简介

Spring cloud ribbon在spring cloud微服务体系中充当着负载均衡的角色。这个负载均衡指的是客户端的负载均衡。本文是ribbon源码分析系列的第一篇，主要内容如下：

* 怎样使用spring cloud ribbon
* ribbon原理概览

怎样使用Spring cloud ribbon

我们知道ribbon是客户端负载均衡，也就是说在相同的服务集群中选择一个，然后进行访问，并从该服务获取到结果。这里面会引申出一个问题，就是相同服务集群的来源。ribbon有两种方式获取，第一种是通过Eureka(注册中心)，这种方式需要使用ribbon的工程是一个Eureka Client也就是说需要在工程的主函数上使用([@EnableDiscoveryClient](https://github.com/EnableDiscoveryClient)),第二种方式是通过properties进行配置。  
本文主要介绍的是第二种。  
下面结合一个例子来说明：

添加对应依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

定义配置类

@Configuration

public class RibbonConfig {

@LoadBalanced

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

如上图所示在该配置类中创建RestTemplate，并且使用[@LoadBalanced](https://github.com/LoadBalanced" \o "@LoadBalanced)注解。该注解使得RestTemplate具有了客户端负载均衡的能力。

properties文件

spring.application.name=ribbon-client

users.ribbon.listOfServers=http://localhost:8081,http://localhost:8082

users.ribbon.listOfServers这个参数很关键，它的含义是指定服务(集群)的地址，其中users是自定义的Key。本文中有两个相同的服务，它们的地址分别为http://localhost:8081以及http://localhost:8082

定义一个Controller(Ribbon-Client端)

@RestController

public class DemoController {

private static final String URL = "http://users/hello";

@Autowired

private RestTemplate restTemplate;

@RequestMapping(value = "/ribbon")

public String ribbon() {

return this.restTemplate.getForObject(DemoController.URL, String.class);

}

}

后端Server代码(8081、8082)

@RequestMapping(value = "demo")

public String demo() {

return "this is 8081 server...";

}

@RequestMapping(value = "demo")

public String demo() {

return "this is 8082 server...";

}

此时当我们访问[http://localhost:8080/ribbon并且不断刷新浏览器(多次访问该接口)，我们可以看到`http://localhost:8081/hello`、`http://localhost:8082/hello`这两个接口反复被调用。(交替返回`this](http://localhost:8080/ribbon%E5%B9%B6%E4%B8%94%E4%B8%8D%E6%96%AD%E5%88%B7%E6%96%B0%E6%B5%8F%E8%A7%88%E5%99%A8(%E5%A4%9A%E6%AC%A1%E8%AE%BF%E9%97%AE%E8%AF%A5%E6%8E%A5%E5%8F%A3)%EF%BC%8C%E6%88%91%E4%BB%AC%E5%8F%AF%E4%BB%A5%E7%9C%8B%E5%88%B0%60http:/localhost:8081/hello%60%E3%80%81%60http:/localhost:8082/hello%60%E8%BF%99%E4%B8%A4%E4%B8%AA%E6%8E%A5%E5%8F%A3%E5%8F%8D%E5%A4%8D%E8%A2%AB%E8%B0%83%E7%94%A8%E3%80%82(%E4%BA%A4%E6%9B%BF%E8%BF%94%E5%9B%9E%60this) is 8081 server…和this is 8082 server…`)  
至此通过这个例子我们完成了使用ribbon来完成客户端负载均衡的功能，接下来通过源码了解下其中的原理。

Ribbon原理概览

通过源码分析，个人认为可以拆解为如下部分：

* 1.获取[@LoadBalanced](https://github.com/LoadBalanced" \o "@LoadBalanced)注解标记的RestTemplate。
* 2.RestTemplate添加一个拦截器(filter)，当使用RestTemplate发起http调用时进行拦截。
* 3.在filter拦截到该请求时，获取该次请求服务集群的全部列表信息。
* 4.根据规则从集群中选取一个服务作为此次请求访问的目标。
* 5.访问该目标，并获取返回结果。

获取[@LoadBalanced](https://github.com/LoadBalanced" \o "@LoadBalanced)注解标记的RestTemplate。

Ribbon将所有标记[@LoadBalanced](https://github.com/LoadBalanced" \o "@LoadBalanced)注解的RestTemplate保存到一个List集合当中，具体源码如下：

@LoadBalanced

@Autowired(required = false)

private List<RestTemplate> restTemplates = Collections.emptyList();

具体源码位置是在LoadBalancerAutoConfiguration中。

RestTemplate添加一个拦截器(filter)

RestTemplate添加拦截器需要有两个步骤，首先是定义一个拦截器，其次是将定义的拦截器添加到RestTemplate中。

定义一个拦截器

实现ClientHttpRequestInterceptor接口就具备了拦截请求的功能，该接口源码如下：

public interface ClientHttpRequestInterceptor {

*/\*\**

*\*实现该方法，在该方法内完成拦截请求后的逻辑内容。*

*\*对于ribbon而言，在该方法内完成了根据具体规则从*

*\*服务集群中选取一个服务，并向该服务发起请求的操作。*

*\*/*

ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution execution) throws IOException;

}

ribbon中对应的实现类是LoadBalancerInterceptor(不使用spring-retry的情况下)具体源码如下：

public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {

private LoadBalancerClient loadBalancer;

private LoadBalancerRequestFactory requestFactory;

*//省略构造器代码...*

@Override

public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,

final ClientHttpRequestExecution execution) throws IOException {

final URI originalUri = request.getURI();

String serviceName = originalUri.getHost();

*/\*\**

*\*拦截请求，并调用loadBalancer.execute()方法*

*\*在该方法内部完成server的选取。向选取的server*

*\*发起请求，并获得返回结果。*

*\*/*

return this.loadBalancer.execute(serviceName, requestFactory.createRequest(request, body, execution));

}

}

将拦截器添加到RestTemplate中

RestTemplate继承了InterceptingHttpAccessor，在InterceptingHttpAccessor中提供了获取以及添加拦截器的方法，具体源码如下：

public abstract class InterceptingHttpAccessor extends HttpAccessor {

*/\*\**

*\* 所有的拦截器是以一个List集合形式进行保存。*

*\*/*

private List<ClientHttpRequestInterceptor> interceptors = new ArrayList<ClientHttpRequestInterceptor>();

*/\*\**

*\* 设置拦截器。*

*\*/*

public void setInterceptors(List<ClientHttpRequestInterceptor> interceptors) {

this.interceptors = interceptors;

}

*/\*\**

*\* 获取当前的拦截器。*

*\*/*

public List<ClientHttpRequestInterceptor> getInterceptors() {

return interceptors;

}

*//省略部分代码...*

}

通过这两个方法我们就可以将刚才定义的LoadBalancerInterceptor添加到有[@LoadBalanced](https://github.com/LoadBalanced" \o "@LoadBalanced)注解标识的RestTemplate中。具体的源码如下(LoadBalancerAutoConfiguration)省略部分代码：

public class LoadBalancerAutoConfiguration {

*/\*\**

*\* 获取所有带有@LoadBalanced注解的restTemplate*

*\*/*

@LoadBalanced

@Autowired(required = false)

private List<RestTemplate> restTemplates = Collections.emptyList();

*/\*\**

*\* 创建SmartInitializingSingleton接口的实现类。Spring会在所有*

*\* 单例Bean初始化完成后回调该实现类的afterSingletonsInstantiated()*

*\* 方法。在这个方法中会为所有被@LoadBalanced注解标识的*

*\* RestTemplate添加ribbon的自定义拦截器LoadBalancerInterceptor。*

*\*/*

@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);

}

}

}

};

}

*/\*\**

*\* 创建Ribbon自定义拦截器LoadBalancerInterceptor*

*\* 创建前提是当前classpath下不存在spring-retry。*

*\* 所以LoadBalancerInterceptor是默认的Ribbon拦截*

*\* 请求的拦截器。*

*\*/*

@Configuration

@ConditionalOnMissingClass("org.springframework.retry.support.RetryTemplate")

static class LoadBalancerInterceptorConfig {

@Bean

public LoadBalancerInterceptor ribbonInterceptor(

LoadBalancerClient loadBalancerClient,

LoadBalancerRequestFactory requestFactory) {

return new LoadBalancerInterceptor(loadBalancerClient, requestFactory);

}

*/\*\**

*\* 添加拦截器具体方法。首先获取当前拦截器集合(List)*

*\* 然后将loadBalancerInterceptor添加到当前集合中*

*\* 最后将新的集合放回到restTemplate中。*

*\*/*

@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);

}

};

}

}

}

至此知道了ribbon拦截请求的基本原理，接下来我们看看Ribbon是怎样选取server的。

Ribbon选取server原理概览

通过上面的介绍我们知道了当发起请求时ribbon会用LoadBalancerInterceptor这个拦截器进行拦截。在该拦截器中会调用LoadBalancerClient.execute()方法，该方法具体代码如下：

@Override

public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException {

*/\*\**

*\*创建loadBalancer的过程可以理解为组装选取服务的规则(IRule)、*

*\*服务集群的列表(ServerList)、检验服务是否存活(IPing)等特性*

*\*的过程(加载RibbonClientConfiguration这个配置类)，需要注意*

*\*的是这个过程并不是在启动时进行的，而是当有请求到来时才会处理。*

*\*/*

ILoadBalancer loadBalancer = getLoadBalancer(serviceId);

*/\*\**

*\* 根据ILoadBalancer来选取具体的一个Server。*

*\* 选取的过程是根据IRule、IPing、ServerList*

*\* 作为参照。*

*\*/*

Server server = getServer(loadBalancer);

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);

}

通过代码我们可知，首先创建一个ILoadBalancer，这个ILoadBalancer是Ribbon的核心类。可以理解成它包含了选取服务的规则(IRule)、服务集群的列表(ServerList)、检验服务是否存活(IPing)等特性，同时它也具有了根据这些特性从服务集群中选取具体一个服务的能力。  
Server server = getServer(loadBalancer);这行代码就是选取举一个具体server。  
最终调用了内部的execute方法，该方法代码如下(只保留了核心代码)：

@Override

public <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException {

try {

*//发起调用*

T returnVal = request.apply(serviceInstance);

statsRecorder.recordStats(returnVal);

return returnVal;

}

catch (IOException ex) {

statsRecorder.recordStats(ex);

throw ex;

}

catch (Exception ex) {

statsRecorder.recordStats(ex);

ReflectionUtils.rethrowRuntimeException(ex);

}

return null;

}

接下来看下request.apply(serviceInstance)方法的具体做了那些事情(LoadBalancerRequestFactory中)：

@Override

public ClientHttpResponse apply(final ServiceInstance instance)

throws Exception {

HttpRequest serviceRequest = new ServiceRequestWrapper(request, instance, loadBalancer);

*//省略部分代码...*

*/\*\**

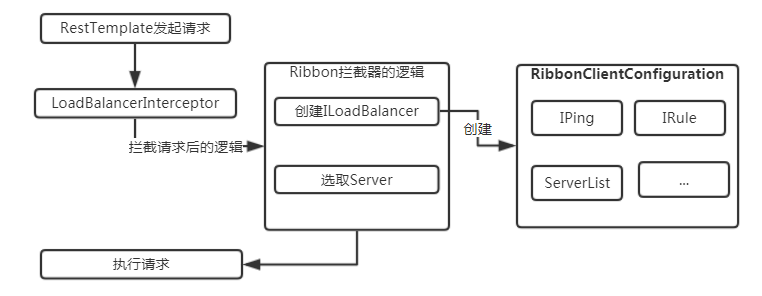
*\* 发起真正请求。*

*\*/*

return execution.execute(serviceRequest, body);

}

看到这里整体流程的原理就说完了，接下来我们结合一张图来回顾下整个过程：

首先获取所有标识[@LoadBalanced](https://github.com/LoadBalanced)注解的RestTemplate(可以理解成获取那些开启了Ribbon负载均衡功能的RestTemplate)，然后将Ribbon默认的拦截器LoadBalancerInterceptor添加到RestTemplate中，这样当使用RestTemplate发起http请求时就会起到拦截的作用。当有请求发起时，ribbon默认的拦截器首先会创建ILoadBalancer(里面包含了选取服务的规则(IRule)、服务集群的列表(ServerList)、检验服务是否存活(IPing)等特性)。在代码层面的含义是加载RibbonClientConfiguration配置类)。然后使用ILoadBalancer从服务集群中选择一个服务，最后向这个服务发送请求。

## 总结

本文首先以一个简单例子介绍了怎样使用ribbon完成客户端负载均衡的功能，然后结合源码简明的说明了ribbon负载均衡的内部原理。但是至于怎样创建ILoadBalancer以及IRule、IPing等具体实现细节还有选取服务的具体过程本文没有详细介绍，后续文章会陆续介绍。