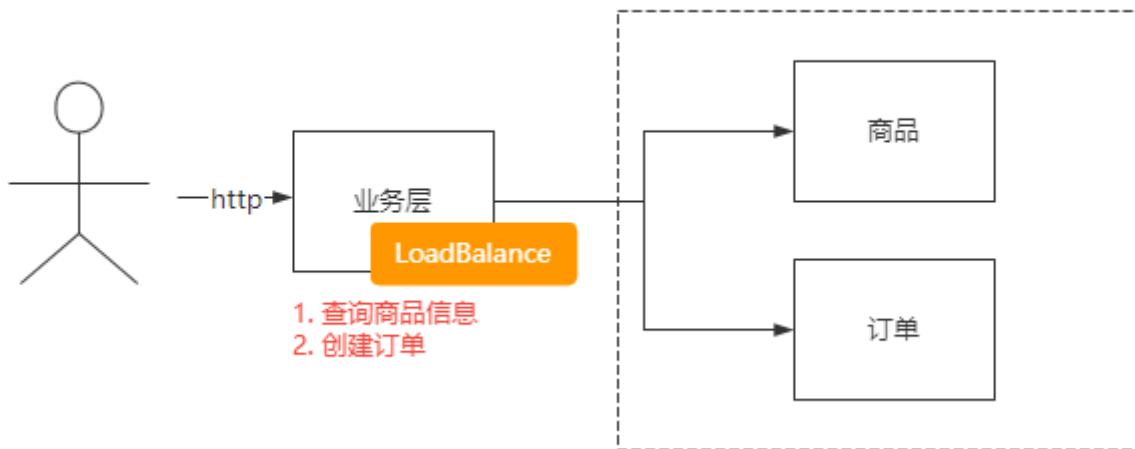


Ribbon的核心作用就是进行请求的负载均衡，它的基本原理如下图所示。就是客户端集成Ribbon这个组件，Ribbon中会针对已经配置的服务提供者地址列表进行负载均衡的计算，得到一个目标地址之后，再发起请求。



那么接下来，我们从两个层面去分析Ribbon的原理

1. @LoadBalanced 注解如何让普通的 RestTemplate 具备负载均衡的能力
2. OpenFeign集成Ribbon的实现原理

@LoadBalancer注解解析过程分析

在使用RestTemplate的时候，我们加了一个@LoadBalance注解，就能让这个RestTemplate 在请求时，就拥有客户端负载均衡的能力。

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

然后，我们打开@LoadBalanced这个注解，可以发现该注解仅仅是声明了一个@Qualifier注解。

```
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@Qualifier  
public @interface LoadBalanced {  
}
```

@Qualifier注解的作用

我们平时在使用注解去注入一个Bean时，都是采用@Autowired。并且大家应该知道 @Autowired是可以注入一个List或者Map的。给大家举个例子（在一个springboot应用中）

| 定义一个TestClass

```
@AllArgsConstructor  
@Data  
public class TestClass {  
    private String name;  
}
```

| 声明一个配置类，并注入TestClass

```
@Configuration  
public class TestConfig {  
  
    @Bean("testClass1")  
    TestClass testClass(){  
        return new TestClass("testClass1");  
    }  
  
    @Bean("testClass2")  
    TestClass testClass2(){  
        return new TestClass("testClass2");  
    }  
}
```

| 定义一个Controller，用于测试，注意，此时我们使用的是@Autowired来注入一个List集合

```
@RestController  
public class TestController {  
  
    @Autowired(required = false)  
    List<TestClass> testClasses= Collections.emptyList();  
  
    @GetMapping("/test")  
    public Object test(){  
        return testClasses;  
    }  
}
```

| 此时访问：<http://localhost:8080/test>，得到的结果是

```
[  
    {  
        name: "testClass1"  
    },  
    {  
        name: "testClass2"  
    }  
]
```

| 修改TestConfig和TestController

```
@Configuration  
public class TestConfig {  
  
    @Bean("testClass1")  
    @Qualifier
```

```

    TestClass testClass(){
        return new TestClass("testClass1");
    }

    @Bean("testClass2")
    TestClass testClass2(){
        return new TestClass("testClass2");
    }
}

@RestController
public class TestController {

    @Autowired(required = false)
    @Qualifier
    List<TestClass> testClasses= Collections.emptyList();

    @GetMapping("/test")
    public Object test(){
        return testClasses;
    }
}

```

再次访问: <http://localhost:8080/test> , 得到的结果是

```
[
  {
    name: "testClass1"
  }
]
```

@LoadBalancer注解筛选及拦截

了解了@qualifier注解的作用后，再回到@LoadBalancer注解上，就不难理解了。

因为我们需要扫描到增加了@LoadBalancer注解的RestTemplate实例，所以，@LoadBalancer可以完成这个动作，它的具体的实现代码如下：

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
@EnableConfigurationProperties(LoadBalancerProperties.class)
public class LoadBalancerAutoConfiguration {

    @LoadBalanced
    @Autowired(required = false)
    private List<RestTemplate> restTemplates = Collections.emptyList();
}

```

从这个代码中可以看出，在LoadBalancerAutoConfiguration这个配置类中，会使用同样的方式，把配置了@LoadBalanced注解的RestTemplate注入到restTemplates集合中。

拿到了RestTemplate之后，在LoadBalancerInterceptorConfig配置类中，会针对这些RestTemplate进行拦截，实现代码如下：

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)

```

```

@EnableConfigurationProperties(LoadBalancerProperties.class)
public class LoadBalancerAutoConfiguration {

    @LoadBalanced
    @Autowired(required = false)
    private List<RestTemplate> restTemplates = Collections.emptyList();

    //省略....

    @Bean
    @ConditionalOnMissingBean
    public LoadBalancerRequestFactory loadBalancerRequestFactory(LoadBalancerClient
loadBalancerClient) {
        return new LoadBalancerRequestFactory(loadBalancerClient, this.transformers);
    }

    @Configuration(proxyBeanMethods = false)
    @Conditional(RetryMissingOrDisabledCondition.class)
    static class LoadBalancerInterceptorConfig {

        //装载一个LoadBalancerInterceptor的实例到IOC容器。
        @Bean
        public LoadBalancerInterceptor loadBalancerInterceptor(LoadBalancerClient
loadBalancerClient,
                LoadBalancerRequestFactory requestFactory) {
            return new LoadBalancerInterceptor(loadBalancerClient, requestFactory);
        }

        //会遍历所有加了@LoadBalanced注解的RestTemplate，在原有的拦截器之上，再增加了一个
LoadBalancerInterceptor
        @Bean
        @ConditionalOnMissingBean
        public RestTemplateCustomizer restTemplateCustomizer(final
LoadBalancerInterceptor loadBalancerInterceptor) {
            return restTemplate -> {
                List<ClientHttpRequestInterceptor> list = new ArrayList<>
(restTemplate.getInterceptors());
                list.add(loadBalancerInterceptor);
                restTemplate.setInterceptors(list);
            };
        }

        }
        //省略....
    }
}

```

LoadBalancerInterceptor

```

@Override
public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
        final ClientHttpRequestExecution execution) throws IOException {
    final URI originalUri = request.getURI();
    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));
}

```

RestTemplate调用过程

我们在程序中，使用下面的代码发起远程请求时

```
restTemplate.getForObject(url, String.class);
```

它的整个调用过程如下。

```
RestTemplate.getForObject  
----> AbstractClientHttpRequest.execute()  
---->AbstractBufferingClientHttpRequest.executeInternal()  
----> InterceptingClientHttpRequest.executeInternal()  
----> InterceptingClientHttpRequest.execute()
```

InterceptingClientHttpRequest.execute()方法的代码如下。

```
@Override  
public ClientHttpResponse execute(HttpServletRequest request, byte[] body) throws IOException  
{  
    if (this.iterator.hasNext()) { //遍历所有的拦截器，通过拦截器进行逐个处理。  
        ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();  
        return nextInterceptor.intercept(request, body, this);  
    }  
    else {  
        HttpMethod method = request.getMethod();  
        Assert.state(method != null, "No standard HTTP method");  
        ClientHttpRequest delegate = requestFactory.createRequest(request.getURI(),  
method);  
        request.getHeaders().forEach((key, value) -> delegate.getHeaders().addAll(key,  
value));  
        if (body.length > 0) {  
            if (delegate instanceof StreamingHttpOutputMessage) {  
                StreamingHttpOutputMessage streamingOutputMessage =  
(StreamingHttpOutputMessage) delegate;  
                streamingOutputMessage.setBody(outputStream -> StreamUtils.copy(body,  
outputStream));  
            }  
            else {  
                StreamUtils.copy(body, delegate.getBody());  
            }  
        }  
        return delegate.execute();  
    }  
}
```

LoadBalancerInterceptor

LoadBalancerInterceptor是一个拦截器，当一个被@Loadbalanced注解修饰的RestTemplate对象发起HTTP请求时，会被LoadBalancerInterceptor的intercept方法拦截，

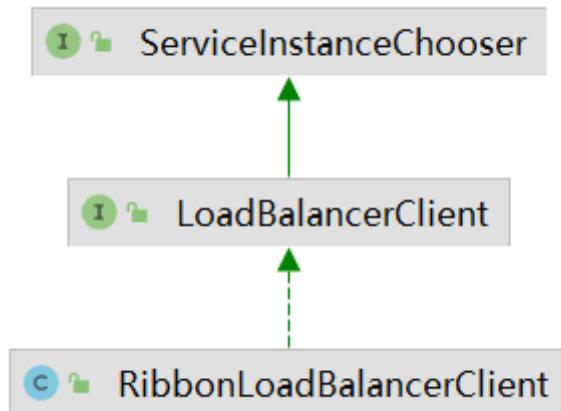
在这个方法中直接通过getHost方法就可以获取到服务名（因为我们在使用RestTemplate调用服务的时候，使用的是服务名而不是域名，所以这里可以通过getHost直接拿到服务名然后去调用execute方法发起请求）

```

@Override
public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
        final ClientHttpRequestExecution execution) throws IOException {
    final URI originalUri = request.getURI();
    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));
}

```

LoadBalancerClient其实是一个接口，我们看一下它的类图，它有一个唯一的实现类：[RibbonLoadBalancerClient](#)。



RibbonLoadBalancerClient.execute

RibbonLoadBalancerClient这个类的代码比较长，我们主要看一下他的核心方法execute

```

public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)
throws IOException {
ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
Server server = getServer(loadBalancer, hint);
if (server == null) {
    throw new IllegalStateException("No instances available for " + serviceId);
}
RibbonServer ribbonServer = new RibbonServer(serviceId, server,
isSecure(server, serviceId),
serverIntrospector(serviceId).getMetadata(server));

return execute(serviceId, ribbonServer, request);
}

```

上述代码的实现逻辑如下：

- 根据serviceId获得一个ILoadBalancer，实例为：ZoneAwareLoadBalancer
- 调用getServer方法去获取一个服务实例
- 判断Server的值是否为空。这里的Server实际上就是传统的一个服务节点，这个对象存储了服务节点的一些元数据，比如host、port等

getServer

getServer是用来获得一个具体的服务节点，它的实现如下

```

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

```

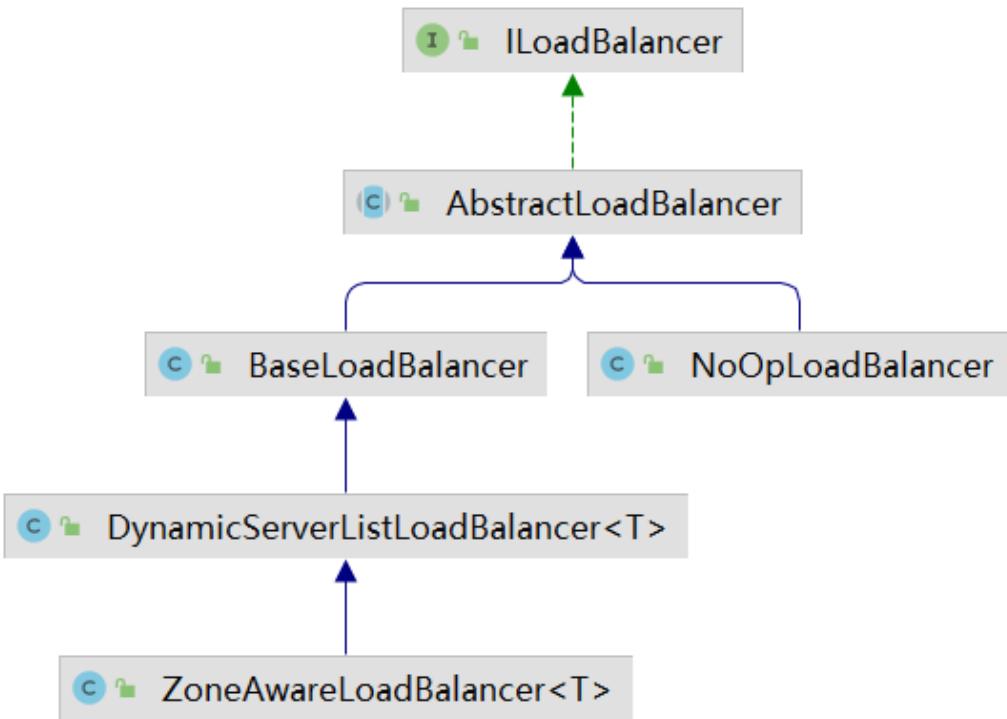
通过代码可以看到，`getServer`实际调用了`ILoadBalancer.chooseServer`这个方法，`ILoadBalancer`这个是一个负载均衡器接口。

```

public interface ILoadBalancer {
    //addServers表示向负载均衡器中维护的实例列表增加服务实例
    public void addServers(List<Server> newServers);
    //chooseServer表示通过某种策略，从负载均衡服务器中挑选出一个具体的服务实例
    public Server chooseServer(Object key);
    //markServerDown表示用来通知和标识负载均衡器中某个具体实例已经停止服务，否则负载均衡器在
    下一次获取服务实例清单前都会认为这个服务实例是正常工作的
    public void markServerDown(Server server);
    //getReachableServers表示获取当前正常工作的服务实例列表
    public List<Server> getReachableServers();
    //getAllServers表示获取所有的服务实例列表，包括正常的服务和停止工作的服务
    public List<Server> getAllServers();
}

```

`ILoadBalancer`的类关系图如下：



从整个类的关系图来看，`BaseLoadBalancer`类实现了基础的负载均衡，而`DynamicServerListLoadBalancer`和`ZoneAwareLoadBalancer`则是在负载均衡策略的基础上做了一些功能扩展。

- `AbstractLoadBalancer`实现了`ILoadBalancer`接口，它定义了服务分组的枚举类/`chooseServer`（用来选取一个服务实例）/`getServerList`（获取某一个分组中的所有服务实例）/`getLoadBalancerStats`用来获得一个`LoadBalancerStats`对象，这个对象保存了每一个服务的状态信息。

- `BaseLoadBalancer`, 它实现了作为负载均衡器的基本功能, 比如服务列表维护、服务存活状态监测、负载均衡算法选择Server等。但是它只是完成基本功能, 在有些复杂场景中还无法实现, 比如动态服务列表、Server过滤、Zone区域意识 (服务之间的调用希望尽可能是在同一个区域内进行, 减少延迟)。
- `DynamicServerListLoadBalancer`是`BaseLoadbalancer`的一个子类, 它对基础负载均衡提供了扩展, 从名字上可以看出, 它提供了动态服务列表的特性
- `ZoneAwareLoadBalancer` 它是在`DynamicServerListLoadBalancer`的基础上, 增加了以Zone的形式来配置多个LoadBalancer的功能。

那在`getServer`方法中, `loadBalancer.chooseServer`具体的实现类是哪一个呢? 我们找到`RibbonClientConfiguration`这个类

```

@Bean
@ConditionalOnMissingBean
public ILoadBalancer ribbonLoadBalancer(IClientConfig config,
                                         ServerList<Server> serverList,
                                         ServerListFilter<Server> serverListFilter,
                                         IRule rule, IPing ping, ServerListUpdater
                                         serverListUpdater) {
    if (this.propertiesFactory.isSet(ILoadBalancer.class, name)) {
        return this.propertiesFactory.get(ILoadBalancer.class, config, name);
    }
    return new ZoneAwareLoadBalancer<>(config, rule, ping, serverList,
                                         serverListFilter, serverListUpdater);
}

```

从上述声明中, 发现如果没有自定义`ILoadBalancer`, 则直接返回一个`ZoneAwareLoadBalancer`

ZoneAwareLoadBalancer

`Zone`表示区域的意思, 区域指的就是地理区域的概念, 一般较大规模的互联网公司, 都会做跨区域部署, 这样做有几个好处, 第一个是为不同地域的用户提供最近的访问节点减少访问延迟, 其次是为了保证高可用, 做容灾处理。

而`ZoneAwareLoadBalancer`就是提供了具备区域意识的负载均衡器, 它的主要作用是对`Zone`进行了感知, 保证每个`Zone`里面的负载均衡策略都是隔离的, 它并不保证A区域过来的请求一定会发动到A区域对应的Server内。真正实现这个需求的是`ZonePreferenceServerListFilter/ZoneAffinityServerListFilter`。

`ZoneAwareLoadBalancer`的核心功能是

- 若开启了区域意识, 且`zone`的个数 > 1 , 就继续区域选择逻辑
- 根据`ZoneAvoidanceRule.getAvailableZones()`方法拿到可用区们 (会去除完全不可用的区域们, 以及可用但是负载最高的一个区域)
- 从可用区`zone`们中, 通过`ZoneAvoidanceRule.randomChooseZone`随机选一个`zone`出来 (该随机遵从权重规则: 谁的`zone`里面`Server`数量最多, 被选中的概率越大)
- 在选中的`zone`里面的所有`Server`中, 采用该`zone`对对应的`Rule`, 进行`choose`

```

@Override
public Server chooseServer(Object key) {
    //ENABLED, 表示是否用区域意识的choose选择Server, 默认是true,
    //如果禁用了区域、或者只有一个zone, 就直接按照父类的逻辑来进行处理, 父类默认采用轮询算法
    if (!ENABLED.get() || getLoadBalancerStats().getAvailableZones().size() <= 1) {
        logger.debug("Zone aware logic disabled or there is only one zone");
        return super.chooseServer(key);
    }
}

```

```

Server server = null;
try {
    LoadBalancerStats lbStats = getLoadBalancerStats();
    Map<String, ZoneSnapshot> zoneSnapshot =
ZoneAvoidanceRule.createSnapshot(lbStats);
    logger.debug("Zone snapshots: {}", zoneSnapshot);
    if (triggeringLoad == null) {
        triggeringLoad = DynamicPropertyFactory.getInstance().getDoubleProperty(
            "ZoneAwareNIWSDiscoveryLoadBalancer." + this.getName() +
".triggeringLoadPerServerThreshold", 0.2d);
    }

    if (triggeringBlackoutPercentage == null) {
        triggeringBlackoutPercentage =
DynamicPropertyFactory.getInstance().getDoubleProperty(
            "ZoneAwareNIWSDiscoveryLoadBalancer." + this.getName() +
".avoidZoneWithBlackoutPercetage", 0.99999d);
    }
    //根据相关阈值计算可用区域
    Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones(zoneSnapshot,
triggeringLoad.get(), triggeringBlackoutPercentage.get());
    logger.debug("Available zones: {}", availableZones);
    if (availableZones != null && availableZones.size() <
zoneSnapshot.keySet().size()) {
        //从可用区域中随机选择一个区域, zone里面的服务器节点越多, 被选中的概率越大
        String zone = ZoneAvoidanceRule.randomChooseZone(zoneSnapshot,
availableZones);
        logger.debug("Zone chosen: {}", zone);
        if (zone != null) {
            //根据zone获得该zone中的LB, 然后根据该Zone的负载均衡算法选择一个server
            BaseLoadBalancer zoneLoadBalancer = getLoadBalancer(zone);
            server = zoneLoadBalancer.chooseServer(key);
        }
    }
} catch (Exception e) {
    logger.error("Error choosing server using zone aware logic for load balancer=
{}", name, e);
}
if (server != null) {
    return server;
} else {
    logger.debug("Zone avoidance logic is not invoked.");
    return super.chooseServer(key);
}
}
}

```

BaseLoadBalancer.chooseServer

假设我们现在没有使用多区域部署, 那么负载策略会执行到
`BaseLoadBalancer.chooseServer`,

```

public Server chooseServer(Object key) {
    if (counter == null) {
        counter = createCounter();
    }
    counter.increment();
    if (rule == null) {
        return null;
    } else {
        try {

```

```

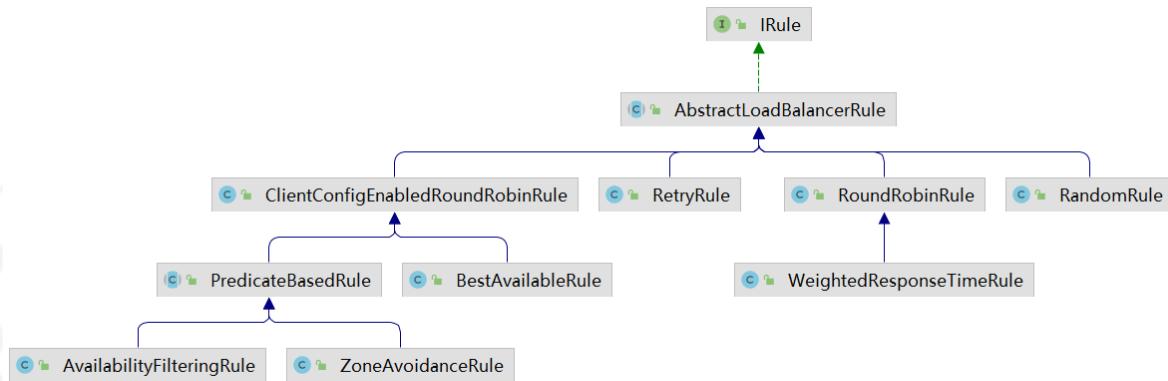
        return rule.choose(key);
    } catch (Exception e) {
        logger.warn("LoadBalancer [{}]: Error choosing server for key {}", name,
key, e);
        return null;
    }
}
}
}

```

根据默认的负载均衡算法来获得指定的服务节点。默认的算法是RoundRobin。

rule.choose

rule代表负载均衡算法规则，它有很多实现，IRule的实现类关系图如下。



默认情况下，rule的实现为ZoneAvoidanceRule，它是在RibbonClientConfiguration这个配置类中定义的，代码如下：

```

@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties
// Order is important here, last should be the default, first should be optional
// see
// https://github.com/spring-cloud/spring-cloud-netflix/issues/2086#issuecomment-
316281653
@Import({ HttpClientConfiguration.class, OkHttpRibbonConfiguration.class,
          RestClientRibbonConfiguration.class, HttpClientRibbonConfiguration.class })
public class RibbonClientConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public IRule ribbonRule(IClientConfig config) {
        if (this.propertiesFactory.isSet(IRule.class, name)) {
            return this.propertiesFactory.get(IRule.class, config, name);
        }
        ZoneAvoidanceRule rule = new ZoneAvoidanceRule();
        rule.initWithNwrsConfig(config);
        return rule;
    }
}

```

所以，在BaseLoadBalancer.chooseServer中调用rule.choose(key);，实际会进入到ZoneAvoidanceRule的choose方法

```

@Override
public Server choose(Object key) {
    ILoadBalancer lb = getLoadBalancer(); //获取负载均衡器
    Optional<Server> server =
getPredicate().chooseRoundRobinAfterFiltering(lb.getAllServers(), key); //通过该方法获取
目标服务
    if (server.isPresent()) {
        return server.get();
    } else {
        return null;
    }
}

```

复合判断server所在区域的性能和server的可用性选择server

主要分析 `chooseRoundRobinAfterFiltering` 方法。

chooseRoundRobinAfterFiltering

从方法名称可以看出来，它是通过对目标服务集群通过过滤算法过滤一遍后，再使用轮询实现负载均衡。

```

public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers, Object
loadBalancerKey) {
    List<Server> eligible = getEligibleServers(servers, loadBalancerKey);
    if (eligible.size() == 0) {
        return Optional.absent();
    }
    return Optional.of(eligible.get(incrementAndGetModulo(eligible.size())));
}

```

CompositePredicate.getEligibleServers

使用主过滤条件对所有实例过滤并返回过滤后的清单，

```

@Override
public List<Server> getEligibleServers(List<Server> servers, Object loadBalancerKey) {
    //
    List<Server> result = super.getEligibleServers(servers, loadBalancerKey);

    //按照fallbacks中存储的过滤器顺序进行过滤（此处就行先ZoneAvoidancePredicate然后
    AvailabilityPredicate）
    Iterator<AbstractServerPredicate> i = fallbacks.iterator();
    while (!(result.size() >= minimalFilteredServers && result.size() > (int)
(servers.size() * minimalFilteredPercentage))
        && i.hasNext()) {
        AbstractServerPredicate predicate = i.next();
        result = predicate.getEligibleServers(servers, loadBalancerKey);
    }
    return result;
}

```

依次使用次过滤条件对主过滤条件的结果进行过滤*

- //不论是主过滤条件还是次过滤条件，都需要判断下面两个条件
- //只要有一个条件符合，就不再过滤，将当前结果返回供线性轮询

- 第1个条件：过滤后的实例总数 \geq 最小过滤实例数（默认为1）
- 第2个条件：过滤后的实例比例 $>$ 最小过滤百分比（默认为0）

getEligibleServers

这里的实现逻辑是，遍历所有服务器列表，调用 `this.apply` 方法进行验证，验证通过的节点，会加入到 `results` 这个列表中返回。

```
public List<Server> getEligibleServers(List<Server> servers, Object loadBalancerKey) {
    if (loadBalancerKey == null) {
        return ImmutableList.copyOf(Iterables.filter(servers,
this.getServerOnlyPredicate()));
    } else {
        List<Server> results = Lists.newArrayList();
        for (Server server: servers) {
            if (this.apply(new PredicateKey(loadBalancerKey, server))) {
                results.add(server);
            }
        }
        return results;
    }
}
```

`this.apply`，会进入到 `CompositePredicate.apply` 方法中，代码如下。

```
//CompositePredicate.apply
```

```
@Override
public boolean apply(@Nullable PredicateKey input) {
    return delegate.apply(input);
}
```

`delegate` 的实例是 `AbstractServerPredicate`，代码如下！

```
public static AbstractServerPredicate ofKeyPredicate(final Predicate<PredicateKey> p)
{
    return new AbstractServerPredicate() {
        @Override
        @edu.umd.cs.findbugs.annotations.SuppressWarnings(value = "NP")
        public boolean apply(PredicateKey input) {
            return p.apply(input);
        }
    };
}
```

也就是说，会通过 `AbstractServerPredicate.apply` 方法进行过滤，其中， `input` 表示目标服务器集群的某一个具体节点。

其中 `p`，表示 `AndPredicate` 实例，这里用到了组合 `predicate` 进行判断，而这里的组合判断是 `and` 的关系，用到了 `AndPredicate` 实现。

```
private static class AndPredicate<T> implements Predicate<T>, Serializable {
    private final List<? extends Predicate<? super T>> components;
    private static final long serialVersionUID = 0L;

    private AndPredicate(List<? extends Predicate<? super T>> components) {
        this.components = components;
    }
```

```

public boolean apply(@Nullable T t) {
    for(int i = 0; i < this.components.size(); ++i) { //遍历多个predicate,逐一进行判断。
        if (!((Predicate)this.components.get(i)).apply(t)) {
            return false;
        }
    }

    return true;
}

```

上述代码中，components是由两个predicate组合而成

1. AvailabilityPredicate，过滤熔断状态下的服务以及并发连接过多的服务。
2. ZoneAvoidancePredicate，过滤掉无可用区域的节点。

所以在AndPredicate的apply方法中，需要遍历这两个predicate逐一进行判断。

AvailabilityPredicate

过滤熔断状态下的服务以及并发连接过多的服务，代码如下：

```

@Override
public boolean apply(@Nullable PredicateKey input) {
    LoadBalancerStats stats = getLBStats();
    if (stats == null) {
        return true;
    }
    return !shouldSkipServer(stats.getServerStat(input.getServer()));
}

```

判断是否要跳过这个目标节点，实现逻辑如下。

```

private boolean shouldSkipServer(ServerStats stats) {
    //niws.loadbalancer.availabilityFilteringRule.filterCircuitTripped是否为true
    if ((CIRCUIT_BREAKER_FILTERING.get() && stats.isCircuitBreakerTripped()) //该Server
        是否为断路状态
        || stats.getActiveRequestsCount() >= activeConnectionsLimit.get()) //本机发往
    这个Server未处理完的请求个数是否大于Server实例最大的活跃连接数
        return true;
    }
    return false;
}

```

Server是否为断路状态是如何判断的呢？

ServerStats源码，这里详细源码我们不贴了，说一下机制：

断路是通过时间判断实现的，每次失败记录上次失败时间。如果失败了，则触发判断，是否大于断路的最小失败次数，则判断：

计算断路持续时间： **(2^{失败次数}) * 断路时间因子**，如果大于最大断路时间，则取最大断路时间。判断当前时间是否大于上次失败时间+短路持续时间，如果小于，则是断路状态。这里又涉及三个配置（这里需要将default替换成你调用的微服务名称）：

- niws.loadbalancer.default.connectionFailureCountThreshold，默认为3，触发判断是否断路的最小失败次数，也就是，默认如果失败三次，就会判断是否要断路了。
- niws.loadbalancer.default.circuitTripTimeoutFactorSeconds， 默认为10，断路时间因子，

- `niws.loadbalancer.default.circuitTripMaxTimeoutSeconds`, 默认为30, 最大断路时间

ZoneAvoidancePredicate

`ZoneAvoidancePredicate`, 过滤掉不可用区域的节点, 代码如下!

```

@Override
public boolean apply(@Nullable PredicateKey input) {
    if (!ENABLED.get()) {//查看niws.loadbalancer.zoneAvoidanceRule.enabled配置的熟悉是否
        //为true(默认为true)如果为false没有开启分片过滤 则不进行过滤
        return true;
    }
    ////获取配置的分区字符串 默认为UNKNOWN
    String serverZone = input.getServer().getZone();
    if (serverZone == null) { //如果没有分区, 则不需要进行过滤, 直接返回即可
        // there is no zone information from the server, we do not want to filter
        // out this server
        return true;
    }
    //获取负载均衡的状态信息
    LoadBalancerStats lbStats = getLBStats();
    if (lbStats == null) {
        // no stats available, do not filter
        return true;
    }
    //如果可用区域小于等于1, 也不需要进行过滤直接返回
    if (lbStats.getAvailableZones().size() <= 1) {
        // only one zone is available, do not filter
        return true;
    }
    //针对当前负载信息, 创建一个区域快照, 后续会用快照数据进行计算 (避免后续因为数据变更导致判
    断计算不准确问题)
    Map<String, ZoneSnapshot> zoneSnapshot =
    ZoneAvoidanceRule.createSnapshot(lbStats);
    if (!zoneSnapshot.keySet().contains(serverZone)) { //如果快照信息中没有包含当前服务器
        //所在区域, 则也不需要进行判断。
        // The server zone is unknown to the load balancer, do not filter it out
        return true;
    }
    logger.debug("Zone snapshots: {}", zoneSnapshot);
    //获取有效区域
    Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones(zoneSnapshot,
triggeringLoad.get(), triggeringBlackoutPercentage.get());
    logger.debug("Available zones: {}", availableZones);
    if (availableZones != null) { //有效区域如果包含当前节点, 则返回true, 否则返回false,
        //返回false表示这个区域不可用, 不需要进行目标节点分发。
        return availableZones.contains(input.getServer().getZone());
    } else {
        return false;
    }
}

```

`LoadBalancerStats`, 在每次发起通讯的时候, 状态信息会在控制台打印如下!

```
DynamicServerListLoadBalancer for client goods-service initialized:  
DynamicServerListLoadBalancer:{NFLoadBalancer:name=goods-service,current list of  
Servers=[localhost:9091, localhost:9081],Load balancer stats=Zone stats: {unknown=  
[Zone:unknown; Instance count:2; Active connections count: 0; Circuit breaker  
tripped count: 0; Active connections per server: 0.0;}  
},Server stats: [[Server:localhost:9091; Zone:UNKNOWN; Total Requests:0;  
Successive connection failure:0; Total blackout seconds:0; Last connection  
made:Thu Jan 01 08:00:00 CST 1970; First connection made: Thu Jan 01 08:00:00 CST  
1970; Active Connections:0; total failure count in last (1000) msecs:0; average  
resp time:0.0; 90 percentile resp time:0.0; 95 percentile resp time:0.0; min  
resp time:0.0; max resp time:0.0; stddev resp time:0.0]  
, [Server:localhost:9081; Zone:UNKNOWN; Total Requests:0; Successive connection  
failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00 CST  
1970; First connection made: Thu Jan 01 08:00:00 CST 1970; Active Connections:0;  
total failure count in last (1000) msecs:0; average resp time:0.0; 90 percentile resp  
time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0;  
stddev resp time:0.0]  
]}ServerList:com.netflix.loadbalancer.ConfigurationBasedServerList@74ddb59a
```

getAvailableZones方法的代码如下，用来计算有效可用区域。

```
public static Set<String> getAvailableZones(  
    Map<String, ZoneSnapshot> snapshot, double triggeringLoad,  
    double triggeringBlackoutPercentage) {  
    if (snapshot.isEmpty()) { //如果快照信息为空，返回空  
        return null;  
    }  
    //定义一个集合存储有效区域节点  
    Set<String> availableZones = new HashSet<String>(snapshot.keySet());  
    if (availableZones.size() == 1) { //如果有效区域的集合只有1个，直接返回  
        return availableZones;  
    }  
    //记录有问题的区域集合  
    Set<String> worstZones = new HashSet<String>();  
    double maxLoadPerServer = 0; //定义一个变量，保存所有zone中，平均负载最高值  
    // true: zone有限可用  
    // false: zone全部可用  
    boolean limitedZoneAvailability = false; //  
  
    //遍历所有的区域信息. 对每个zone进行逐一分析  
    for (Map.Entry<String, ZoneSnapshot> zoneEntry : snapshot.entrySet()) {  
        String zone = zoneEntry.getKey(); //得到zone字符串  
        ZoneSnapshot zoneSnapshot = zoneEntry.getValue(); //得到该zone的快照信息  
        int instanceCount = zoneSnapshot.getInstanceCount();  
        if (instanceCount == 0) { //若该zone内一个实例都木有了，那就是完全不可用，那就移除  
            // 该zone, 然后标记zone是有限可用的（并非全部可用）  
            availableZones.remove(zone);  
            limitedZoneAvailability = true;  
        } else {  
            double loadPerServer = zoneSnapshot.getLoadPerServer(); //获取该区域的平均负  
            载  
            // 机器的熔断总数 / 总实例数已经超过了阈值（默认为1，也就是全部熔断才会认为该zone  
            完全不可用）  
            if (((double) zoneSnapshot.getCircuitTrippedCount())  
                / instanceCount >= triggeringBlackoutPercentage  
                || loadPerServer < 0) { //loadPerServer表示当前区域所有节点都熔断了。  
                availableZones.remove(zone);  
                limitedZoneAvailability = true;  
            } else { // 进入到这个逻辑，说明并不是完全不可用，就看看区域的状态  
            
```

```

        // 如果当前负载和最大负载相当，那认为当前区域状态很不好，加入到worstZones中
        if (Math.abs(loadPerServer - maxLoadPerServer) < 0.00001d) {
            // they are the same considering double calculation
            // round error
            worstZones.add(zone);

        } else if (loadPerServer > maxLoadPerServer) { // 或者若当前负载大于最大负载了。
            maxLoadPerServer = loadPerServer;
            worstZones.clear();
            worstZones.add(zone);
        }
    }
}

// 如果最大负载小于设定的负载阈值 并且limitedZoneAvailability=false
// 说明全部zone都可用，并且最大负载都还没有达到阈值，那就把全部zone返回
if (maxLoadPerServer < triggeringLoad && !limitedZoneAvailability) {
    // zone override is not needed here
    return availableZones;
}

//若最大负载超过阈值，就不能全部返回，则直接从负载最高的区域中随机返回一个，这么处理的目的是把负载最高的那个哥们剔除掉，再返回结果。
String zoneToAvoid = randomChooseZone(snapshot, worstZones);
if (zoneToAvoid != null) {
    availableZones.remove(zoneToAvoid);
}
return availableZones;

}

```

上述逻辑还是比较复杂的，我们通过一个简单的文字进行说明：

1. 如果 `zone` 为 `null`，那么也就是没有可用区域，直接返回 `null`
2. 如果 `zone` 的可用区域为 1，也没有什么可以选择的，直接返回这一个
3. 使用 `Set<String> worstZones` 记录所有 `zone` 中比较状态不好的的 `zone` 列表，用 `maxLoadPerServer` 表示所有 `zone` 中负载最高的区域；用 `limitedZoneAvailability` 表示是否是部分 `zone` 可用（`true`: 部分可用，`false`: 全部可用），接着我们需要遍历所有的 `zone` 信息，逐一进行判断从而对有效 `zone` 的结果进行处理。
 1. 如果当前 `zone` 的 `instanceCount` 为 0，那就直接把这个区域移除就行，并且标记 `limitedZoneAvailability` 为部分可用，没什么好说的。
 2. 获得当前总的平均负载 `loadPerServer`，如果 `zone` 内的 `熔断实例数 / 总实例数 >= triggeringBlackoutPercentage` 或者 `loadPerServer < 0` 的话，说明当前区域有问题，直接执行 `remove` 移除当前 `zone`，并且 `limitedZoneAvailability=true`。
 1. (`熔断实例数 / 总实例数 >= 阈值`，标记为当前 `zone` 就不可用了（移除掉），这个很好理解。这个阈值为 `0.99999d` 也就说所有的 `Server` 实例被熔断了，该 `zone` 才算不可用了）。
 2. `loadPerServer = -1`，也就说当所有实例都熔断了。这两个条件判断都差不多，都是判断这个区域的可用性。
 3. 如果当前 `zone` 没有达到阈值，则判断区域的负载情况，从所有 `zone` 中找到负载最高的区域（负载差值在 `0.00001d`），则把这些区域加入到 `worstZones` 列表，也就是这个集合保存的是负载较高的区域。
4. 通过上述遍历对区域数据进行计算后，最后要设置返回的有效区域数据。

- 最高负载 `maxLoadPerServer` 仍旧小于提供的 `triggeringLoad` 阈值，并且并且 `limitedZoneAvailability=false`（就是说所有 zone 都可用的情况下），那就返回所有的 zone： `availableZones`。（也就是所有区域的负载都在阈值范围内并且每个区域内的节点都还存活状态，就全部返回）
- 否则，最大负载超过阈值或者某些区域存在部分不可用的节点时，就从这些负载较高的节点 `worstZones` 中随机移除一个

AbstractServerPredicate

在回答下面的代码，通过 `getEligibleServers` 判断可用服务节点后，如果可用节点不为 0，则执行 `incrementAndGetModulo` 方法进行轮询。

```
public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers, Object loadBalancerKey) {
    List<Server> eligible = getEligibleServers(servers, loadBalancerKey);
    if (eligible.size() == 0) {
        return Optional.absent();
    }
    return Optional.of(eligible.get(incrementAndGetModulo(eligible.size())));
}
```

该方法是通过轮询来实现，代码如下！

```
private int incrementAndGetModulo(int modulo) {
    for (;;) {
        int current = nextIndex.get();
        int next = (current + 1) % modulo;
        if (nextIndex.compareAndSet(current, next) && current < modulo)
            return current;
    }
}
```

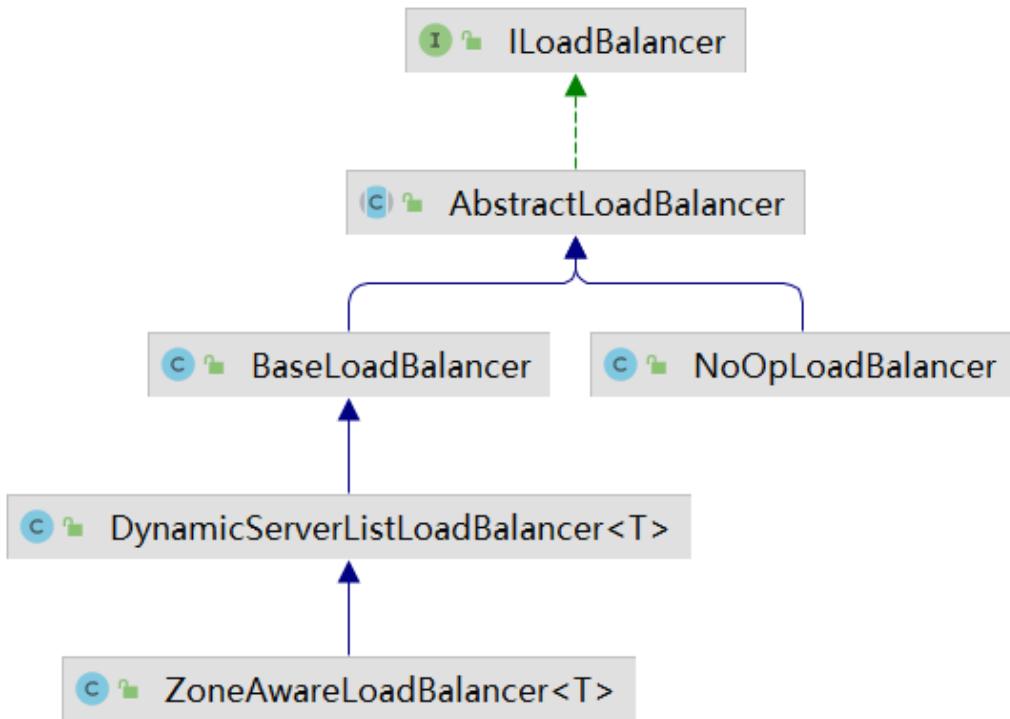
服务列表的加载过程

在本实例中，我们将服务列表配置在 `application.properties` 文件中，意味着在某个时候会加载这个列表，保存在某个位置，那它是在什么时候加载的呢？

在 `RibbonClientConfiguration` 这个配置类中，有下面这个 Bean 的声明，（该 Bean 是条件触发）它用来定义默认的负载均衡实现。

```
@Bean
@ConditionalOnMissingBean
public ILoadBalancer ribbonLoadBalancer(IClientConfig config,
                                         ServerList<Server> serverList,
                                         ServerListFilter<Server> serverListFilter,
                                         IRule rule, IPing ping, ServerListUpdater
                                         serverListUpdater) {
    if (this.propertiesFactory.isSet(ILoadBalancer.class, name)) {
        return this.propertiesFactory.get(ILoadBalancer.class, config, name);
    }
    return new ZoneAwareLoadBalancer<>(config, rule, ping, serverList,
                                         serverListFilter, serverListUpdater);
}
```

前面分析过，它的类关系图如下！



当 `ZoneAwareLoadBalancer` 在初始化时，会调用父类 `DynamicServerListLoadBalancer` 的构造方法，代码如下。

```

public DynamicServerListLoadBalancer(IClientConfig clientConfig, IRule rule, IPing
ping,
                                         ServerList<T> serverList, ServerListFilter<T>
filter,
                                         ServerListUpdater serverListUpdater) {
    super(clientConfig, rule, ping);
    this.serverListImpl = serverList;
    this.filter = filter;
    this.serverListUpdater = serverListUpdater;
    if (filter instanceof AbstractServerListFilter) {
        ((AbstractServerListFilter)
filter).setLoadBalancerStats(getLoadBalancerStats());
    }
    restOfInit(clientConfig);
}
  
```

restOfInit

`restOfInit` 方法主要做两件事情。

1. 开启动态更新Server的功能
2. 更新Server列表

```

void restOfInit(IClientConfig clientConfig) {
    boolean primeConnection = this.isEnabledPrimingConnections();
    // turn this off to avoid duplicated asynchronous priming done in
    BaseLoadBalancer.setServerList()
    this.setEnabledPrimingConnections(false);
    enableAndInitLearnNewServersFeature(); //开启动态更新Server
    updateListOfServers(); //更新Server列表
  
```

```

    if (primeConnection && this.getPrimeConnections() != null) {
        this.getPrimeConnections()
            .primeConnections(getReachableServers());
    }
    this.setEnablePrimingConnections(primeConnection);
    LOGGER.info("DynamicServerListLoadBalancer for client {} initialized: {}", 
clientConfig.getClientName(), this.toString());
}

```

updateListOfServers

全量更新一次服务列表。

```

public void updateListOfServers() {
    List<T> servers = new ArrayList<T>();
    if (serverListImpl != null) {
        servers = serverListImpl.getUpdatedListOfServers();
        LOGGER.debug("List of Servers for {} obtained from Discovery client: {}",
                     getIdentifier(), servers);

        if (filter != null) {
            servers = filter.getFilteredListOfServers(servers);
            LOGGER.debug("Filtered List of Servers for {} obtained from Discovery
client: {}",
                         getIdentifier(), servers);
        }
    }
    updateAllServerList(servers);
}

```

上述代码解释如下

1. 由于我们是通过 `application.properties` 文件配置的静态服务地址列表，所以此时 `serverListImpl` 的实例为： `ConfigurationBasedServerList`，调用 `getUpdatedListOfServers` 方法时，返回的是在 `application.properties` 文件中定义的服务列表。
2. 判断是否需要 `filter`，如果有，则通过 `filter` 进行服务列表过滤。

最后调用 `updateAllServerList`，更新所有 Server 到本地缓存中。

```

protected void updateAllServerList(List<T> ls) {
    // other threads might be doing this - in which case, we pass
    if (serverListUpdateInProgress.compareAndSet(false, true)) {
        try {
            for (T s : ls) {
                s.setAlive(true); // set so that clients can start using these
                // servers right away instead
                // of having to wait out the ping cycle.
            }
            setServersList(ls);
            super.forceQuickPing();
        } finally {
            serverListUpdateInProgress.set(false);
        }
    }
}

```

动态Ping机制

在Ribbon中，基于Ping机制，目标服务地址也会发生动态变更，具体的实现方式在 `DynamicServerListLoadBalancer.restOfInit` 方法中

```
void restOfInit(IClientConfig clientConfig) {
    boolean primeConnection = this.isEnablePrimingConnections();
    // turn this off to avoid duplicated asynchronous priming done in
    BaseLoadBalancer.setServerList()
    this.setEnablePrimingConnections(false);
    enableAndInitLearnNewServersFeature(); //开启定时任务动态更新

    updateListOfServers();
    if (primeConnection && this.getPrimeConnections() != null) {
        this.getPrimeConnections()
            .primeConnections(getReachableServers());
    }
    this.setEnablePrimingConnections(primeConnection);
    LOGGER.info("DynamicServerListLoadBalancer for client {} initialized: {}", clientConfig.getClientName(), this.toString());
}
```

```
public void enableAndInitLearnNewServersFeature() {
    LOGGER.info("Using serverListUpdater {}", serverListUpdater.getClass().getSimpleName());
    serverListUpdater.start(updateAction);
}
```

注意，这里会启动一个定时任务，而定时任务所执行的程序是 `updateAction`，它是一个匿名内部类，定义如下。

```
protected final ServerListUpdater.UpdateAction updateAction = new
ServerListUpdater.UpdateAction() {
    @Override
    public void doUpdate() {
        updateListOfServers();
    }
};
```

定时任务的启动方法如下，这个任务每隔30s执行一次。

```
public synchronized void start(final UpdateAction updateAction) {
    if (isActive.compareAndSet(false, true)) {
        final Runnable wrapperRunnable = new Runnable() {
            @Override
            public void run() {
                if (!isActive.get()) {
                    if (scheduledFuture != null) {
                        scheduledFuture.cancel(true);
                    }
                    return;
                }
                try {
                    updateAction.doUpdate(); //执行具体的任务。
                    lastUpdated = System.currentTimeMillis();
                } catch (Exception e) {
                    logger.warn("Failed one update cycle", e);
                }
            }
        };
        scheduledFuture = executor.schedule(wrapperRunnable, 30, TimeUnit.SECONDS);
    }
}
```

```

        }
    };

    scheduledFuture = getRefreshExecutor().scheduleWithFixedDelay(
        wrapperRunnable,
        initialDelayMs, //1000
        refreshIntervalMs, //30000
        TimeUnit.MILLISECONDS
    );
} else {
    logger.info("Already active, no-op");
}
}
}

```

当30s之后触发了 `doUpdate` 方法后，最终进入到 `updateAllServerList` 方法

```

protected void updateAllServerList(List<T> ls) {
    // other threads might be doing this - in which case, we pass
    if (serverListUpdateInProgress.compareAndSet(false, true)) {
        try {
            for (T s : ls) {
                s.setAlive(true); // set so that clients can start using these
                // servers right away instead
                // of having to wait out the ping cycle.
            }
            setServersList(ls);
            super.forceQuickPing();
        } finally {
            serverListUpdateInProgress.set(false);
        }
    }
}

```

其中，会调用 `super.forceQuickPing();` 进行心跳健康检测。

```

public void forceQuickPing() {
    if (canSkipPing()) {
        return;
    }
    logger.debug("LoadBalancer [{}]: forceQuickPing invoking", name);

    try {
        new Pinger(pingStrategy).runPinger();
    } catch (Exception e) {
        logger.error("LoadBalancer [{}]: Error running forceQuickPing()", name, e);
    }
}

```

RibbonLoadBalancerClient.execute

经过上述分析，再回到 `RibbonLoadBalancerClient.execute` 方法！

```

public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)
    throws IOException {
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
    Server server = getServer(loadBalancer, hint);
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }
    RibbonServer ribbonServer = new RibbonServer(serviceId, server,
                                                   isSecure(server, serviceId),
                                                   serverIntrospector(serviceId).getMetadata(server));
}

return execute(serviceId, ribbonServer, request);
}

```

此时，`Server server = getServer(loadBalancer, hint);`这行代码，会返回一个具体的目标服务器。

其中，在调用`execute`方法之前，会包装一个`RibbonServer`对象传递下去，它的主要作用是用来记录请求的负载信息。

```

@Override
public <T> T execute(String serviceId, ServiceInstance serviceInstance,
                      LoadBalancerRequest<T> request) throws IOException {
    Server server = null;
    if (serviceInstance instanceof RibbonServer) {
        server = ((RibbonServer) serviceInstance).getServer();
    }
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }

    RibbonLoadBalancerContext context = this.clientFactory
        .getLoadBalancerContext(serviceId);
    RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);

    try {
        T returnVal = request.apply(serviceInstance);
        statsRecorder.recordStats(returnVal); //记录请求状态
        return returnVal;
    }
    // catch IOException and rethrow so RestTemplate behaves correctly
    catch (IOException ex) {
        statsRecorder.recordStats(ex); //记录请求状态
        throw ex;
    }
    catch (Exception ex) {
        statsRecorder.recordStats(ex);
        ReflectionUtils.rethrowRuntimeException(ex);
    }
    return null;
}

```

request.apply

`request`是`LoadBalancerRequest`接口，它里面提供了一个`apply`方法，但是从代码中我们发现这个方法并没有实现类，那么它是在哪里实现的呢？

继续又往前分析发现，这个request对象是从 `LoadBalancerInterceptor` 的 `intercept` 方法中传递过来的。

```
public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
final ClientHttpRequestExecution execution) throws IOException {
    URI originalUri = request.getURI();
    String serviceName = originalUri.getHost();
    Assert.state(serviceName != null, "Request URI does not contain a valid hostname:
" + originalUri);
    return (ClientHttpResponse)this.loadBalancer.execute(serviceName,
this.requestFactory.createRequest(request, body, execution));
}
```

而 `request` 的传递，是通过 `this.requestFactory.createRequest(request, body, execution)` 创建而来，于是我们找到这个方法。

```
public LoadBalancerRequest<ClientHttpResponse> createRequest(final HttpRequest
request, final byte[] body, final ClientHttpRequestExecution execution) {
    return (instance) -> {
        HttpRequest serviceRequest = new ServiceRequestWrapper(request, instance,
this.loadBalancer);
        LoadBalancerRequestTransformer transformer;
        if (this.transformers != null) {
            for(Iterator var6 = this.transformers.iterator(); var6.hasNext();
serviceRequest = transformer.transformRequest((HttpRequest)serviceRequest, instance))
{
                transformer = (LoadBalancerRequestTransformer)var6.next();
            }
        }

        return execution.execute((HttpRequest)serviceRequest, body);
    };
}
```

从代码中发现，它是一个用lambda表达式实现的匿名内部类。在该内部类中，创建了一个 `ServiceRequestWrapper`，这个 `ServiceRequestWrapper` 实际上就是 `HttpRequestWrapper` 的一个子类，`ServiceRequestWrapper` 重写了 `HttpRequestWrapper` 的 `getURI()` 方法，重写的 `URI` 实际上就是通过调用 `LoadBalancerClient` 接口的 `reconstructURI` 函数来重新构建一个 `URI` 进行访问。

InterceptingClientHttpRequest.execute

上述代码执行的 `execution.execute`，又会进入到 `InterceptingClientHttpRequest.execute` 方法中，代码如下。

```

public ClientHttpResponse execute(HttpServletRequest request, byte[] body) throws IOException
{
    if (this.iterator.hasNext()) {           ClientHttpRequestInterceptor
        nextInterceptor = this.iterator.next();      return
        nextInterceptor.intercept(request, body, this); }   else {       HttpMethod
        method = request.getMethod();           Assert.state(method != null, "No standard HTTP
        method");           ClientHttpRequest delegate =
        requestFactory.createRequest(request.getURI(), method); //注意这里
        request.getHeaders().forEach((key, value) -> delegate.getHeaders().addAll(key,
        value));           if (body.length > 0) {           if (delegate instanceof
        StreamingHttpOutputMessage) {           StreamingHttpOutputMessage
        streamingOutputMessage = (StreamingHttpOutputMessage) delegate;
        streamingOutputMessage.setBody(outputStream -> StreamUtils.copy(body, outputStream));
        }           else {           StreamUtils.copy(body,
        delegate.getBody()); }     }   return delegate.execute();  }
}

```

此时需要注意，`request`对象的实例是`HttpRequestWrapper`。

request.getURI()

当调用`request.getURI()`获取目标地址创建http请求时，会调用`ServiceRequestWrapper`中的`.getURI()`方法。

```

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

```

在这个方法中，调用`RibbonLoadBalancerClient`实例中的`reconstructURI`方法，根据`service-id`生成目标服务地址。

RibbonLoadBalancerClient.reconstructURI

```

public URI reconstructURI(ServiceInstance instance, URI original) {
Assert.notNull(instance, "instance can not be null");   String serviceId =
instance.getServiceId(); //获取实例id，也就是服务名称   RibbonLoadBalancerContext
context = this.clientFactory           .getLoadBalancerContext(serviceId); //获取
RibbonLoadBalancerContext上下文，这个是从spring容器中获取的对象实例。   URI uri;
Server server;   if (instance instanceof RibbonServer) { //如果instance为
RibbonServer   RibbonServer ribbonServer = (RibbonServer) instance;
server = ribbonServer.getServer(); //获取目标服务器的Server信息   uri =
updateToSecureConnectionIfNeeded(original, ribbonServer); //判断是否需要更新成一个安全连
接。 }   else { //如果是一个普通的http地址   server = new
Server(instance.getScheme(), instance.getHost(),
instance.getPort());   IClientConfig clientConfig =
clientFactory.getClientConfig(serviceId);   ServerIntrospector
serverIntrospector = serverIntrospector(serviceId);   uri =
updateToSecureConnectionIfNeeded(original, clientConfig,
serverIntrospector, server); }   return
context.reconstructURIWithServer(server, uri); //调用这个方法拼接成一个真实的目标服务器地
址。 }

```

