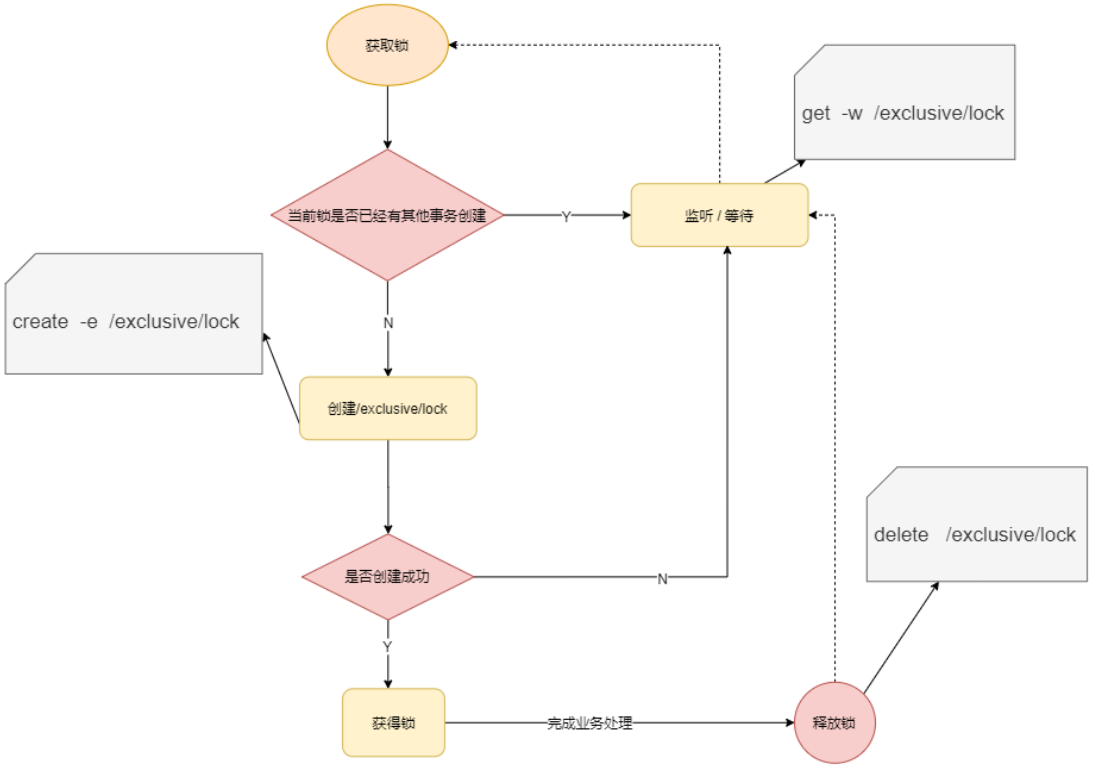
Zookeeper典型使用场景实战

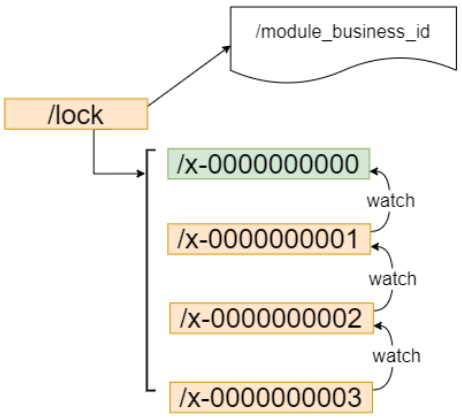
1. Zookeeper 非公平锁/公平锁/共享锁
2. Leader 选举在分布式场景中的应用
3. Spring Cloud Zookeeper注册中心实战

Zookeeper分布式锁实战

Zookeeper 分布式锁加锁原理



如上实现方式在并发问题比较严重的情况下，性能会下降的比较厉害，主要原因是，所有的连接都在对同一个节点进行监听，当服务器检测到删除事件时，要通知所有的连接，所有的连接同时收到事件，再次并发竞争，这就是**羊群效应。**这种加锁方式是**非公平锁**的具体实现：如何避免呢，我们看下面这种方式。



1. 请求进来，直接在/lock 节点下创建一个临时顺序节点
2. 判断自己是不是lock节点下，最小的节点

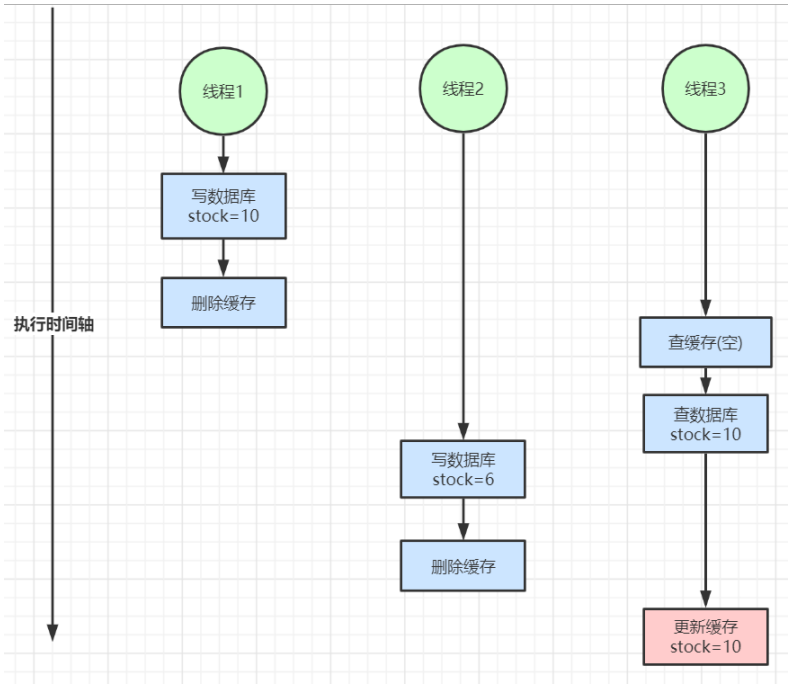
   a. 是最小的，获得锁
   b. 不是。对前面的节点进行监听(watch)
3. 获得锁的请求，处理完释放锁，即 delete 节点，然后后继第一个节点将收到通知，重复第2 步判断

如上借助于临时顺序节点，可以避免同时多个节点的并发竞争锁，缓解了服务端压力。这种实现方式所有加锁请求都进行排队加锁，是**公平锁**的具体实现。

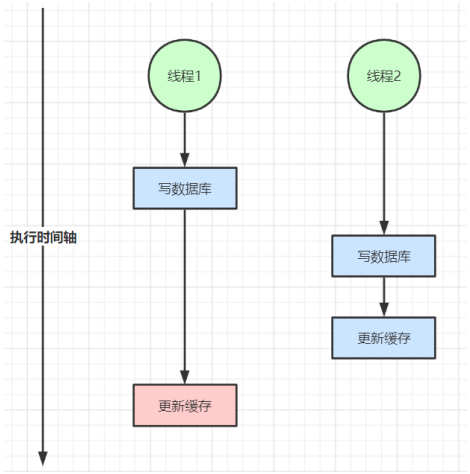
前面这两种加锁方式有一个共同的特质，就是都是**互斥锁**，同一时间只能有一个请求占用，如果是大量的并发上来，性能是会急剧下降的，所有的请求都得加锁，那是不是真的所有的请求都需要加锁呢？答案是否定的，比如如果数据没有进行任何修改的话，是不需要加锁的，但是如果读数据的请求还没读完，这个时候来了一个写请求，怎么办呢？有人已经在读数据了，这个时候是不能写数据的，不然数据就不正确了。直到前面读锁全部释放掉以后，写请求才能执行，所以需要给这个读请求加一个标识（读锁），让写请求知道，这个时候是不能修改数据的。不然数据就不一致了。如果已经有人在写数据了，再来一个请求写数据，也是不允许的，这样也会导致数据的不一致，所以所有的写请求，都需要加一个写锁，是为了避免同时对共享数据进行写操作。

举个例子

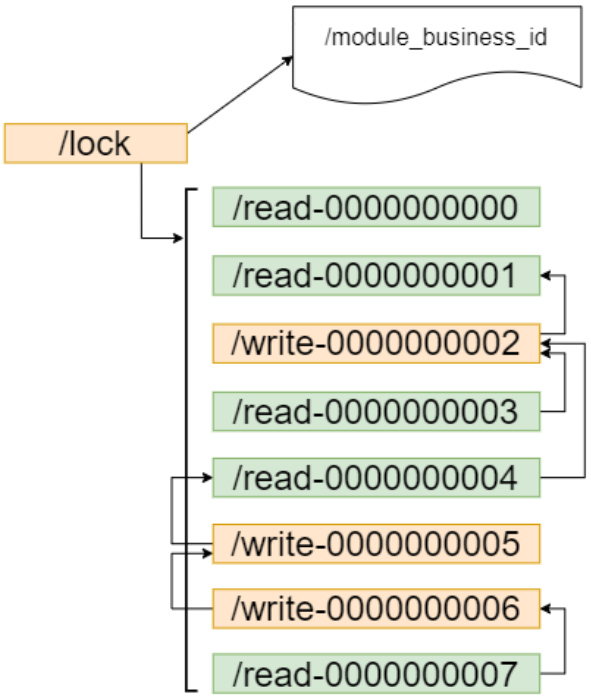
1、读写并发不一致



2、双写不一致情况



Zookeeper 共享锁实现原理



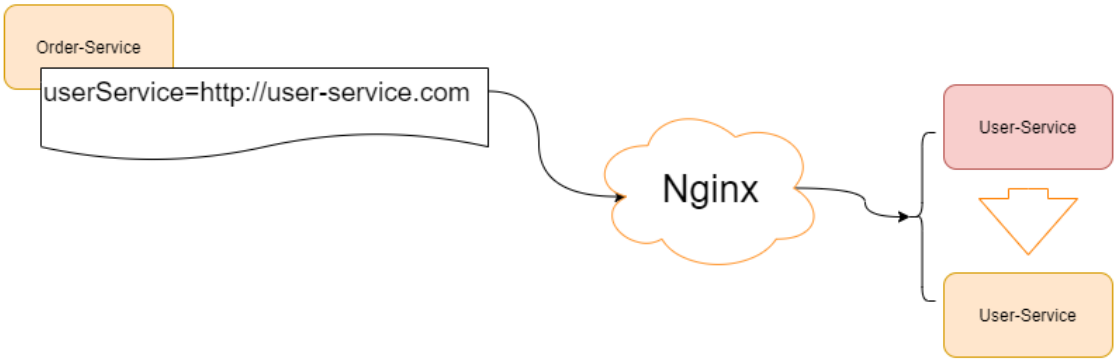
注册中心实战

注册中心场景分析：

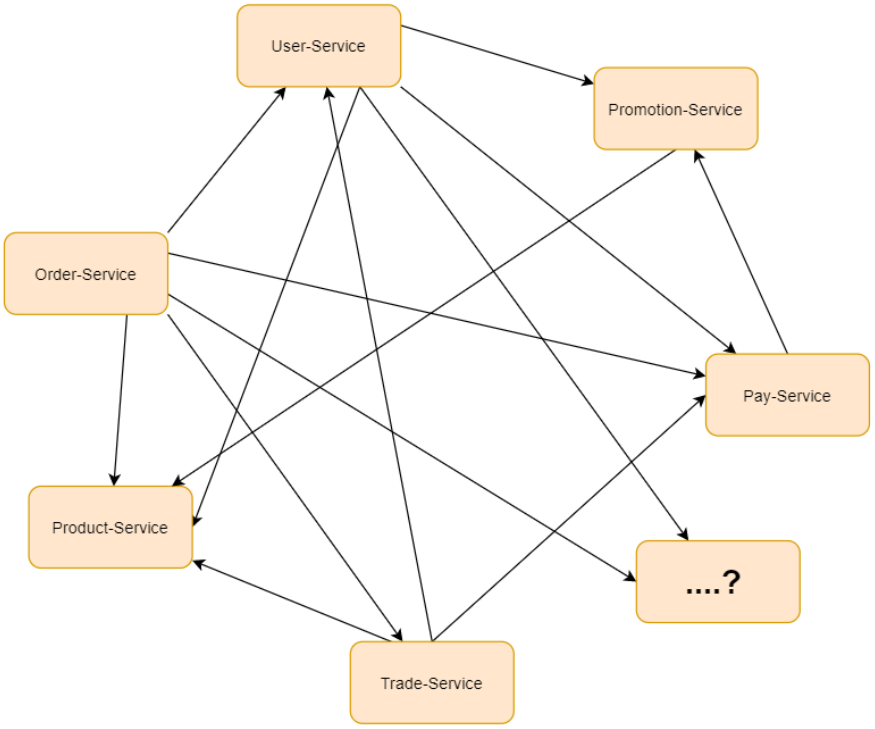
1. 在分布式服务体系结构比较简单的场景下，我们的服务可能是这样的

现在 Order-Service 需要调用外部服务的 User-Service ,对于外部的服务依赖，我们直接配置在我们的服务配置文件中,在服务调用关系比较简单的场景，是完全OK的。随着服务的扩张，User-Service 可能需要进行集群部署，如下：

如果系统的调用不是很复杂，可以通过配置管理，然后实现一个简单的客户端负载均衡也是OK的，但是随着业务的发展，服务模块进行更加细粒度的划分，业务也变得更加复杂，再使用简单的配置文件管理，将变得难以维护。当然我们可以再前面加一个服务代理，比如nginx做反向代理， 如下

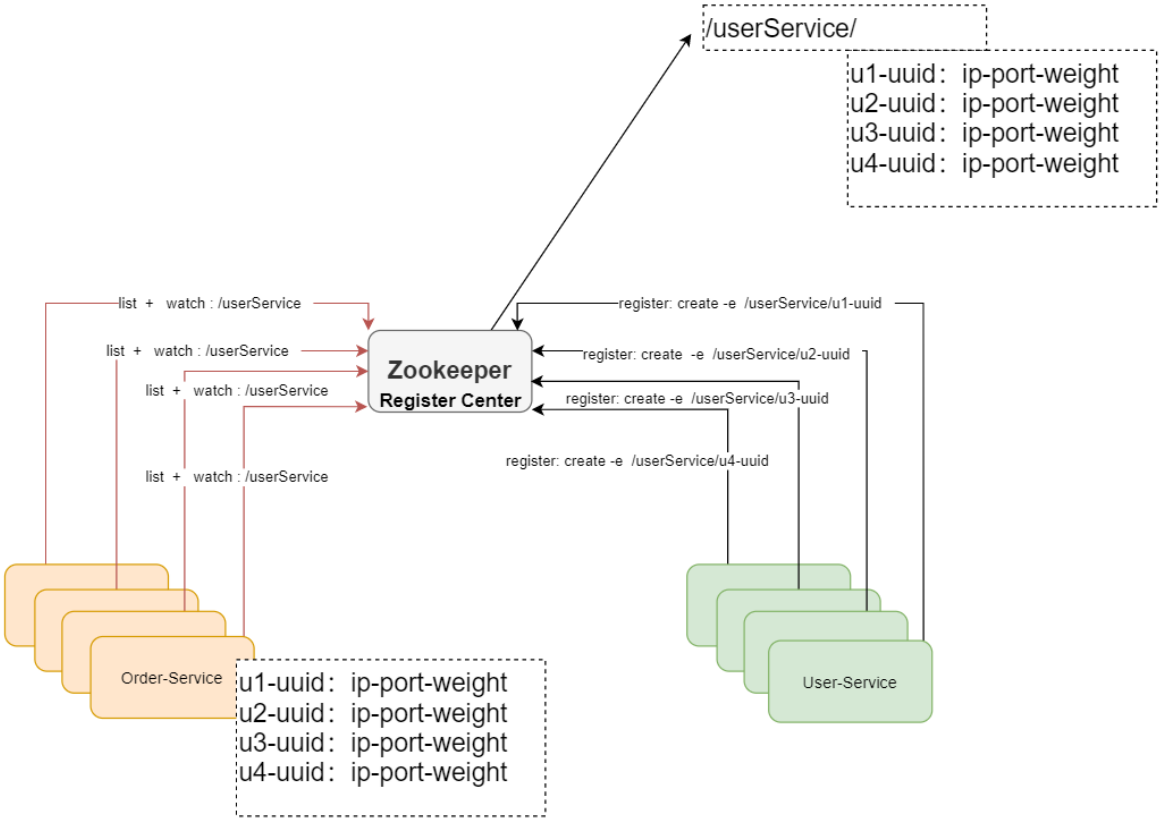


如果我们是如下场景呢?



服务不再是A-B,B-C 那么简单，而是错综复杂的微小服务的调用

这个时候我们可以借助于Zookeeper的基本特性来实现一个注册中心,什么是注册中心，顾名思义，就是让众多的服务，都在Zookeeper中进行注册，啥是注册，注册就是把自己的一些服务信息，比如IP，端口，还有一些更加具体的服务信息，都写到 Zookeeper节点上， 这样有需要的服务就可以直接从zookeeper上面去拿，怎么拿呢？ 这时我们可以定义统一的名称，比如，User-Service, 那所有的**用户服务**在**启动**的时候，都在User-Service 这个节点下面创建一个子节点（临时节点），这个子节点保持唯一就好，代表了每个服务实例的唯一标识，有依赖**用户服务**的比如**Order-Service** 就可以通过**User-Service** 这个父节点，就能获取所有的User-Service 子节点，并且获取所有的子节点信息（IP，端口等信息），拿到子节点的数据后**Order-Service**可以对其进行缓存，然后实现一个客户端的负载均衡，同时还可以对这个User-Service 目录进行监听， 这样有新的节点加入，或者退出，**Order-Service**都能收到通知，这样**Order-Service**重新获取所有子节点，且进行数据更新。这个用户服务的子节点的类型为临时节点。 第一节课有讲过，Zookeeper中临时节点生命周期是和SESSION绑定的，如果SESSION超时了，对应的节点会被删除，被删除时，Zookeeper 会通知对该节点父节点进行监听的客户端, 这样对应的客户端又可以刷新本地缓存了。当有新服务加入时，同样也会通知对应的客户端，刷新本地缓存，要达到这个目标需要客户端重复的注册对父节点的监听。这样就实现了服务的自动注册和自动退出。



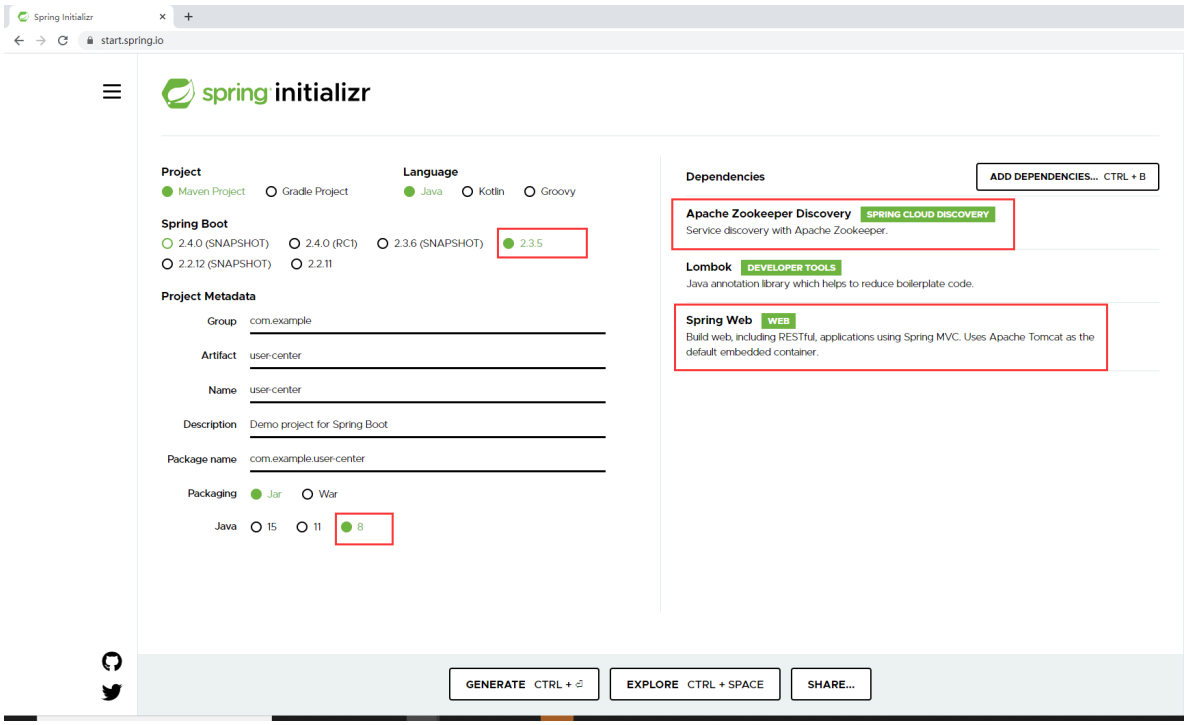
Spring Cloud 生态也提供了Zookeeper注册中心的实现，这个项目叫 Spring Cloud Zookeeper 下面我们来进行实战。

项目说明：
为了简化需求，我们以两个服务来进行讲解，实际使用时可以举一反三
　user-center : 用户服务
product-center: 产品服务

用户调用产品服务，且实现客户端的负载均衡，产品服务自动加入集群，自动退出服务。

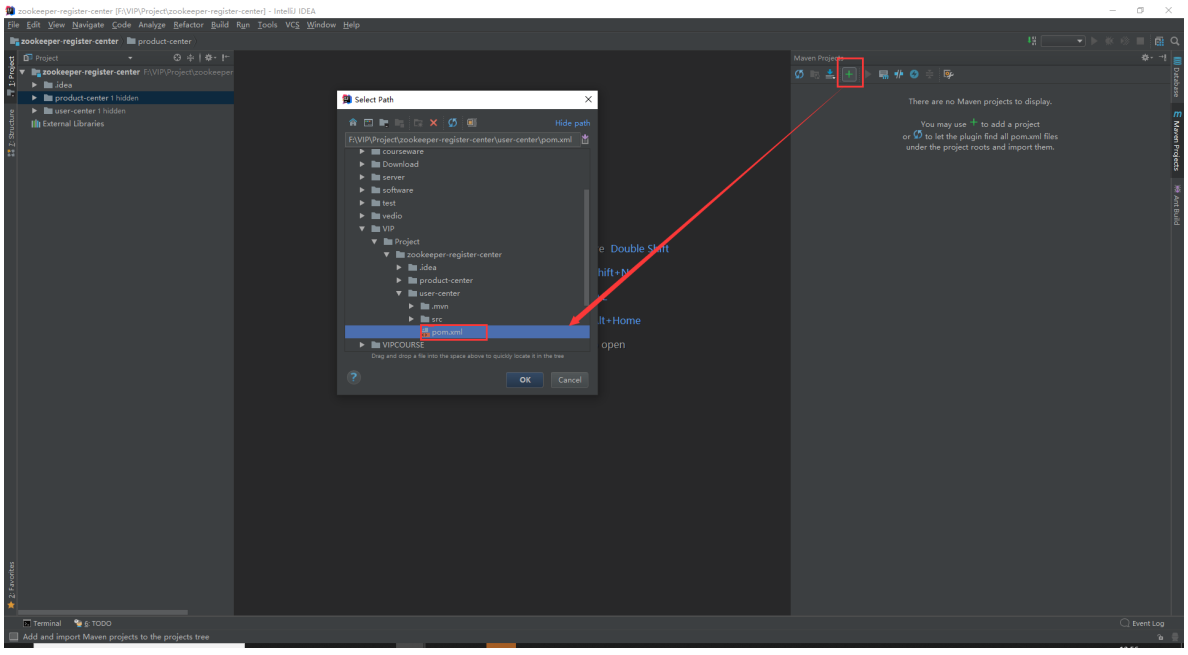
项目构建

1. 创建user-center 项目



同样的方式创建一个 product-center

2. 解压项目用idea打开，用maven导入项目



同样的方式引入product-center 项目

3. 配置zookeeper

user-center 服务：

application.properties

```
spring.application.name=user-center
#zookeeper 连接地址 ，
#如果使用了 spring cloud zookeeper config这个配置应该配置在 bootstrap.yml/bootstrap.properties中
```

```
spring.cloud.zookeeper.connect-string=192.168.109.200:2181
#将本服务注册到zookeeper，如果不希望自己被发现可以配置为false，默认为 true
spring.cloud.zookeeper.discovery.register=true
```

代码编写：

配置 Resttemplate 支持负载均衡方式

```
@SpringBootApplication
public class UserCenterApplication {

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


 @Bean
  @LoadBalanced
  public RestTemplate restTemplate(){
  return new RestTemplate();
  }
}
```

编写测试类：

TestController， Spring Cloud 支持 Feign, Spring RestTemplate,WebClient 以 逻辑名称，替代具体url的形式访问。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class TestController {

 @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/test")
  public String test(){
  return this.restTemplate.getForObject( "http://product-center/getInfo" ,String.class);
  }
}
```

product-center 服务：

application.properties

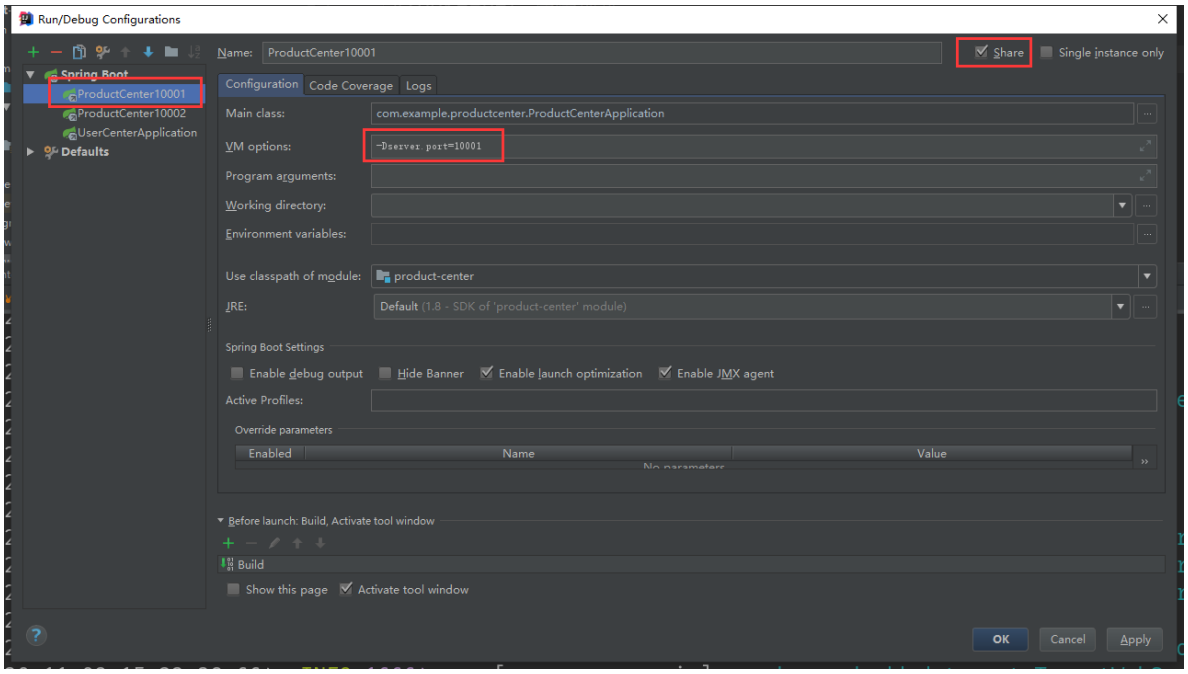
```
spring.application.name=user-center
#zookeeper 连接地址
spring.cloud.zookeeper.connect-string=192.168.109.200:2181
#将本服务注册到zookeeper
spring.cloud.zookeeper.discovery.register=true
```

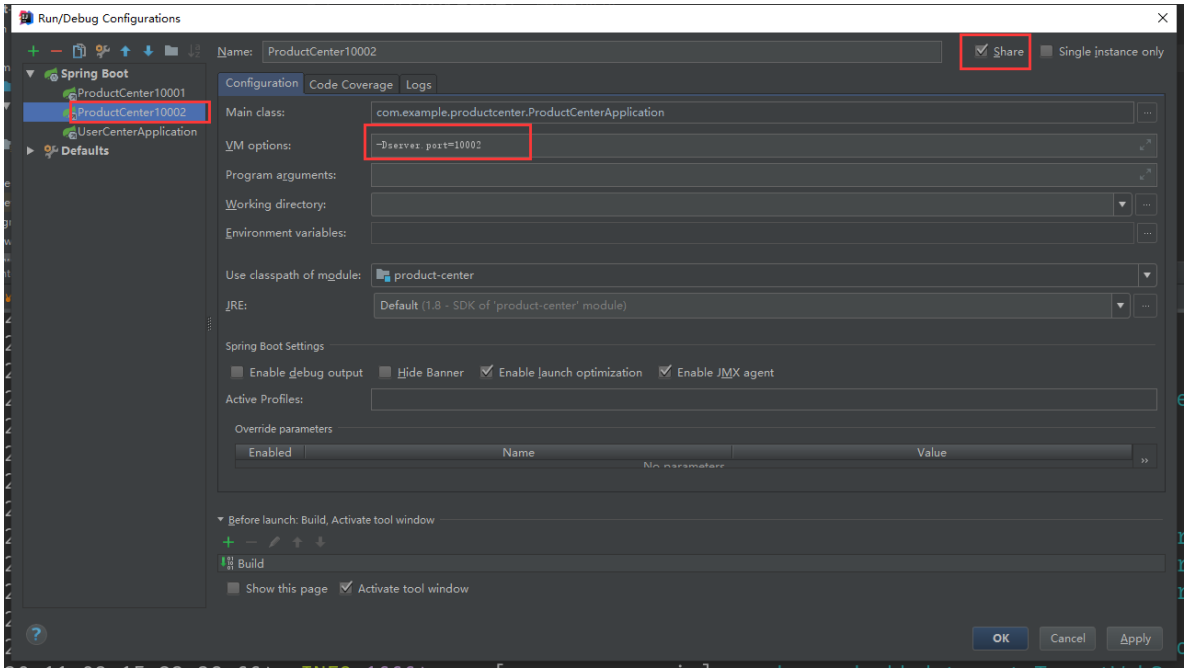
主类，接收一个getInfo 请求

```
@SpringBootApplication
@RestController
public class ProductCenterApplication {

 @Value("${server.port}")
 private String port;

 @Value( "${spring.application.name}" )
 private String name;

  @GetMapping("/getInfo")
  public String getServerPortAndName(){

  return this.name +" : "+ this.port;
  }
  public static void main(String[] args) {
  SpringApplication.run(ProductCenterApplication.class, args);
  }

}
```

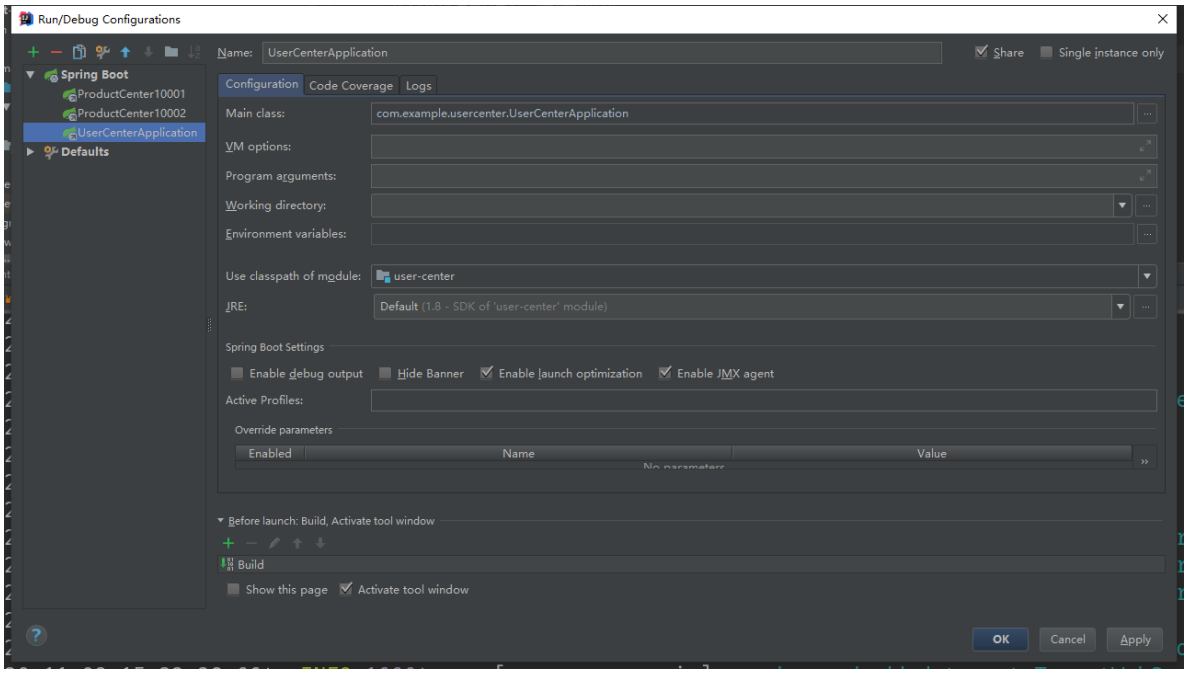
启动两个product-center 实例

通过idea 配置，启动多个实例，用端口区分不同的应用

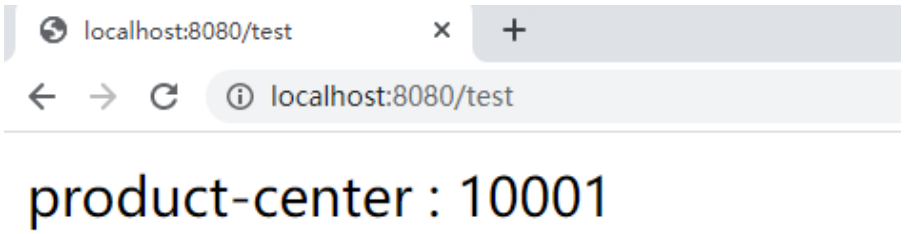


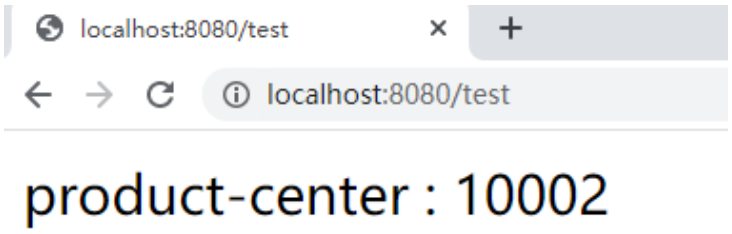


启动一个user-center 实例，默认8080端口



启动服务： 访问 http://localhost:8080/test





已经实现了，服务端的自动发现和客户端负载均衡。

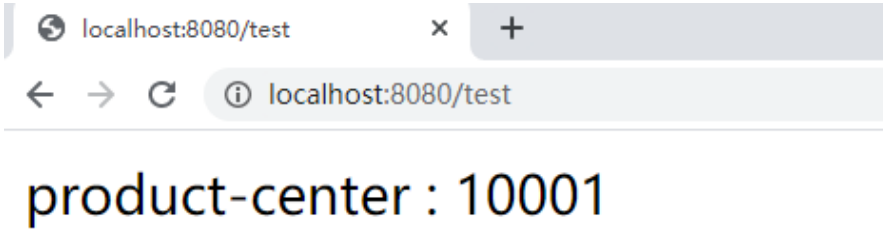
停掉product-center: 10002，再次访问

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Nov 09 15:38:33 CST 2020
There was an unexpected error (type=Internal Server Error, status=500).

一定的超时时间过去之后，product-center: 10002 会从zookeeper中剔除，zookeeper会通知客户端，进行本地缓存刷新，再次访问， 已经实现了失效节点的自动退出。



product-center : 10001

文档：VIP-03 Zookeeper典型使用场景实战.not...
链接：http://note.youdao.com/noteshare?id=80d7bcb710a0e21616248239b9dfd40d&sub=3E9635FCEEB5449B80885533C285C94B