主讲老师：fox

1.什么是Ribbon

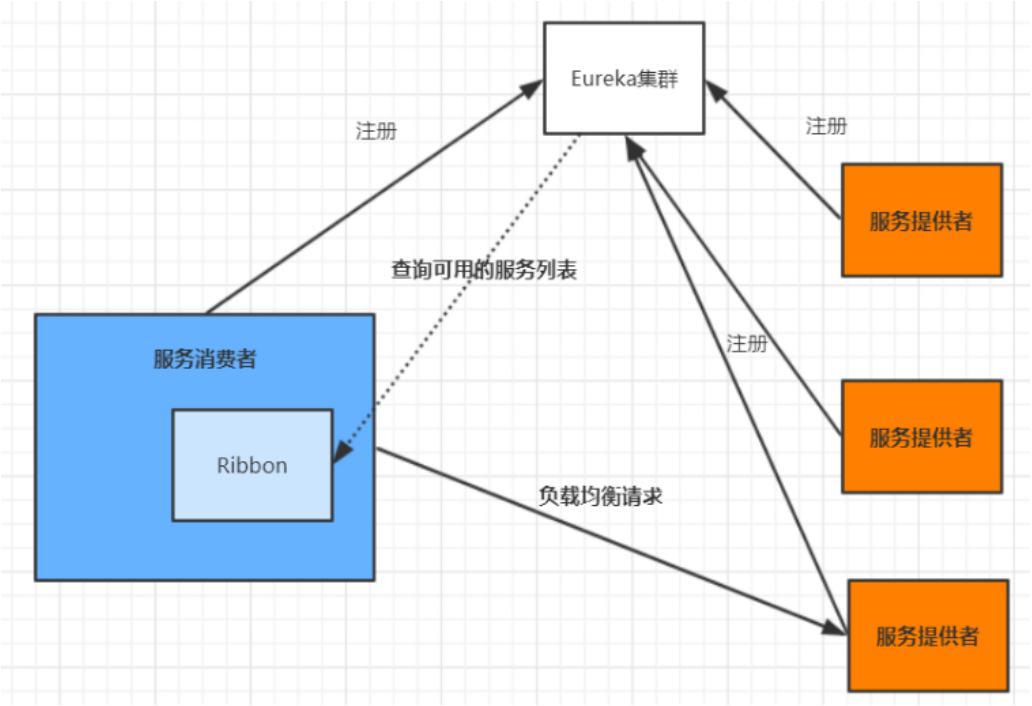
目前主流的负载方案分为以下两种：

- 集中式负载均衡，在消费者和服务提供方中间使用独立的代理方式进行负载，有硬件的（比如F5），也有软件的（比如 Nginx）。
- 客户端根据自己的请求情况做负载均衡，Ribbon 就属于客户端自己做负载均衡。

Spring Cloud Ribbon是基于Netflix Ribbon 实现的一套客户端的负载均衡工具，Ribbon客户端组件提供一系列的完善的配置，如超时，重试等。通过Load Balancer获取到服务提供的所有机器实例，Ribbon会自动基于某种规则(轮询，随机)去调用这些服务。Ribbon也可以实现我们自己的负载均衡算法。

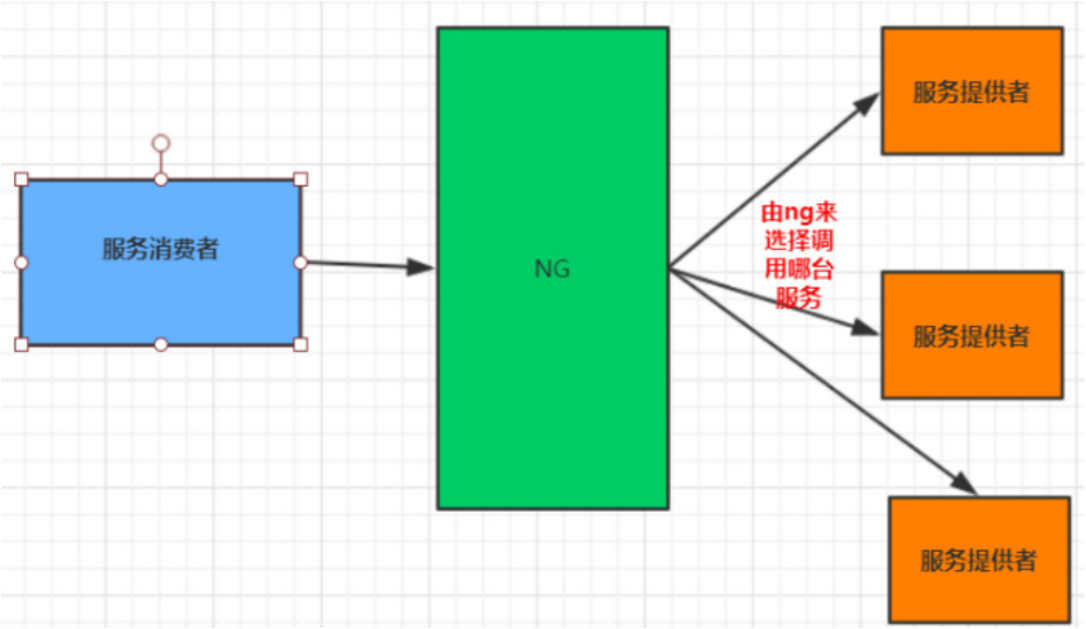
1.1 客户端的负载均衡

例如spring cloud中的ribbon，客户端会有一个服务器地址列表，在发送请求前通过负载均衡算法选择一个服务器，然后进行访问，这是客户端负载均衡；即在客户端就进行负载均衡算法分配。



1.2 服务端的负载均衡

例如Nginx，通过Nginx进行负载均衡，先发送请求，然后通过负载均衡算法，在多个服务器之间选择一个进行访问；即在服务器端再进行负载均衡算法分配。



1.3 常见负载均衡算法

- 随机，通过随机选择服务进行执行，一般这种方式使用较少;
- 轮训，负载均衡默认实现方式，请求来之后排队处理;
- 加权轮训，通过对服务器性能的分型，给高配置，低负载的服务器分配更高的权重，均衡各个服务器的压力;
- 地址Hash，通过客户端请求的地址的HASH值取模映射进行服务器调度。
- 最小链接数，即使请求均衡了，压力不一定会均衡，最小连接数法就是根据服务器的情况，比如请求积压数等参数，将请求分配到当前压力最小的服务器上。

1.4 Ribbon模块

名 称	说 明
ribbon-loadbalancer	负载均衡模块，可独立使用，也可以和别的模块一起使用。
Ribbon	内置的负载均衡算法都实现在其中。
ribbon-eureka	基于 Eureka 封装的模块，能够快速、方便地集成 Eureka。
ribbon-transport	基于 Netty 实现多协议的支持，比如 HTTP、Tcp、Udp 等。
ribbon-httpclient	基于 Apache HttpClient 封装的 REST 客户端，集成了负载均衡模块，可以直接在项目中使用来调用接口。
ribbon-example	Ribbon 使用代码示例，通过这些示例能够让你的学习事半功倍。
ribbon-core	一些比较核心且具有通用性的代码，客户端 API 的一些配置和其他 API 的定义。

1.5 Ribbon使用

编写一个客户端来调用接口

```
public class RibbonDemo {

 public static void main(String[] args) {

```

```
5  // 服务列表
6  List<Server> serverList = Lists.newArrayList(
7  new Server("localhost", 8020),
8  new Server("localhost", 8021));
9  // 构建负载实例
10  ILoadBalancer loadBalancer = LoadBalancerBuilder.newBuilder()
11  .buildFixedServerListLoadBalancer(serverList);
12  // 调用 5 次来测试效果
13  for (int i = 0; i < 5; i++) {
14  String result = LoadBalancerCommand.<String>builder()
15  .withLoadBalancer(loadBalancer).build()
16  .submit(new ServerOperation<String>() {
17  @Override
18  public Observable<String> call(Server server) {
19  String addr = "http://" + server.getHost() + ":" +
20  server.getPort() + "/order/findOrderByUserId/1";
21  System.out.println(" 调用地址: " + addr);
22  URL url = null;
23  try {
24  url = new URL(addr);
25  HttpURLConnection conn = (HttpURLConnection) url.openConnection();
26  conn.setRequestMethod("GET");
27  conn.connect();
28  InputStream in = conn.getInputStream();
29  byte[] data = new byte[in.available()];
30  in.read(data);
31  return Observable.just(new String(data));
32  } catch (Exception e) {
33  e.printStackTrace();
34  }
35  return null;
36  }
37  }).toBlocking().first();
38
39  System.out.println(" 调用结果: " + result);
40  }
41  }
42
43 }
```

上述这个例子主要演示了 Ribbon 如何去做负载操作，调用接口用的最底层的 HttpURLConnection。

```
13:49:26.356 [main] DEBUG com.netflix.loadbalancer.LoadBalancerContext - default using LB returned Server: localhost
 调用地址：http://localhost:8021/order/findOrderByUserId/1
 调用结果：{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}
13:49:34.416 [main] DEBUG com.netflix.loadbalancer.LoadBalancerContext - default using LB returned Server: localhost
 调用地址：http://localhost:8020/order/findOrderByUserId/1
 调用结果：{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}
13:49:42.434 [main] DEBUG com.netflix.loadbalancer.LoadBalancerContext - default using LB returned Server: localhost
 调用地址：http://localhost:8021/order/findOrderByUserId/1
 调用结果：{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}
13:49:50.441 [main] DEBUG com.netflix.loadbalancer.LoadBalancerContext - default using LB returned Server: localhost
 调用地址：http://localhost:8020/order/findOrderByUserId/1
 调用结果：{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}
13:49:58.446 [main] DEBUG com.netflix.loadbalancer.LoadBalancerContext - default using LB returned Server: localhost
 调用地址：http://localhost:8021/order/findOrderByUserId/1
 调用结果：{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}
```

2. Spring Cloud快速整合Ribbon

1) 引入依赖

```
<!--添加ribbon的依赖-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

nacos-discovery依赖了ribbon，可以不用再引入ribbon依赖

- com.alibaba.cloud:spring-cloud-starter-alibaba-nacos-discovery:2.2.1.RELEASE
  - com.alibaba.nacos:nacos-client:1.2.1
  - com.alibaba.spring:spring-context-support:1.0.6
  - org.springframework.cloud:spring-cloud-commons:2.2.2.RELEASE
  - org.springframework.cloud:spring-cloud-context:2.2.2.RELEASE
  - org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.2.RELEASE

2) 添加@LoadBalanced注解

```
@Configuration
public class RestConfig {
 @Bean
 @LoadBalanced
 public RestTemplate restTemplate() {
 return new RestTemplate();
 }
}
```

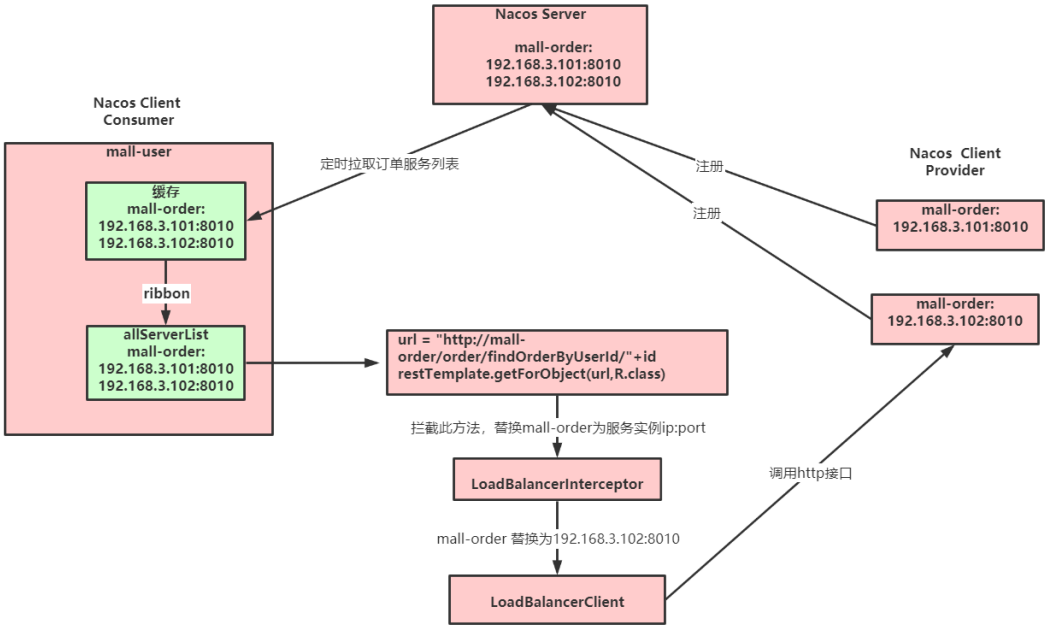
3) 修改controller

```
@Autowired
private RestTemplate restTemplate;

@RequestMapping(value = "/findOrderByUserId/{id}")
public R findOrderByUserId(@PathVariable("id") Integer id) {
 // RestTemplate调用
 //String url = "http://localhost:8020/order/findOrderByUserId/"+id;
 //模拟ribbon实现
 //String url = getUri("mall-order")+"/order/findOrderByUserId/"+id;
  // 添加@LoadBalanced
  String url = "http://mall-order/order/findOrderByUserId/"+id;
```

```
 R result = restTemplate.getForObject(url,R.class);

 return result;
}
```

3. Ribbon内核原理

3.1 Ribbon原理



3.1.1 模拟ribbon实现

```

@Autowired
private RestTemplate restTemplate;

@RequestMapping(value = "/findOrderByUserId/{id}")
public R findOrderByUserId(@PathVariable("id") Integer id) {
 // RestTemplate调用
 //String url = "http://localhost:8020/order/findOrderByUserId/"+id;
 //模拟ribbon实现
  String url = getUri("mall-order")+"/order/findOrderByUserId/"+id;
  // 添加@LoadBalanced
  //String url = "http://mall-order/order/findOrderByUserId/"+id;
  R result = restTemplate.getForObject(url,R.class);
  return result;
 }

 @Autowired
 private DiscoveryClient discoveryClient;
 public String getUri(String serviceName) {
  List<ServiceInstance> serviceInstances = discoveryClient.getInstances(serviceName);
  if (serviceInstances == null || serviceInstances.isEmpty()) {
  return null;
```

```
23  }
24  int serviceSize = serviceInstances.size();
25  //轮询
26  int indexServer = incrementAndGetModulo(serviceSize);
27  return serviceInstances.get(indexServer).getUri().toString();
28 }
29 private AtomicInteger nextIndex = new AtomicInteger(0);
30 private int incrementAndGetModulo(int modulo) {
31  for (;;) {
32  int current = nextIndex.get();
33  int next = (current + 1) % modulo;
34  if (nextIndex.compareAndSet(current, next) && current < modulo){
35  return current;
36  }
37  }
38 }
```

3.1.2 @LoadBalanced 注解原理

参考源码： LoadBalancerAutoConfiguration

@LoadBalanced利用@Qualifier作为restTemplates注入的筛选条件，筛选出具有负载均衡标识的RestTemplate。

```
public class LoadBalancerAutoConfiguration {

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

被@LoadBalanced注解的restTemplate会被定制，添加LoadBalancerInterceptor拦截器。

```
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 restTemplate -> {
            List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                    restTemplate.getInterceptors());
            list.add(loadBalancerInterceptor);
            restTemplate.setInterceptors(list);
        };
    }
```

添加了loadBancerInterceptor拦截器

3.1.3 Ribbon相关接口

参考： **org.springframework.cloud.netflix.ribbon.RibbonClientConfiguration**

IClientConfig：Ribbon的客户端配置，默认采用**DefaultClientConfigImpl**实现。

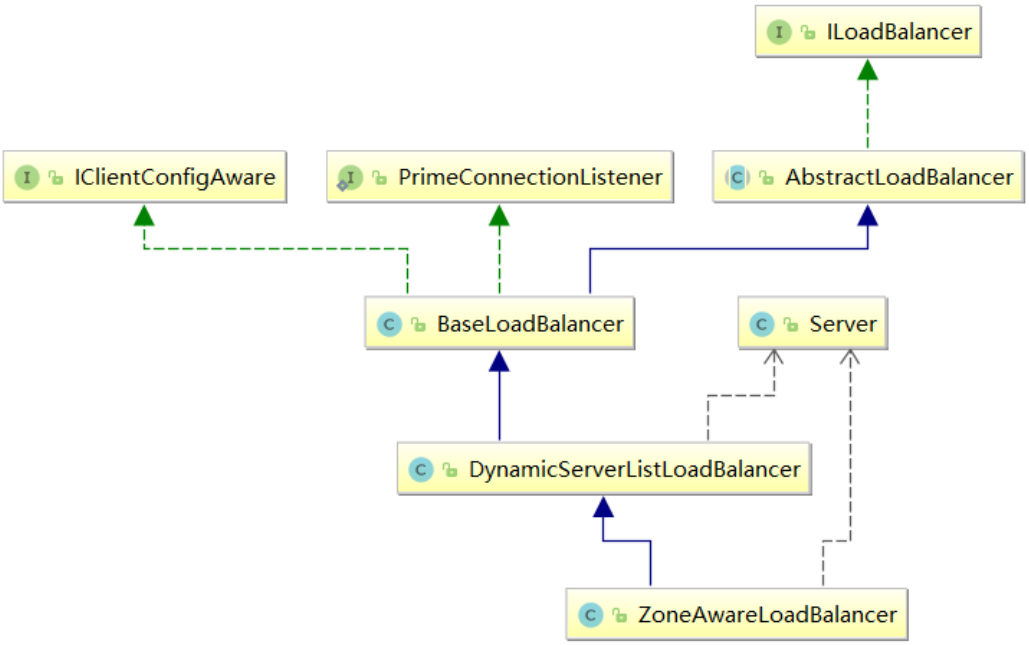
IRule：Ribbon的负载均衡策略，默认采用**ZoneAvoidanceRule**实现，该策略能够在多区域环境下选出最佳区域的实例进行访问。

IPing：Ribbon的实例检查策略，默认采用**DummyPing**实现，该检查策略是一个特殊的实现，实际上它并不会检查实例是否可用，而是始终返回true，默认认为所有服务实例都是可用的。

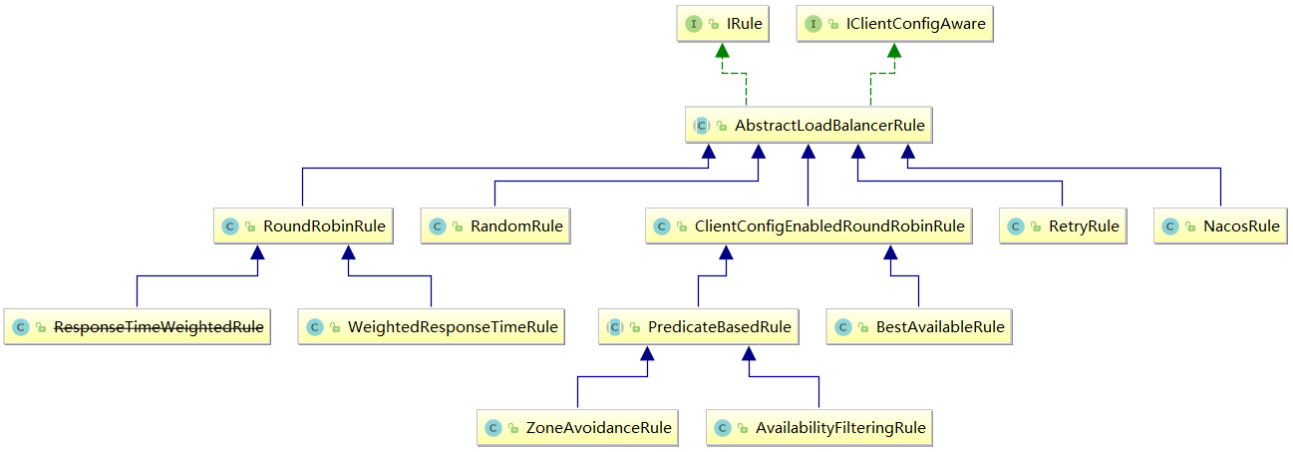
ServerList：服务实例清单的维护机制，默认采用**ConfigurationBasedServerList**实现。

ServerListFilter：服务实例清单过滤机制，默认采**ZonePreferenceServerListFilter**，该策略能够优先过滤出与请求方处于同区域的服务实例。

ILoadBalancer：负载均衡器，默认采用**ZoneAwareLoadBalancer**实现，它具备了区域感知的能力。



3.2 Ribbon负载均衡策略



1. **RandomRule**： 随机选择一个Server。
2. **RetryRule**： 对选定的负载均衡策略机上重试机制，在一个配置时间段内当选择Server不成功，则一直尝试使用subRule的方式选择一个可用的server。
3. **RoundRobinRule**： 轮询选择， 轮询index，选择index对应位置的Server。

4. AvailabilityFilteringRule： 过滤掉一直连接失败的被标记为circuit tripped的后端Server，并过滤掉那些高并发的后端Server或者使用一个AvailabilityPredicate来包含过滤server的逻辑，其实就是检查status里记录的各个Server的运行状态。

5. BestAvailableRule： 选择一个最小的并发请求的Server，逐个考察Server，如果Server被tripped了，则跳过。

6. WeightedResponseTimeRule： 根据响应时间加权，响应时间越长，权重越小，被选中的可能性越低。

7. ZoneAvoidanceRule： 默认的负载均衡策略，即复合判断Server所在区域的性能和Server的可用性选择Server，在没有区域的环境下，类似于轮询(RandomRule)

8. NacosRule: 同集群优先调用

3.2.1 修改默认负载均衡策略

全局配置：调用其他微服务，一律使用指定的负载均衡算法

```
@Configuration
public class RibbonConfig {

 /**
 * 全局配置
 * 指定负载均衡策略
 * @return
 */
 @Bean
  public IRule() {
  // 指定使用Nacos提供的负载均衡策略（优先调用同一集群的实例，基于随机权重）
  return new NacosRule();
  }
}
```

局部配置：调用指定微服务提供的服务时，使用对应的负载均衡算法

修改application.yml

```
# 被调用的微服务名
mall-order:
 ribbon:
 # 指定使用Nacos提供的负载均衡策略（优先调用同一集群的实例，基于随机&权重）
 NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule
```

3.2.2 自定义负载均衡策略

通过实现 IRule 接口可以自定义负载策略，主要的选择服务逻辑在 choose 方法中。

1）实现基于Nacos权重的负载均衡策略

```
@Slf4j
public class NacosRandomWithWeightRule extends AbstractLoadBalancerRule {

 @Autowired
 private NacosDiscoveryProperties nacosDiscoveryProperties;
```

```

@Override
public Server choose(Object key) {
DynamicServerListLoadBalancer loadBalancer = (DynamicServerListLoadBalancer) getLoadBalancer();
String serviceName = loadBalancer.getName();
NamingService namingService = nacosDiscoveryProperties.namingServiceInstance();
try {
//nacos基于权重的算法
Instance instance = namingService.selectOneHealthyInstance(serviceName);
return new NacosServer(instance);
} catch (NacosException e) {
log.error("获取服务实例异常: {}", e.getMessage());
e.printStackTrace();
}
return null;
}
@Override
public void initWithNiwsConfig(IClientConfig clientConfig) {

}
}
```

2) 配置自定义的策略

2.1）局部配置：

修改application.yml

```
# 被调用的微服务名
mall-order:
 ribbon:
 # 自定义的负载均衡策略（基于随机&权重）
 NFLoadBalancerRuleClassName: com.tuling.mall.ribbondemo.rule.NacosRandomWithWeightRule
```

2.2）全局配置

```
@Bean
public IRule ribbonRule() {
 return new NacosRandomWithWeightRule();
}
```

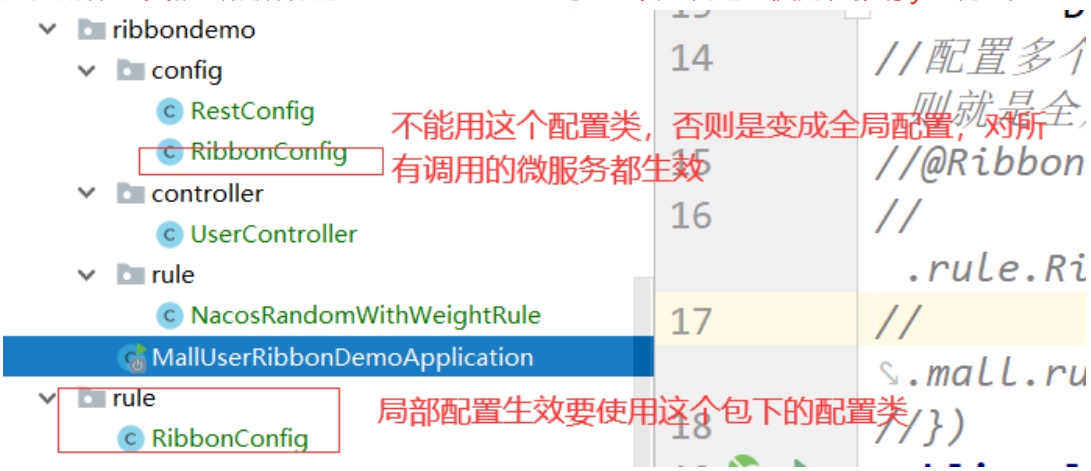
3）局部配置第二种方式

可以利用@RibbonClient指定微服务及其负载均衡策略。

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class,
 DruidDataSourceAutoConfigure.class})
//@RibbonClient(name = "mall-order",configuration = RibbonConfig.class)
//配置多个 RibbonConfig不能被@SpringbootApplication的@CompentScan扫描到，否则就是全局配置的效果
```

```
@RibbonClients(value = {
 // 在SpringBoot主程序扫描的包外定义配置类
 @RibbonClient(name = "mall-order",configuration = RibbonConfig.class),
 @RibbonClient(name = "mall-account",configuration = RibbonConfig.class)
})
public class MallUserRibbonDemoApplication {

 public static void main(String[] args) {
 SpringApplication.run(MallUserRibbonDemoApplication.class, args);
 }
}
```

注意：此处有坑。不能写在@SpringbootApplication注解的@CompentScan扫描得到的地方，否则自定义的配置类就会被所有的 RibbonClients共享。 不建议这么使用，推荐yml方式



Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`, as shown in the following example:

```
@Configuration
@RibbonClient(name = "custom", configuration = CustomConfiguration.class)
public class TestConfiguration {
}
```

In this case, the client is composed from the components already in `RibbonClientConfiguration`, together with any in `CustomConfiguration` (where the latter generally overrides the former).

> The `CustomConfiguration` clas must be a `@Configuration` class, but take care that it is not in a `@ComponentScan` for the main application context. Otherwise, it is shared by all the `@RibbonClients`. If you use `@ComponentScan` (or `@SpringBootApplication`), you need to take steps to avoid it being included (for instance, you can put it in a separate, non-overlapping package or specify the packages to scan explicitly in the `@ComponentScan`).

3.3 饥饿加载

在进行服务调用的时候，如果网络情况不好，第一次调用会超时。

Ribbon默认懒加载，意味着只有在发起调用的时候才会创建客户端。

```
c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
web.servlet.DispatcherServlet        : Completed initialization in 6 ms
tflix.loadbalancer.BaseLoadBalancer  : Client: mall-order instantiated a LoadBalancer: DynamicServerListLoadBalanc
l.DynamicServerListLoadBalancer      : Using serverListUpdater PollingServerListUpdater
l.DynamicServerListLoadBalancer      : DynamicServerListLoadBalancer for client mall-order initialized: DynamicSer
  Total Requests:0;   Successive connection failure:0;    Total blackout seconds:0;   Last connection made:Thu Jan
:0;   Successive connection failure:0;    Total blackout seconds:0;   Last connection made:Thu Jan 01 08:00:00 GMT
:0;   Successive connection failure:0;    Total blackout seconds:0;   Last connection made:Thu Jan 01 08:00:00 GMT
5723d
ibaba.cloud.nacos.ribbon.NacosRule   : A cross-cluster call occurs. name = mall-order, clusterName = DEFAULT, inst
ibaba.cloud.nacos.ribbon.NacosRule   : A cross-cluster call occurs. name = mall-order, clusterName = DEFAULT, inst
```

第一次调用

开启饥饿加载，解决第一次调用慢的问题

```
ribbon:
```

```yaml
  eager-load:
    # 开启ribbon饥饿加载
    enabled: true
    # 配置mall-user使用ribbon饥饿加载，多个使用逗号分隔
    clients: mall-order
```

源码对应属性配置类：RibbonEagerLoadProperties

测试：

```
[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherS
rvlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
rvlet.DispatcherServlet        : Completed initialization in 6 ms
cloud.nacos.ribbon.NacosRule   : A cross-cluster call occurs, name = mall-order, cl
```

第一次调用

文档：02 微服务负载均衡器Ribbon实战.note

链接：http://note.youdao.com/noteshare?id=983c803c0f366af153e5c336aa4ac834&sub=9647D8B130564806A6331428B277036B