SpringCloud服务降级

1.概述

1.分布式系统面临的问题

复杂的分布式系统可能面临着复杂的依赖问题，一个服务可能依赖多个服务，然而依赖关系在某些时候将不可避免的面临着失败，如果一个服务出现超时，可能会面临服务雪崩的情况

服务雪崩：多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其他的微服务，这就是所谓的“扇出”，如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器的所有资源在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

所以，通常当你发现一个模块下的某个实例失败之后，这时候这个模块依然还会接收流量，然后这个有问题的模块还调用了其他的模块，这样就会发生级联故障，或者叫雪崩。

2.Hystrix是什么?

Hystrix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，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。**

“断路器”本身是一种开关装置，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、不必要的占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

3.Hystrix重要概念

1.服务降级

假设对方系统不可用了，你需要给我一个兜底的解决方法。

服务器繁忙，请稍后再试，不让客户端等待并立刻返回一个友好的提示，fallback。

哪些情况下会触发服务降级呢?

程序运行异常，超时，服务熔断触发服务降级，线程池/信号量打满也会导致服务降级

2.服务熔断

类比保险丝达到最大服务访问之后，直接拒绝访问，拉闸限电，然后**调用服务降级的方法**并返回友好提示

3.服务限流

秒杀等高并发操作，严禁一窝蜂的过来拥挤，大家排队，一秒N个，有序进行

4.补充

服务降级:系统有限的资源的合理协调

- 概念：服务降级一般是指在服务器压力剧增的时候，根据实际业务使用情况以及流量，对一些服务和页面有策略的不处理或者用一种简单的方式进行处理，从而**释放服务器资源的资源以保证核心业务的正常高效运行。**
- 原因： 服务器的资源是有限的，而请求是无限的。在用户使用即并发高峰期，会影响整体服务的性能，严重的话会导致宕机，以至于某些重要服务不可用。故高峰期为了保证核心功能服务的可用性，就需要对某些服务降级处理。可以理解为舍小保大
- 应用场景： 多用于微服务架构中，一般当整个微服务架构整体的负载超出了预设的上限阈值（和服务器的配置性能有关系），或者即将到来的流量预计会超过预设的阈值时（比如双11、6.18等活动或者秒杀活动）
- 服务降级是从整个系统的负荷情况出发和考虑的，对某些负荷会比较高的情况，为了预防某些功能（业务场景）出现负荷过载或者响应慢的情况，在其内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的fallback（退路）错误处理信息。这样，虽然提供的是一个有损的服务，但却保证了整个系统的稳定性和可用性。
- 需要考虑的问题：
- - 区分那些服务为核心？那些非核心
  - 降级策略（处理方式，一般指如何给用户友好的提示或者操作）
  - 自动降级还是手动降

服务熔断：应对雪崩效应的链路自我保护机制。可看作降级的特殊情况

- 概念：应对微服务雪崩效应的一种链路保护机制，类似股市、保险丝
- 原因： 微服务之间的数据交互是通过远程调用来完成的。服务A调用服务，服务B调用服务c，某一时间链路上对服务C的调用响应时间过长或者服务C不可用，随着时间的增长，对服务C的调用也越来越多，然后服务C崩溃了，但是链路调用还在，对服务B的调用也在持续增多，然后服务B崩溃，随之A也崩溃，导致雪崩效应
- 服务熔断是应对雪崩效应的一种微服务链路保护机制。例如在高压电路中，如果某个地方的电压过高，熔断器就会熔断，对电路进行保护。同样，在微服务架构中，熔断机制也是起着类似的作用。当**调用链路的某个微服务不可用或者响应时间太长时，会进行服务熔断，不再有该节点微服务的调用，快速返回错误的响应信息。**当检测到该节点微服务调用响应正常后，恢复调用链路。
- 服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，暂时停止对该服务的调用。

在Spring Cloud框架里，熔断机制通过Hystrix实现。**Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败，就会启动熔断机制。**

-应用场景：微服务架构中，多个微服务相互调用处使用

- 需要考虑问题：
- - 如何所依赖的服务对象不稳定
  - 失败之后如何快速恢复依赖对象，如何探知依赖对象是否恢复

服务降级和服务熔断区别

- 触发原因不一样，服务熔断由链路上某个服务引起的，服务降级是从整体的负载考虑
- 管理目标层次不一样，服务熔断是一个框架层次的处理，服务降级是业务层次的处理
- - 实现方式不一样，服务熔断一般是自我熔断恢复，服务降级相当于人工控制
- 触发原因不同 服务熔断一般是某个服务（下游服务）故障引起，而服务降级一般是从整体负荷考虑；

一句话：

服务熔断是应对系统服务雪崩的一种保险措施，给出的一种特殊降级措施。而服务降级则是更加宽泛的概念，主要是对系统整体资源的合理分配以应对压力。

服务熔断是服务降级的一种特殊情况，他是防止服务雪崩而采取的措施。系统发生异常或者延迟或者流量太大，都会触发该服务的服务熔断措施，链路熔断，返回兜底方法。这是对局部的一种保险措施。

服务降级是对系统整体资源的合理分配。区分核心服务和非核心服务。对某个服务的访问延迟时间、异常等情况做出预估并给出兜底方法。这是一种全局性的考量，对系统整体负荷进行管理。

限流：限制并发的请求访问量，超过阈值则拒绝；

降级：服务分优先级，牺牲非核心服务（不可用），保证核心服务稳定；从整体负荷考虑；

熔断：依赖的下游服务故障触发熔断，避免引发本系统崩溃；系统自动执行和恢复

4.Hystrix案例

1.将7001恢复成单机版。方便演示

```
defaultZone: http://eureka7001.com:7001/eureka/ #单机版
```

2.构建携带Hystrix的服务提供方cloud-provider-hystrix-payment-8001

pom

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>SpringCloudStudyNew</artifactId>
        <groupId>com.echo</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>cloud-provider-hystrix-payment-8001</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <dependencies>
<!--        引入hystrix-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
        <!--        引入EurekaClient-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!--        引入自定义的公共模块-->
        <dependency>
            <groupId>com.echo</groupId>
            <artifactId>cloud-api-common</artifactId>
            <version>${project.version}</version>
        </dependency>
        <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.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-starter</artifactId>
            <version>1.1.10</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <!--        springboot热部署-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <!--            runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其参与。与compile相比，跳过编译而已，-->
            <scope>runtime</scope>
            <!--            <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

3.配置文件

```yaml
server:
  port: 8001

spring:
  application:
    name: cloud-provider-hystrix-payment

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/
```

4.业务类

```java
package com.echo.service;

import org.springframework.stereotype.Service;

import java.util.concurrent.TimeUnit;

@Service
public class PaymentService {
    /**
     * 正常访问肯定没问题的方法
     * @param id
     * @return
     */
    public String paymentInfoOk(Integer id){
        return "线程池: " + Thread.currentThread().getName() + "
PaymentInfo_OK,id: " + id + "\t" + "哈哈";
    }

    /**
     * 会超时的方法
     * @param id
     * @return
     */
    public String paymentInfoTimeout(Integer id){
        int timeNumber = 3;
        try {
            TimeUnit.SECONDS.sleep(3);
        }
        catch (InterruptedException e){
            e.printStackTrace();
        }
        return "线程池: " + Thread.currentThread().getName() + "
PaymentInfo_Timeout,id: " + id
                + "\t" + "哈哈 耗时:" + timeNumber + "秒钟";

    }
}
```

5.控制器

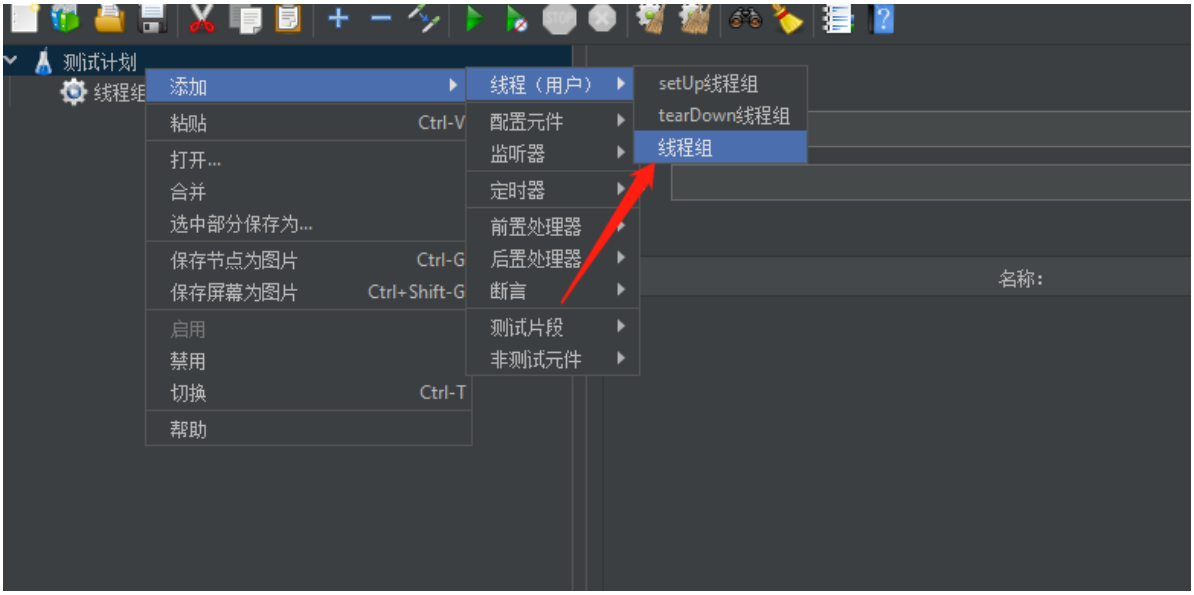
```java
package com.echo.controller;

import com.echo.service.PaymentService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
@Slf4j
public class PaymentController {
    @Autowired
    private PaymentService paymentService;
    @Value("${server.port}")
    private String serverPort;

    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfoOk(@PathVariable("id") Integer id){
        String result = paymentService.paymentInfoOk(id);
        log.info("result:" + result);
        return result;
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfoTimeout(@PathVariable("id") Integer id){
        String result = paymentService.paymentInfoTimeout(id);
        log.info(result);
        return result;
    }
}
```

以该平台为根基，从正确->错误->降级熔断->恢复

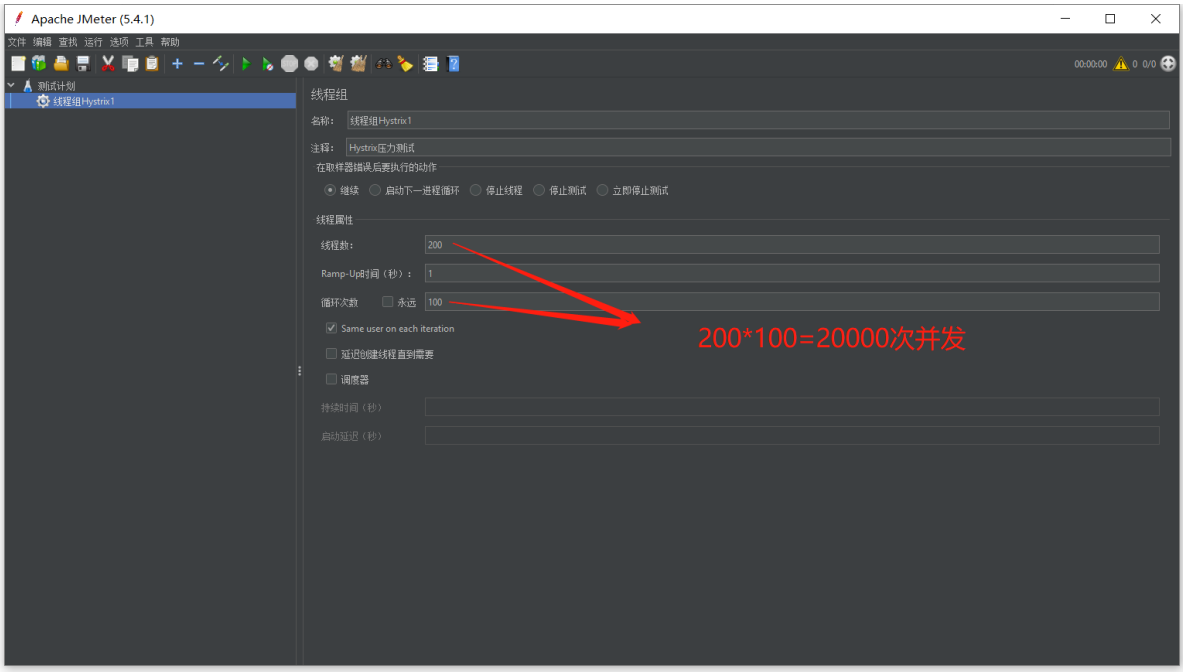
5.Hystrix高并发测试

1.使用JMeter进行高并发压力测试

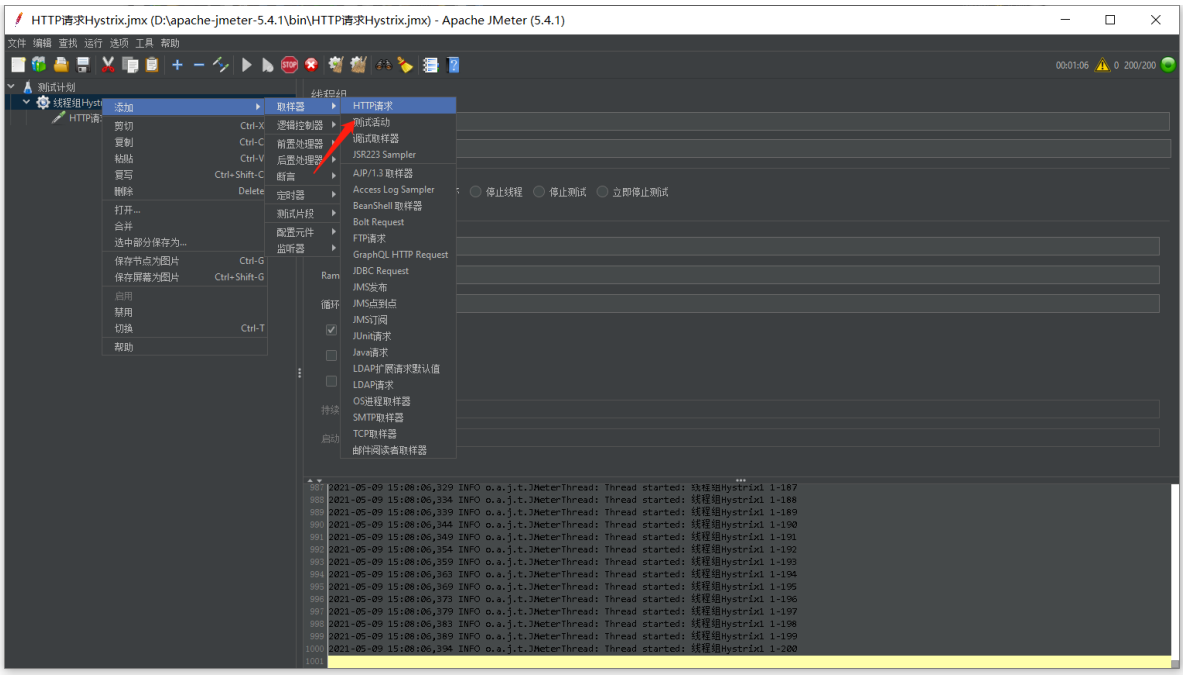
添加线程组

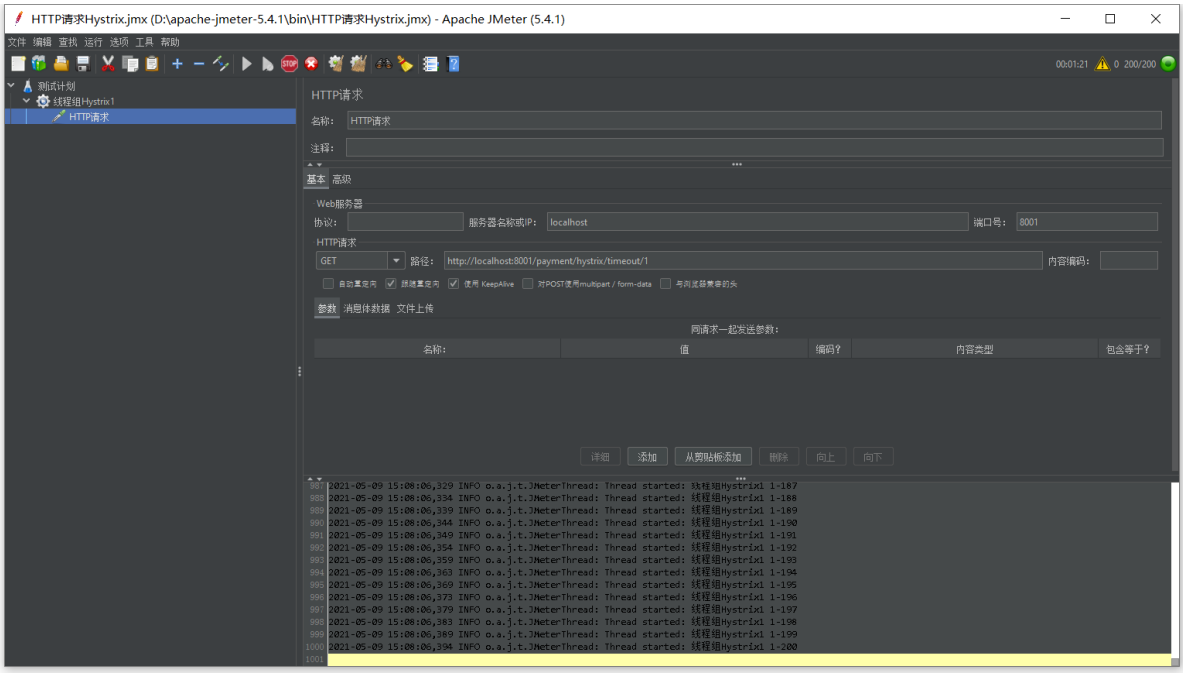


设置并发数



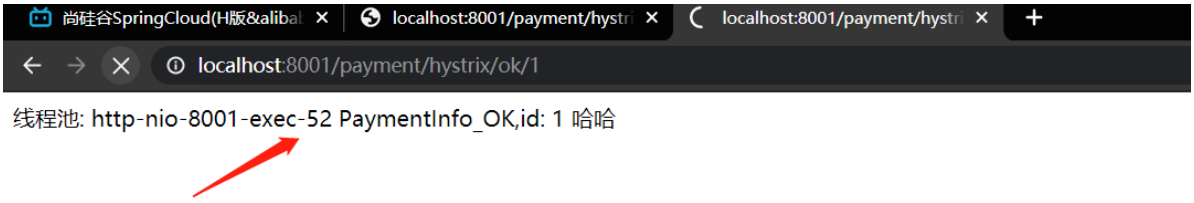
添加一个HTTP请求





保存，点击测试，**注意这里测试的是timeout的那个接口**

查看结果



可以看到，原来秒回的ok请求，也被拖缓了速度，也开始转圈等待。停止了所有对http://localhost:8001/payment/hystrix/timeout/1压力测试的线程之后，http://localhost:8001/payment/hystrix/ok/1 OK这个接口也就恢复了正常。

由此可以得出结论：

在同一个微服务下提供的业务接口，其中一个接口收到了高并发压力，其余接口都会受到影响

2.为什么会卡死?

Springboot提供的默认容器是Tomcat，Tomcat的默认工作线程是200个，其工作线程被打满了，没有多余的线程来分解压力和处理。

3.JMeter压力测试可以得出的结论

上面还是服务提供者8001自己测试，假如此时外部的消费者80也来访问，那消费者只能干等着，最终导致消费端80不满意，服务端8001直接被拖死

6.新建cloud-consumer-feign-hystrix-order-80

1.新建该工程集成feign和hystrix

pom

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <parent>
        <artifactId>SpringCloudStudyNew</artifactId>
        <groupId>com.echo</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>cloud-consumer-feign-hystrix-order-80</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <dependencies>
        <!--        hystrix-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
        <!--        引入openfeign,它天生整合了ribbon-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
        <!--        引入EurekaClient-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!--        引入自定义的公共模块-->
        <dependency>
            <groupId>com.echo</groupId>
            <artifactId>cloud-api-common</artifactId>
            <version>${project.version}</version>
        </dependency>
        <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>
        <!--        springboot热部署-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <!--            runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其参与。与compile相比，跳过编译而已，-->
            <scope>runtime</scope>
            <!--            <optional>true</optional>表示两个项目之间依赖不传递；不设置optional或者optional是false，表示传递依赖。-->
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

yml

```
server:
  port: 80

eureka:
  client:
    register-with-eureka: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka
```

主启动

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

2.业务类

service

```
@Component
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")
public interface PaymentHystrixService {
    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfoOk(@PathVariable("id") Integer id);

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfoTimeout(@PathVariable("id") Integer id);
}
```

controller

```
@RestController
@RequestMapping("/consumer")
public class PaymentHystrixController {
    @Autowired
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfoOk(@PathVariable("id") Integer id){
```

```
        return paymentHystrixService.paymentInfoOk(id);
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfoTimeout(@PathVariable("id") Integer id){
        return paymentHystrixService.paymentInfoTimeout(id);
    }
}
```

2.故障出现原因

8001同一层次的其他接口服务被困死，因为tomcat线程池里面的工作线程已经被挤占完毕，80此时调用8001，客户端访问缓慢

7.降级容错的维度要求

1.超时导致服务器变慢（转圈）

应该超时不再等待，不应该将错误页面返回前端

2.出错，宕机或者程序运行出错

出错要有兜底

3.解决

对方服务（8001）超时了，调用者（80）不能一直卡死等待，必须有服务降级

对方服务（8001）宕机了，调用者（80）不能一直卡死等待，必须有服务降级

对方服务（8001）没问题，调用者（80）自己出故障或者有自我要求（自己的等待时间小于服务提供者），自己处理降级

8.Hystrix之服务降级，服务提供方fallback

1.降级的配置@HystrixCommand

设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，做服务降级fallback

2.8001修改业务类

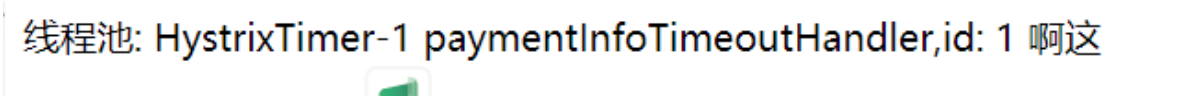
```
@Service
public class PaymentService {
    /**
     * 正常访问肯定没问题的方法
     * @param id
     * @return
     */
    public String paymentInfoOk(Integer id){
        return "线程池: " + Thread.currentThread().getName() + "
PaymentInfo_OK,id: " + id + "\t" + "哈哈";
    }

    /**
     * 会超时的方法
     * @param id
     * @return
     */
    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",
```

```
            commandProperties = {
            //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
            //设置这个线程的超时时间是3秒钟
            @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="3000")
    })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(Integer id){
        /**
         * 约定 3秒钟，正常。
         * 超过3秒钟，异常
         * 一旦调用服务方法失败并抛出了错误信息之后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中指定的方法
         */
        int timeNumber = 5;
        try {
            TimeUnit.SECONDS.sleep(timeNumber);
        }
        catch (InterruptedException e){
            e.printStackTrace();
        }
        return "线程池: " + Thread.currentThread().getName() + "
PaymentInfo_Timeout,id: " + id
                + "\t" + "哈哈 耗时:" + timeNumber + "秒钟";

    }


    public String paymentInfoTimeoutHandler(Integer id){
        return "线程池: " + Thread.currentThread().getName() + "
paymentInfoTimeoutHandler,id: " + id
                + "\t" + "啊这";
    }
}
```

3.主启动类激活

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker   //激活hystrix
public class PaymentHystrixApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentHystrixApplication.class,args);
    }
}
```

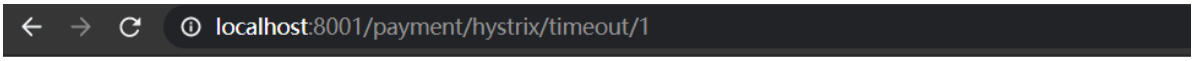
4.结果



可以看到正常处理了，而且线程池那里，使用的是HystrixTimer，即Hystrix自己的线程池，做到了与服务提供方线程的隔离

5.再修改服务为直接抛出异常，看是否能正常处理

```
    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",
            commandProperties = {
            //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
            //设置这个线程的超时时间是3秒钟
            @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="3000")
    })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(Integer id){
        /**
         * 约定 3秒钟，正常。
         * 超过3秒钟，异常
         * 一旦调用服务方法失败并抛出了错误信息之后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中指定的方法
         */
//        int timeNumber = 5;
//        try {
//            TimeUnit.SECONDS.sleep(timeNumber);
//        }
//        catch (InterruptedException e){
//            e.printStackTrace();
//        }
        //直接抛出异常
        int x = 1 / 0;
        return "线程池: " + Thread.currentThread().getName() + "  PaymentInfo_Timeout,id: " + id
                + "\t" + "哈哈 耗时:"  + "秒钟";

    }
```



线程池: hystrix-PaymentService-2 paymentInfoTimeoutHandler,id: 1 系统繁忙或运行出错，请稍后再试

可以看到，无论是运行异常，还是超时，都可以正确服务降级

6.服务端降级测试正常，修改回正常状态，默认超时时间是3秒，让系统睡2秒

```
    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",
            commandProperties = {
            //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
            //设置这个线程的超时时间是3秒钟
            @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="3000")
    })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(Integer id){
        /**
         * 约定 3秒钟，正常。
         * 超过3秒钟，异常
         * 一旦调用服务方法失败并抛出了错误信息之后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中指定的方法
         */
        int timeNumber = 2;
        try {
            TimeUnit.SECONDS.sleep(timeNumber);
        }
```

```
        catch (InterruptedException e){
            e.printStackTrace();
        }
        //直接抛出异常
        //int x = 1 / 0;
        return "线程池: " + Thread.currentThread().getName() + "
PaymentInfo_Timeout,id: " + id
                + "\t" + "哈哈 耗时:"  + timeNumber + "秒钟";

    }
```

9.Hystrix之服务降级，服务消费方fallback

1.首先，服务降级既可以放在消费方，也可以放在提供方，但是**一般都是放在消费方。**

2.题外话：我们自己配置的热部署方式对java代码的改动明显，但是对于@HystrixCommand内属性的修改建议重启微服务

3.修改服务消费方的yml，开启hystrix

```
server:
  port: 80

eureka:
  client:
    register-with-eureka: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka

feign:
  hystrix:
    enabled: true
```

4.主启动类添加Hystrix的支持

```
@SpringBootApplication
@EnableFeignClients
@EnableHystrix  //开启Hystrix
public class FeignHystrixOrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(FeignHystrixOrderApplication.class,args);
    }
}
```

5.修改controller，添加熔断方法

```
@RestController
@RequestMapping("/consumer")
@Slf4j
public class PaymentHystrixController {
    @Autowired
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfoOk(@PathVariable("id") Integer id){
        return paymentHystrixService.paymentInfoOk(id);
```

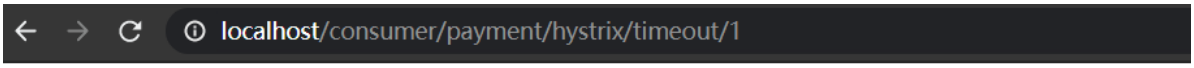
```
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",
            commandProperties = {
                    //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
                    //设置这个线程的超时时间是1.5秒钟,而服务提供方的提供服务的时间是2秒，超时时间是3秒，所以会熔断
                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="1500")
            })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(@PathVariable("id") Integer id){
        return paymentHystrixService.paymentInfoTimeout(id);
    }

    public String paymentInfoTimeoutHandler(Integer id){
        return "我是消费者80，服务提供方的服务繁忙，请10秒钟之后再试，或者自己运行出错，请检查自己。。。。";
    }
}
```

结果

我是消费者80，服务提供方的服务繁忙，请10秒钟之后再试，或者自己运行出错，请检查自己。。。。

6.改为出现异常

```
@GetMapping("/payment/hystrix/timeout/{id}")
@HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",
                commandProperties = {
                    //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
                    //设置这个线程的超时时间是1.5秒钟,而服务提供方的提供服务的时间是2秒，超时时间是3秒，所以会熔断
                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="1500")
                })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
public String paymentInfoTimeout(@PathVariable("id") Integer id){
    int age = 1 / 0;
    return paymentHystrixService.paymentInfoTimeout(id);
}
```



我是消费者80，服务提供方的服务繁忙，请10秒钟之后再试，或者自己运行出错，请检查自己。。。。

10.Hystrix之全局服务降级，DefaultProperties

1.当前问题

每个业务方法，都对应一个熔断方法，代码膨胀。需要有一个全局的服务降级方法，同时也可以对每个方法有独自的服务降级方法

2.全局fallback方法

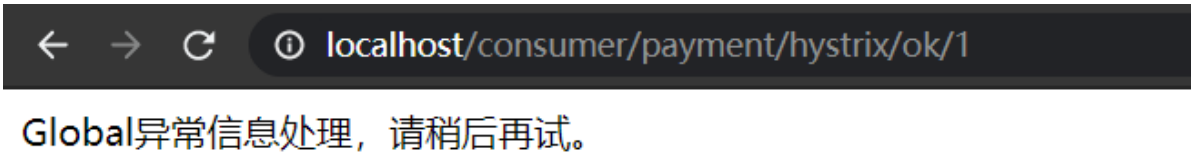
@DefaultProperties(defaultFallback="")

如果说，每一个方法都配置一个fallback方法，这在技术上是可以的，但是实际开发过程中是没必要的。所以，应该是除了个别核心业务有专属的fallback方法之外，其他的方法可以通过使用@DefaultProperties(defaultFallback="")统一跳转到统一处理结果页面。

3.修改服务消费方的controller

```
@RestController
@RequestMapping("/consumer")
@Slf4j
@DefaultProperties(defaultFallback = "paymentGlobalFallbackMethod")
public class PaymentHystrixController {
    @Autowired
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/payment/hystrix/ok/{id}")
    @HystrixCommand //添加注解但是不单独指明熔断方法，则标识，改方法也可以触发服务降级，并且使用全局fallback
    public String paymentInfoOk(@PathVariable("id") Integer id){
        int x = 1/0;
        return paymentHystrixService.paymentInfoOk(id);
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",    //不使用全局fallback，使用指定的fallback
            commandProperties = {
                    //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback
                    //设置这个线程的超时时间是1.5秒钟,而服务提供方的提供服务的时间是2秒，超时时间是3秒，所以会熔断
                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value="1500")
            })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(@PathVariable("id") Integer id){
        int age = 1 / 0;
        return paymentHystrixService.paymentInfoTimeout(id);
    }

    public String paymentInfoTimeoutHandler(Integer id){
        return "我是消费者80，服务提供方的服务繁忙，请10秒钟之后再试，或者自己运行出错，请检查自己。。。。";
    }

    //下面是全局fallback方法
    public String paymentGlobalFallbackMethod(){
        return "Global异常信息处理，请稍后再试。";
    }
}
```

4.访问ok方法，那里面添加了异常，看是否会触发全局熔断



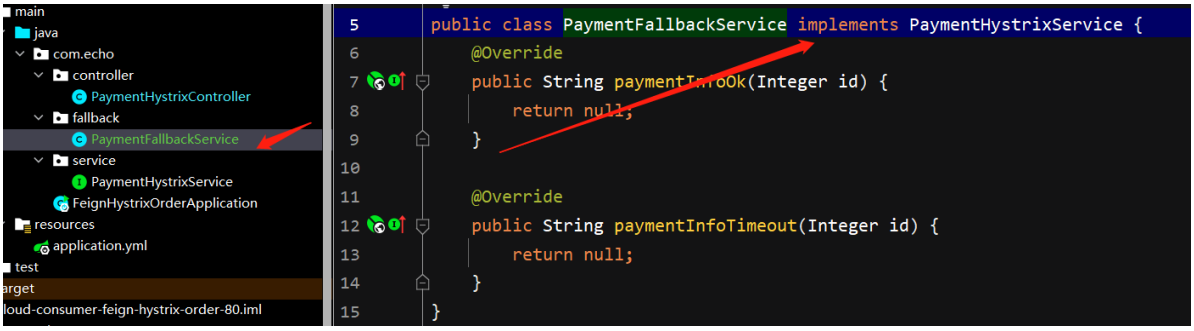
结果触发成功，记得把异常注释掉。

11.Hystrix之通配服务降级Fallback

1.服务降级，客户端去调用服务端，碰上服务端宕机或者关闭

本次案例服务降级处理是在客户端80实现完成，与服务端8001没有关系，只需要为Feign客户端定义的接口添加一个服务降级处理的实现类即可实现解耦

2.根据cloud-consumer-feign-hystrix-order-80中已经有的PaymentHystrixService接口，重新新建一个类(PaymentFallbackService)实现该接口，统一为接口里面的方法进行异常处理。




```
@Component
public class PaymentFallbackService implements PaymentHystrixService {
    @Override
    public String paymentInfoOk(Integer id) {
        return "***PaymentFallbackService***paymentInfoOk***fallback***";
    }

    @Override
    public String paymentInfoTimeout(Integer id) {
        return "***PaymentFallbackService***paymentInfoTimeout***fallback***";
    }
}
```

3.修改PaymentFallbackService 指定fallback

```
@Component
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT",fallback =
PaymentFallbackService.class)
public interface PaymentHystrixService {
    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfoOk(@PathVariable("id") Integer id);

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfoTimeout(@PathVariable("id") Integer id);
}
```

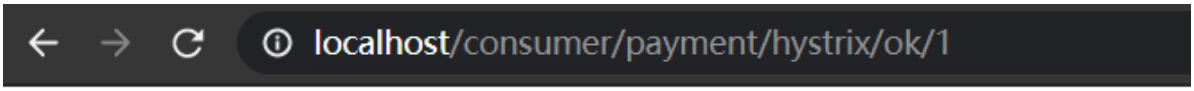
4.保证配置文件中开启了Hystrix

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

5.将原来的@HystrixCommand注释掉

```java
public class PaymentHystrixController {
    @Autowired
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/payment/hystrix/ok/{id}")
    //@HystrixCommand //添加注解但是不单独指明熔断方法，则标识，改方法也可以触发服务降级，并且使用全局fallback
    public String paymentInfoOk(@PathVariable("id") Integer id){
        //int x = 1/0;
        return paymentHystrixService.paymentInfoOk(id);
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
//    @HystrixCommand(fallbackMethod = "paymentInfoTimeoutHandler",    //不使用全局fallback，使用指定的fal
//            commandProperties = {
//                    //设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级
//                    //设置这个线程的超时时间是1.5秒钟,而服务提供方的提供服务的时间是2秒，超时时间是3秒，所以
//                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value=
//            })   //如果该方法出现异常/超时，就会调用fallback中指定的方法
    public String paymentInfoTimeout(@PathVariable("id") Integer id){
        int age = 1 / 0;
        return paymentHystrixService.paymentInfoTimeout(id);
    }
```

6关掉服务提供方8001测试



PaymentFallbackServicepaymentInfoOk***fallback***

成功

12.Hystrix之服务熔断理论

1.断路器：就是保险丝

2.熔断机制概述

熔断机制是应对雪崩效应的一种微服务链路保护机制，当扇出链路的某个微服务出错不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息。**当检测到该节点微服务调用响应正常后，恢复调用链路。**

在Spring Cloud框架中，熔断机制通过Hystrix实现。Hystrix会监控微服务间的调用的状况，当**失败的调用到一定的阈值，缺省是5秒内20次调用失败，就会启动熔断机制。**熔断机制的注解是@HystrixCommand

13.Hystrix之服务熔断案例

1.修改cloud-provider-hystrix-payment-8001

在paymentService中添加服务熔断的代码,这些配置可以在官网找到

```
//============服务熔断
@HystrixCommand(
    fallbackMethod = "paymentCircuitBreakerFallback",
    commandProperties = {
        //是否开启断路器
        @HystrixProperty(name = "circuitBreaker.enabled",value = "true"),
        //请求次数
        @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold",value="10"),
        //时间窗口期
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value
= "10000"),
        //失败率达到多少后，跳闸
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value
= "60")
        //总结一下就是在10秒的时间窗口期，以10秒中为一个时间单位，如果在10请求中，有60%失败，
则跳闸
    }
)
public String paymentCircuitBreaker(@PathVariable("id")Integer id){
    if (id < 0){
        throw new RuntimeException("****ID 不能为负数");
    }
    String serialNumber = IdUtil.simpleUUID();
    return Thread.currentThread().getName() + "\t" + "调用成功,流水号: " +
serialNumber;
}

public String paymentCircuitBreakerFallback(@PathVariable("id")Integer id){
    return "ID 不能为负数，请稍后再试，/(T o T)/~~   id:" + id;
}
```

2.在controller中添加测试熔断的方法

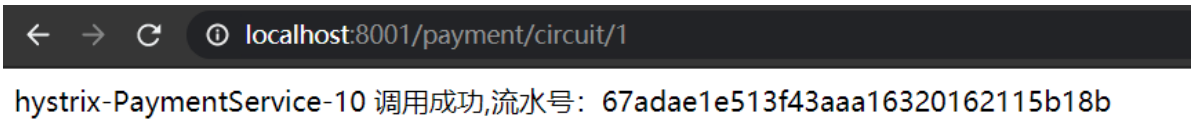
```
//=======服务熔断
@GetMapping("/payment/circuit/{id}")
public String paymentCircuitBreaker(@PathVariable("id") Integer id){
    String result = paymentService.paymentCircuitBreaker(id);
    log.info("****result" + result);
    return result;
}
```

3.测试，在10秒内一直使用负数进行请求，让服务报错

然后访问正常的请求

可以发现，断路器触发。

但是，触发以后，错误少了，错误率下降了，这种“半开”的状态就会慢慢打开，从而回复正常。



14.Hystrix小总结

1.熔断打开

请求不再待用当前服务，内部设置始终一般为MTTR（平均故障时间），当打开时间达到所设时钟则进入半熔断状态

2.熔断关闭

熔断关闭不会对服务进行熔断

3.熔断半开

部分请求根据规则调用当前服务，如果请求成功且符合规则则认为当前服务回复正常，关闭熔断

4.断路器在什么情况下开始起作用?

```
 //===========服务熔断
    @HystrixCommand(
            fallbackMethod = "paymentCircuitBreakerFallback",
            commandProperties = {
                    //是否开启断路器
                    @HystrixProperty(name = "circuitBreaker.enabled",value =
"true"),
                    //请求次数
                    @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold",value="10"),
                    //时间窗口期
                    @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds",value = "10000"),
                    //失败率达到多少后，跳闸
                    @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage",value = "60")
```

```
                    //总结一下就是在10秒的时间窗口期，以10秒中为一个时间单位，如果在10请求
中，有60%失败，则跳闸
            }
    )
```

涉及到断路器的三个重要参数：

快照时间窗口，请求总阈值，错误百分比阈值

快照时间窗口：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗口，默认为最近的10秒

请求总阈值：在快照时间窗内，必须满足请求总阈值才有资格熔断，默认为20次，意味着在10秒内，如果该Hystrix命令的调用次数不足20次，即使所有的请求都超时或者其他原因失败，断路器都不会打开。

错误百分比阈值：当请求总数在快照时间窗内超过了阈值，比如发生了30次调用，如果在这30次调用中，有15次发生了异常，也就是超过了50%的错误百分比，在默认设定50%阈值情况下，这时候断路器就会打开。

5.断路器开启或者关闭的条件

- 当满足一定的阈值时（默认10秒内超过20个请求次数）
- 当失败率达到一定的时候（默认10秒内超过50%的请求失败）
- 到达以上阈值，断路器将会开启
- 当断路器开启的时候，所有请求都不会转发
- 一段时间之后（默认是5秒钟），这个时候，断路器是半开的状态，会让其中一个请求进行转发，如果成功，断路器会关闭，若失败，继续开启。重复4和5

6.断路器打开之后

- 再有请求调用的时候，将不会调用主逻辑，而是直接调用降级fallback，通过断路器，实现了自动地发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果
- 原来的主逻辑如何恢复呢？对于这一问题，Hystrix也为我们实现了自动恢复功能，当断路器打开的时候，对主逻辑进行了熔断，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑上，如果此次请求返回正常，那么断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

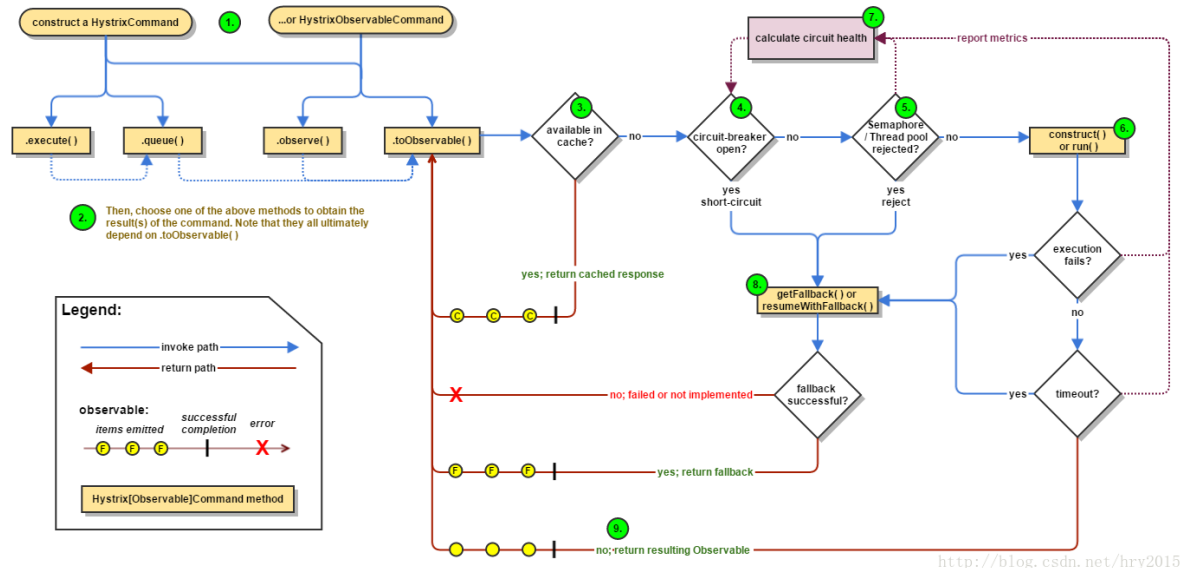
7.所有配置

```
@HystrixCommand(
            fallbackMethod = "paymentCircuitBreakerFallback",
            groupKey = "strGroupCommand",
            commandKey = "strCommand",
            threadPoolKey = "strThreadPool",
            commandProperties = {
                    //设置隔离策略，THREAD标识线程池SEMAPHORE：信号池隔离
                    @HystrixProperty(name =
"execution.isolation.strategy",value="THREAD"),
                    //当隔离策略选择信号池隔离的时候，用来设置信号池的大小（最大并发数）
                    @HystrixProperty(name =
"execution.isolation.semaphore.maxConcurrentRequests",value = "10"),
                    //配置命令执行的超时时间
                    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds",value = "3000"),
                    //是否启用超时时间
                    @HystrixProperty(name = "execution.timeout.enable",value =
"true"),
```

```
                    //执行超时的时候是否中断
                    @HystrixProperty(name =
"execution.isolation.thread.interruptOnTimeout",value = "true"),
                    //执行被取消的时候是否中断
                    @HystrixProperty(name =
"execution.isolation.thread.interruptOnCancel",value = "true"),
                    //允许回调方法执行的最大并发数
                    @HystrixProperty(name =
"execution.isolation.semaphore.maxConcurrentRequests",value = "10"),
                    //服务降级是否启用，是否执行回调函数
                    @HystrixProperty(name = "fallback.enabled",value = "true"),
                    //是否启用断路器
                    @HystrixProperty(name = "circuitBreaker.enabled",value =
"true"),
                    //该属性用来设置在滚动时间窗中，断路器熔断的最小请求数。例如，默认值为20
时，如果滚动时间窗（默认10s)
                    //内仅收到了19个请求，即使这19个请求都失败了，断路器也不会打开
                    @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold",value = "20"),
                    //该属性用来设置在滚动时间窗中，请求数量超过
circuitBreaker.requestVolumeThreshold的情况下，如果错误请求
                    //的百分比超过了50，就把断路器设置为打开的状态，否则就设置为关闭的状态
                    @HystrixProperty(name =
"circuitBreaker.errorThresholdPercentage",value = "50"),
                    //该属性用来设置断路器打开后的休眠时间窗，休眠时间窗结束之后，会将断路器
设置为“半开”的状态，尝试熔断的请求命令，
                    //如果依然失败就将断路器继续设置为“打开”的状态，如果成功就设置为“关闭”的
状态
                    @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds",value = "5000"),
                    //断路器强制打开
                    @HystrixProperty(name = "circuitBreaker.forceOpen",value =
"false"),
                    //断路器强制关闭
                    @HystrixProperty(name = "circuitBreaker.forceClosed",value =
"false"),
                    //滚动时间窗设置，该时间用于断路器判断健康度时，需要收集信息的持续时间
                    @HystrixProperty(name =
"metrics.rollingStats.timeInMilliseconds",value = "10000"),
                    //该属性用来设置滚动时间窗统计指标信息时，划分“桶”的数量，断路器在收集指
标信息的时候，会根据设置的时间窗长度
                    //拆分成多个“桶”来累计各个度量值，每个“桶”记录了一段时间内的采集指标。
                    //比如10秒内拆分成10个“桶”收集，所以timeInMilliseconds必须能被
numBuckets整除，否则会出现异常
                    @HystrixProperty(name =
"metrics.rollingStats.numBuckets",value = "10"),
                    //该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算，如果设置为
false，那么所有的概要统计都要返回-1
                    @HystrixProperty(name =
"metrics.rollingPercentile.enabled",value = "false"),
                    //该属性用来设置百分位统计的滚动窗口的持续时间，单位为毫秒
                    @HystrixProperty(name =
"metrics.rollingPercentile.timeInMilliseconds",value = "600000"),
                    //该属性用来设置百分位统计的滚动窗口中使用的“桶”的数量
                    @HystrixProperty(name =
"metrics.rollingPercentile.numBuckets",value = "600000"),
                    //该属性用来设置在执行过程中每个“桶”中保留的最大执行次数，如果在滚动时间
窗内发生超过该设定值的执行次数，
```

```
                    //就从最初的位置开始重写。例如，将该值设置为100，滚动窗口为10秒，若在10秒内一个“桶”中发生了500次执行，
                    //那么该“桶”中只保留最后的100次执行的统计，另外，增加该值的大小将会增加内存量的消耗，并增加排序百分位数所需要的计算时间。
                    @HystrixProperty(name = "metrics.rollingPercentile.bucketSize",value = "100"),
                    //该属性用来设置采集影响断路器状态的健康快照（请求的成功，错误百分比）的间隔等待时间
                    @HystrixProperty(name = "metrics.healthSnapshot.intervalInMilliseconds",value = "500"),
                    //是否开启请求缓存
                    @HystrixProperty(name = "requestCache.enabled",value = "true"),
                    //HystrixCommand的执行和时间是否打印日志到HystrixRequestLog中
                    @HystrixProperty(name = "requestLog.enabled",value = "true"),
            },
            threadPoolProperties = {
                    //该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最大并发量
                    @HystrixProperty(name = "coreSize",value = "10"),
                    //该参数用来设置线程池的最大队列大小。当设置为-1时，线程池将使用SynchronousQueue实现的队列，否则将使用LinkedBlockingQueue实现的队列
                    @HystrixProperty(name = "maxQueueSize",value = "-1"),
                    //该参数用来为队列设置拒绝阈值，通过该参数，即使队列没有达到最大值也能拒绝请求，该参数主要是对LinkedBlockingQueue队列的补充，
                    //因为LinkedBlockingQueue队列不能动态修改它的对象的大小，而通过该属性就可以调整拒绝请求的队列的大小了
                    @HystrixProperty(name = "queueSizeRejectionThreshold",value = "5"),
            }
    )
```

8.官网流程图



15.服务监控hystrixDashboard

1.概述

除了隔离依赖服务的调用意外，Hystrix还提供了准实时的调用监控，Hystrix会持续地记录所有通过Hystrix发起的请求执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求，多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。SpringCloud也提供了Hystrix Dashboard的整合，对监控内容转化成可视化界面。

2.新建cloud-consumer-hystrix-dashboard-9001

pom

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>SpringCloudStudyNew</artifactId>
        <groupId>com.echo</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>cloud-consumer-hystrix-dashboard-9001</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard</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-actuator</artifactId>
        </dependency>
        <!--        springboot热部署-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <!--            runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
周期需要其参与。与compile相比，跳过编译而已，-->
            <scope>runtime</scope>
            <!--            <optional>true</optional>表示两个项目之间依赖不传递；不设
置optional或者optional是false，表示传递依赖。-->
            <optional>true</optional>
        </dependency>
        <dependency>
```

```
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

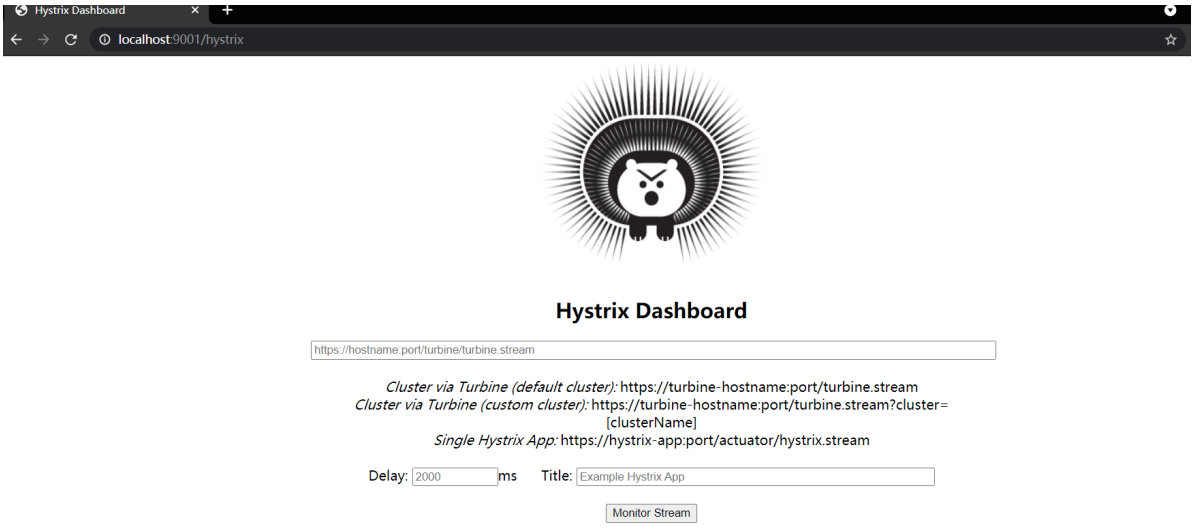
yml

```
server:
  port: 9001
```

主启动

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication.class,args);
    }
}
```

2.启动后访问



3.继续配置

要监视哪个服务，哪个服务里一定要配置

```
<!--        监控信息的完善-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
```

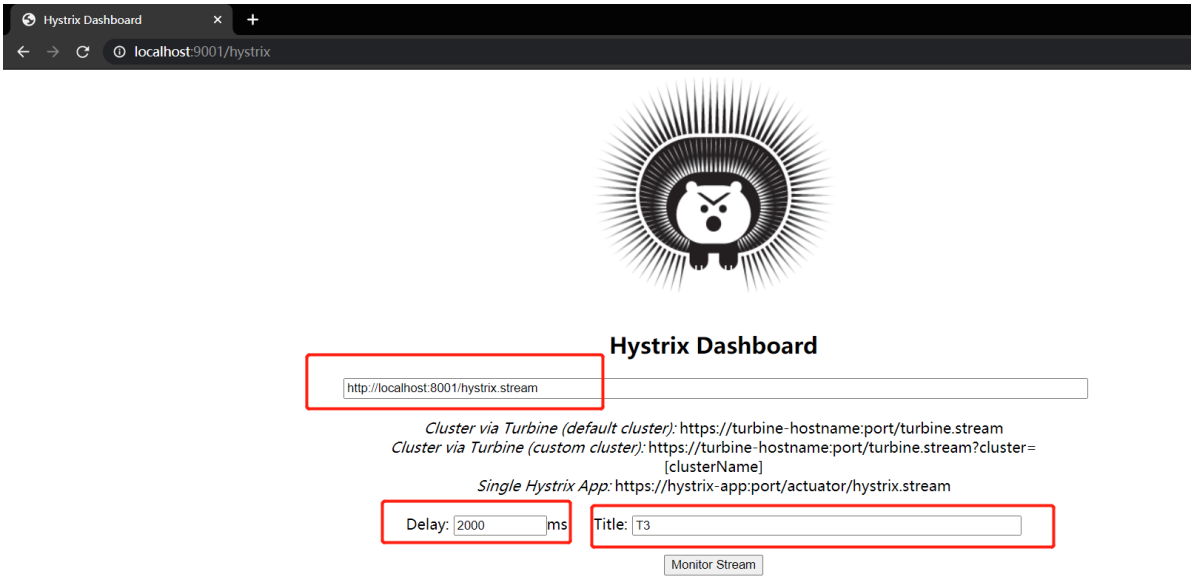
用hystrix-dashboard -9001监控hystrix-payment-8001

首先一定要配置一个坑，在hystrix-payment-8001主启动类中配置

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker    //激活hystrix
public class PaymentHystrixApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentHystrixApplication.class,args);
    }

    /**
     * 此配置是为了服务监控而配置，与服务容错本身无关，SpringCloud升级之后的坑
     * ServletRegistrationBean因为Springboot的默认路径不是"/hystrix.stream",
     * 只要在自己的项目里配置上下面的servlet就可以了
     * @return
     */
    @Bean
    public ServletRegistrationBean getServlet(){
        HystrixMetricsStreamServlet hystrixMetricsStreamServlet = new
HystrixMetricsStreamServlet();
        ServletRegistrationBean<HystrixMetricsStreamServlet> registrationBean =
new ServletRegistrationBean<>(hystrixMetricsStreamServlet);
        registrationBean.setLoadOnStartup(1);
        registrationBean.addUrlMappings("/hystrix.stream");
        registrationBean.setName("HystrixMetricsStreamServlet");
        return registrationBean;
    }
}
```

4.启动一个eureka或者eureka集群都可

5.在dashboard中填写监控地址

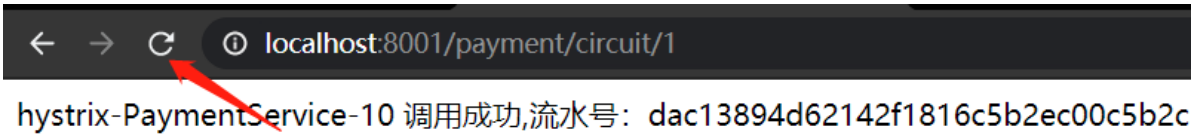


6.测试原来两个不同的地址，一个正确的一个错误的

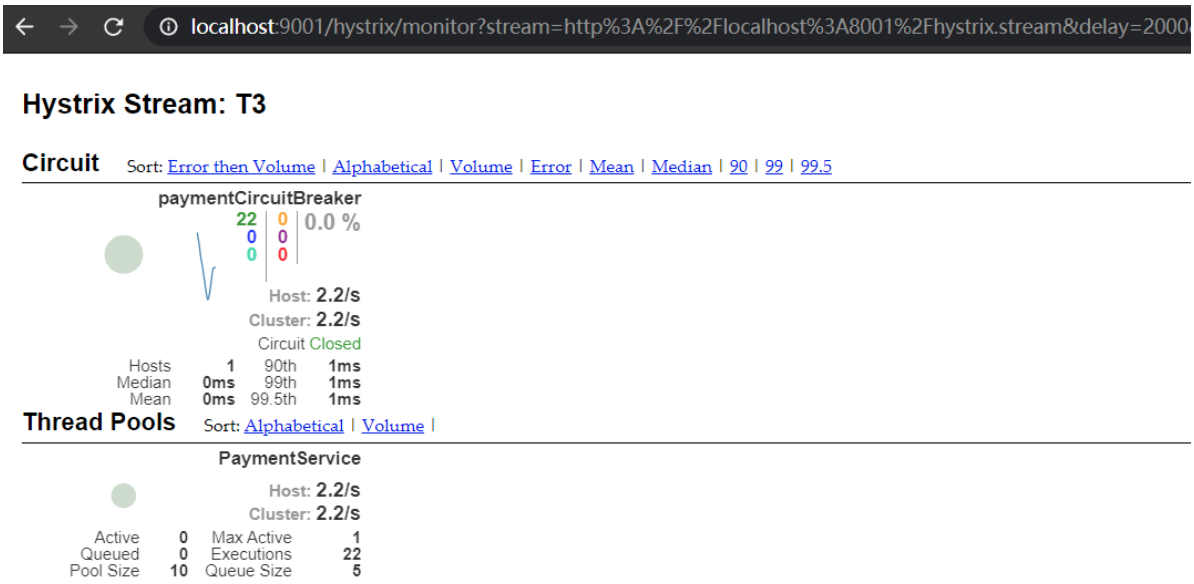
http://localhost:8001/payment/circuit/1 正确的

http://localhost:8001/payment/circuit/-1 错误的

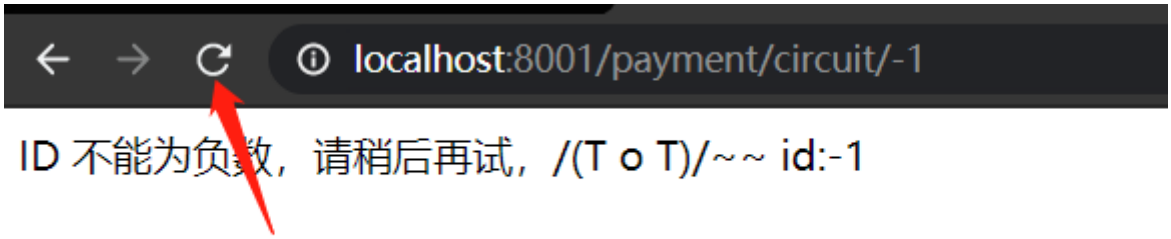
7.测试结果

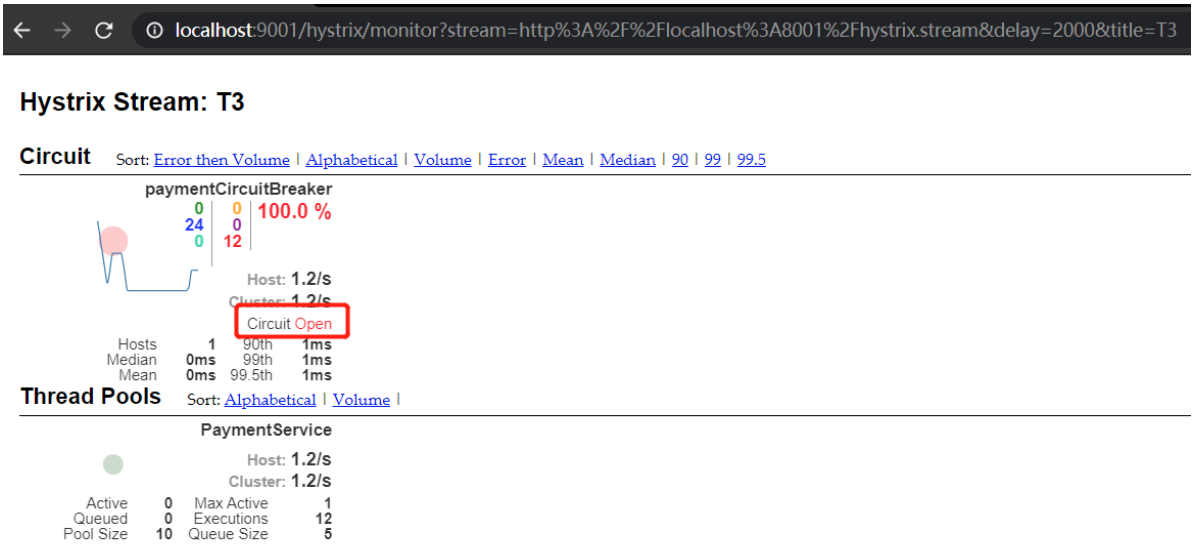


疯狂点，进行访问



然后请求错误的，使断路器打开





图的说明

整图说明


最近10s
错误百分比
成功数 1
熔断数 0
错误请求数 0
0
0
0.0 %
超时数
线程池拒绝数
失败/异常数
Host: 0.1/s
服务请求频率
Cluster: 0.1/s
Circuit Closed
断路状态