13 服务请求负载均衡

更新时间：2019-06-19 17:56:16

> 如果不想在世界上虚度一生，那就要学习一辈子。
>
> ——高尔基

通过前面文章的学习，大家已经了解到如何搭建服务注册中心，如何将一个 provider 注册到服务注册中心，consumer 又如何从服务注册中心获取到 provider 的地址，在 consumer 获取 provider 地址时，我们一直采用了 DiscoveryClient 来手动获取，这样出现了大量冗余代码，而且负载均衡功能也没能实现。因此，本文我将和大家分享在微服务中如何实现负载均衡，以及负载均衡的实现原理、常见的负载均衡策略等。

准备工作

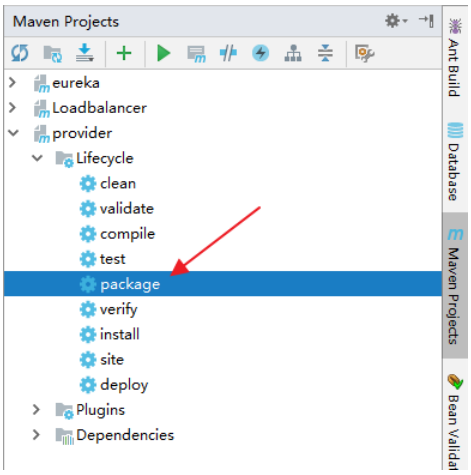
参考前面的文章，我们首先创建一个名为 Loadbalancer 的父工程，然后在父工程中创建一个名为 eureka 的服务注册中心，再创建一个名为 provider 的微服务， provider 中提供一个 `/hello` 接口， provider 中的 `/hello` 接口如下：

```
@RestController
public class HelloController {
    @Value("${server.port}")
    Integer port;

    @GetMapping("/hello")
    public String hello(String name) {
        return "hello " + name + " ; " + port;
    }
}
```

注意这里的 `/hello` 接口和前面我们学习的 `/hello` 接口有一点点不一样，这里我将当前应用运行的端口号注入进来，并且在接口中将端口号返回，这样主要是为了当大家从 consumer 中调用 provider 时，能够知道到底是哪个 provider 提供了服务。最后再在 provider 的 application.properties 文件中添加配置，将 provider 注册到服务注册中心上，如下：

```
spring.application.name=provider
server.port=4001
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

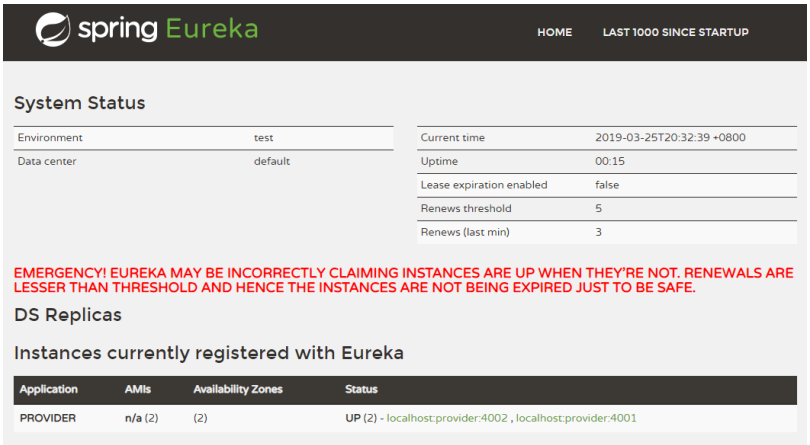
完成之后，首先启动 eureka，然后在 IntelliJ IDEA 的右侧找到 Maven Project 选项，再找到 provider 项目，对 provider 项目进行打包，如下：



打包成功后，在 provider/target 目录下会生成一个名为 provider-0.0.1-SNAPSHOT.jar 的 jar 包，然后打开两个命令行窗口，定位到 provider/target 目录下，分别执行如下两行命令：

```
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=4001
java -jar provider-0.0.1-SNAPSHOT.jar --server.port=4002
```

这两个命令各自在一个窗口中执行，执行完成后，将启动两个 provider 实例，两个实例的端口分别是 4001 和 4002，此时打开 eureka 控制面板，可以看到两个 provider 实例，如下：



如此之后，我们的准备工作就算做完了。

手动实现负载均衡

现在，再创建一个 consumer ，并将之注册到 eureka 上，这个具体过程我这里就不再重复，不记得的同学可以参考本章前面两篇文章。

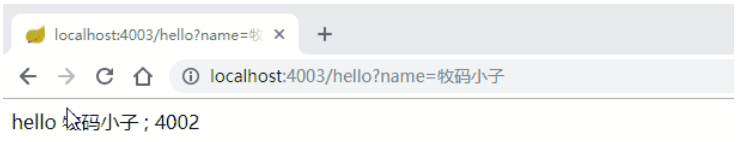
consumer 创建成功后，在 consumer 中创建一个 UseHelloController 的类，里边创建一个 `/hello` 接口，然后利用 DiscoveryClient 。我们先手动实现一个简单的基于轮询策略的负载均衡效果，具体代码如下：

```
@RestController
public class UseHelloController {
    @Autowired
    DiscoveryClient discoveryClient;
    @Autowired
    RestTemplate restTemplate;
    int count = 0;

    @GetMapping("/hello")
    public String hello(String name) {
        List<ServiceInstance> list = discoveryClient.getInstances("provider");
        ServiceInstance instance = list.get(count % list.size());
        count++;
        String host = instance.getHost();
        int port = instance.getPort();
        String s = restTemplate.getForObject("http://" + host + ":" + port + "/hello?name={1}", String.class, name);
        return s;
    }
}
```

这里一个简单的轮询策略的负载均衡实现思路如下：

1. 由于 DiscoveryClient 可以从 eureka 上获取到 provider 的所有实例，这些获取到的所有实例保存在一个 List 集合中，因此所谓的轮询，实际上就是从 List 集合中循环取出 ServiceInstance 实例，组装成相关的地址去运行；
2. 基于第一步的分析，我们创建一个全局变量 count ，每次将集合大小与 count 取模，以取模的结果为下标，从 List 集合中取出相关的实例，然后拼接出请求地址。

这个过程其实很简单，轮询的逻辑也很简单，该方法执行结果如下图：



可以看到，服务是由端口为 4001 的 provider 和端口为 4002 的 provider 来轮流提供的。有没有发现负载均衡其实很简单！

使用 @Loadbalancer 实现负载均衡

上面是一个自己手动实现的负载均衡，代码量虽然不大，但是如果每次请求都这么写，实际上还是很头大，因为充斥着大量的代码冗余，此时，有人可能会想到上篇文章我们在讲解 RestTemplate 时提到的拦截器，如果能够在拦截器中将请求地址拦截下来，然后自动进行负载均衡计算，进而使用一个合适的地址去发送请求的话，就会方便很多。没错，实际上在 Spring Cloud 中也是这么做的！

在 Spring Cloud 中，要实现负载均衡其实非常容易，只需要在 RestTemplate 的Bean上添加一个 @LoadBalanced 注解即可，如下：

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

此时的 RestTemplate 就是一个具备负载均衡功能的 RestTemplate 了，接下来，在 UseHelloController 中继续添加如下代码：

```
@Autowired
@Qualifier("loadBalancer")
RestTemplate loadBalancer;
@GetMapping("/hello2")
public String hello2(String name) {
    String s = loadBalancer.getForObject("http://provider/hello?name={1}", String.class, name);
    return s;
}
```

是不是一下清爽了很多？关于这段代码，解释如下：

1. 由于现在项目中存在两个 RestTemplate 的实例，因此这里我加了 @Qualifier(“loadBalancer”) 注解，表示通过名称来查找 RestTemplate 的实例（如果继续根据类型来查找，系统会不知道到底注入哪个实例）；
2. 此时注入进来的 RestTemplate 实例自动就具备了负载均衡功能，但需要注意的是这里的请求地址，原本的 `Host+":"+Port` 被微服务名称所替代，实际上这个很好理解，因为这里如果还继续明确直接指定了服务地址的话，那还怎么负载均衡呀？
3. 这里使用了 provider 来代替 `Host+":"+Port` ，在真正的请求发起时，会通过拦截器将请求拦截下来，然后将 provider 换成一个具体的服务地址。

使用了具备负载均衡功能的 RestTemplate 之后，当我们再次启动 consumer ，发起一个网络访问时，可以在 IntelliJ IDEA 的控制台看到如下日志：

可以看到， consumer 会自动从 eureka 上根据 provider 获取 provider 所有实例的信息，不仅仅包括实例的地址信息，也包括实例的历史请求数据等信息，根据这些辅助数据，可以实现不同的负载均衡策略。

请求失败重试

具备了负载均衡功能的 RestTemplate 也可以开启请求重试功能。有的时候，在微服务调用的过程中，由于网络抖动等原因造成了访问失败，这个时候如果直接就认定访问失败显然是不划算的，可以多尝试几次，多次尝试之后，如果还是请求失败，再判定访问失败。默认情况下，重试功能是没有开启的，开启重试功能很简单，开发者只需要在 consumer 中添加 Spring Retry 依赖即可，如下：

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

只要添加了该依赖，我们的 RestTemplate 此时就自动具备了请求失败重试的功能（注意，加入依赖后请求失败重试功能就会自动开启了），如果开发者加入了该依赖，但是又不想开启请求失败重试功能，可以在 application.properties中添加如下配置：

```
spring.cloud.loadbalancer.retry.enabled=false
```

表示关闭请求失败重试功能。至于请求失败重试的次数以及切换实例的次数，可以通过如下配置实现：

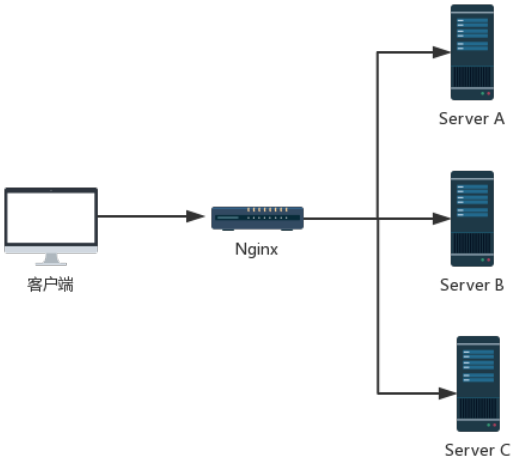
```
# 最大的重试次数，不包括第一次请求
ribbon.MaxAutoRetries=3
# 最大重试server的个数，不包括第一个 server
ribbon.MaxAutoRetriesNextServer=1
```

另外也可以指定是否开启任何异常都重试：

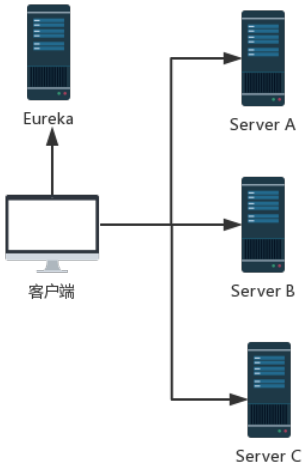
```
ribbon.OkToRetryOnAllOperations=true
```

客户端负载均衡

细心的同学可能发现，这里我们所说的负载均衡和大家平时所了解到的负载均衡不太一样，平时我们所说的负载均衡一般是指 Nginx 或者 F5 之类的工具，其中 Nginx也叫反向代理代理服务器，它的工作流程如下图：



所有的请求首先到达负载均衡服务器，再由负载均衡服务器根据提前配置好的负载均衡策略，将请求转发到不同的 Real Server 上，对于客户端来说，它并不知道究竟是哪一个 Real Server 提供的服务，这种负载均衡方式我们一般称之为服务端负载均衡。而上文我们提到的负载均衡则与这种方式不同，上文的负载均衡是先将 provider 的所有地址拿到，然后 consumer 根据本地配置的负载均衡策略，从 provider 地址列表中挑选一个地址去调用，调用过程如下图：



在这个调用过程中，客户端首先去 Eureka 中获取 provider 地址，获取到 provider 地址列表之后，再根据本地提前配置好的负载均衡策略从地址列表中挑选一个地址去调用，这个过程没有中间代理服务器，到底调用哪一个服务是由 consumer 自己决定的，因此这种负载均衡我们也称之为客户端负载均衡。

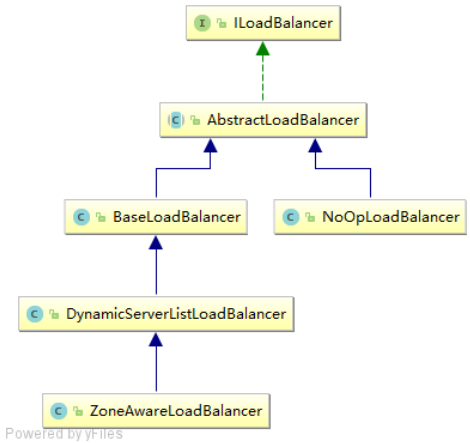
常见负载均衡策略

通过上面的学习，大家了解到，默认的负载均衡策略实际上是轮询，那么除了这种负载均衡策略之外，还有哪些负载均衡策略呢？又是如何配置的呢？我们继续来看。

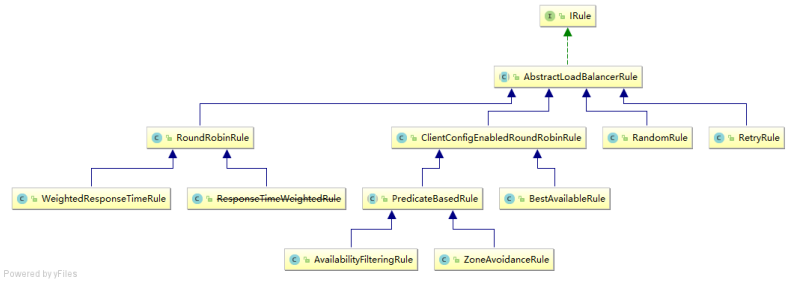
要说负载均衡策略，就需要先向大家介绍一个接口叫做 ILoadBalancer ，它的源码如下：

```
public interface ILoadBalancer {
 public void addServers(List<Server> newServers);
 public Server chooseServer(Object key);
 public void markServerDown(Server server);
 @Deprecated
 public List<Server> getServerList(boolean availableOnly);
    public List<Server> getReachableServers();
 public List<Server> getAllServers();
}
```

从方法名大致就可以看出，这里有服务的添加、选择、标记服务下线、获取服务列表等功能，这个接口的实现类如下：



在它的实现类 BaseLoadBalancer 中，去向 Eureka 获取服务列表，并不断地通过心跳消息去检查服务是否可用，有了服务列表之后，再根据具体的 IRule 进行负载均衡，所以，我们再来关注下 IRule ，这也是一个接口，它的实现类如下：



这里一个实现类就代表了一个负载均衡实现策略，例如：

- RandomRule 表示随机策略
- RoundRobinRule 表示轮询策略
- WeightedResponseTimeRule 表示加权策略（即将过期，功能和 ResponseTimeWeightedRule 一致）
- ResponseTimeWeightedRule 也是加权，它是根据每一个 Server 的平均响应时间动态加权，响应时间越长，权重越小，处理请求的机会也越小
- RetryRule 表示一个具备重试功能的负载均衡策略，内部默认使用了 RoundRobinRule 这个内部也可以自己传其他的负载均衡策略进去

- BestAvailableRule 策略表示使用并发数最小的服务

那么这些不同的负载均衡策略要如何去配置呢？很简单，需要哪种负载均衡，就提供哪种负载均衡的 IRule 实例就行了。例如，需要使用随机策略，那么只需要在 consumer 的配置类中提供一个 RandomRule 的实例即可，如下：

```
@Bean
IRule iRule() {
    return new RandomRule();
}
```

此时，多次访问 provider 接口，就会发现负载均衡策略已经变为随机策略了。

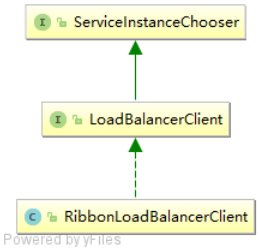
负载均衡原理

网上关于负载均衡策略源码分析的文章很多，但是大多数都看得初学者丈二和尚摸不着头脑，因为这里确实涉及到的类比较多，因此本文换一个思路来和大家讲这个原理。关于负载均衡，大家可能会比较感兴趣服务列表到底是什么时候通过什么方式加载的？为什么在 RestTemplate 访问中不需要写服务的具体地址，而只需要给一个服务名就行了？所有的配置又是如何切入到 RestTemplate 中的？接下来，带着这些问题，我来和大家分析负载均衡具体的实现逻辑。

服务列表加载问题

首先第一个问题就是服务列表的加载问题。在前面的文章中，我们都是自己通过 DiscoveryClient 手动获取服务地址列表的，自从用了 @LoadBalanced 注解之后，我们不再需要自己手动加载地址了，直接写服务名就可以了，不用看源码我们也知道，服务名最终肯定要被转为一个具体的地址才能使用，那么这个过程到底是在哪里呢？

通过查看 @LoadBalanced 的源码我们发现凡是加了该注解的 RestTemplate 都会被自动配置一个 LoadBalancerClient，LoadBalancerClient 只是一个接口，这个接口继承自 ServiceInstanceChooser 并且只有一个实现类 RibbonLoadBalancerClient，如下图：



其中，ServiceInstanceChooser 里边只定义了一个方法，如下：

```
public interface ServiceInstanceChooser {
 ServiceInstance choose(String serviceId);
}
```

看名字就知道，这个方法是用来根据 serviceId 获取 ServiceInstance 的，ServiceInstance 中则保存了一个服务的详细信息，LoadBalancerClient 接口在此基础上又增加了三个方法，如下：

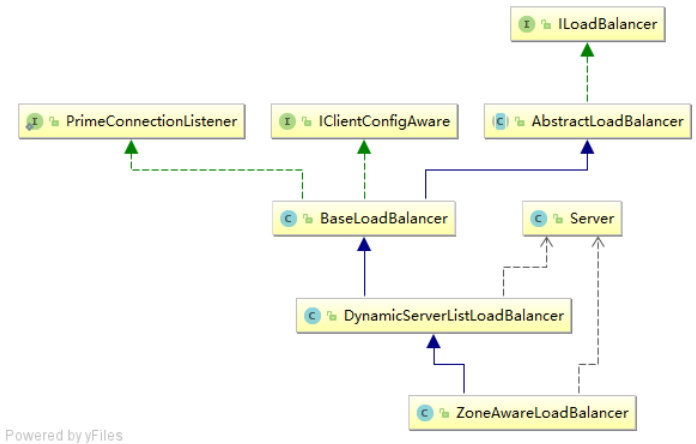
```
public interface LoadBalancerClient extends ServiceInstanceChooser {

 <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);

}
```

这里方法有三个， execute 方法用来做执行操作， reconstructURI 方法则用来重构 Url，也就是把 `http://provider/hello` 这样的地址变为 `http://localhost:4001/hello` 这样的形式。

这里只是一个接口，具体的实现在 RibbonLoadBalancerClient 类中。在该类中，首先来看服务的选择问题，choose 方法经过一系列的跳转最终来到了这里：

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

在这个方法中，首先使用 getServer 方法去获取服务，这个方法最终会来到 ILoadBalancer 接口的 chooseServer 方法中， ILoadBalancer 接口的继承关系如下图：



默认情况下，在 RibbonLoadBalancerClient 类中注入的 ILoadBalancer 的实例是 ZoneAwareLoadBalancer ，但是 ZoneAwareLoadBalancer 继承自 DynamicServerListLoadBalancer ，而在 DynamicServerListLoadBalancer 的构造方法中，调用了 `restOfInit(clientConfig);` 方法，该方法源码如下：

```
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
}
```

在这个方法中，通过 updateListOfServers(); 去加载服务列表，而该方法会调用到 ServerList 接口中的 getUpdatedListOfServers 方法， ServerList 是一个接口，这个接口中定义了获取所有注册微服务信息的方法，它的最终实现类是 DomainExtractingServerList 。在 DomainExtractingServerList 中，我们终于看到了通过 EurekaClient 去获取服务列表的代码了，如下：

```
    private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
        EurekaClient eurekaClient = eurekaClientProvider.get();
        if (vipAddresses!=null){
            for (String vipAddress : vipAddresses.split(",")) {
                List<InstanceInfo> listOfInstanceInfo = eurekaClient.getInstancesByVipAddress(vipAddress, isSecure, targetR
egion);
                for (InstanceInfo ii : listOfInstanceInfo) {
                    //省略
                }
            }
        }
        return serverList;
    }
```

毫无疑问，这里最终会来到 DiscoveryClient 中获取服务地址列表。至此，大家应该知道了服务列表是在哪里加载了吧！

请求地址替换问题

那么请求地址又是怎么样从 `http://provider/hello` 变为 `http://localhost:4001/hello` 呢？上文我们提到了在 LoadBalancerClient 类中有一个方法叫做 reconstructURI ，这个方法就是用来重构 Url ，它的具体实现在 LoadBalancerContext 类的 reconstructURIWithServer 方法中，具体代码如下：

```
public URI reconstructURIWithServer(Server server, URI original) {
    String host = server.getHost();
    int port = server.getPort();
    String scheme = server.getScheme();

    if (host.equals(original.getHost())
            && port == original.getPort()
            && scheme == original.getScheme()) {
        return original;
    }
    if (scheme == null) {
        scheme = original.getScheme();
    }
    if (scheme == null) {
        scheme = deriveSchemeAndPortFromPartialUri(original).first();
    }
    try {
        StringBuilder sb = new StringBuilder();
        sb.append(scheme).append("://");
        if (!Strings.isNullOrEmpty(original.getRawUserInfo())) {
            sb.append(original.getRawUserInfo()).append("@");
        }
        sb.append(host);
        if (port >= 0) {
            sb.append(":").append(port);
        }
        sb.append(original.getRawPath());
        if (!Strings.isNullOrEmpty(original.getRawQuery())) {
            sb.append("?").append(original.getRawQuery());
        }
        if (!Strings.isNullOrEmpty(original.getRawFragment())) {
            sb.append("#").append(original.getRawFragment());
        }
        URI newURI = new URI(sb.toString());
        return newURI;
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
}
```

方法参数 Server 中包含了服务的基本信息，original 则是待转换的 Url ，这里的逻辑整体比较好理解，就是将 Server 中的数据拿出来重新拼接地址。

如何切入到 **RestTemplate** 中

那么上面这些东西又是如何切入到 RestTemplate 中的呢？这里需要大家回忆一下上篇文章我们在介绍 RestTemplate 时讲过的拦截器，实际上，这里的功能也都是通过拦截器嵌入到 RestTemplate 的执行中的。

当我们的 classpath 下存在 RestTemplate ，并且项目中存在 LoadBalancerClient 的实例时，在 LoadBalancerAutoConfiguration 类中就会启动一个自动配置，部分源码如下：

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

  @Bean
  @ConditionalOnMissingBean
  public RetryLoadBalancerInterceptor ribbonInterceptor(
    LoadBalancerClient loadBalancerClient,
    LoadBalancerRetryProperties properties,
    LoadBalancerRequestFactory requestFactory,
    LoadBalancedRetryFactory loadBalancedRetryFactory) {
   return new RetryLoadBalancerInterceptor(loadBalancerClient, properties,
     requestFactory, loadBalancedRetryFactory);
  }

  @Bean
  @ConditionalOnMissingBean
  public RestTemplateCustomizer restTemplateCustomizer(
    final RetryLoadBalancerInterceptor loadBalancerInterceptor) {
   return restTemplate -> {
    List<ClientHttpRequestInterceptor> list = new ArrayList<>(
      restTemplate.getInterceptors());
    list.add(loadBalancerInterceptor);
    restTemplate.setInterceptors(list);
   };
  }

 }
}
```

可以看到，在这里，自动向 RestTemplate 中加了一个拦截器 RetryLoadBalancerInterceptor ，正是这个拦截器将上面提到的所有功能给集成进来了，有兴趣的读者可以了解下具体的拦截过程，在 RetryLoadBalancerInterceptor 类的 intercept 方法中，通过 LoadBalancerClient 的实例进行了服务获取、负载均衡以及请求 Url 重构，该方法的逻辑就比较简单了，这里就不再赘述。

小结

本文首先带领读者手动实现了一个简易的负载均衡功能，然后向读者介绍了 RestTemplate 如何在框架中实现负载均衡以及请求失败重试等配置，并带着读者大致过了一下源码，实际上这里的源码并不复杂，抓住主线，然后通过 Debug 的方式很容易将思路理清。由于集群化部署是微服务一个重要的特点，因此，负载均衡策略以及配置方式大家一定要掌握。

本文作者：纯洁的微笑、江南一点雨

12 RestTemplate 用法详解

14 声明式服务调用 Feign