# Ribbo的源码分析

## 3.1 Ribbon 要做什么 事情？

我们通过：

|  |
| --- |
| http://provider/info |

就已经完成了对provider的调用 ，那我们思考自己该怎么实现？

## 3.2执行请求

|  |
| --- |
| String result = rest.getForObject(http://provider/info, String.**class**); |

要把上面的请求执行成功,需要多少步？

1: 拦截该请求

2 获取本次请求里面 访问的url:http://provider/info

3 截取该url 地址里面服务名称provider

4 通过服务的名称得到服务的列表（服务发现的过程）List<ServiceInstance> instances

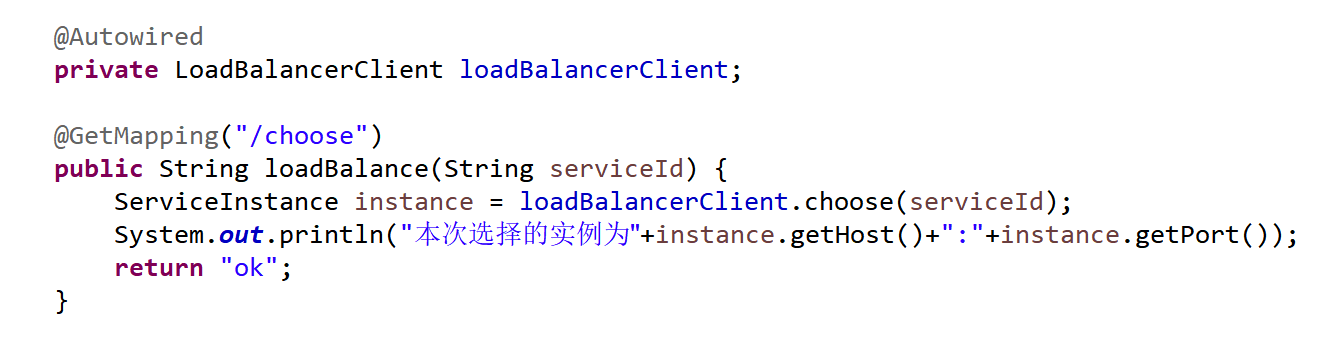
5 从该列表里面选一个处理 instance =loadbalance(instances);

6 重构该url地址

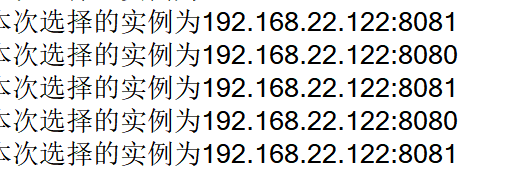
|  |
| --- |
| String result = rest.getForObject(http://ip1:port1/info, String.**class**); |

7 继续使用RestTemplate 来发起请求

## 3.3 Ribbon的负载均衡的测试

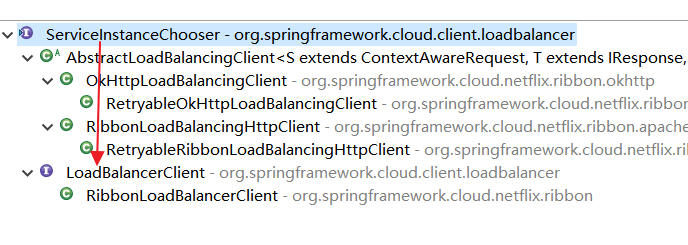


## 3.4 测试该选择的过程

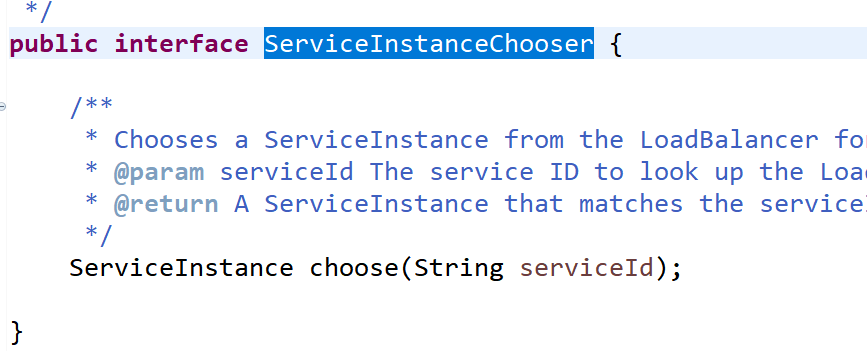


选择过程就是负载均衡的过程，也是Ribbon的核心过程

## 3.5 LoadBalancerClient



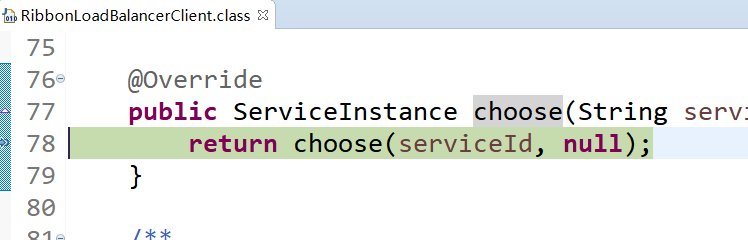
LoadBalancerClient 继承 ServiceInstanceChooser，而它里面有个方法：

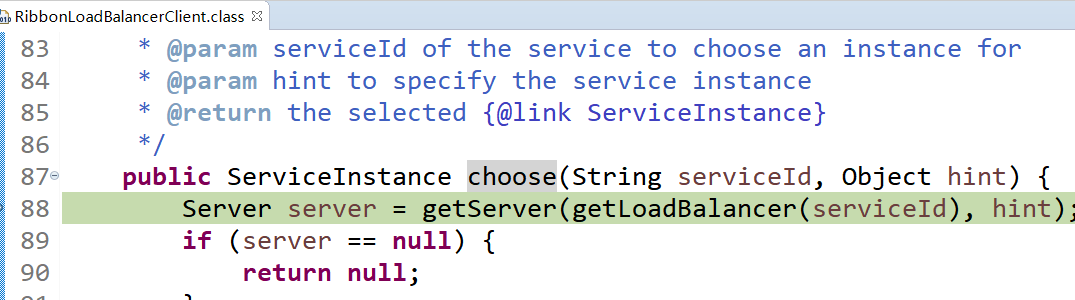


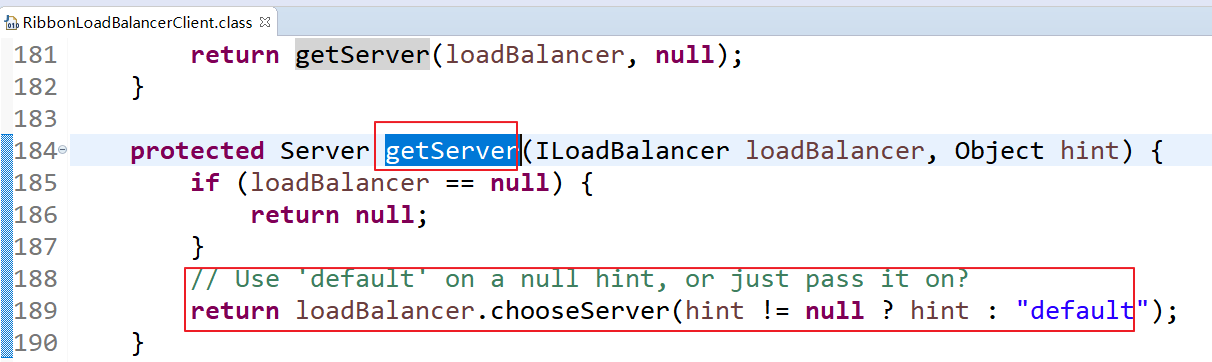
Ioc 里面默认就已经注入了：对象，该对象实现LoadBalancerClient 来实现里面choose方法

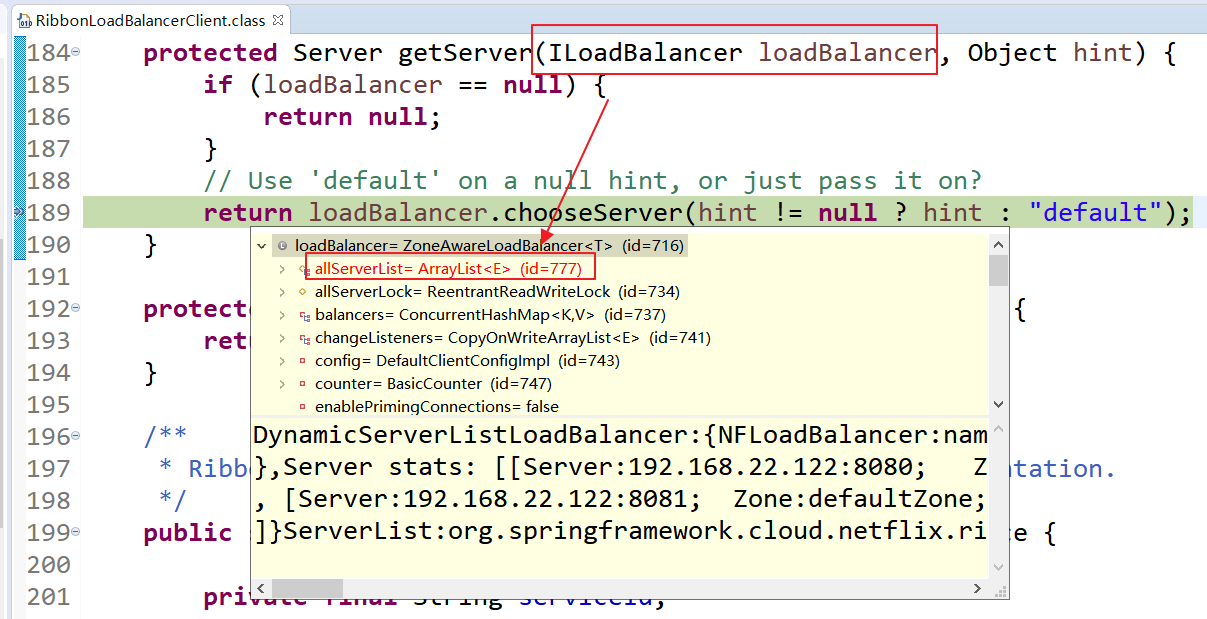
|  |
| --- |
| RibbonLoadBalancerClient |

## 3.6 RibbonLoadBalancerClient.choose()方法

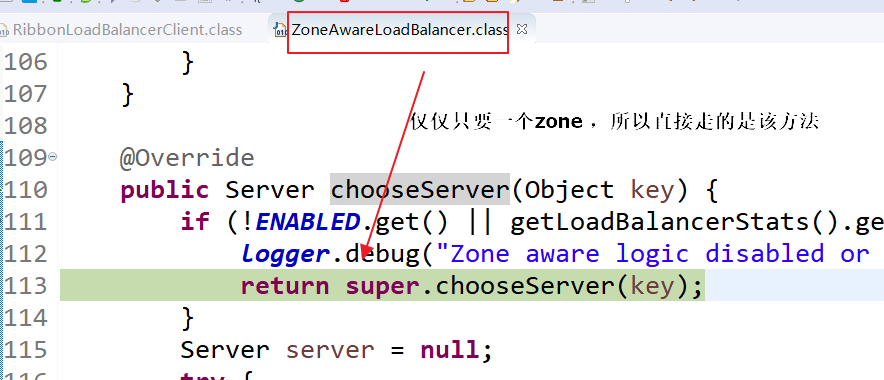


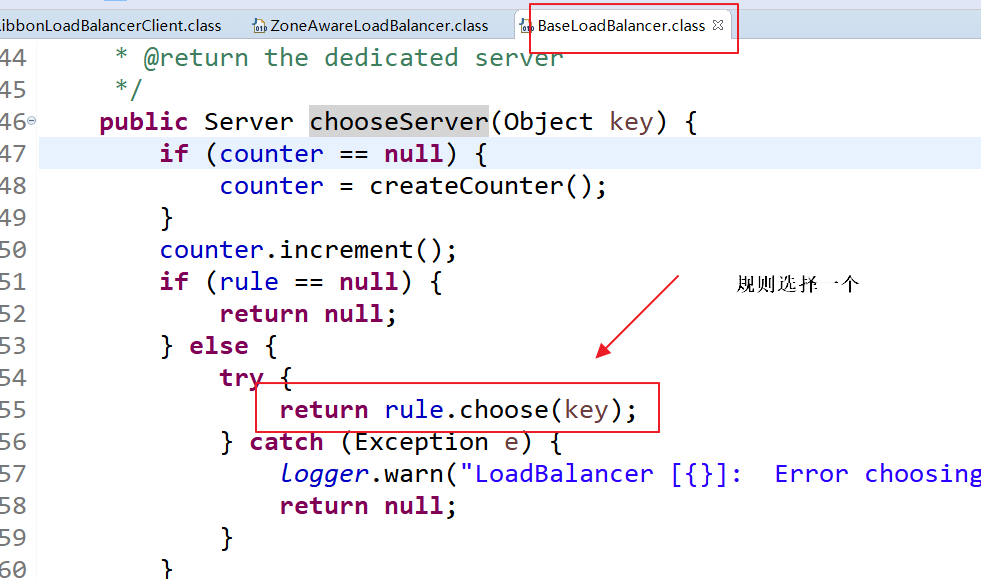




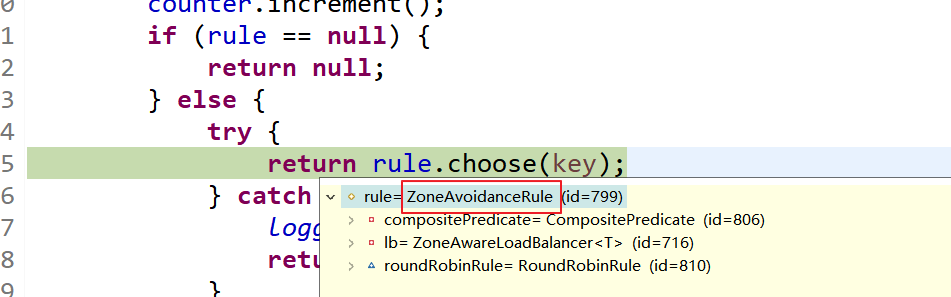


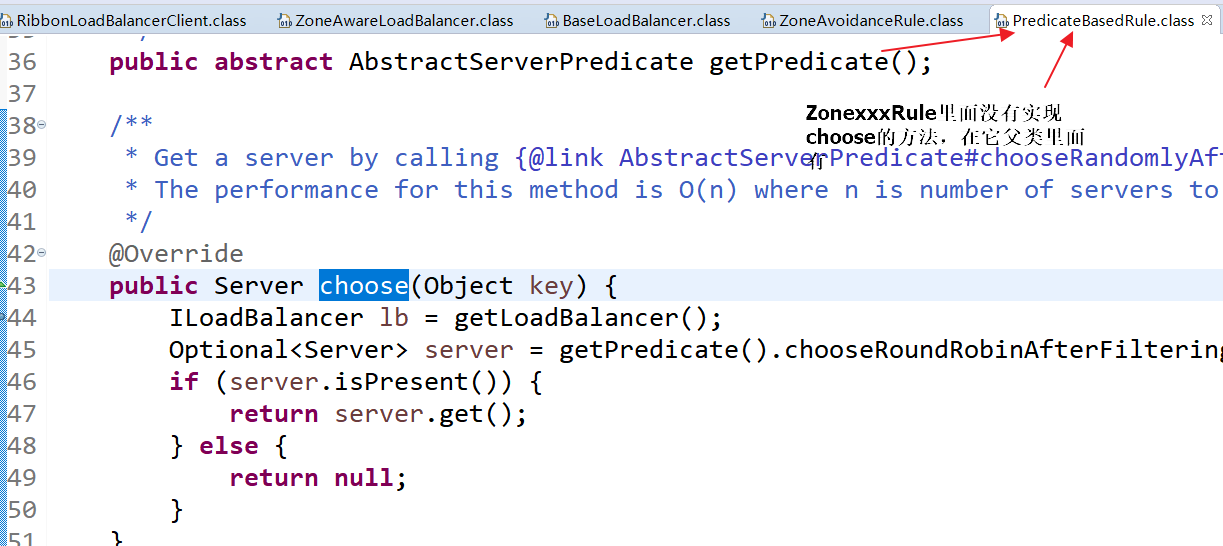
此时 loadBalancer 已经有所有的服务的列表了



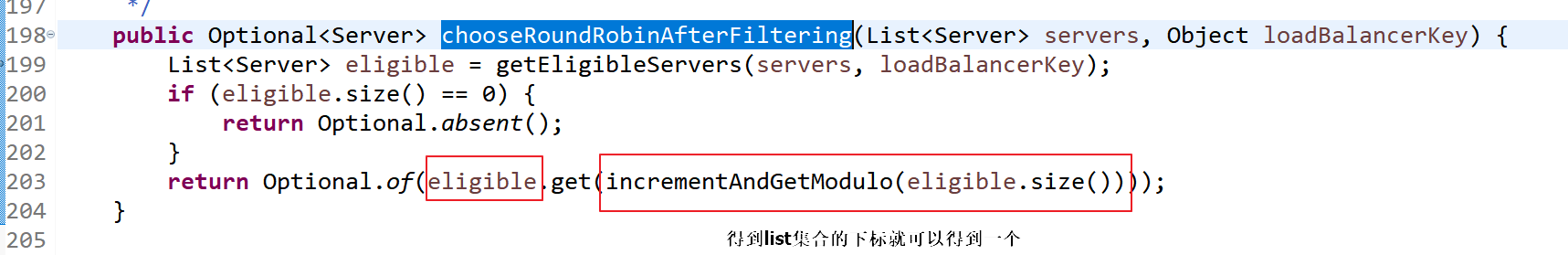


利用该选择算法：





最后使用 轮询算法选择了一个



## 3.7 思考？

1 在选择之前的服务列表来自哪里？

来自ILoadBalance

2 选择的算法该怎么自定义？

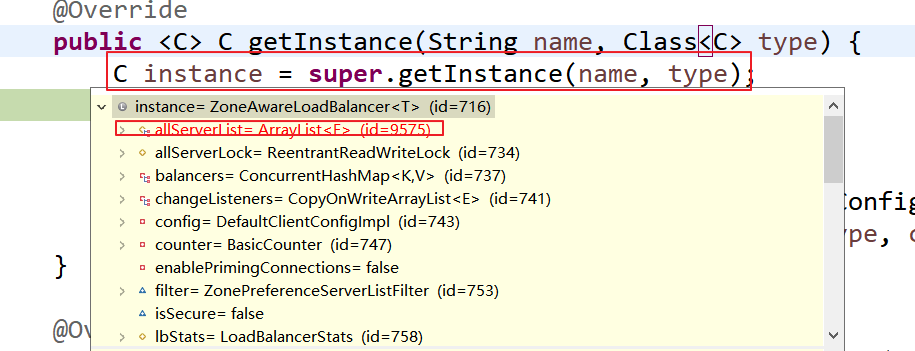
3 ILoadBalancer 有什么意义？

ILoadBalancer 是最关键的一个类，该类是这样的设计的数据结果

每一个服务的提供者都有一个ILoadBalancer

Map<String,ILoadBalancer >

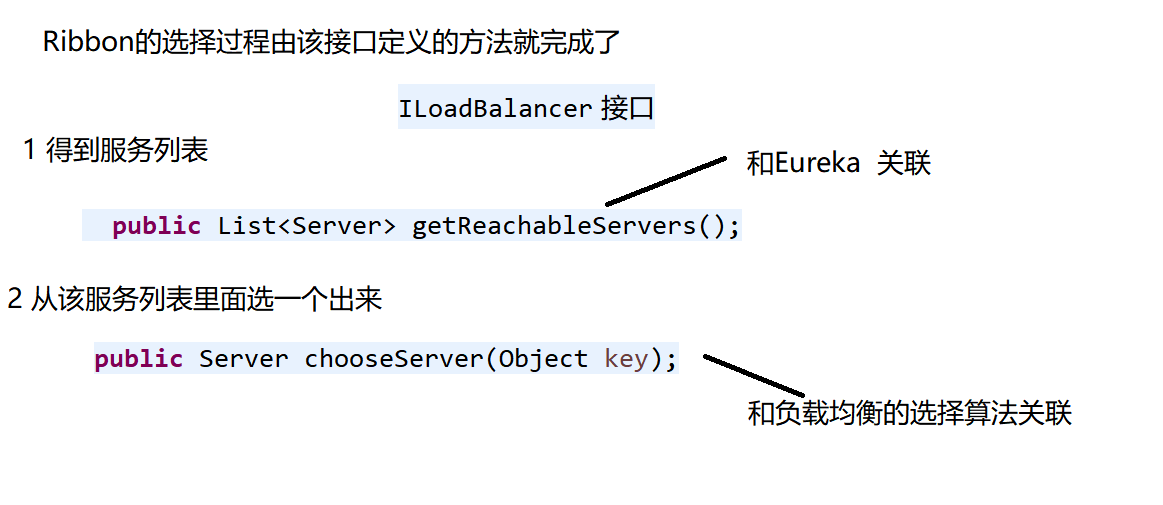
而ILoadBalancer 里面就已经有 该服务的列表了：



## 3.8 ILoadBalance的源码注解(Ribbon里面最 核心的组件)

Map<String,ILoadBalance>

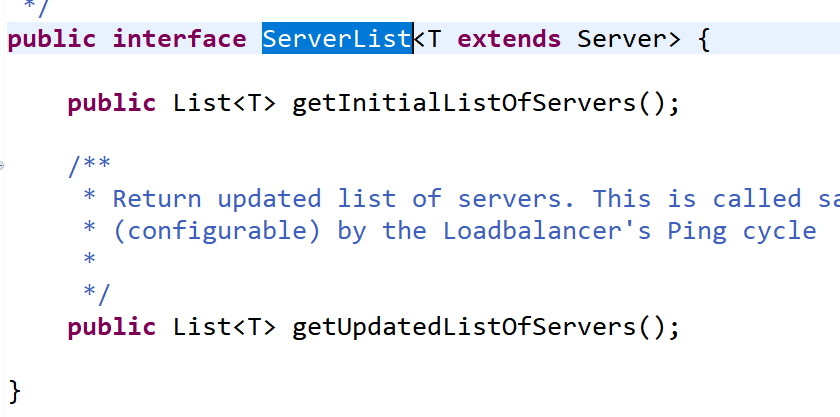
|  |
| --- |
| **public** **interface** ILoadBalancer {  // 往该ILoadBalancer添加该对应服务的地址  **public** **void** addServers(List<Server> newServers);    // 选择一个可以调用的实例 ，该key并不是服务的名称，而是区的id（暂时没有接触）  **public** Server chooseServer(Object key);    // 标记下线的服务  **public** **void** markServerDown(Server server);    // 得到该服务可以的服务列表  **public** List<Server> getReachableServers();  // 得到所以的服务列表  **public** List<Server> getAllServers();  } |



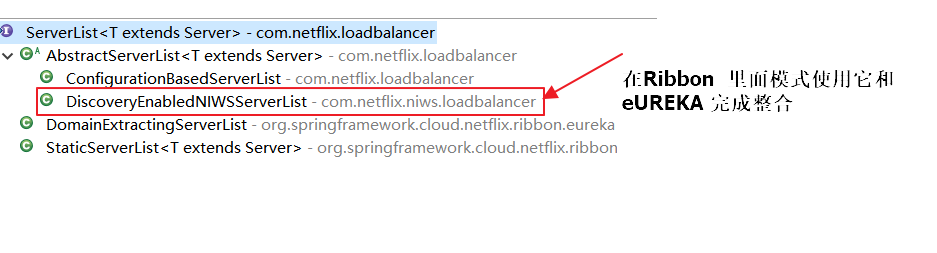
## 3.9 Ribbon 和Eureka的关联(DynamicServerListLoadBalancer)

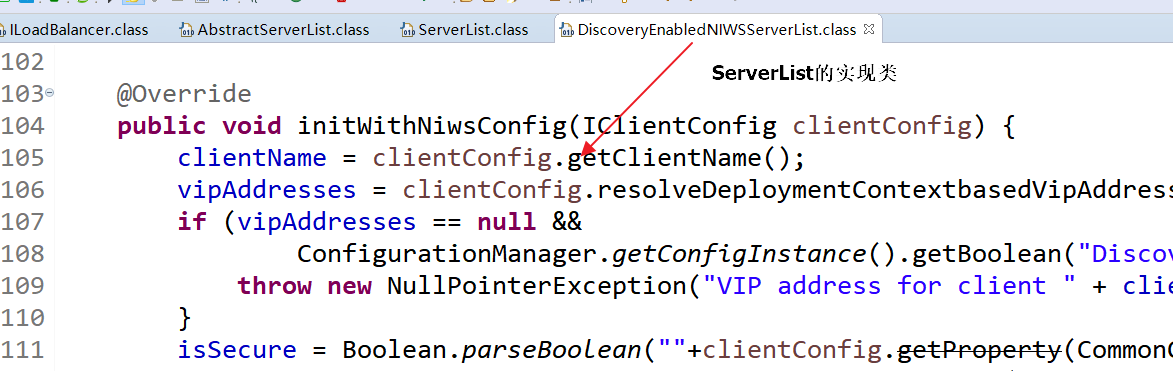
Ribbon 在选择之前需要获取服务列表，ribbon本身没有服务发现的功能，而erueka有该功能，说明ribbon 要解决eureka 来解决获取服务列表的问题！

### 3.9.1 ServiceList 的获取

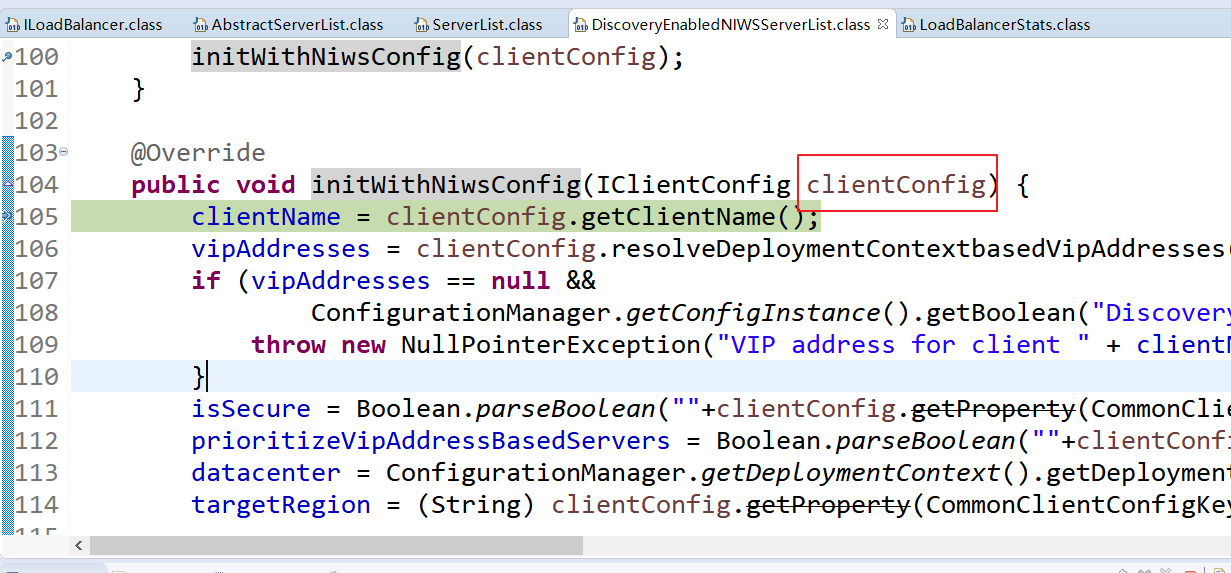


里面定义的解决说明了我该怎么获取服务的列表

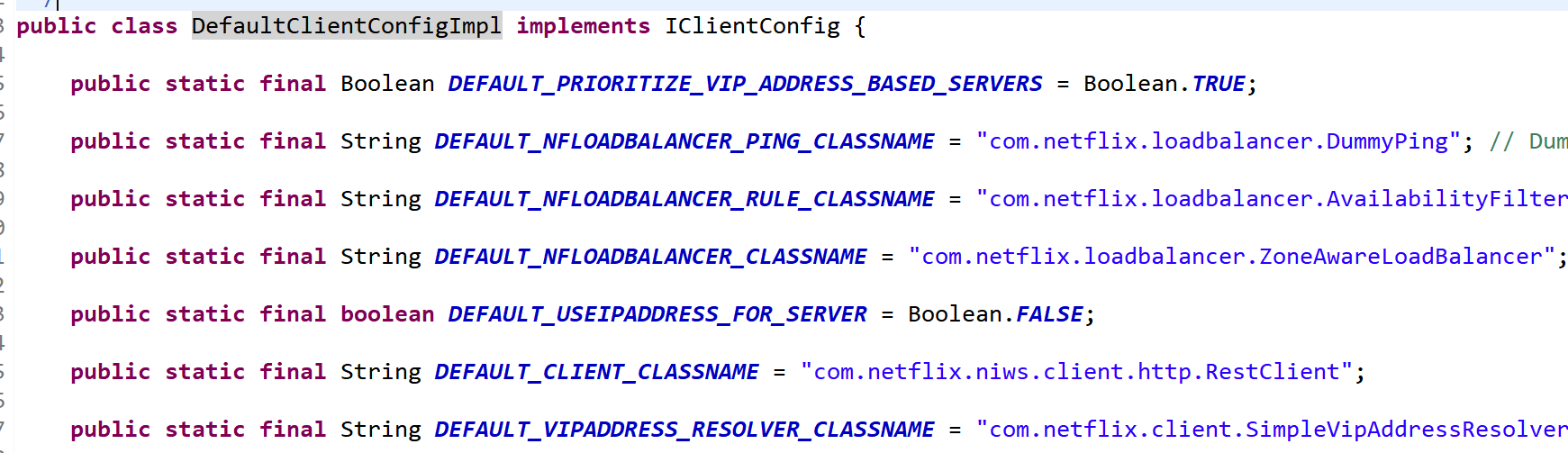




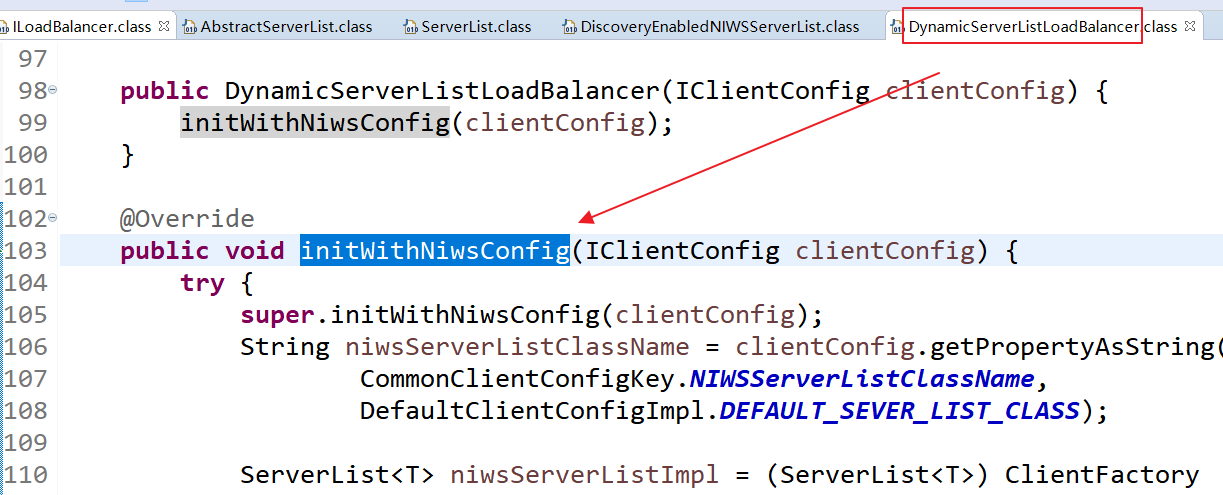
启动就会执行该方法



该类就是以后我们要使用的一些默认的配置类

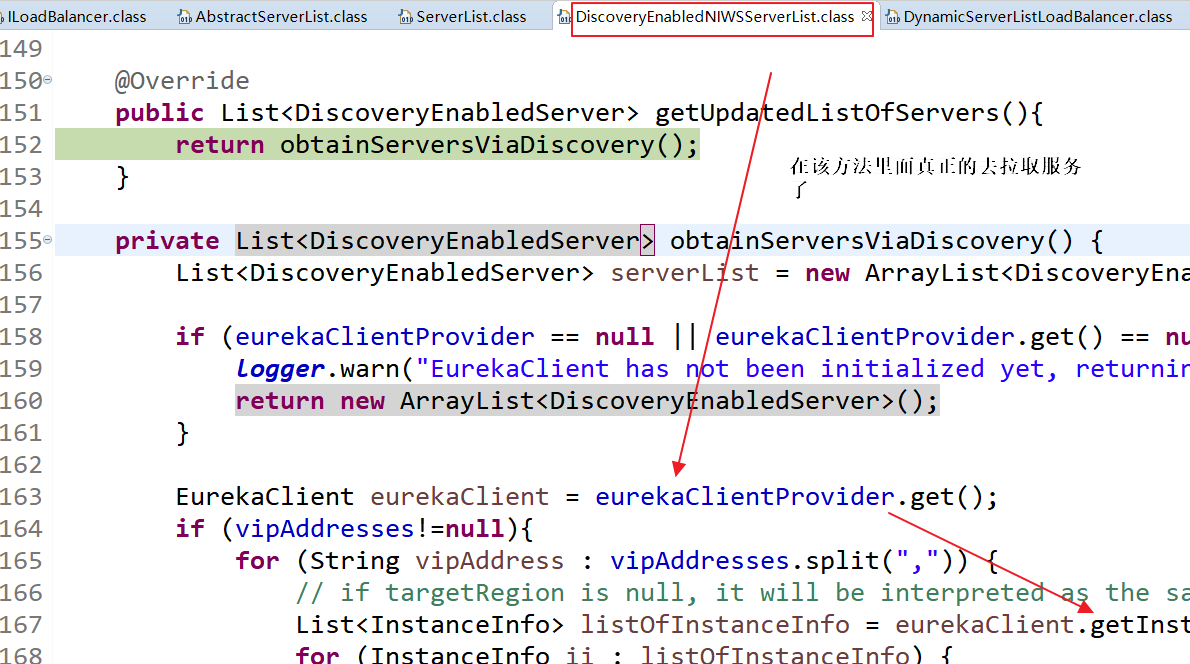


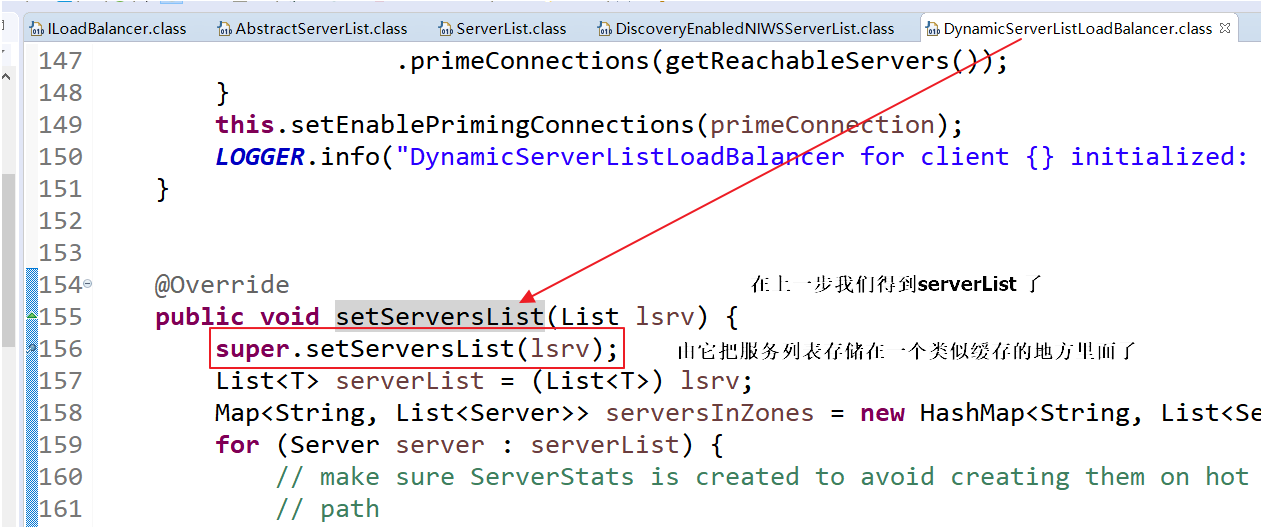
Ribbbo 什么时候去Eureka 里面拉取服务列表？初始化DynamicServerListLoadBalancer 类时执行的服务列表的拉取：



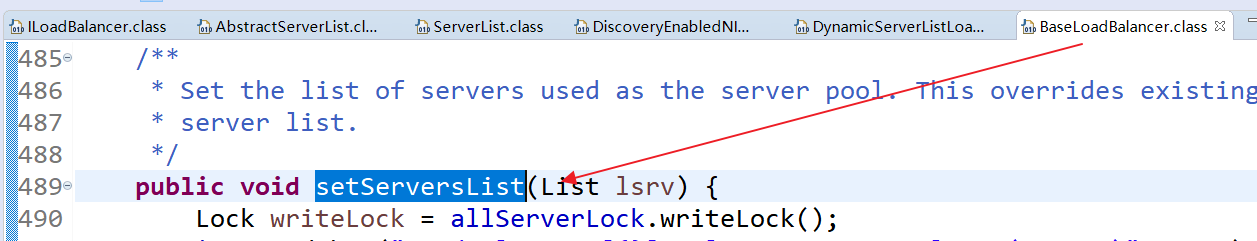
具体在该方法里面的：

|  |
| --- |
| @Override  **public** **void** initWithNiwsConfig(IClientConfig clientConfig) {  **try** {  **super**.initWithNiwsConfig(clientConfig);  String niwsServerListClassName = clientConfig.getPropertyAsString(  CommonClientConfigKey.***NIWSServerListClassName***,  DefaultClientConfigImpl.***DEFAULT\_SEVER\_LIST\_CLASS***);  ServerList<T> niwsServerListImpl = (ServerList<T>) ClientFactory  .*instantiateInstanceWithClientConfig*(niwsServerListClassName, clientConfig);  **this**.serverListImpl = niwsServerListImpl;  **if** (niwsServerListImpl **instanceof** AbstractServerList) {  AbstractServerListFilter<T> niwsFilter = ((AbstractServerList) niwsServerListImpl)  .getFilterImpl(clientConfig);  niwsFilter.setLoadBalancerStats(getLoadBalancerStats());  **this**.filter = niwsFilter;  }  String serverListUpdaterClassName = clientConfig.getPropertyAsString(  CommonClientConfigKey.***ServerListUpdaterClassName***,  DefaultClientConfigImpl.***DEFAULT\_SERVER\_LIST\_UPDATER\_CLASS***  );  **this**.serverListUpdater = (ServerListUpdater) ClientFactory  .*instantiateInstanceWithClientConfig*(serverListUpdaterClassName, clientConfig);  执行全部服务的获取  restOfInit(clientConfig);  } **catch** (Exception e) {  **throw** **new** RuntimeException(  "Exception while initializing NIWSDiscoveryLoadBalancer:"  + clientConfig.getClientName()  + ", niwsClientConfig:" + clientConfig, e);  }  } |

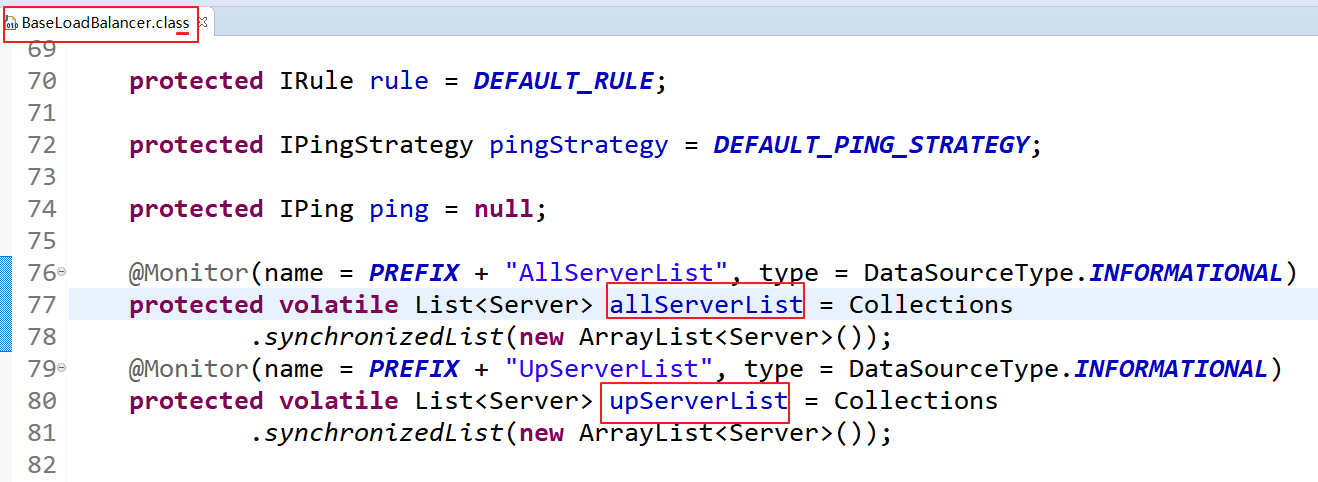




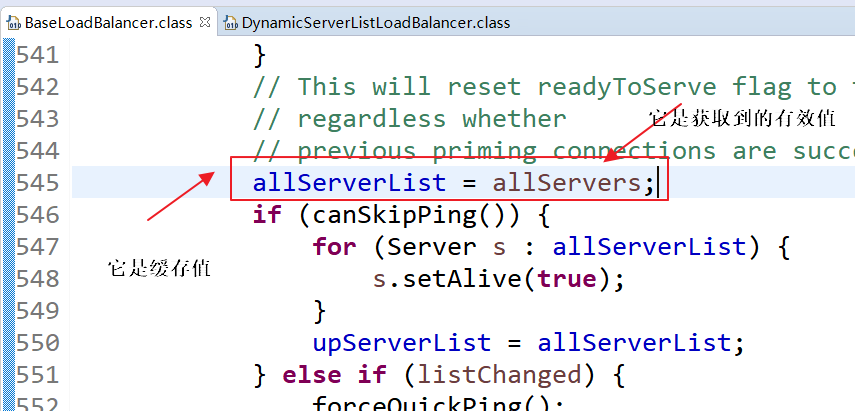
Super就是下面的类的调用



先关注上面的2 个容器



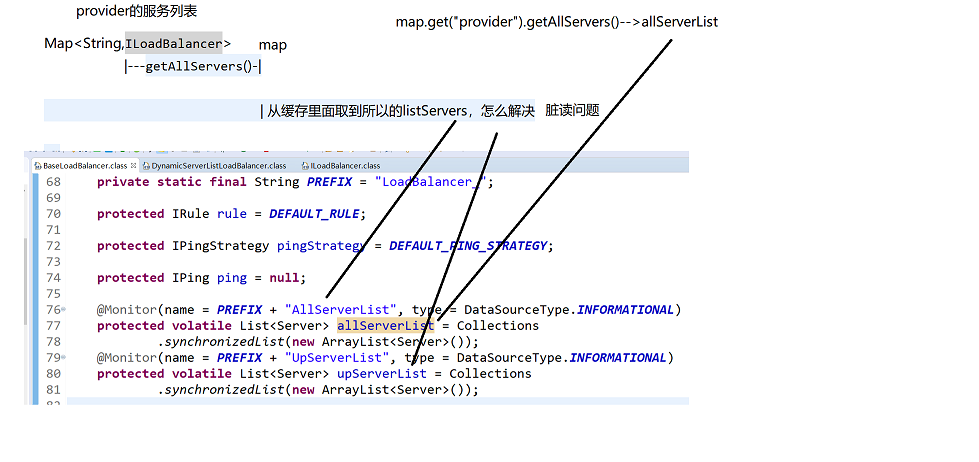
该集合在：

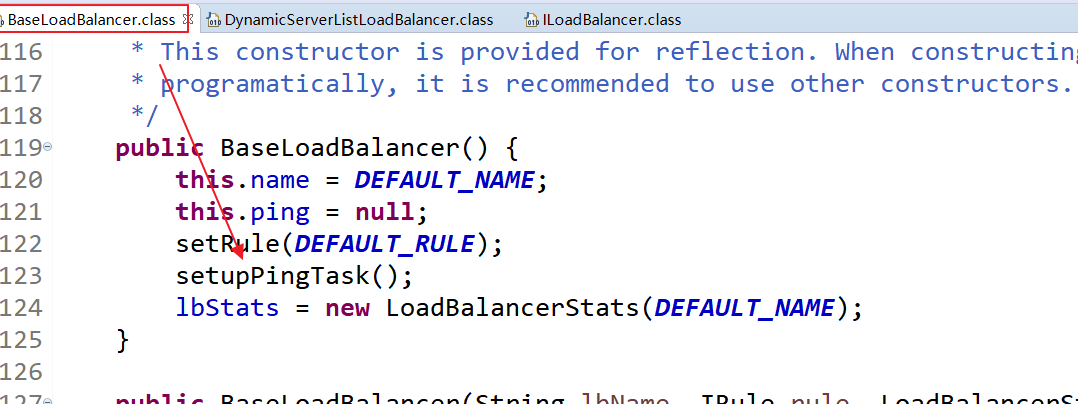


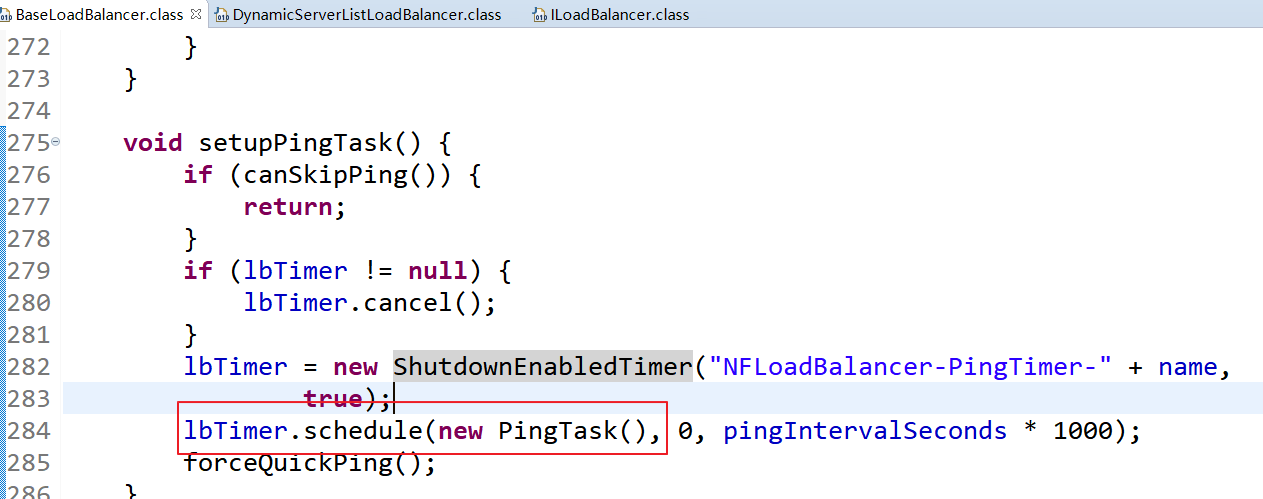
被修改了

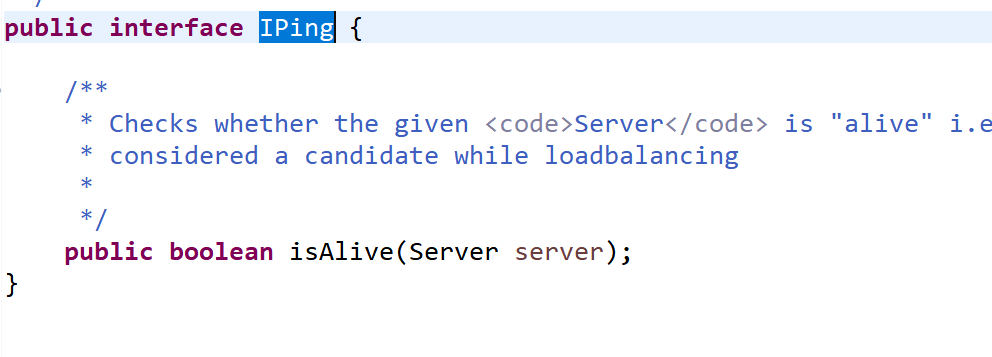
### 3.9.2 ribbon把serverList值缓存起来了，放在一个List 集合里面

缓存的脏读怎么解决!





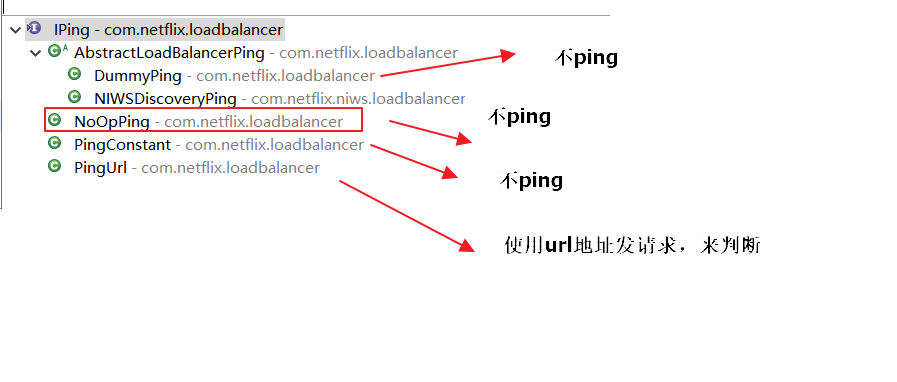




我从Eureka 得到了一系列机器，我把它缓存起来，我以后调用时，我不知道它给我们的机器是否依旧存活，所以我搞了一个ping 机制，使用定时任务，隔一段实际就去ping 一次 里面的机器，看它是否存活

IPing:

|  |
| --- |
| **public** **boolean** isAlive(Server server); |

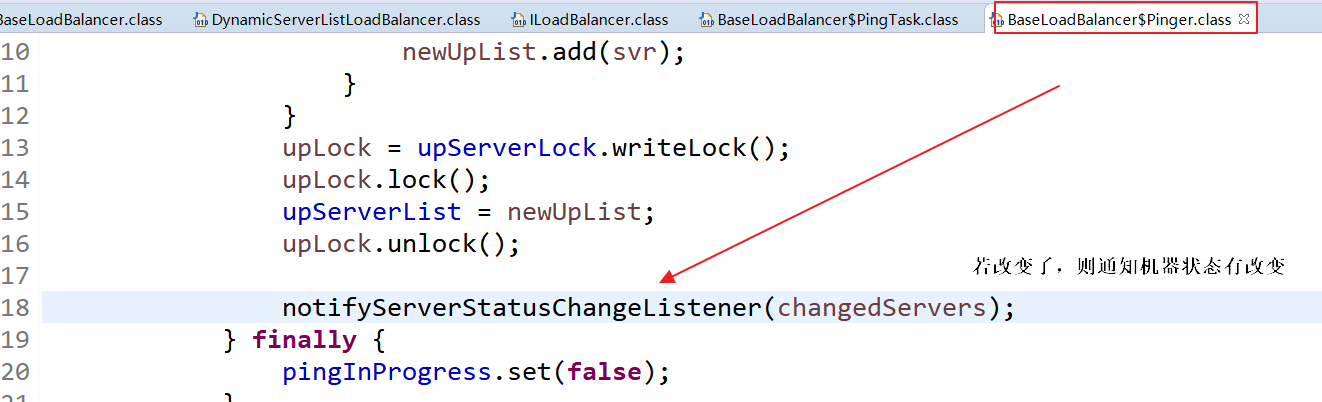


在ping机制里面，默认使用NiwDisPing机制：

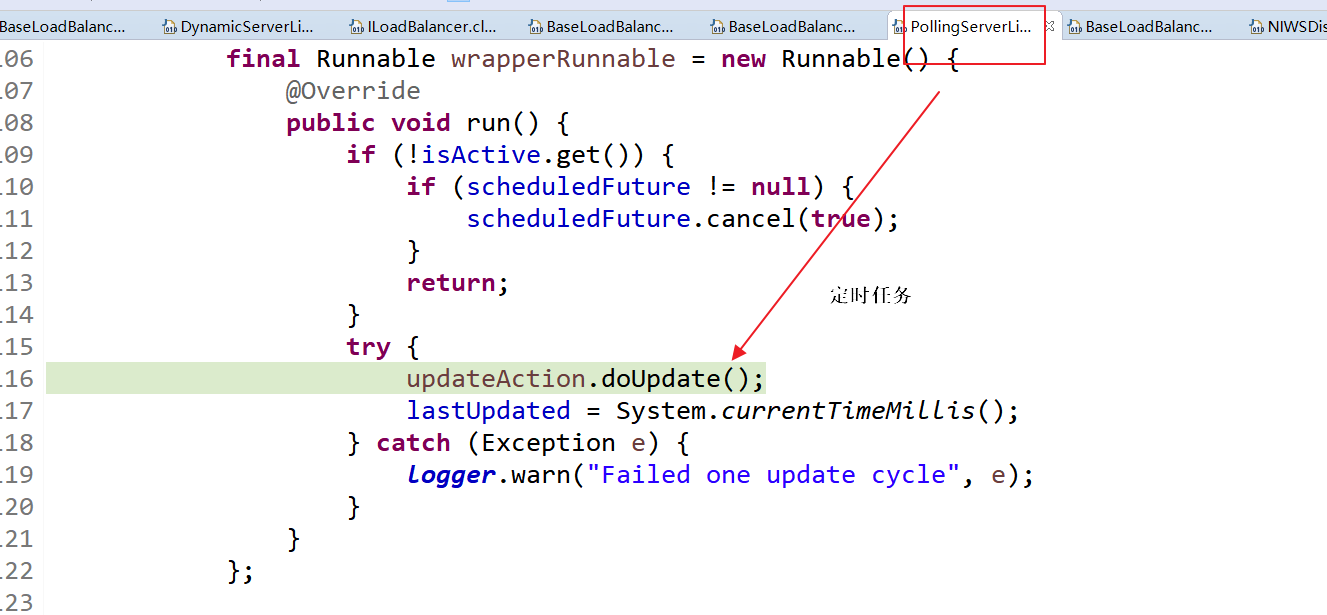
它的ping 操作为：

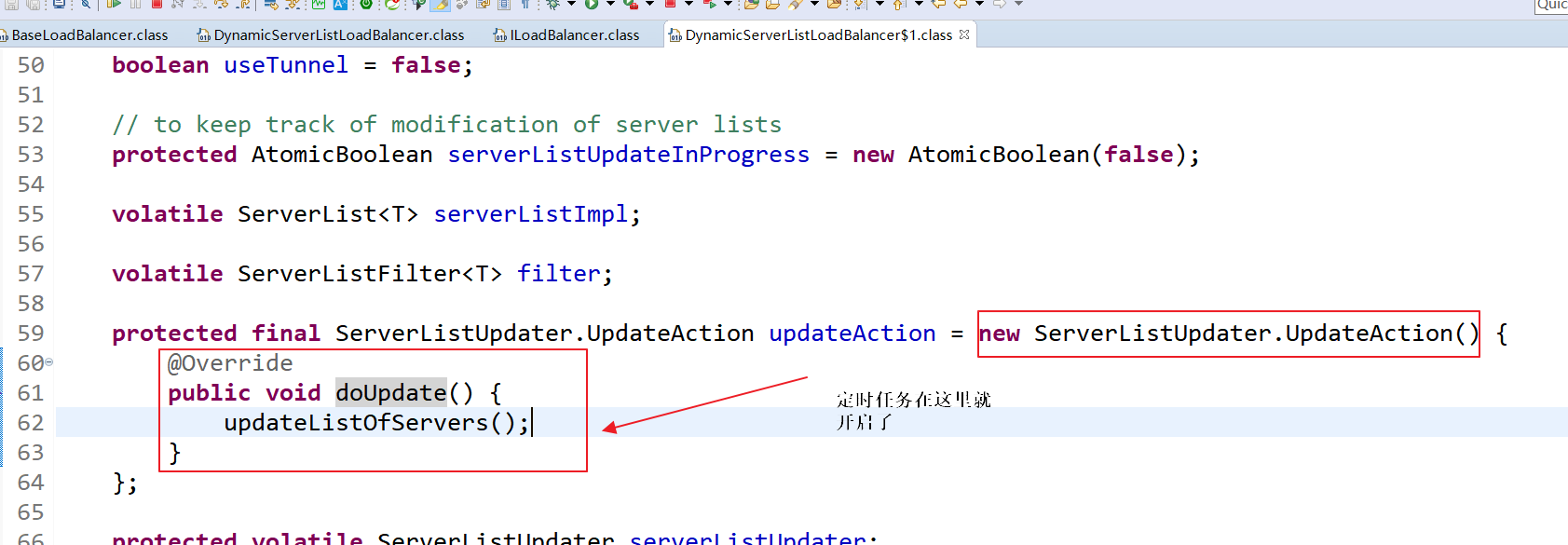
|  |
| --- |
| **public** **boolean** isAlive(Server server) {  **boolean** isAlive = **true**;  **if** (server!=**null** && server **instanceof** DiscoveryEnabledServer){  DiscoveryEnabledServer dServer = (DiscoveryEnabledServer)server;  // 从eureka 去查询该实例的状态  InstanceInfo instanceInfo = dServer.getInstanceInfo();  **if** (instanceInfo!=**null**){  //如果eureka里面显示该实例在线，我就返回true，否则为false  InstanceStatus status = instanceInfo.getStatus();  **if** (status!=**null**){  isAlive = status.equals(InstanceStatus.***UP***);  }  }  }  **return** isAlive;  } |

若ping 到有机器状态改变了，则发通知



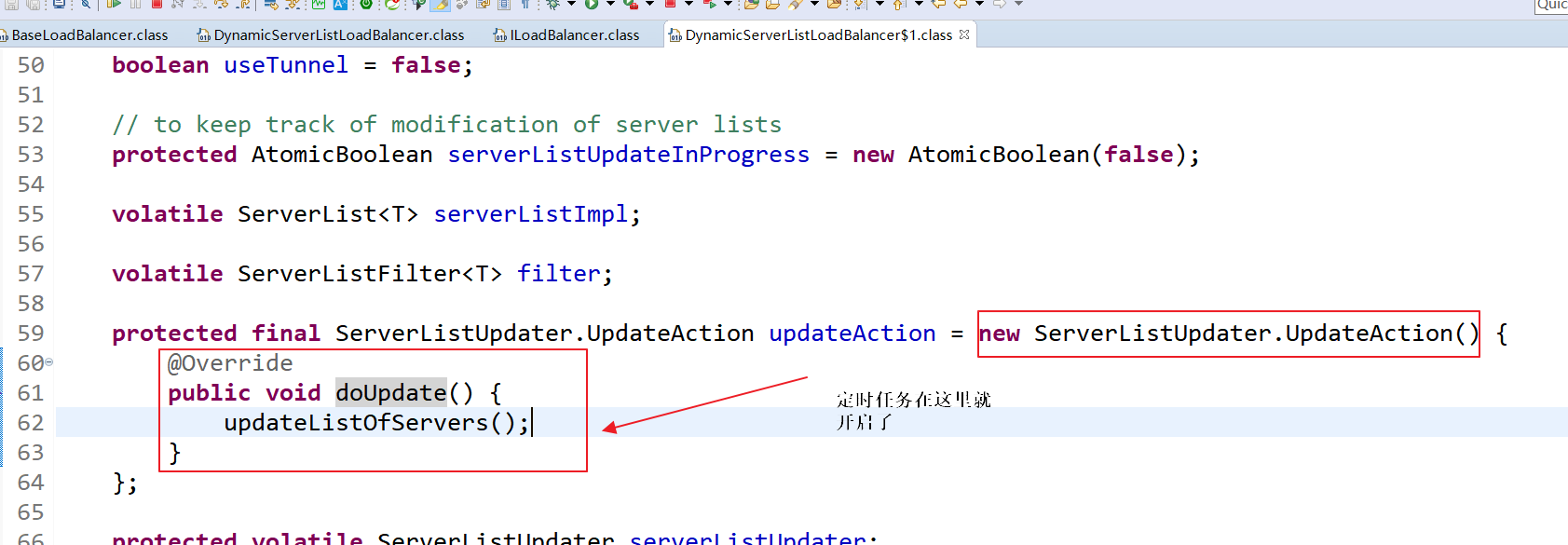
但是我们关闭一个机器后，发现ping 机制还没有来得及ping ，就有人把缓存值修改了，是这个人修改了：PollingServerListUpdater





解决脏读有2 种策略：

#### 1 更新机制

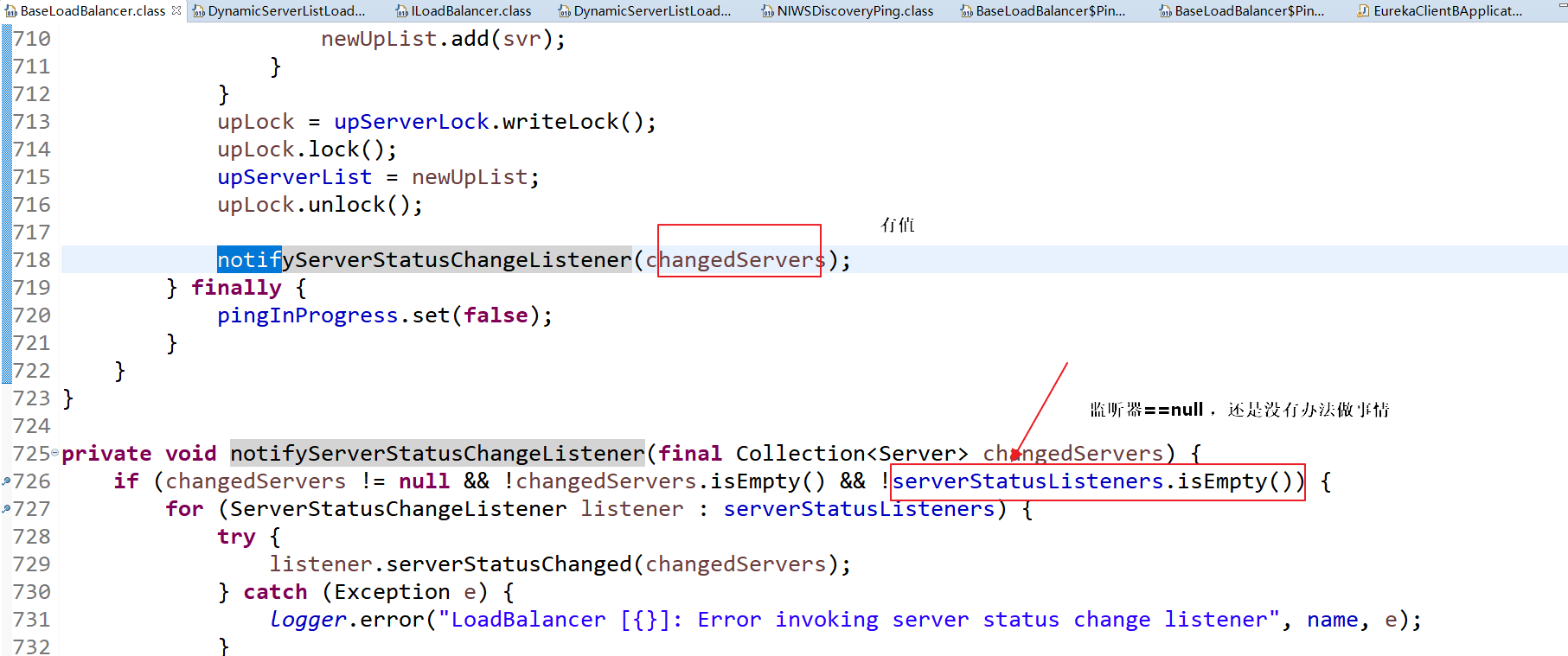


#### 2 ping 机制

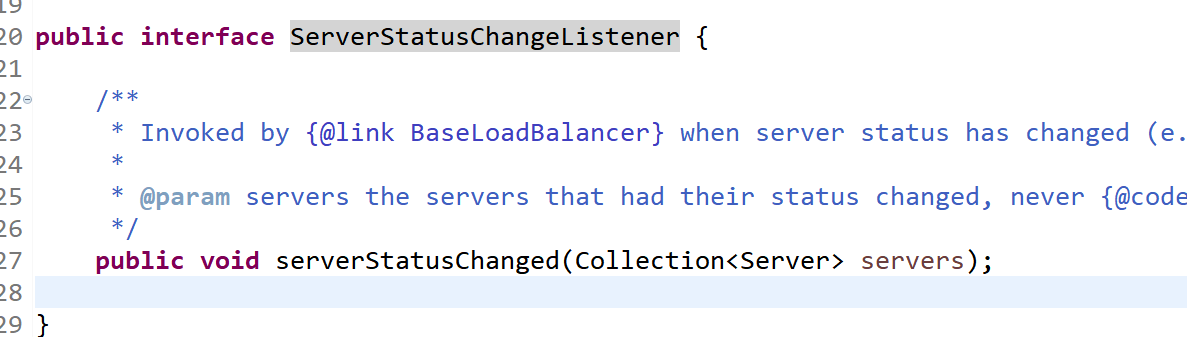
都是为了解决脏读的现象而生的

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！

我们后面通过修改ping的结果，导致有false 但是我们又发现，虽然是falase ，事件也要发了，但是没有监听器。



并且监听器没有实现类也没有匿名实现类：

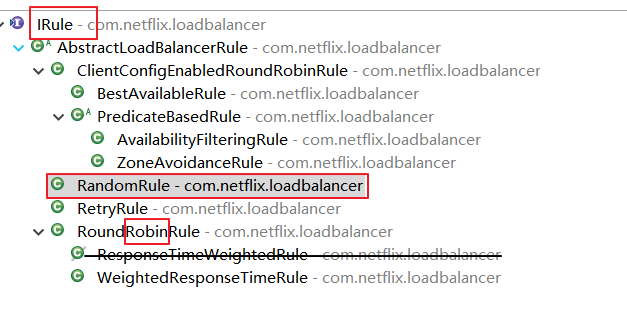


所以ping 机制做不了事情

## 3.10 Ribbon 真正的实现负载均衡算法的实现

我们在一步源码里面获取到了服务的列表，接下来我们从list 集合里面选一个出来，

IRule



1.RoundRobinRule--轮询

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

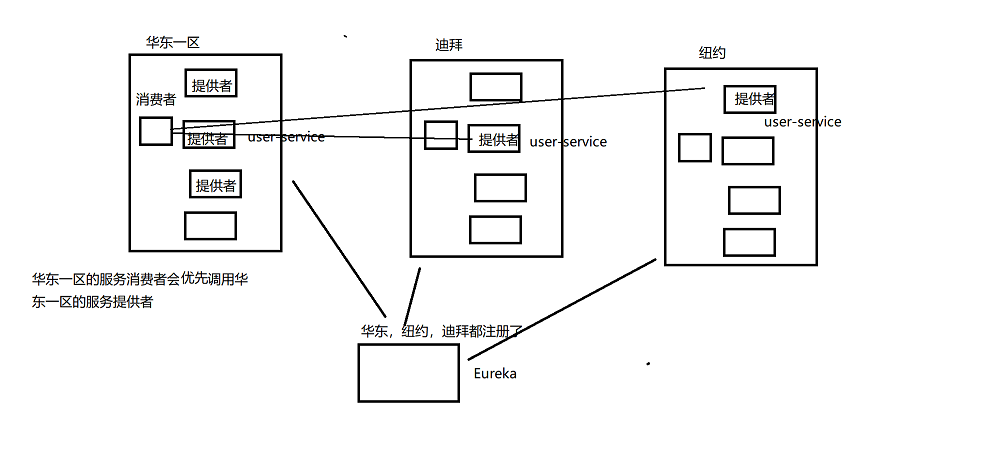
5.RetryRule-- 先按照轮询的策略获取服务，如果获取服务失败则在指定的时间内会进行重试，获取可用的服务

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

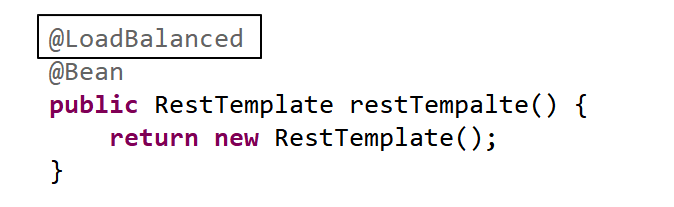
7.ZoneAvoidanceRule -- 默认规则，复合判断Server所在区域的性能和Server的可用行选择服务器。

Ribbon 默认使用哪一个负载均衡算法：

ZoneAvoidanceRule ：区间内轮询的算法！



# 为什么在LoadBalance 添加注解后就有负载均衡效果了



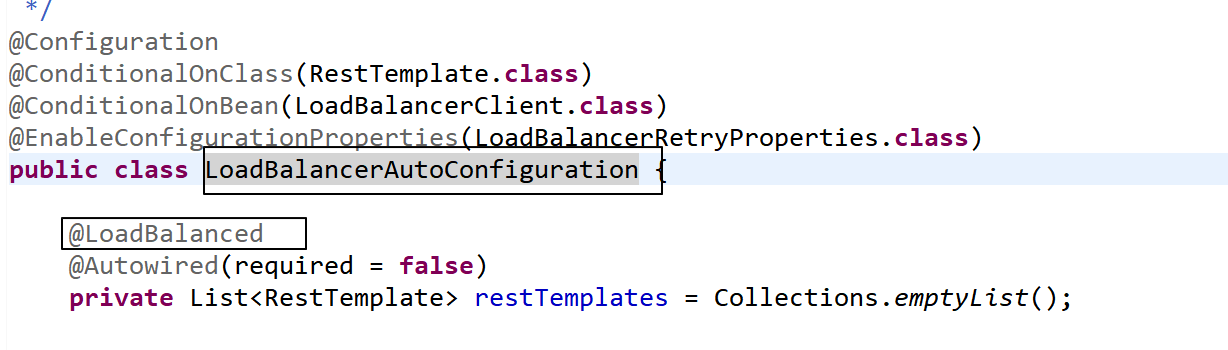
## 4.1 调用形式

|  |
| --- |
| String url = “http://provider/info”  String result = rest.getForObject(url, String.class); |

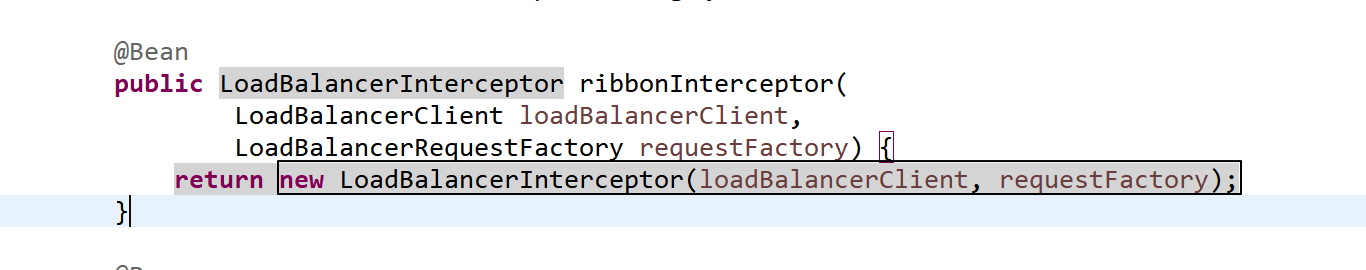
## 4.2 源码分析ribbon 怎么做把把该调用形式完成的

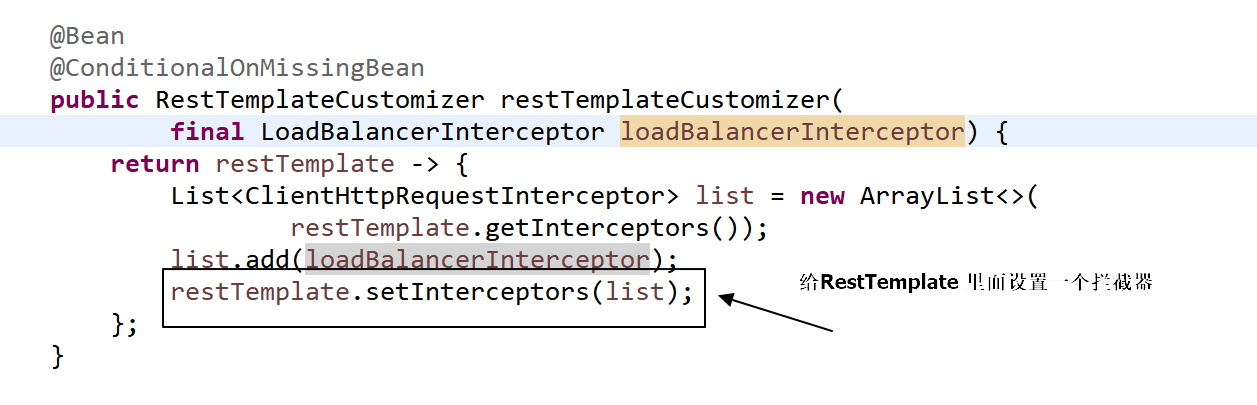
### 4.2.1 拦截器

在执行rest.getForObject(url, String.class) 先拦截下来

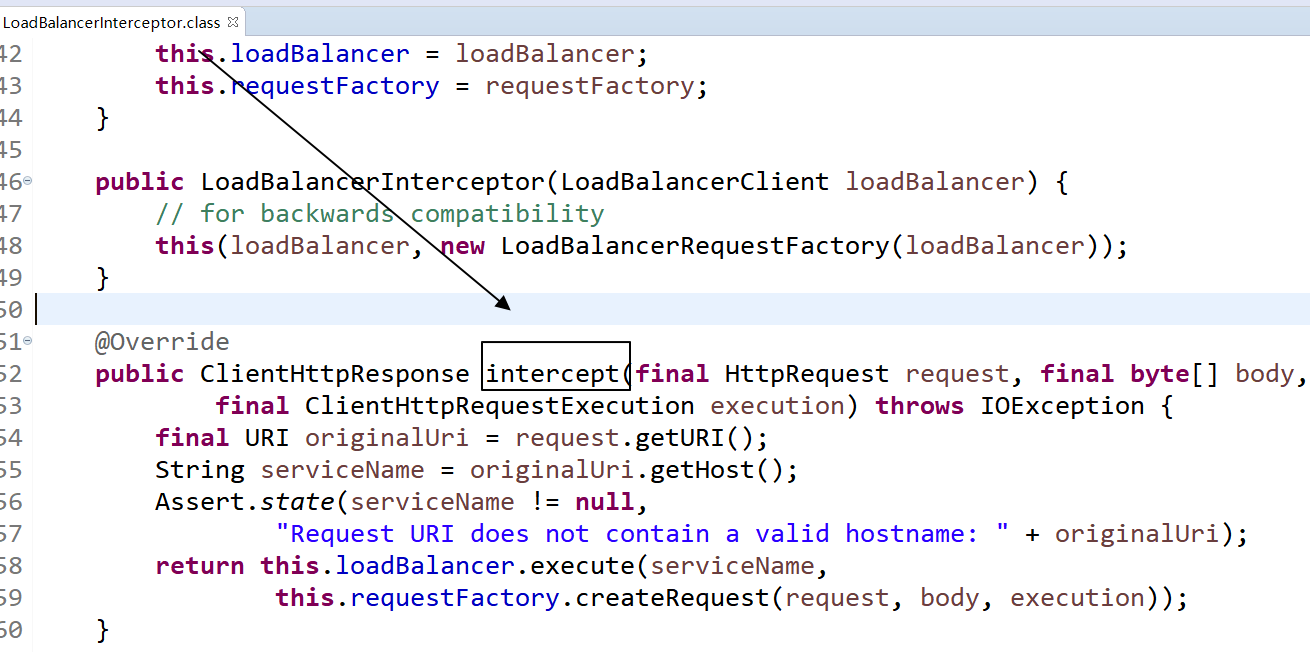


拦截器：





拦截做了啥：



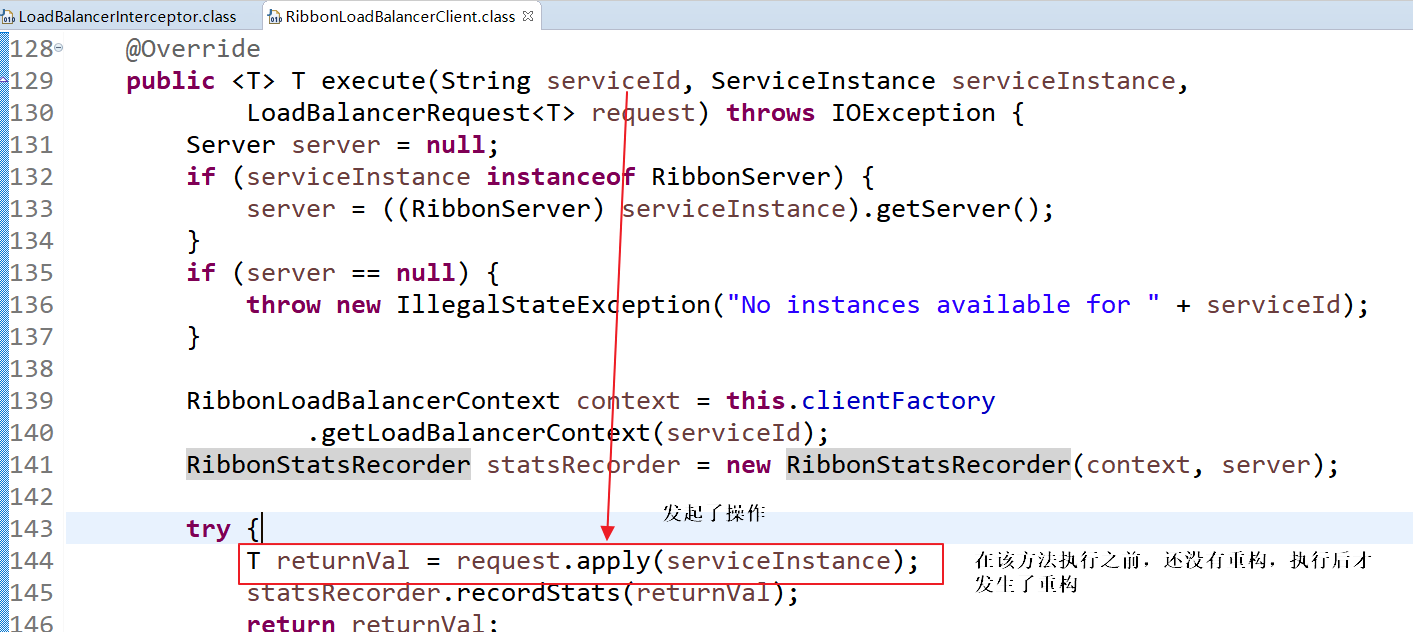
源码注释：

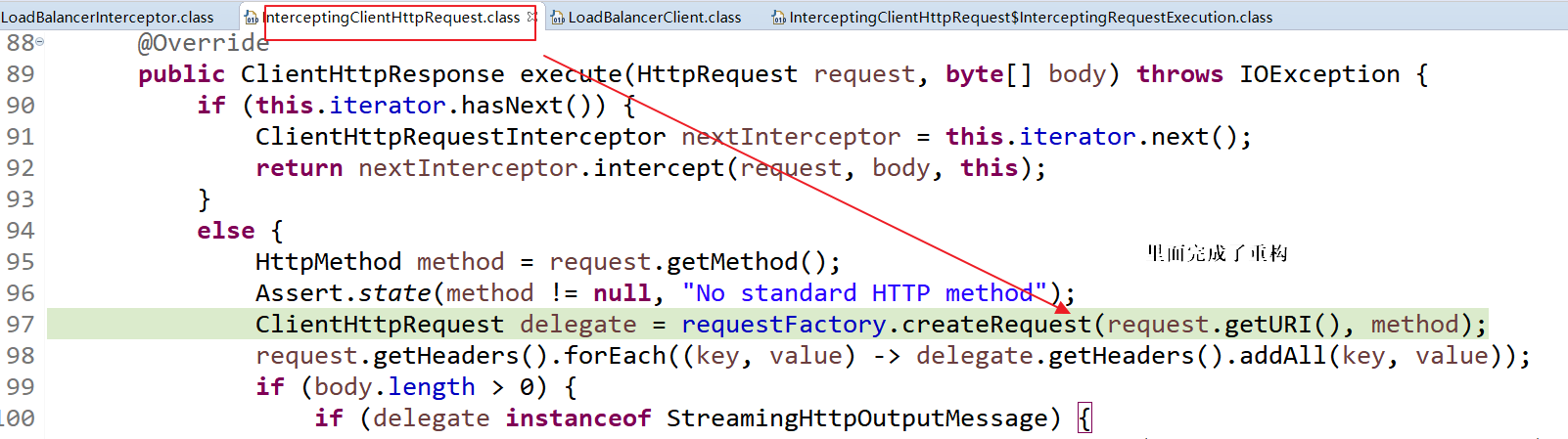
|  |
| --- |
| // 在Resttempalte 每次执行请求方法时，该拦截都会进行拦截  // http://provider/info  **public** ClientHttpResponse intercept(**final** HttpRequest request, **final** **byte**[] body,  **final** ClientHttpRequestExecution execution) **throws** IOException {  // 获取本次访问的url地址 http://provider/info  **final** URI originalUri = request.getURI();  // 获取本次url地址里面的主机名称  // 而主机名称就是服务的名称：provider  String serviceName = originalUri.getHost();  Assert.*state*(serviceName != **null**,  "Request URI does not contain a valid hostname: " + originalUri);  **return** **this**.loadBalancer.execute(serviceName,  **this**.requestFactory.createRequest(request, body, execution));  } |

### 4.2.2 将url地址重构

[http://provider/info->http://ip:port/info](http://provider/info-%3ehttp:/ip:port/info)

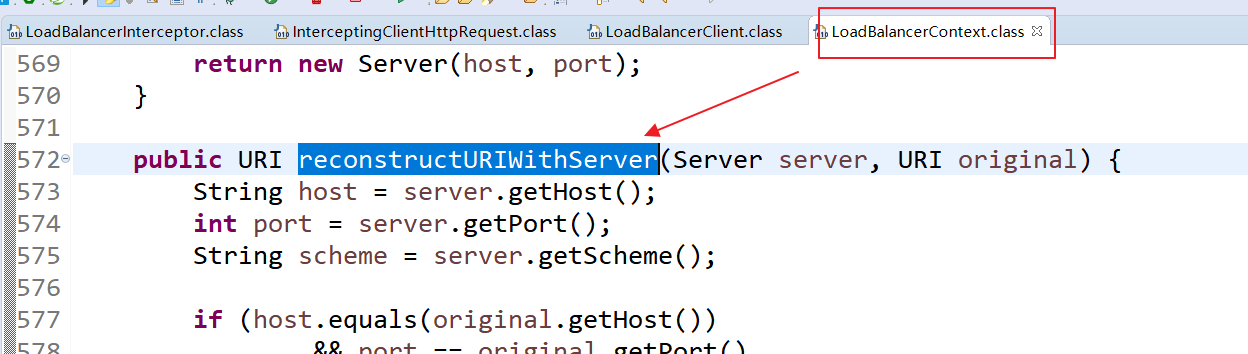






执行到了ServiceRequestWrapper 对象里面getURI的方法：

|  |
| --- |
| @Override  **public** URI getURI() {  URI uri = **this**.loadBalancer.reconstructURI(**this**.instance, getRequest().getURI());  **return** uri;  } |



在LoadBalancerContext的reconstructURIWithServer该方法

完成了拼接url的过程：

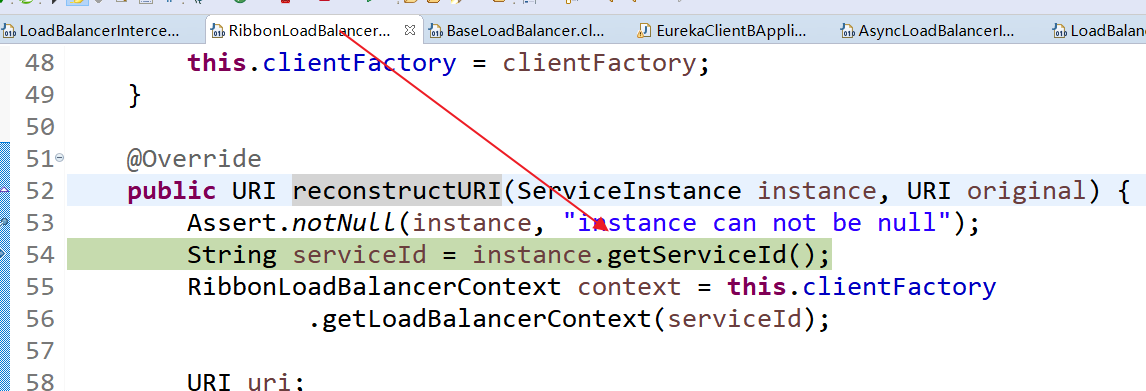
|  |
| --- |
| **public** URI reconstructURIWithServer(Server server, URI original) {  String host = server.getHost();  **int** port = server.getPort();  String scheme = server.getScheme();    **if** (host.equals(original.getHost())  && port == original.getPort()  && scheme == original.getScheme()) {  **return** original;  }  **if** (scheme == **null**) {  scheme = original.getScheme();  }  **if** (scheme == **null**) {  scheme = deriveSchemeAndPortFromPartialUri(original).first();  }  **try** {  StringBuilder sb = **new** StringBuilder();  // 拼接 //->http://  sb.append(scheme).append("://");  **if** (!Strings.*isNullOrEmpty*(original.getRawUserInfo())) {  sb.append(original.getRawUserInfo()).append("@");  }  http://192.168.231.125  sb.append(host);  **if** (port >= 0) {  //http://192.168.231.125:8080  sb.append(":").append(port);  }  // http://192.168.231.125:8080/info  sb.append(original.getRawPath());  **if** (!Strings.*isNullOrEmpty*(original.getRawQuery())) {  sb.append("?").append(original.getRawQuery());  }  **if** (!Strings.*isNullOrEmpty*(original.getRawFragment())) {  sb.append("#").append(original.getRawFragment());  }  URI newURI = **new** URI(sb.toString());  **return** newURI;  } **catch** (URISyntaxException e) {  **throw** **new** RuntimeException(e);  }  } |

而重构定义在LoadBalancerClient 接口里面：

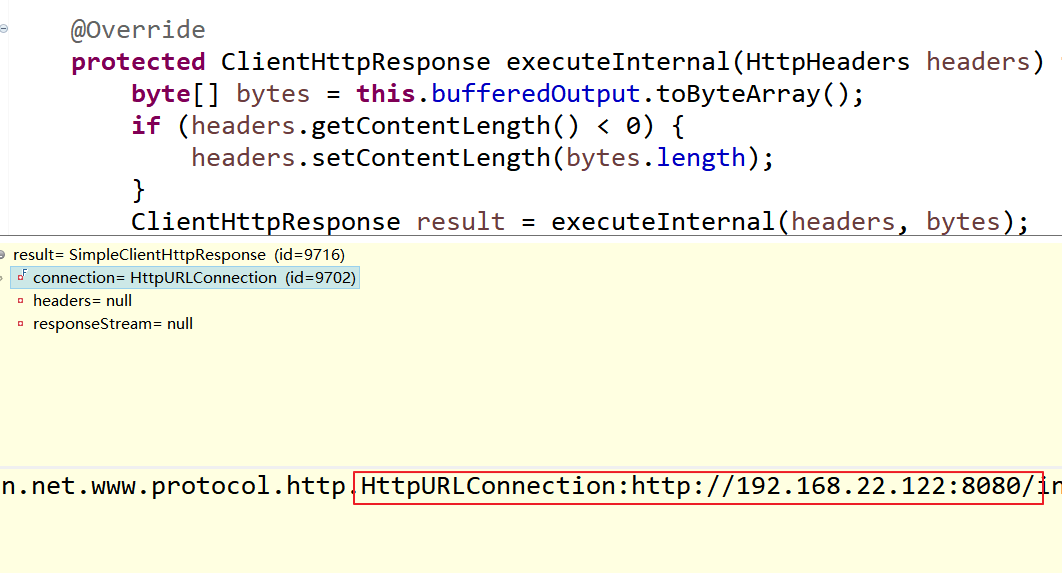
|  |
| --- |
| **public** **interface** LoadBalancerClient **extends** ServiceInstanceChooser |

有个方法：

|  |
| --- |
| URI reconstructURI(ServiceInstance instance, URI original); |



最后使用



发起了Http的请求：

RestTemplate 底层使用HttpUrlConnection 来发起了请求

# Ribbon的源码总结

## 5.1 Ribbon 里面有一个核心的类

ILoadBalancer

|  |
| --- |
| **public** Server chooseServer(Object key);  **public** List<Server> getReachableServers();  /\*\*  \* **@return** All known servers, both reachable and unreachable.  \*/  **public** List<Server> getAllServers(); |

## 5.2 ILoadBalancer 为什么重要

起了承上启下的作用

承上：Eureka-> 获取到了服务地址

启下：IRule-> 利用负载均衡算法选择注解

它的设计思想是这样的：

每个服务提供者

User-service：-> 自己的ILoadBalancer

Teacher-server：->自己的ILoadBalancer

Map<String,ILoadBalancer > iLoadBalancers

## 5.3 怎么实现了负载均衡：

ILoadBalancer balance = iLoadBalancers.get(“user-service”);

List<Server> server = balance.getReachableServers(); // list-》缓存值->eureka

Server server = balance.chooseServer(key:区域信息);-》IRule-》ZoneAvoidanceRule

## 5.4 默认的配置类有哪些：

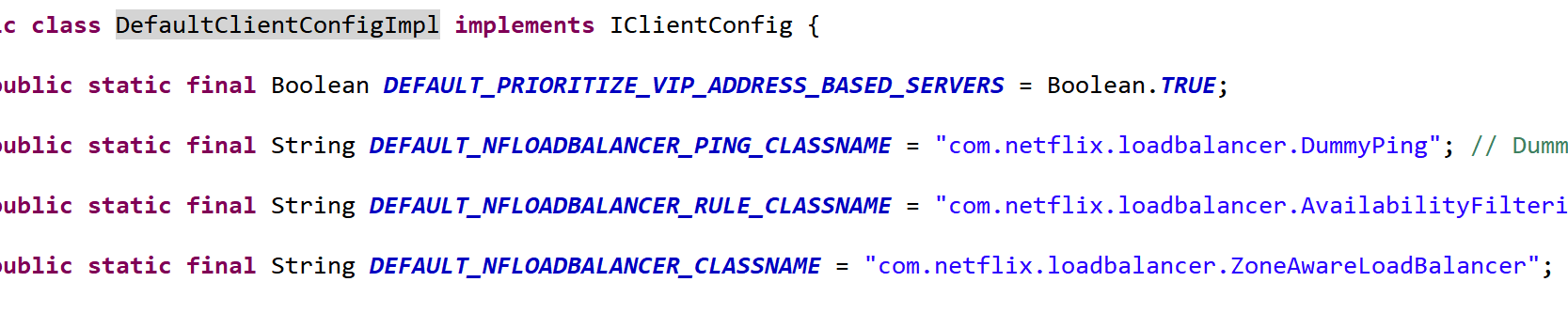
IPING 有好多个

IRULE 也有好多个

ServerList 也有好多个

。。

到底Ribbon默认使用的是哪些实现类：



IPing:

DummyPing

IRULE :

ZoneAvoidanceRule

ServerList:

DiscoveryEnabledNIWSServerList

ServerListFilter:

ZonePreferenceServerListFilter:优先过滤非一个区的服务列表

ILoadBalancer：

ZoneAwareLoadBalancer

## 5.5 怎么修改Ribbon的配置文件

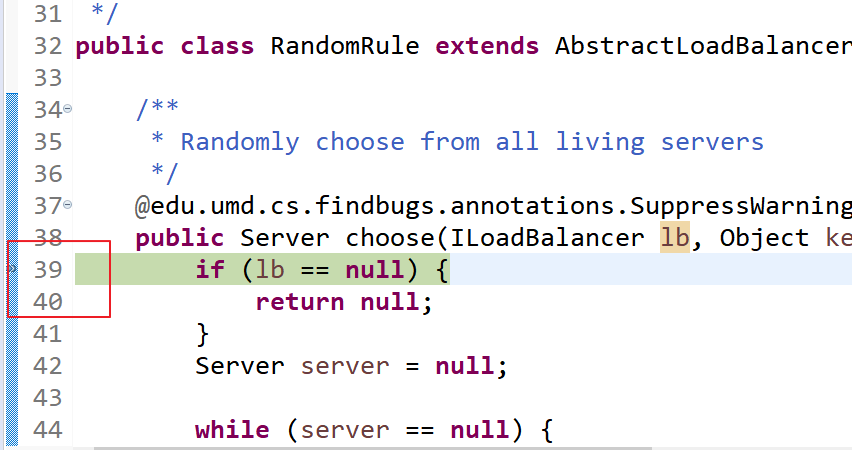
全局的配置：

|  |
| --- |
| 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) |

针对每个服务提供者的：

|  |
| --- |
| provider:# 服务的名称  ribbon:  # 采用那个负载均衡的规则  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule  # NFLoadBalancerClassName:  # NFLoadBalancerPingClassName:  # NFLoadBalancerRuleClassName:  # NIWSServerListClassName:  # NIWSServerListFilterClassName: |

配置完provider的IRule规则为随机算法：发现已经进来了



总结：

综上所述，Ribbon的负载均衡，主要通过LoadBalancerClient来实现的，而LoadBalancerClient具体交给了ILoadBalancer来处理，ILoadBalancer的实现类DynamicServerListLoadBalancer通过配置IRule、IPing等信息，并向EurekaClient获取注册列表的信息，并默认10秒一次向EurekaClient发送“ping”,进而检查是否更新服务列表，最后，得到注册列表后，ILoadBalancer根据IRule的策略进行负载均衡。

而RestTemplate 被@LoadBalance注解后，能过用负载均衡，主要是维护了一个被@LoadBalance注解的RestTemplate列表，并给列表中的RestTemplate添加拦截器，进而交给负载均衡器去处理。

# 为什么没有导入Ribbon 就能用？

Eureka 需要依赖Ribbon ，所以我们的项目只要有eureka，就有ribbon

总到来说，Feign的源码实现的过程如下：

首先通过@EnableFeignCleints注解开启FeignCleint

根据Feign的规则实现接口，并加@FeignCleint注解

程序启动后，会进行包扫描，扫描所有的@ FeignCleint的注解的类，并将这些信息注入到ioc容器中。

当接口的方法被调用，通过jdk的代理，来生成具体的RequesTemplate

RequesTemplate在生成Request

Request交给Client去处理，其中Client可以是HttpUrlConnection、HttpClient也可以是Okhttp

最后Client被封装到LoadBalanceClient类，这个类结合类Ribbon做到了负载均衡。

————————————————

版权声明：本文为CSDN博主「方志朋」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/forezp/article/details/73480304