

Spring Cloud

- 1 微服务是什么
- 2 Spring Cloud是什么
- 3 Spring Cloud Eureka
- 4 Spring Cloud Ribbon
- 5 Spring Cloud OpenFeign
- 6 **Spring Cloud Hystrix**
- 7 Spring Cloud Gateway
- 8 Spring Cloud Config
- 9 Spring Cloud Alibaba是什么
- 10 Spring Cloud Alibaba Nacos
- 11 Spring Cloud Alibaba Sentinel
- 12 Spring Cloud Alibaba Seata

首页 > Spring Cloud

Hystrix: Spring Cloud服务熔断与降级组件 (非常详细)

< 上一节

下一节 >

在微服务架构中，一个应用往往由多个服务组成，这些服务之间相互依赖，依赖关系错综复杂。

例如一个微服务系统中存在 A、B、C、D、E、F 等多个服务，它们的依赖关系如下图。

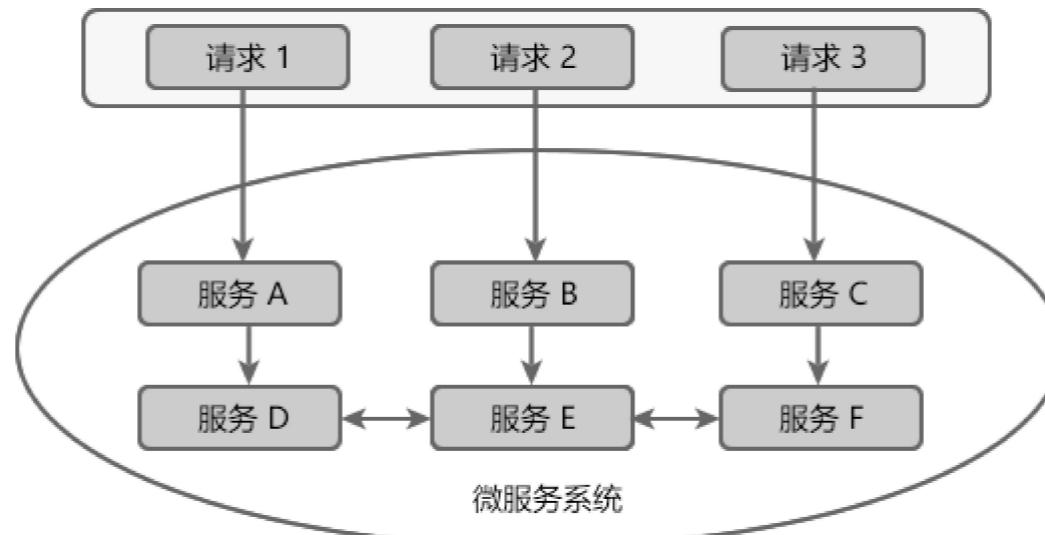


图1：服务依赖关系

通常情况下，一个用户请求往往需要多个服务配合才能完成。如图 1 所示，在所有服务都处于可用状态时，请求 1 需要调用 A、D、E、F 四个服务才能完成，请求 2 需要调用 B、E、D 三个服务才能完成，请求 3 需要调用服务 C、F、E、D 四个服务才能完成。

当服务 E 发生故障或网络延迟时，会出现以下情况：

1. 即使其他所有服务都可用，由于服务 E 的不可用，那么用户请求 1、2、3 都会处于阻塞状态，等待服务 E 的响应。在高并发的场景下，会导致整个服务器的线程资源在短时间内迅速消耗殆尽。
2. 所有依赖于服务 E 的其他服务，例如服务 B、D 以及 F 也都会处于线程阻塞状态，等待服务 E 的响应，导致这些服务的不可用。
3. 所有依赖服务 B、D 和 F 的服务，例如服务 A 和服务 C 也会处于线程阻塞状态，以等待服务 D 和服务 F 的响应，导致服务 A 和服务 C 也不可用。

从以上过程可以看出，当微服务系统的一个服务出现故障时，故障会沿着服务的调用链路在系统中疯狂蔓延，最终导致整个微服务系统的瘫痪，这就是“雪崩效应”。为了防止此类事件的发生，微服务架构引入了“熔断器”的一系列服务容错和保护机制。

熔断器

熔断器 (Circuit Breaker) 一词来源物理学中的电路知识，它的作用是当线路出现故障时，迅速切断电源以保护电路的安全。

在微服务领域，熔断器最早是由 Martin Fowler 在他发表的《Circuit Breaker》一文中提出。与物理学中的熔断器作用相似，微服务架构中的熔断器能够在某个服务发生故障后，向服务调用方返回一个符合预期的、可处理的降级响应 (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、不必要地占用，避免故障在微服务系统中的蔓延，防止系统雪崩效应的发生。



Spring Cloud Hystrix

Spring Cloud Hystrix 是一款优秀的服务容错与保护组件，也是 Spring Cloud 中最重要的组件之一。

Spring Cloud Hystrix 是基于 Netflix 公司的开源组件 Hystrix 实现的，它提供了熔断器功能，能够有效地阻止分布式微服务系统中出现联动故障，以提高微服务系统的弹性。Spring Cloud Hystrix 具有服务降级、服务熔断、线程隔离、请求缓存、请求合并以及实时故障监控等强大功能。

Hystrix [hɪst'riks]，中文含义是豪猪，豪猪的背上长满了棘刺，使它拥有了强大的自我保护能力。而 Spring Cloud Hystrix 作为一个服务容错与保护组件，也可以让服务拥有自我保护的能力，因此也有人将其戏称为“豪猪哥”。

在微服务系统中，Hystrix 能够帮助我们实现以下目标：

- **保护线程资源**：防止单个服务的故障耗尽系统中的所有线程资源。
- **快速失败机制**：当某个服务发生了故障，不让服务调用方一直等待，而是直接返回请求失败。
- **提供降级（FallBack）方案**：在请求失败后，提供一个设计好的降级方案，通常是一个兜底方法，当请求失败后即调用该方法。
- **防止故障扩散**：使用熔断机制，防止故障扩散到其他服务。
- **监控功能**：提供熔断器故障监控组件 Hystrix Dashboard，随时监控熔断器的状态。

Hystrix 服务降级

Hystrix 提供了服务降级功能，能够保证当前服务不受其他服务故障的影响，提高服务的健壮性。

服务降级的使用场景有以下 2 种：

- 在服务器压力剧增时，根据实际业务情况及流量，对一些不重要、不紧急的服务进行有策略地不处理或简单处理，从而释放服务器资源以保证核心服务正常运作。
- 当某些服务不可用时，为了避免长时间等待造成服务卡顿或雪崩效应，而主动执行备用的降级逻辑立刻返回一个友好的提示，以保障主体业务不受影响。

我们可以通过重写 HystrixCommand 的 getFallback() 方法或 HystrixObservableCommand 的 resumeWithFallback() 方法，使服务支持服务降级。

Hystrix 服务降级 FallBack 既可以放在服务端进行，也可以放在客户端进行。

Hystrix 会在以下场景下进行服务降级处理：

- 程序运行异常
- 服务超时
- 熔断器处于打开状态
- 线程池资源耗尽

示例1

下面我们就通过一个案例，分别演示下 Hystrix 服务端服务降级和客户端服务降级。

服务端服务降级

1. 在主工程 spring-cloud-demo2 下创建一个名为 micro-service-cloud-provider-dept-hystrix-8004 的服务提供者，并在其 pom.xml 中添加以下依赖。

```
01. <?xml version="1.0" encoding="UTF-8"?>
```



```
02. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
04.   <modelVersion>4.0.0</modelVersion>
05.   <!--父pom-->
06.   <parent>
07.     <artifactId>spring-cloud-demo2</artifactId>
08.     <groupId>net.biancheng.c</groupId>
09.     <version>0.0.1-SNAPSHOT</version>
10.   </parent>
11.
12.   <groupId>net.biancheng.c</groupId>
13.   <artifactId>micro-service-cloud-provider-dept-hystrix-8004</artifactId>
14.   <version>0.0.1-SNAPSHOT</version>
15.   <name>micro-service-cloud-provider-dept-hystrix-8004</name>
16.   <description>Demo project for Spring Boot</description>
17.   <properties>
18.     <java.version>1.8</java.version>
19.   </properties>
20.
21.   <dependencies>
22.     <dependency>
23.       <groupId>org.springframework.boot</groupId>
24.       <artifactId>spring-boot-starter-web</artifactId>
25.     </dependency>
26.     <dependency>
27.       <groupId>org.springframework.boot</groupId>
28.       <artifactId>spring-boot-devtools</artifactId>
29.       <scope>runtime</scope>
30.       <optional>true</optional>
31.     </dependency>
32.     <dependency>
33.       <groupId>org.projectlombok</groupId>
34.       <artifactId>lombok</artifactId>
35.       <optional>true</optional>
36.     </dependency>
37.     <dependency>
38.       <groupId>org.springframework.boot</groupId>
39.       <artifactId>spring-boot-starter-test</artifactId>
40.       <scope>test</scope>
41.     </dependency>
42.     <!--添加 Spring Boot 的监控模块-->
43.     <dependency>
44.       <groupId>org.springframework.boot</groupId>
45.       <artifactId>spring-boot-starter-actuator</artifactId>
46.     </dependency>
47.     <!-- eureka 客户端-->
```



```

48.      <dependency>
49.          <groupId>org.springframework.cloud</groupId>
50.              <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
51.      </dependency>
52.      <!--hystrix 依赖-->
53.      <dependency>
54.          <groupId>org.springframework.cloud</groupId>
55.              <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
56.      </dependency>
57.  </dependencies>
58.
59.  <build>
60.      <plugins>
61.          <plugin>
62.              <groupId>org.springframework.boot</groupId>
63.              <artifactId>spring-boot-maven-plugin</artifactId>
64.              <configuration>
65.                  <excludes>
66.                      <exclude>
67.                          <groupId>org.projectlombok</groupId>
68.                          <artifactId>lombok</artifactId>
69.                      </exclude>
70.                  </excludes>
71.              </configuration>
72.          </plugin>
73.      </plugins>
74.  </build>
75. </project>

```

2. 在类路径（即 /resources 目录）下添加一个配置文件 application.yml，配置内容如下。

```

01. spring:
02.   application:
03.     name: microServiceCloudProviderDeptHystrix #微服务名称，对外暴露的微服务名称，十分重要
04.
05.   server:
06.     port: 8004
07. ###### Spring cloud 自定义服务名称和 ip 地址 #####
08.   eureka:
09.     client: #将客户端注册到 eureka 服务列表内
10.     service-url:
11.       #defaultZone: http://eureka7001:7001/eureka #这个地址是 7001注册中心在 application.yml 中暴露出来的注册地址（单机版）
12.       defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/ #
13.         将服务注册到 Eureka 集群
14.     instance:

```



```
14.     instance-id: spring-cloud-provider-8004 #自定义服务名称信息
15.     prefer-ip-address: true #显示访问路径的 ip 地址
16. #####spring cloud 使用 Spring Boot actuator 监控完善信息#####
17. # Spring Boot 2.50对 actuator 监控屏蔽了大多数的节点, 只暴露了 heath 节点, 本段配置 (*) 就是为了开启所有的节点
18. management:
19.   endpoints:
20.     web:
21.       exposure:
22.         include: "*" # * 在yaml 文件属于关键字, 所以需要加引号
23. info:
24.   app.name: micro-service-cloud-provider-dept-hystrix
25.   company.name: c.biancheng.net
26.   build.aetifactId: @project.artifactId@
27.   build.version: @project.version@
```

3. 在 net.biancheng.c.service 包下创建一个名为 DeptService 的接口, 代码如下。

```
01. package net.biancheng.c.service;
02.
03. public interface DeptService {
04.
05.     // hystrix 熔断器示例 ok
06.     public String deptInfo_Ok(Integer id);
07.
08.     //hystrix 熔断器超时案例
09.     public String deptInfo_Timeout(Integer id);
10. }
```

4. 在 net.biancheng.c.service.impl 包下, 创建 DeptService 接口的实现类 DeptServiceImpl, 代码如下。

```
01. package net.biancheng.c.service.impl;
02.
03. import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
04. import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
05. import net.biancheng.c.service.DeptService;
06. import org.springframework.stereotype.Service;
07.
08. import java.util.concurrent.TimeUnit;
09.
10. @Service("deptService")
11. public class DeptServiceImpl implements DeptService {
12.
13.     @Override
14.     public String deptInfo_Ok(Integer id) {
15.         return "线程池: " + Thread.currentThread().getName() + " deptInfo_Ok, id: " + id;
16.     }
}
```



```

17.
18. //一旦该方法失败并抛出了异常信息后，会自动调用 @HystrixCommand 注解标注的 fallbackMethod 指定的方法
19. @HystrixCommand(fallbackMethod = "dept_TimeoutHandler",
20.     commandProperties =
21.         //规定 5 秒钟以内就不报错，正常运行，超过 5 秒就报错，调用指定的方法
22.         {@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")})
23.
24. @Override
25. public String deptInfo_Timeout(Integer id) {
26.     int outTime = 6;
27.     try {
28.         TimeUnit.SECONDS.sleep(outTime);
29.     } catch (InterruptedException e) {
30.         e.printStackTrace();
31.     }
32.     return "线程池：" + Thread.currentThread().getName() + " deptInfo_Timeout, id: " + id + " 耗时：" + outTime;
33.
34. // 当服务出现故障后，调用该方法给出友好提示
35. public String dept_TimeoutHandler(Integer id) {
36.     return "C语言中文网提醒您，系统繁忙请稍后再试！" + "线程池：" + Thread.currentThread().getName() +
37.     deptInfo_Timeout, id: " + id;
38. }

```

我们可以看到 deptInfo_Timeout() 方法上使用 @HystrixCommand 注解，该注解说明如下：

- 参数 fallbackMethod 属性用于指定降级方法。
- 参数 execution.isolation.thread.timeoutInMilliseconds 用于设置自身调用超时时间的峰值，峰值内可以正常运行，否则执行降级方法

5. 在 net.biancheng.c.controller 包下创建一个名为 DeptController 的 Controller 类，代码如下。

```

01. package net.biancheng.c.controller;
02.
03. import lombok.extern.slf4j.Slf4j;
04. import net.biancheng.c.service.DeptService;
05. import org.springframework.beans.factory.annotation.Autowired;
06. import org.springframework.beans.factory.annotation.Value;
07. import org.springframework.web.bind.annotation.*;
08.
09. @RestController
10. @Slf4j
11. public class DeptController {
12.     @Autowired
13.     private DeptService deptService;
14.     @Value("${server.port}")
15.     private String serverPort;
16.

```



```

17. @RequestMapping(value = "/dept/hystrix/ok/{id}")
18. public String deptInfo_Ok(@PathVariable("id") Integer id) {
19.     String result = deptService.deptInfo_Ok(id);
20.     log.info("端口号: " + serverPort + " result:" + result);
21.     return result + ", 端口号: " + serverPort;
22. }
23.
24. // Hystrix 服务超时降级
25. @RequestMapping(value = "/dept/hystrix/timeout/{id}")
26. public String deptInfo_Timeout(@PathVariable("id") Integer id) {
27.     String result = deptService.deptInfo_Timeout(id);
28.     log.info("端口号: " + serverPort + " result:" + result);
29.     return result + ", 端口号: " + serverPort;
30. }
31.
32. }

```

6. 在 micro-service-cloud-provider-dept-hystrix-8004 的主启动类上，使用 @EnableCircuitBreaker 注解开启熔断器功能，代码如下。

```

01. package net.biancheng.c;
02. import org.springframework.boot.SpringApplication;
03. import org.springframework.boot.autoconfigure.SpringBootApplication;
04. import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
05. import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
06.
07. @SpringBootApplication
08. @EnableEurekaClient //开启 Eureka 客户端功能
09. @EnableCircuitBreaker //激活熔断器功能
10. public class MicroServiceCloudProviderDeptHystrix8004Application {
11.
12.     public static void main(String[] args) {
13.         SpringApplication.run(MicroServiceCloudProviderDeptHystrix8004Application.class, args);
14.     }
15.
16. }

```

7. 依次启动服务注册中心 (Eureka Server) 集群和 micro-service-cloud-provider-dept-hystrix-8004，并使用浏览器访问“<http://eureka7001.com:8004/dept/hystrix/ok/1>”，结果如下图。





图2: Hystrix 正常服务案例

8. 使用浏览器访问 “<http://eureka7001.com:8004/dept/hystrix/timeout/1>” , 结果如下图。



图3: Hystrix 服务端服务降级

客户端服务降级

通常情况下，我们都会在客户端进行服务降级，当客户端调用的服务端的服务不可用时，客户端直接进行服务降级处理，避免其线程被长时间、不必要地占用。

客户端服务降级步骤如下。

1. 在 micro-service-cloud-consumer-dept-feign 的 pom.xml 中添加 Hystrix 的依赖，代码如下。

```
01. <!--hystrix 依赖-->
02. <dependency>
03.   <groupId>org.springframework.cloud</groupId>
04.   <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
05. </dependency>
```

2. 在 micro-service-cloud-consumer-dept-feign 的 application.yml 中添加以下配置，开启客户端的 Hystrix 功能。

```
01. feign:
02.   hystrix:
03.     enabled: true #开启客户端 hystrix
```

3. 在 net.biancheng.c.service 包下，创建一个名为 DeptHystrixService 的服务绑定接口，与 micro-service-cloud-provider-dept-hystrix-8004 中提供的服务接口进行绑定，代码如下。



```
01. package net.biancheng.c.service;
02.
03. import org.springframework.cloud.openfeign.FeignClient;
04. import org.springframework.stereotype.Component;
05. import org.springframework.web.bind.annotation.PathVariable;
06. import org.springframework.web.bind.annotation.RequestMapping;
07.
08. @Component
09. @FeignClient(value = "MICROSERVICECLOUDPROVIDERDEPTHYSTRIX")
10. public interface DepthHystrixService {
11.     @RequestMapping(value = "/dept/hystrix/ok/{id}")
12.     public String deptInfo_Ok(@PathVariable("id") Integer id);
13.
14.     @RequestMapping(value = "/dept/hystrix/timeout/{id}")
15.     public String deptInfo_Timeout(@PathVariable("id") Integer id);
16. }
```

4. 在 net.biancheng.c.controller 包下创建一个名为 HystrixController_Consumer 的 Controller，代码如下。

```
01. package net.biancheng.c.controller;
02.
03. import com.netflix.hystrix.contrib.javanica.annotation.DefaultProperties;
04. import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
05. import lombok.extern.slf4j.Slf4j;
06. import net.biancheng.c.service.DepthHystrixService;
07. import org.springframework.web.bind.annotation.PathVariable;
08. import org.springframework.web.bind.annotation.RequestMapping;
09. import org.springframework.web.bind.annotation.RestController;
10.
11. import javax.annotation.Resource;
12.
13. @Slf4j
14. @RestController
15. public class HystrixController_Consumer {
16.
17.     @Resource
18.     private DepthHystrixService deptHystrixService;
19.
20.     @RequestMapping(value = "/consumer/dept/hystrix/ok/{id}")
21.     public String deptInfo_Ok(@PathVariable("id") Integer id) {
22.         return deptHystrixService.deptInfo_Ok(id);
23.     }
24.
25.     //在客户端进行降级
26.     @RequestMapping(value = "/consumer/dept/hystrix/timeout/{id}")
27.     @HystrixCommand(fallbackMethod = "dept_TimeoutHandler") //为该请求指定专属的回退方法
```



```

28.     public String deptInfo_Timeout(@PathVariable("id") Integer id) {
29.         String s = deptHystrixService.deptInfo_Timeout(id);
30.         log.info(s);
31.         return s;
32.     }
33.
34.     // deptInfo_Timeout方法的 专用 fallback 方法
35.     public String dept_TimeoutHandler(@PathVariable("id") Integer id) {
36.         log.info("deptInfo_Timeout 出错，服务已被降级！");
37.         return "C语言中文网提醒您：服务端系统繁忙，请稍后再试！（客户端 deptInfo_Timeout 专属的回退方法触发）";
38.     }
39. }
```

5. 在配置文件 application.yml 中添加以下配置，在客户端配置请求超时的时间。

```

01. ##### Ribbon 客户端超时控制 #####
02. ribbon:
03.     ReadTimeout: 6000 #建立连接所用的时间，适用于网络状况正常的情况下，两端两端连接所用的时间
04.     ConnectionTimeout: 6000 #建立连接后，服务器读取到可用资源的时间
05. ##### 配置请求超时时间#####
06. hystrix:
07.   command:
08.     default:
09.       execution:
10.         isolation:
11.           thread:
12.             timeoutInMilliseconds: 7000
13. ##### 配置具体方法超时时间为 3 秒#####
14.   DeptHystrixService#deptInfo_Timeout(Integer):
15.     execution:
16.       isolation:
17.         thread:
18.           timeoutInMilliseconds: 3000
```

在配置文件中设计请求的超时时间时，需要注意以下 2 点：

1) Hystrix 可以来为所有请求（方法）设置超时时间（单位为毫秒），若请求超时则触发全局的回退方法进行处理。

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=mmm
```

2) Hystrix 还可以为某个特定的服务请求（方法）设置超时时间，格式如下：

```
hystrix.command.xxx#yyy(zzz).execution.isolation.thread.timeoutInMilliseconds=mmm
```

格式说明如下：



- xxx: 为包含该服务方法的类的名称（通常为服务绑定接口的名称），例如 DeptHystrixService 接口。
- yyy: 服务方法名，例如 deptInfo_Timeout() 方法。
- zzz: 方法内的参数类型，例如 Integer、String 等等
- mmm: 要设置的超时时间，单位为毫秒（1 秒 =1000 毫秒）

6. 在 micro-service-cloud-consumer-dept-feign 的主启动类上，使用 @EnableHystrix 注解开启客户端 Hystrix 功能，代码如下。

```

01. package net.biancheng.c;
02.
03. import org.springframework.boot.SpringApplication;
04. import org.springframework.boot.autoconfigure.SpringBootApplication;
05. import org.springframework.cloud.netflix.hystrix.EnableHystrix;
06. import org.springframework.cloud.openfeign.EnableFeignClients;
07.
08. @SpringBootApplication
09. @EnableFeignClients //开启 OpenFeign 功能
10. @EnableHystrix //启用 Hystrix
11. public class MicroServiceCloudConsumerDeptFeignApplication {
12.
13.     public static void main(String[] args) {
14.         SpringApplication.run(MicroServiceCloudConsumerDeptFeignApplication.class, args);
15.     }
16. }
```

7. 修改 micro-service-cloud-provider-dept-hystrix-8004 中 DeptServiceImpl 的代码，将 deptInfo_Timeout() 方法的运行时间修改为 4 秒（小于超时时间 5 秒），以保证服务端请求正常不被降级，代码如下。

```

01. //一旦该方法失败并抛出了异常信息后，会自动调用 @HystrixCommand 注解标注的 fallbackMethod 指定的方法
02. @HystrixCommand(fallbackMethod = "dept_TimeoutHandler",
03.     commandProperties =
04.         //规定 5 秒钟以内就不报错，正常运行，超过 5 秒就报错，调用指定的方法
05.         {@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")})
06. @Override
07. public String deptInfo_Timeout(Integer id) {
08.     int outTime = 4;
09.     try {
10.         TimeUnit.SECONDS.sleep(outTime);
11.     } catch (InterruptedException e) {
12.         e.printStackTrace();
13.     }
14.     return "线程池：" + Thread.currentThread().getName() + " deptInfo_Timeout, id: " + id + " 耗时：" + outTime;
15. }
```

8. 重启 micro-service-cloud-provider-dept-hystrix-8004 和 micro-service-cloud-consumer-dept-feign，使用浏览器访问“<http://eureka7001.com:8004/dept/hystrix/timeout/1>”，直接调用服务端的 deptInfo_Timeout() 方法，结果如下图。



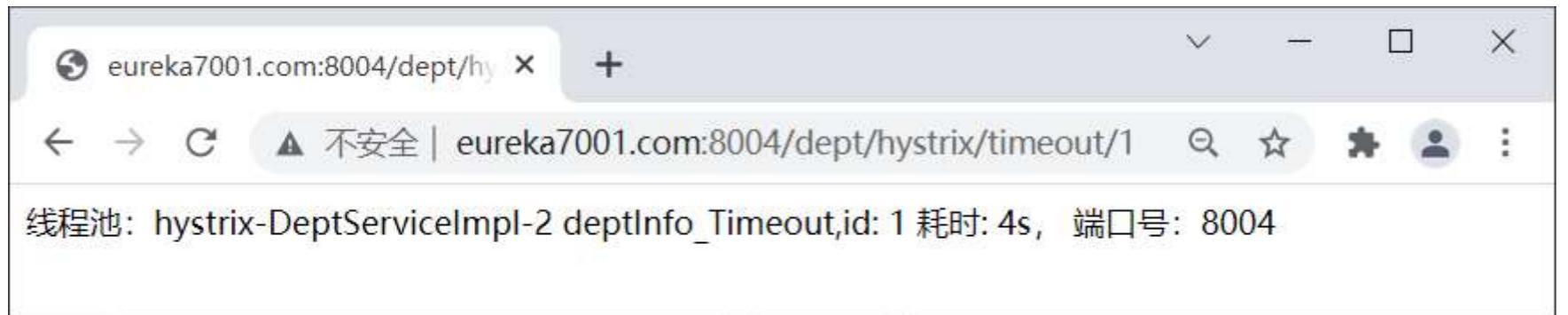


图4: Hystrix 服务端请求正常

9. 使用浏览器访问 “<http://eureka7001.com/consumer/dept/hystrix/timeout/1>” , 结果如下图。

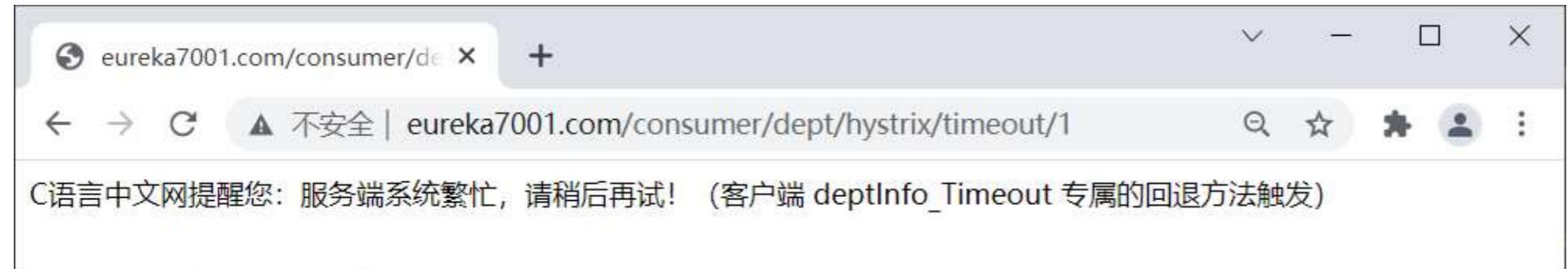


图5: Hystrix 客户端服务降级

由图 5 可以看出, 由于服务请求的耗时为 4 秒, 超过了客户端为该请求指定的超时时间 (3 秒) , 因此该服务被降级处理, 触发了其指定的回退方法。

全局降级方法

通过上面的方式实现服务降级时, 需要针对所有业务方法都配置降级方法, 这极有可能会造成代码的急剧膨胀。为了解决该问题, 我们还可以为所有业务方法指定一个全局的回退方法, 具体步骤如下。

1. 在 `HystrixController_Consumer` 的类名上标注 `@DefaultProperties` 注解, 并通过其 `defaultFallback` 属性指定一个全局的降级方法, 代码如下。

```
01. @Slf4j
02. @RestController
03. @DefaultProperties(defaultFallback = "dept_Global_FallbackMethod") //全局的服务降级方法
04. public class HystrixController_Consumer {
05. .....
06. }
```

2. 在 `HystrixController_Consumer` 中, 创建一个名为 `dept_Global_FallbackMethod` 的全局回方法, 代码如下。

```
01. /**
02. * 全局的 fallback 方法,
03. * 回退方法必须和 hystrix 的执行方法在相同类中
04. * @DefaultProperties(defaultFallback = "dept_Global_FallbackMethod") 类上注解, 请求方法上使用 @HystrixCommand 注解
05. */
06. public String dept_Global_FallbackMethod() {
```



```
07.     return "C语言中文网提醒您，运行出错或服务端系统繁忙，请稍后再试！（客户端全局回退方法触发，）";
08. }
```

注意：降级（FallBack）方法必须与其对应的业务方法在同一个类中，否则无法生效。

3. 在所有的业务方法上都标注 @HystrixCommand 注解，这里我们将 deptInfo_Timeout() 方法上的 @HystrixCommand(fallbackMethod = "dept_TimeoutHandler") 修改为 @HystrixCommand 即可，代码如下。

```
01. //在客户端进行降级
02. @RequestMapping(value = "/consumer/dept/hystrix/timeout/{id}")
03. @HystrixCommand
04. public String deptInfo_Timeout(@PathVariable("id") Integer id) {
05.     String s = deptHystrixService.deptInfo_Timeout(id);
06.     log.info(s);
07.     return s;
08. }
```

注意：全局降级方法的优先级较低，只有业务方法没有指定其降级方法时，服务降级时才会触发全局回退方法。若业务方法指定它自己的回退方法，那么在服务降级时，就只会直接触发它自己的回退方法，而非全局回退方法。

4. 重启 micro-service-cloud-consumer-dept-feign，使用浏览器访问 “<http://eureka7001.com/consumer/dept/hystrix/timeout/1>” ，结果如下图。

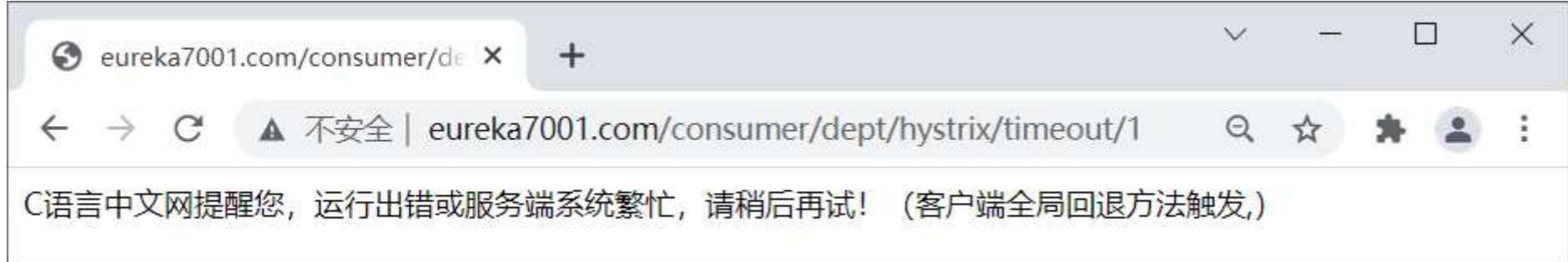


图6：全局回退方法

解耦降级逻辑

不管是业务方法指定的降级方法还是全局降级方法，它们都必须和业务方法在同一个类中才能生效，业务逻辑与降级逻辑耦合度极高。

下面我们将对业务逻辑与降级逻辑进行解耦，操作步骤如下。

1. 在 micro-service-cloud-consumer-dept-feign 的 net.biancheng.c.service 包下，新建 DeptHystrixService 接口的实现类 DeptHystrixFallbackService，统一为 DeptHystrixService 中的方法提供服务降级处理，代码如下。

```
01. package net.biancheng.c.service;
02.
03. import org.springframework.stereotype.Component;
04.
05. /**
06. * Hystrix 服务降级
```



```
07. * 解耦回退逻辑
08. */
09. @Component
10. public class DeptHystrixFallBackService implements DeptHystrixService {
11.     @Override
12.     public String deptInfo_Ok(Integer id) {
13.         return "-----C语言中文网提醒您，系统繁忙，请稍后重试！（解耦回退方法触发）-----";
14.     }
15.
16.     @Override
17.     public String deptInfo_Timeout(Integer id) {
18.         return "-----C语言中文网提醒您，系统繁忙，请稍后重试！（解耦回退方法触发）-----";
19.     }
20. }
```

注意：该类必须以组件的形式添加 Spring 容器中才能生效，最常用的方式就是在类上标注 @Component 注解。

2. 在服务绑定接口 DeptHystrixService 标注的 @FeignClient 注解中添加 fallback 属性，属性值为 DeptHystrixFallBackService.class，代码如下。

```
01. package net.biancheng.c.service;
02.
03. import org.springframework.cloud.openfeign.FeignClient;
04. import org.springframework.stereotype.Component;
05. import org.springframework.web.bind.annotation.PathVariable;
06. import org.springframework.web.bind.annotation.RequestMapping;
07.
08. @Component
09. @FeignClient(value = "MICROSERVICECLOUDPROVIDERDEPTHYSTRIX", fallback = DeptHystrixFallBackService.class)
10. public interface DeptHystrixService {
11.     @RequestMapping(value = "/dept/hystrix/ok/{id}")
12.     public String deptInfo_Ok(@PathVariable("id") Integer id);
13.
14.     @RequestMapping(value = "/dept/hystrix/timeout/{id}")
15.     public String deptInfo_Timeout(@PathVariable("id") Integer id);
16. }
```

3. 重启 micro-service-cloud-consumer-dept-feign，然后关闭服务端 micro-service-cloud-provider-dept-hystrix-8004，使用浏览器访问 “<http://eureka7001.com/consumer/dept/hystrix/ok/1>” ，结果如下图。



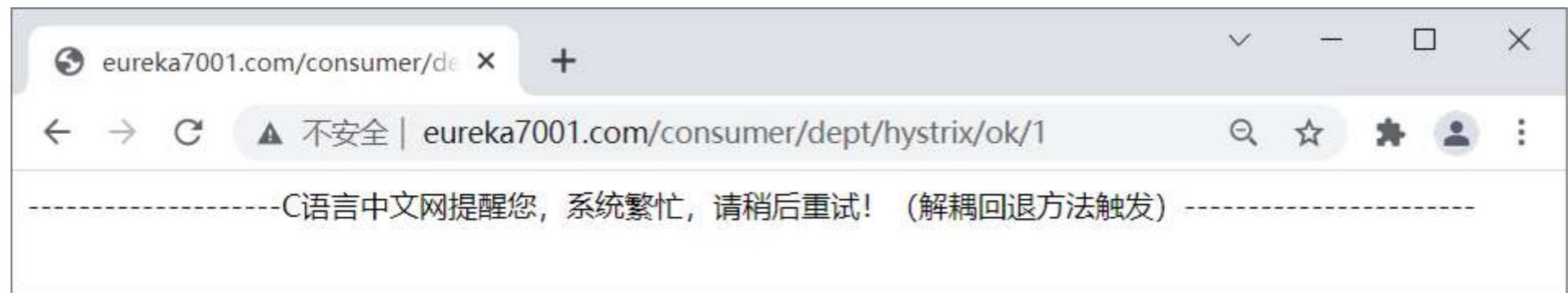


图7: Hystrix 解耦回退方法

Hystrix 服务熔断

熔断机制是为了应对雪崩效应而出现的一种微服务链路保护机制。

当微服务系统中的某个微服务不可用或响应时间太长时，为了保护系统的整体可用性，熔断器会暂时切断请求对该服务的调用，并快速返回一个友好的错误响应。这种熔断状态不是永久的，在经历了一定的时间后，熔断器会再次检测该微服务是否恢复正常，若服务恢复正常则恢复其调用链路。

熔断状态

在熔断机制中涉及了三种熔断状态：

- 熔断关闭状态 (Closed)：当服务访问正常时，熔断器处于关闭状态，服务调用方可以正常地对服务进行调用。
- 熔断开启状态 (Open)：默认情况下，在固定时间内接口调用出错率达到一个阈值（例如 50%），熔断器会进入熔断开启状态。进入熔断状态后，后续对该服务的调用都会被切断，熔断器会执行本地的降级 (Fallback) 方法。
- 半熔断状态 (Half-Open)：在熔断开启一段时间之后，熔断器会进入半熔断状态。在半熔断状态下，熔断器会尝试恢复服务调用方对服务的调用，允许部分请求调用该服务，并监控其调用成功率。如果成功率达到预期，则说明服务已恢复正常，熔断器进入关闭状态；如果成功率仍旧很低，则重新进入熔断开启状态。

三种熔断状态之间的转化关系如下图：

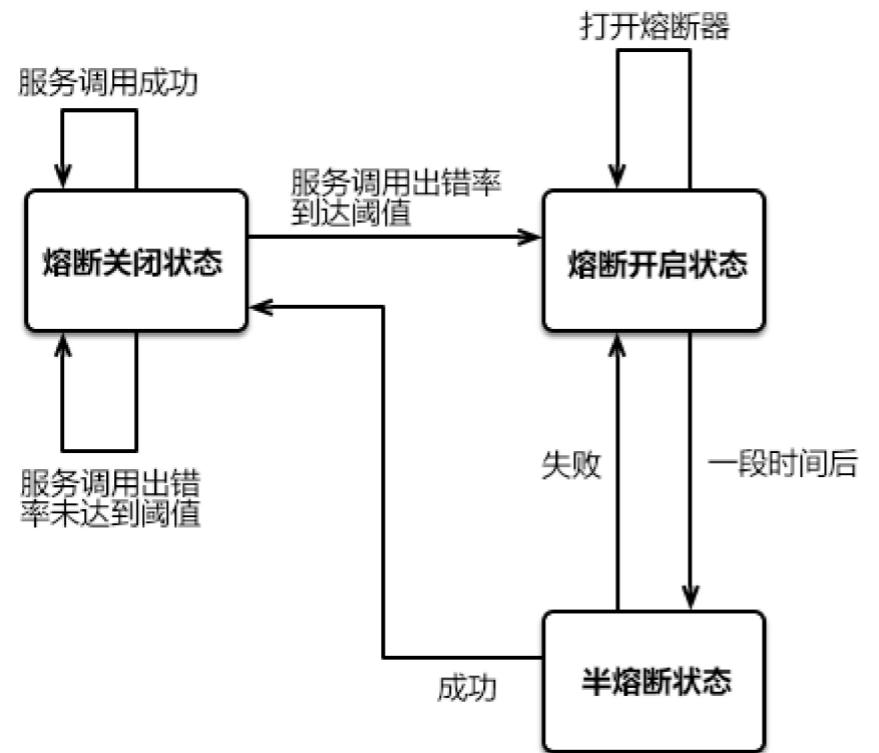


图8: 三种熔断状态转换

Hystrix 实现熔断机制

在 Spring Cloud 中，熔断机制是通过 Hystrix 实现的。Hystrix 会监控微服务间调用的状况，当失败调用到一定比例时（例如 5 秒内失败 20 次），就会启动熔断机制。

Hystrix 实现服务熔断的步骤如下：

1. 当服务的调用出错率达到或超过 Hystrix 规定的比率（默认为 50%）后，熔断器进入熔断开启状态。
2. 熔断器进入熔断开启状态后，Hystrix 会启动一个休眠时间窗，在这个时间窗内，该服务的降级逻辑会临时充当业务主逻辑，而原来的业务主逻辑不可用。
3. 当有请求再次调用该服务时，会直接调用降级逻辑快速地返回失败响应，以避免系统雪崩。
4. 当休眠时间窗到期后，Hystrix 会进入半熔断转态，允许部分请求对服务原来的主业务逻辑进行调用，并监控其调用成功率。
5. 如果调用成功率达到预期，则说明服务已恢复正常，Hystrix 进入熔断关闭状态，服务原来的主业务逻辑恢复；否则 Hystrix 重新进入熔断开启状态，休眠时间窗口重新计时，继续重复第 2 到第 5 步。

示例

下面我们就通过一个实例来验证下 Hystrix 是如何实现熔断机制的。

1. 在 micro-service-cloud-provider-dept-hystrix-8004 中的 DeptService 接口中添加一个 deptCircuitBreaker() 方法，代码如下。

```
01. package net.biancheng.c.service;
02.
03. public interface DeptService {
04.
05.     // hystrix 熔断器示例 ok
06.     public String deptInfo_0k(Integer id);
07.
08.     //hystrix 熔断器超时案例
09.     public String deptInfo_Timeout(Integer id);
10.
11.     // Hystrix 熔断机制案例
12.     public String deptCircuitBreaker(Integer id);
13. }
```

2. 在 DeptService 接口的实现类 DeptServiceImpl 添加 deptCircuitBreaker() 的方法实现及其回退方法，代码如下。

```
01. //Hystrix 熔断案例
02. @Override
03. @HystrixCommand(fallbackMethod = "deptCircuitBreaker_fallback", commandProperties = {
04.     //以下参数在 HystrixCommandProperties 类中有默认配置
05.     @HystrixProperty(name = "circuitBreaker.enabled", value = "true"), //是否开启熔断器
06.     @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "1000"), //统计时间窗
07.     @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"), //统计时间窗内请求次数
08.     @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), //休眠时间窗口期
09.     @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60") //在统计时间窗口期以内，请求失败率达到 60% 时进入熔断状态
10. })
11. public String deptCircuitBreaker(Integer id) {
```



```

12.     if (id < 0) {
13.         //当传入的 id 为负数时，抛出异常，调用降级方法
14.         throw new RuntimeException("c语言中文网提醒您，id 不能是负数！");
15.     }
16.     String serialNum = IdUtil.simpleUUID();
17.     return Thread.currentThread().getName() + "\t" + "调用成功，流水号为：" + serialNum;
18. }
19.
20. //deptCircuitBreaker 的降级方法
21. public String deptCircuitBreakerFallback(Integer id) {
22.     return "c语言中文网提醒您，id 不能是负数，请稍后重试!\t id:" + id;
23. }

```

在以上代码中，共涉及到了 4 个与 Hystrix 熔断机制相关的重要参数，这 4 个参数的含义如下表。

参数	描述
metrics.rollingStats.timeInMilliseconds	统计时间窗。
circuitBreaker.sleepWindowInMilliseconds	休眠时间窗，熔断开启状态持续一段时间后，熔断器会自动进入半熔断状态，这段时间就被称为休眠窗口期。
circuitBreaker.requestVolumeThreshold	请求总数阀值。 在统计时间窗内，请求总数必须到达一定的数量级，Hystrix 才可能会将熔断器打开进入熔断开启转态，而这个请求数量级就是 请求总数阀值。Hystrix 请求总数阀值默认为 20，这就意味着在统计时间窗内，如果服务调用次数不足 20 次，即使所有的请求都调用出错，熔断器也不会打开。
circuitBreaker.errorThresholdPercentage	错误百分比阀值。 当请求总数在统计时间窗内超过了请求总数阀值，且请求调用出错率超过一定的比例，熔断器才会打开进入熔断开启转态，而这个比例就是错误百分比阀值。错误百分比阀值设置为 50，就表示错误百分比为 50%，如果服务发生了 30 次调用，其中有 15 次发生了错误，即超过了 50% 的错误百分比，这时候将熔断器就会打开。

3. 在 DeptController 中添加一个 deptCircuitBreaker() 方法对外提供服务，代码如下。

```

01. // Hystrix 服务熔断
02. @RequestMapping(value = "/dept/hystrix/circuit/{id}")
03. public String deptCircuitBreaker(@PathVariable("id") Integer id) {
04.     String result = deptService.deptCircuitBreaker(id);
05.     log.info("result:" + result);
06.     return result;
07. }

```

4. 重启 micro-service-cloud-provider-dept-hystrix-8004，使用浏览器访问 “<http://eureka7001.com:8004/dept/hystrix/circuit/1>” ，结果如下图。





图9: Hystrix 实现熔断机制 调用正确示例

5. 浏览器多次（调用次数大于请求总数阀值）访问 “<http://eureka7001.com:8004/dept/hystrix/circuit/-2>” , 使调用出错率大于错误百分比阀值，结果下图。

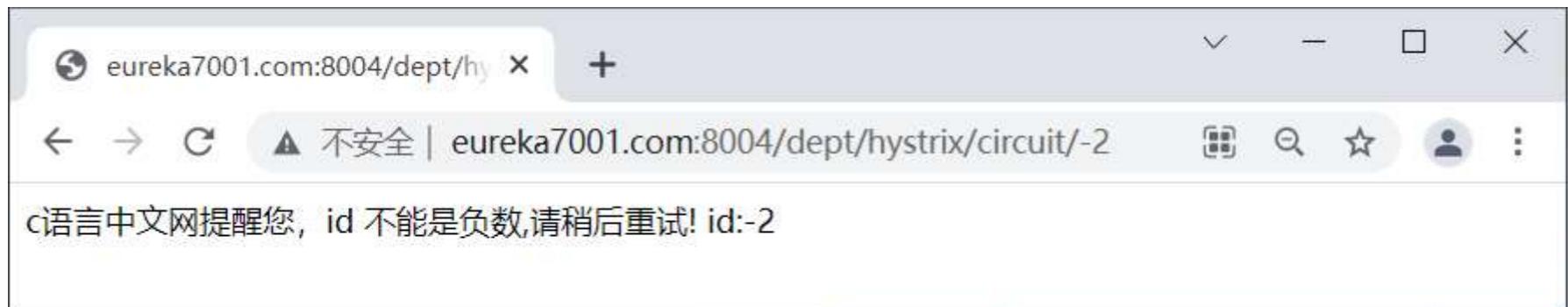


图10: Hystrix 实现熔断机制 错误调用

6. 重新将参数修改为正数（例如参数为 3）, 使用浏览器访问 “<http://eureka7001.com:8004/dept/hystrix/circuit/3>” , 结果如下图。

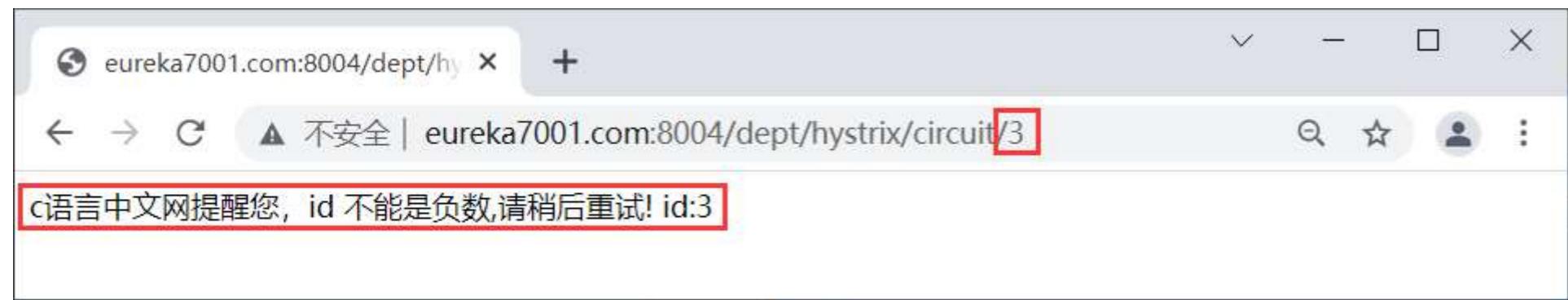


图11: Hystrix 熔断开启状态传入正数

通过图 11 可以看到，在熔断开启状态下，即使我们传入的参数已经是正数，调用的依然降级逻辑。

7. 继续连续访问 “<http://eureka7001.com:8004/dept/hystrix/circuit/3>” , 结果下图。



图12: Hystrix 进入熔断关闭状态

通过图 12 可以看出，当服务调用正确率上升到一定的利率后，Hystrix 进入熔断关闭状态。

Hystrix 故障监控

Hystrix 还提供了准实时的调用监控 (Hystrix Dashboard) 功能，Hystrix 会持续地记录所有通过 Hystrix 发起的请求的执行信息，并以统计报表的形式展示给用户，包括每秒执行请求的数量、成功请求的数量和失败请求的数量等。

下面我们就通过一个实例来搭建 Hystrix Dashboard，监控 micro-service-cloud-provider-dept-hystrix-8004 的运行情况。

1. 在父工程下新建一个名为 micro-service-cloud-consumer-dept-hystrix-dashboard-9002 的子模块，并在其 pom.xml 中添加以下依赖。

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
04. <modelVersion>4.0.0</modelVersion>
05. <parent>
06.   <artifactId>spring-cloud-demo2</artifactId>
07.   <groupId>net.biancheng.c</groupId>
08.   <version>0.0.1-SNAPSHOT</version>
09. </parent>
10. <groupId>net.biancheng.c</groupId>
11. <artifactId>micro-service-cloud-consumer-dept-hystrix-dashboard-9002</artifactId>
12. <version>0.0.1-SNAPSHOT</version>
13. <name>micro-service-cloud-consumer-dept-hystrix-dashboard-9002</name>
14. <description>Demo project for Spring Boot</description>
15. <properties>
16.   <java.version>1.8</java.version>
17. </properties>
18. <dependencies>
19.   <dependency>
20.     <groupId>org.springframework.boot</groupId>
21.     <artifactId>spring-boot-starter</artifactId>
22.   </dependency>
23.   <!--Spring Boot 测试依赖-->
24.   <dependency>
25.     <groupId>org.springframework.boot</groupId>
```

```

26.      <artifactId>spring-boot-starter-test</artifactId>
27.      <scope>test</scope>
28.    </dependency>
29.    <!--hystrix-dashboard 监控的依赖-->
30.    <dependency>
31.      <groupId>org.springframework.cloud</groupId>
32.      <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
33.    </dependency>
34.    <!--添加 Spring Boot 的监控模块-->
35.    <dependency>
36.      <groupId>org.springframework.boot</groupId>
37.      <artifactId>spring-boot-starter-actuator</artifactId>
38.    </dependency>
39.    <dependency>
40.      <groupId>org.springframework.boot</groupId>
41.      <artifactId>spring-boot-devtools</artifactId>
42.    </dependency>
43.    <dependency>
44.      <groupId>org.projectlombok</groupId>
45.      <artifactId>lombok</artifactId>
46.    </dependency>
47.  </dependencies>
48.
49.  <build>
50.    <plugins>
51.      <plugin>
52.        <groupId>org.springframework.boot</groupId>
53.        <artifactId>spring-boot-maven-plugin</artifactId>
54.      </plugin>
55.    </plugins>
56.  </build>
57. </project>

```

2. 在 micro-service-cloud-consumer-dept-hystrix-dashboard-9002 的 application.yml 中添加以下配置。

```

01. server:
02.   port: 9002 #端口号
03.
04. #http://eureka7001.com:9002/hystrix 熔断器监控页面
05. # localhost:8004//actuator/hystrix.stream 监控地址
06. hystrix:
07.   dashboard:
08.     proxy-stream-allow-list:
09.       - "localhost"

```

3. 在 micro-service-cloud-consumer-dept-hystrix-dashboard-9002 的主启动类上添加 @EnableHystrixDashboard 注解，开启 Hystrix 监控功能，代



码如下。

```
01. package net.biancheng.c;
02.
03. import org.springframework.boot.SpringApplication;
04. import org.springframework.boot.autoconfigure.SpringBootApplication;
05. import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;
06.
07. @SpringBootApplication
08. @EnableHystrixDashboard
09. public class MicroServiceCloudConsumerDeptHystrixDashboard9002Application {
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(MicroServiceCloudConsumerDeptHystrixDashboard9002Application.class, args);
13.     }
14.
15. }
```

4. 在 micro-service-cloud-provider-dept-hystrix-8004 的 net.biancheng.c.config 包下，创建一个名为 HystrixDashboardConfig 的配置类，代码如下。

```
01. package net.biancheng.c.config;
02.
03. import com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet;
04. import org.springframework.boot.web.servlet.ServletRegistrationBean;
05. import org.springframework.context.annotation.Bean;
06. import org.springframework.context.annotation.Configuration;
07.
08. @Configuration
09. public class HystrixDashboardConfig {
10.     /**
11.      * Hystrix dashboard 监控界面必须配置
12.      * @return
13.      */
14.     @Bean
15.     public ServletRegistrationBean getServlet() {
16.         HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
17.         ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);
18.         registrationBean.setLoadOnStartup(1);
19.         registrationBean.addUrlMappings("/actuator/hystrix.stream");//访问路径
20.         registrationBean.setName("hystrix.stream");
21.         return registrationBean;
22.     }
23.
24. }
```



5. 启动 micro-service-cloud-consumer-dept-hystrix-dashboard-9002，使用浏览器访问 “<http://eureka7001.com:9002/hystrix>” ，结果如下图。



图13：Hystrix 监控页面

6. 重启 micro-service-cloud-provider-dept-hystrix-8004，并将以下信息填到 Hystrix 监控页面中，如下图。

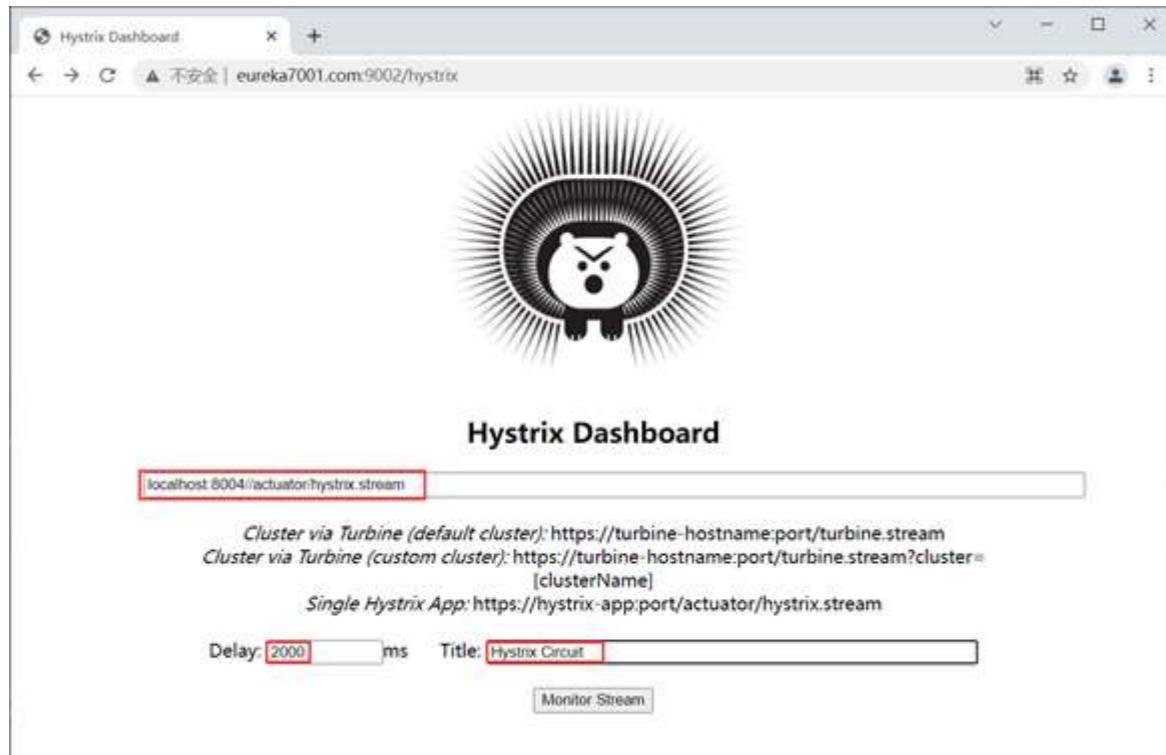


图14：Hystrix 监控信息

7. 点击下方的 Monitor Stream 按钮，跳转到 Hystrix 对 micro-service-cloud-provider-dept-hystrix-8004 的监控页面，如下图。



图15: Hystrix 监控微服务运行情况

8. 使用浏览器多次访问 “<http://eureka7001.com:8004/dept/hystrix/circuit/1>” 和 “<http://eureka7001.com:8004/dept/hystrix/circuit/-1>” , 查看 Hystrix 监控页面，如下图。



图16: Hystrix 监控服务运行情况

< 上一节

下一节 >

推荐阅读

[Go语言类型分支 \(switch判断空接口中变量的类型\)](#)

[C++ exception类：C++标准异常的基类](#)

[三元组顺序表，稀疏矩阵的三元组表示及（C语言）实现](#)

[物理地址（MAC地址）是什么？](#)

[Linux /etc/rc.d/rc.local配置文件用法](#)

[C# goto语句](#)

[解密Qt安装目录的结构](#)

[文本打开方式和二进制打开方式的区别是什么？](#)



精美而实用的网站，分享优质编程教程，帮助有志青年。千锤百炼，只为大作；精益求精，处处斟酌；这种教程，看一眼就倾心。

[关于网站](#) | [联系我们](#) | [网站地图](#)

Copyright ©2012-2023 biancheng.net

biancheng.net

