深入理解Ribbon

LoadBalancerClient

在Riibon中一个非常重要的组件为LoadBalancerClient，它作为负载均衡的一个客户端。它在spring-cloud-commons包下：
的LoadBalancerClient是一个接口，它继承ServiceInstanceChooser，它的实现类是RibbonLoadBalancerClient，这三者之间的关系如下图：

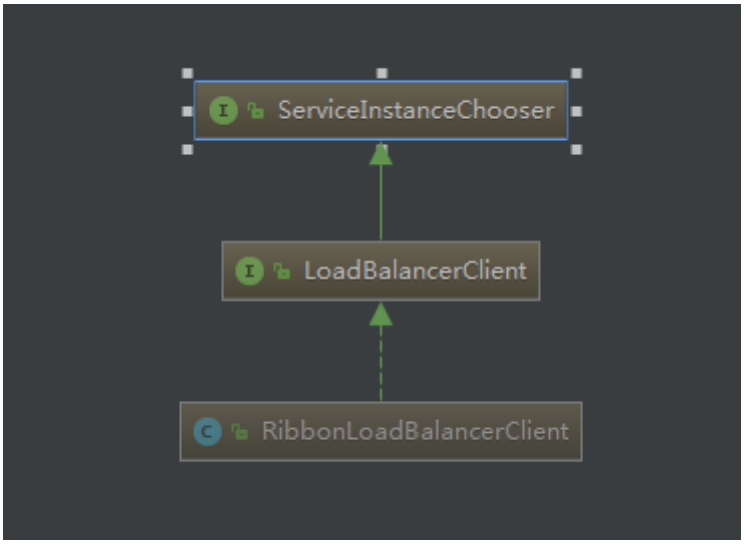

image.png

其中LoadBalancerClient接口，有如下三个方法，其中excute()为执行请求，reconstructURI()用来重构url：

```
public interface LoadBalancerClient extends ServiceInstanceChooser {
  <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
IOException;
  <T> T execute(String serviceId, ServiceInstance serviceInstance,
LoadBalancerRequest<T> request) throws IOException;
  URI reconstructURI(ServiceInstance instance, URI original);

}
```

ServiceInstanceChooser接口，主要有一个方法，用来根据serviceId来获取ServiceInstance，代码如下:

```
public interface ServiceInstanceChooser {

    ServiceInstance choose(String serviceId);
}
```

LoadBalancerClient的实现类为RibbonLoadBalancerClient，这个类是非常重要的一个类，最终的负载均衡的请求处理，由它来执行。它的部分源码如下：

```
public class RibbonLoadBalancerClient implements LoadBalancerClient {

...//省略代码

@Override
    public ServiceInstance choose(String serviceId) {
        Server server = getServer(serviceId);
        if (server == null) {
            return null;
        }
        return new RibbonServer(serviceId, server, isSecure(server, serviceId),
                serverIntrospector(serviceId).getMetadata(server));
    }



protected Server getServer(String serviceId) {
        return getServer(getLoadBalancer(serviceId));
    }


protected Server getServer(ILoadBalancer loadBalancer) {
        if (loadBalancer == null) {
            return null;
        }
        return loadBalancer.chooseServer("default"); // TODO: better handling of
key
    }


protected ILoadBalancer getLoadBalancer(String serviceId) {
        return this.clientFactory.getLoadBalancer(serviceId);
    }

    ...//省略代码
```

在RibbonLoadBalancerClient的源码中，其中choose()方法是选择具体服务实例的一个方法。该方法通过getServer()方法去获取实例，经过源码跟踪，最终交给了ILoadBalancer类去选择服务实例。

```
public interface ILoadBalancer {

    public void addServers(List<Server> newServers);
    public Server chooseServer(Object key);
    public void markServerDown(Server server);
    public List<Server> getReachableServers();
    public List<Server> getAllServers();
}
```

其中，addServers()方法是添加一个Server集合；chooseServer()方法是根据key去获取Server；markServerDown()方法用来标记某个服务下线；getReachableServers()获取可用的Server集合；getAllServers()获取所有的Server集合。

DynamicServerListLoadBalancer

它的继承类为BaseLoadBalancer，它的实现类为DynamicServerListLoadBalancer，这三者之间的关系如下：

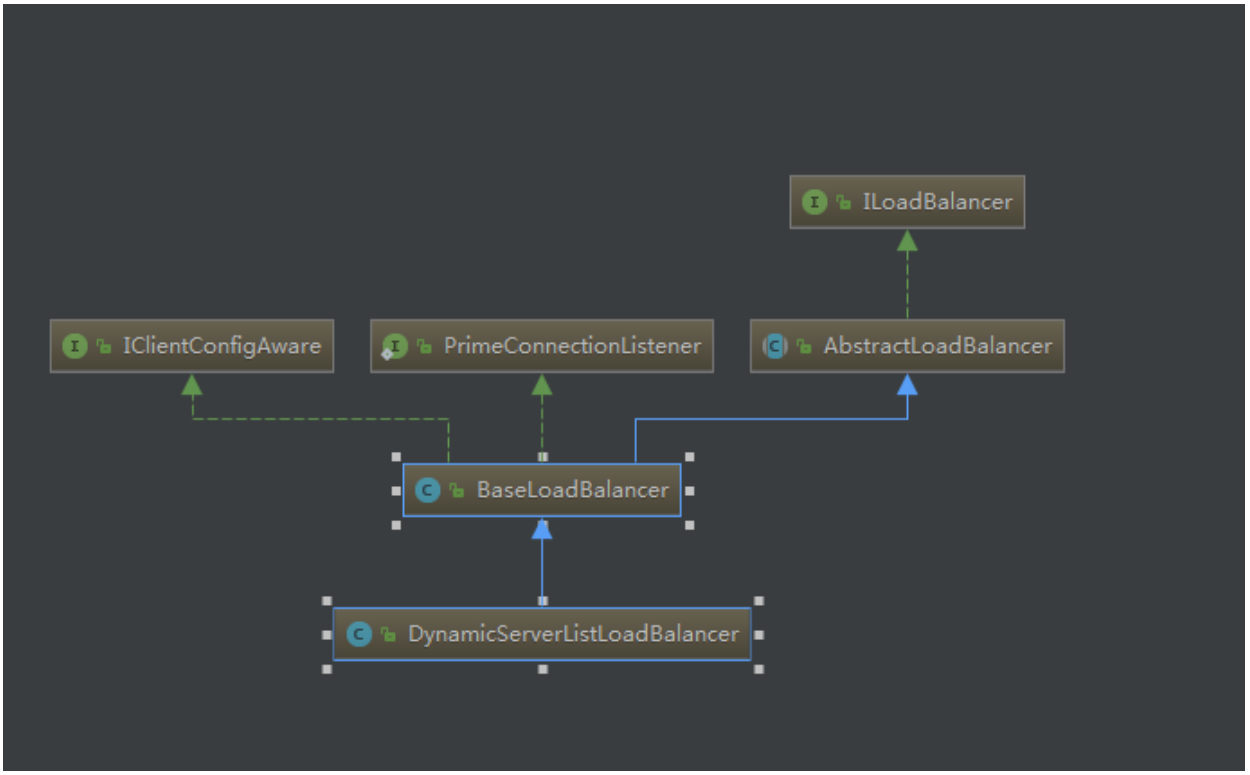

image.png

查看上述三个类的源码，可用发现，配置以下信息，IClientConfig、IRule、IPing 、 ServerList 、 ServerListFilter 和 ILoadBalancer ， 查 看 BaseLoadBalancer类，它默认的情况下，实现了以下配置：

- IClientConfig ribbonClientConfig: DefaultClientConfigImpl配置
- IRule ribbonRule: RoundRobinRule 路由策略
- IPing ribbonPing: DummyPing
- ServerList ribbonServerList: ConfigurationBasedServerList
- ServerListFilter ribbonServerListFilter: ZonePreferenceServerListFilter
- ILoadBalancer ribbonLoadBalancer: ZoneAwareLoadBalancer

IClientConfig 用于对客户端或者负载均衡的配置，它的默认实现类为DefaultClientConfigImpl。
IRule用于复杂均衡的策略，它有三个方法，其中choose()是根据key 来获取server,setLoadBalancer() 和 getLoadBalancer() 是 用 来 设 置 和 获 取 ILoadBalancer的，它的源码如下：

```
public interface IRule{

    public Server choose(Object key);

    public void setLoadBalancer(ILoadBalancer lb);

    public ILoadBalancer getLoadBalancer();
}
```

IRule有很多默认的实现类，这些实现类根据不同的算法和逻辑来处理负载均衡。Ribbon实现的IRule有一下。在大多数情况下，这些默认的实现类是可以满足需求的，如果有特性的需求，可以自己实现。

- BestAvailableRule 选择最小请求数
- ClientConfigEnabledRoundRobinRule 轮询
- RandomRule 随机选择一个server
- RoundRobinRule 轮询选择server
- RetryRule 根据轮询的方式重试

- WeightedResponseTimeRule 根据响应时间去分配一个weight， weight越低，被选择的可能性就越低
- ZoneAvoidanceRule 根据server的zone区域和可用性来轮询选择

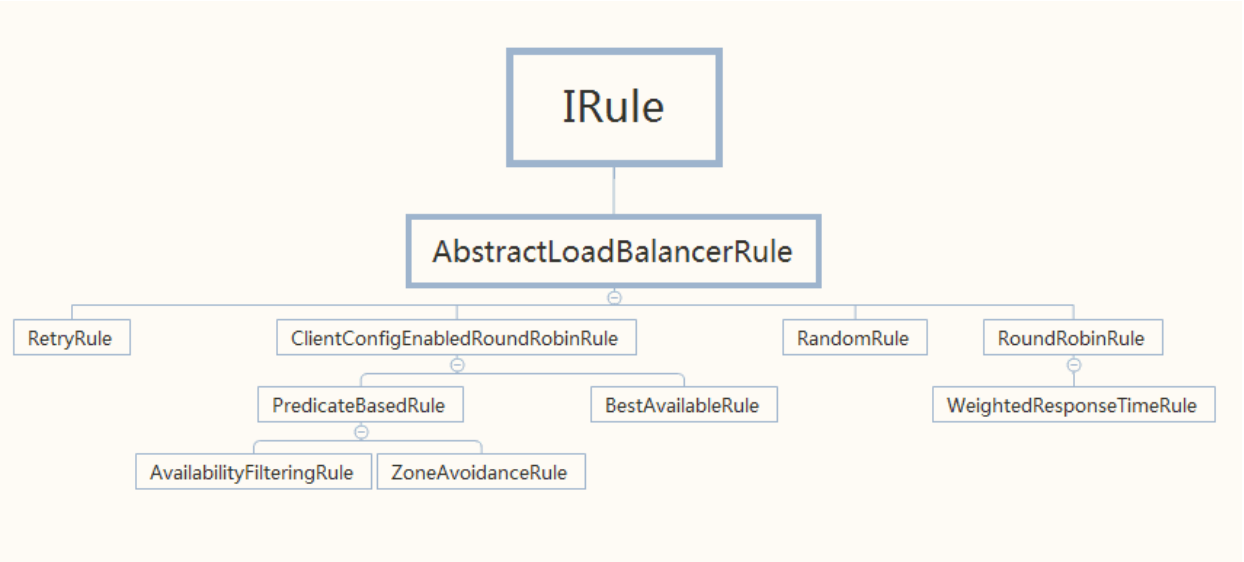

image.png

IPing是用来想server发生”ping”，来判断该server是否有响应，从而判断该server是否可用。它有一个isAlive()方法，它的源码如下：

```
public interface IPing {
    public boolean isAlive(Server server);
}
```

IPing 的实现类有 PingUrl、 PingConstant、 NoOpPing、 DummyPing 和 NIWSDiscoveryPing。它门之间的关系如下：

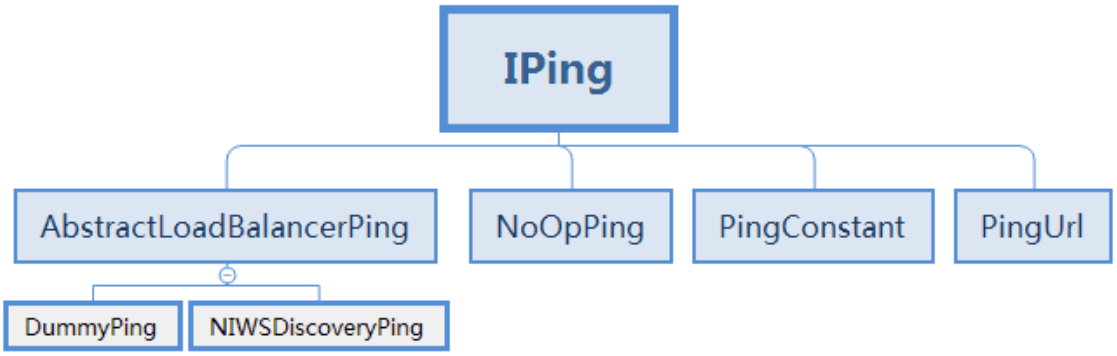

image.png

- PingUrl 真实的去ping 某个url，判断其是否alive
- PingConstant 固定返回某服务是否可用，默认返回true，即可用
- NoOpPing 不去ping,直接返回true,即可用。
- DummyPing 直接返回true，并实现了initWithNiwsConfig方法。

ServerList是定义获取所有的server的注册列表信息的接口，它的代码如下：

```
public interface ServerList<T extends Server> {

    public List<T> getInitialListOfServers();
    public List<T> getUpdatedListOfServers();

}
```

ServerListFilter接口，定于了可根据配置去过滤或者根据特性动态获取符合条件的server列表的方法，代码如下：

```
public interface ServerListFilter<T extends Server> {

    public List<T> getFilteredListOfServers(List<T> servers);

}
```

阅读DynamicServerListLoadBalancer的源码，DynamicServerListLoadBalancer的构造函数中有个initWithNiwsConfig()方法。在改方法中，经过一系列的初始化配置，最终执行了restOfInit()方法。其代码如下：

```
 public DynamicServerListLoadBalancer(IClientConfig clientConfig) {
        initWithNiwsConfig(clientConfig);
    }

    @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);

            restOfInit(clientConfig);
        } catch (Exception e) {
            throw new RuntimeException(
                    "Exception while initializing NIWSDiscoveryLoadBalancer:"
                            + clientConfig.getClientName()
                            + ", niwsClientConfig:" + clientConfig, e);
        }
    }
```

在restOfInit()方法上，有一个 updateListOfServers()的方法，该方法是用来获取所有的ServerList的。

```
 void restOfInit(IClientConfig clientConfig) {
        boolean primeConnection = this.isEnablePrimingConnections();
        // turn this off to avoid duplicated asynchronous priming done in
```

```
BaseLoadBalancer.setServerList()
        this.setEnablePrimingConnections(false);
        enableAndInitLearnNewServersFeature();

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

进一步跟踪updateListOfServers()方法的源码，最终由serverListImpl.getUpdatedListOfServers()获取所有的服务列表的，代码如下：

```
   @VisibleForTesting
    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);
    }
```

而serverListImpl是ServerList接口的具体实现类。跟踪代码，ServerList的实现类为DiscoveryEnabledNIWSServerList，在ribbon-eureka.jar的

com.netflix.niws.loadbalancer下。其中DiscoveryEnabledNIWSServerList有getInitialListOfServers()和getUpdatedListOfServers()方法，具体代码如下：

```
@Override
    public List<DiscoveryEnabledServer> getInitialListOfServers(){
        return obtainServersViaDiscovery();
    }

    @Override
    public List<DiscoveryEnabledServer> getUpdatedListOfServers(){
        return obtainServersViaDiscovery();
    }
```

继续跟踪源码，obtainServersViaDiscovery（），是根据eurekaClientProvider.get()来回去EurekaClient，再根据EurekaClient来获取注册列表信息，代码如下：

```
 private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
        List<DiscoveryEnabledServer> serverList = new
ArrayList<DiscoveryEnabledServer>();

        if (eurekaClientProvider == null || eurekaClientProvider.get() == null)
{
            logger.warn("EurekaClient has not been initialized yet, returning an
empty list");
            return new ArrayList<DiscoveryEnabledServer>();
        }

        EurekaClient eurekaClient = eurekaClientProvider.get();
        if (vipAddresses!=null){
            for (String vipAddress : vipAddresses.split(",")) {
                // if targetRegion is null, it will be interpreted as the same
region of client
                List<InstanceInfo> listOfInstanceInfo =
eurekaClient.getInstancesByVipAddress(vipAddress, isSecure, targetRegion);
                for (InstanceInfo ii : listOfInstanceInfo) {
                    if (ii.getStatus().equals(InstanceStatus.UP)) {

                        if(shouldUseOverridePort){
```

```
                                if(logger.isDebugEnabled()){
                                    logger.debug("Overriding port on client name: "
+ clientName + " to " + overridePort);
                                }

                                // copy is necessary since the InstanceInfo builder
just uses the original reference,
                                // and we don't want to corrupt the global eureka
copy of the object which may be
                                // used by other clients in our system
                                InstanceInfo copy = new InstanceInfo(ii);

                                if(isSecure){
                                    ii = new
InstanceInfo.Builder(copy).setSecurePort(overridePort).build();
                                }else{
                                    ii = new
InstanceInfo.Builder(copy).setPort(overridePort).build();
                                }
                            }

                            DiscoveryEnabledServer des = new
DiscoveryEnabledServer(ii, isSecure, shouldUseIpAddr);
                            des.setZone(DiscoveryClient.getZone(ii));
                            serverList.add(des);
                        }
                    }
                    if (serverList.size()>0 && prioritizeVipAddressBasedServers){
                        break; // if the current vipAddress has servers, we dont use
subsequent vipAddress based servers
                    }
                }
            }
            return serverList;
        }
```

其中eurekaClientProvider的实现类是LegacyEurekaClientProvider，它是一个获取eurekaClient类，通过静态的方法去获取eurekaClient，其代码如下：

```
class LegacyEurekaClientProvider implements Provider<EurekaClient> {

    private volatile EurekaClient eurekaClient;

    @Override
    public synchronized EurekaClient get() {
        if (eurekaClient == null) {
            eurekaClient = DiscoveryManager.getInstance().getDiscoveryClient();
        }

        return eurekaClient;
    }
}
```

EurekaClient的实现类为DiscoveryClient，在之前已经分析了它具有服务注册、获取服务注册列表等的全部功能。
由此可见，负载均衡器是从EurekaClient获取服务信息，并根据IRule去路由，并且根据IPing去判断服务的可用性。
那么现在还有个问题，负载均衡器多久一次去获取一次从Eureka Client获取注册信息呢。
在BaseLoadBalancer类下，BaseLoadBalancer的构造函数，该构造函数开启了一个PingTask任务，代码如下：

```
   public BaseLoadBalancer(String name, IRule rule, LoadBalancerStats stats,
            IPing ping, IPingStrategy pingStrategy) {
        ...//代码省略
        setupPingTask();
         ...//代码省略
    }
```

setupPingTask()的具体代码逻辑，它开启了ShutdownEnabledTimer执行PingTask任务，在默认情况下pingIntervalSeconds为10，即每10秒钟，想EurekaClient发送一次”ping”。

```
    void setupPingTask() {
        if (canSkipPing()) {
            return;
        }
```

```
        if (lbTimer != null) {
            lbTimer.cancel();
        }
        lbTimer = new ShutdownEnabledTimer("NFLoadBalancer-PingTimer-" + name,
                true);
        lbTimer.schedule(new PingTask(), 0, pingIntervalSeconds * 1000);
        forceQuickPing();
    }
```

PingTask源码，即new一个Pinger对象，并执行runPinger()方法。

```
class PingTask extends TimerTask {
        public void run() {
            try {
                new Pinger(pingStrategy).runPinger();
            } catch (Exception e) {
                logger.error("LoadBalancer [{}]: Error pinging", name, e);
            }
        }
    }
```

查看Pinger的runPinger()方法，最终根据 pingerStrategy.pingServers(ping, allServers)来获取服务的可用性，如果该返回结果，如之前相同，则不去向EurekaClient获取注册列表，如果不同则通知ServerStatusChangeListener或者changeListeners发生了改变，进行更新或者重新拉取。

```
  public void runPinger() throws Exception {
            if (!pingInProgress.compareAndSet(false, true)) {
                return; // Ping in progress - nothing to do
            }

            // we are "in" - we get to Ping

            Server[] allServers = null;
            boolean[] results = null;

            Lock allLock = null;
            Lock upLock = null;

            try {
                /*
```

```
                 * The readLock should be free unless an addServer operation is
                 * going on...
                 */
                allLock = allServerLock.readLock();
                allLock.lock();
                allServers = allServerList.toArray(new
Server[allServerList.size()]);
                allLock.unlock();

                int numCandidates = allServers.length;
                results = pingerStrategy.pingServers(ping, allServers);

                final List<Server> newUpList = new ArrayList<Server>();
                final List<Server> changedServers = new ArrayList<Server>();

                for (int i = 0; i < numCandidates; i++) {
                    boolean isAlive = results[i];
                    Server svr = allServers[i];
                    boolean oldIsAlive = svr.isAlive();

                    svr.setAlive(isAlive);

                    if (oldIsAlive != isAlive) {
                        changedServers.add(svr);
                        logger.debug("LoadBalancer [{}]:  Server [{}] status
changed to {}",
                            name, svr.getId(), (isAlive ? "ALIVE" : "DEAD"));
                    }

                    if (isAlive) {
                        newUpList.add(svr);
                    }
                }
                upLock = upServerLock.writeLock();
                upLock.lock();
                upServerList = newUpList;
                upLock.unlock();

                notifyServerStatusChangeListener(changedServers);
```

```
            } finally {
                pingInProgress.set(false);
            }
        }
```

由此可见，LoadBalancerClient是在初始化的时候，会向Eureka回去服务注册列表，并且向通过10s一次向EurekaClient发送“ping”，来判断服务的可用性，如果服务的可用性发生了改变或者服务数量和之前的不一致，则更新或者重新拉取。LoadBalancerClient有了这些服务注册列表，就可以根据具体的IRule来进行负载均衡。

RestTemplate是如何和Ribbon结合的

最后，回答问题的本质，为什么在RestTemplate加一个@LoadBalance注解就可可以开启负载均衡呢?

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

全局搜索ctr+shift+f @LoadBalanced有哪些类用到了LoadBalanced有哪些类用到了， 发现LoadBalancerAutoConfiguration类，即LoadBalancer自动配置类。

```
@Configuration
@ConditionalOnClass(RestTemplate.class)
@ConditionalOnBean(LoadBalancerClient.class)
@EnableConfigurationProperties(LoadBalancerRetryProperties.class)
public class LoadBalancerAutoConfiguration {

@LoadBalanced
    @Autowired(required = false)
    private List<RestTemplate> restTemplates = Collections.emptyList();
}
    @Bean
    public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
            final List<RestTemplateCustomizer> customizers) {
        return new SmartInitializingSingleton() {
            @Override
            public void afterSingletonsInstantiated() {
```

```
                for (RestTemplate restTemplate :
LoadBalancerAutoConfiguration.this.restTemplates) {
                    for (RestTemplateCustomizer customizer : customizers) {
                        customizer.customize(restTemplate);
                    }
                }
            }
        };
    }


    @Configuration
    
@ConditionalOnMissingClass("org.springframework.retry.support.RetryTemplate")
    static class LoadBalancerInterceptorConfig {
        @Bean
        public LoadBalancerInterceptor ribbonInterceptor(
                LoadBalancerClient loadBalancerClient,
                LoadBalancerRequestFactory requestFactory) {
            return new LoadBalancerInterceptor(loadBalancerClient,
requestFactory);
        }

        @Bean
        @ConditionalOnMissingBean
        public RestTemplateCustomizer restTemplateCustomizer(
                final LoadBalancerInterceptor loadBalancerInterceptor) {
            return new RestTemplateCustomizer() {
                @Override
                public void customize(RestTemplate restTemplate) {
                    List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                            restTemplate.getInterceptors());
                    list.add(loadBalancerInterceptor);
                    restTemplate.setInterceptors(list);
                }
            };
        }
    }
```

```
}
```

在该类中，首先维护了一个被@LoadBalanced修饰的RestTemplate对象的List，在初始化的过程中，通过调用customizer.customize(restTemplate)方法来给RestTemplate增加拦截器LoadBalancerInterceptor。
而LoadBalancerInterceptor，用于实时拦截，在LoadBalancerInterceptor这里实现来负载均衡。LoadBalancerInterceptor的拦截方法如下：

```
@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, requestFactory.createRequest(request, body, execution));
    }
```

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，能实现负载均衡，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。