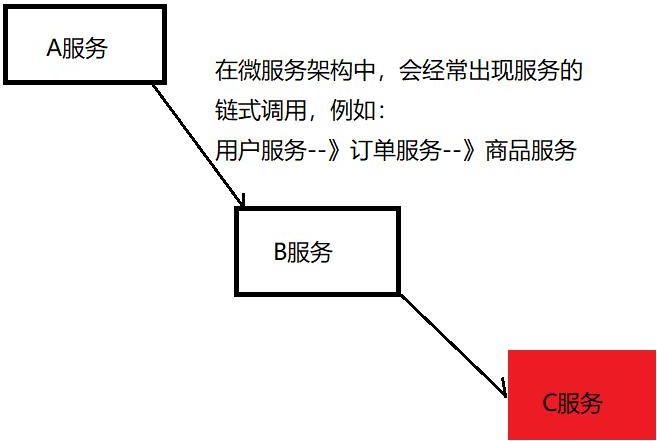
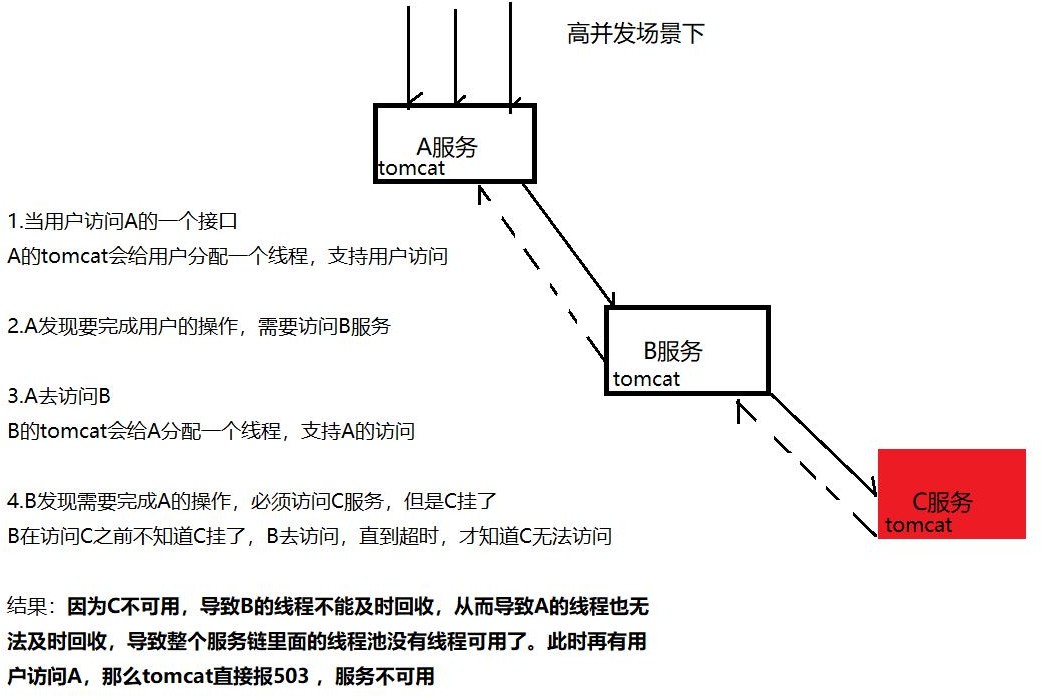
# 【Spring Cloud Hystrix】

## 前言

### 什么是服务雪崩





服务雪崩的本质：线程没有及时回收。

不管是调用成功还是失败，只要线程可以及时回收，就可以解决服务雪崩

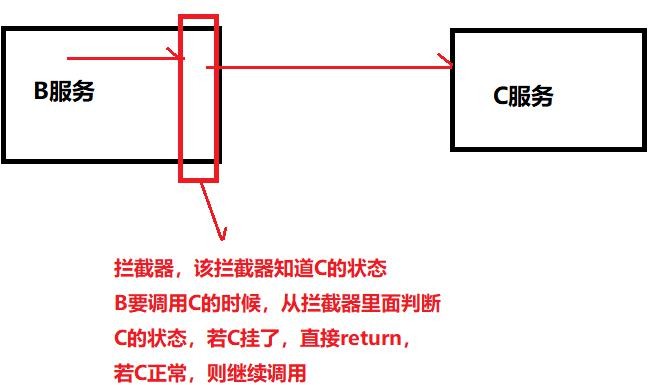
### 服务雪崩怎么解决

#### 修改调用的超时时长（不推荐）

将服务间的调用超时时长改小，这样就可以让线程及时回收，保证服务可用优点：非常简单，也可以有效的解决服务雪崩

缺点：不够灵活，有的服务需要更长的时间去处理（写库，整理数据）

* + 1. 设置拦截器



# Spring Cloud Hystrix 简介

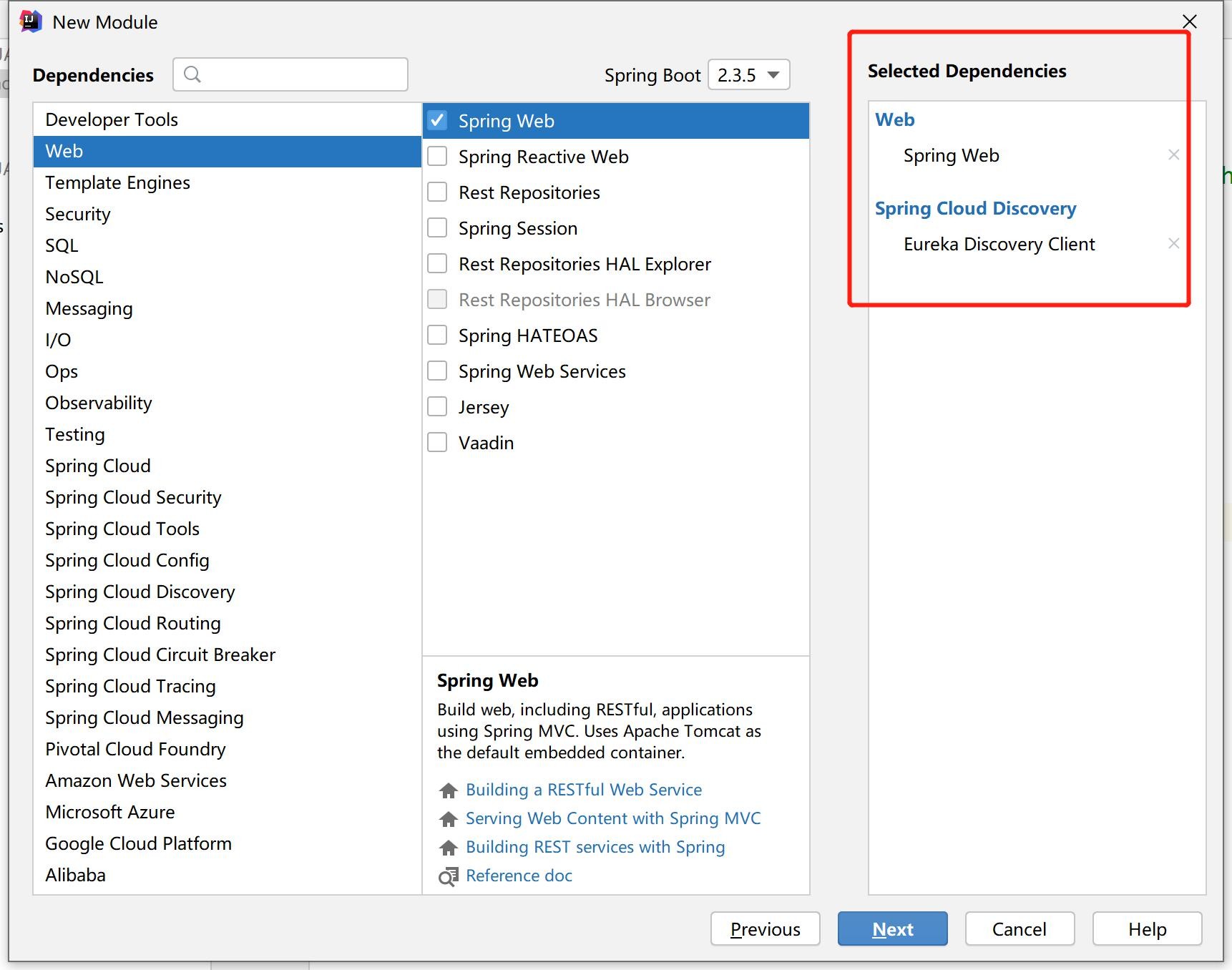
##### 熔断器，也叫断路器！（正常情况下 断路器是关的 只有出了问题才打开）用来保护微服务不雪崩的方法。思想和我们上面画的拦截器一样。

Hystrix 是 Netflix 公司开源的一个项目，它提供了熔断器功能，能够阻止分布式系统中出现联动故障。Hystrix 是通过隔离服务的访问点阻止联动故障的，并提供了故障的解决方案，从而提高了整个分布式系统的弹性。微博 弹性云扩容 Docker K8s

1. **Hystrix** 快速入门

##### 当有服务调用的时候，才会出现服务雪崩，所以 Hystrix 常和 OpenFeign，Ribbon 一起出现

* 1. 在 **OpenFeign** 中使用 **Hystrix**（重点）
     1. 启动 **provider-order-service**
        1. 先创建 **provider-order-service**，选择依赖



* + - 1. **provider-order-service** 修改配置文件

server:

port: 8082 spring:

application:

name: provider-order-service

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka instance:

instance-id: ${spring.application.name}:${server.port} prefer-ip-address: true

* + - 1. **provider-order-service** 修改启动类增加一个访问接口

package com.bjpowernode.controller;

import org.springframework.web.bind.annotation.GetMapping; import org.springframework.web.bind.annotation.RestController;

*/\*\**

* *@Author:* 北京动力节点

*\*/*

@RestController

public class OrderController {

*/\*\**

* + 订单服务下单接口

*\**

* + *@return*

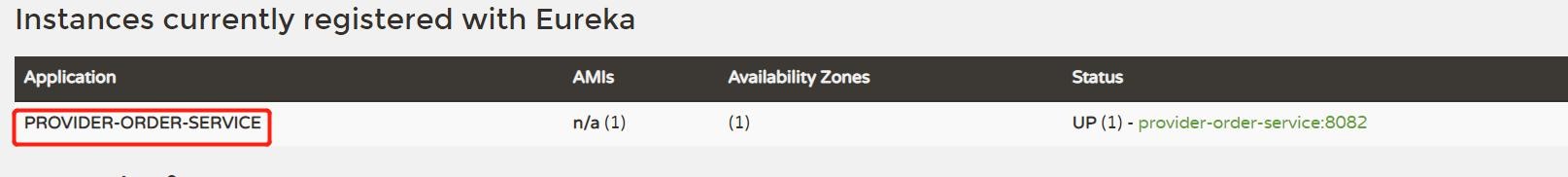
*\*/* @GetMapping("doOrder") public String doOrder() {

System.*out*.println("有用户来下单了"); return "下单成功";

}

}

* + - 1. **provider-order-service** 启动测试访问



* + 1. 修改 **consumer-user-service**
       1. 创建 **OrderServiceHystrix** 实现 **OrderServiceFeign**（代替方案）（上面是调用生产者成功，下面是失败

@Component

public class OrderServiceHystrix implements OrderServiceFeign { @Override

public String doOrder() {

System.*out*.println("调用下单服务失败，我走 hystrix 了");

return "我是 hystrix 的 doOrder，说明下单失败了";

}

......省略其他的实现方法

}

* + - 1. 修改 **OrderServiceFeign** 增加一个 **fallback**

@FeignClient(value = "provider-order-service", fallback =

OrderServiceHystrix.class)

* + - 1. 修改 **yml** 配置文件

feign:

hystrix:

enabled: true #开启断路器的使用

* + 1. 启动 **consumer-user-service** 访问测试



* + 1. 关掉 **provider-order-service** 访问测试



##### 说明 Hystrix 生效了

* 1. 在 **Ribbon** 中使用 **Hystrix**（了解）
     1. 启动 **eureka-server**
     2. 启动 **provider-order-service**
     3. 修改 **consumer-user-service**
        1. 1.消费者添加 **Hystrix** 的依赖，**ribbon** 没有集成 **hystrix**

**（2.给**CustomerRentFeign一个实现类

*/\*\*  
 \* 这里需要加入ioc容器  
 \*/*@Component  
public class CustomerRentFeignHystrix implements CustomerRentFeign {  
  
 */\*\*  
 \* 这个方法就是备选方案  
 \* @return  
 \*/* @Override  
 public String rent() {  
 return "我是备选方案";  
 }  
}

**3.给接口指定熔断走哪个类**fallback

@FeignClient(value = "rent-car-service",fallback = CustomerRentFeignHystrix.class)  
public interface CustomerRentFeign {  
  
 @GetMapping("rent")  
 public String rent();  
  
  
}

4.添加配置

feign:  
 hystrix:  
 enabled: true *# 在cloud的F版以前 是默认开启的 但是因为后来有了其他的熔断组件*

**就可以了。当消费者重新上线后，就不去调用备选方案了**

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

#### 修改启动类

@SpringBootApplication @EnableEurekaClient @EnableFeignClients

@EnableCircuitBreaker //开启断路器

public class ConsumerUserServiceApplication {

public static void main(String[] args) { SpringApplication.*run*(ConsumerUserServiceApplication.class, args);

}

* + - 1. 修改 **controller**

/\*\*

* 用户下单方法 ribbon 的熔断

\*

* @return
* @HystrixCommand(fallbackMethod = "ribbonHystrix")
* 指定熔断的方法

\*/ @GetMapping("userDoOrderRibbon")

@HystrixCommand(fallbackMethod = "ribbonHystrix") public String testRibbonHystrix(String serviceId) {

String result = restTemplate.getForObject("http:" + serviceId + "/doOrder", String.class); System.*out*.println(result);

return "成功";

}

//方法签名要和原来的方法一致

public String ribbonHystrix(String serviceId) { return "我是ribbon 的备选方案";

}

* + - 1. 启动 **consumer** 测试

http://localhost:8081/userDoOrderRibbon?serviceId=provider-order-service

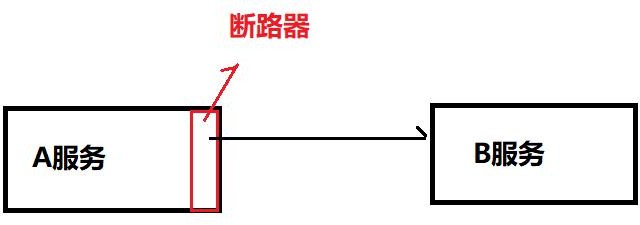


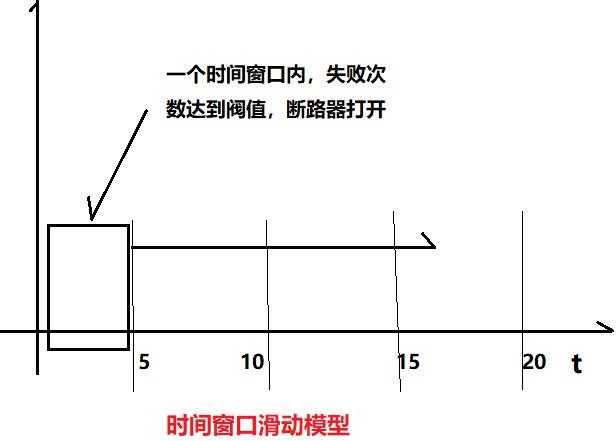
* + - 1. 关掉 **provider** 测试



## 手写断路器

### 断路器的设计





* 1. 断路器的状态说明以及状态转变

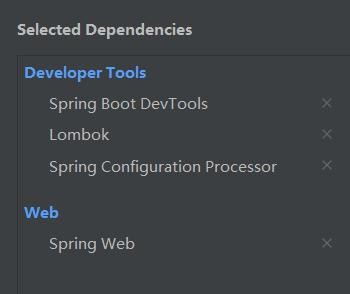
关：服务正常调用 A---》B 默认关闭，前几次不知道是否 能调用成功，也是要去调用的

开：在一段时间内，调用失败次数达到阀值（5s 内失败 3 次）（**5s** 失败 **30** 次的）则断路器打开，直接 return

半开：断路器打开后，过一段时间，让少许流量尝试调用 B 服务，如果成功则断路器关闭，使服务正常调用，如果失败，则继续半开

### 开始设计断路器模型

#### 创建项目选择依赖



* + 1. 创建断路器状态模型 **HystrixStatus**

public enum HystrixStatus {

*OPEN*(0, "打开"), *CLOSE*(1, "关闭"), *HALF\_OPEN*(2, "半开");

HystrixStatus(Integer status, String desc) {

}

}

* + 1. 创建断路器 **Hystrix**

@Data

public class Hystrix {

/\*\*

* 断路器状态：默认是关闭的

\*/

private HystrixStatus status = HystrixStatus.CLOSE;

/\*\*

* 断路器的窗口时间，多少时间内出现问题

\*/

private static final long *WINDOWS\_SLEEP\_TIME* = 5L;

/\*\*

\* 最大失败次数，阀值

\*/

private static final int *MAX\_FAIL\_COUNT* = 3;

/\*\*

\* 当前失败的次数

\*/

private AtomicInteger currentFailCount = new AtomicInteger(0);

/\*\*

\* 锁对象

\*/

public Object *lock* = new Object();

/\*\*

\* 创建线程池用于计数和清除失败次数

\*/

private ThreadPoolExecutor threadPool = new ThreadPoolExecutor( 2,

5,

3,

TimeUnit.*SECONDS*,

new LinkedBlockingQueue<>(), Executors.*defaultThreadFactory*(), new ThreadPoolExecutor.AbortPolicy()

);

//如何实现每个 5s 内 统计到失败次数达到阀值呢？

// 我们反向思考，每 5s 就清空断路器的统计次数，这样就可以了

{

threadPool.execute(() -> {

//死循环

while (true) { try {

//进来先睡几秒 TimeUnit.*SECONDS*.sleep(5);

//睡了五秒以后呢？就清零吗？还要判断断路器状态是否是关闭的

if (this.getStatus() == HystrixStatus.CLOSE) {

//如果该断路器状态是关闭的，说明规定时间没 没有达到阀值，就清零

this.currentFailCount.set(0);

} else {

//此时线程在这里运行没有意义，我们让他等待，释放掉锁 synchronized (*lock*) {

*lock*.wait();

//当半开调用成功以后，线程被唤醒了，往下执行，又开始了循环统计了 System.*out*.println("测试调用成功，我们统计线程再次启动");

}

}

} catch (InterruptedException e) { e.printStackTrace();

}

}

});

}

/\*\*

* 描述: 失败后增加次数，以及修改断路器状态和重置失败次数

\*

* @param :
* @return void

\*/

public void addFallCount() {

//获取失败的次数

int fallCount = this.currentFailCount.incrementAndGet(); if (fallCount >= *MAX\_FAIL\_COUNT*) {

//如果失败的次数超过了阀值，则断路器打开 this.setStatus(HystrixStatus.OPEN);

//开启一个线程，过 5s 去把当前断路器状态改为半开

threadPool.execute(() -> { try {

TimeUnit.*SECONDS*.sleep(*WINDOWS\_SLEEP\_TIME*); this.setStatus(HystrixStatus.HALF\_OPEN);

//清空失败次数 this.currentFailCount.set(0);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

}

}

}

#### 引入切面类比拦截器

<dependency>

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

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

</dependency>

* + 1. 创建 **HystrixAspect**

@Component @Aspect

public class HystrixAspect {

/\*\*

* Aop 切入点

\*/

public static final String *POINTCUT* = "execution(\* com.bjpowernode.controller.TestController.testRpc(..))";

/\*\*

* key=哪个服务
* value=该服务提供者对应的断路器

\*/

private static Map<String, Hystrix> hystrixs = new HashMap<>(2);

static {

hystrixs.put("provider", new Hystrix());

}

/\*\*

* 随机数，用于产生少许流量

\*/

private Random random = new Random();

/\*\*

* 环绕通知，类比拦截器

\*

* @param point
* @return

\*/

@Around(value = *POINTCUT*)

public Object HystrixInterceptor(ProceedingJoinPoint point) {

//先得到该服务的熔断器

Hystrix hystrix = hystrixs.get("provider");

Object proceed = null;

//执行调用前先判断断路器的状态 switch (hystrix.getStatus()) {

case CLOSE:

//断路器关闭，则执行远程调用 try {

proceed = point.proceed(); return proceed;

} catch (Throwable throwable) {

//远程调用失败

//记录次数 hystrix.addFallCount(); proceed = "我是备胎"; return proceed;

}

case OPEN:

//断路器打开,直接返回 proceed = "我是备胎"; break;

case HALF\_OPEN:

//断路器半开，用少许的流量(20%)去远程调用 int i = random.nextInt(5); System.*out*.println(i);

if (i == 1) { try {

//去访问

proceed = point.proceed();

//成功 把断路器关掉 hystrix.setStatus(HystrixStatus.CLOSE); synchronized (Hystrix.lock) {

//锁住，唤醒计数器线程开始计数 Hystrix.lock.notifyAll();

}

//返回

return proceed;

} catch (Throwable throwable) { System.*out*.println("少许流量调用失败");

}

}

proceed = "我是备胎"; break;

default:

}

return proceed;

}

}

* + 1. 创建 **TestController** 去测试

@RestController

public class TestController {

@Autowired

private RestTemplate restTemplate;

@RequestMapping("testRpc") public String testRpc() {

String result = restTemplate.getForObject("http://localhost:8082/info", String.class); System.*out*.println(result);

return "调用成功";

}

}

1. **Hystrix** 的常用配置

server:

port: 8081 spring:

application:

name: consumer-user-service

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/ fetch-registry: true

register-with-eureka: true instance:

instance-id: ${spring.application.name}:${server.port} prefer-ip-address: true

feign:

hystrix:

enabled: true

hystrix: #hystrix 的全局控制

command:

default: #default 是全局控制，也可以换成单个方法控制，把 default 换成方法名即可

fallback:

isolation:

semaphore:

maxConcurrentRequests: 1000 #信号量隔离级别最大并发数 circuitBreaker:

enabled: true #开启断路器 requestVolumeThreshold: 3 #失败次数（阀值）

sleepWindowInMilliseconds: 20000 #窗口时间

errorThresholdPercentage: 60 #失败率 execution:

isolation:

Strategy: thread #隔离方式 thread 线程隔离集合和SEMAPHORE 信号量隔离

级别

thread:

timeoutInMilliseconds: 3000 #调用超时时长

ribbon:

ReadTimeout: 5000 #要结合feign 的底层 ribbon 调用的时长

ConnectTimeout: 5000

#隔离方式 两种隔离方式 thread 线程池 按照 group（10 个线程）划分服务提供者，用户请求的线程和做远程的线程不一样

# 好处 当 B 服务调用失败了 或者请求 B 服务的量太大了 不会对 C 服务造成影响 用户访问比较大的情况下使用比较好 异步的方式

# 缺点 线程间切换开销大，对机器性能影响

# 应用场景 调用第三方服务 并发量大的情况下

# SEMAPHORE 信号量隔离 每次请进来 有一个原子计数器 做请求次数的++ 当请求完成以后 -- # 好处 对 cpu 开销小

# 缺点 并发请求不易太多 当请求过多 就会拒绝请求 做一个保护机制

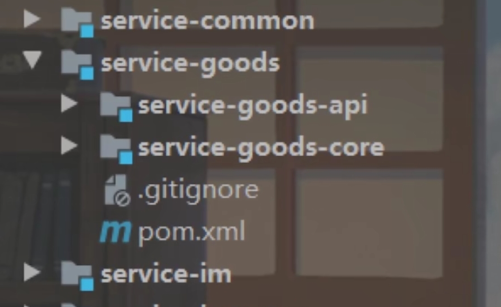
# 场景 使用内部调用 ，并发小的情况下

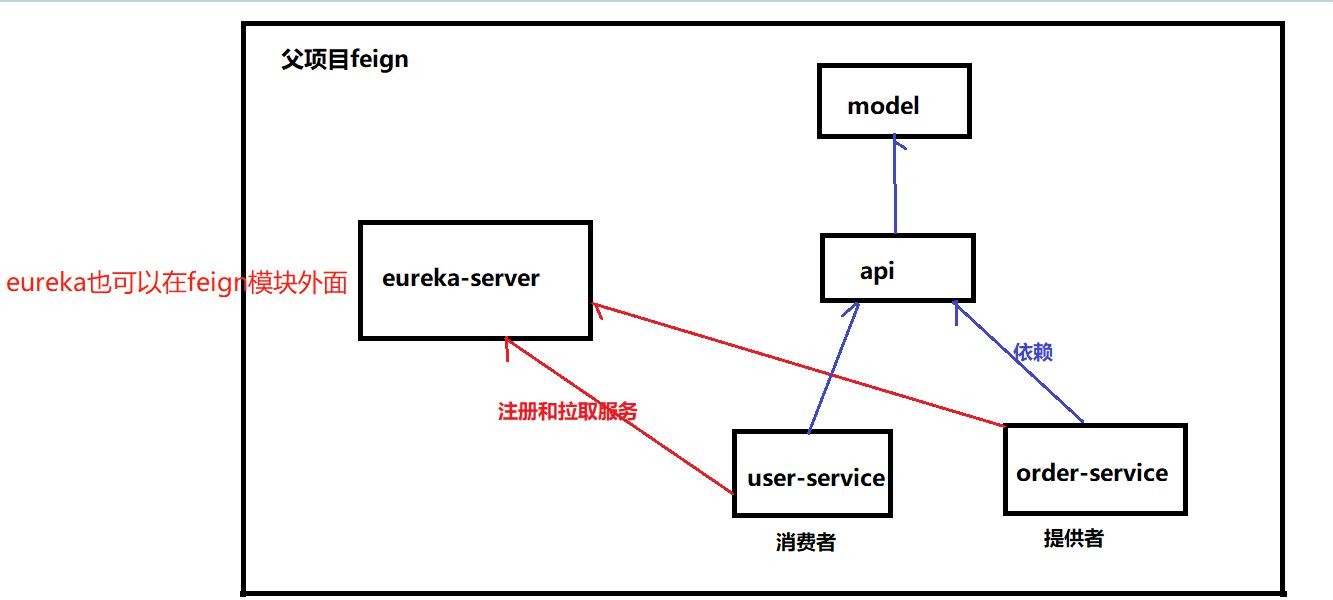
# 源码入门 HystrixCommand AbstractCommand HystrixThreadPool

1. **Feign** 的工程化实例

##### eureka ribbon openfeign hystrix

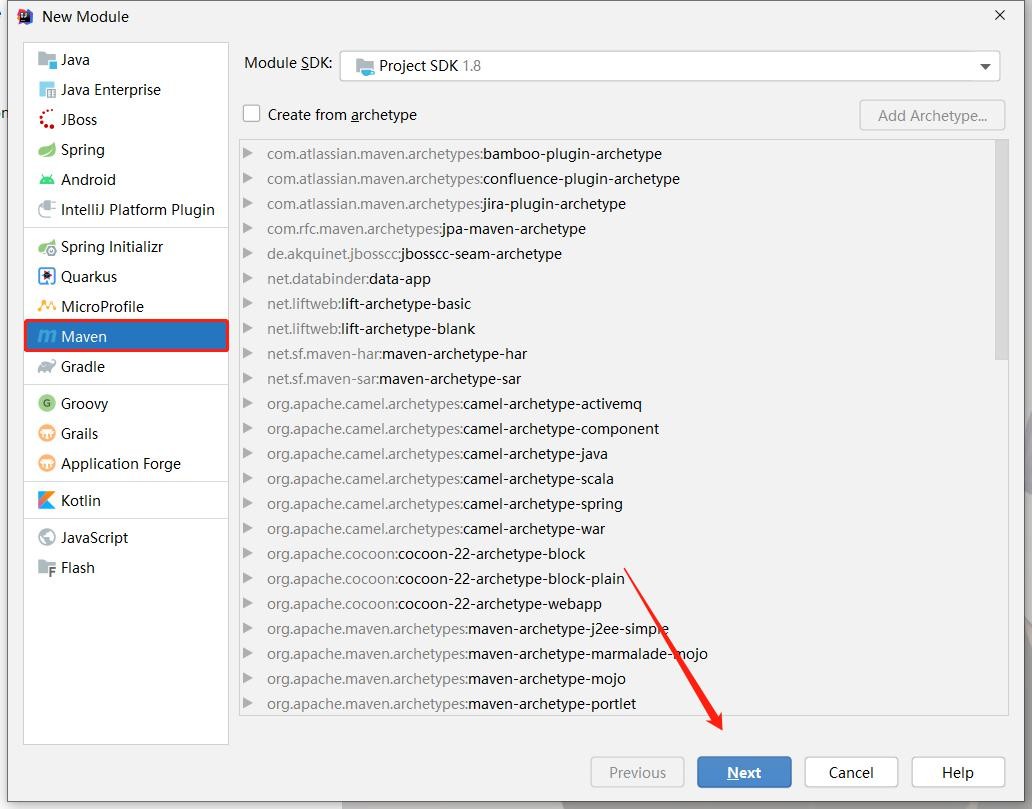
一个项目设计方案。api放一些对外的接口实体啥的，core则放实际业务。当然还有其他方案，如所有接口实体都放一个模块







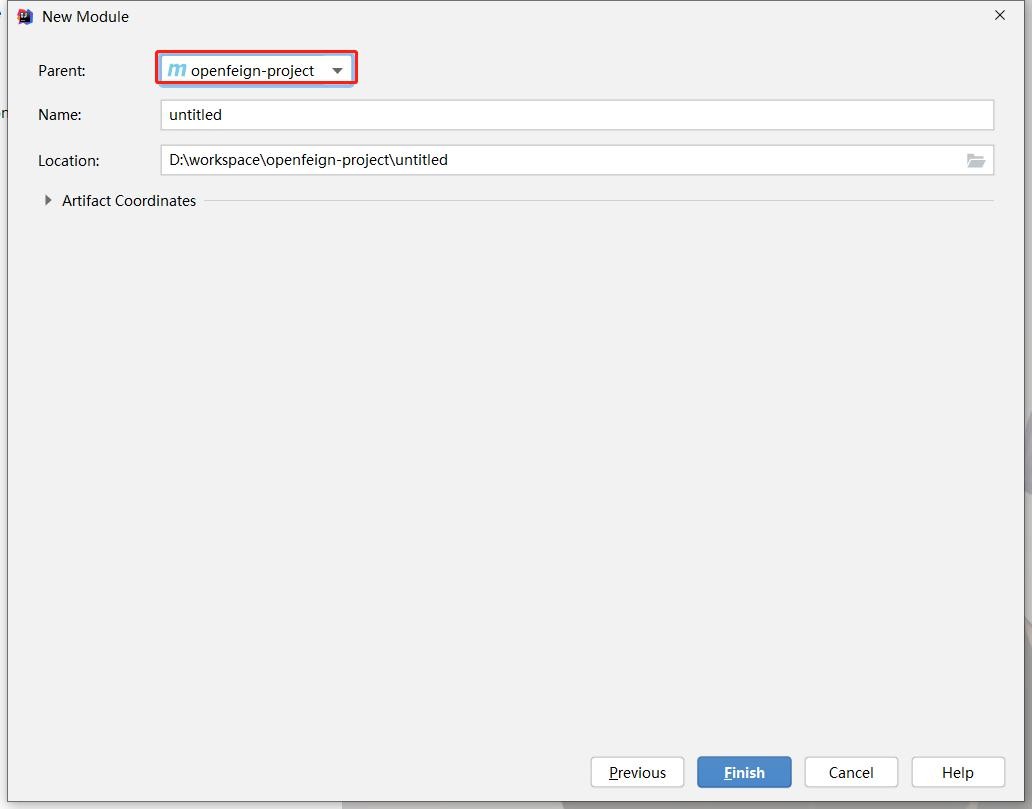
* 1. 创建父项目 **feign**



* 1. 创建子 **module**

##### Consumer-user-service 消费者 Provider-order-service 提供者 Model 公共实体类

Provider-api 消费者接口



* 1. 父项目 **feign** 的 **pom.xml**

*<?*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>

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

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

<version>2.3.3.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<modelVersion>4.0.0</modelVersion>

<groupId>org.bjpowernode</groupId>

<artifactId>feign</artifactId>

<packaging>pom</packaging>

<version>1.0-SNAPSHOT</version>

<modules>

<module>consumer-user-service</module>

<module>provider-order-service</module>

<module>provider-api</module>

<module>model</module>

</modules>

<properties>

<java.version>1.8</java.version>

<spring-cloud.version>Hoxton.SR8</spring-cloud.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

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

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

<version>2.2.4.RELEASE</version>

</dependency>

</dependencies>

</dependencyManagement>

</project>

* 1. **Provider-order-service** 的 **controller**

直接实现 OrderServiceFeign 注解也会被一起带过来

@RestController

public class OrderController **implements OrderServiceFeign** {

@Override

public String doOrder() {

System.*out*.println("provider-order-service 的下订单");

return "下单 ok";

}

@Override

public BaseResult addOrder(List<Order> orders) { return null;

}

@Override

public BaseResult addOrder2(Order orders) { return null;

}

@Override

public BaseResult getOneOrder(String orderId) { return null;

}

@Override

public BaseResult getAllOrder(String userId, Integer page, Integer size)

{

return null;

}

@Override

public String test() { return null;

}

@Override

public String apiTest() { return null;

}

}

* 1. 新增配置文件以及启动类，测试即可