

【Spring Cloud Ribbon】

1. 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。轮询 hash 权重 ...

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！

<https://docs.spring.io/spring-framework/docs/current/reference/html/integration.html#spring-integration>

```
/**
 * 测试发送 get 请求
 */
@Test
void testGet() {
    RestTemplate restTemplate = new RestTemplate();
    String url = "http://localhost:8080/testGet?name=cxs";
    ResponseEntity<String> result = restTemplate.getForEntity(url, String.class);
    System.out.println(result.getStatusCodeValue());
}

/**
 * 测试发送 post 表单参数
 */
@Test
void testPost() {
```

```
RestTemplate restTemplate = new RestTemplate();
String url = "http://localhost:8080/testPost";
LinkedMultiValueMap<String, String> map = new LinkedMultiValueMap<>();
map.add("name", "cxs");
map.add("age", "18");
ResponseEntity<String> result = restTemplate.postForEntity(url, map, String.class);
System.out.println(result.getStatusCodeValue());
}

/**
 * 测试发送post JSON 参数
 */
@Test
void testPost2() {
    RestTemplate restTemplate = new RestTemplate();
    String url = "http://localhost:8080/testPost2";
    User user = new User();
    user.setName("cxs");
    user.setAge(18);
    user.setHobby("编码");
    ResponseEntity<String> result = restTemplate.postForEntity(url, user, String.class);
    System.out.println(result.getStatusCodeValue());
}
```

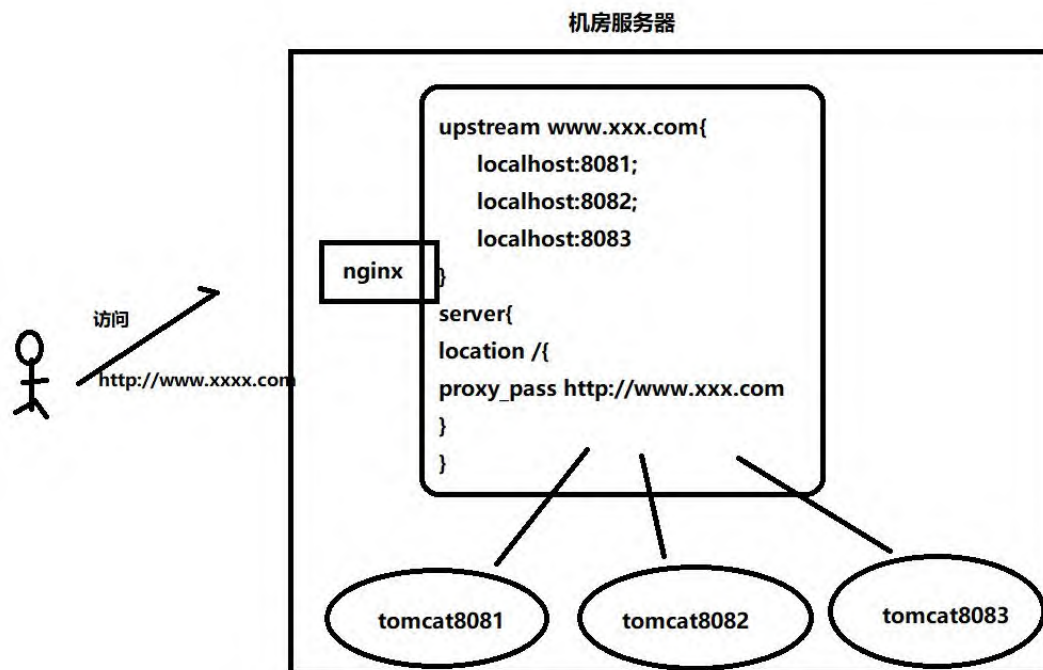
2. 负载均衡

负载均衡，英文名称为 Load Balance (**LB**) [http:// lb://](http://lb://)（负载均衡协议），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。

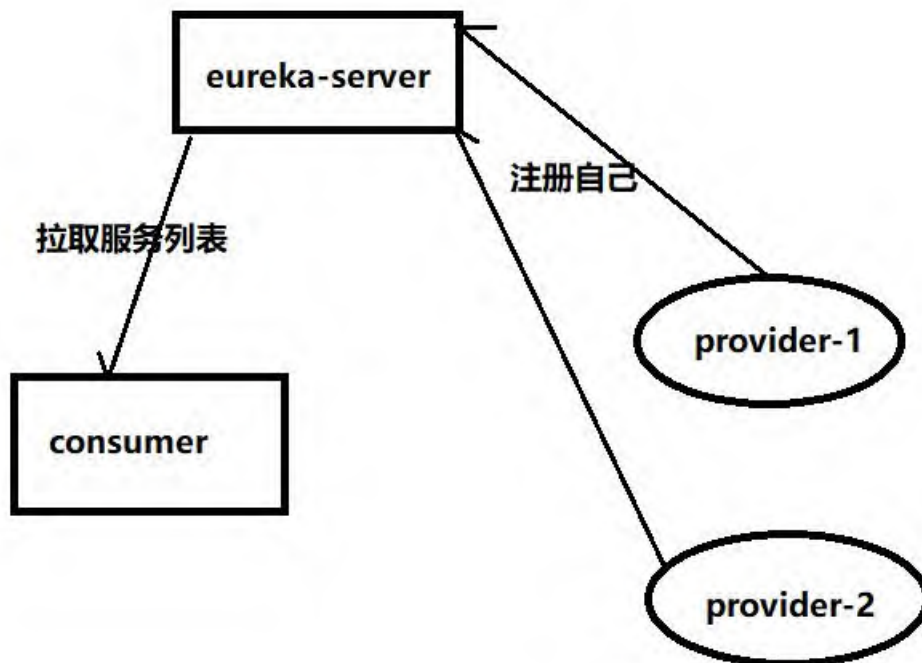
2.1 服务器的负载均衡

Nginx, F5



3. Ribbon 快速入门

3.1 本次调用设计图



3.2 项目搭建

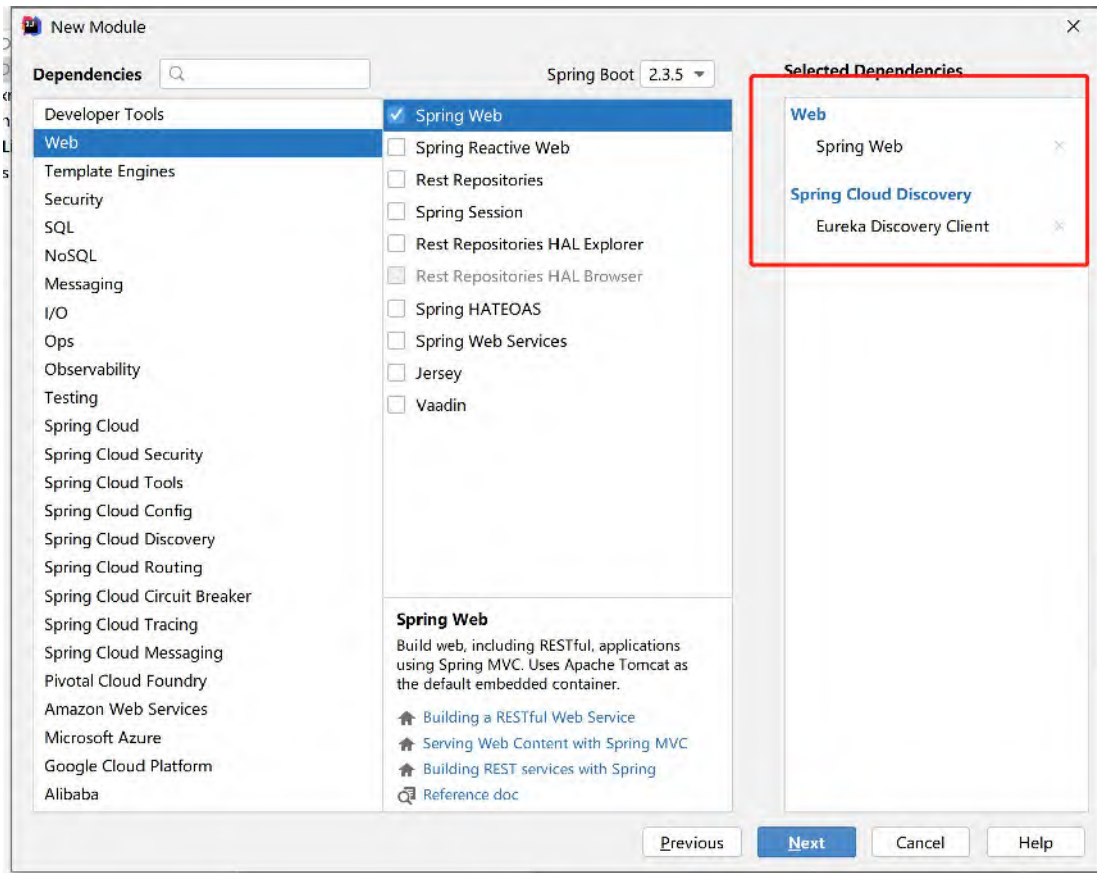
consumer 和 provider-1 和 provider-2 都是 eureka-client

注意这三个依赖是 eureka-client

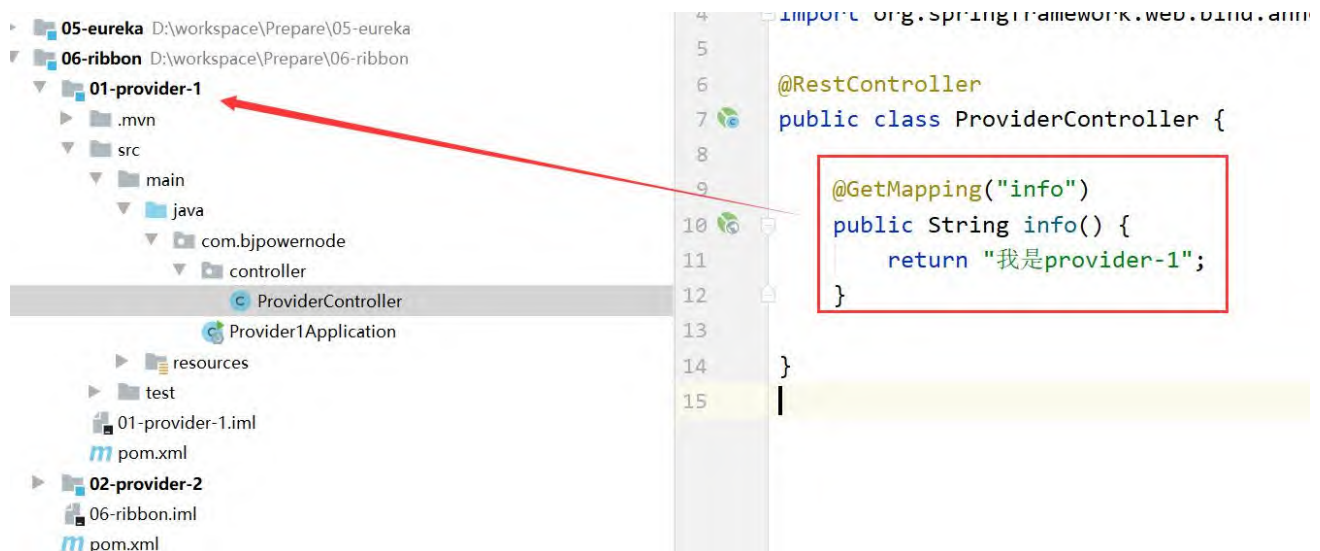
注意 provider-1 和 provider-2 的 spring.application.name=provider

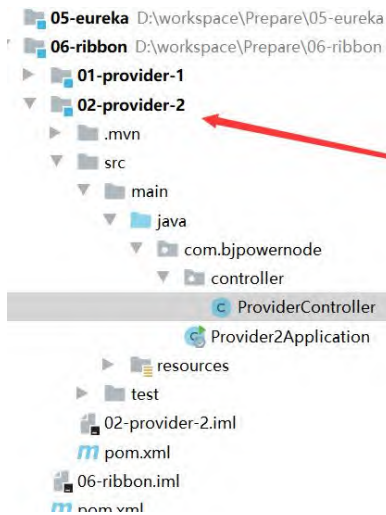
注意启动类的注解和配置文件的端口以及服务名称

3.3 创建 provider-1 和 provider-2



3.4 编写 provider-1 和 provider-2



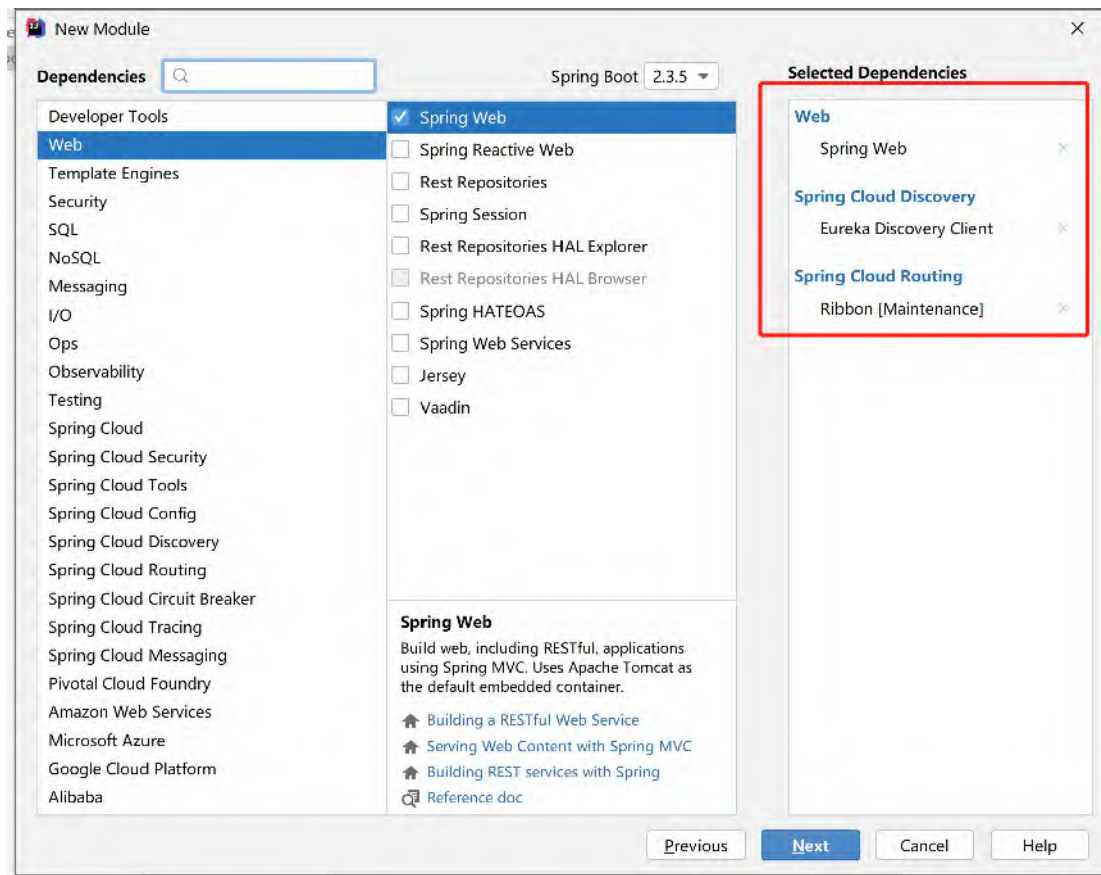


```

6 @RestController
7 public class ProviderController {
8
9
10 @GetMapping("info")
11 public String info() {
12     return "我是provider-2";
13 }
14 }
15

```

3.5 创建 consumer



```

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

```

```
<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
<version>2.2.9.RELEASE</version>
</dependency>
```

3.6 编写 consumer 的启动类

```
package com.bjpowernode;

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();
    }
}
```

3.7 编写 consumer 的 TestController

```
package com.bjpowernode.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: 北京动力节点
 */
@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;
    }
}
```

3.8 启动测试

首选确保都注册上去了

Renews (last min) 12

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER	n/a (1)	(1)	UP (1) - localhost:consumer:8001
EUREKA-SERVER	n/a (1)	(1)	UP (1) - localhost:eureka-server:8761
PROVIDER	n/a (2)	(2)	UP (2) - localhost:provider:8001, localhost:provider:8002

然后访问调用

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

本次我调用的是http://192.168.188.1:8001/info

我是provider-1

本次我调用的是http://192.168.188.1:8001/info

我是provider-1

本次我调用的是http://192.168.188.1:8002/info

我是provider-2

本次我调用的是http://192.168.188.1:8001/info

我是provider-1

本次我调用的是http://192.168.188.1:8002/info

我是provider-2

本次我调用的是http://192.168.188.1:8001/info

我是provider-1

本次我调用的是http://192.168.188.1:8001/info

我是provider-1

本次我调用的是http://192.168.188.1:8002/info

我是provider-2

3.9 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;
}
```

3.10 改造后测试效果

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

我是provider-1
我是provider-2
我是provider-1
我是provider-2
我是provider-1
我是provider-2
我是provider-1
我是provider-2
我是provider-1

4. Ribbon 源码分析

4.1 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, 我们需要以下步骤

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, 重构原来 URL 中的 provider
7. 真正的发送 `restTemplate.getForObject("http://ip:port/info", String.class)`

4.2 Ribbon 负载均衡的测试

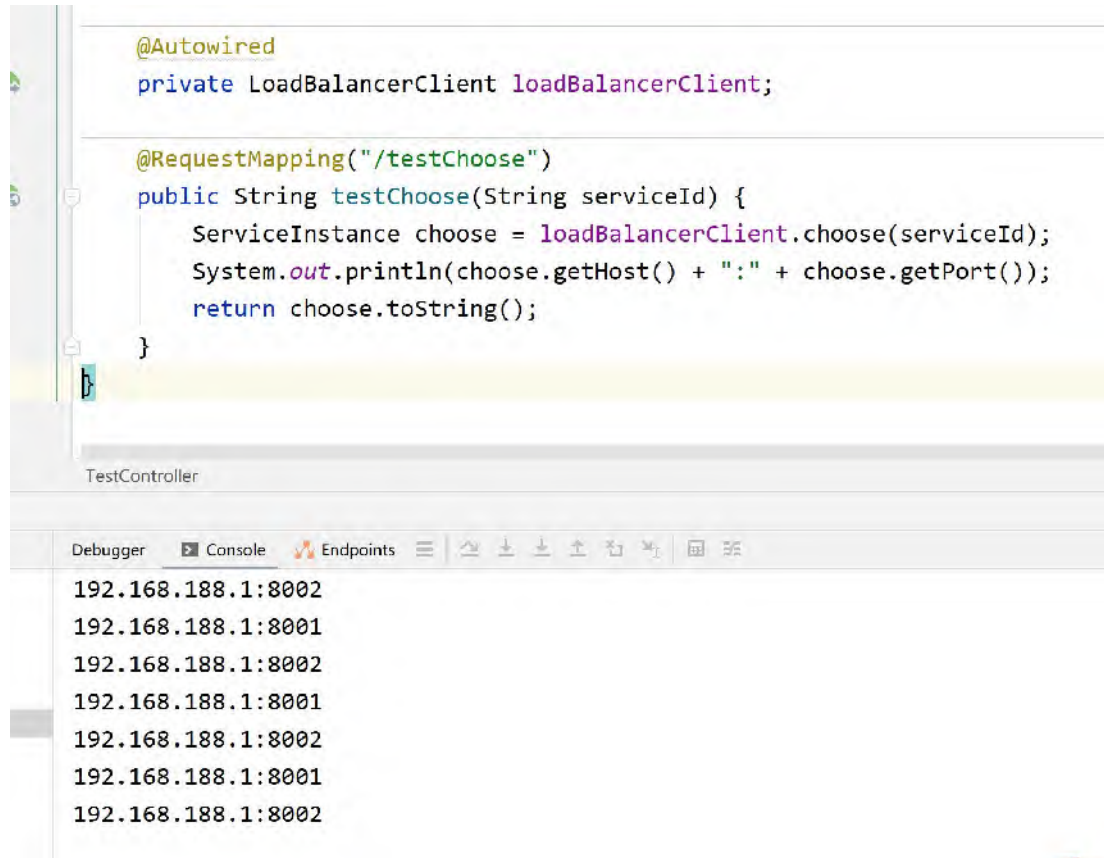
新增 controller

```
@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>



4.3 从 choose 方法入手，查看 Ribbon 负载均衡的源码



走进 `getServer()` 方法


```

184 protected Server getServer(ILoadBalancer loadBalancer, Object hint) {
185     if (loadBalancer == null) {
186         return null;
187     }
188     // Use 'default' on a null hint, or just pass it on?
189     return loadBalancer.chooseServer(hint != null ? hint : "default");
190 }
191

```

在 chooseServer() 里面得到 rule 是哪个对象

```

743
744 * @return the dedicated server
745 */
746 public Server chooseServer(Object key) { key: "default"
747     if (counter == null) {
748         counter = createCounter();
749     }
750     counter.increment(); counter: "BasicCounter{config=MonitorConfig{name=LoadBalancer_ChooseServer, tag
751     if (rule == null) { 在这里有一个choose方法，看看这个rule是哪个实现类
752         return null;
753     } else {
754         try {
755             return rule.choose(key); rule: ZoneAvoidanceRule@85514 key: "default"
756         } catch (Exception e) {
757             logger.error(e);
758         }
759     }
760 }

```

BaseLoadBalancer > chooseServer()

- rule: ZoneAvoidanceRule@85514
- compositePredicate = CompositePredicate@85553
- roundRobinRule = RoundRobinRule@8564
- lb = ZoneAwareLoadBalancer@8507 DynamicsServerListLoadBalancer(NFLoadBalancer{name=provider,current list of Servers=[1... View

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

```

35 */
36 public class ZoneAvoidanceRule extends PredicateBasedRule {
37

```

最终进入 PredicateBasedRule 类的 choose() 方法

```

37
38 /**
39 * Get a server by calling {@link AbstractServerPredicate#chooseRandomlyAfterFiltering(java.util.List,
40 * The performance for this method is O(n) where n is number of servers to be filtered.
41 */
42 @Override
43 public Server choose(Object key) { key: "default"
44     ILoadBalancer lb = getLoadBalancer(); lb: "DynamicServerListLoadBalancer(NFLoadBalancer{name=provi
45     Optional<Server> server = getPredicate().chooseRoundRobinAfterFiltering(lb.getAllServers(), key);
46     if (server.isPresent()) {
47         return server.get();
48     } else {
49         return null;
50 }

```

默认是轮训算法

```

196  * Choose a server in a round robin fashion after the predicate filters a given list of servers and load
197  */
198  public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers, Object loadBalancerKey) {
199      List<Server> eligible = getEligibleServers(servers, loadBalancerKey); eligible: size = 2 servers:
200      if (eligible.size() == 0) { eligible: size = 2
201          return Optional.absent();
202      }
203      return Optional.of(eligible.get(incrementAndGetModulo(eligible.size())));
204  }
205

```

根据服务id拿到对应的服务列表 (list集合)

这里从list里面get了一个

com.netflix.loadbalancer.AbstractServerPredicate#incrementAndGetModulo

```

147  * return the next value.
148  */
149  private int incrementAndGetModulo(int modulo) {
150      for (;;) {
151          int current = nextIndex.get();
152          int next = (current + 1) % modulo;
153          if (nextIndex.compareAndSet(current, next) && current < modulo)
154              return current;
155      }
156  }

```

modulo就是机器的数量, 我们现在是2台

请求次数%机器数量

1%2...1
2%2...0
3%2...1
...

使用自旋加cas, 得到一个值, 返回出去

再从list.get(i)就实现了轮询算法

4.4 负载均衡之前的服务列表是从何而来呢?

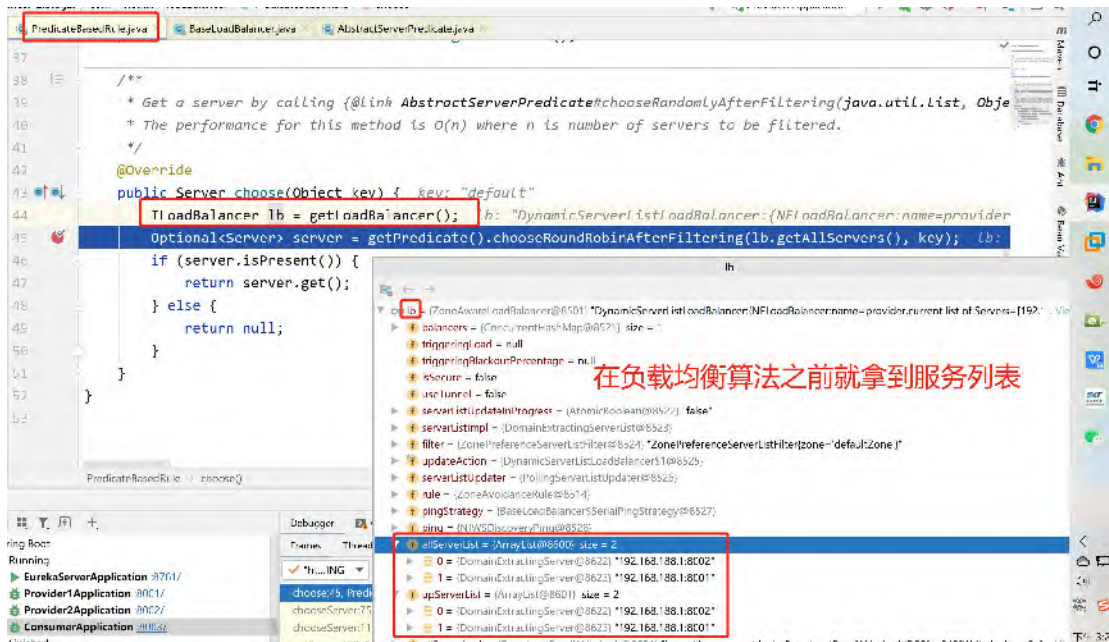
Ribbon 里面有没有服务列表?

Ribbon 只做负载均衡和远程调用

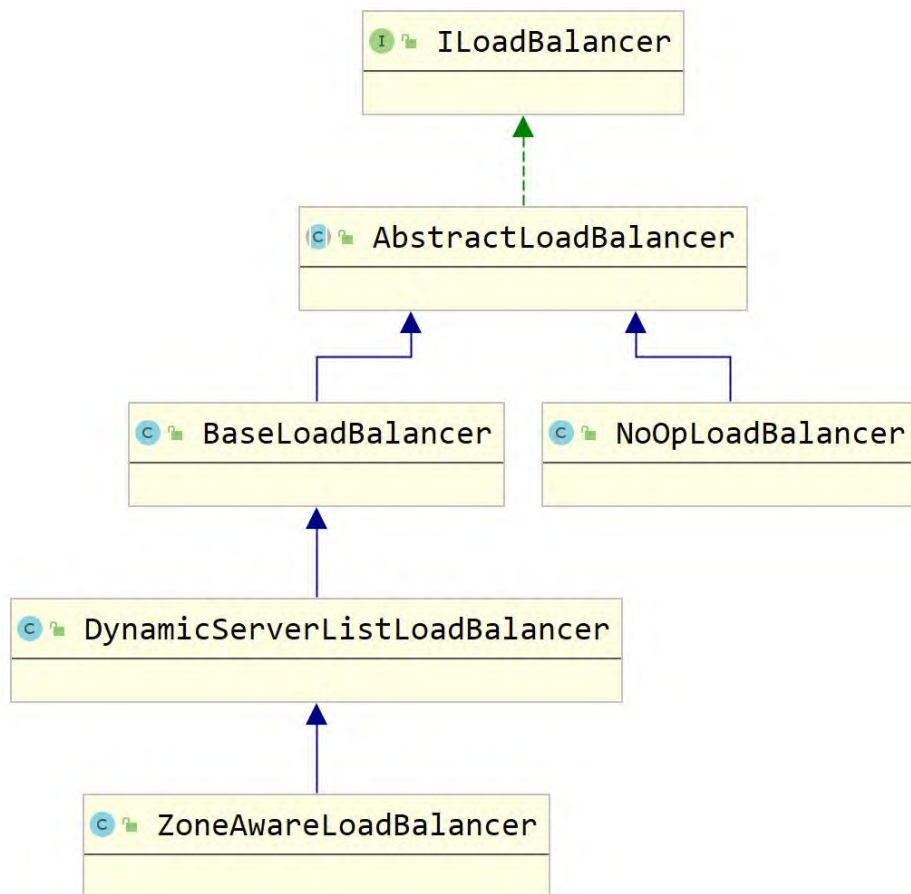
服务列表从哪来? 从 eureka 来

Ribbon 有一个核心接口 ILoadBalance (承上(eureka)启下 (Rule))

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

```

29  *
30  */
31  public interface ILoadBalancer {
32
33      /** Initial list of servers. ...*/
34      public void addServers(List<Server> newServers); 往ILoadBalancer中添加服务列表
35
36      /** Choose a server from load balancer. ...*/
37      public Server chooseServer(Object key); 选择一个服务
38
39      /** To be called by the clients of the load balancer to notify that a Server is down ...*/
40      public void markServerDown(Server server); 标记服务下线
41
42      /** @deprecated 2016-01-20 This method is deprecated in favor of the ...*/
43      @Deprecated
44      public List<Server> getServerList(boolean availableOnly);
45
46      /** @return Only the servers that are up and reachable. */
47      public List<Server> getReachableServers(); 和eureka结合获得可用的服务列表
48
49      /** @return All known servers, both reachable and unreachable. */
50      public List<Server> getAllServers(); 得到所有的服务列表
51  }

```

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

ribbon的选择过程由ILoadBalance接口定义的方法完成



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

```
BaseLoadBalancer.java
58  *
59  */
60  public class BaseLoadBalancer extends AbstractLoadBalancer implements
61      PrimeConnections.PrimeConnectionListener, IClientConfigAware {
62
63      private static Logger logger = LoggerFactory
64          .getLogger(BaseLoadBalancer.class);
65      private final static IRule DEFAULT_RULE = new RoundRobinRule();
66      private final static SerialPingStrategy DEFAULT_PING_STRATEGY = new SerialPingStrategy();
67      private static final String DEFAULT_NAME = "default";
68      private static final String PREFIX = "LoadBalancer_";
69
70      protected IRule rule = DEFAULT_RULE;
71
72      protected IPingStrategy pingStrategy = DEFAULT_PING_STRATEGY;
73
74      protected IPing ping = null;
75
76      @Monitor(name = PREFIX + "AllServerList", type = DataSourceType.INFORMATIONAL)
77      protected volatile List<Server> allServerList = Collections
78          .synchronizedList(new ArrayList<Server>());
79      @Monitor(name = PREFIX + "UpServerList", type = DataSourceType.INFORMATIONAL)
80      protected volatile List<Server> upServerList = Collections
81          .synchronizedList(new ArrayList<Server>());
```

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

```
45  */
46  public class DynamicServerListLoadBalancer<T> extends Server<T> extends BaseLoadBalancer {
47      private static final Logger LOGGER = LoggerFactory.getLogger(DynamicServerListLoadBalancer.class)
48
49      boolean isSecure = false;
50      boolean useTunnel = false;
51
52      // to keep track of modification of server lists
53      protected AtomicBoolean serverListUpdateInProgress = new AtomicBoolean( initialValue: false);
54
55      volatile ServerList<T> serverListImpl;
56
57      volatile ServerListFilter<T> filter;
58  }
```

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


```

154
155 private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
156     List<DiscoveryEnabledServer> serverList = new ArrayList<>();
157
158     if (eurekaClientProvider == null || eurekaClientProvider.get() == null) {
159         logger.warn("EurekaClient has not been initialized yet, returning an empty list");
160         return new ArrayList<DiscoveryEnabledServer>();
161     }
162
163     EurekaClient eurekaClient = eurekaClientProvider.get(); 从eureka中拿到服务列表
164     if (vipAddresses != null) {
165         循环 for (String vipAddress : vipAddresses.split(regex, ",")) {
166             // if targetRegion is null, it will be interpreted as the same region of client
167             List<InstanceInfo> listOfInstanceInfo = eurekaClient.getInstanceByVipAddress(vipAddress, isSecure, ta
168             for (InstanceInfo ii : listOfInstanceInfo) {
169                 if (ii.getStatus().equals(InstanceStatus.UP)) { 判断服务状态
170
171                     if (shouldUseOverridePort) {...}
172
173                     DiscoveryEnabledServer des = createServer(ii, isSecure, shouldUseIpAddr);
174                     serverList.add(des); 放到一开始new的集合中
175                 }
176             }
177             if (serverList.size() > 0 && prioritizeVipAddressBasedServers) {...}
178         }
179     }
180     return serverList; 将服务列表集合返回出去
181 }

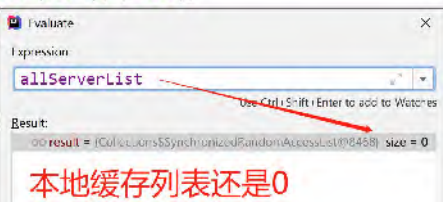
```

再回到 BaseLoadBalancer 中真正的存放服务列表

```

484
485 /**
486  * Set the list of servers used as the server pool. This overrides existing
487  * server list. 拉取到了两个服务实例
488  */
489 public void setServersList(List lsrv) { lsrv.size() = 2
490     Lock writeLock = allServerLock.writeLock(); allServerLock: "java.util.concurrent.locks.ReentrantR
491     logger.debug("LoadBalancer {}: clearing server list (SET op)", name);
492
493     ArrayList<Server> newServers = new ArrayList<>();
494     writeLock.lock();
495     try {
496         ArrayList<Server> allServers = new ArrayList<>();
497         for (Object server : lsrv) {
498             if (server == null) {
499                 continue;

```



```

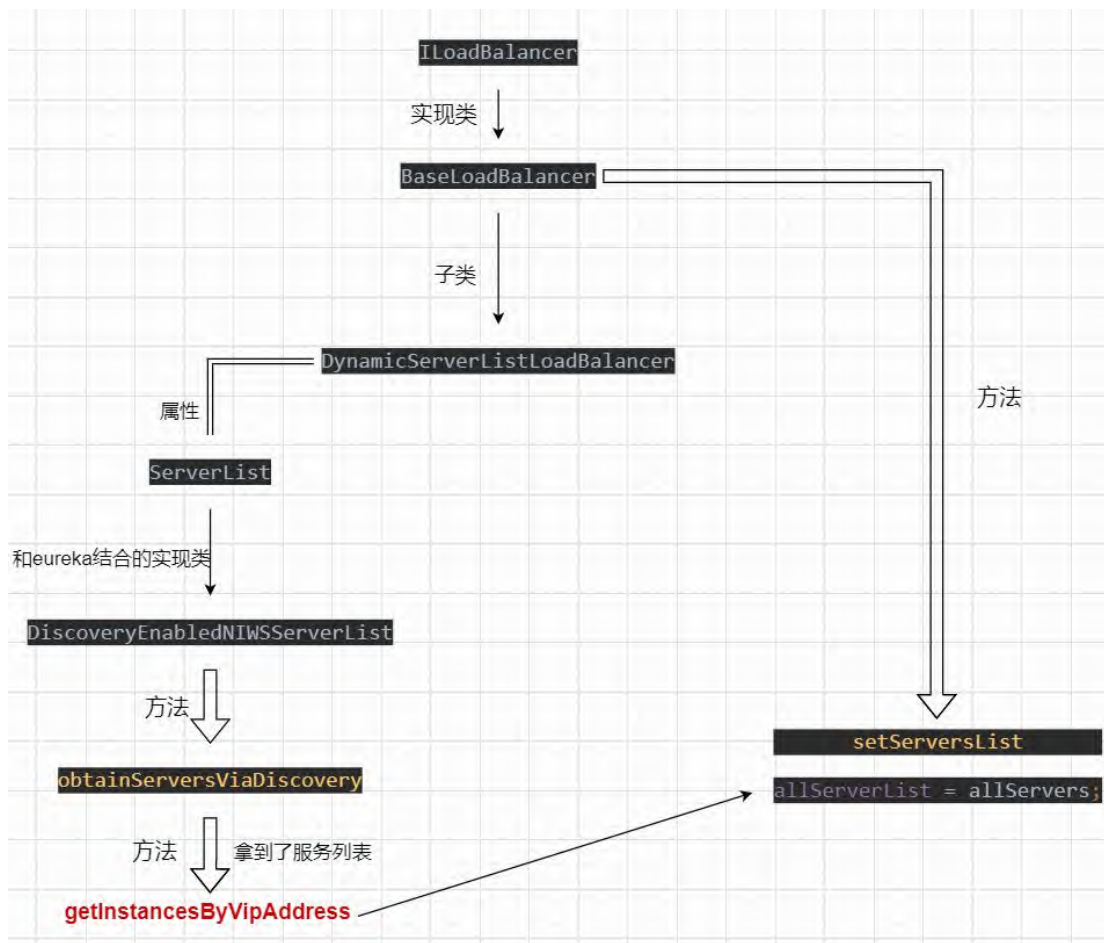
541     }
542     // This will reset readyToServe flag to true on all servers
543     // regardless whether
544     // previous priming connections are success or not
545     allServerList = allServers; allServerList: size = 0 allServers: size = 2
546     if (canSkipPing()) {
547         for (Server s : allServerList) {
548             s.setAlive(true);
549         }
550         upServerList = allServerList;
551     } else if (listChanged = true ) {
552         forceQuickPing();
553     }
554     } finally {
555         writeLock.unlock();
556     }
557 }

```

在这里放到了ribbon自己的本地缓存列表

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

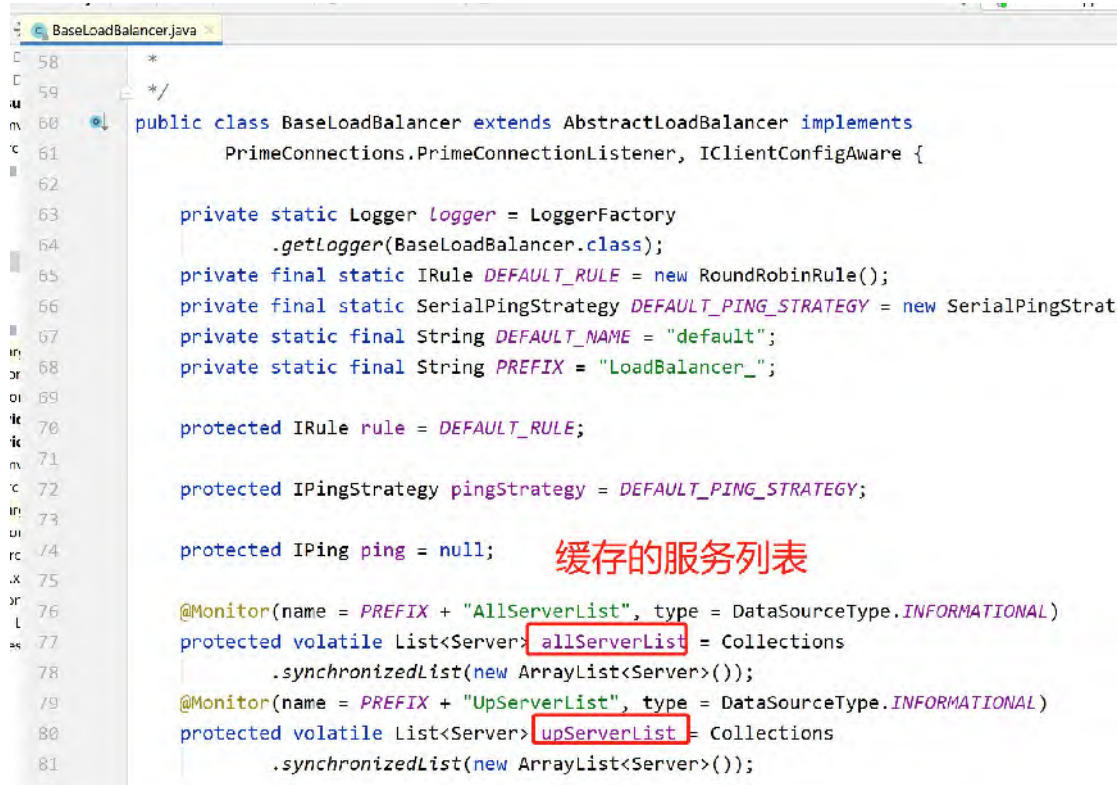
也就是说并不是服务一启动就拉取了服务列表缓存起来，流程图如下：



4.5 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



```
58 *
59 */
60 public class BaseLoadBalancer extends AbstractLoadBalancer implements
61     PrimeConnections.PrimeConnectionListener, IClientConfigAware {
62
63     private static Logger logger = LoggerFactory
64         .getLogger(BaseLoadBalancer.class);
65     private final static IRule DEFAULT_RULE = new RoundRobinRule();
66     private final static SerialPingStrategy DEFAULT_PING_STRATEGY = new SerialPingStrat
67     private static final String DEFAULT_NAME = "default";
68     private static final String PREFIX = "LoadBalancer_";
69
70     protected IRule rule = DEFAULT_RULE;
71
72     protected IPingStrategy pingStrategy = DEFAULT_PING_STRATEGY;
73
74     protected IPing ping = null;
75
76     @Monitor(name = PREFIX + "AllServerList", type = DataSourceType.INFORMATIONAL)
77     protected volatile List<Server> allServerList = Collections
78         .synchronizedList(new ArrayList<Server>());
79
80     @Monitor(name = PREFIX + "UpServerList", type = DataSourceType.INFORMATIONAL)
81     protected volatile List<Server> upServerList = Collections
82         .synchronizedList(new ArrayList<Server>());
```

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

```
BaseLoadBalancer.java
144 public BaseLoadBalancer(String name, IRule rule, LoadBalancerStats stats,
145     IPing ping, IPingStrategy pingStrategy) {
146
147     logger.debug("LoadBalancer [{}]: initialized", name);
148
149     this.name = name;
150     this.ping = ping;
151     this.pingStrategy = pingStrategy;
152     setRule(rule);
153     setupPingTask();
154     lbStats = stats;
155     init();
156 }
```

在这个构造器里

有负载均衡算法设置和 ping 机制，防止缓存脏读

我们查看 IPing 这个接口

```
BaseLoadBalancer.java  IPing.java
1  package com.netflix.loadbalancer;
2
3  /**
4   * Interface that defines how we "ping" a server to check if its ali
5   * @author stonse
6   */
7
8  public interface IPing {
9
10     /**
11      * Checks whether the given <code>Server</code> is "alive" i.e.
12      * considered a candidate while loadbalancing
13      */
14     public boolean isAlive(Server server);
15 }
```

默认使用这个 ping

查看节点是否正常

我们就想看 NIWSDiscoveryPing

```
BaseLoadBalancer.java  IPing.java  NIWSDiscoveryPing.java
56 public boolean isAlive(Server server) { server: "192.168.188.1:8001"
57     boolean isAlive = true;
58     if (server != null && server instanceof DiscoveryEnabledServer) {
59         DiscoveryEnabledServer dServer = (DiscoveryEnabledServer) server;
60         InstanceInfo instanceInfo = dServer.getInstanceInfo();
61         if (instanceInfo != null) {
62             InstanceStatus status = instanceInfo.getStatus();
63             if (status != null) {
64                 isAlive = status.equals(InstanceStatus.UP);
65             }
66         }
67     }
68     return isAlive;
69 }
```

判断阶段状态

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


```

RibbonClientConfiguration.java
141 @Bean
142 @ConditionalOnMissingBean
143 public ServerListUpdater ribbonServerListUpdater(IClientConfig config) {
144     return new PollingServerListUpdater(config);
145 }
146

PollingServerListUpdater.java
98 public PollingServerListUpdater(final long initialDelayMs, final long refreshIntervalMs) {
99     this.initialDelayMs = initialDelayMs;
100     this.refreshIntervalMs = refreshIntervalMs;
101 }
102
103 @Override
104 public synchronized void start(final UpdateAction updateAction) {
105     if (isActive.compareAndSet(expect: false, update: true)) {
106         final Runnable wrapperRunnable = () -> {
107             if (!isActive.get()) {
108                 if (scheduledFuture != null) {
109                     scheduledFuture.cancel(true);
110                 }
111                 return;
112             }
113             try {
114                 updateAction.doUpdate();
115                 lastUpdated = System.currentTimeMillis();
116             } catch (Exception e) {
117                 Logger.warn("Failed one update cycle", e);
118             }
119         };
120         scheduledFuture = getRefreshExecutor().scheduleWithFixedDelay(
121             wrapperRunnable,
122             initialDelayMs, refreshIntervalMs, TimeUnit.MILLISECONDS);
123     }
124 }

```

在配置类中这个地方更新了服务列表

有一个定时任务

做服务列表的更新

定时任务在哪里开始执行的呢？我们查找 doUpdate() 方法

```

DynamicServerListLoadBalancer.java
58
59 protected final ServerListUpdater.UpdateAction updateAction = new ServerListUpdater.UpdateAction() {
60     @Override
61     public void doUpdate() {
62         updateListOfServers();
63     }
64 };
65
66 protected volatile ServerListUpdater serverListUpdater;

```

这里做了服务列表的更新

解决脏读机制的总结：

1. Ping
2. 更新机制

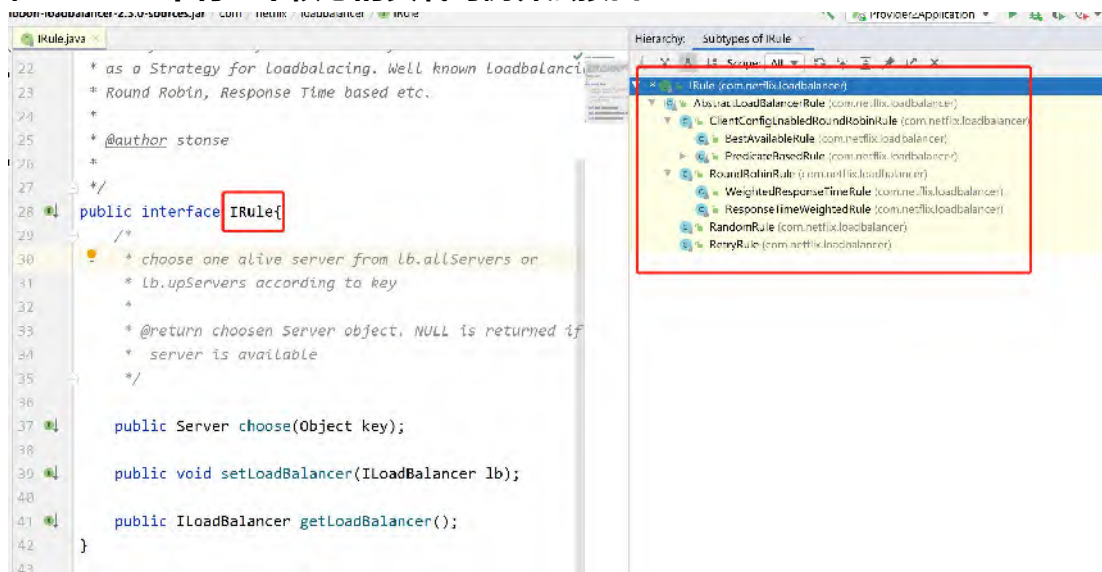
都是为了解决脏读的现象而生的

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

4.6 Ribbon 负载均衡的实现和几种算法【重点】

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.权重

4. iphash

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

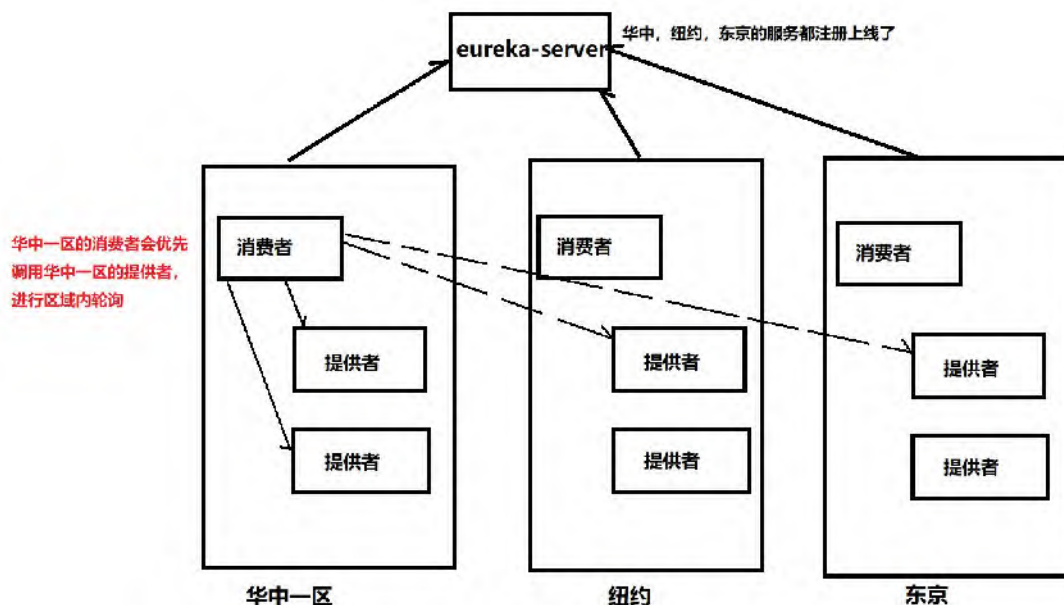
6.BestAvailableRule --会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量小的服务

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 ： 区间内亲和轮询的算法！通过一个 key 来区分

负载均衡算法：随机 **轮训** 权重 iphash （响应时间最短算法，区域内亲和（轮训）算法）



5. 如何修改默认的负载均衡算法

5.1 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
```


5.2 测试调用该服务（这里使用随机规则）

```
我是provider-1  
我是provider-2  
我是provider-1  
我是provider-1  
我是provider-2  
我是provider-2  
我是provider-2  
我是provider-1
```

5.3 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();  
}
```

6. Ribbon 的配置文件和常用配置

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

```

82  /*
83  public class DefaultClientConfigImpl implements IClientConfig {
84
85      public static final Boolean DEFAULT_PRIORITIZE_VIP_ADDRESS_BASED_SERVERS = Boolean.TRUE;
86
87      public static final String DEFAULT_NFLOADBALANCER_PING_CLASSNAME = "com.netflix.loadbalancer.DummyPing"; // DummyPing
88
89      public static final String DEFAULT_NFLOADBALANCER_RULE_CLASSNAME = "com.netflix.loadbalancer.AvailabilityFilteringRule";
90
91      public static final String DEFAULT_NFLOADBALANCER_CLASSNAME = "com.netflix.loadbalancer.ZoneAwareLoadBalancer";
92
93      public static final boolean DEFAULT_USEIPADDRESS_FOR_SERVER = Boolean.FALSE;
94
95      public static final String DEFAULT_CLIENT_CLASSNAME = "com.netflix.niws.client.http.RestClient";
96
97      public static final String DEFAULT_VIPADDRESS_RESOLVER_CLASSNAME = "com.netflix.client.SimpleVipAddressResolver";
98
99      public static final String DEFAULT_PRIME_CONNECTIONS_URI = "/";
100
101      public static final int DEFAULT_MAX_TOTAL_TIME_TO_PRIME_CONNECTIONS = 30000;
102
103      public static final int DEFAULT_MAX_RETRIES_PER_SERVER_PRIME_CONNECTION = 9;
104
105      public static final Boolean DEFAULT_ENABLE_PRIME_CONNECTIONS = Boolean.FALSE;
106
107      public static final int DEFAULT_MAX_REQUESTS_ALLOWED_PER_WINDOW = Integer.MAX_VALUE;

```

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URLConnection 来完成的, 这里面设置为true, 可以把URLConnection->HttpClient

okhttp:

enabled: false #URLConnection 来完成的, 这里面设置为true, 可以把URLConnection->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

7.Ribbon 总结（后面的代码中 不会出现 ribbon）

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`

`Map<String, ILoadBalancer> iLoadBalancers` 消费者端

服务提供者的名称 value （服务列表 算法规则）

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

你就可以使用 eureka+ribbon 做分布式项目