Hystrix详解

> 更多学习资料，一点课堂：www.yidiankt.com
>
> 一点课堂QQ群：984370849， QQ：2868289889
>
> 微信号：chengweixin9
>
> 免费公开课：https://ke.qq.com/course/394307

> https://github.com/Netflix/Hystrix/wiki/How-it-Works#Isolation

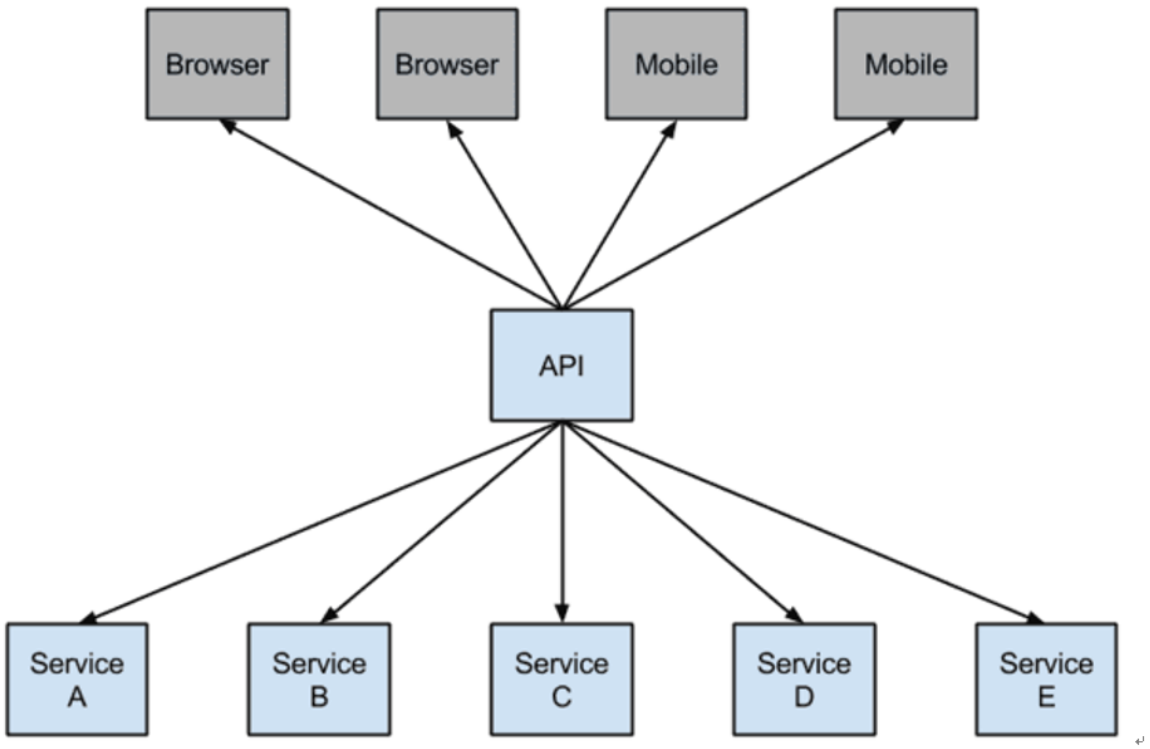
为什么需要断路器?

在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用，在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的"雪崩"效应。

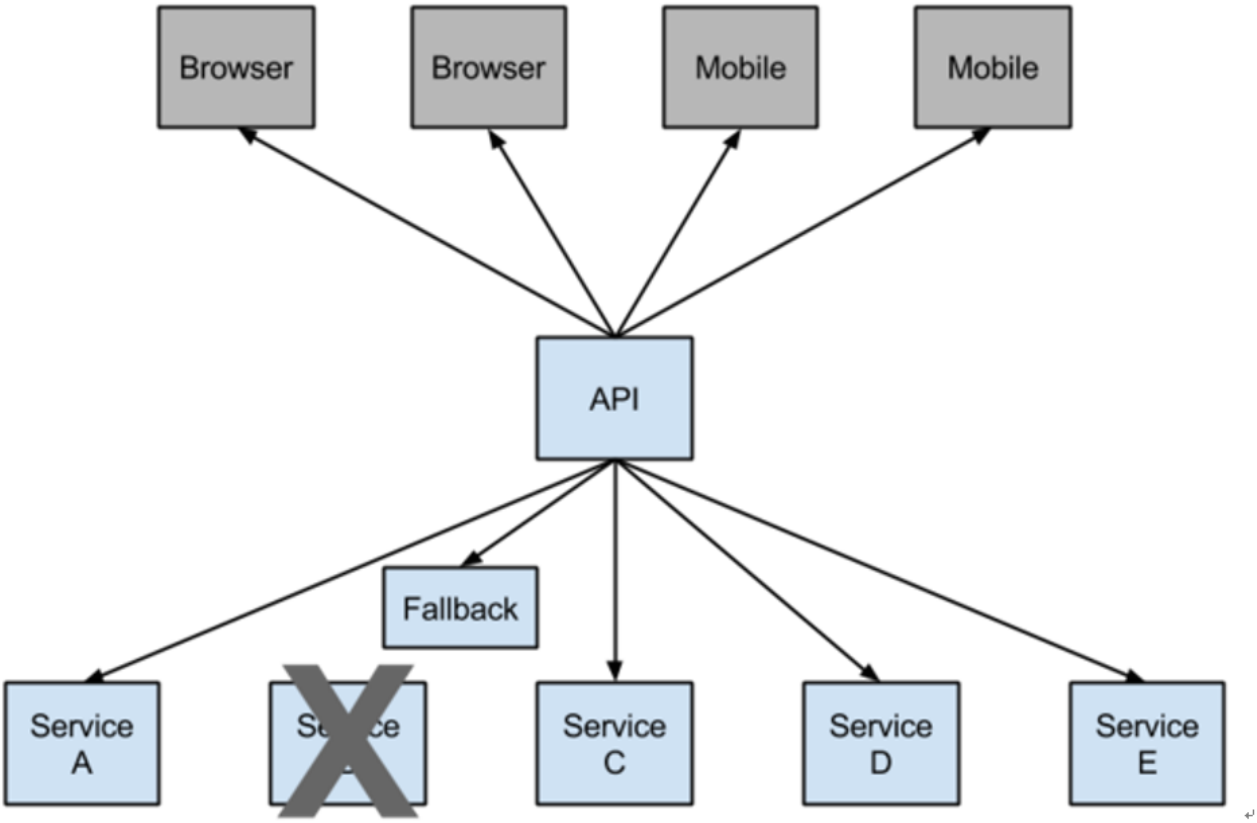
为了解决这个问题，业界提出了断路器模型。

断路器简介

Netflix开源了Hystrix组件，实现了断路器模式，SpringCloud对这一组件进行了整合。 在微服务架构中，一个请求需要调用多个服务是非常常见的，如下图：



较底层的服务如果出现故障，会导致连锁故障。当对特定的服务的调用的不可用达到一个阀值（Hystric 是5秒20次） 断路器将会被打开。



断路打开后，可用避免连锁故障，fallback方法可以直接返回一个固定值。

Hystrix特性：

1. 请求熔断： 当Hystrix Command请求后端服务失败数量超过一定比例(默认50%), 断路器会切换到开路状态(Open). 这时所有请求会直接失败而不会发送到后端服务. 断路器保持在开路状态一段时间后(默认5秒), 自动切换到半开路状态(HALF-OPEN).

   这时会判断下一次请求的返回情况, 如果请求成功, 断路器切回闭路状态(CLOSED), 否则重新切换到开路状态(OPEN). Hystrix的断路器就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力.

2. 服务降级：Fallback相当于是降级操作. 对于查询操作, 我们可以实现一个fallback方法, 当请求后端服务出现异常的时候, 可以使用fallback方法返回的值. fallback方法的返回值一般是设置的默认值或者来自缓存.告知后面的请求服务不可用了，不要再来了。

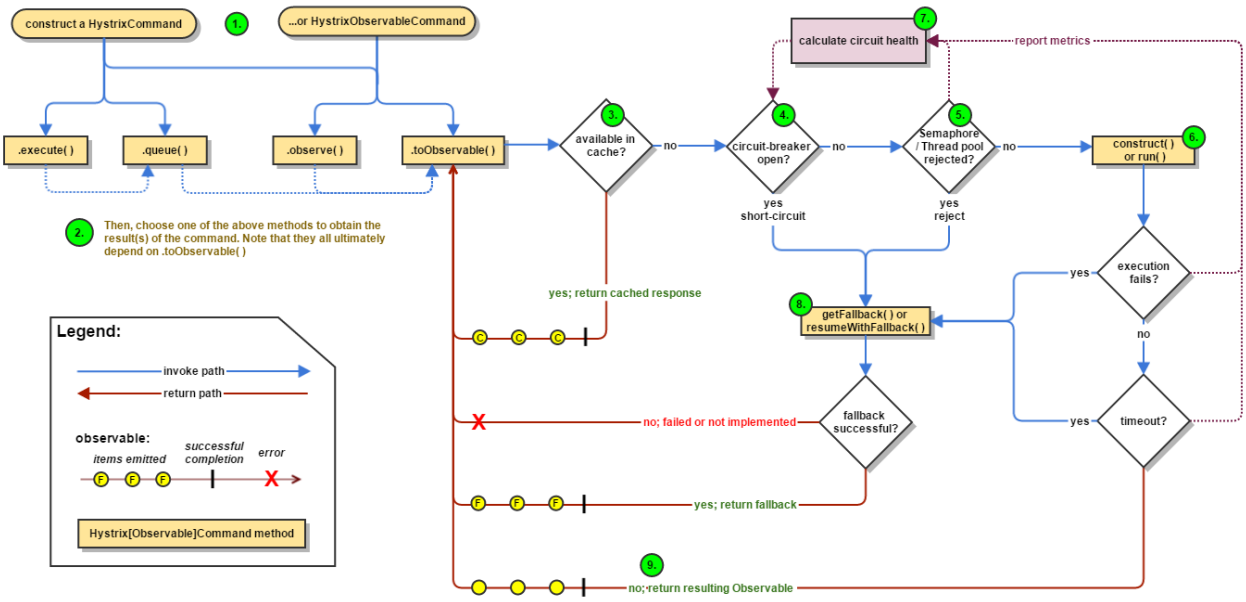
3. **依赖隔离(采用舱壁模式，Docker就是舱壁模式的一种)：在Hystrix中, 主要通过线程池来实现资源隔离。通常在使用的时候我们会根据调用的远程服务划分出多个线程池。比如说，一个服务调用两外两个服务，你如果调用两个服务都用一个线程池，那么如果一个服务卡在哪里，资源没被释放，后面的请求又来了，导致后面的请求都卡在哪里等待，导致你依赖的A服务把你卡在哪里，耗尽了资源，也导致了你另外一个B服务也不可用了。这时如果依赖隔离，某一个服务调用A B两个服务，如果这时我有100个线程可用，我给A服务分配50个，给B服务分配50个，这样就算A服务挂了，我的B服务依然可以用。**

4. 请求缓存：比如一个请求过来请求我userId=1的数据，你后面的请求也过来请求同样的数据，这时我不会继续走原来的那条请求链路了，而是把第一次请求缓存过了，把第一次的请求结果返回给后面的请求。

   > 请求缓存是在同一请求多次访问中保证只调用一次这个服务提供者的接口，在这同一次请求第一次的结果会被缓存，保证同一请求中同样的多次访问返回结果相同。

5. 请求合并：我依赖于某一个服务，我要调用N次，比如说查数据库的时候，我发了N条请求发了N条SQL然后拿到一堆结果，这时候我们可以把多个请求合并成一个请求，发送一个查询多条数据的SQL的请求，这样我们只需查询一次数据库，提升了效率。

Hystrix流程结构解析



- 流程说明

```
1:每次调用创建一个新的HystrixCommand,把依赖调用封装在run()方法中.
2:执行execute()/queue做同步或异步调用.
3:判断熔断器(circuit-breaker)是否打开,如果打开跳到步骤8,进行降级策略,如果关闭进入步骤.
4:判断线程池/队列/信号量是否跑满，如果跑满进入降级步骤8,否则继续后续步骤.
5:调用HystrixCommand的run方法.运行依赖逻辑
5a:依赖逻辑调用超时,进入步骤8.
6:判断逻辑是否调用成功
6a:返回成功调用结果
6b:调用出错，进入步骤8.
7:计算熔断器状态,所有的运行状态(成功， 失败， 拒绝,超时)上报给熔断器，用于统计从而判断熔断器状态.
8:getFallback()降级逻辑.
  以下四种情况将触发getFallback调用:
 (1):run()方法抛出非HystrixBadRequestException异常。
 (2):run()方法调用超时
 (3):熔断器开启拦截调用
 (4):线程池/队列/信号量是否跑满
8a:没有实现getFallback的Command将直接抛出异常
8b:fallback降级逻辑调用成功直接返回
8c:降级逻辑调用失败抛出异常
9:返回执行成功结果
```

集成Hystrix功能

- 添加pom文件

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

- 启动类中开启

```
@EnableHystrix
```

- 添加fallback类

```
@Component
public class MyFallback implements FeignService {
    @Override
    public String getUser(int id) {
        return "error getUser";
    }

    @Override
    public String getUser2(User user) {
        return "error getUser2";
    }
}
```

- feign注解添加

```
@FeignClient(value = "yidiankt-user", fallback = MyFallback.class)
public interface FeignService {

    @RequestMapping(value = "/user/{id}", method = RequestMethod.GET)
    String getUser(@PathVariable("id") int id);

    @RequestMapping(value = "/user2", method = RequestMethod.POST)
    String getUser2(User user);

}
```

- yml开启hystrix功能

```
feign:
  hystrix:
    enabled: true
```

- resttemplate方式

```
public class OrderService {
```

```
    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "userFallback")
    public String getUser(int id) {
        // 获取用户信息？？？
        String url = "http://yidiankt-user/user/{id}";
        String info = restTemplate.getForObject(url, String.class, id);
        return info;
    }

    // 添加服务器降级处理方法
    public String userFallback(int id) {
        return "error user fallback";
    }
}
```

依赖隔离

- 添加OrderCommand

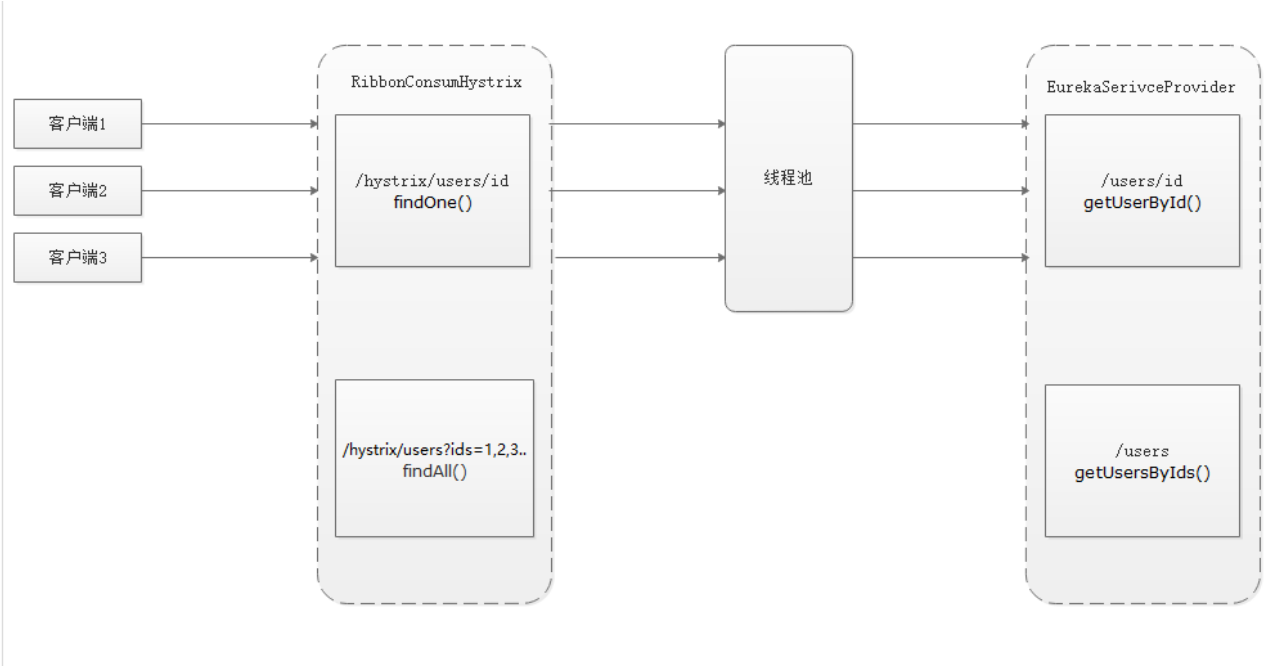
```
package com.yidiankt.service.pool;

import com.netflix.hystrix.*;

public class OrderCommand extends HystrixCommand<String> {

    private String value;

    public OrderCommand(String value) {
        super(Setter.withGroupKey(
                //服务分组
                HystrixCommandGroupKey.Factory.asKey("UserGroup"))
                //线程分组
                .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("UserPool"))
                //线程池配置

.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()
                        .withCoreSize(10)
                        .withKeepAliveTimeMinutes(5)
                        .withMaxQueueSize(10)
                        .withQueueSizeRejectionThreshold(10000))
                .andCommandPropertiesDefaults(
                        HystrixCommandProperties.Setter()

.withExecutionIsolationStrategy(HystrixCommandProperties.ExecutionIsolationStrateg
y.THREAD)));

        this.value = value;
    }

    @Override
    protected String run() throws Exception {
```

```
//        String url = "http://yidiankt-user/user/{id}";
//        String info = restTemplate.getForObject(url, String.class, id);
        String threadName = Thread.currentThread().getName();
        return threadName + " || " + value;
    }
}
```

- 测试

```
// 测试依赖隔离
public String testPool() throws ExecutionException, InterruptedException {

    UserCommand userCommand = new UserCommand("库里");

    OrderCommand orderCommand1 = new OrderCommand("篮球");
    OrderCommand orderCommand2 = new OrderCommand("足球");

    // 同步调用
    String val1 = userCommand.execute();
    String val2 = orderCommand1.execute();
    String val3 = orderCommand2.execute();

    // 异步调用
    Future<String> f1 = userCommand.queue();
    Future<String> f2 = userCommand.queue();
    Future<String> f3 = userCommand.queue();

    //        return "val1=" + val1 + "val2=" + val2 + "val3=" + val3;
    return "f1=" + f1.get() + "f2=" + f2.get() + "f3=" + f3.get();
}
```

请求合并是做什么的?

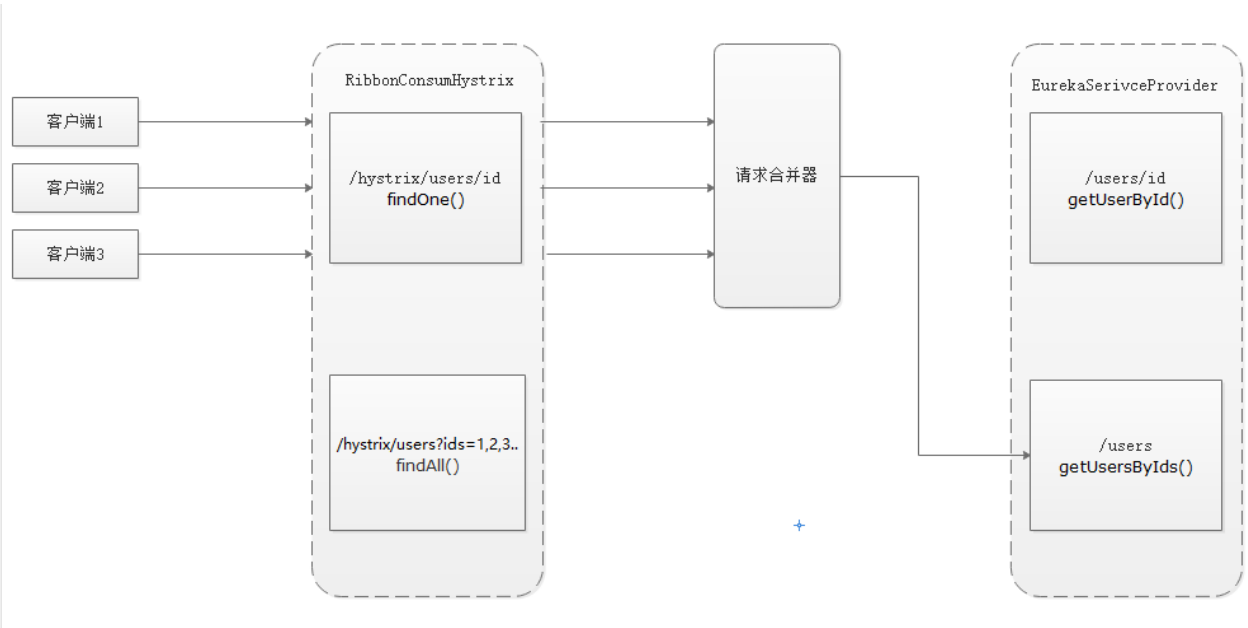


如图，多个客户端发送请求调用(消费者)项目中的findOne方法，这时候在这个项目中的线程池中会发申请与请求数量相同的线程数，对EurekaServiceProvider(服务提供者)的getUserById方法发起调用，每个线程都要调用一次，在高并发的场景下，这样势必会对服务提供者项目产生巨大的压力。

请求合并就是将单个请求合并成一个请求，去调用服务提供者，从而降低服务提供者负载的，一种应对高并发的解决办法

二、请求合并的原理

NetFlix在Hystrix为我们提供了应对高并发的解决方案----请求合并，如下图



通过请求合并器设置延迟时间，将时间内的，多个请求单个的对象的方法中的参数（id）取出来，拼成符合服务提供者的多个对象返回接口（getUsersByIds方法）的参数，指定调用这个接口（getUsersByIds方法），返回的对象List再通过一个方法（mapResponseToRequests方法），按照请求的次序将结果对象对应的装到Request对应的Response中返回结果。

三、请求合并适用的场景

在服务提供者提供了返回单个对象和多个对象的查询接口，并且单个对象的查询并发数很高，服务提供者负载较高的时候，我们就可以使用请求合并来降低服务提供者的负载

四、请求合并带来的问题

问题：即然请求合并这么好，我们是否就可以将所有返回单个结果的方法都用上请求合并呢？答案自然是否定的！

原因：

1. 我们为这个请求人为的设置了延迟时间，这样在并发不高的接口上使用请求缓存，会降低响应速度
2. 实现请求合并比较复杂

使用注解示例

```
@HystrixCommand(groupKey = "productStockOpLog", commandKey = "addProductStockOpLog",
fallbackMethod = "addProductStockOpLogFallback",
                commandProperties = {
                    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "400"),//指定多久超时，单位毫秒。超时进fallback
                    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value
= "10"),//判断熔断的最少请求数，默认是10；只有在一个统计窗口内处理的请求数量达到这个阈值，才会进行熔断与否的判断
                    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",
value = "10"),//判断熔断的阈值，默认值50，表示在一个统计窗口内有50%的请求处理失败，会触发熔断
                }
               )
public void addProductStockOpLog(Long sku_id, Object old_value, Object new_value) throws
Exception {
    if (new_value != null && !new_value.equals(old_value)) {
        doAddOpLog(null, null, sku_id, null, ProductOpType.PRODUCT_STOCK, old_value != null
? String.valueOf(old_value) : null, String.valueOf(new_value), 0, "C端", null);
    }
}

public void addProductStockOpLogFallback(Long sku_id, Object old_value, Object new_value)
throws Exception {
    LOGGER.warn("发送商品库存变更消息失败,进入Fallback,skuId:{},oldValue:{},newValue:{}",
sku_id, old_value, new_value);
}
```

```
@HystrixCommand(groupKey="UserGroup", commandKey = "GetUserByIdCommand",
                commandProperties = {
```

```
                    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "100"),//指定多久超时，单位毫秒。超
时进fallback
                    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value
= "10"),//判断熔断的最少请求数，默认是10；只有在一个统计窗口内处理的请求数量达到这个阈值，才会进行熔断与否
的判断
                    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",
value = "10"),//判断熔断的阈值，默认值50，表示在一个统计窗口内有50%的请求处理失败，会触发熔断
                },
                threadPoolProperties = {
                    @HystrixProperty(name = "coreSize", value = "30"),
                    @HystrixProperty(name = "maxQueueSize", value = "101"),
                    @HystrixProperty(name = "keepAliveTimeMinutes", value = "2"),
                    @HystrixProperty(name = "queueSizeRejectionThreshold", value = "15"),
                    @HystrixProperty(name = "metrics.rollingStats.numBuckets", value =
"12"),
                    @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds",
value = "1440")
                })
```

- 参数配置解释：https://www.jianshu.com/p/138f92aa83dc
- 配置信息最常用的几项

超时时间（默认1000ms，单位：ms）

（1）hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds

在调用方配置，被该调用方的所有方法的超时时间都是该值，优先级低于下边的指定配置

（2）hystrix.command.HystrixCommandKey.execution.isolation.thread.timeoutInMilliseconds

在调用方配置，被该调用方的指定方法（HystrixCommandKey方法名）的超时时间是该值

线程池核心线程数

hystrix.threadpool.default.coreSize（默认为10）

Queue

（1）hystrix.threadpool.default.maxQueueSize（最大排队长度。默认-1，使用SynchronousQueue。其他值则使用 LinkedBlockingQueue。如果要从-1换成其他值则需重启，即该值不能动态调整，若要动态调整，需要使用到下边这个配置）

（2）hystrix.threadpool.default.queueSizeRejectionThreshold（排队线程数量阈值，默认为5，达到时拒绝，如果配置了该选项，队列的大小是该队列）

注意：如果maxQueueSize=-1的话，则该选项不起作用

断路器

（1）hystrix.command.default.circuitBreaker.requestVolumeThreshold（当在配置时间窗口内达到此数量的失败后，进行短路。默认20个）

```
For example, if the value is 20, then if only 19 requests are received in the rolling
window (say a window of 10 seconds) the circuit will not trip open even if all 19 failed.

简言之，10s内请求失败数量达到20个，断路器开。

（2）hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds（短路多久以后开始尝试是否恢复，默认5s）

（3）hystrix.command.default.circuitBreaker.errorThresholdPercentage（出错百分比阈值，当达到此阈值后，开始短路。默认50%）

fallback

hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests（调用线程允许请求HystrixCommand.GetFallback()的最大数量，默认10。超出时将会有异常抛出，注意：该项配置对于THREAD隔离模式也起作用）
```

- 其他属性说明：https://blog.csdn.net/tongtong_use/article/details/78611225

（三）hystrix监控

hystrix的监控信息以”text/event-stream”的方式暴露给外部系统，因此我们可以很方便的看到这些信息。

使用ribbon调用的服务只需添加”actuator”依赖，访问”/hystrix”路径即可看到ribbon调用时的监控信息； 使用feign调用的服务中需要添加”hystrix”依赖，并在启动类设置”@@EnableCircuitBreaker”注解，然后即可通过”/hystrix.stream”访问监控信息； 可以另起一个项目，添加”hystrix-dashboard”依赖，在启动类上添加”@EnableHystrixDashboard”注解，然后即可在可视化的界面中查看指定调用链路的监控信息； （四）Turbine聚合监控数据 前面介绍的都是单服务的数据监控，实际意义并不大，我们可以使用turbine实现服务的聚合监控。通过turbine可以将单个服务的”hystrix.stream”聚合成为”turbine.stream”，并通过dashboard可视化即可查看聚合监控数据。

```
1. 新建dashboard项目;
2. 新建turbine项目，引入turbine依赖，并在启动类配置"@EnableTurbine"，并通过配置文件指明进行监控数据聚合的服
务，配置如下:
    server:
    port: 9998
    spring:
      application:
        name: cms-monitor
    eureka:
      client:
        service-url:
          defaultZone: http://localhost:11000/eureka/
      instance:
        prefer-ip-address: true
    turbine:
      app-config: cms-gateway,cms-admin
```

```
      cluster-name-expression: "'default'"
```

3.依次启动，并进行服务调用，在dashboadr访问"turbine.stream"，即可看到聚合的监控数据：

hystrix dashboard界面监控参数