主讲老师：Fox

课前须知：

1. 没有接触过微服务的同学，可以先快速的学习一下spring cloud alibaba的基础课程：https://vip.tulingxueyuan.cn/detail/p_60decb8be4b0151fc94c41a4/8?product_id=p_60decb8be4b0151fc94c41a4

2. 微服务专题怎么学

- 第一步：基础比较薄弱的同学，先掌握微服务组件是做什么的，有什么用，怎么用，先系统的把spring cloud&spring cloud alibaba大部分组件学一遍 https://www.processon.com/view/link/60519545f346fb348a97c9d5
- 第二步：深入学习每个组件的设计理念，设计思路，底层原理，扩展应用
- 第三步：掌握spring，spring boot源码（主线流程）前提下，再去看微服务组件源码
- 第四步：综合应用实战

```
有道笔记链接：
笔记文档：02 微服务负载均衡器Ribbon实战.note
链接：http://note.youdao.com/noteshare?id=2a9ae1c1d98031b2d652d8981e4b3501&sub=39C9CD5C3A1842D6A7EF7B2D7757821D
```

1.负载均衡介绍

> 负载均衡（Load Balance），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FTP服务器、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。

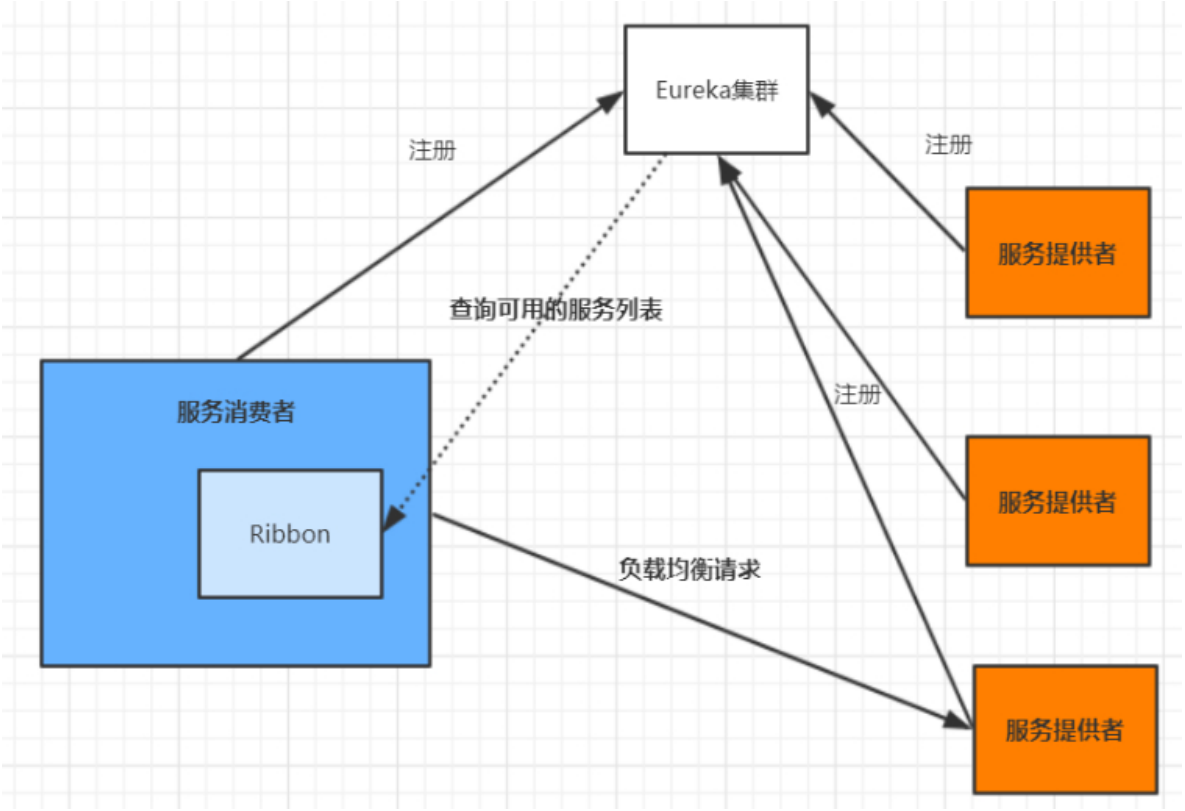
思考： 如果有多个provider实例，consumer应该如何调用呢?

目前主流的负载均衡方案分为以下两种：

- 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。

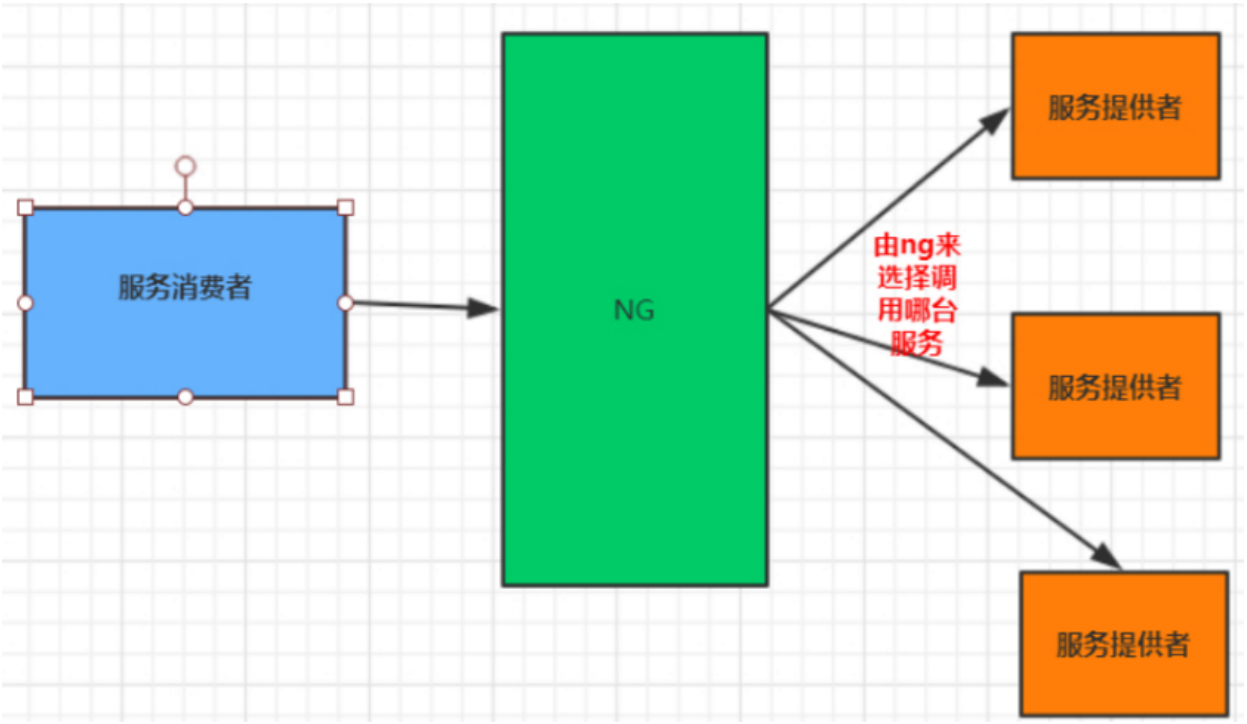
1.1 客户端的负载均衡

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。
- 最小连接数，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。

2. 什么是Ribbon

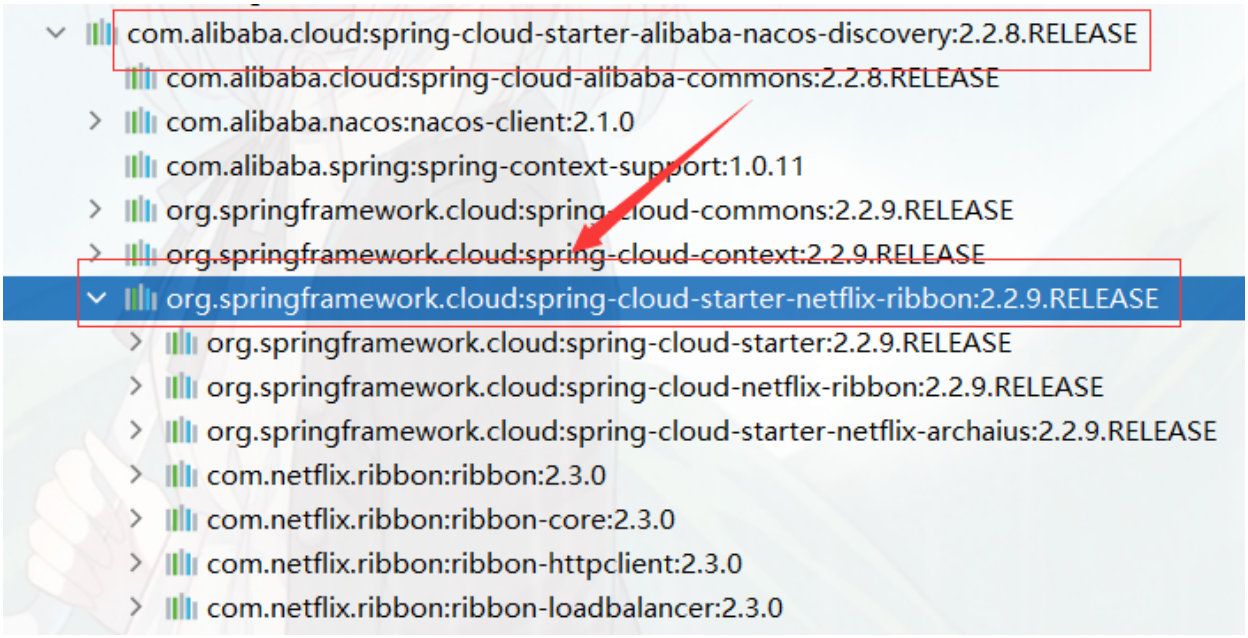
Spring Cloud Ribbon是基于Netflix Ribbon 实现的一套客户端的负载均衡工具，Ribbon客户端组件提供一系列的完善的配置，如超时，重试等。通过Load Balancer获取到服务提供的所有机器实例，Ribbon会自动基于某种规则(轮询，随机)去调用这些服务。Ribbon也可以实现我们自己的负载均衡算法。

2.1 Spring Cloud Alibaba整合Ribbon快速开始

1) 引入ribbon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



2) RestTemplate 添加@LoadBalanced注解，让RestTemplate在请求时拥有客户端负载均衡的能力

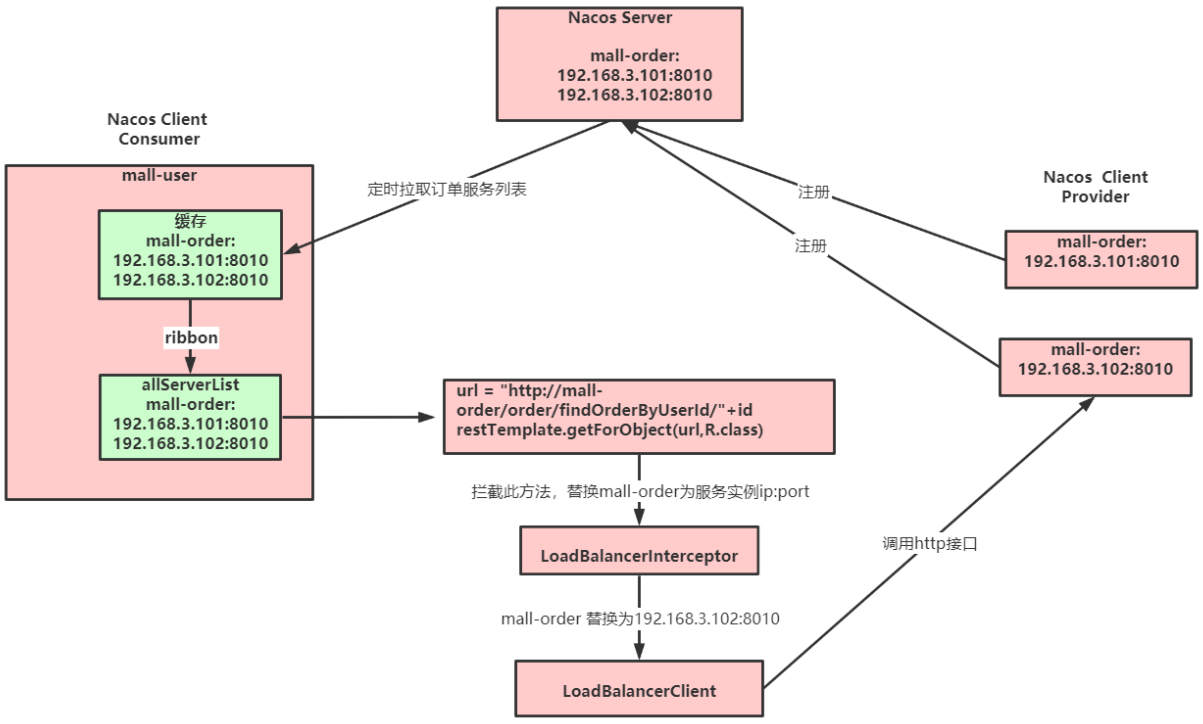
```
@Configuration
public class RestConfig {
```

```
    @Bean
    @LoadBalanced //开启负载均衡
    public RestTemplate restTemplate() {
    return new RestTemplate();
    }
}
```

3) 测试，使用微服务名发起调用

```
@Autowired
private RestTemplate restTemplate;

@RequestMapping(value = "/findOrderByUserId/{id}")
public R findOrderByUserId(@PathVariable("id") Integer id) {
    // ribbon实现，restTemplate需要添加@LoadBalanced注解
    String url = "http://mall-order/order/findOrderByUserId/"+id;
    R result = restTemplate.getForObject(url,R.class);

     return result;
}
```

2.2 Ribbon内核原理



模拟ribbon实现

```

@Autowired
private RestTemplate restTemplate;
```

```
4
5  @RequestMapping(value = "/findOrderByUserId/{id}")
6  public R findOrderByUserId(@PathVariable("id") Integer id) {
7   //模拟ribbon实现
8   String url = getUri("mall-order")+"/order/findOrderByUserId/"+id;
9   R result = restTemplate.getForObject(url,R.class);
10   return result;
11  }
12
13  @Autowired
14  private DiscoveryClient discoveryClient;
15  public String getUri(String serviceName) {
16   List<ServiceInstance> serviceInstances = discoveryClient.getInstances(s
erviceName);
17   if (serviceInstances == null || serviceInstances.isEmpty()) {
18   return null;
19   }
20   int serviceSize = serviceInstances.size();
21   //轮询
22   int indexServer = incrementAndGetModulo(serviceSize);
23   return serviceInstances.get(indexServer).getUri().toString();
24  }
25  private AtomicInteger nextIndex = new AtomicInteger(0);
26  private int incrementAndGetModulo(int modulo) {
27   for (;;) {
28   int current = nextIndex.get();
29   int next = (current + 1) % modulo;
30   if (nextIndex.compareAndSet(current, next) && current < modulo){
31   return current;
32   }
33   }
34  }
```

@LoadBalanced 注解原理

参考源码： LoadBalancerAutoConfiguration。

@LoadBalanced使用了@Qualifier，spring中@Qualifier用于筛选限定注入Bean。

@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。

```java
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。

```java
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

注意： SmartInitializingSingleton是在所有的bean都实例化完成之后才会调用的，所以在bean的实例化期间使用@LoadBalanced修饰的restTemplate是不具备负载均衡作用的。

如果不使用@LoadBalanced注解，也可以通过添加LoadBalancerInterceptor拦截器让restTemplate起到负载均衡器的作用。

```java
@Bean
public RestTemplate restTemplate(LoadBalancerInterceptor loadBalancerInterceptor) {
    RestTemplate restTemplate = new RestTemplate();
    //注入loadBalancerInterceptor拦截器
    restTemplate.setInterceptors(Arrays.asList(loadBalancerInterceptor));
    return restTemplate;
}
```

2.3 Ribbon扩展功能

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。

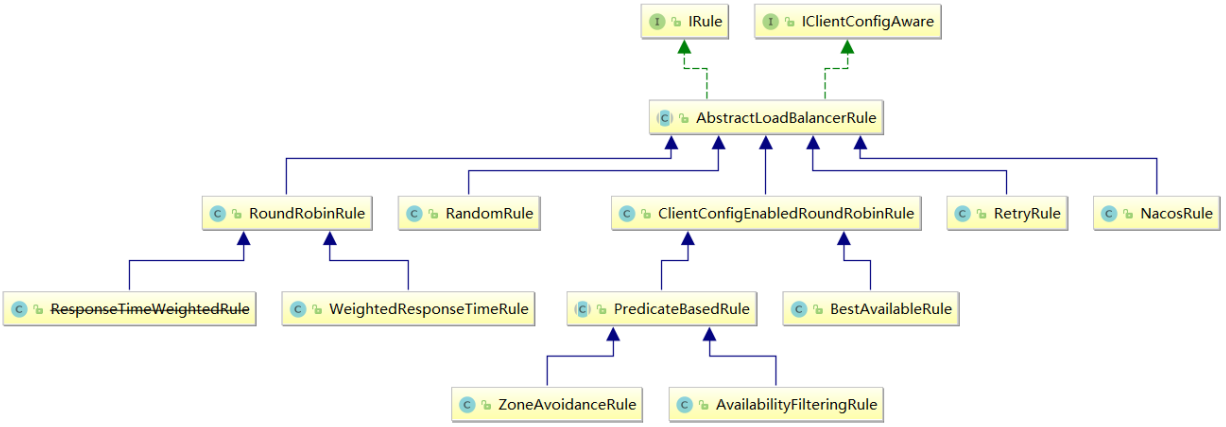
IPing：Ribbon的实例检查策略，默认采用**DummyPing**实现，该检查策略是一个特殊的实现，实际上它并不会检查实例是否可用，而是始终返回true，默认认为所有服务实例都是可用的。

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。

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
8. **NacosRule:** 优先调用同一集群的实例，基于随机权重

修改默认负载均衡策略

全局配置：调用的微服务，一律使用指定的负载均衡策略

```
@Configuration
public class RibbonConfig {

 /**
 * 全局配置
 * 指定负载均衡策略
 * @return
 */
 @Bean
  public IRule ribbonRule() {
  // 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：调用指定微服务时，使用对应的负载均衡策略

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
```

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 = (DynamicServerListLoadBalan
cer) getLoadBalancer();
  String serviceName = loadBalancer.getName();
```

```
11   NamingService namingService = nacosDiscoveryProperties.namingServiceIns
tance();
12   try {
13   //nacos基于权重的算法
14   Instance instance =
namingService.selectOneHealthyInstance(serviceName);
15   return new NacosServer(instance);
16   } catch (NacosException e) {
17   log.error("获取服务实例异常: {}", e.getMessage());
18   e.printStackTrace();
19   }
20   return null;
21   }
22   @Override
23   public void initWithNiwsConfig(IClientConfig clientConfig) {
24
25   }
26  }
```

2) 配置自定义的策略

2.1）全局配置

```
1  @Bean
2  public IRule ribbonRule() {
3   return new NacosRandomWithWeightRule();
4  }
```

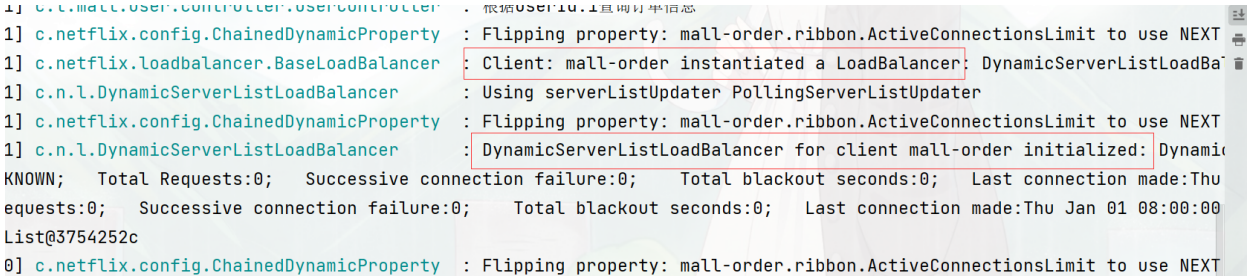
2.2）局部配置：

修改application.yml

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

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。



开启饥饿加载，解决第一次调用慢的问题

```yaml
ribbon:
  eager-load:
    enabled: true
    clients: mall-order
```

参数说明：

- ribbon.eager-load.enabled：开启ribbon的饥饿加载模式
- ribbon.eager-load.clients：指定需要饥饿加载的服务名，也就是你需要调用的服务，如果有多个服务，则用逗号隔开

3. 什么是LoadBalancer

Spring Cloud LoadBalancer是Spring Cloud官方自己提供的客户端负载均衡器, 用来替代Ribbon。

Spring官方提供了两种客户端都可以使用loadbalancer：

RestTemplate

RestTemplate是Spring提供的用于访问Rest服务的客户端，RestTemplate提供了多种便捷访问远程Http服务的方法，能够大大提高客户端的编写效率。默认情况下，RestTemplate默认依赖jdk的HTTP连接工具。

WebClient

WebClient是从Spring WebFlux 5.0版本开始提供的一个非阻塞的基于响应式编程的进行Http请求的客户端工具。它的响应式编程的基于Reactor的。WebClient中提供了标准Http请求方式对应的get、post、put、delete等方法，可以用来发起相应的请求。

3.1 RestTemplate整合LoadBalancer

1）引入依赖

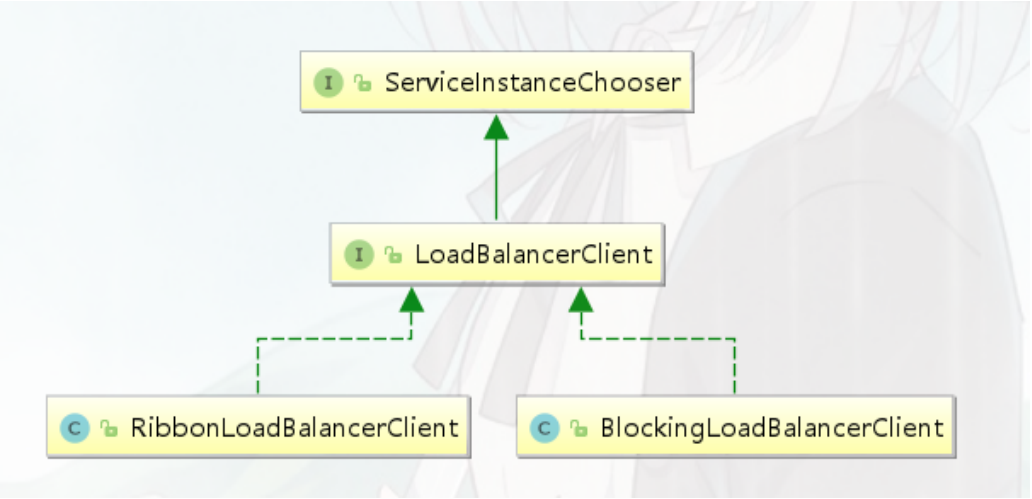
```xml
<!-- LoadBalancer -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

```
6
7  <!-- 提供了RestTemplate支持 -->
8  <dependency>
9   <groupId>org.springframework.boot</groupId>
10   <artifactId>spring-boot-starter-web</artifactId>
11  </dependency>
12
13  <!-- nacos服务注册与发现 移除ribbon支持-->
14  <dependency>
15   <groupId>com.alibaba.cloud</groupId>
16   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
17   <exclusions>
18   <exclusion>
19   <groupId>org.springframework.cloud</groupId>
20   <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
21   </exclusion>
22   </exclusions>
23  </dependency>
```

注意： nacos-discovery中引入了ribbon，需要移除ribbon的包

如果不移除，也可以在yml中配置不使用ribbon。默认情况下，如果同时拥有RibbonLoadBalancerClient和BlockingLoadBalancerClient，为了保持向后兼容性，将使用RibbonLoadBalancerClient。要覆盖它，可以设置spring.cloud.loadbalancer.ribbon.enabled属性为false。

```
1  spring:
2   application:
3   name: mall-user-loadbalancer-demo
4   cloud:
5   nacos:
6   discovery:
7   server-addr: 127.0.0.1:8848
8   # 不使用ribbon，使用loadbalancer
9   loadbalancer:
10   ribbon:
11   enabled: false
```



2）使用@LoadBalanced注解修饰RestTemplate，开启客户端负载均衡功能

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

```

3) 使用

```
@RestController
@RequestMapping("/user")
public class UserController {

 @Autowired
 private RestTemplate restTemplate;

 @RequestMapping(value = "/findOrderByUserId/{id}")
 public R findOrderByUserId(@PathVariable("id") Integer id) {
  String url = "http://mall-order/order/findOrderByUserId/"+id;
  R result = restTemplate.getForObject(url,R.class);
  return result;
 }
}
```

3.2 WebClient整合LoadBalancer

1）引入依赖

```xml
<!-- LoadBalancer -->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>

<!-- webflux -->
<dependency>
 <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
 </dependency>

 <!-- nacos服务注册与发现 -->
 <dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
  <exclusions>
  <exclusion>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
  </exclusion>
  </exclusions>
 </dependency>
```

2) 配置WebClient作为负载均衡器的client

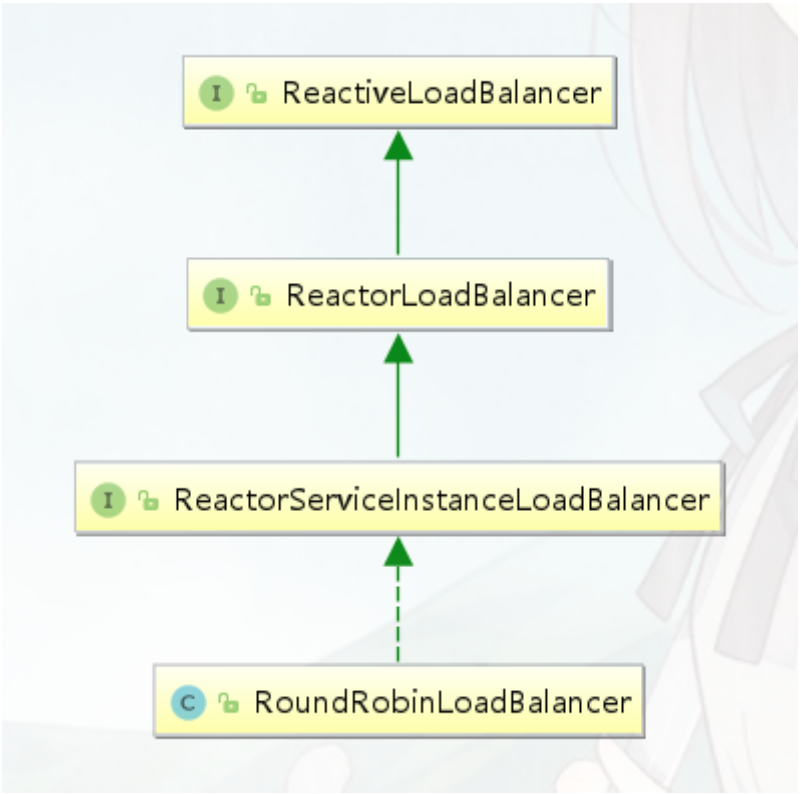
```java
@Configuration
public class WebClientConfig {

 @LoadBalanced
 @Bean
 WebClient.Builder webClientBuilder() {
 return WebClient.builder();
 }

  @Bean
  WebClient webClient() {
  return webClientBuilder().build();
  }
 }
```

3) 使用

```
@Autowired
```

```
2  private WebClient webClient;
3
4  @RequestMapping(value = "/findOrderByUserId2/{id}")
5  public Mono<R> findOrderByUserIdWithWebClient(@PathVariable("id") Integer
id) {
6
7   String url = "http://mall-order/order/findOrderByUserId/"+id;
8   //基于WebClient
9   Mono<R> result = webClient.get().uri(url)
10   .retrieve().bodyToMono(R.class);
11   return result;
12  }
```

原理： 底层会使用ReactiveLoadBalancer



引入webFlux

```
1  @Autowired
2  private ReactorLoadBalancerExchangeFilterFunction lbFunction;
3
4  @RequestMapping(value = "/findOrderByUserId3/{id}")
5  public Mono<R> findOrderByUserIdWithWebFlux(@PathVariable("id") Integer i
d) {
6
7   String url = "http://mall-order/order/findOrderByUserId/"+id;
8   //基于WebClient+webFlux
9   Mono<R> result = WebClient.builder()
```

```
10   .filter(lbFunction)
11   .build()
12   .get()
13   .uri(url)
14   .retrieve()
15   .bodyToMono(R.class);
16  return result;
17 }
```