# 微服务专题-一站式微服务架构SpringCloud

主讲：Cat 老师

**北京动力节点教育科技有限公司**  
**2019 • 北京**

**动力节点版权所有，禁止私自传播**

# 快速回顾

1. 分布式与微服务架构的理论梳理；
2. 什么是Spring Cloud？
3. Spring Cloud的整体架构 （Dubbo）
4. 服务消费者Controller直连调用服务提供者Controller
5. Spring Cloud的注册中心Eureka
6. Spring Cloud Eureka与Zookeeper比较
7. Spring Cloud Eureka高可用集群
8. Spring Cloud Eureka自我保护机制
9. Spring Cloud Ribbon负载均衡
10. Spring Cloud Feign声明式服务调用

# Spring Cloud Hystrix

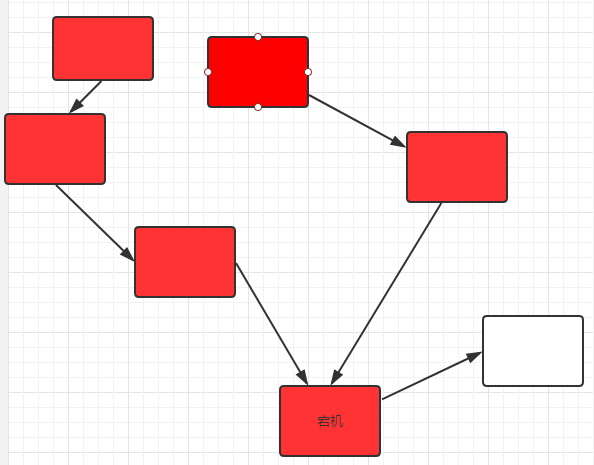
## Hystrix 是什么

Hystrix被称为熔断器，它是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多服务之间通过远程调用实现信息交互，调用时不可避免会出现调用失败，比如超时、异常等原因导致调用失败，Hystrix能够保证在一个服务出问题的情况下，不会导致整体服务失败，避免级联故障（服务雪崩），以提高分布式系统的弹性；

熔断器也有叫断路器，他们表示同一个意思，最早来源于微服务之父 Martin

Fowler 的论文 CircuitBreaker 一文。“熔断器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，能够及时切断故障电路，防止发生过载、发热甚至起火等严重后果。

所以当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。



比如：

比如电商中的用户下订单，我们有两个服务，一个下订单服务，一个减库存服务，

当用户下订单时调用下订单服务，然后下订单服务又调用减库存服务，如果减库

存服务响应延迟或者没有响应，则会造成下订单服务的线程挂起等待，如果大量

的用户请求下订单，或导致大量的请求堆积，引起下订单服务也不可用，如果还

有另外一个服务依赖于订单服务，比如用户服务，它需要查询用户订单，那么用

户服务查询订单也会引起大量的延迟和请求堆积，导致用户服务也不可用。

所以在微服务架构中，很容易造成服务故障的蔓延，引发整个微服务系统瘫痪不

可用。

Spring Cloud Hystrix 实现了熔断器、线程隔离等一系列服务保护功能。该功能也是基于 Netflix 的开源框架 Hystrix 实现的，该框架的目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。 (Spring Cloud Hystrix 对 Netflix Hystrix 做了一个starter)

程序BUG，数据不匹配，响应时间过长，服务不可用 等等.....都可能导致服务雪崩；

针对上面的问题，hystrix提供了 :

熔断降级

请求限流

服务降级是指当某个微服务响应时间过长，发生异常，或者服务不可用了，我们不能把错误信息返回回来，或者让它一直卡在那里，所以要准备一个对应的策略（一个方法），当发生这种问题时，我们直接调用这个备用的方法来快速返回一个默认的结果，让请求得到快速响应，而不是一直卡在那里；

操作步骤

在 Spring Cloud 中使用熔断器 Hystrix 是非常简单和方便的，只需要简单两步即可：

1、加依赖

*<!-- spring-cloud-starter-netflix-hystrix -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**>  
</**dependency**>

2、在入口类中使用@EnableCircuitBreaker 注解或 @EnableHystrix

开启断路器功能，也可以使用一个名为@SpringCloudApplication 的注解代替主类上的三个注解；

3、在调用远程服务的方法上添加注解：

@HystrixCommand(fallbackMethod = **"fallback"**)

hystrix 默认超时时间是 1000 毫秒，如果你后端的响应超过此时间，就会触发

断路器；

修改 hystrix 的默认超时时间：

@RequestMapping(**"/cloud/goodsHystrix"**)  
@HystrixCommand(fallbackMethod = **"fallback"**,  
 commandProperties={  
 @HystrixProperty(name=**"execution.timeout.enabled"**, value=**"true"**),  
 @HystrixProperty(name=**"execution.isolation.thread.timeoutInMilliseconds"**, value=**"5000"**)  
 })  
**public** ResultObject goodsHystrix()

或者在配置文件进行配置：

**ribbon.ReadTimeout**=**6000  
ribbon.ConnectTimeout**=**3000  
hystrix.command.default.execution.timeout.enabled**=**true  
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds**=**5000**

这里有个坑需要注意一下：

如果hystrix.command.default.execution.timeout.enabled为true,则会有两个执行方法超时的配置,一个就是ribbon的ReadTimeout,一个就是熔断器hystrix的timeoutInMilliseconds, 此时谁的值小谁生效；

如果hystrix.command.default.execution.timeout.enabled为false,则熔断器不进行超时熔断,而是根据ribbon的ReadTimeout抛出的异常而熔断,也就是取决于ribbon的ConnectTimeout,配置的是请求服务的超时时间,除非服务找不到,或者网络原因,这个时间才会生效；

**ribbon.ReadTimeout**=**6000  
ribbon.ConnectTimeout**=**3000**

## Hystrix的异常处理

我们在调用服务提供者时，服务提供者可能抛出异常，我们自己也可能抛异常，默认情况下方法抛了异常会自动进行服务降级，交给服务降级中的方法去处理；

当我们自己发生异常后，只需要在服务降级方法中添加一个 Throwable 类型的

参数就能够获取到抛出的异常的类型，如下：

**public** ResultObject fallback(Throwable throwable) {  
 System.***out***.println(throwable.getMessage());  
 **return new** ResultObject(Constant.***ONE***,**"服务降级"**);  
}

当然远程服务发生了异常也可以获取到异常信息；

如果远程服务有一个异常抛出后我们不希望进入到服务降级方法中去处理，而是

直接将异常抛给用户，那么我们可以在@HystrixCommand 注解中添加忽略异

常，如下：

@HystrixCommand(fallbackMethod = **"fallback"**, ignoreExceptions=Throwable.**class**,

降级是作用？

1. 可以监听你的请求有没有超时；（默认是1秒，时间可以改）
2. 异常或报错了可以快速让请求返回，不会一直等待；（避免线程累积）
3. 当的系统马上迎来大量的并发(双十一秒杀这种或者促销活动) 此时如果系统承载不了这么大的并发时，可以考虑先关闭一些不重要的微服务（在降级方法中返回一个比较友好的信息)，把资源让给核心微服务，待高峰流量过去，再开启回来。

## Hystrix限流

限流有很多方案：

1. Nginx
2. Redis + Lua
3. Sentinel
4. 基于限流算法自己实现（令牌桶、漏桶算法）

### hystrix限流就是限制你某个微服务的使用量（可用线程数、信号量）

hystrix通过线程池的方式来管理微服务的调用，它默认是一个线程池（大小10个） 管理你的所有微服务，你可以给某个微服务开辟新的线程池：

@RequestMapping(**"/cloud/goodsHystrix2"**)  
@HystrixCommand(fallbackMethod = **"fallback"**,  
 threadPoolKey = **"goods"**,  
 threadPoolProperties = {@HystrixProperty(name = **"coreSize"**, value = **"2"**),  
 @HystrixProperty(name = **"maxQueueSize"**, value = **"1"**)})  
**public** ResultObject goodsHystrix2() **throws** InterruptedException {

threadPoolKey 是线程池唯一标识， hystrix 会使用该标识来计数，看线程占用是否超过了， 超过了就会直接降级该次调用；

这里coreSize给他值为2 那么假设你这个方法调用时间是1s执行完， 那么在1s内如果有超过2个请求进来的话，剩下的请求则全部降级；

其中maxQueueSize是一个线程队列，里面只能放一个请求线程，本来线程数有2个，队列里面允许放一个，那么总共只能有3个请求线程执行，如果超过了就会限流；

## feign整合hystrix

feign 默认是支持hystrix的， 但是在Spring cloud Dalston 版本之后就默认关闭了， 因为业务需求不一定要使用；

所以现在要使用首先得打开他，在yml文件加上如下配置:

**feign.hystrix.enabled**=**true**

加上配置之后降级方法怎么写呢？

@FeignClient(value=**"34-SPRINGCLOUD-SERVICE-GOODS"**, fallback = GoodsRemoteClientFallBack.**class**)  
**public interface** GoodsRemoteClient {  
  
 */\*\*  
 \* 声明一个feign的接口，它的实现是服务提供者的controller实现  
 \*  
 \** ***@return*** *\*/* @RequestMapping(**"/service/goods"**)  
 **public** ResultObject goods();  
}

在feign客户端的注解上 有个属性叫fallback 然后指向一个类

GoodsRemoteClientFallBack类：

@Component  
**public class** GoodsRemoteClientFallBack **implements** GoodsRemoteClient {  
  
 @Override  
 **public** ResultObject goods() {  
 **return new** ResultObject(Constant.***ONE***,**"feign服务调用降级"**);  
 }  
}

如此方法降级便可以了；

当然如果需要拿到具体的服务错误信息，那么可以这样:

客户端指定一个fallbackFactory即可；

@FeignClient(value=**"34-SPRINGCLOUD-SERVICE-GOODS"**, fallbackFactory = GoodsRemoteClientFallBackFactory.**class**)  
**public interface** GoodsRemoteClient {  
  
 */\*\*  
 \* 声明一个feign的接口，它的实现是服务提供者的controller实现  
 \*  
 \** ***@return*** *\*/* @RequestMapping(**"/service/goods"**)  
 **public** ResultObject goods();  
}

@Component  
**public class** GoodsRemoteClientFallBackFactory **implements** FallbackFactory<GoodsRemoteClient> {  
  
 @Override  
 **public** GoodsRemoteClient create(Throwable throwable) {  
 **return new** GoodsRemoteClient() {  
 @Override  
 **public** ResultObject goods() {  
 String message = throwable.getMessage();  
 System.***out***.println(**"feign远程调用异常："** + message);  
 **return new** ResultObject();  
 }  
 };  
 }  
}

这个message 就是错误信息，至此，就完成了feign与hystrix的整合；

Spring Cloud Feign超时时间设置

Feign调用服务的默认时长是1秒钟，也就是如果超过1秒没连接上或者超过1秒没响应，那么会相应的报错。而实际情况是因为业务的不同可能出现超出1秒的情况，这时我们需要调整超时时间。

Feign 的负载均衡底层用的就是 **Ribbon**  
在application.properties中添加如下配置，超过5秒没连接上报连接超时，如果超过5秒没有响应，报请求超时；

*# 参考RibbonClientConfiguration  
# 请求连接的超时时间 默认的时间为 1 秒***ribbon.ConnectTimeout**=**5000***# 请求处理的超时时间***ribbon.ReadTimeout**=**5000**

ribbon还有MaxAutoRetries对当前实例的重试次数,

MaxAutoRetriesNextServer对切换实例的重试次数,

如果ribbon的ReadTimeout超时,或者ConnectTimeout连接超时,会进行重试操作

由于ribbon的重试机制,通常熔断hystrix的超时时间需要配置的比ReadTimeout长,ReadTimeout比ConnectTimeout长,否则还未重试,就熔断了；

为了确保重试机制的正常运作,理论上（以实际情况为准）建议hystrix的超时时间为:(1 + MaxAutoRetries + MaxAutoRetriesNextServer) \* ReadTimeout

## hystrix相关配置:

Execution相关的属性的配置  
hystrix.command.default.execution.isolation.strategy 隔离策略，默认是Thread, 可选Thread｜Semaphore（信号量）  
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 命令执行超时时间，默认1000ms  
hystrix.command.default.execution.timeout.enabled 执行是否启用超时，默认启用true  
hystrix.command.default.execution.isolation.thread.interruptOnTimeout 发生超时是是否中断，默认true  
hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests 最大并发请求数，默认10，该参数当使用ExecutionIsolationStrategy.SEMAPHORE策略时才有效。如果达到最大并发请求数，请求会被拒绝。理论上选择semaphore size的原则和选择thread size一致，但选用semaphore时每次执行的单元要比较小且执行速度快（ms级别），否则的话应该用thread。semaphore应该占整个容器（tomcat）的线程池的一小部分。 Fallback相关的属性 这些参数可以应用于Hystrix的THREAD和SEMAPHORE策略；  
hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequest如果并发数达到该设置值，请求会被拒绝和抛出异常并且fallback不会被调用。默认10  
hystrix.command.default.fallback.enabled 当执行失败或者请求被拒绝，是否会尝试调用hystrixCommand.getFallback() 。默认true  
Circuit Breaker相关的属性   
hystrix.command.default.circuitBreaker.enabled 用来跟踪circuit的健康性，如果未达标则让request短路。默认true  
hystrix.command.default.circuitBreaker.requestVolumeThreshold 一个rolling window内最小的请求数。如果设为20，那么当一个rolling window的时间内（比如说1个rolling window是10秒）收到19个请求， 即使19个请求都失败，也不会触发circuit break。默认20  
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds 触发短路的时间值，当该值设为5000时，则当触发circuit break后的5000毫秒内都会拒绝request，也就是5000毫秒后才会关闭circuit。 默认5000  
hystrix.command.default.circuitBreaker.errorThresholdPercentage错误比率阀值，如果错误率>=该 值，circuit会被打开，并短路所有请求触发fallback。默认50  
hystrix.command.default.circuitBreaker.forceOpen 强制打开熔断器，如果打开这个开关，那么拒绝所 有request，默认false  
hystrix.command.default.circuitBreaker.forceClosed 强制关闭熔断器 如果这个开关打开，circuit将 一直关闭且忽略circuitBreaker.errorThresholdPercentage  
  
Metrics相关参数  
hystrix.command.default.metrics.rollingStats.timeInMilliseconds 设置统计的时间窗口值的，毫秒值，circuit break 的打开会根据1个rolling window的统计来计算。若rolling window被设为10000毫秒， 则rolling window会被分成n个buckets，每个bucket包含success，failure，timeout，rejection的次数的统计信息。默认10000  
  
hystrix.command.default.metrics.rollingStats.numBuckets 设置一个rolling window被划分的数 量，若numBuckets＝10，rolling window＝10000，那么一个bucket的时间即1秒。必须符合rolling window  % numberBuckets == 0。默认10  
hystrix.command.default.metrics.rollingPercentile.enabled 执行时是否enable指标的计算和跟踪， 默认true  
hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds 设置rolling  percentile window的时间，默认60000  
hystrix.command.default.metrics.rollingPercentile.numBuckets 设置rolling percentile  window的numberBuckets。逻辑同上。默认6  
hystrix.command.default.metrics.rollingPercentile.bucketSize 如果bucket size＝100，window ＝10s，若这10s里有500次执行，只有最后100次执行会被统计到bucket里去。增加该值会增加内存开销以及排序 的开销。默认100  
hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds 记录health 快照（用 来统计成功和错误绿）的间隔，默认500ms  
Request Context 相关参数  
hystrix.command.default.requestCache.enabled 默认true，需要重载getCacheKey()，返回null时不 缓存  
 hystrix.command.default.requestLog.enabled 记录日志到HystrixRequestLog，默认true  
 Collapser Properties 相关参数  
 hystrix.collapser.default.maxRequestsInBatch 单次批处理的最大请求数，达到该数量触发批处理，默认 Integer.MAX\_VALU  
hystrix.collapser.default.timerDelayInMilliseconds 触发批处理的延迟，也可以为创建批处理的时间 ＋该值，默认10  
hystrix.collapser.default.requestCache.enabled 是否对HystrixCollapser.execute() and  HystrixCollapser.queue()的cache，默认true  
ThreadPool 相关参数  
线程数默认值10适用于大部分情况（有时可以设置得更小），如果需要设置得更大，那有个基本得公式可以 follow： requests per second at peak when healthy × 99th percentile latency in seconds + some  breathing room 每秒最大支撑的请求数 (99%平均响应时间 + 缓存值) 比如：每秒能处理1000个请求，99%的请求响应时间是60ms，那么公式是： 1000 （0.060+0.012）基本得原则时保持线程池尽可能小，他主要是为了释放压力，防止资源被阻塞。 当一切都是正常的时候，线程池一般仅会有1到2个线程激活来提供服务  
hystrix.threadpool.default.coreSize 并发执行的最大线程数，默认10  
hystrix.threadpool.default.maxQueueSize BlockingQueue的最大队列数，当设为－1，会使用  
SynchronousQueue，值为正时使用LinkedBlcokingQueue。该设置只会在初始化时有效，之后不能修改threadpool的queue size，除非reinitialising thread executor。默认－1。  
hystrix.threadpool.default.queueSizeRejectionThreshold 即使maxQueueSize没有达到，达到 queueSizeRejectionThreshold该值后，请求也会被拒绝。因为maxQueueSize不能被动态修改，这个参数将允 许我们动态设置该值。if maxQueueSize == ­1，该字段将不起作用 hystrix.threadpool.default.keepAliveTimeMinutes 如果corePoolSize和maxPoolSize设成一样（默认 实现）该设置无效。如果通过plugin（https://github.com/Netflix/Hystrix/wiki/Plugins）使用自定义 实现，该设置才有用，默认1.  
hystrix.threadpool.default.metrics.rollingStats.timeInMilliseconds 线程池统计指标的时间，默 认10000  
hystrix.threadpool.default.metrics.rollingStats.numBuckets 将rolling window划分为n个 buckets，默认10；

## **Hystrix 仪表盘监控**

Hystrix 仪表盘（Hystrix Dashboard），就像汽车的仪表盘实时显示汽车的各

项数据一样，Hystrix 仪表盘主要用来监控 Hystrix 的实时运行状态，通过它我

们可以看到 Hystrix 的各项指标信息，从而快速发现系统中存在的问题进而解决它；

要使用 Hystrix 仪表盘功能，我们首先需要有一个 Hystrix Dashboard项目，这个功能我们可以在原来的消费者应用上添加，让原来的消费者应用具备 Hystrix 仪表盘功能，但一般地，微服务架构思想是推崇服务的拆分，Hystrix Dashboard 也是一个服务，所以通常会单独创建一个新的工程专门用做 Hystrix Dashboard 服务；

**搭建一个 Hystrix Dashboard 服务的步骤：**

**第一步：**创建一个普通的 Spring Boot 工程

比如创建一个名为 springcloud-hystrix-dashboard 的 Spring Boot 工程，建立好基本的结构和配置；

**第二步：**添加相关依赖

在创建好的 Spring Boot 项目的 pom.xml 文件中添加相关依赖，如下：

**过时了：**

<**dependency**>

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

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

<**version**>1.4.5.RELEASE</**version**>

</**dependency**>

新的依赖：

*<!-- spring-cloud-starter-netflix-hystrix-dashboard -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-hystrix-dashboard</**artifactId**>  
</**dependency**>

**第三步：**入口类上添加注解

添加好依赖之后，在入口类上添加@EnableHystrixDashboard 注解开启仪表盘功能，如下：

@SpringBootApplication

@EnableHystrixDashboard

**public class** Application {

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

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

}

}

**第四步：**属性配置

最后，我们可以根据个人习惯配置一下 application.properties 文件，如下：

server.port=3721

至此，我们的 Hystrix 监控环境就搭建好了；



Hystrix 仪表盘工程已经创建好了，现在我们需要有一个服务，让这个服务提供

一个路径为/actuator/hystrix.stream 接口，然后就可以使用 Hystrix 仪表盘来对该服务进行监控了；

我们改造消费者服务，让其能提供/actuator/hystrix.stream 接口，步骤如下：

1. 消费者项目需要有 hystrix 的依赖：

过时的

*<!--Spring Cloud 熔断器起步依赖-->*

<**dependency**>

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

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

<**version**>1.4.5.RELEASE</**version**>

</**dependency**>

新的：

*<!-- spring-cloud-starter-netflix-hystrix -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**>  
</**dependency**>

2、需要有一个 spring boot 的服务监控依赖：

<**dependency**>

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

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

</**dependency**>

3、配置文件需要配置 spring boot 监控端点的访问权限：

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

这个是用来暴露 endpoints 的，由于 **endpoints** 中会包含很多敏感信息，除

了 health 和 info 两个支持直接访问外，其他的默认不能直接访问，所以我们

让它都能访问，或者指定：

**management.endpoints.web.exposure.include**=**hystrix.stream**

4、访问入口 http://localhost:8081/actuator/hystrix.stream

**注意**：这里有一个细节需要注意，要访问/hystrix.stream 接口，首先得访问

consumer 工程中的任意一个其他接口，否则直接访问/hystrix.stream 接口时

会输出出一连串的 ping: ping: …，先访问 consumer 中的任意一个其他接口，

然后再访问/hystrix.stream 接口即可；

## Hystrix 仪表盘监控数据解读

