置

下面这些属性控制着请求指令执行getFallback

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| fallback.isolation.semaphore.maxConcurrentRequests | 从调用线程中允许执行getfallbakc的最大并发执行数，当达到最大请求时，后续请求会被拒绝并抛出异常 | 全局默认 | 10 |
| 全局配置属性 | hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests |
| 实例默认配置 | HystrixCommandProperties.Setter().withFallbackIsolationSemaphoreMaxConcurrentRequests(int value) 也可以注解@HystrixProperty(name = "fallback.isolation.semaphore.maxConcurrentRequests", value = "1") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.fallback.isolation.semaphore.maxConcurrentRequests |
| fallback.enabled | 表示服务降级策略是否可用，设置false时，当请求失败是，不会调用降级 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.fallback.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withFallbackEnabled(boolean value) 也可以注解@HystrixProperty(name = "fallback.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.fallback.enabled |

##### 3.2.1.3 Circuit配置

下面是断路器的属性配置，用来控制断路器的行为HystrixCircuitBreaker

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| circuitBreaker.enabled | 当服务请求命令失败时，是否使用断路器来跟踪其健康指标和熔断请求 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.circuitBreaker.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerEnabled(boolean value) 也可以注解@HystrixProperty(name = "circuitBreaker.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.enabled |
| circuitBreaker.requestVolumeThreshold | 在活动时间窗口中，断路器熔断的最小请求数 | 全局默认 | 20 |
| 全局配置属性 | hystrix.command.default.circuitBreaker.requestVolumeThreshold |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerRequestVolumeThreshold(int value) 也可以注解@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "30") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.requestVolumeThreshold |
| circuitBreaker.sleepWindowInMilliseconds | 断路器打开后的休眠时间，休眠时间结束后会将断路器设为半开状态，尝试熔断请求命令，如果依然失败就将断路器设为打开状态，如果成功就关闭断路器 | 全局默认 | 5000 |
| 全局配置属性 | hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerSleepWindowInMilliseconds(int value) 也可以注解@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "3000") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.sleepWindowInMilliseconds |
| circuitBreaker.errorThresholdPercentage | 设置断路器打开的错误百分比条件 | 全局默认 | 50 |
| 全局配置属性 | hystrix.command.default.circuitBreaker.errorThresholdPercentage |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerErrorThresholdPercentage(int value) 也可以注解@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.errorThresholdPercentage |
| circuitBreaker.forceOpen | 设置为true，断路器会强制进入打开，拒绝所有请求。属性优先于circuitBreaker.forceClosed | 全局默认 | false |
| 全局配置属性 | hystrix.command.default.circuitBreaker.forceOpen |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerForceOpen(boolean value) 也可以注解@HystrixProperty(name = "circuitBreaker.forceOpen", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.forceOpen |
| circuitBreaker.forceClosed | 设置true,断路器将强制进入关闭状态，会接收所有请求。如果circuitBreaker.forceOpen=true，则该属性不生效 | 全局默认 | false |
| 全局配置属性 | hystrix.command.default.circuitBreaker.forceClosed |
| 实例默认配置 | HystrixCommandProperties.Setter().withCircuitBreakerForceClosed(boolean value) 也可以注解@HystrixProperty(name = "circuitBreaker.forceClosed", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.circuitBreaker.forceClosed |
|  |  |  |  |

##### 3.2.1.4 Metric配置

下面这些属性是在HystrixCommand和HystrixObservableCommand执行中捕获的指标信息相关

说明：该参数只在应用启动时生效，不能动态刷新，为了避免运行期间出现检测数据丢失

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| metrics.rollingStats.timeInMilliseconds | 设置活动窗口时间单位毫秒。用于断路器判断健康度时需要收集信息的持续时间。断路器会根据设置的时间在收集信息指标的时候将时间窗口长度拆分成多个桶来累计度量。每个桶记录一段时间内的采集指标 | 全局默认 | 10000 |
| 全局配置属性 | hystrix.command.default.metrics.rollingStats.timeInMilliseconds |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowInMilliseconds(int value) 也可以注解@HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "20000") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingStats.timeInMilliseconds |
| metrics.rollingStats.numBuckets | 设置活动时间窗口统计指标划分桶的数量 | 全局默认 | 10 |
| 全局配置属性 | hystrix.command.default.metrics.rollingStats.numBuckets |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowBuckets(int value) 也可以注解@HystrixProperty(name = "metrics.rollingStats.numBuckets", value = "20") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingStats.numBuckets |
| metrics.rollingPercentile.enabled | 对请求命令执行的延迟是否采用百分比来跟踪和计算 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.metrics.rollingPercentile.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingPercentileEnabled(boolean value) 也可以注解@HystrixProperty(name = "metrics.rollingPercentile.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingPercentile.enabled |
| metrics.rollingPercentile.timeInMilliseconds | 设置用百分比统计的活动窗口的持续时间单位毫秒 | 全局默认 | 60000 |
| 全局配置属性 | hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowInMilliseconds(int value) 也可以注解@HystrixProperty(name = "metrics.rollingPercentile.timeInMilliseconds", value = "60000") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingPercentile.timeInMilliseconds |
| metrics.rollingPercentile.numBuckets | 设置百分比统计时桶的数量 | 全局默认 | 6 |
| 全局配置属性 | hystrix.command.default.metrics.rollingPercentile.numBuckets |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowBuckets(int value) 也可以注解@HystrixProperty(name = "metrics.rollingPercentile.numBuckets", value = "6") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingPercentile.numBuckets |
| metrics.rollingPercentile.bucketSize | 设置执行过程中每个桶保留的最大执行次数。如果执行期间超过设定的次数，就从最初位置开始重写。  说明：该值大小会影响内存的消耗，并增加排序百分位数计算的时间消耗 | 全局默认 | 100 |
| 全局配置属性 | hystrix.command.default.metrics.rollingPercentile.bucketSize |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsRollingPercentileBucketSize(int value) 也可以注解@HystrixProperty(name = "metrics.rollingPercentile.bucketSize", value = "100") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.rollingPercentile.bucketSize |
| metrics.healthSnapshot.intervalInMilliseconds | 设置采集影响断路器状态的间隔等待时间 | 全局默认 | 500 |
| 全局配置属性 | hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds |
| 实例默认配置 | HystrixCommandProperties.Setter().withMetricsHealthSnapshotIntervalInMilliseconds(int value) 也可以注解@HystrixProperty(name = "metrics.healthSnapshot.intervalInMilliseconds", value = "500") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.metrics.healthSnapshot.intervalInMilliseconds |

##### 3.2.1.5 Request配置

下面的参数属性涉及到HytrixCommand使用的熔断请求上下文属性设置HystrixRequestContext

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| requestCache.enabled | 是否开启请求缓存 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.requestCache.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withRequestCacheEnabled(boolean value) 也可以注解@HystrixProperty(name = "requestCache.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.requestCache.enabled |
| requestLog.enabled | 是否记录请求名和事件的日志 | 全局默认 | true |
| 全局配置属性 | hystrix.command.default.requestLog.enabled |
| 实例默认配置 | HystrixCommandProperties.Setter().withRequestLogEnabled(boolean value) 也可以注解@HystrixProperty(name = "requestLog.enabled", value = "false") |
| 实例配置属性 | hystrix.command.HystrixCommandKey.requestLog.enabled |

#### 3.2.2 HystrixCollaper 属性

该属性除了在代码和配置文件中设置外，也可以通过注解进行配置@HystrixCollapser()的collapserProperties属性进行设置

下面这些属性控制着其行为

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| maxRequestsInBatch | 设置一次请求合并批处理的最大请求数 | 全局默认 | Integer.MAX\_VALUE |
| 全局配置属性 | hystrix.collapser.default.maxRequestsInBatch |
| 实例默认配置 | HystrixCollapserProperties.Setter().withMaxRequestsInBatch(int value)也可以注解@HystrixProperty(name = "maxRequestsInBatch", value = "30") |
| 实例配置属性 | hystrix.collapser.HystrixCollapserKey.maxRequestsInBatch |
| timerDelayInMilliseconds | 设置批处理过程每个命令延迟的时间单位毫秒 | 全局默认 | 10 |
| 全局配置属性 | hystrix.collapser.default.timerDelayInMilliseconds |
| 实例默认配置 | HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(int value)也可以注解@HystrixProperty(name = "timerDelayInMilliseconds", value = "30") |
| 实例配置属性 | hystrix.collapser.HystrixCollapserKey.timerDelayInMilliseconds |
| requestCache.enabled | 批处理过程中是否开启请求缓存 | 全局默认 | true |
| 全局配置属性 | hystrix.collapser.default.requestCache.enabled |
| 实例默认配置 | HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(int value)也可以注解@HystrixProperty(name = "requestCache.enabled", value = "false") |
| 实例配置属性 | hystrix.collapser.HystrixCollapserKey.requestCache.enabled |
| metrics.rollingStats.timeInMilliseconds | 用于采集批处理过程中信息指标参数含义和HystrixCommand一样 | | |
| metrics.rollingStats.numBuckets |
| metrics.rollingPercentile.enabled |
| metrics.rollingPercentile.timeInMilliseconds |
| metrics.rollingPercentile.numBuckets |
| metrics.rollingPercentile.bucketSize |

#### 3.2.3线程池threadPool属性

线程池属性能在代码、配置文件、注解中进行配置

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名称 | 描述 | 属性级别 | 默认值、配置方式、配置属性 |
| allowMaximumSizeToDivergeFromCoreSize | 是否运行去设置线程池maximumPoolSize，默认是不会启用的，这个需要使用者根据自己业务情况开启 | 全局默认 | fasle |
| 全局配置属性 | hystrix.threadpool.default.timerDelayInMilliseconds |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withAllowMaximumSizeToDivergeFromCoreSize(boolean value)也可以注解@HystrixProperty(name = "allowMaximumSizeToDivergeFromCoreSize", value = "false") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.allowMaximumSizeToDivergeFromCoreSize |
| coreSize | 设置线程池核心线程数，也是请求命令执行最大并发数 | 全局默认 | 10 |
| 全局配置属性 | hystrix.threadpool.default.requestCache.enabled |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withCoreSize(int value)也可以注解@HystrixProperty(name = "coreSize", value = "30") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.oreSize |
| maximumSize | 线程池最大大小 | 全局默认 | 10 |
| 全局配置属性 | hystrix.threadpool.default.maximumSize |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withTimerDelayInMilliseconds(int value)也可以注解@HystrixProperty(name = "maximumSize", value = "30") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.maximumSize |
| keepAliveTimeMinutes | 线程由idle装等转为working等等时间单位分钟 | 全局默认 | 1 |
| 全局配置属性 | hystrix.threadpool.default.keepAliveTimeMinutes |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withKeepAliveTimeMinutes(int value)也可以注解@HystrixProperty(name = "keepAliveTimeMinutes", value = "false") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.keepAliveTimeMinutes |
| maxQueueSize | 设置线程池最大队列大小，设置-1表示会使用SynchronousQueue队列，否则会使用LinkedBlockingQueue队列 | 全局默认 | -1 |
| 全局配置属性 | hystrix.threadpool.default.maxQueueSize |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withMaxQueueSize(int value)也可以注解@HystrixProperty(name = "maxQueueSize", value = "30") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.maxQueueSize |
| queueSizeRejectionThreshold | 设置队列拒绝阈值，表示没有达到队列最大数也会拒绝请求。该参数是对LinkedBlockQueue的补充说明,LinkedBlockQueue是不能动态修改器队列大小的，通过这个操作可以拒绝请求接入 | 全局默认 | 5 |
| 全局配置属性 | hystrix.threadpool.default.queueSizeRejectionThreshold |
| 实例默认配置 | HystrixThreadPoolProperties.Setter().withQueueSizeRejectionThreshold(int value)也可以注解@HystrixProperty(name = "queueSizeRejectionThreshold", value = "10") |
| 实例配置属性 | hystrix.threadpool.HystrixThreadPoolKey.queueSizeRejectionThreshold |
| metrics.rollingStats.timeInMilliseconds | 用于采集线程中信息指标参数含义和HystrixCommand一样 | | |
| metrics.rollingStats.numBuckets |

### 3.3 Hystrix监控

前面提到了对应用指标的采集信息，这里通过Hystrix Dashboard来实时监控hrstrix的各项指标信息。

#### 3.3.1 构建简单的监控服务

1. 创建spring boot应用
2. 引入pom依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

3.在应用主类加上@EnableHystrixDashboard，启用Hrstrix Dashboard功能

4.在配置文件application-{profile}.yml中指定配置如

spring:

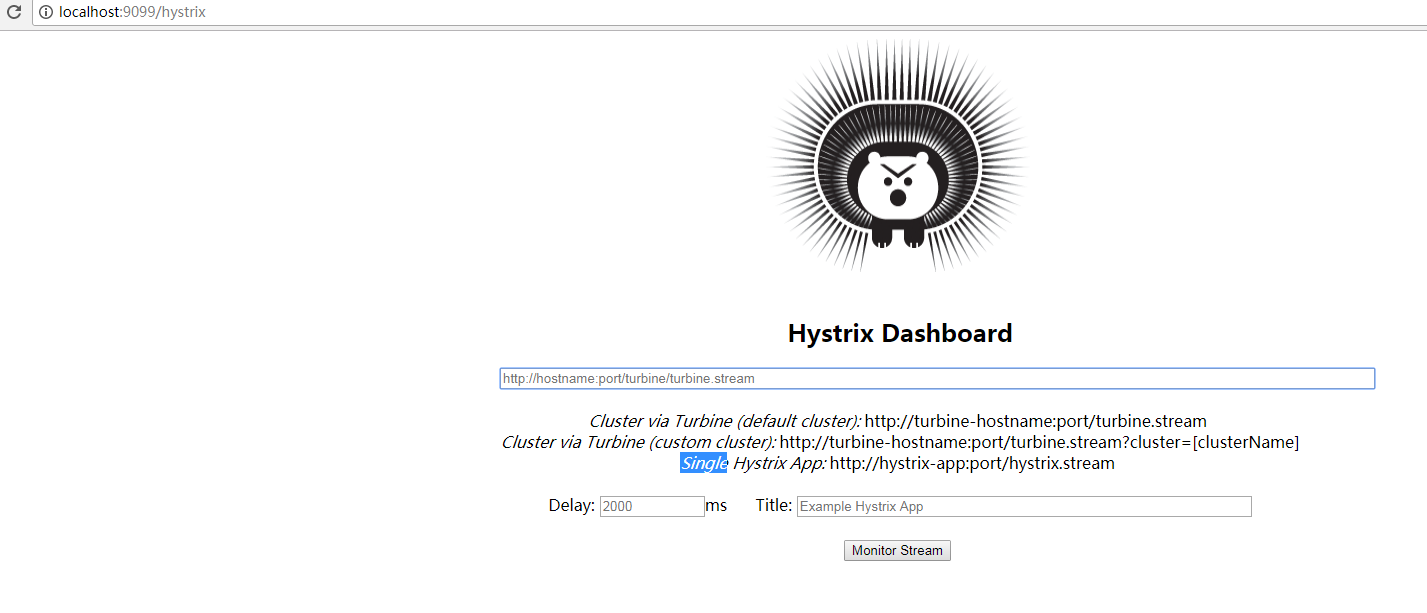
application:

name: hystrix-dashboard

server:

port: 9099

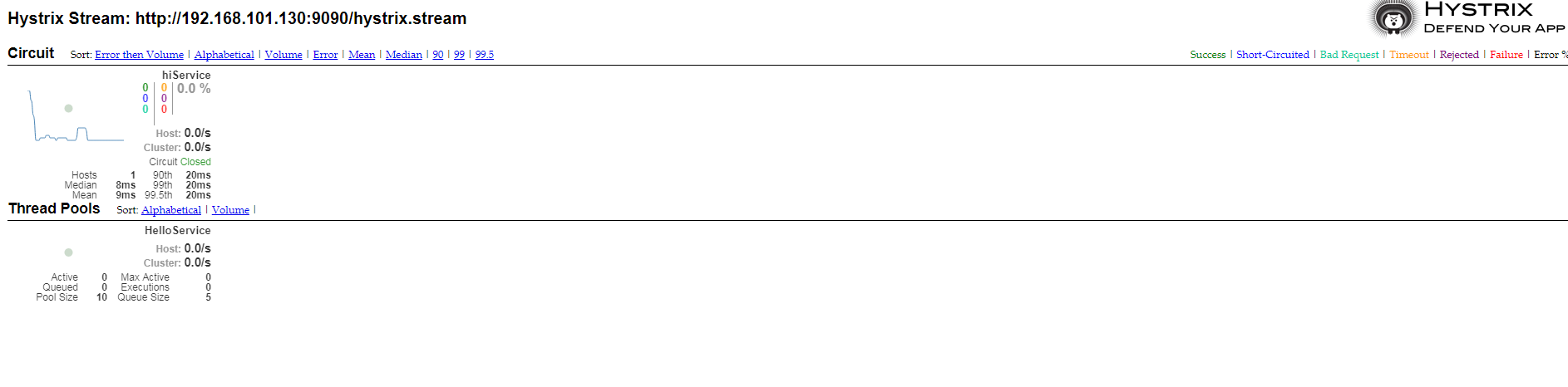
5.启动应用后，进入http://localhost:9099/hystrix,可以看见这样的场景



从监控首页我们能看出，支持三种不同的监控方式：

1. 默认的集群监控：通过URL：<http://turbine-hostname:port/turbine.stream开启对默认集群的监控>
2. 指定的集群监控：通过URL：[http://turbine-hostname:port/turbine.stream?cluster=[claustername]](http://turbine-hostname:port/turbine.stream?cluster=%5bclaustername%5d),开启对clusterName的集群监控
3. 单体应用的监控：通过URL <http://hystrix-app:port/hystrix.stream开启实现对谋个具体服务实例的监控>

对单体应用监控简单实例如下：



单个实例的维护监控不方便。接下来我们看集群的监控

#### 3.3.2 Turbine 集群监控

构建监控服务，在之前用例基础上pom我们引入

1. <dependency>

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

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

</dependency>

2.在应用主类加入注解@EnableTurbine，开启Turbine功能

3.在配置文件application中加入Turbine相关配置如下：

spring.application.name=turbine

server.port=9070

turbine.combine-host-port=true

eureka.client.service-url.defaultZone=http://localhost:8082/eureka/,http://localhost:8083/eureka/

#1集群配置方式一

#turbine.appConfig=ribbon-consumer,2ribbon-consumer

#turbine.aggregator.clusterConfig= default

#turbine.clusterNameExpression= new String("default")

#turbine.clusterNameExpression="'default'"（bad）

#http://localhost:9070/turbine.stream

#2集群配置方式二

turbine.appConfig=ribbon-consumer,2ribbon-consumer

turbine.aggregator.clusterConfig=2RIBBON-CONSUMER,RIBBON-CONSUMER

# http://localhost:9070/turbine.stream?cluster=RIBBON-CONSUMER&delay=30000

#3集群配置方式三

#turbine.appConfig=ribbon-consumer,2ribbon-consumer

#turbine.cluster-name-expression=metadata["cluster"]

#turbine.aggregator.clusterConfig=ABC

#http://localhost:9070/turbine.stream?cluster=ABC

#turbine.appConfig ：配置Eureka中的serviceId列表，表明监控哪些服务

#turbine.aggregator.clusterConfig ：指定聚合哪些集群，多个使用”,”分割，默认为default。

# 可使用http://.../turbine.stream?cluster={clusterConfig之一}访问

#turbine.clusterNameExpression ：

# 1. clusterNameExpression指定集群名称,默认表达式appName；此时：turbine.aggregator.clusterConfig需要配置想要监控的应用名称

# 2. 当clusterNameExpression: default时，turbine.aggregator.clusterConfig可以不写，因为默认就是default；

# 3. 当clusterNameExpression: metadata[‘cluster’]时，假设想要监控的应用配置了eureka.instance.metadata-map.cluster: ABC，

# 则需要配置，同时turbine.aggregator.clusterConfig: ABC

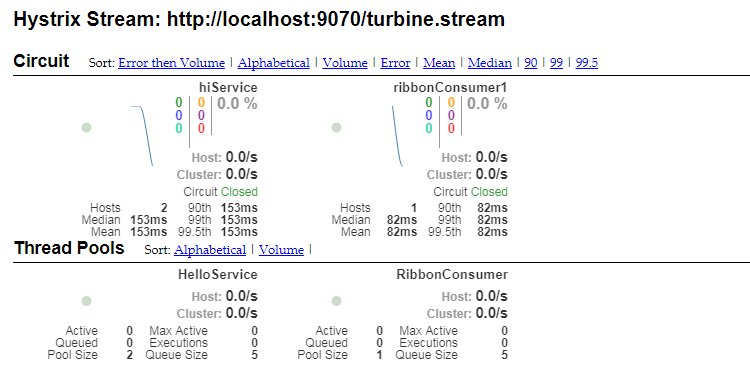
##########

#eureka.client.region=us-east-1

#eureka.client.availability-zones.us-east-1=zone1,zone2

#eureka.client.prefer-same-zone-eureka=true

实例监控效果如下:



#### 3.3.3 Turbine 与消息代理的结合

该节详细会在后续的spring cloud Bus处讲到。

## 4 Spring Cloud Feign声明式服务调用

通过前节知道了服务如何调用，这里会说到spring cloud feign声明式服务调用，基于Netflix Feign的实现，具有ribbon和Hystrix的功能外，提供了额外的声明式web服务客户端的定义方式。在第2大节中我们知道在进行服务调用时需要借用RestTemplate来进行服务调用的封装，而这里spring cloud Feign在此基础上，进一步的封装了服务调用的绑定，简化了spring cloud ribbon自行封装调用时的代码开发量。并且提供可插拔的注解支持(支持Feign和JAX-RS注解)，同时支持MVC注解，也提供对Feign自身一些组件的扩展比如编码器。

### 4.1 简单实例开发

1. 首先需要加入相关依赖项到Pom

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

2. 应用主类引入注解:

@EnableFeignClients, 开启Feign特效

@EnableDiscoveryClient 开启服务注册发现

1. 实例简单调用

@FeignClient(value = "hi-service")

public interface HelloService {

@RequestMapping("hi/h1")

String hi();

}

@RestController

public class HiController {

@Autowired

HelloService helloService;

@RequestMapping("/f-hi")

public String hiService() {

return helloService.hi();

}

}

1. 配置文件引入

spring.application.name=f-hi

server.port=9084

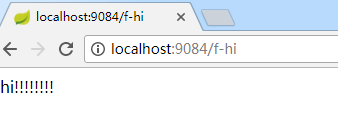
eureka.client.service-url.defaultZone=http://localhost:8082/eureka/,http://localhost:8083/eureka/

eureka.client.region=us-east-1

eureka.client.availability-zones.us-east-1=zone1,zone2

eureka.client.prefer-same-zone-eureka=true

5.应用访问：



通过以上步骤就能完成一个简单的feign服务的搭建

特别说明在使用PathVariable、RequestHeader、RequestParam等可以指定参数名称的注解的时候，必须指定value值，Feign会进行检查，要是没有会抛出异常。

### 4.2 Feign继承特性

通过上面的实例，发现服务提供端和服务调用端有重复代码，而spring cloud feign提供了一种继承特性来解决这一的操作，减少代码量。

### 4.3 Feign配置

由于spring cloud feign是基于spring cloud ribbon实现的，可以使用基于ribbon的配置方式进行配置。也将Feign客户端封装到了Hystrix命令中进行熔断操作，实现服务保护与容错。这里主要说说关于它们的配置项。

1. ribbon配置

Spring cloud feign配置可以直接采用ribbon中的配置来自定义各个服务客户端的调用。

注意：在指定服务名的时候，需要同@FeignClient中定义的名字保持一致如：

@FeignClient(value=”test”)

Test.ribbon.ReadTimeout=2000

2.Hystrix配置

与spring cloud [Hystrix配置保持一致。另外需要注意的是feign.hystrix.enabled参数设置为false会关闭Feign客户端的Hystrix的支持。除了实现对Hystrix全局开关的控制，也能针对谋个服务客户端进行开关Hystrix控制只需要添加@Bean注解的@Scope(“prototype”)为客户端指定客户端配置Feign.Builder实例](mailto:Hystrix配置保持一致。另外需要注意的是feign.hystrix.enabled参数设置为false会关闭Feign客户端的Hystrix的支持。除了实现对Hystrix全局开关的控制，也能针对谋个服务客户端进行开关Hystrix控制只需要添加@Bean注解的@Scope()。

对于Hystrix命令的配置，需要注意的是hystrix.command.commandKey，这里commandKey会采用Feign客户端中声明的方法名作标识。这里需要注意项相同方法名的Hystrix配置会共用。

这里的服务降级与Ribbon中有一定的差别，HystrixCommand命令被Feign封装了，需要在@FeignClient中通过属性fallback来定义服务降级的引用

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 描述 | 属性 | | 实例说明 |
| @FeignClient | 注解接口声明为Rest客户端 | value：服务名  name: 同value  url：服务绝对路径URL或者能识别主机名  decode404：Whether 404s should be decoded instead of throwing FeignExceptions  configuration：A custom @Configuration for the feign client. Can contain override @Bean definition for the pieces that make up the client, for instance feign.codec.Decoder, feign.codec.Encoder, feign.Contract.  fallback：Fallback class for the specified Feign client interface. The fallback class must implement the interface annotated by this annotation and be a valid spring bean.  fallbackFactory：Define a fallback factory for the specified Feign client interface. The fallback factory must produce instances of fallback classes that implement the interface annotated by FeignClient. The fallback factory must be a valid spring bean. | |  |
| @EnableFeignClients | 开启引入Feign客户端功能 | value | Alias for the basePackages() attribute. Allows for more concise annotation declarations e.g.: @ComponentScan("org.my.pkg") instead of @ComponentScan(basePackages="org.my.pkg"). |  |
| basePackages | Base packages to scan for annotated components.  value() is an alias for (and mutually exclusive with) this attribute.  Use basePackageClasses() for a type-safe alternative to String-based package names. |
| basePackageClasses | Type-safe alternative to basePackages() for specifying the packages to scan for annotated components. The package of each class specified will be scanned.  Consider creating a special no-op marker class or interface in each package that serves no purpose other than being referenced by this attribute. |
| defaultConfiguration | A custom @Configuration for all feign clients. Can contain override @Bean definition for the pieces that make up the client, for instance feign.codec.Decoder, feign.codec.Encoder, feign.Contract. |
| clients | List of classes annotated with @FeignClient. If not empty, disables classpath scanning. |
| Feign |  |  | |  |
| Logger.Level | Feign默认日志级别是NONE级别，如果需要指定服务日志级别需要全局引入该日志级别，然后在需要指定的服务上加入日志级别,  NONE, 不记录任何信息，默认值。  BASIC, 记录请求方法、请求URL、状态码和用时。  HEADERS, 在BASIC的基础上再记录一些常用信息。  FULL: 记录请求和响应报文的全部内容。 |  | |  |
| logging.level.feignclient | 其中feignclient为服务接口完整路径 | logging.level.com.spring.cloud.feign.service.HiuService=DEBUG | | |

#Hystrix支持，如果为true，hystrix库必须在classpath中

feign.hystrix.enabled=false

#请求和响应GZIP压缩支持

feign.compression.request.enabled=true

feign.compression.response.enabled=true

#支持压缩的mime types

feign.compression.request.enabled=true

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

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

## Spring cloud Zuul 服务API网关

这里主要说API网关是什么，为什么需要API网关这里不做解释。API网关是外部客户端访问都需要经过它来进行调度和过滤，实现请求路由、负载均衡、校验过滤等功能外，还提供与服务治理框架相结合、请求转发时的熔断机制、服务的聚合等一系列更丰富的操作服务。

Spring cloud zuul 是基于Netflix zuul实现的一套API网关组件，实现这一些列操作：

1. 动态路由
2. 监控
3. 安全
4. 认证鉴权
5. 压力测试
6. 审查
7. 服务迁移
8. 负载剪裁
9. 静态应答处理

### 5.1 构建Simple zuul

1. 同上面一样，构建spring cloud zuul网关应用需要先引入pom配置

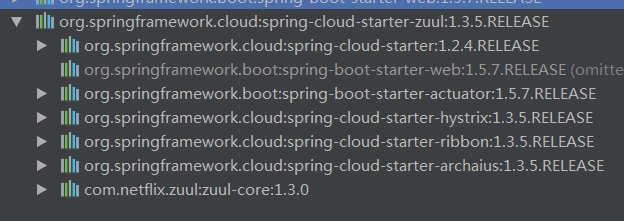
<dependency>

<groupId>org.springframework.boot</groupId>

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

</dependency>

通过查看依赖项



能看见依赖了hystrix：对服务转发时候的保护机制，通过线程隔离和断路器，防止服务故障引发故障；Ribbon：实现了再转发时的客户端服务时的负载均衡及请求重试；actuator：提供常用监控管理以及/routes来获取当前的路由规则。

1. 创建应用启动类@EnableZuulProxy,开启zuul的API网关功能
2. 配置application-{profile}.properties、application-{profile}.yml文件简单配置如下

spring.application.name=api-gateway

server.port=9066

zuul.routes.api-hi-url.path=/hi/\*\* ###所有符合api-hi-url.path规则的访问都会不转发到api-hi-url地址上

zuul.routes.api-hi-url.url=http://localhost:9084

1. 通过上面几步就完成了一个简单网关功能。在此我们还能面向服务进行路由以及完成相应的请求过滤功能

在面向服务治理框架时，我们还需要引入eureka依赖

<dependency>

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

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

</dependency>

在引入依赖后，再加入注册中心配置并配置路由:

zuul.routes.api-hi-url.path=/hi/\*\*

#zuul.routes.api-hi-url.url=http://localhost:9084

zuul.routes.api-hi-url.serviceId=hi-service

zuul.routes.api-hu-url.path=/hi2/\*\*

zuul.routes.api-hu-url.serviceId=hi-service2

zuul.routes.api-rc.path=/rc/\*\*

zuul.routes.api-rc.serviceId=2RIBBON-CONSUMER

zuul.routes.api-fui.path=/fui/\*\*

zuul.routes.api-fui.serviceId=F-HI

eureka.client.service-url.defaultZone=http://localhost:8082/eureka/,http://localhost:8083/eureka/

通过这样的配置就完成了面向服务的路由转发，通过path与serviceId映射组合，借助eureka的服务发现功能，完成zuul的自动化服务实例的维护功能。

5.通过自定义实现ZuulFilter能够处理安全、权限校验，对请求过滤。

### 5.2 路由配置

#### 5.2.1 传统路由配置

非服务治理框架下的路由配置即传统路由配置

1.单实例配置

zuul.routes.api-hi-url.path=/hi/\*\*

zuul.routes.api-hi-url.url=http://localhost:9084

1. 多实例配置

zuul.routes.api-hi-url.path=/hi/\*\*

zuul.routes.api-hi-url.serviceId=hi-service

ribbon.eureka.enabled=false

hi-service.ribbon.listOfServers=http://localhost:9084,http://localhost:9083

这里通过serviceId和listOfServers来实现服务和实例的维护，由Spring Cloud Ribbon来实现负载均衡的，由于默认ribbon会默认启用服务治理发现服务这里，这里需要禁用此功能ribbon.eureka.enabled=false。

在传统配置中，每一对路由映射都需要指定一个名词即api-hi-url,通过url或者serviceId来表示请求具体实例地址或者服务名。

#### 5.2.2 面向服务的路由配置

Spring Cloud zuul 与服务治理框架Spring Cloud eureka结合在一起使用能够实现服务实例的自动化维护。只想配置zuul.routes.<route>.path与zuul.routes.<route>.serviceId结合使用，通过服务治理自行发现服务地址，除了这样配置还能使用zuul.routes.<serviceId>=<path>的方式来进行配置。

##equal zuul.routes.F-HI=/fui/\*\*

#zuul.routes.api-fui.path=/fui/\*\*

#zuul.routes.api-fui.serviceId=F-HI

zuul.routes.F-HI=/fui/\*\*

虽然zuul与eureka自动维护了实例的很多配置，我们只需配置请求路径匹配表达式与服务名间的映射关系。

1. 路由规则

在默认规则下zuul与eureka会将主持中心上的服务自动的创建映射关系来进行路由，这个时候，我们不希望对外的服务有可能被访问到。这里我们可以通过配置来过滤掉那些不想自动被创建路由的服务，只有配置了route的服务会被创建，配置如下：

zuul.routes.F-HI=/fui/\*\*

zuul.ignored-services=\*

#zuul.routes.api-hi-url.path=/hi/\*\*

#zuul.routes.api-hi-url.serviceId=hi-service

1. 自定义规则

在服务治理中，有时候为了升级方便，会以一定规则来创建服务，方便服务升级。比如服务命令遵循：hiService-v1,hiService-v2,这个时候，我们可以在网关创建路由匹配规则类似/v1/hiService/\*\*的路由匹配规则

@Bean

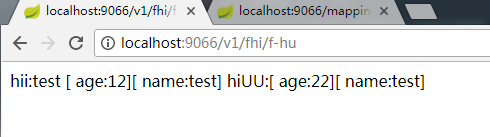
public PatternServiceRouteMapper serviceRouteMapper() {

return new PatternServiceRouteMapper(

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

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

}



没有匹配的会使用默认路由规则进行映射即采用完整服务名作为前缀路径表达式

1. 路径匹配

在Zuul中路径风格采用Ant风格定义。通配符配置如下

|  |  |
| --- | --- |
| 通配符 | 含义 |
| ? | 匹配单个字符 |
| \* | 匹配任意数量的字符 |
| \*\* | 匹配任意数量的字符，支持多级目录 |

路径匹配优先顺序问题：

由于Zuul的路由匹配算法是线性遍历的，因此在配置映射路径的时候，注意配置顺序，由于properties配置内容无法保证有序，为了保证顺序性这里用yaml文件配置，以达到有序的路由。

4. 路由配置能够设置不希望被API网关路由的URL路径，这里的配置是对所有路由而言：

zuul.ignored-services=/\*\*/hi/\*\*

1. 关于Zuul http客户端配置

默认http客户端采用Apache http client替换已经不再使用ribbon rest client。如果想使用需要配置：

ribbon.restclient.enabled=true或ribbon.okhttp.enabled=true

如果想自定义apache http客户端或OK http client需要提供实例bean:ClosableHttpClient` or `OkHttpClient

在使用zuul进行文件传递时，需要注意在传递大文件时，客户端编码传输请求需要指定:Transfer-Encoding: chunked头，因为有些浏览器默认是不支持自动添加的.传输路径前缀：zuul.servlet-path=/testzuul，默认值/zuul

1. zuul关于安全的处理

Zuul作为对外网关，有些敏感信息是不希望对外访问的。如果要实现web应用的一些安全处理操作。Spring cloud zuul提供了敏感头信息过滤。

|  |  |  |
| --- | --- | --- |
| 序号 | 属性 | 描述 |
| 1 | zuul.sensitiveHeaders | 设置全局参数，开启所有请求对敏感参数的传递 |
| 2 | zuul.routes.<router>.customSensitiveHeaders | 对指定路由router开启自定义敏感头，仅对指定web应用路由开启true |
| 3 | zuul.routes.<router>.sensitiveHeaders | 将指定路由router的敏感头设置为空，仅对指定web应用路由开启 |
| 4 | zuul.addHostHeader | 需要将值自定为true防止转发请求路由失败 |
| 5 | zuul.retryable=false | Zuul全局关闭重试机制 |
| 6 | zuul.routes.<router>.retryable=false | Zuul针对某路由关闭重试机制 |

1. zuul对Hystrix 和Ribbon的支持

在构建应用的时候，我们看见zuul对hystrix和ribbon模块做了依赖，所以zuul生来就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能力。这里需要注意的是只有path和serviceId相结合的路由配置才能发挥这些功能。

|  |  |  |
| --- | --- | --- |
| 序号 | 类/接口/属性 | 说明 |
| 1 | ZuulFallbackProvider | 路由失败的时候，可以调用fallback  这个类在将来的发行版本会被FallbackProvider替换掉。  提供两个方法：  string getRoute()：路由调用失败的时候，那些路由可以进行fallback回调，如果想提供对多有路由的fallback调用可以返回\*或者null.  clientHttpResponse fallbackResponse():fallback响应回复给客户端 |

### 5.3 过滤器配

在spring cloud zuul中过滤器完成了路由功能，路由功能的路由映射和请求转发都是由几个不同的过滤器一起完成的。在这里实现的过滤器有几个基本特征：过滤类型、执行顺序、执行条件、具体操作。即ZuulFilter接口中定义的几个方法：

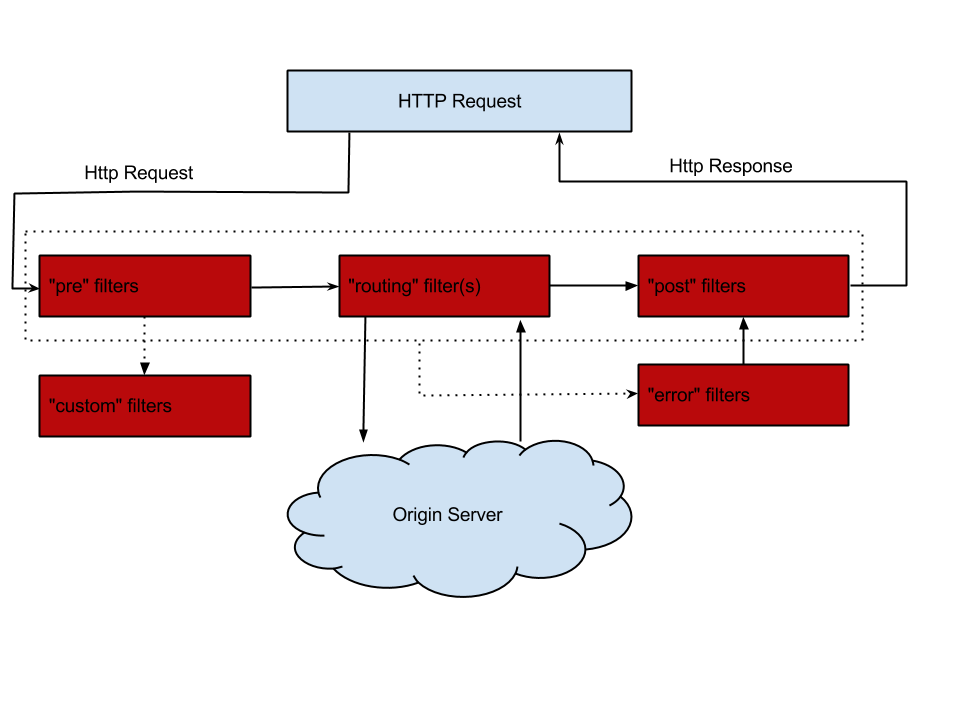
String filterType()：过滤器的类型，Zuul定义了4种不同的生命周期过长：

pre：可以在请求路由调用之前调用；routing:在路由请求时调用；post:在routing和error过滤器之后调用；error:处理请求是发生错误时被调用。

int filterOrder()：定义过滤器执行的顺序，数组越小优先级越高

boolean shouldFilter():过滤器是否需要被执行，可以通过该参数指定过滤器的有效范围

Object run();过滤器的具体逻辑。生命周期图谱：



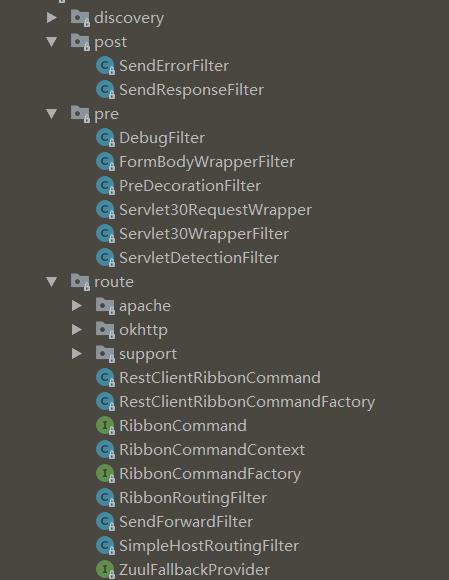
#### 5.3.1 禁用Zuul过滤器

Spring Cloud默认为Zuul编写并启用了一些过滤器，例如DebugFilter、FormBodyWrapperFilter、PreDecorationFilter等。这些过滤器都存放在spring-cloud-netflix-core这个Jar包的org.springframework.cloud.netflix.zuul.filters包中。

一些场景下，我们想要禁用掉部分过滤器，此时该怎么办呢？只需设置zuul.<SimpleClassName>.<filterType>.disable=true ，即可禁用springclassnam所对应的过滤器。以过滤器org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter为例，只需设置zuul.SendResponseFilter.post.disable=true ，即可禁用该过滤器。

#### 5.3.2 核心过滤器

Zuul默认启用3中不同的过滤器对外部请求进行处理，可以在此基础上进行扩展完成自身的一些功能，如下图：



1. pre过滤器

|  |  |  |
| --- | --- | --- |
| 序号 | 类/属性 | 说明 |
| 1 | ServletDetectionFilter | 执行属性-3，最先执行注意检查当前请求是被DispatcherServlet执行还是被ZuulServlet执行 |
| 2 | FormBodyWrapperFilter | 执行顺序-1，过滤器对请求进行包装层FormBodyRequestWrapper对象 |
| 3 | DebugFilter | 执行顺序1，是否执行会根据配置zuul.debug.request和请求debug中参数来进行判断 |
| 4 | PreDecorationFilter | 执行属性5，会判断当前上下文中是否有forward.to和serviceId参数，判断当前过滤器是否该执行，会对路由请求信息进行预加载为后续进行处理提供依据 |
| 5 | Servlet30WrapperFilter | 执行属性-2对所有请求生效，讲原始请求对象封装成Servlet30RequestWrapper |

2. route过滤器

Route过滤器也提供3类过滤器来完成路由转发

RibbonRoutingFilter：执行属性10，完成路由转发通过ribbon和hystrix向服务实例发起请求并返回处理结果

SendForwardFilter: 执行顺序100，该处理器只处理参数中有routeHost参数的请求，这类请求没有线程隔离和断路器保护

SimpleHostRoutingFilter：执行顺序500，请对上下文中存在forward.to参数的请求，用来处理断路器请求中的本地调整

1. post过滤器

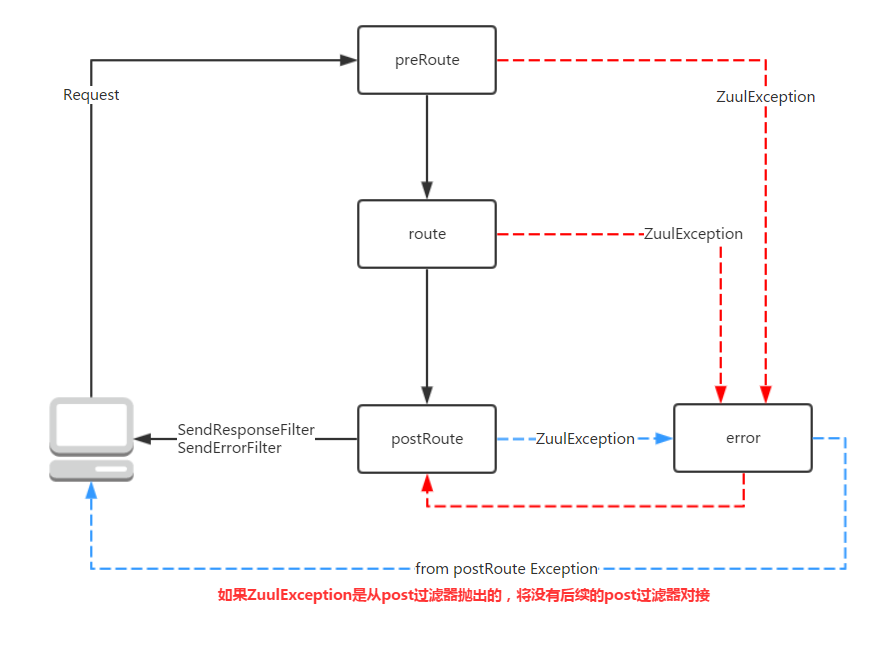
SendResponseFilter：组织上下文信息返回给客户端

LocationRewriteFilter:目前版本不支持，将来版本会引入的对地址转发的处理

4. error过滤器

SendErrorFilter：执行顺序0，利用上下文中的错误信息来组装forward到网关的/error错误端点来生成错误响应

### 5.4 异常处理



针对异常情况，在图中我们标出了不同的颜色。从pre和route阶段抛出的异常将会进入error阶段，再进入到post阶段进行返回。由于SendErrorFilter需要判断请求上下文中是否包含error.status\_code属性：有的话就用SendErrorFilter处理错误结果；没有的话就用SendResponseFilter返回正常结果，但是error.status\_code属性默认是在各个阶段过滤器中自己put进去的，这就导致，各个阶段过滤器抛出异常之后，是没有办法返回错误结果的。因此，我们扩展了一个ErrorFilter来捕获异常，然后手工的设置error.status\_code属性，让SendErrorFilter能正常运作。

通过上面你的改造，从pre和route阶段的异常都能处理了，但是post阶段抛出异常后，是不会再进入post阶段的，这使得ErrorFilter设置了设置error.status\_code属性之后，也没有过滤器去组织返回结果，所以我们通过继承SendErrorFilter在error阶段增加了一个返回错误信息的过滤器。

而这次在Dalston版本中，做了很巧妙的变动：就是上文所述的对SendErrorFilter过滤器类型的变更，这一变动使得所有阶段的异常都会被SendErrorFilter处理，直接解决的上面的第二个问题。当然只是做个变动还是不够的，为了区分SendErrorFilter和SendResponseFitler分别处理出现异常和未出现异常的情况，修改原来根据error.status\_code属性判断的逻辑，而是改为根据请求上下文中是否包含Throwable来作为基本依据，而这个对象是在过滤器出现异常之后，Zuul往请求上下文中置入的，所以可以更为准确的判断当前请求处理是否出现了异常，而不再需要我们之前扩展的ErrorFilter了。

### 5.5 动态加载

通过zuul实现的api网关服务具备动态路由和动态过滤器的功能。可以在不重启网关前提下，为其动态修改路由规则和添加删除过滤器。

在看这里之前可以先了解下[分布式配置中心Spring Cloud Config章](#_分布式配置中心Spring Cloud Config)

1. 动态路由

对于zuul动态路由这里是和spring cloud config相结合一起使用，实心动态路由刷新规则。

1. 动态过滤器

基于请求过滤器的动态加载这里需要借助基于JVM实现的动态语言：groovy

zuul支持使用groovy语言来动态修改filter，它是基于jvm的语言，语法简单并且很多与java类似。

需要先将groovy的jar包引入

<dependency>

<groupId>org.codehaus.groovy</groupId>

<artifactId>groovy-all</artifactId>

</dependency>

动态过滤器配置的参数：

zuul.filter.root：指定动态过滤器加载的过滤器存储路径

zuul.filter.interval: 配置动态加载的间隔时间，单位秒

Zuul通过FilterLoader，FilterFileManager会对groovy文件变更加载如内存：

FilterLoader filterLoader = FilterLoader.getInstance();

filterLoader.setCompiler(new GroovyCompiler());

FilterFileManager.setFilenameFilter(new GroovyFileFilter());

FilterFileManager.init(dynamicFilterConfiguration.getInterval(), dynamicFilterConfiguration.getRoot());

## 6分布式配置中心Spring Cloud Config

Spring cloud config为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，配置中心分服务端和客户端。其实现的配置中心默认采用Git来存储配置信息，也支持其他的SVN，本地化文件系统，这里说到的全是以git为基准。

### 6.1 构建配置中心

构建一个简单的配置中心：

1. 引入pom依赖

服务端

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

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

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

</dependency>

客户端：

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

1. 在配置文件中添加配置中心的基本配置信息以及git仓库信息：

spring:

application:

name: conf-srv

cloud:

config:

server:

git:

uri: ### git仓库位置

username: ###仓库用户名

password: ###git仓库密码

search-paths: ###配置仓库路径下的相对搜索位置，多个位置逗号分隔

server:

port: 7001

4 .启动类添加@EnableConfigServer，激活对配置中心的支持

@EnableDiscoveryClient

@EnableConfigServer

@SpringBootApplication

public class ConfigApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigApplication.class, args);

}

}

仓库中的配置文件会被转换成web接口，访问可以参照以下的规则：

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

/{application}-{profile}.yml

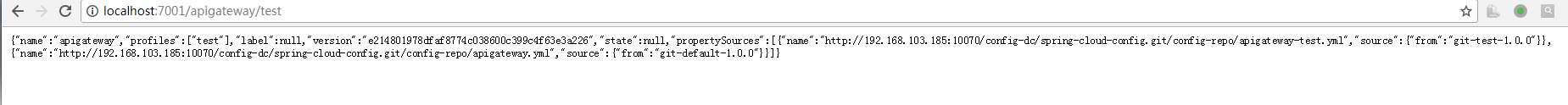
/{label}/{application}-{profile}.yml

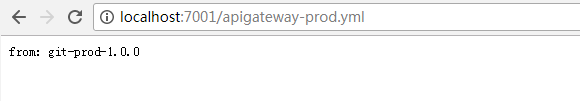
/{application}-{profile}.properties

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

以apigateway-dev.properties为例子，它的application是apigateway，profile是dev，客户端会根据填写的参数来选择读取对应的配置。

测试信息：





5. 客户端简单实例

bootstrap.yml配置如下：

spring:

cloud:

config:

label: master

profile: dev

discovery:

enabled: true

service-id: CONF-SRV

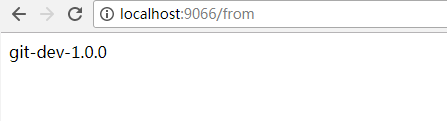
eureka:

client:

service-url:

defaultZone: [http://localhost:8082/eureka/,http://localhost:8083/eureka/](http://localhost:8082/eureka/,http:/localhost:8083/eureka/)

测试信息



spring.application.name：对应{application}部分

spring.cloud.config.profile：对应{profile}部分

spring.cloud.config.label：对应git的分支。如果配置中心使用的是本地存储，则该参数无用

spring.cloud.config.uri：配置中心的具体地址

spring.cloud.config.discovery.service-id：指定配置中心的service-id，便于扩展为高可用配置集群。

1. 配置中心除了支持yml、property文件格式外，也支持纯文本活自定义文件格式如：xml,txt等等。访问格式：/{name}/{profile}/{label}/{path} 如：其中name，profile，label与之前传统环境命名含义一样，path这里指文件名如a.xml

<http://localhost:7001/ribbon-consumer/default/master/a.xml>

注意：就像环境配置的源文件一样，“配置文件”用于解析文件名，所以如果你想要一个特定于文件的文件，那么/\*/development/\*/logback.xml将被一个名为logback-development.xml的文件解析（优先于logback.xml）。

### 6.2 配置中心服务端

配置服务中心含有以下几个部分：

1. 远程存储仓库如git
2. Config server构建配置服务中心
3. 本地git仓库，config server会在本地创建一个远程配置的副本
4. 服务通过指定配置中心地址或服务来访问config server获取自己的配置信息

Git仓库

EnvironmentRepository的默认实现使用Git后端，这对于管理升级和物理环境以及审核更改非常方便。要更改存储库的位置，可以在Config Server中设置“spring.cloud.config.server.git.uri”配置属性（例如application.yml）。如果您使用file:前缀进行设置，则应从本地存储库中运行，以便在没有服务器的情况下快速方便地启动，但在这种情况下，服务器将直接在本地存储库中进行操作，而不会克隆它如果它不是裸机，因为配置服务器永远不会更改“远程”资源库）。要扩展Config Server并使其高度可用，您需要将服务器的所有实例指向同一个存储库，因此只有共享文件系统才能正常工作。即使在这种情况下，最好使用共享文件系统存储库的ssh:协议，以便服务器可以将其克隆并使用本地工作副本作为缓存。

该存储库实现将HTTP资源的{label}参数映射到git标签（提交ID，分支名称或标签）。如果git分支或标签名称包含斜杠（“/”），则应使用特殊字符串“（\_）”指定HTTP URL中的标签，以避免与其他URL路径模糊。例如，如果标签是foo/bar，则替换斜杠将导致类似于foo(\_)bar的标签。如果您使用像curl这样的命令行客户端（例如使用引号将其从shell中转出来），请小心URL中的方括号。

Git URI中的占位符

Spring Cloud Config服务器支持一个Git仓库URL，其中包含{application}和{profile}（以及{label}）的占位符，如果需要，请记住，标签应用为git标签）。因此，您可以使用（例如）轻松支持“每个应用程序的一个repo”策略：

spring:

cloud:

config:

server:

git:

uri: https://github.com/myorg/{application}

或使用类似模式“{一个回报每个配置文件”策略，但使用{profile}。

模式匹配和多个存储库

还可以通过应用程序和配置文件名称的模式匹配来支持更复杂的需求。模式格式是带有通配符的{application}/{profile}名称的逗号分隔列表（其中以通配符开头的模式可能需要引用）。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

simple: https://github.com/simple/config-repo

special:

pattern: special\*/dev\*,\*special\*/dev\*

uri: https://github.com/special/config-repo

local:

pattern: local\*

uri: file:/home/configsvc/config-repo

如果{application}/{profile}不匹配任何模式，它将使用在“spring.cloud.config.server.git.uri”下定义的默认uri。在上面的例子中，对于“简单”存储库，模式是simple/\*（即所有配置文件中只匹配一个名为“简单”的应用程序）。“本地”存储库与所有配置文件中以“local”开头的所有应用程序名称匹配（将/\*后缀自动添加到任何没有配置文件匹配器的模式）。

注意

在上述“简单”示例中使用的“单行”快捷方式只能在唯一要设置的属性为URI的情况下使用。如果您需要设置其他任何内容（凭据，模式等），则需要使用完整的表单。

repo中的pattern属性实际上是一个数组，因此您可以使用属性文件中的YAML数组（或[0]，[1]等后缀）绑定到多个模式。如果要运行具有多个配置文件的应用程序，则可能需要执行此操作。例：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

repos:

development:

pattern:

- \*/development

- \*/staging

uri: https://github.com/development/config-repo

staging:

pattern:

- \*/qa

- \*/production

uri: https://github.com/staging/config-repo

注意

Spring Cloud将猜测包含不在\*中的配置文件的模式意味着您实际上希望匹配从该模式开始的配置文件列表（所以\*/staging是["\*/staging", "\*/staging,\*"]的快捷方式） 。这是常见的，您需要在本地的“开发”配置文件中运行应用程序，但也可以远程运行“云”配置文件。

每个存储库还可以选择将配置文件存储在子目录中，搜索这些目录的模式可以指定为searchPaths。例如在顶层：

spring:

cloud:

config:

server:

git:

uri: https://github.com/spring-cloud-samples/config-repo

searchPaths: foo,bar\*

在此示例中，服务器搜索顶级和“foo /”子目录以及名称以“bar”开头的任何子目录中的配置文件。

默认情况下，首次请求配置时，服务器克隆远程存储库。服务器可以配置为在启动时克隆存储库。例如在顶层：

spring:

cloud:

config:

server:

git:

uri: https://git/common/config-repo.git

repos:

team-a:

pattern: team-a-\*

cloneOnStart: true

uri: http://git/team-a/config-repo.git

team-b:

pattern: team-b-\*

cloneOnStart: false

uri: http://git/team-b/config-repo.git

team-c:

pattern: team-c-\*

uri: http://git/team-a/config-repo.git

在此示例中，服务器在启动之前克隆了team-a的config-repo，然后它接受任何请求。所有其他存储库将不被克隆，直到请求从存储库配置。

### 6.3 配置中心客户端

对于spring应用的一些特点：

* 绑定配置中心，初始化spring 环境通过远程属性源
* 对属性值进行加解密
* 通过@RefreshScope重新初始化@Beans对象配置
* 一些管理端点：
* /env更新Environment并重新绑定@ConfigurationProperties属性和日志级别
* /refresh刷新@RefreshScope属性
* /restart重启spring 上下文默认不可用
* /pause、/resume通过调用stop，start方法应用上下文生命周期

#### 6.3.1 URI 指定配置中心

在配置spring cloud config的客户端启动的时候，会加载配置信息，只有配置了spring.cloud.config.uri时，客户端才会连接到config服务端来远程获取配置并初始化spring环境配置，在bootstrap.yml配置如下：

spring:

cloud:

config:

label: master

profile: dev

uri: http://localhost:7001

#### 6.3.2 服务化配置中心

高可用的服务化配置中心，通过服务来发现配置中心配置如下：

spring:

cloud:

config:

label: master

profile: dev

discovery:

enabled: true

service-id: CONF-SRV

# uri: http://localhost:7001

eureka:

client:

service-url:

defaultZone: http://localhost:8082/eureka/,http://localhost:8083/eureka/

#### 6.3.2 动态刷新配置

通过以上小节，可以发现config也是可以实现自动实时动态更新的。

在客户端引入spring-boot-starter-actuator监控模块，其中有对/refresh端点的实现，可以使用该端点来重新刷新和获取最新配置。

通过POST请求发送curl -d "" <http://lenovo-pc:9066/refresh可以看见变化内容返回>

通过该功能我们可以使用GIT仓库的web hook功能来实现关联应用，当git提交变化时，会给响应应用发送/refresh请求实现配置的实时更新。

除了通过git来实现应用配置的时候刷新之外，也可以借助spring cloud bus消息总线来实现配置变更通知。

#### 6.3.3 失败快速响应与重试

Spring 应该客户端可能会加载很多预加载信息，而有时候希望先知道config配置中心的对当前应用的配置情况又不希望等待过久，要客户端优先判断配置中心是否正常并快速响应只需要在bootstrap.yml配置参数中配置

spring:

cloud:

config:

fail-fast: true

有了这样的参数就能保证客户端先判断配置中心是否正常，为了让直接启动失败降低，需要加入重试机制，增加如下参数：

retry:

initial-interval: 1000 初始重试间隔时间单位毫秒

max-attempts: 6 最大重试次数默认6次

multiplier: 1000 下一间隔的乘数默认1.1

max-interval: 1000 最大间隔时间单位毫秒

并引入spring-retry和spring-boot-starter-aop依赖：

特别说明：关于嵌入config server到应用这里不推荐使用，对它特殊参数配置这里不做说明，可以查看类ConfigServerBootstrapConfiguration、ConfigServerBootstrapApplicationListener了解详情。

## 7 消息总线 Spring Cloud Bus

Bus可以将分布式的节点用轻量的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。这里主要是用Spring Cloud Bus实现通知微服务架构应用下的配置文件的实时动态更新spring-cloud-starter-bus-amqp或spring-cloud-starter-bus-kafk，这里主要以KAFKA为消息中间进行说明。

对6长节奏实例进行扩展

对config-client依赖新增：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

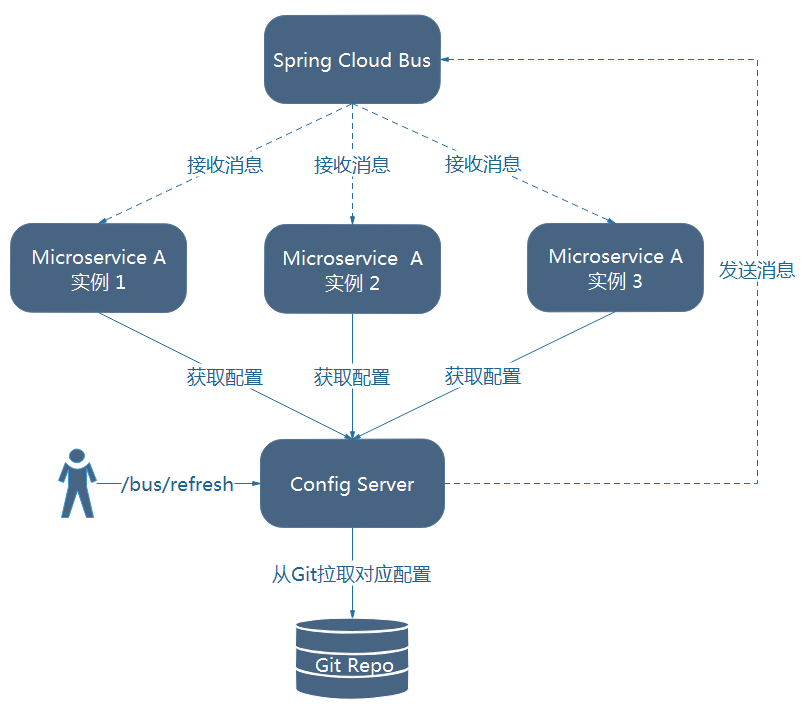
<dependency>

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

<artifactId>spring-cloud-starter-bus-kafka</artifactId>

</dependency>

Spring cloud bus 与spring cloud config整合原理流程：



微服务可以通过消息总线连接到一起，这个时候，git发生改变时，向任意节点发送/bus/refresh都可以让消息总线上的应用都会接收更新请求，当然我们这里一般都是通过config server配置中心来刷新，除了刷新所有节点服务外，也能够指定刷新范围：

1. /bus/refresh?destination=<servicename>:\*\*该请求会触发servicename服务的所有实例进行刷新
2. /bus/refresh?destination=<appinstanceName>:<port>该请求会刷新指定应用实例名，应用命名命名规则安装eureka中提到的命名生成规则进行：

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id}:${server:port}

如：

curl -d "" post <http://localhost:7002/bus/refresh?destination=conf-demo:7001>

curl -d "" post http://localhost:7002/bus/refresh?destination=conf-demo:\*\*

1. 消息代理简要设置

stream:

kafka:

binder:

zkNodes: 192.168.137.4:2181

brokers: 192.168.137.4:9092

## 8参数实例

|  |  |  |
| --- | --- | --- |
| 序号 | 组件 | 实例module |
| 1 | 服务注册中心 | eureka-server |
| 2 | 服务注册、发现 | eureka-client |
| 3 | 服务消费ribbon实例 | ribbon-consumer |
| 4 | 服务消费Feign实例 | feign |
| 5 | 服务熔断监控单实例 | hystrix |
| 6 | 服务熔断监控集群实例 | tunrbine |
| 7 | 服务网关 | zuul |
| 8 | 分布式配置中心 | Config-server |
| 9 | 消息总线 | Config-client |