**【Spring Cloud Hystrix】**

**【主要内容】**

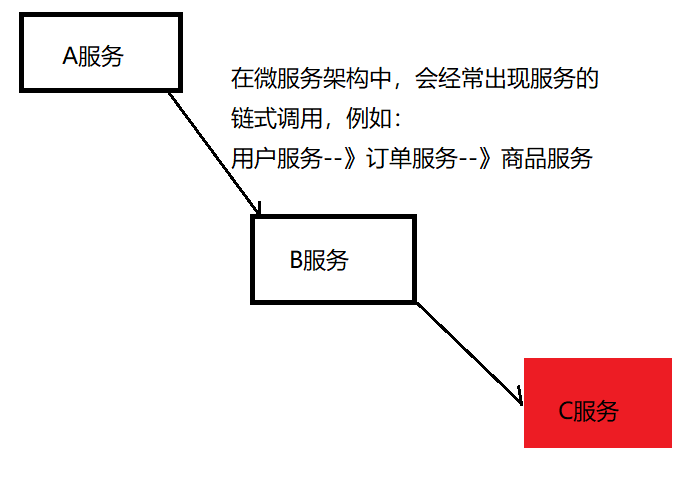
1. 服务雪崩原因
2. Hystrix简介
3. Hystrix快速入门
4. 手写断路器
5. Hystrix常用配置

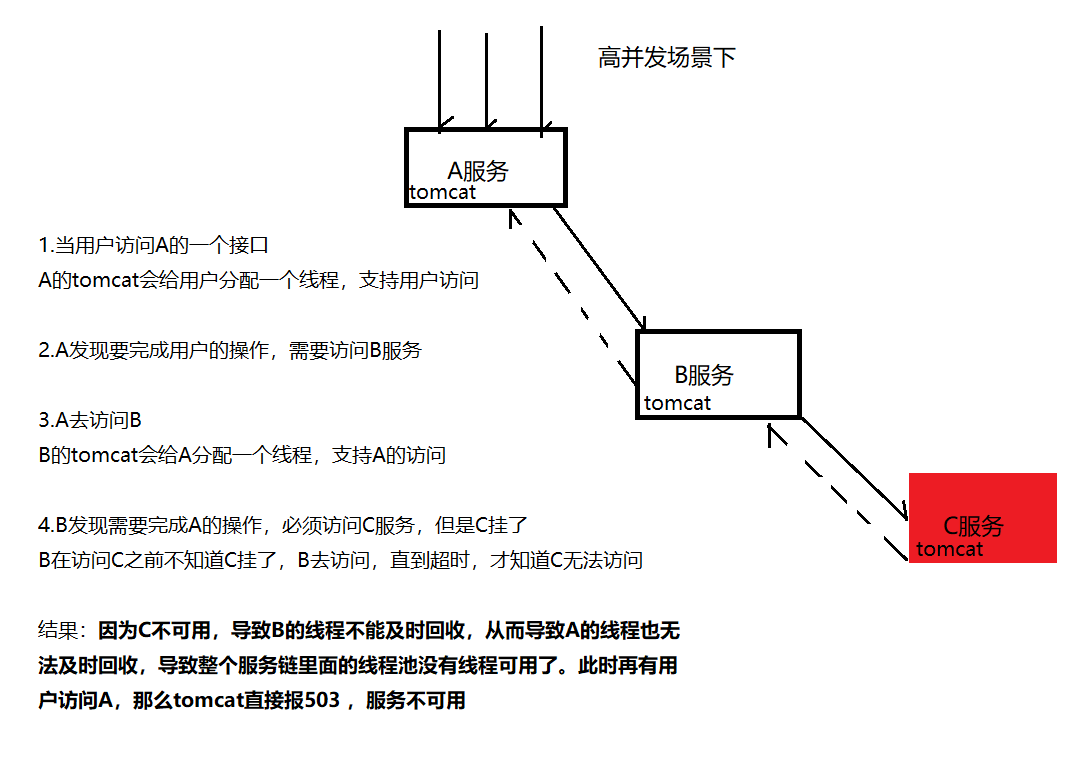
**【学习目标】**

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 服务雪崩原因 | 了解 |
| Hystrix简介 | 掌握 |
| Hystrix快速入门 | 掌握 |
| 手写断路器 | 掌握 |
| Hystrix常用配置 | 掌握 |

# 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

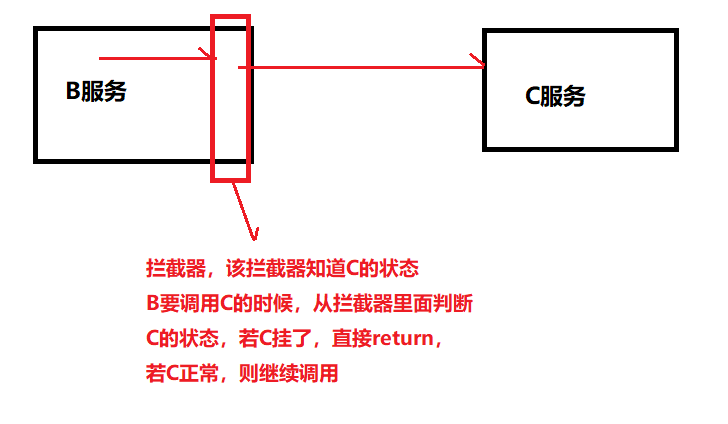
### 修改调用的超时时长

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）

### 设置拦截器



# Spring Cloud Hystrix简介

熔断器，也叫断路器！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。

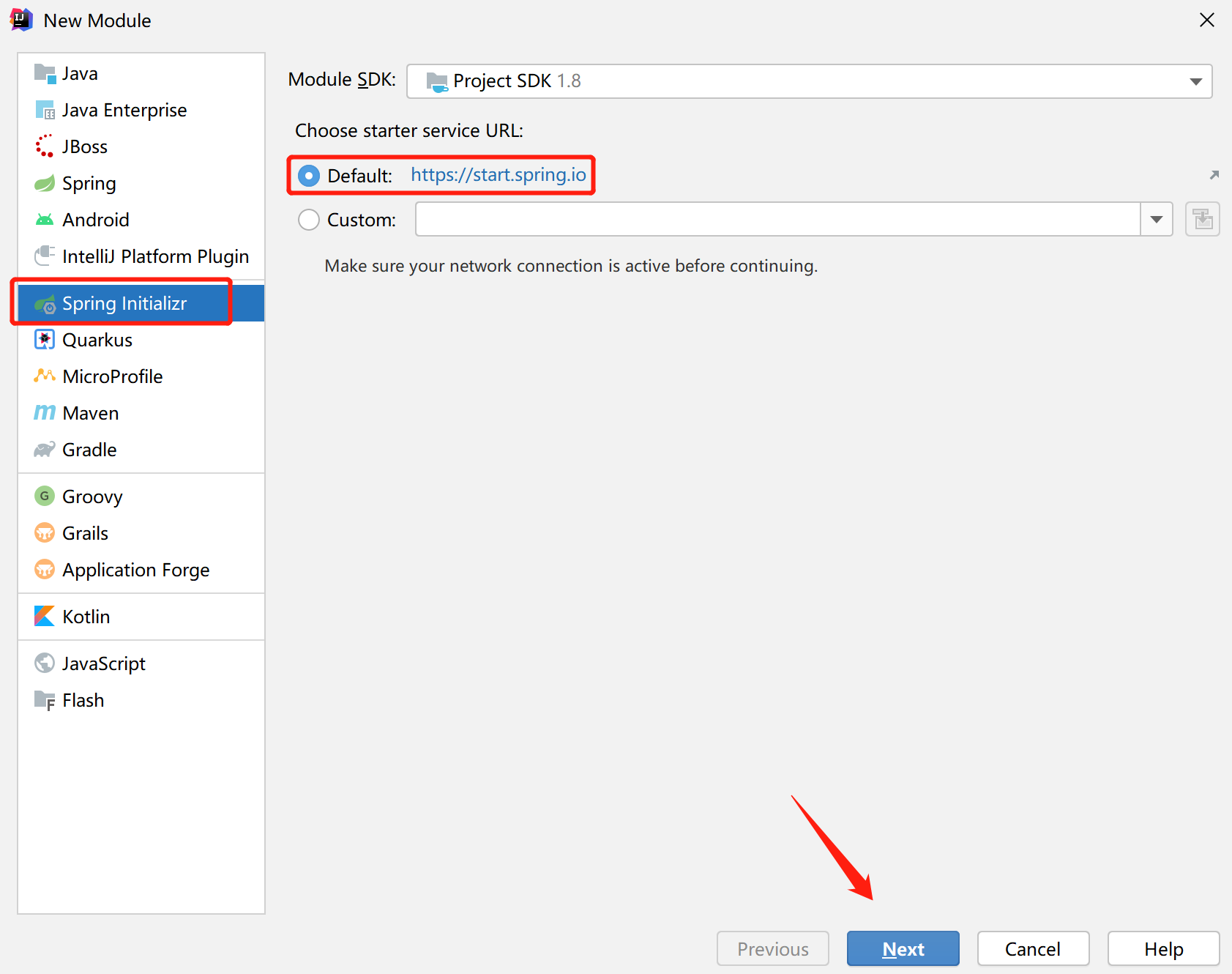
# Hystrix快速入门

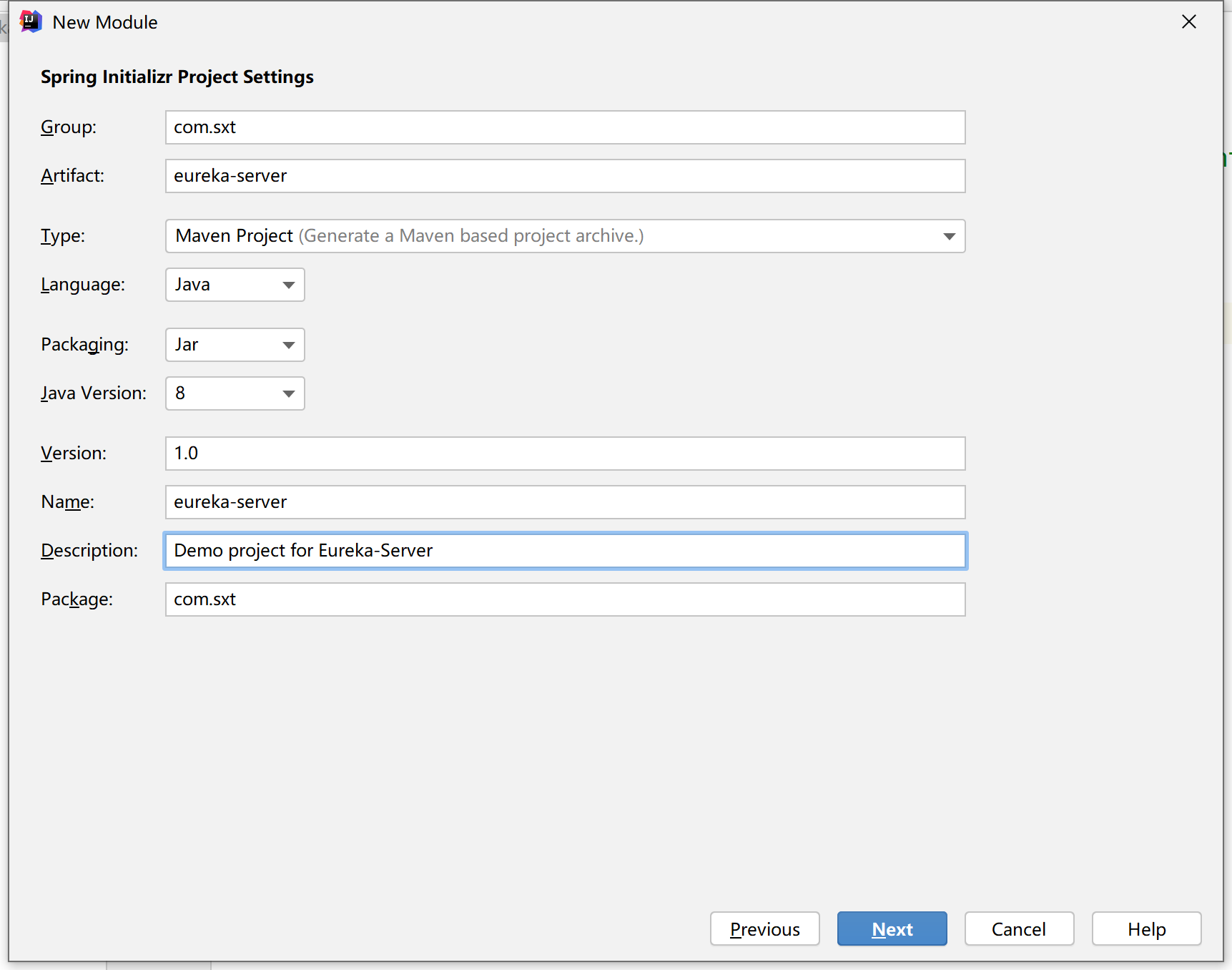
当有服务调用的时候，才会出现服务雪崩，所以Hystrix常和OpenFeign，Ribbon一起出现

## 在OpenFeign中使用Hystrix（重点）

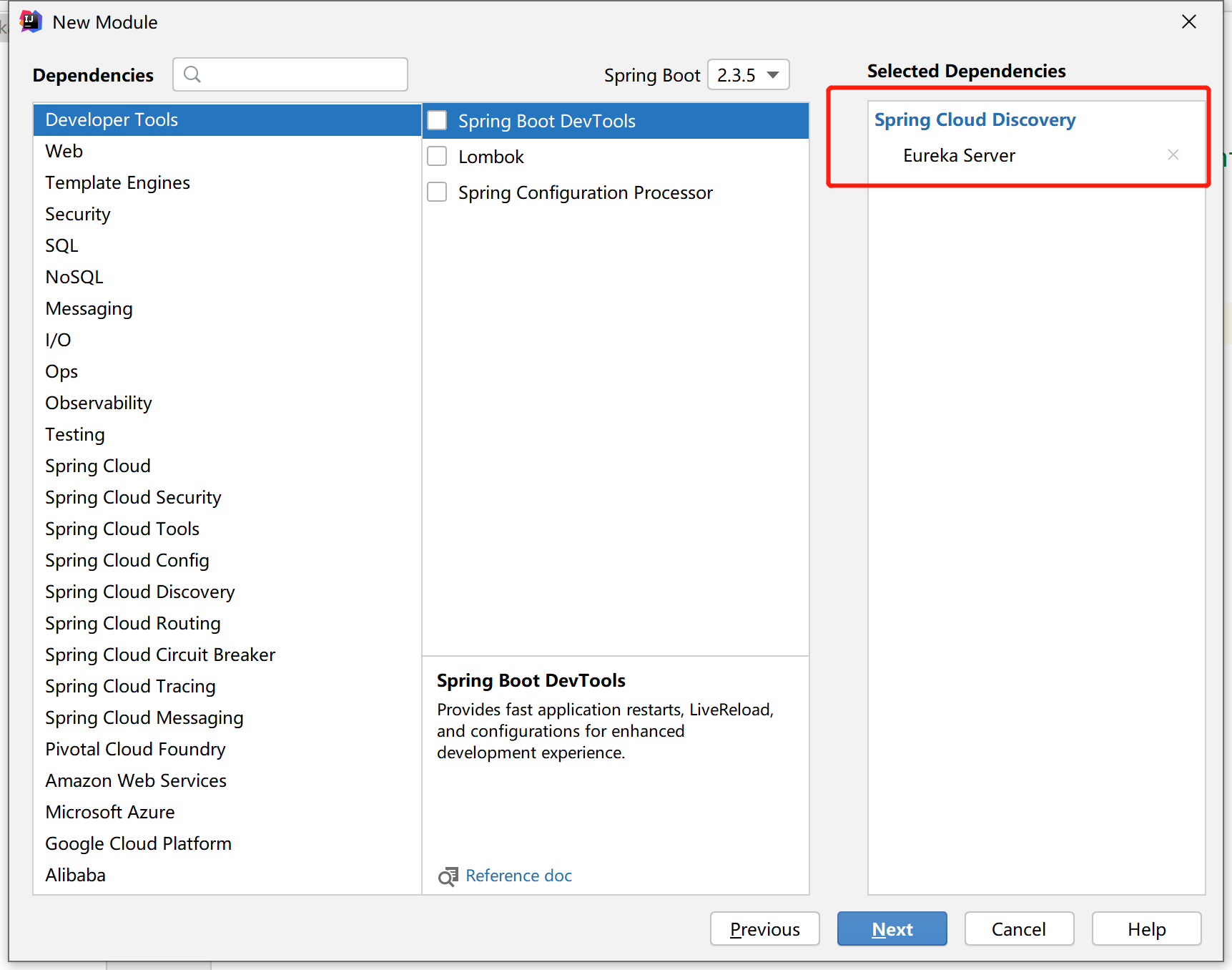
### 启动eureka-server

#### 创建项目





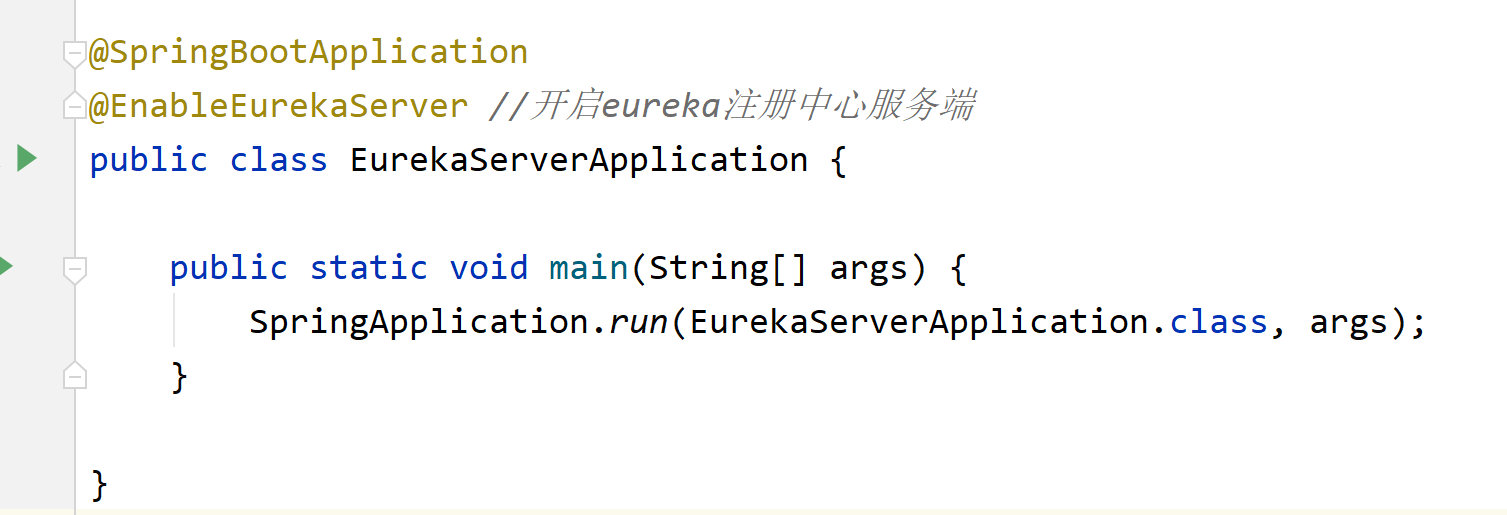
#### 选择依赖



#### 分析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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  *<!-- 实质还是springboot项目-->* <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.3.4.RELEASE</version>  <relativePath/> *<!-- lookup parent from repository -->* </parent>  <groupId>com.sxt</groupId>  <artifactId>eureka-server</artifactId>  <version>1.0</version>  <name>eureka-server</name>  <description>Demo project for Eureka-Server</description>   <properties>  <java.version>1.8</java.version>  *<!-- 这里控制了springcloud的版本-->* <spring-cloud.version>Hoxton.SR8</spring-cloud.version>  </properties>   <dependencies>  *<!-- eureka注册中心的服务端-->* <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  </dependency>   <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  <exclusions>  <exclusion>  <groupId>org.junit.vintage</groupId>  <artifactId>junit-vintage-engine</artifactId>  </exclusion>  </exclusions>  </dependency>  </dependencies>  *<!-- 依赖管理，cloud的依赖-->* <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>  </dependencies>  </dependencyManagement>   <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

#### 修改启动类

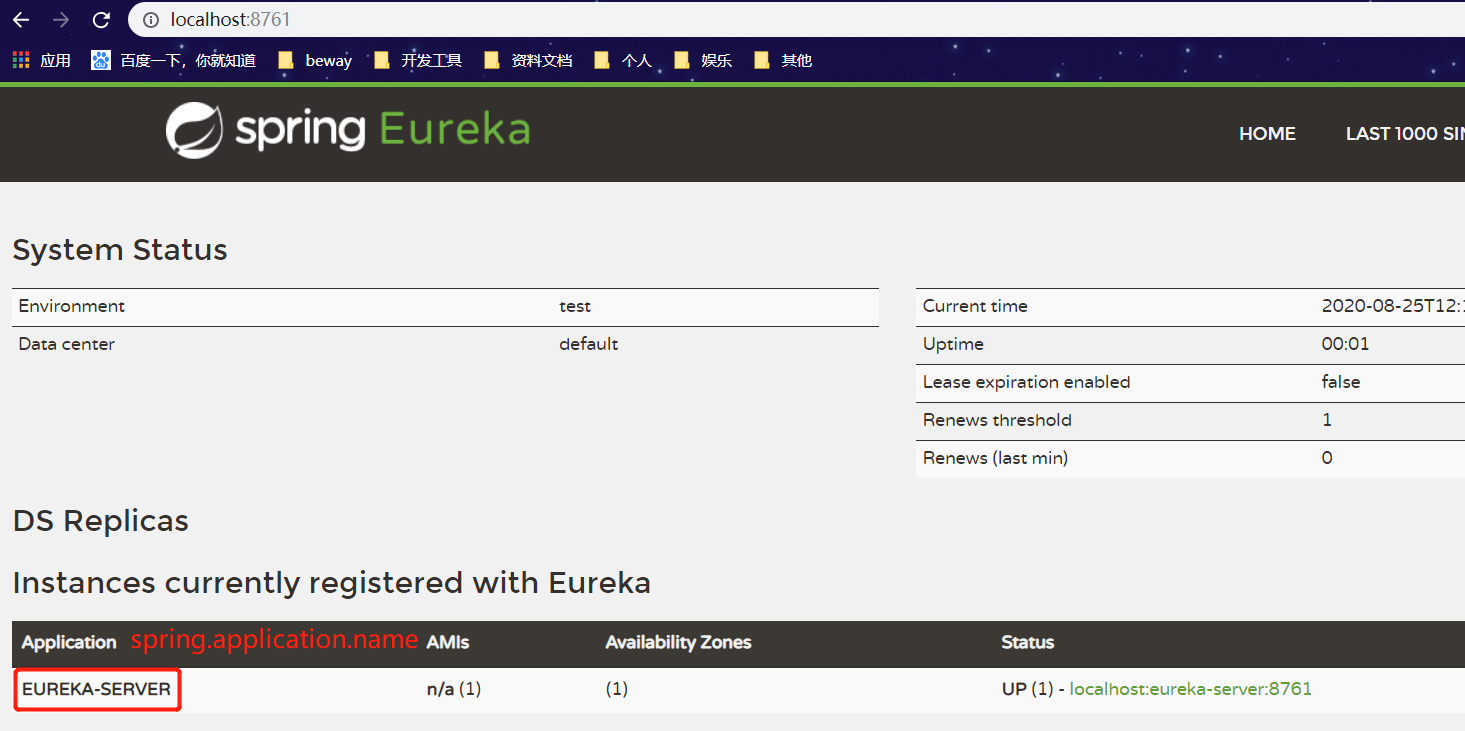


#### 修改配置文件

|  |
| --- |
| server:  port: 8761 spring:  application:  name: eureka-server *#服务名称* |

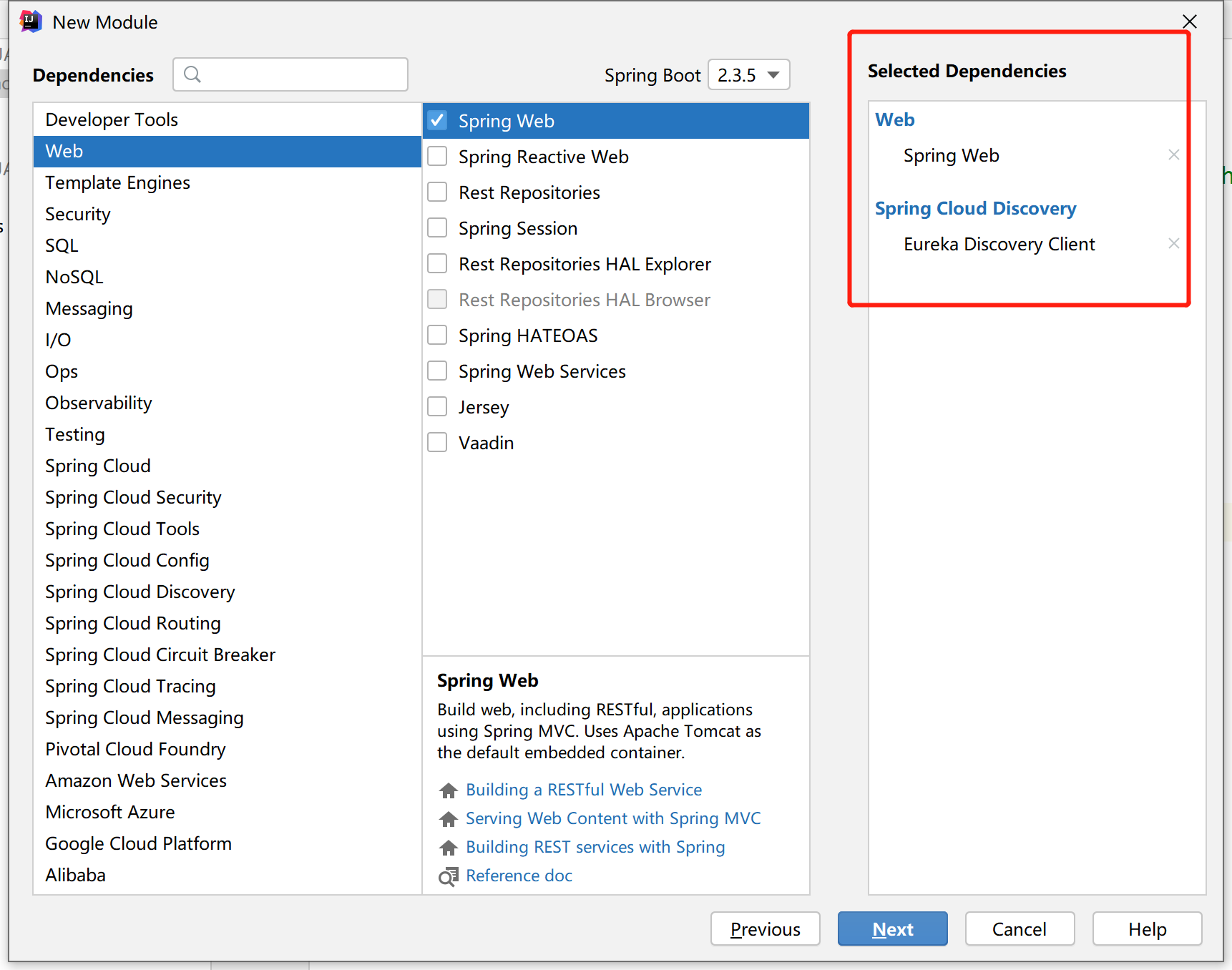
#### 访问测试

<http://localhost:8761>



### 启动provider-order-service

#### 先创建provider-order-service，选择依赖



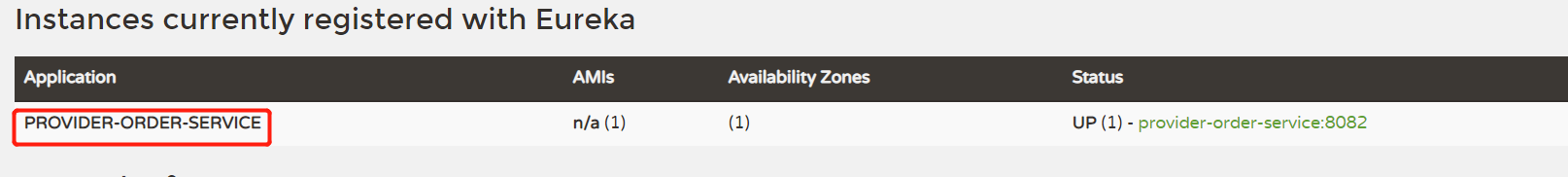
#### provider-order-service修改配置文件

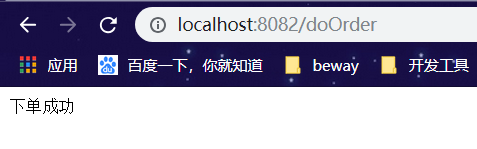
|  |
| --- |
| server:  port: 8082 spring:  application:  name: provider-order-service eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true |

#### provider-order-service修改启动类增加一个访问接口

|  |
| --- |
| package com.sxt.controller;  import org.springframework.web.bind.annotation.GetMapping; import org.springframework.web.bind.annotation.RestController;  */\*\*  \* @Author: 武汉尚学堂  \*/* @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";  } } |

#### provider-order-service启动测试访问





### 修改consumer-user-service

#### 创建OrderServiceHystrix实现OrderServiceFeign（代替方案）

|  |
| --- |
| @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 } |

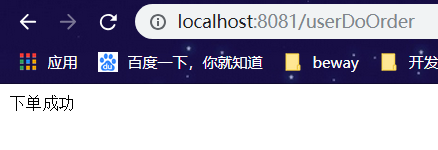
#### 修改OrderServiceFeign增加一个fallback

|  |
| --- |
| @FeignClient(value = "provider-order-service", fallback = OrderServiceHystrix.class) |

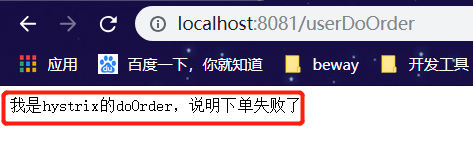
#### 修改yml配置文件

|  |
| --- |
| feign:  hystrix:  enabled: true #开启断路器的使用 |

### 启动consumer-user-service访问测试



### 关掉provider-order-service访问测试



说明Hystrix生效了

## 在Ribbon中使用Hystrix（了解）

### 启动eureka-server

### 启动provider-order-service

### 修改consumer-user-service

#### 添加Hystrix的依赖，ribbon没有集成hystrix

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

#### 修改启动类

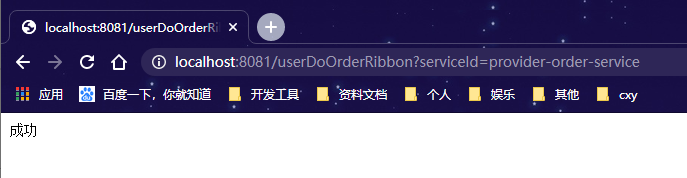
|  |
| --- |
| @SpringBootApplication @EnableEurekaClient @EnableFeignClients @EnableCircuitBreaker //开启断路器 public class ConsumerUserServiceApplication {   public static void main(String[] args) {  SpringApplication.*run*(ConsumerUserServiceApplication.class, args);  } |

#### 修改controller

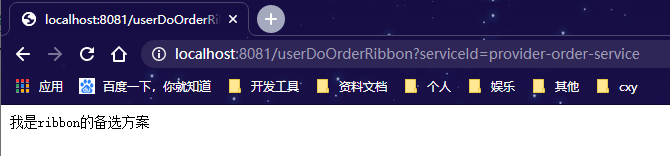
|  |
| --- |
| /\*\*  \* 用户下单方法 ribbon的熔断  \*  \* @return  \* @HystrixCommand(fallbackMethod = "ribbonHystrix")  \* 指定熔断的方法  \*/ @GetMapping("userDoOrderRibbon") @HystrixCommand(fallbackMethod = "ribbonHystrix") public String testRibbonHystrix(String serviceId) {  String result = restTemplate.getForObject("http:" + 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"; } |

#### 启动consumer测试

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

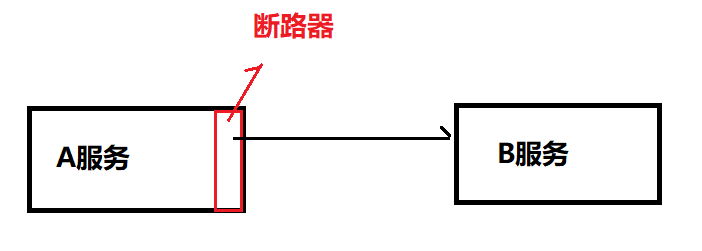


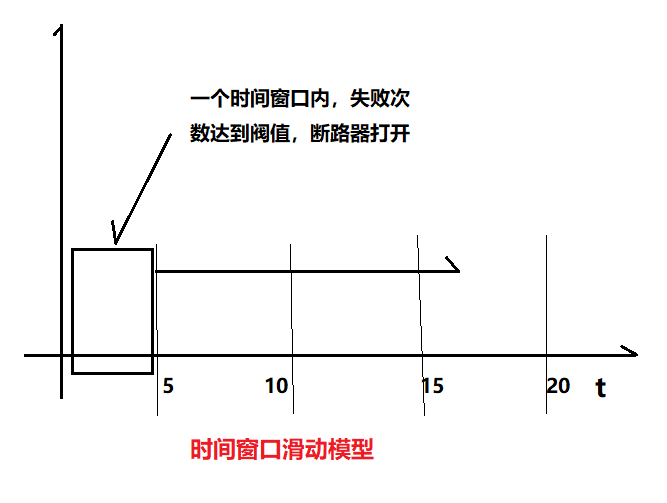
#### 关掉provider测试



# 手写断路器

## 断路器的设计





## 断路器的状态说明以及状态转变

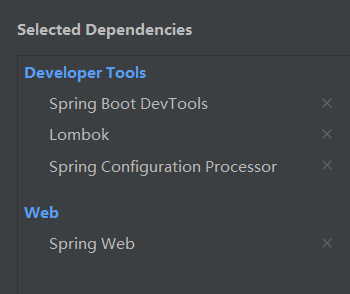
**关：服务正常调用 A---》B**

**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 （随机数）**

## 开始设计断路器模型

### 创建项目选择依赖



### 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) {  } } |

### 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> |

### 创建HystrixAspect

|  |
| --- |
| @Component @Aspect public class HystrixAspect {   /\*\*  \* Aop切入点  \*/  public static final String *POINTCUT* = "execution(\* com.sxt.controller.TestController.testRpc(..))";   /\*\*  \* key=哪个服务  \* value=该服务提供者对应的断路器  \*/  private Map<String, Hystrix> hystrixs = new HashMap<>(2);   {  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;  } } |

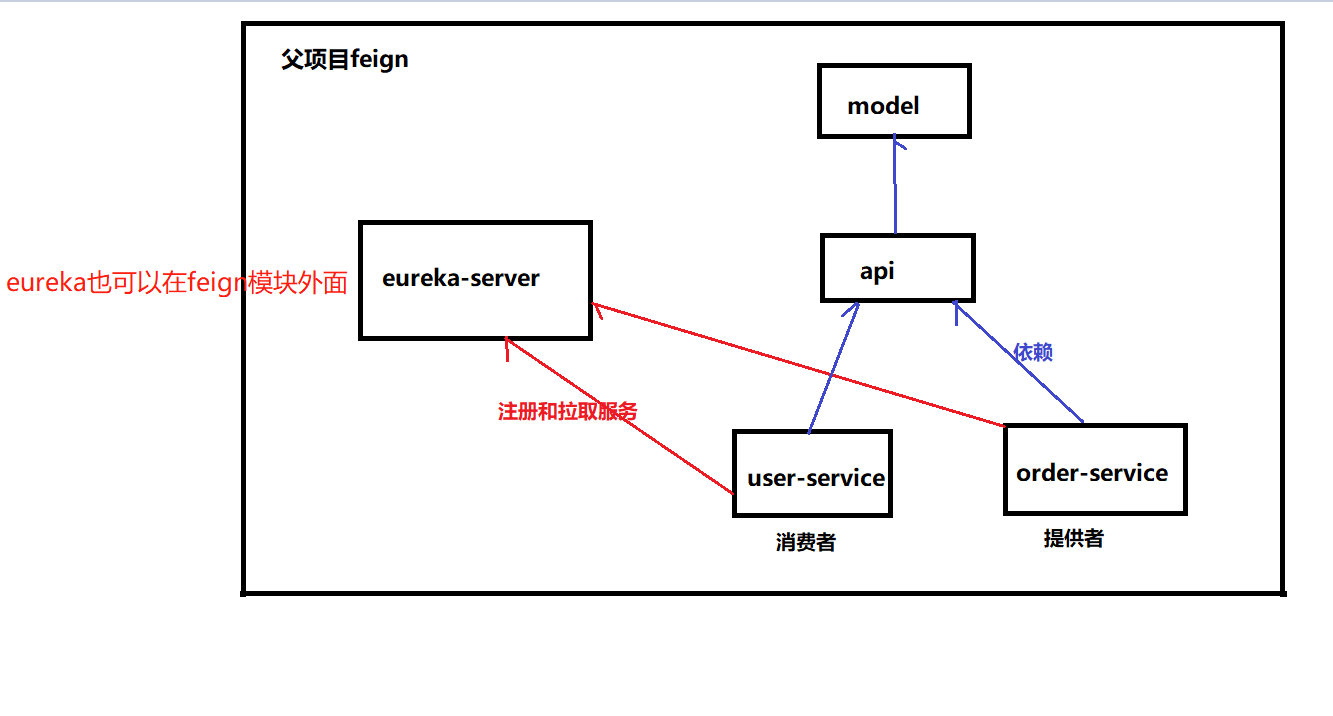
### 创建TestController去测试

|  |
| --- |
| @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";  } } |

# Hystrix的常用配置

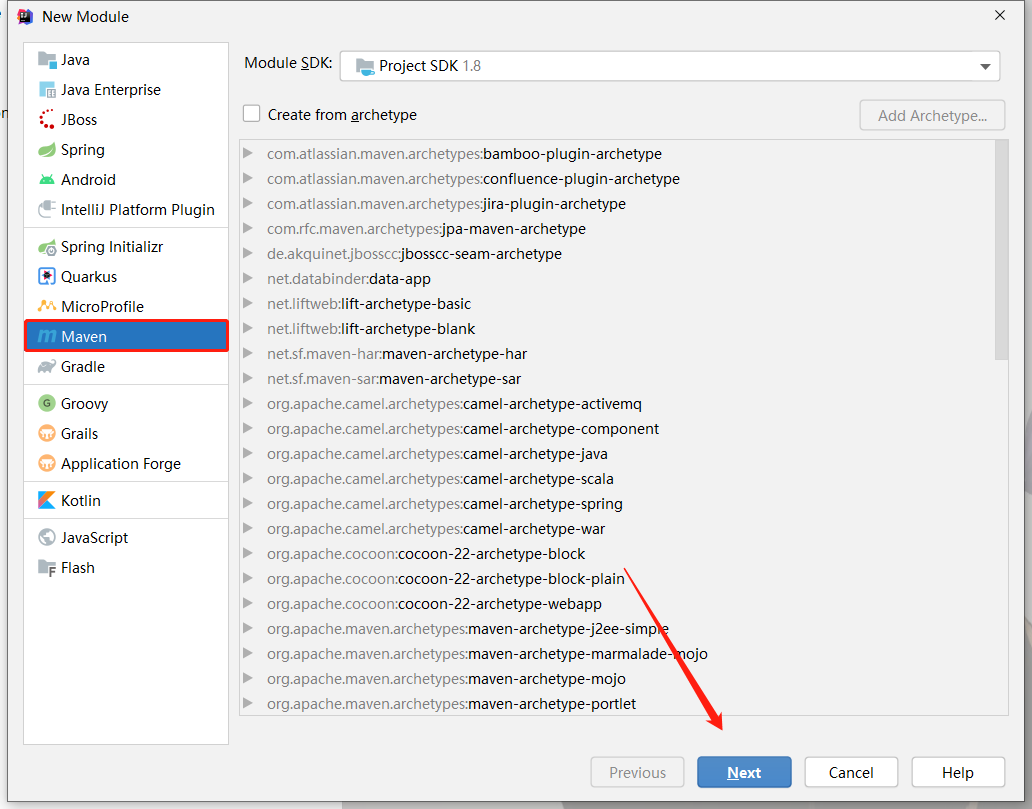
|  |
| --- |
| server:  port: 8081 spring:  application:  name: consumer-user-service eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka/  fetch-registry: true  register-with-eureka: true  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true 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 |

# Feign的工程化实例





## 创建父项目feign



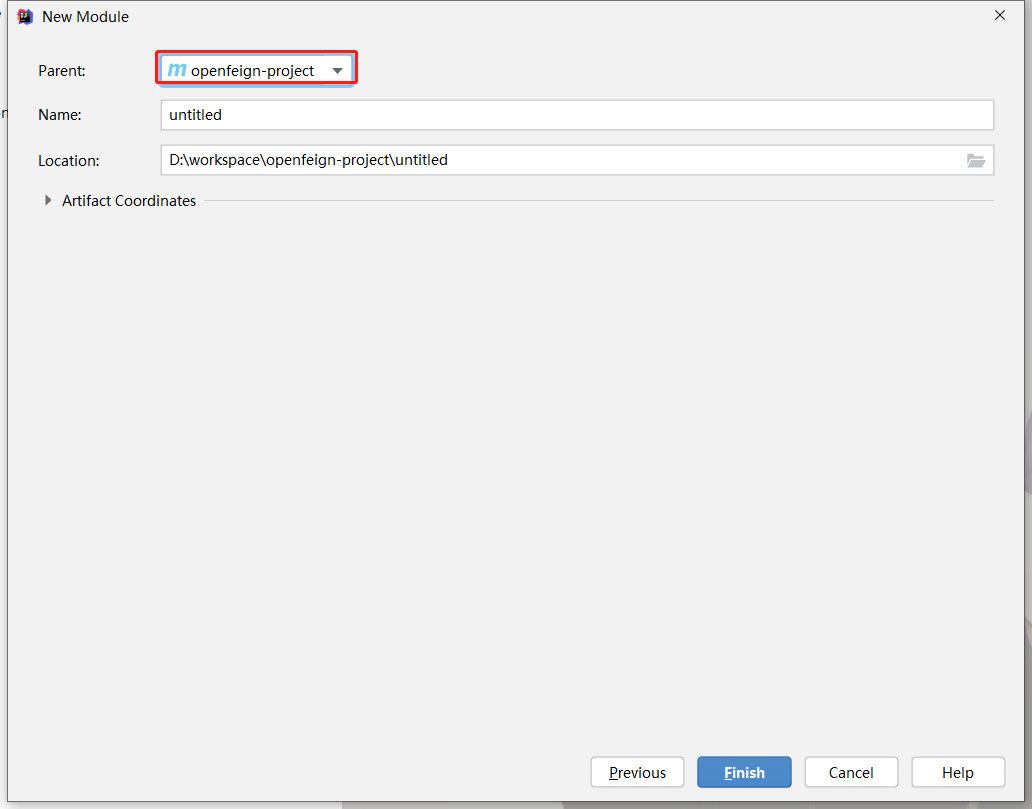
## 创建子module

**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**



## 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.sxt</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> |

## 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;  } } |

## 新增配置文件以及启动类，测试即可