# 【Spring Cloud Ribbon】

**【主要内容】**

1. Ribbon简介
2. Ribbon快速入门
3. Ribbon源码分析
4. Ribbon负载均衡算法

**【学习目标】**

|  |  |
| --- | --- |
| 知识点 | 要求 |
| Ribbon简介 | 了解 |
| Ribbon快速入门 | 掌握 |
| Ribbon源码分析 | 了解 |
| Ribbon负载均衡算法 | 掌握 |

# Ribbon概述

Spring Cloud Ribbon是一个基于HTTP和TCP的**客户端负载均衡工具**，它基于**Netflix** Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。

简单的说Ribbon就是netfix公司的一个开源项目，主要功能是提供**客户端负载均衡算法和服务调用**。Ribbon客户端组件提供了一套完善的配置项，比如连接超时，重试等。

在Spring Cloud 构建的微服务系统中， Ribbon 作为服务**消费者**的负载均衡器，有两种使用方式，一种是和**RestTemplate**相结合，另一种是和OpenFeign相结合。OpenFeign 已经默认集成了Ribbon,关于OpenFeign的内容将会在下一章进行详细讲解。Ribbon 有很多子模块，但很多模块没有用于生产环境!





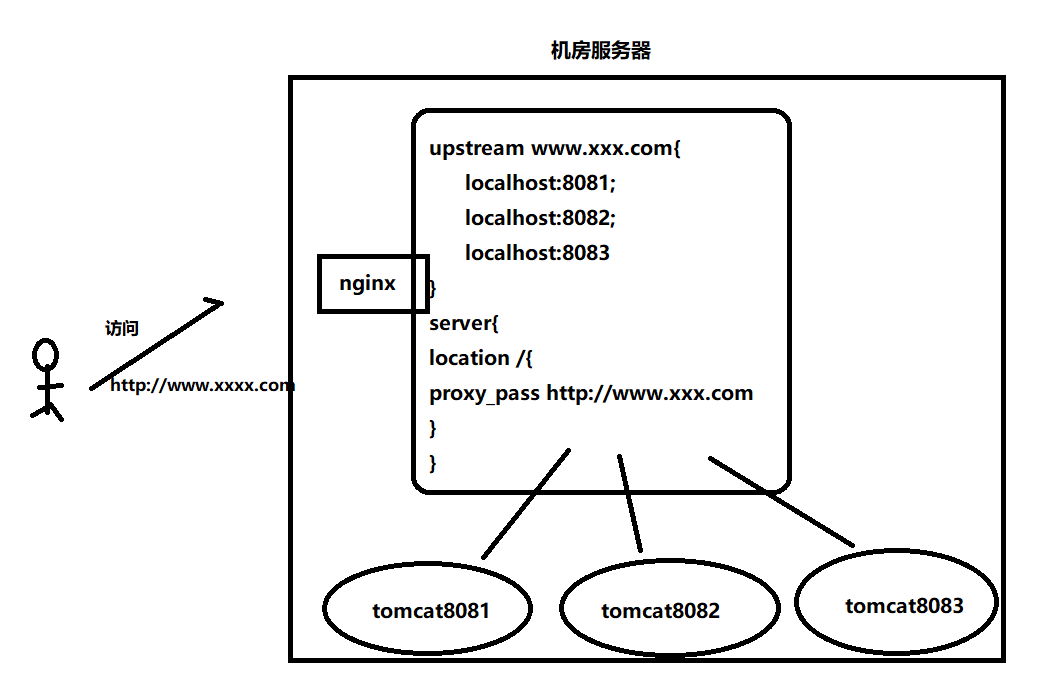
# 负载均衡

负载均衡，英文名称为Load Balance（LB），其含义就是指将负载（工作任务）进行平衡、分摊到多个操作单元上进行运行，例如Web服务器、企业核心应用服务器和其它主要任务服务器等，从而协同完成工作任务。

负载均衡构建在原有网络结构之上，它提供了一种透明且廉价有效的方法扩展服务器和网络设备的带宽、加强网络数据处理能力、增加吞吐量、提高网络的可用性和灵活性。

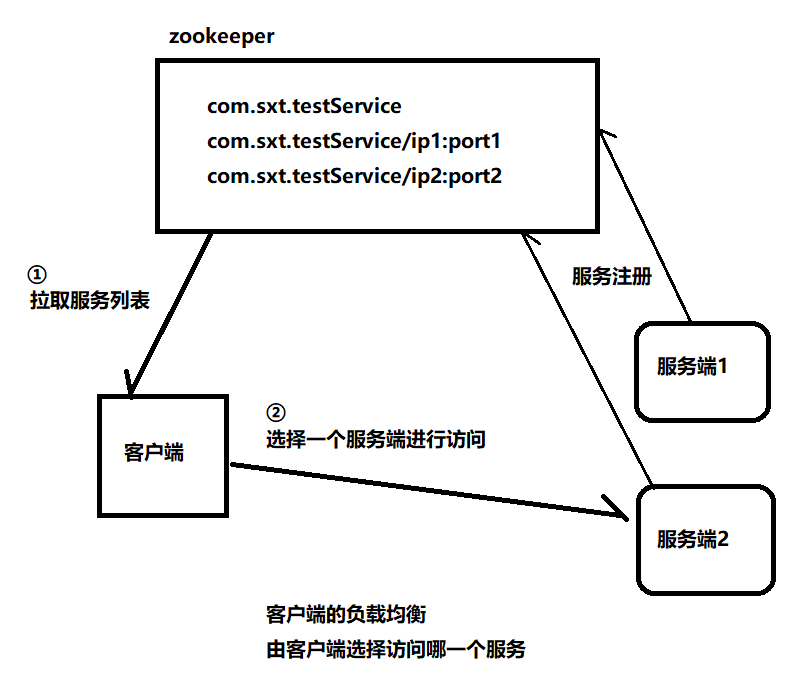
## 服务器的负载均衡

Nginx，F5



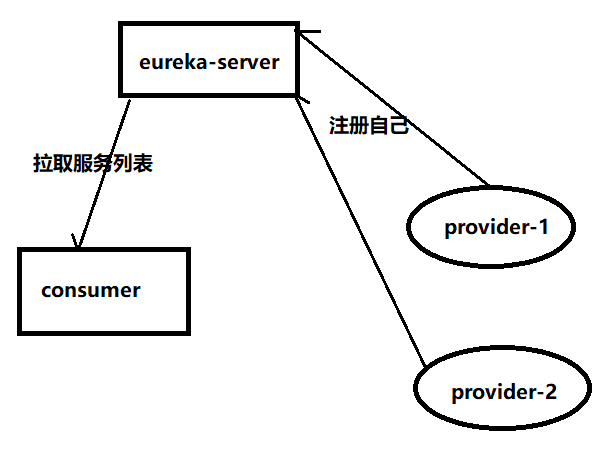
## 客户端的负载均衡

Zookeeper



# Ribbon快速入门

## 本次调用设计图



## 项目搭建

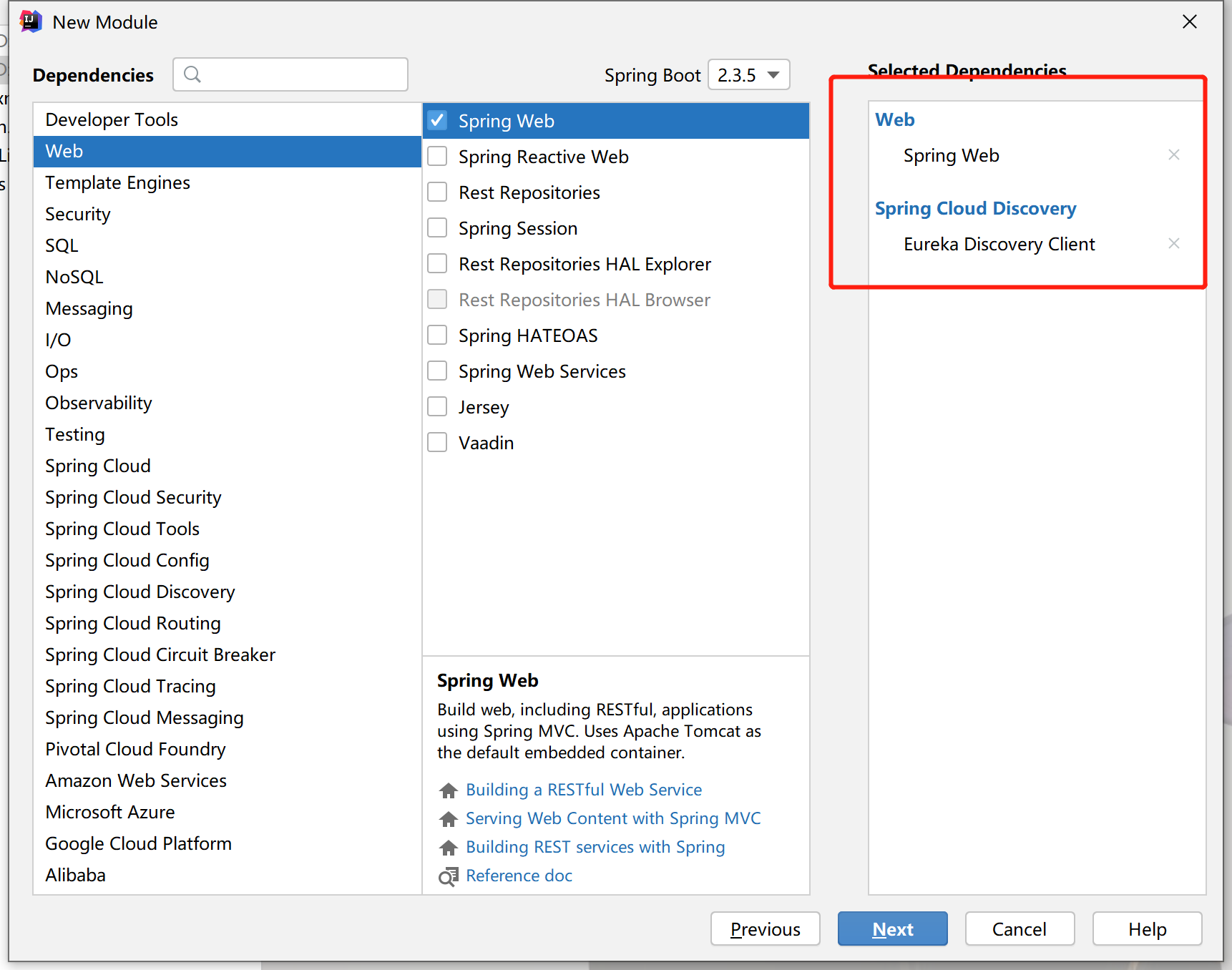
**consumer 和 provider-1和 provider-2 都是eureka-client**

**注意这三个依赖是eureka-client**

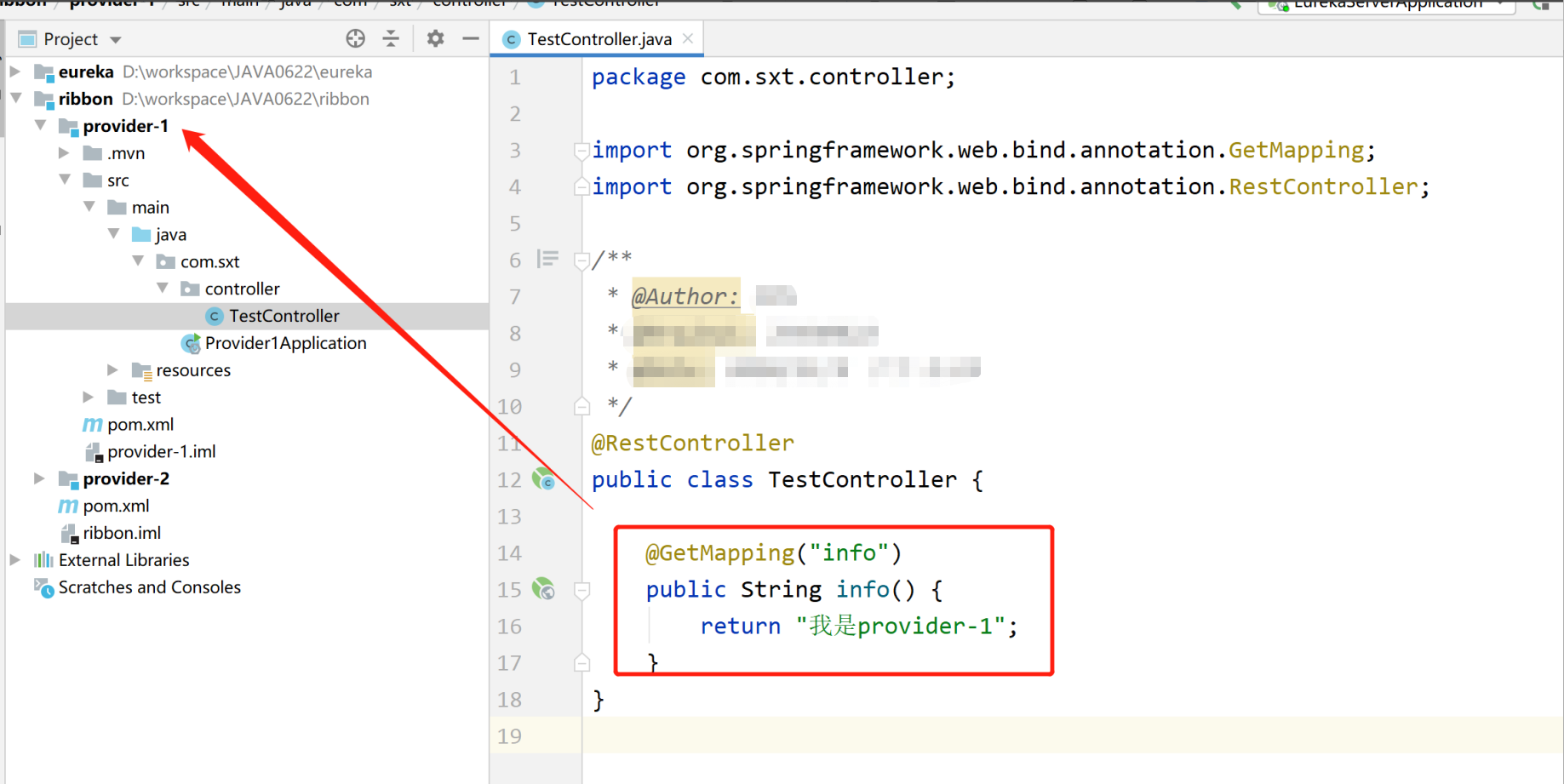
**注意provider-1和provider-2的spring.application.name=provider**

**注意启动类的注解和配置文件的端口以及服务名称**

## 创建provider-1和provider-2



## 编写provider-1和provider-2





## 创建consumer



## 编写consumer的启动类

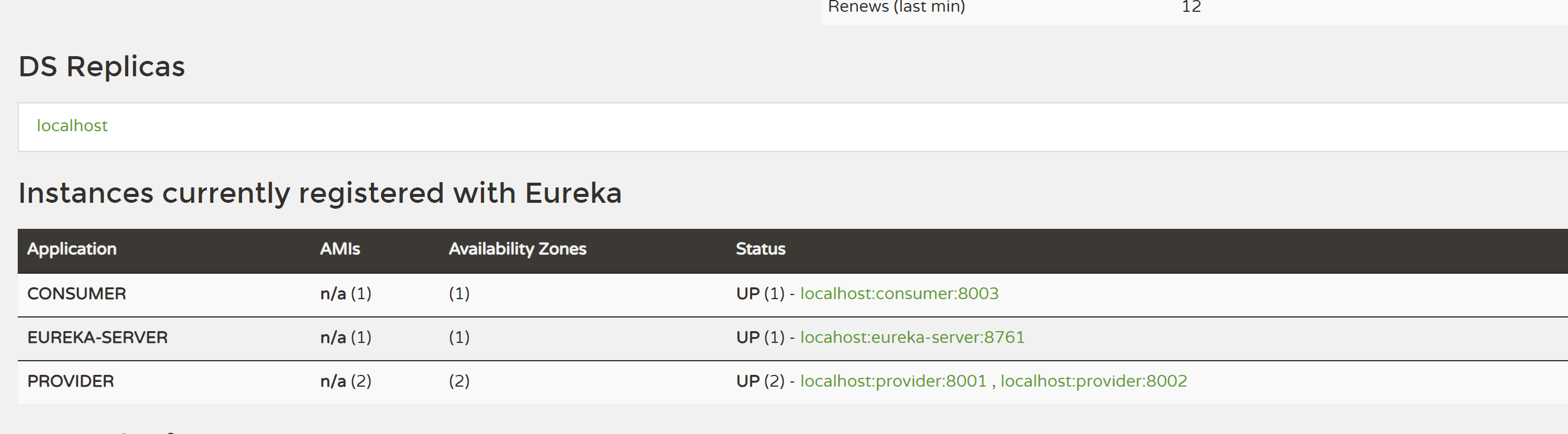
|  |
| --- |
| package com.sxt;  import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.cloud.netflix.eureka.EnableEurekaClient; import org.springframework.context.annotation.Bean; import org.springframework.web.client.RestTemplate;  @SpringBootApplication @EnableEurekaClient public class ConsumerApplication {   public static void main(String[] args) {  SpringApplication.*run*(ConsumerApplication.class, args);  }   */\*\*  \* 用来发请求的  \*  \* @return  \*/* @Bean  public RestTemplate restTemplate() {  return new RestTemplate();  } } |

## 编写consumer的TestController

|  |
| --- |
| package com.sxt.controller;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.cloud.client.ServiceInstance; import org.springframework.cloud.client.discovery.DiscoveryClient; import org.springframework.util.ObjectUtils; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; import org.springframework.web.client.RestTemplate;  import java.util.List; import java.util.Random;  */\*\*  \* @Author: 武汉尚学堂  \*/* @RestController public class TestController {   @Autowired  private RestTemplate restTemplate;   @Autowired  private DiscoveryClient discoveryClient;   static Random *random* = new Random();   @RequestMapping("/testBalance")  public String testBalance(String serviceId) {  *//获取服务列表* List<ServiceInstance> instances = discoveryClient.getInstances(serviceId);  if (ObjectUtils.*isEmpty*(instances)) {  return "服务列表为空";  }  *//如果服务列表不为空，先自己做一个负载均衡* ServiceInstance serviceInstance = loadBalance(instances);   String host = serviceInstance.getHost();  int port = serviceInstance.getPort();  String url = "http://" + host + ":" + port + "/info";  System.*out*.println("本次我调用的是" + url);  String forObject = restTemplate.getForObject(url, String.class);  System.*out*.println(forObject);  return forObject;  }   private ServiceInstance loadBalance(List<ServiceInstance> instances) {  *//拼接url去调用 ip:port 先自己实现不用ribbon* ServiceInstance serviceInstance = instances.get(*random*.nextInt(instances.size()));  return serviceInstance;  } } |

## 启动测试

首选确保都注册上去了



然后访问调用

<http://localhost:8003/testBalance?serviceId=provider>



## 使用Ribbon改造

只需要对consumer改造即可，改造启动类

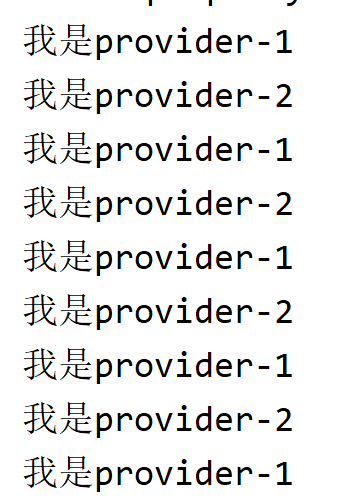
**改造controller**

|  |
| --- |
| *\*\*  \* 用来发请求的  \*  \* @return  \*/* @Bean @LoadBalanced *//ribbon的负载均衡注解* public RestTemplate restTemplate() {  return new RestTemplate(); } |

|  |
| --- |
| */\*\*  \* 测试ribbon的负载均衡  \*  \* @param serviceId  \* @return  \*/* @RequestMapping("/testRibbonBalance") public String testRibbonBalance(String serviceId) {  *//直接用服务名称替换ip:port* String url = "http://" + serviceId + "/info";  String forObject = restTemplate.getForObject(url, String.class);  System.*out*.println(forObject);  return forObject; } |

## 改造后测试效果

访问 <http://localhost:8003/testRibbonBalance?serviceId=provider>



# Ribbon源码分析

## Ribbon要做什么事情？

**先通过 "http://" + serviceId + "/info" 我们思考ribbon在真正调用之前需要做什么？**

**restTemplate.getForObject(“http://provider/info”, String.class);**

想要把上面这个请求执行成功，我们需要以下几步

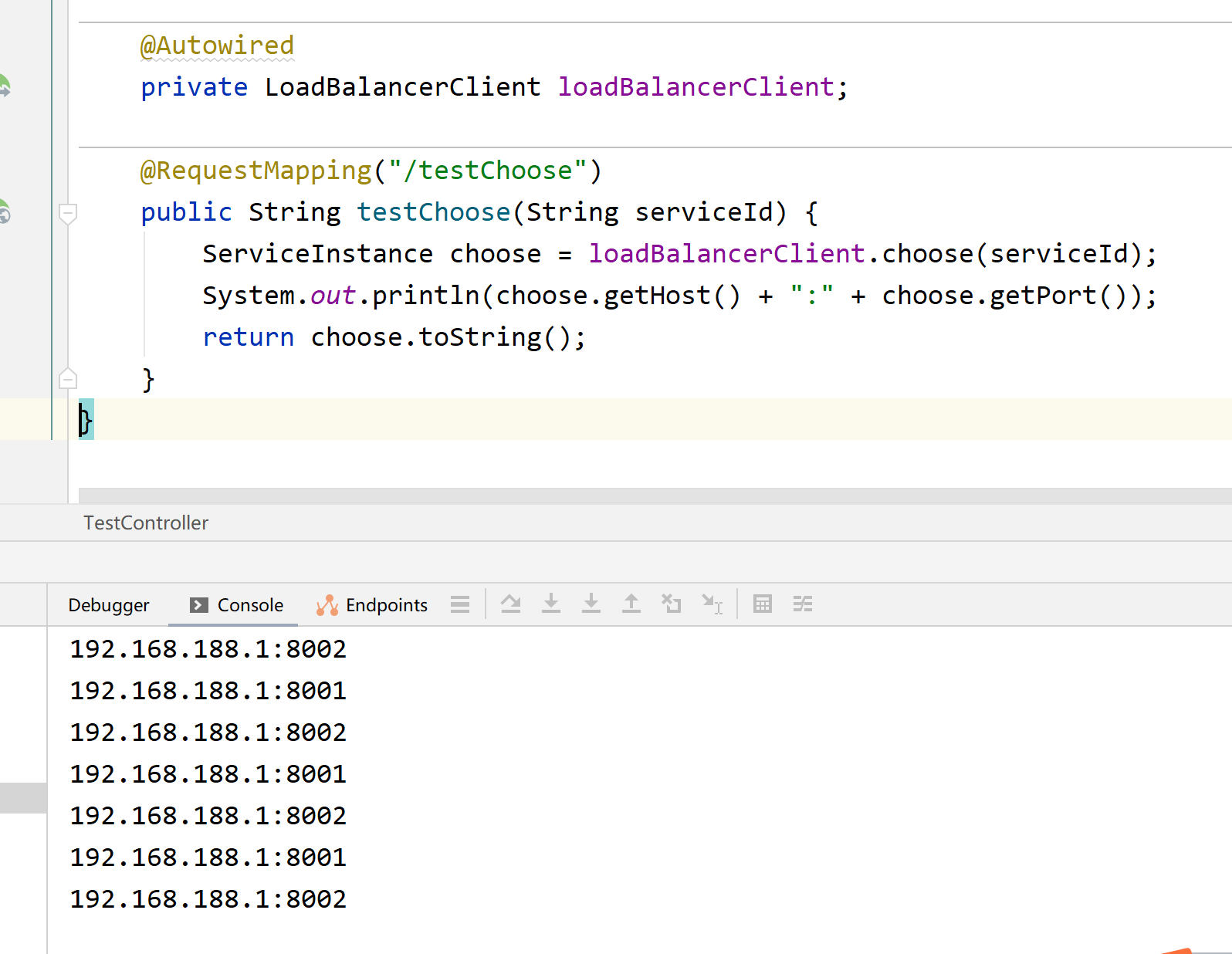
1. 拦截该请求；
2. 获取该请求的URL地址:http://provider/info
3. 截取URL地址中的provider
4. 从服务列表中找到key为provider的服务实例的集合(服务发现)
5. 根据负载均衡算法选出一个符合的实例
6. 拿到该实例的host和port，替换原来URL中的provider
7. 真正的发送restTemplate.getForObject(“http://ip:port/info”，String.class)

## Ribbon负载均衡的测试

新增controller

|  |
| --- |
| @Autowired private LoadBalancerClient loadBalancerClient;  @RequestMapping("/testChoose") public String testChoose(String serviceId) {  ServiceInstance choose = loadBalancerClient.choose(serviceId);  System.*out*.println(choose.getHost() + ":" + choose.getPort());  return choose.toString(); } |

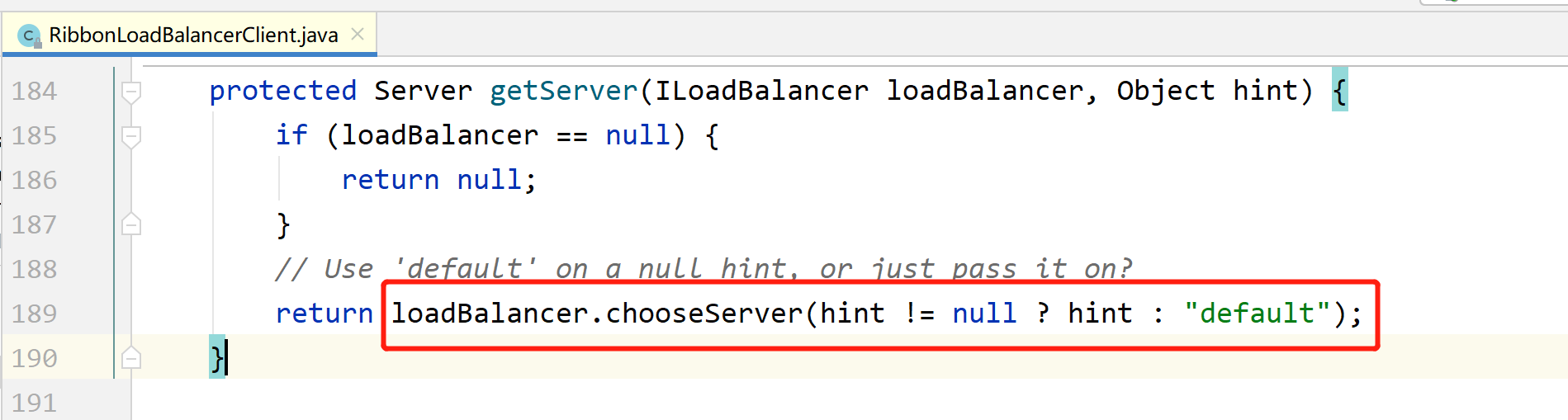
访问：<http://localhost:8003/testChoose?serviceId=provider>



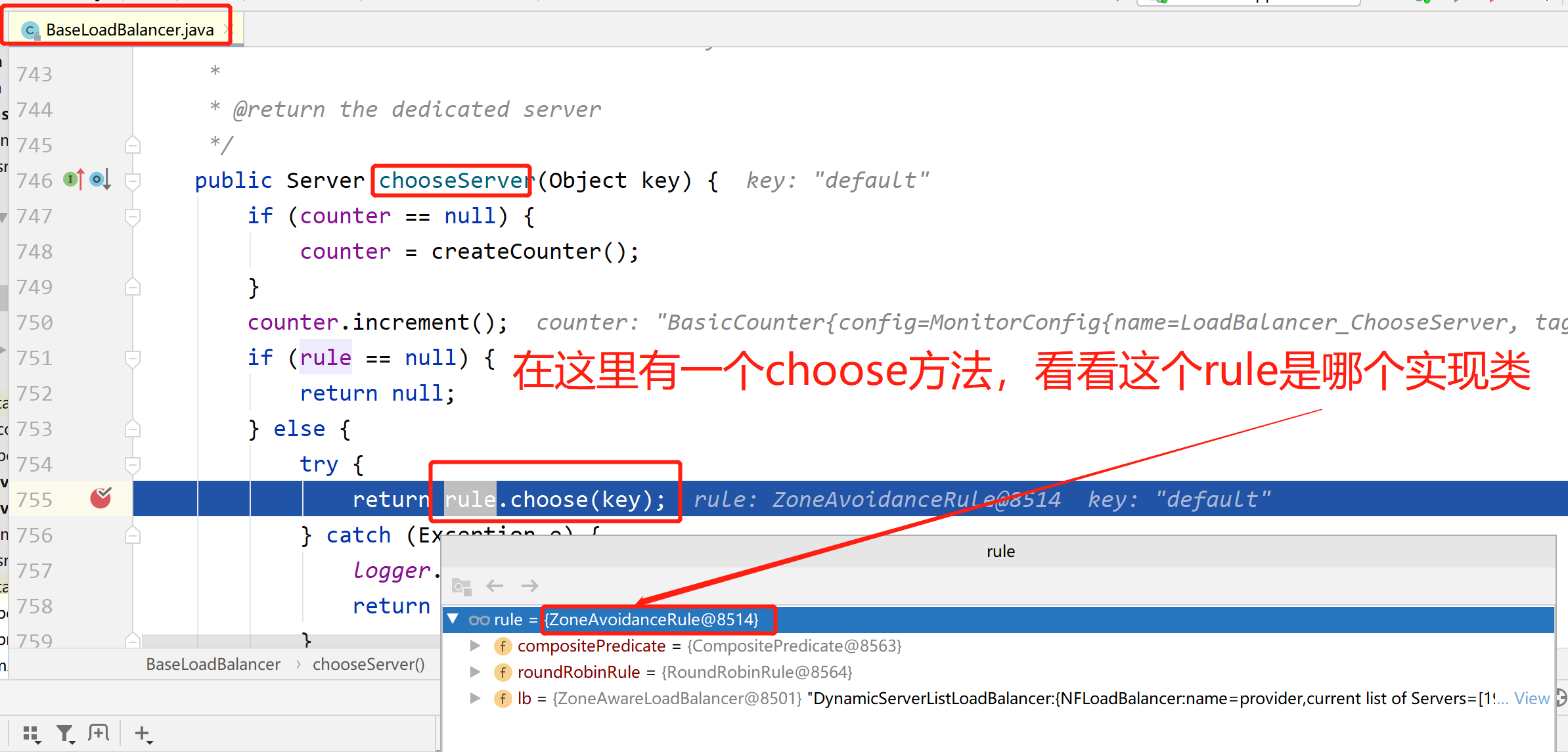
## 从choose方法入手，查看Ribbon负载均衡的源码



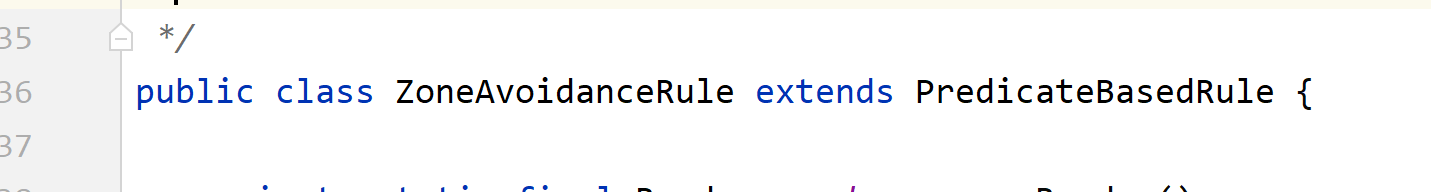
**走进getServer()方法**



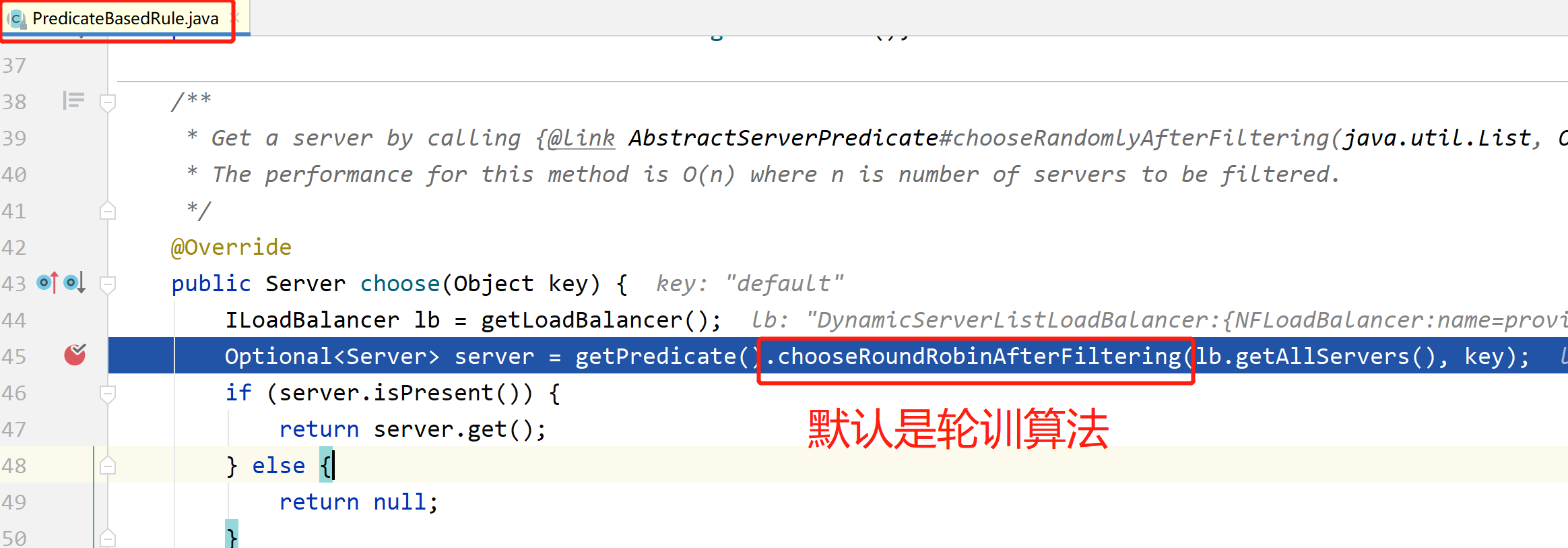
**在chooseServer()里面得到rule是哪个对象**

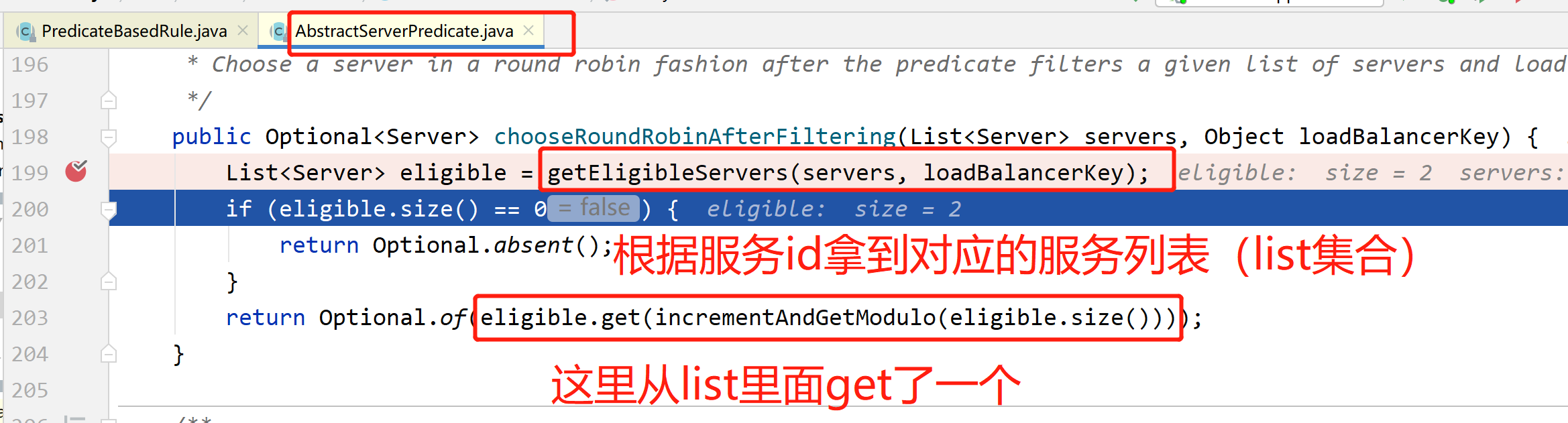


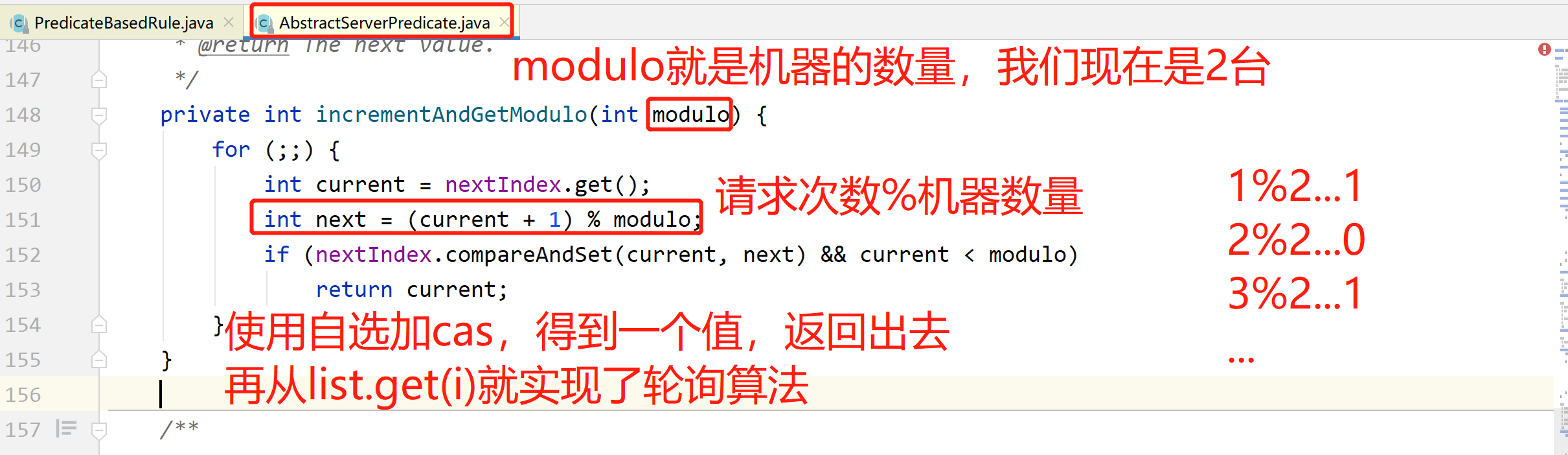
**发现当前的rule是ZoneAvoidanceRule对象，而他只有一个父类PredicateBasedRule**



**最终进入PredicateBasedRule类的choose()方法**

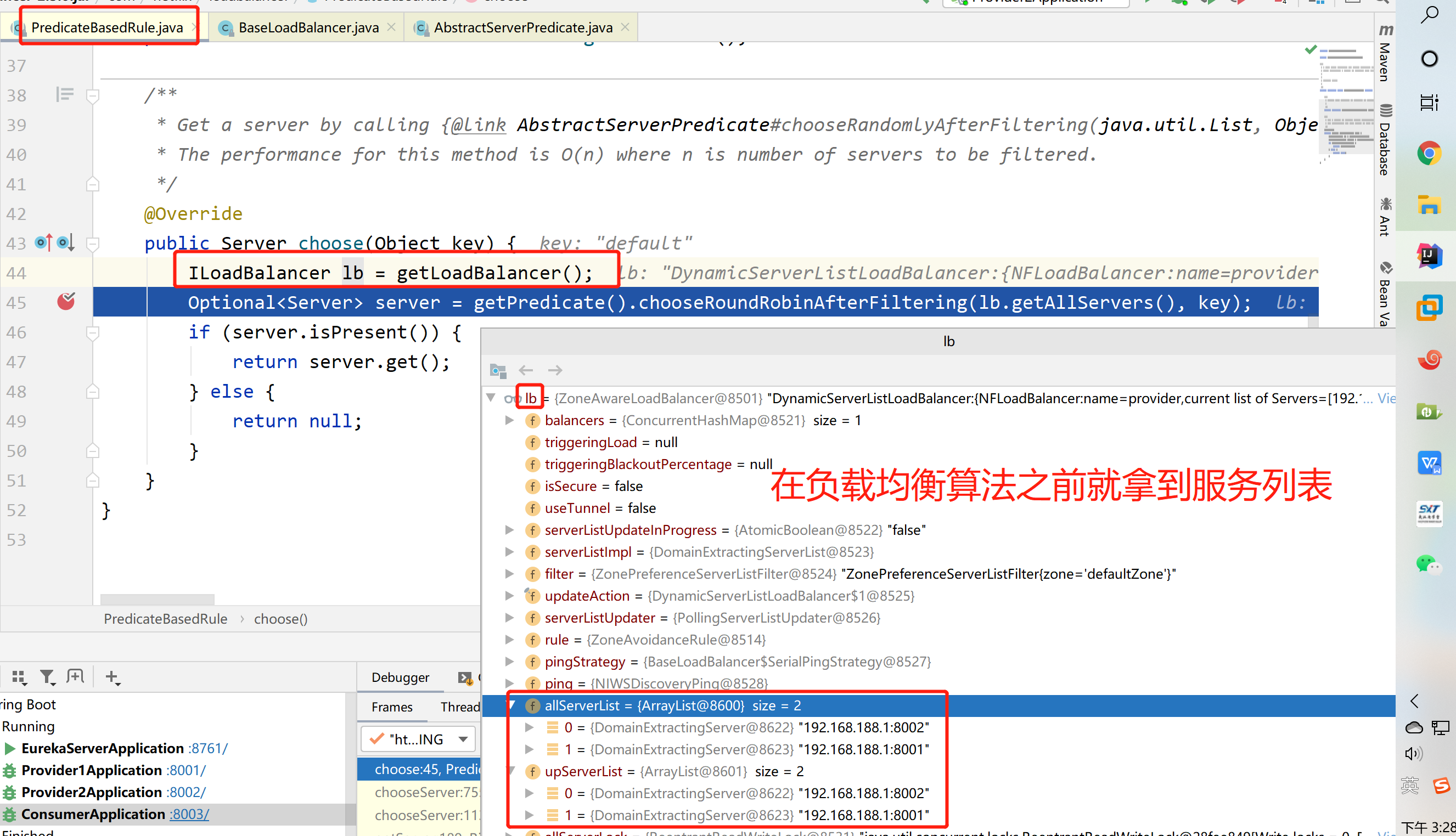




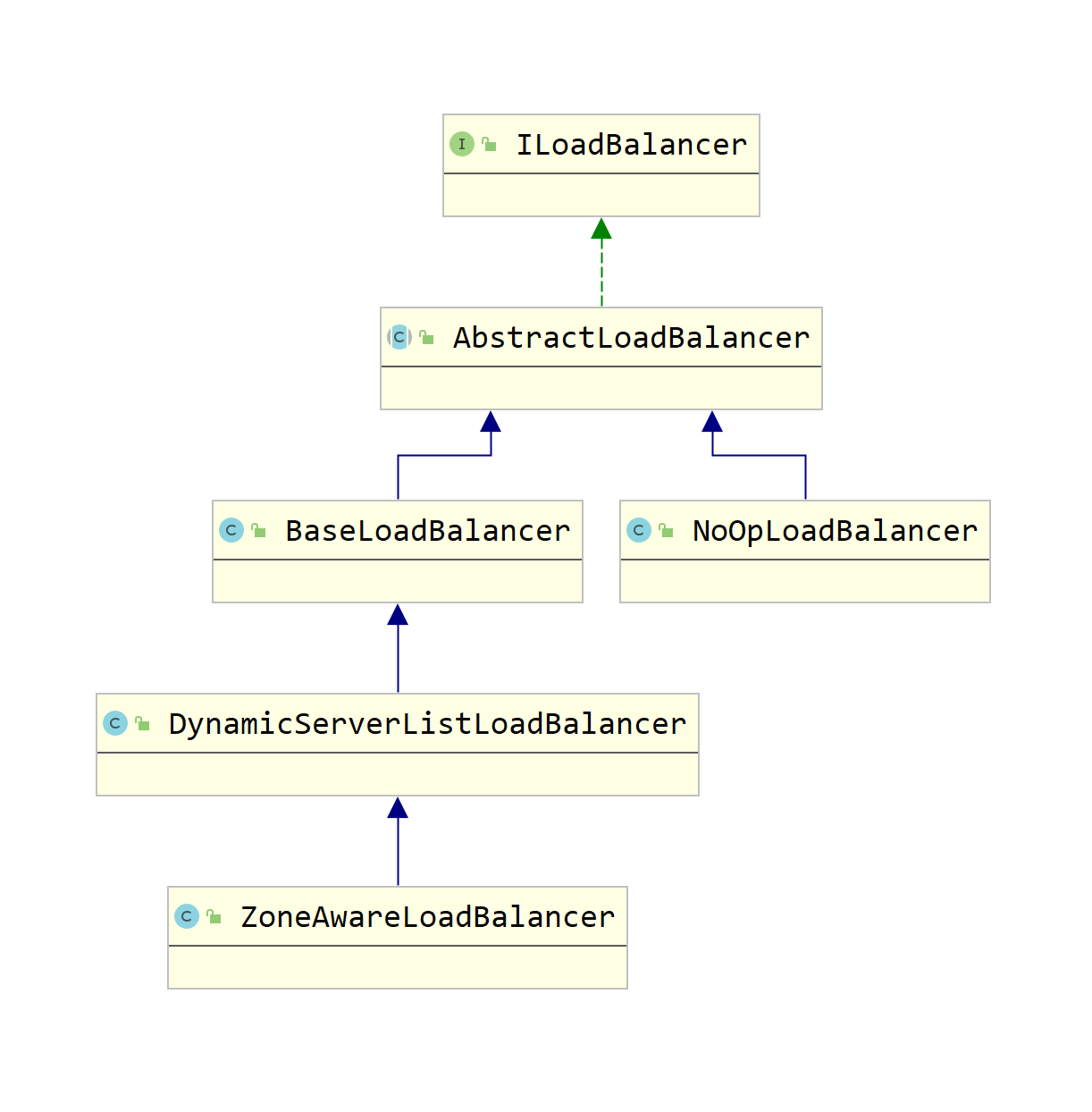


## 负载均衡之前的服务列表是从何而来呢？

我们发现在负载均衡之前，服务列表已经有数据了



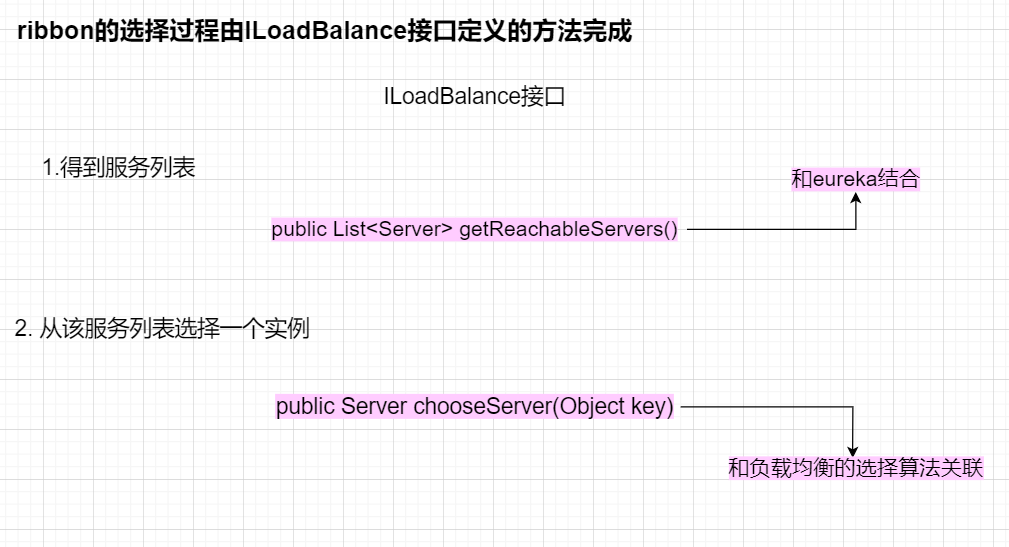
**重点接口ILoadBalancer**



**重点接口ILoadBalancer**



**Ribbon没有服务发现的功能，但是eureka有，所以ribbon和eureka完美结合，我们继续干源码学习**



**首先关注这两个集合，就是存放从eureka服务端拉取的服务列表然后缓存到本地**



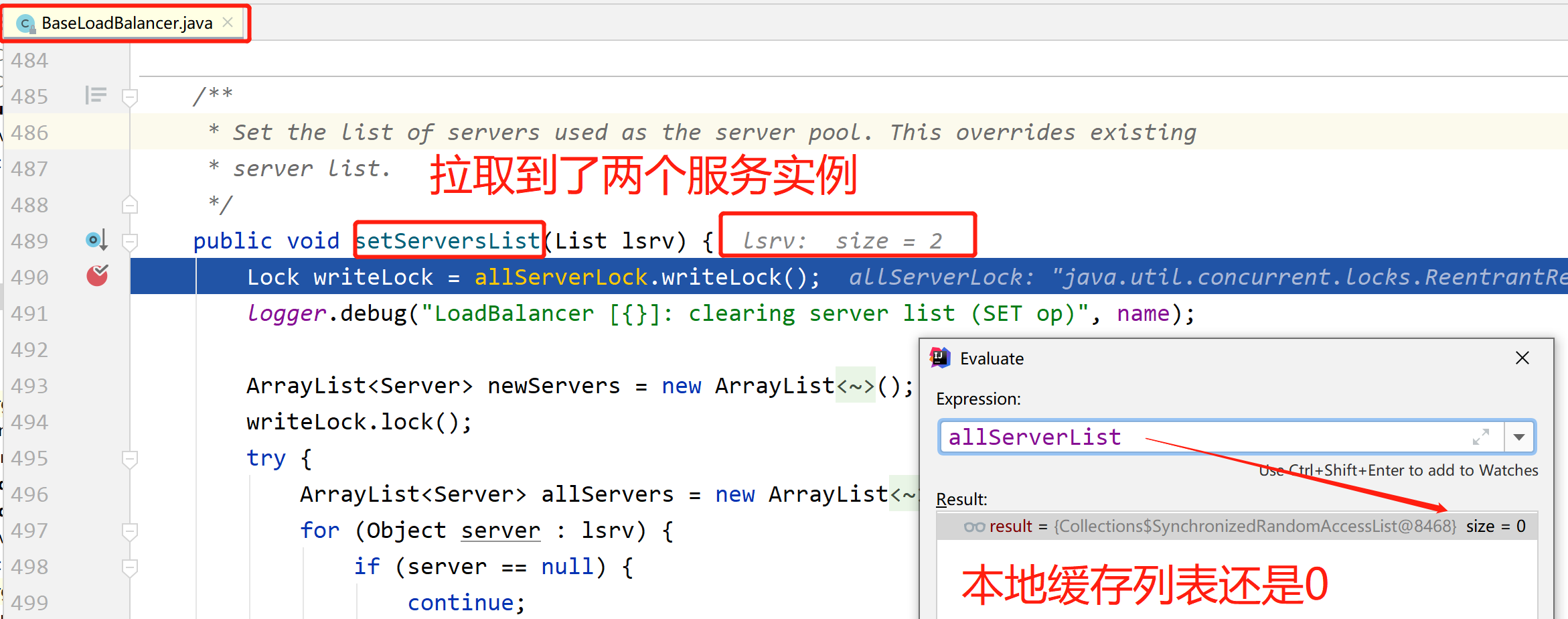
**我们去看DynamicServerListLoadBalancer类如何获取服务列表，然后放在ribbon的缓存里面**



**ServerList<T extends Server> 实现类（DiscoveryEnabledNIWSServerList）**



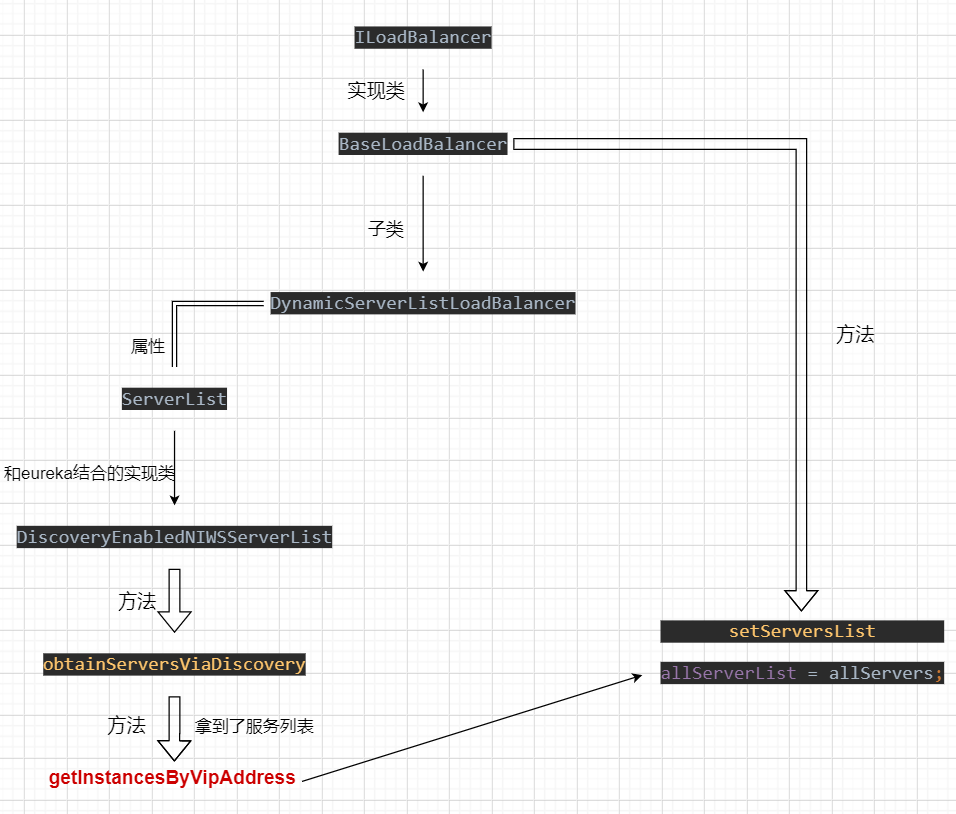
**再回到BaseLoadBalancer中真正的存放服务列表**





**最后我们得知，只有在初始化DynamicServerListLoadBalancer类时，去做了服务拉取和缓存**

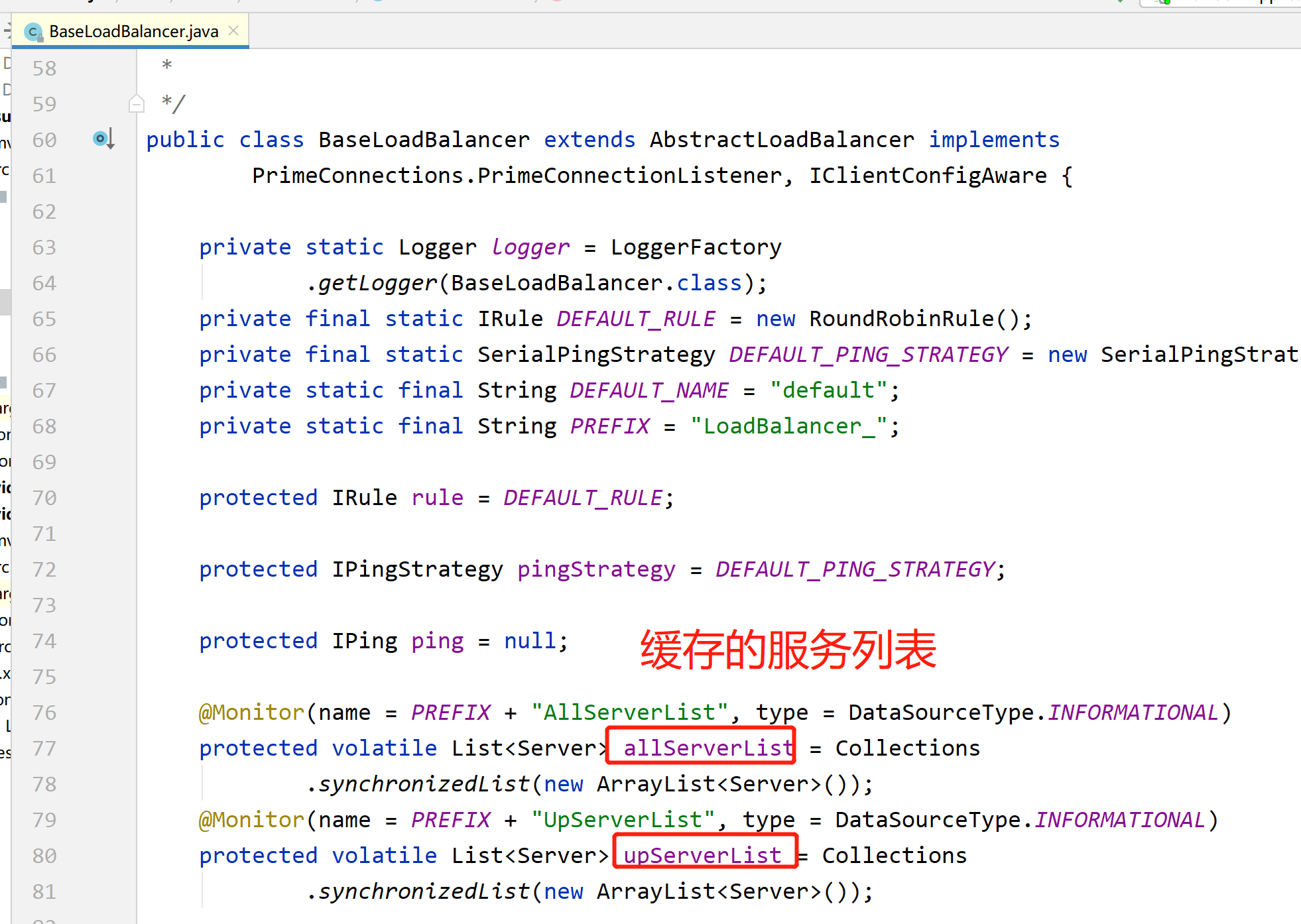
**也就是说并不是服务一启动就拉取了服务列表缓存起来，流程图如下:**



## **Ribbon把serverList缓存起来，脏读怎么处理？（选学**）

根据上面缓存服务列表我们得知，ribbon的每个客户端都会从eureka-server中把服务列表缓存起来

主要的类是BaseLoadBalancer，那么有新的服务上线或者下线，这么保证缓存及时同步呢

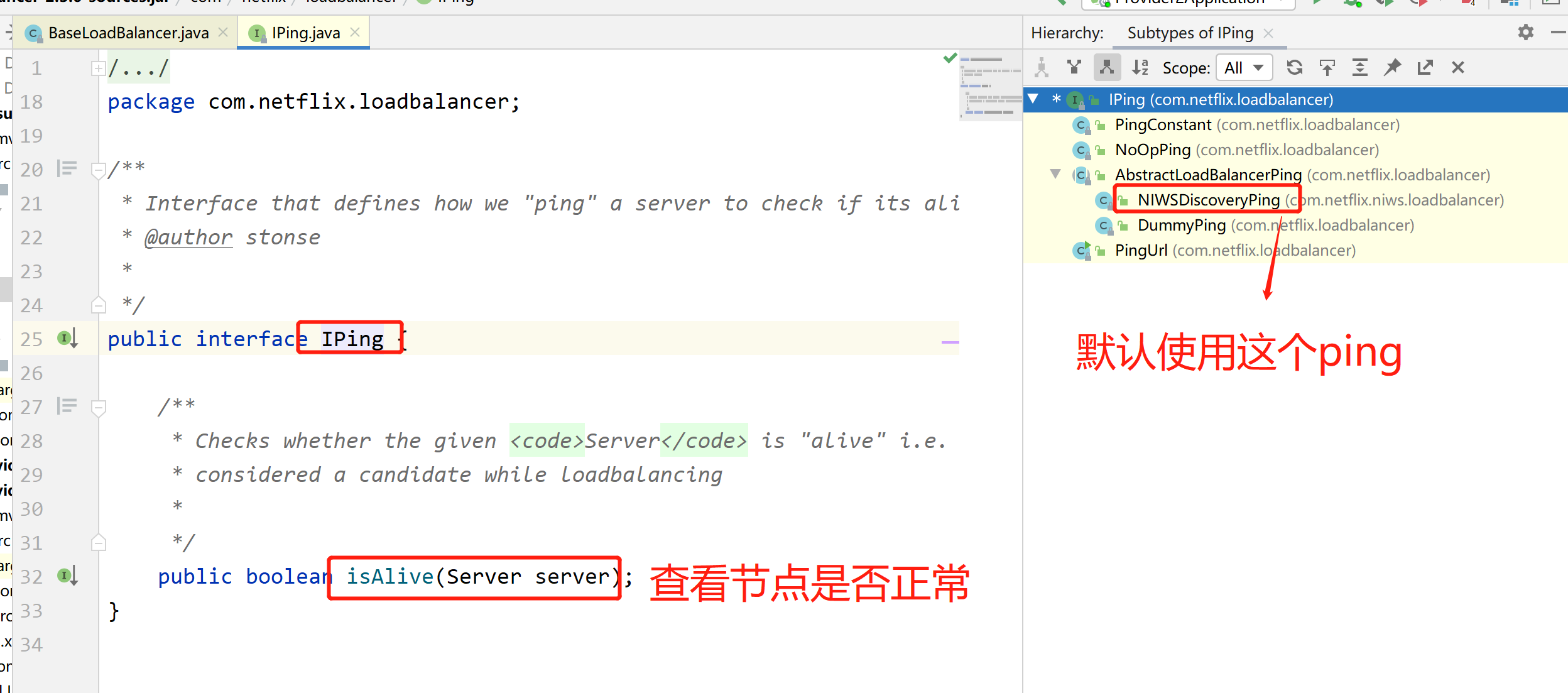


**Ribbon中使用了一个PING机制**

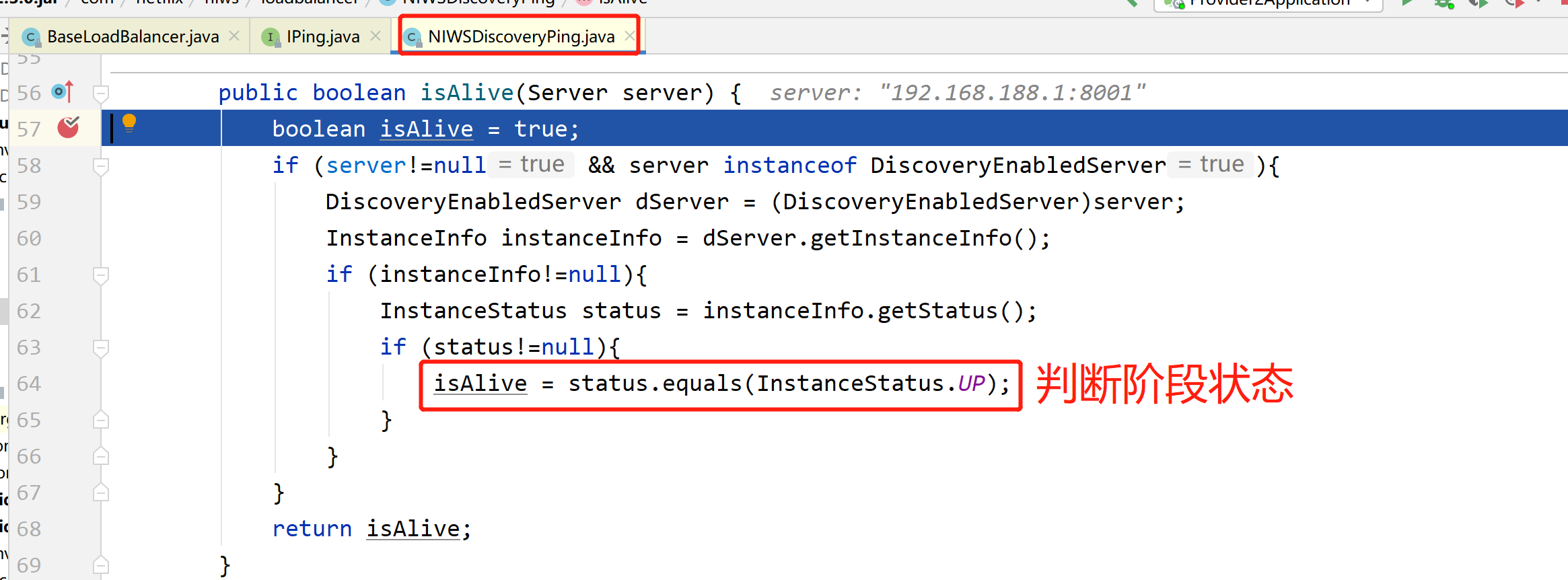
**从eureka中拿到服务列表，缓存到本地，ribbon搞了个定时任务，隔一段时间就去循环ping一下每个服务节点是否存活**



**我们查看IPing这个接口**



**我们就想看NIWSDiscoveryPing**



**跟着isAlive一直往上找，看哪里去修改本地缓存列表**

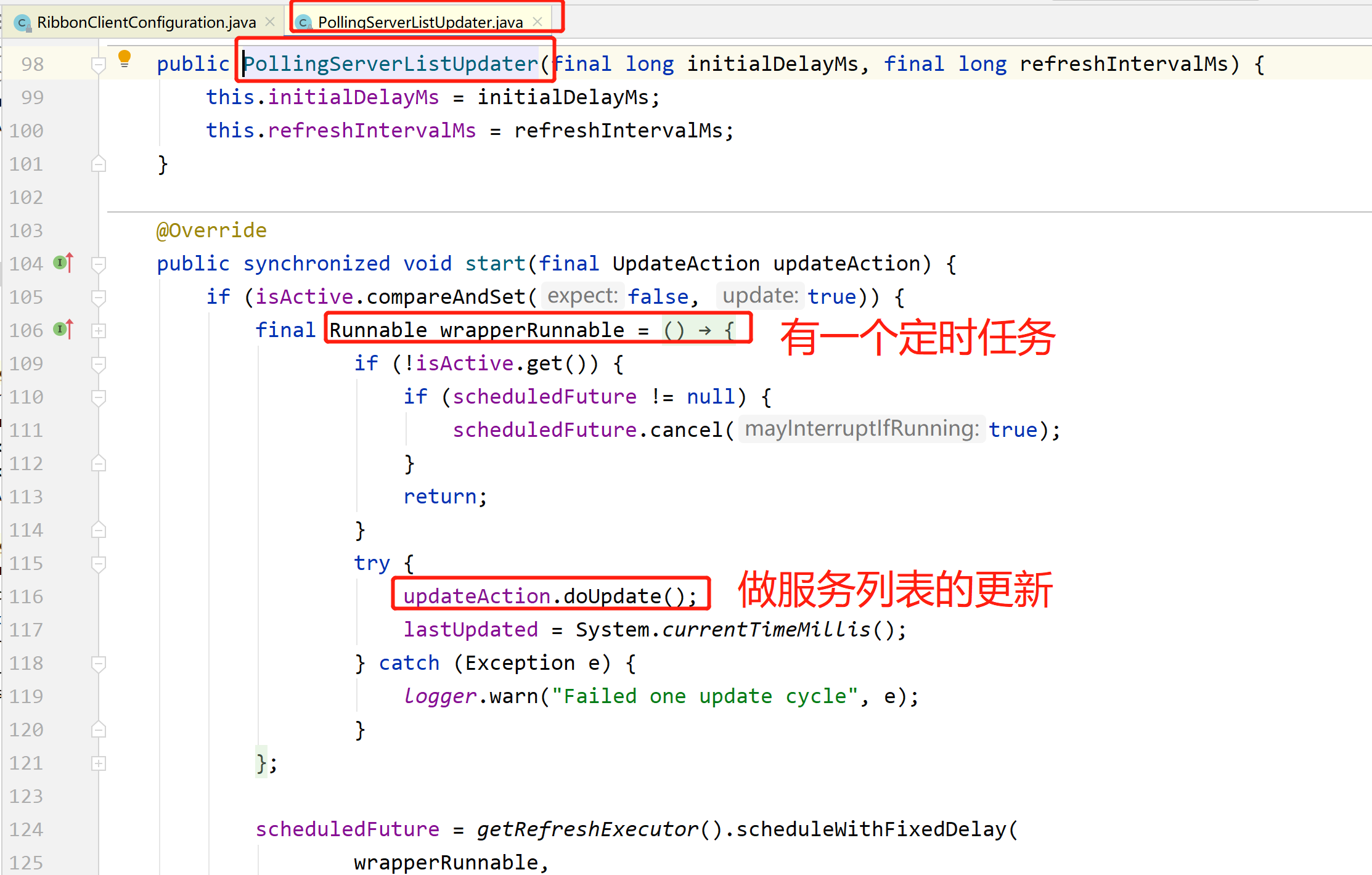




**查看notifyServerStatusChangeListener发现只是一个空壳的接口，并没有对缓存的服务节点做出是实际操作，那么到底在哪里修改了缓存列表的值呢？**

**我们发现在ribbon的配置类中RibbonClientConfiguration有一个更新服务列表的方法**





**定时任务在哪里开始执行的呢？我们查找doUpdate()方法**



**解决脏读机制的总结：**

1. **Ping**
2. **更新机制**

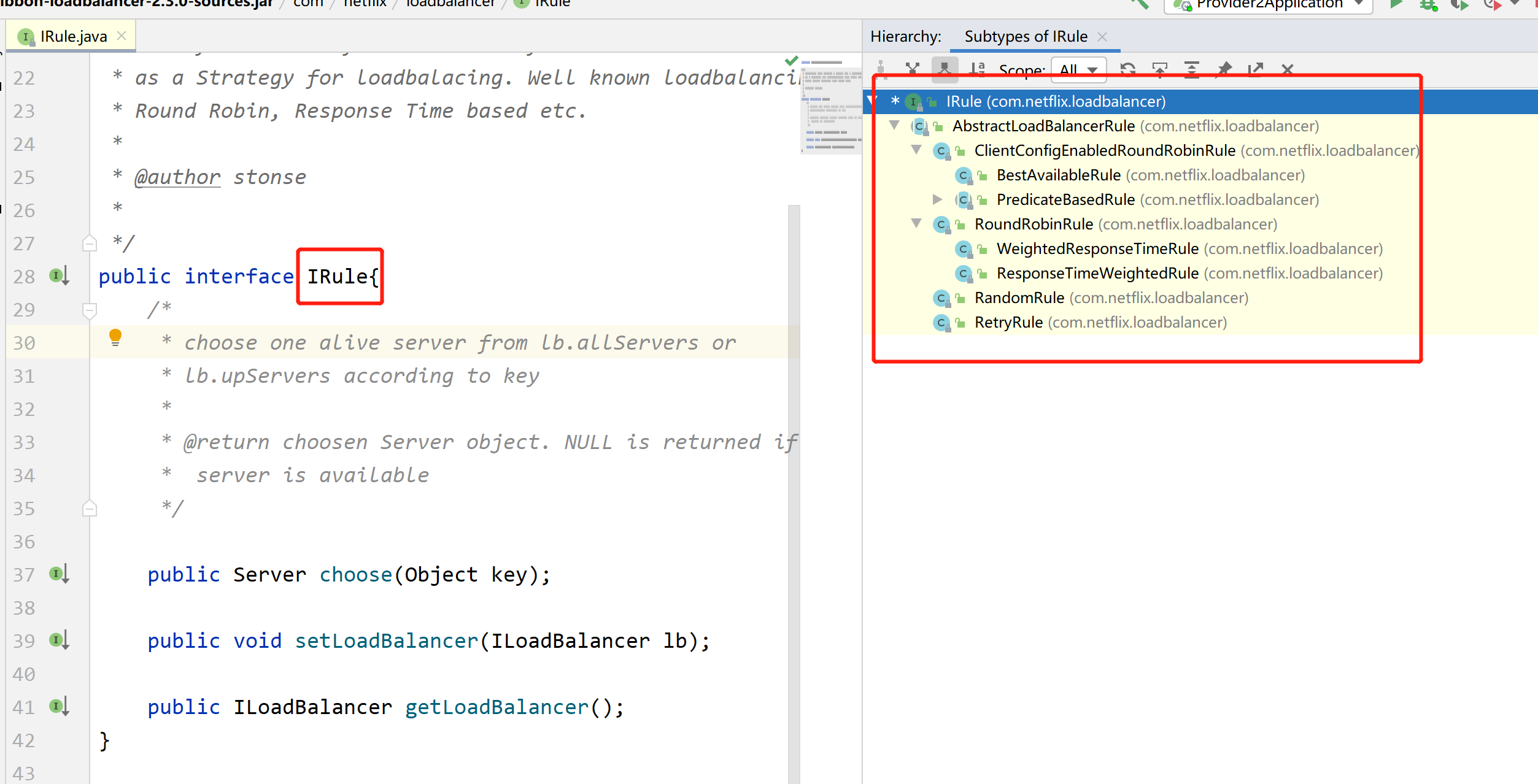
**都是为了解决脏读的现象而生的**

**测试发现：更新机制和ping 有个重回，而且在ping的时候不能运行更新机制，在更新的时候不能运行ping机制，导致我们很难测到ping失败的现象！**

**Ping机制做不了事情**

## Ribbon负载均衡的实现和几种算法

**在ribbon中有一个核心的负载均衡算法接口IRule**



**1.RoundRobinRule--轮询 请求次数 % 机器数量**

**2.RandomRule--随机**

**3.AvailabilityFilteringRule --会先过滤掉由于多次访问故障处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对于剩余的服务列表按照轮询的策略进行访问**

**4.WeightedResponseTimeRule--根据平均响应时间计算所有服务的权重，响应时间越快服务权重越大被选中的概率越大。刚启动时如果同统计信息不足，则使用轮询的策略，等统计信息足够会切换到自身规则**

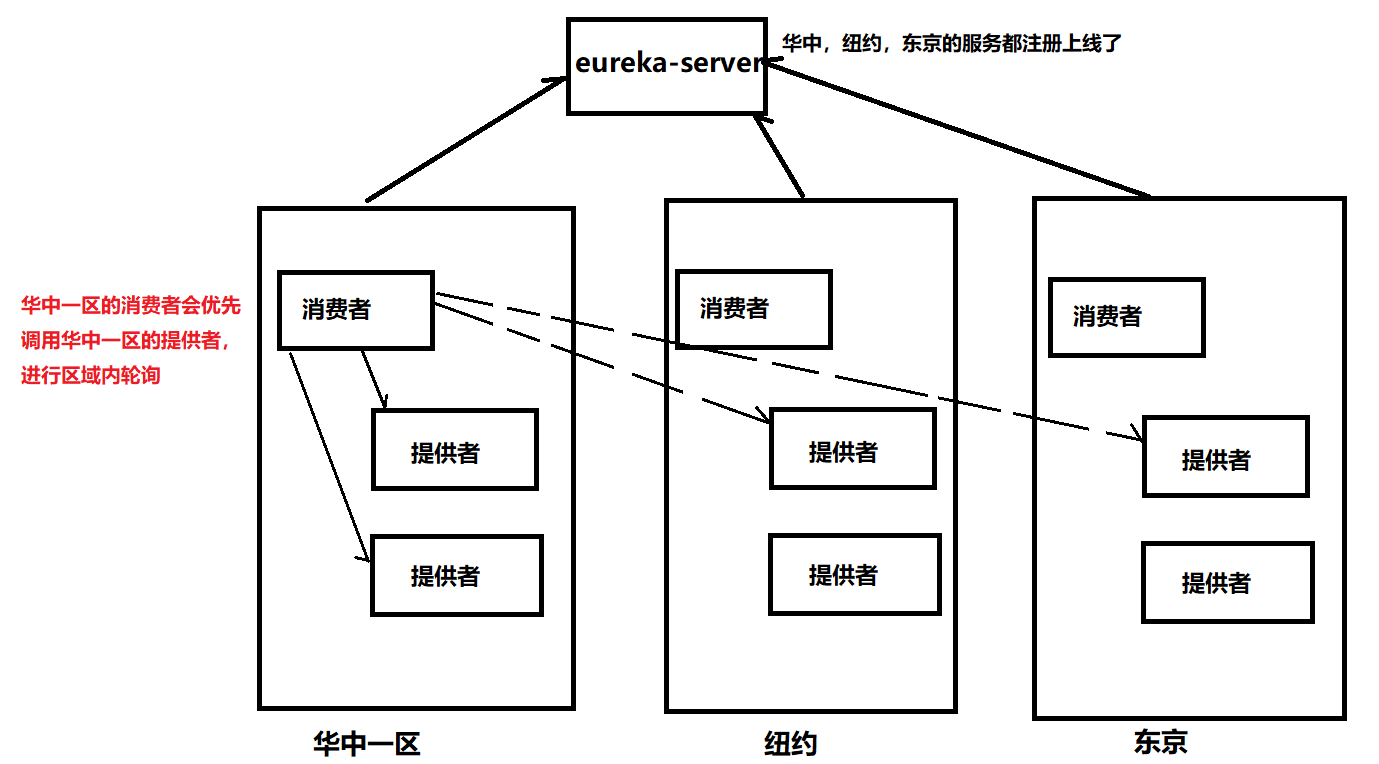
**5.RetryRule-- 先按照轮询的策略获取服务，如果获取服务失败则在指定的时间内会进行重试，获取可用的服务**

**6.BestAvailableRule --会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量小的服务**

**7.ZoneAvoidanceRule -- 默认规则，复合判断Server所在区域的性能和Server的可用行选择服务器。**

**Ribbon 默认使用哪一个负载均衡算法：**

**ZoneAvoidanceRule ：区间内轮询的算法！**

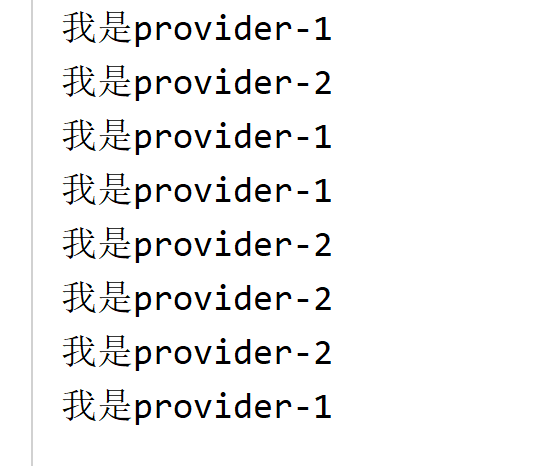


# 如何修改默认的负载均衡算法

## 修改yml配置文件（指定某一个服务使用什么算法）

|  |
| --- |
| provider: *#提供者的服务名称,那么访问该服务的时候就会按照自定义的负载均衡算法* ribbon:  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule *#几种算法的全限定类名* |

## 测试调用该服务（这里使用随机规则）

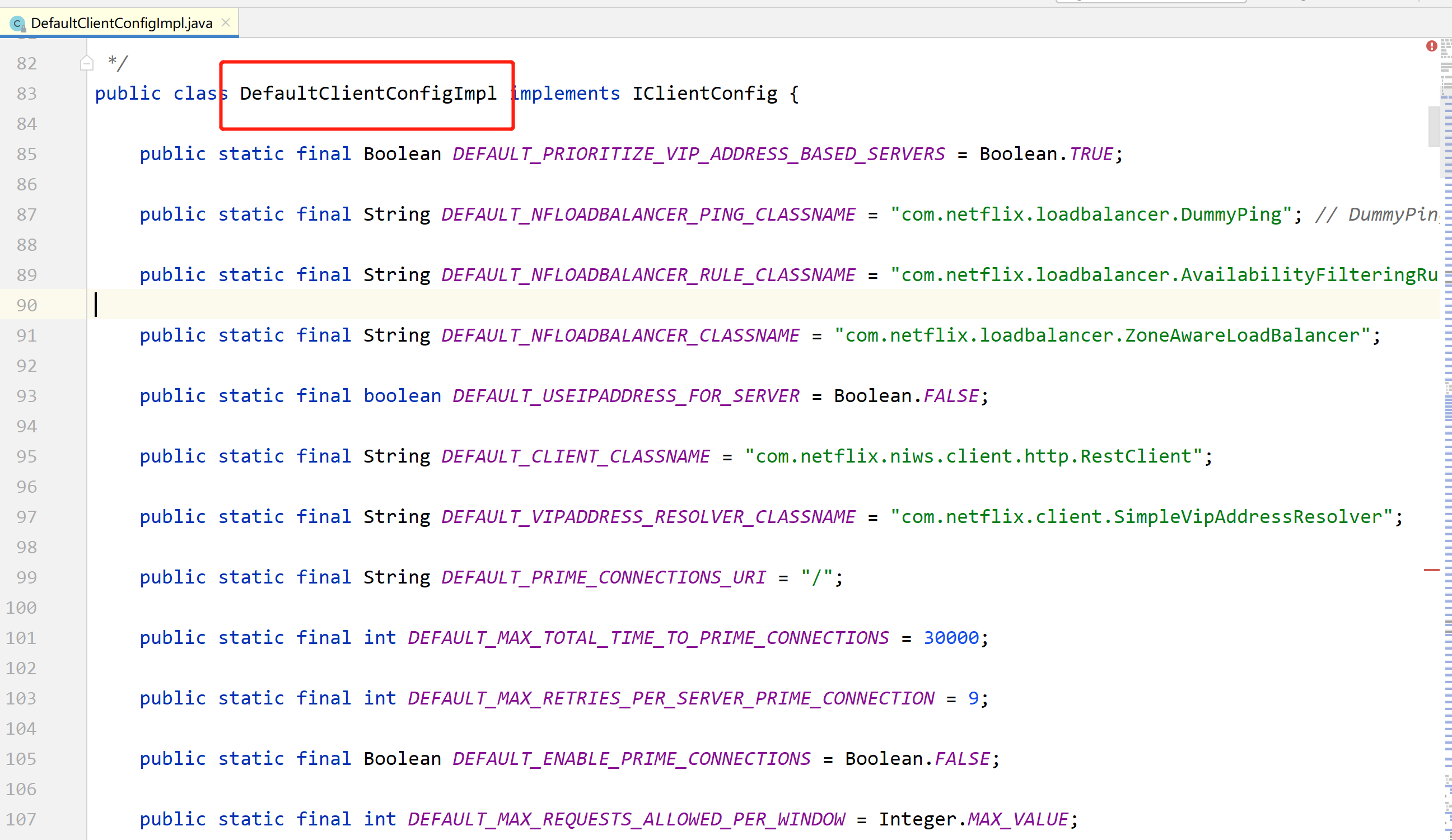


## 配置此消费者调用任何服务都用某种算法

|  |
| --- |
| @Bean public IRule myRule() {  *//指定调用所有的服务都用此算法* return new RandomRule(); } |

# Ribbon的配置文件和常用配置

**Ribbon有很多默认的配置，查看DefaultClientConfigImpl**



|  |
| --- |
| ribbon: *#全局的设置* eager-load:  enabled: false *# ribbon 一启动不会主动去拉取服务列表，当实际使用时才去拉取 是否立即加载* http:  client:  enabled: false *# 在ribbon 最后要发起Http的调用调用，我们认为是RestTemplate 完成的，其实最后是HttpURLConnection 来完成的，这里面设置为true ，可以把HttpUrlConnection->HttpClient* okhttp:  enabled: false *#HttpUrlConnection 来完成的，这里面设置为true ，可以把HttpUrlConnection->OkHttpClient(也是发http请求的，它在移动端的开发用的多)* provider: *#提供者的服务名称,那么访问该服务的时候就会按照自定义的负载均衡算法* ribbon:  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule *#修改默认负载均衡算法，几种算法的全限定类名 # NFLoadBalancerClassName: #loadBalance策略 # NFLoadBalancerPingClassName: #ping机制策略 # NIWSServerListClassName: #服务列表策略 # NIWSServerListFilterClassName: #服务列表过滤策略 ZonePreferenceServerListFilter 默认是优先过滤非一个区的服务列表* |

# Ribbon总结

**Ribbon是客户端实现负载均衡的远程调用组件，用法简单**

**Ribbon源码核心：**

**ILoadBalancer接口：起到承上启下的作用**

1. **承上：从eureka拉取服务列表**
2. **启下：使用IRule算法实现客户端调用的负载均衡**

**设计思想：每一个服务提供者都有自己的ILoadBalancer**

**userService---》客户端有自己的ILoadBalancer**

**TeacherService---》客户端有自己的ILoadBalancer**

**在客户端里面就是Map<String,ILoadBalancer> iLoadBalancers**

**如何实现负载均衡的呢？**

**iloadBalancer loadbalance = iloadBalancers.get(“user-service”)**

**List<Server> servers = Loadbalance.getReachableServers();//缓存起来**

**Server server = loadbalance .chooseServer(key) //key是区id，--》IRule算法**

**chooseServer下面有一个IRule算法**

IRule下面有很多实现的负载均衡算法