Spring Cloud中的负载均衡 – Ribbon

在学习本节内容前，先简单介绍一下Ribbon。Ribbon是Netflix旗下的负载均衡项目，它在集群中为各个客户端的通信提供了支持，它主要实现中间层应用程序的负载均衡。

Ribbon提供了这些特性：

（1）负载均衡器，可以自定义负载均衡规则；

（2）支持多协议，如HTTP，TCP，UDP等；

（3）集成了负载均衡的客户端。

其中负载均衡器又提供了这些基础功能：

（1）维护服务器的IP、DNS名称等信息；

（2）根据特定的逻辑在服务器列表中循环。

负载均衡器包含以下三大模块：

（1）Rule: 看名称也知道，这个模块是负责负载均衡规则的，里面可以定义一些逻辑来决定从服务器列表中返回哪个服务器的实例；

（2）Ping: 定时ping各个服务器来确保网络连接正常；

（3）Server List：服务器列表，可以静态和动态的指定服务器列表，负载均衡器会从其中来抽取合适的实例。

*(概念参考《疯狂Spring Cloud微服务架构实战》一书，感谢作者归纳总结)*

好了，啰嗦完了，开干吧！

（插个广告：想看如何搭建项目环境请看博客第一节，想copy代码的请前往github: ）

**1. 自定义负载均衡规则**

整体项目结构：一个Eureka Server, 两个Service Provider实例（方便负载均衡演示）， 一个Service Invoker：

eureka-server

first-service-provider

second-service-provider(仅server port和第一个不同，只是作为演示用)

first-service-invoker

=================eureka-server中的代码==================================

很简单，我们只需要启动一个简单eureka实例就行：

启动类的配置：

@SpringBootApplication  
@EnableEurekaServer  
**public class** EurekaServerApplication {  
  
 **public static void** main(String[] args) {  
 **new** SpringApplicationBuilder(EurekaServerApplication.**class**).run(args);  
 }  
  
}

application.properties中的配置：

**server.port**=**8761  
spring.application.name**=**eureka-server  
eureka.client.fetch-registry**=**false  
eureka.client.register-with-eureka**=**false**

=================first-service-invoker中的代码==============================

我们在first-service-invoker里面自定义一个Rule 和 Ping机制：

新建MyRule类，实现IRule接口：

**public class** MyRule **implements** IRule {  
  
 **private** ILoadBalancer **loadBalancer**;  
  
 @Override  
 **public** Server choose(Object o) {  
 List<Server> servers = **loadBalancer**.getAllServers();  
 System.***out***.println(**"自定义服务器规则，以下是服务器信息："**);  
 **for** (Server server : servers) {  
 System.***out***.println(server.getHostPort());  
 }  
 *//默认情况下，Ribbon的负载均衡规则会轮换着返回不同的服务器实例  
 //这里为了方便演示自定义规则，我们指定每次都返回第一个服务器实例* **return** servers.get(0);  
 }  
  
 @Override  
 **public void** setLoadBalancer(ILoadBalancer iLoadBalancer) {  
 **this**.**loadBalancer** = iLoadBalancer;  
 }  
  
 @Override  
 **public** ILoadBalancer getLoadBalancer() {  
 **return this**.**loadBalancer**;  
 }  
}

新建MyPing类，实现IPing接口：

**public class** MyPing **implements** IPing {  
  
 @Override  
 **public boolean** isAlive(Server server) {  
 System.***out***.println(**"自定义Ping，服务器信息："** + server.getHostPort());  
 *//为了方便演示，永远返回true，即服务器一直是正常状态* **return true**;  
 }  
  
}

新建一个配置类，将这两个Bean交给Spring容器拓宽

@Configuration  
**public class** MyConfig {  
  
 @Bean  
 **public** IRule getRule() {  
 **return new** MyRule();  
 }  
  
 @Bean  
 **public** IPing getPing() {  
 **return new** MyPing();  
 }  
}

新建一个RibbonClient的配置类，非常简单，只需要加个注解就行：

@RibbonClient(name = **"first-service-provider"**, configuration = MyConfig.**class**)  
**public class** CloudProviderConfig {  
}

新建一个Controller类，我们定义一个测试接口：

RestTemplate上面加了@Bean 和 @LoadBalanced注解，前者表示加入到Spring托管，@LoadBalanced表示将使用负载均衡机制（当然这里的负载均衡会使用刚才配置的Rule和Ping规则，因为Spring Cloud集成了Ribbon）。

*NOTE：不要忘了加上@Configuration, 否则RestTemplate不会被扫描到，也就不会被Spring托管了，然后请求first-service-provider就会报UnknownHostException*

@RestController  
@Configuration  
**public class** InvokerController {  
  
 @Bean  
 @LoadBalanced  
 **public** RestTemplate getRestTemplate() {  
 **return new** RestTemplate();  
 }  
  
 @RequestMapping(**"/test"**)  
 **public** String getMsg() {  
 RestTemplate restTemplate = getRestTemplate();  
 ResponseEntity<String> result = restTemplate.exchange(**"http://first-service-provider/message"**, HttpMethod.***GET***, **null**, String.**class**);  
 **return** result.getBody();  
 }  
  
}

以下是启动类里面的代码，很简单，基本不需要做啥改动，但是要注意加上@EnableEurekaClient注解：

@SpringBootApplication  
@EnableEurekaClient  
**public class** FirstServiceInvokerApplication {  
  
 **public static void** main(String[] args) {  
 **new** SpringApplicationBuilder(FirstServiceInvokerApplication.**class**).run(args);  
 }  
  
}

以下是application.properties里面的配置：

**server.port**=**9000  
spring.application.name**=**first-service-invoker  
eureka.instance.hostname**=**localhost  
eureka.client.service-url.default-zone**=**http://localhost:8761/eureka/**

==================first-service-provider中的代码=============================

OK，接下来我们只需要在first-service-provider里面加个测试接口，因为在上面的/test接口中([**http://first-service-provider/message**](http://first-service-provider/message))我们请求的是message接口，所以我们来写一个message接口：

直接在启动类里面加个message接口：

@SpringBootApplication  
@EnableEurekaClient  
@RestController  
**public class** FirstServiceProviderApplication {  
  
 **public static void** main(String[] args) {  
 **new** SpringApplicationBuilder(FirstServiceProviderApplication.**class**)  
 .run(args);  
 }  
  
 @RequestMapping(**"/message"**)  
 **public** String getMsg(HttpServletRequest request) {  
 **return "Test success\n"** + request.getRequestURL().toString();  
 }  
}

application.properties里面的配置：

**spring.application.name**=**first-service-provider  
server.port**=**8080  
eureka.instance.hostname**=**localhost  
eureka.client.service-url.default-zone**=**http://localhost:8761/eureka/**

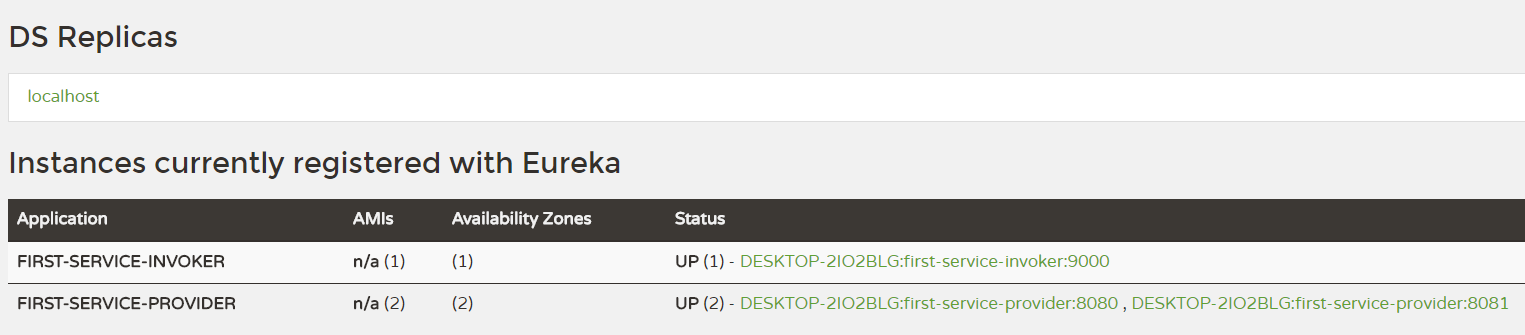
（second-service-provider中的代码和这个一样，只是application.properties里面的server.port=8081，我们主要是为了方便启动两个实例来进行演示）

===================开始测试=========================

好了，代码都编写完了，依次启动eureka-server, first-service-provider, second-service-provider, first-service-invoker

访问：<http://localhost:8761/>

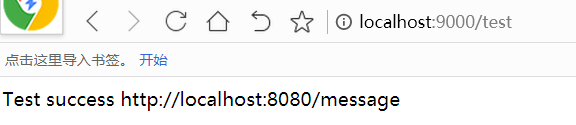
看看我们的服务是不是都注册成功了：



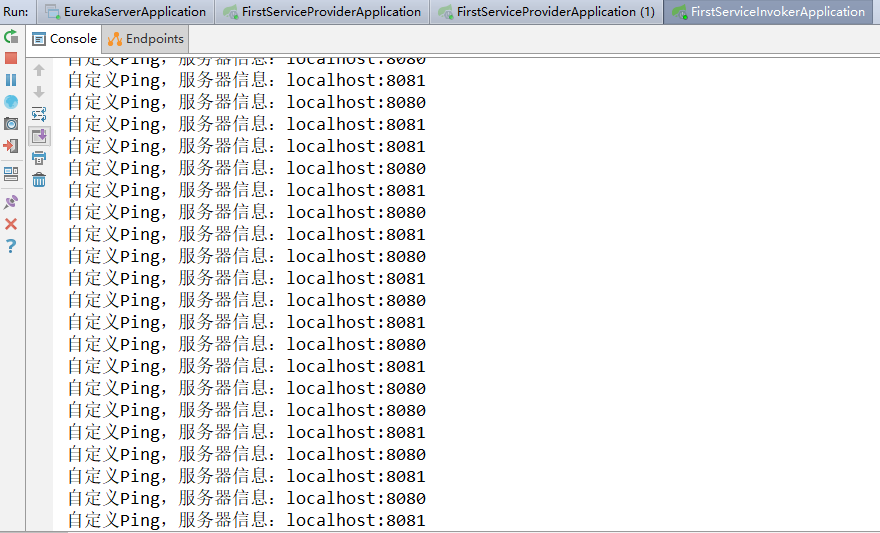
可以看到两个service-provider都可以注册好了，这个时候我们访问test接口试试：

<http://localhost:9000/test>

service-invoker的test接口回去调用service-provider的message接口，一般情况下，如果没有定义Rule规则，@LoadBalanced这个注解会使用Ribbon的默认负载均衡规则，所以肯定会显示8080端口的service-provider和8081端口的service-provider各调用一次，但是刚才我们自定义了Rule，并且在Rule里面指定了每次都返回第一个server，所以现在实际情况是，不管访问test接口多少次，显示的都是调用第一个service-provider的message接口。



然后可以看看控制台，Ribbon在定时去Ping服务器的时候，会进入我们刚才自己定义的Ping机制，即：打印服务器信息，并且永远返回true。

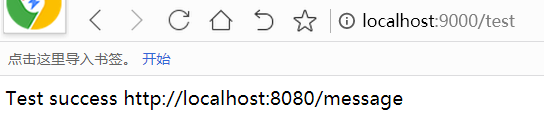


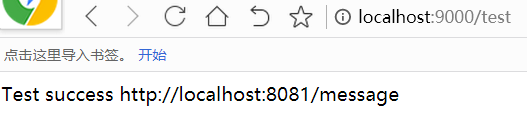
好了，现在我们将刚才的自定义负载均衡规则去掉，看看访问test接口会怎样：

*//@RibbonClient(name = "first-service-provider", configuration = MyConfig.class)***public class** CloudProviderConfig {  
}

*//@Configuration***public class** MyConfig {

会发现现在是8080和8081切换着来了：





刚才注释掉的配置不改，我们还可以在配置文件中配置Ribbon：

**first-service-provider.ribbon.NFLoadBalancerRuleClassName**: **org.cd.cloud.MyRule  
first-service-provider.ribbon.NFLoadBalancerPingClassName**: **org.cd.cloud.MyPing  
first-service-provider.ribbon.listOfServers**: **http://localhost:8080,http://localhost:8081/**

再次启动service-invoker, 效果是一样的。

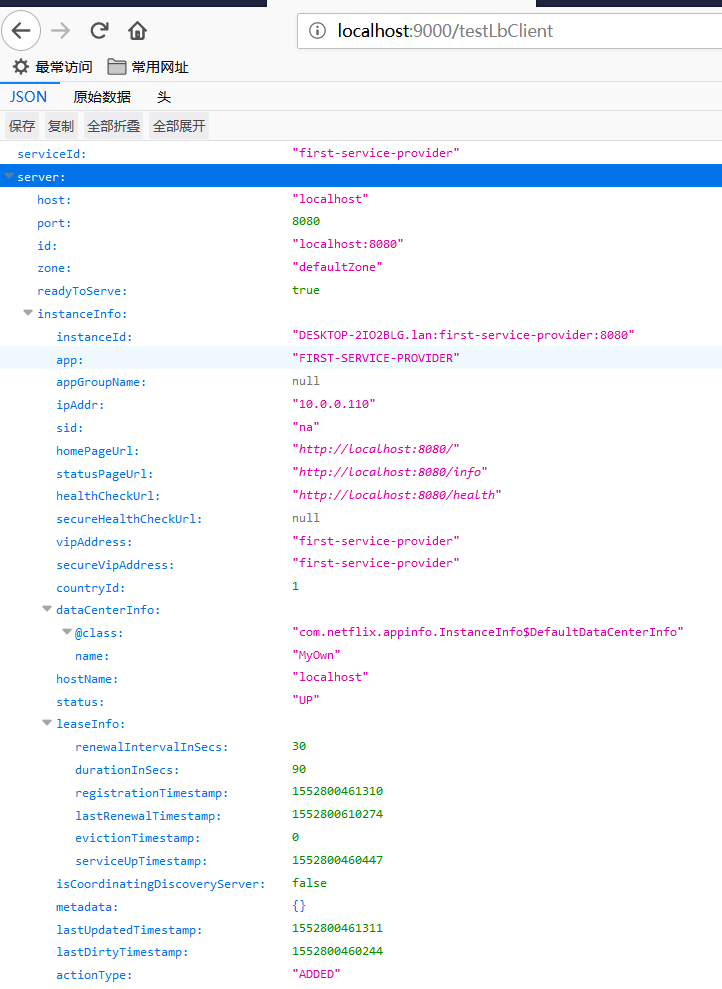
**2. Spring Cloud中使用Ribbon的API：**

Spring Cloud对Ribbon进行封装，我们可以直接使用Spring的LoadBalancerClient来处理请求以及服务器的选择。

我们在service-invoker中的InvokerController加一个接口testLbClient，获取loadBalancerClient获取到的信息：

*//==================Test Spring LoadBalancerClient==========================*@Autowired  
**private** LoadBalancerClient **loadBalancerClient**;  
  
@RequestMapping(value = **"/testLbClient"**, produces = MediaType.***APPLICATION\_JSON\_VALUE***)  
**public** ServiceInstance testLbClient() {  
 ServiceInstance serviceInstance = **loadBalancerClient**.choose(**"first-service-provider"**);  
 **return** serviceInstance;  
}

访问testLbClient接口，可以得到ServiceInstance的详细信息：



我们再加一个接口，测试在Spring Cloud中使用Ribbon原生API获取服务器实例的各种信息以及Ribbon的配置信息：

*//==================Test Origin Ribbon API==========================*@Autowired  
**private** SpringClientFactory **springClientFactory**;  
  
@RequestMapping(value = **"/testLbApi"**)  
**public** String testLbApi() {  
 System.***out***.println(**"=======输出默认配置："**);  
 *//获取默认配置* ZoneAwareLoadBalancer zalb = (ZoneAwareLoadBalancer) **springClientFactory**.getLoadBalancer(**"default"**);  
 System.***out***.println(**"IClientConfig: "** + **springClientFactory**.getLoadBalancer(**"default"**).getClass().getName());  
 System.***out***.println(**"IRule: "** + zalb.getRule().getClass().getName());  
 System.***out***.println(**"IPing: "** + zalb.getPing().getClass().getName());  
 System.***out***.println(**"ServerList: "** + zalb.getServerListImpl().getClass().getName());  
  
 System.***out***.println(**"ServerListFilter: "** + zalb.getFilter().getClass().getName());  
 System.***out***.println(**"ILoadBalancer: "** + zalb.getClass().getName());  
 System.***out***.println(**"PingInterval: "** + zalb.getPingInterval());  
  
 System.***out***.println(**"=======输出first-service-provider配置："**);  
 *//获取service-provider的配置* ZoneAwareLoadBalancer zalb2 = (ZoneAwareLoadBalancer) **springClientFactory**.getLoadBalancer(**"first-service-provider"**);  
 System.***out***.println(**"IClientConfig: "** + **springClientFactory**.getLoadBalancer(**"first-service-provider"**).getClass().getName());  
 System.***out***.println(**"IRule: "** + zalb2.getRule().getClass().getName());  
 System.***out***.println(**"IPing: "** + zalb2.getPing().getClass().getName());  
 System.***out***.println(**"ServerList: "** + zalb2.getServerListImpl().getClass().getName());  
  
 System.***out***.println(**"ServerListFilter: "** + zalb2.getFilter().getClass().getName());  
 System.***out***.println(**"ILoadBalancer: "** + zalb2.getClass().getName());  
 System.***out***.println(**"PingInterval: "** + zalb2.getPingInterval());  
  
 **return "Success"**;  
}

当访问这个API的时候，控制台输出以下内容：

=======输出默认配置：

IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer

IRule: com.netflix.loadbalancer.ZoneAvoidanceRule

IPing: com.netflix.niws.loadbalancer.NIWSDiscoveryPing

ServerList: org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList

ServerListFilter: org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter

ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer

PingInterval: 30

=======输出first-service-provider配置：

IClientConfig: com.netflix.loadbalancer.ZoneAwareLoadBalancer

**IRule: org.cd.cloud.MyRule**

**IPing: org.cd.cloud.MyPing**

ServerList: org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList

ServerListFilter: org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter

ILoadBalancer: com.netflix.loadbalancer.ZoneAwareLoadBalancer

PingInterval: 30

当然这些都是简单示例，但是通过这些helloword代码我们可以了解到Spring Cloud中如何通过Spring给出的封装接口以及Ribbon原生自带的接口来控制服务器的负载均衡。

使用Spring给出的接口无疑更加方便，可以尽量使用，但是可能不能适应所有场景，有时还是得把原生的搬出来进行改造。

**3. RestTemplate负载均衡**

首先思考个问题，为什么加入@LoadBalanced注解之后，RestTemplate就具有负载均衡的规则了呢，原因很简单，就是Spring Cloud会为所有加入这个注解的RestTemplate添加拦截器，拦截器中通过LoadBalancerClient来处理请求，通过这样的间接处理是RestTemplate有了负载均衡的功能。

我们试着自己玩玩自己写一个@LoadBalanced注解，也能使RestTemplate具有负载均衡功能。无疑这样能更加深刻的是我们认识Ribbon。

首先定义一个@MyLoadBalanced注解：

@Target({ElementType.***METHOD***, ElementType.***FIELD***, ElementType.***PARAMETER***})  
@Retention(RetentionPolicy.***RUNTIME***)  
@Qualifier  
**public** @**interface** MyLoadBalanced {  
}

然后自定义一个拦截器：

**public class** MyInterceptor **implements** ClientHttpRequestInterceptor {  
 @Override  
 **public** ClientHttpResponse intercept(HttpRequest httpRequest, **byte**[] bytes, ClientHttpRequestExecution clientHttpRequestExecution) **throws** IOException {  
 System.***out***.println(**"========这是自定义拦截器========"**);  
 System.***out***.println(**"原来的URI: "** + httpRequest.getURI());  
 *//更换URI* MyHttpRequest myHttpRequest = **new** MyHttpRequest(httpRequest);  
 System.***out***.println(**"更换后的URI: "** + myHttpRequest.getURI());  
 **return** clientHttpRequestExecution.execute(myHttpRequest, bytes);  
 }  
}

这个拦截器中，自定义了一个MyHttpRequest，这个是处理请求的，代码如下：

**public class** MyHttpRequest **implements** HttpRequest {  
  
 **private** HttpRequest **sourceRequest**;  
  
 **public** MyHttpRequest() {}  
  
 **public** MyHttpRequest(HttpRequest request) {  
 **this**.**sourceRequest** = request;  
 }  
  
 @Override  
 **public** HttpMethod getMethod() {  
 **return sourceRequest**.getMethod();  
 }  
  
 @Override  
 **public** URI getURI() {  
 **try** {  
 String oldUri = **sourceRequest**.getURI().toString();  
 System.***out***.println(**"Old URI: "** + oldUri);  
 URI newUri = **new** URI(**"http://localhost:8080/message"**);  
 **return** newUri;  
 } **catch** (URISyntaxException e) {  
 e.printStackTrace();  
 }  
 **return sourceRequest**.getURI();  
 }  
  
 @Override  
 **public** HttpHeaders getHeaders() {  
 **return sourceRequest**.getHeaders();  
 }  
}

我把request URI指定了<http://localhost:8080/message>, 所以会一直路由到这个服务器的这个API上去。你也可以再这里面加入复杂的处理机制，其实负载均衡主要就是处理怎么合理的按照你要求的方式路由到不同的服务器。

最后，就是把这个拦截器配置到Spring容器：

@Configuration  
**public class** MyLoadBalancerConfig {  
  
 @Autowired(required = **false**)  
 @MyLoadBalanced  
 **private** List<RestTemplate> **myTemplates** = Collections.*emptyList*();  
  
 **public** SmartInitializingSingleton myLoadBalancedRestTemplate() {  
 **return new** SmartInitializingSingleton() {  
 @Override  
 **public void** afterSingletonsInstantiated() {  
 **for** (RestTemplate restTemplate : **myTemplates**) {  
 MyInterceptor myInterceptor = **new** MyInterceptor();  
 List list = **new** ArrayList(restTemplate.getInterceptors());  
 list.add(myInterceptor);  
 restTemplate.setInterceptors(list);  
 }  
 }  
 };  
 }  
}

好了，将first-service-invoker中的InvokerController中的@LoadBalanced注解换成@MyLoadBalanced试试吧！