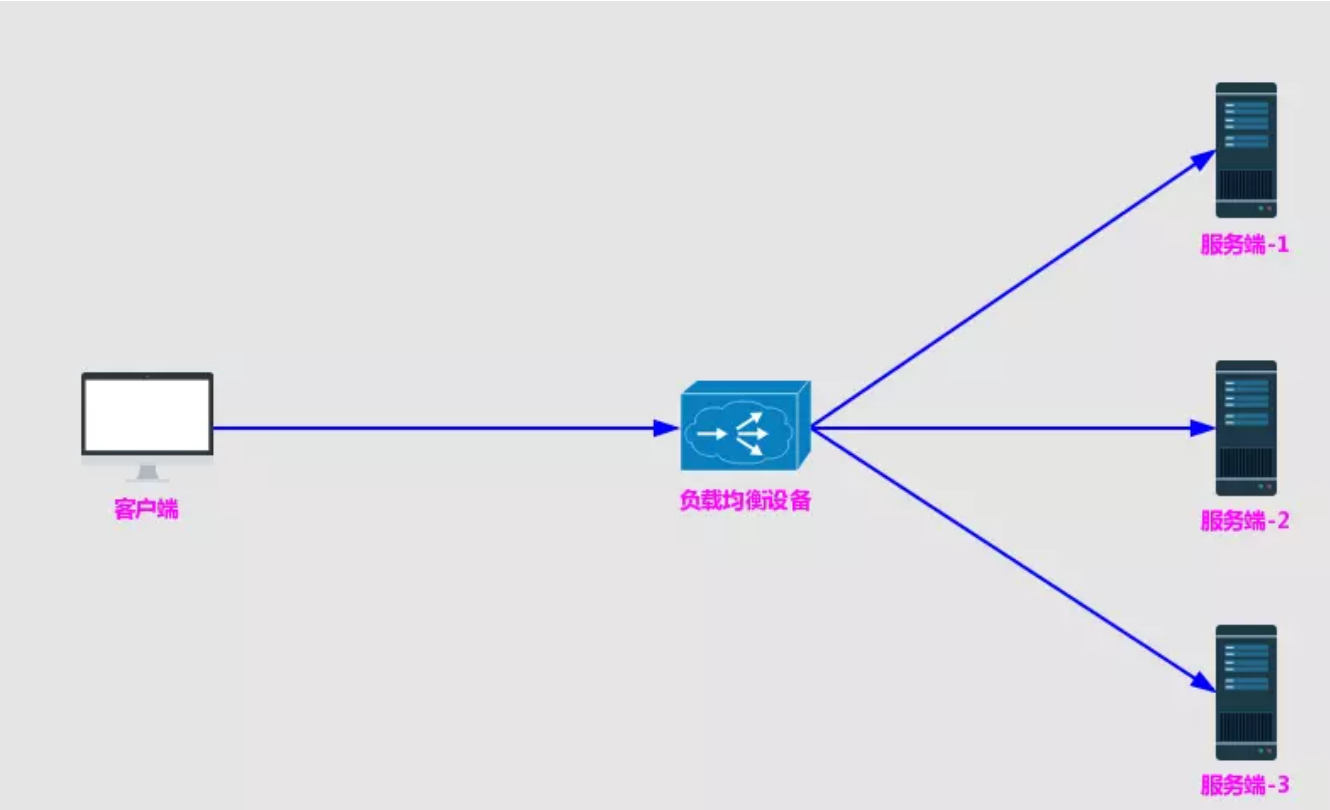
Ribbon详解

> 更多学习资料，一点课堂：www.yidiankt.com
>
> 一点课堂QQ群：984370849， QQ：2868289889
>
> 微信号：chengweixin9
>
> 免费公开课：https://ke.qq.com/course/394307

> https://www.jianshu.com/p/1bd66db5dc46

客户端负载均衡

负载均衡在系统架构中是一个非常重要，并且是不得不去实施的内容。因为负载均衡是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一。我们通常所说的负载均衡都指的是服务端负载均衡，其中分为硬件负载均衡和软件负载均衡。硬件负载均衡主要通过在服务器节点之间按照专门用于负载均衡的设备，比如F5等；而软件负载均衡则是通过在服务器上安装一些用于负载均衡功能或模块等软件来完成请求分发工作，比如Nginx等。不论采用硬件负载均衡还是软件负载均衡，只要是服务端都能以类似下图的架构方式构建起来：



负载均衡架构图

硬件负载均衡的设备或是软件负载均衡的软件模块都会维护一个下挂可用的服务端清单，通过心跳检测来剔除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端发送请求到负载均衡设备的时候，该设备按某种算法（比如线性轮询、按权重负载、按流量负载等）从维护的可用服务端清单中取出一台服务端端地址，然后进行转发。

而客户端负载均衡和服务端负载均衡最大的不同点在于上面所提到服务清单所存储的位置。在客户端负载均衡中，所有客户端节点都维护着自己要访问的服务端清单，而这些服务端端清单来自于服务注册中心，比如上一章我们介绍的Eureka服务端。同服务端负载均衡的架构类似，在客户端负载均衡中也需要心跳去维护服务端清单的健康性，默认会创建针对各个服务治理框架的Ribbon自动化整合配置，

比如Eureka中的org.springframework.cloud.netflix.ribbon.eureka.RibbonEurekaAutoConfiguration，

Consul中的org.springframework.cloud.consul.discovery.RibbonConsulAutoConfiguration。

在实际使用的时候，我们可以通过查看这两个类的实现，以找到它们的配置详情来帮助我们更好地使用它。

通过Spring Cloud Ribbon的封装，我们在微服务架构中使用客户端负载均衡调用非常简单，只需要如下两步：

▪服务提供者只需要启动多个服务实例并注册到一个注册中心或是多个相关联的服务注册中心。

▪服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate来实现面向服务的接口调用。

这样，我们就可以将服务提供者的高可用以及服务消费者的负载均衡调用一起实现了。

内置负载均衡规则类	**规则描述**
RoundRobinRule	简单轮询服务列表来选择服务器。它是Ribbon默认的负载均衡规则。
AvailabilityFilteringRule	对以下两种服务器进行忽略： （1）在默认情况下，这台服务器如果3次连接失败，这台服务器就会被设置为“短路”状态。短路状态将持续30秒，如果再次连接失败，短路的持续时间就会几何级地增加。 （2）并发数过高的服务器。如果一个服务器的并发连接数过高，配置了AvailabilityFilteringRule规则的客户端也会将其忽略。并发连接数的上线，可以由客户端的进行配置。
WeightedResponseTimeRule	为每一个服务器赋予一个权重值。服务器响应时间越长，这个服务器的权重就越小。这个规则会随机选择服务器，这个权重值会影响服务器的选择。
ZoneAvoidanceRule	以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。
BestAvailableRule	忽略哪些短路的服务器，并选择并发数较低的服务器。
RandomRule	随机选择一个可用的服务器。
Retry	重试机制的选择逻辑

修改负载均衡策略配置

```
yidiankt-user:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
```

Ribbon源码分析

LoadBalancerAutoConfiguration.java为实现客户端负载均衡器的自动化配置类。

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

在自动化配置中主要做三件事：

- 创建一个LoadBalancerInterceptor的Bean，用于实现对客户端发起请求时进行拦截，以实现客户端负载均衡。
- 创建一个RestTemplateCustomizer的Bean，用于给RestTemplate增加LoadBalancerInterceptor拦截器。
- 维护了一个被@LoadBalanced注解修饰的RestTemplate对象列表，并在这里进行初始化，通过调用RestTemplateCustomizer的实例来给需要客户端负载均衡的RestTemplate增加LoadBalancerInterceptor拦截器。

从@LoadBalanced注解码的注释中，可以知道该注解用来给RestTemplate标记，以使用负载均衡的客户端（LoadBalancerClient）来配置它。

- LoadBalancerClient

```
public interface LoadBalancerClient extends ServiceInstanceChooser {
    <T> T execute(String var1, LoadBalancerRequest<T> var2) throws IOException;

    // 从负载均衡器中挑选出的服务实例来执行请求内容。
    <T> T execute(String var1, ServiceInstance var2, LoadBalancerRequest<T> var3) throws
IOException;

    // 为了给一些系统使用，创建一个带有真实host和port的URI。
    // 一些系统使用带有原服务名代替host的URI，比如http://myservice/path/to/service。
    // 该方法会从服务实例中取出host:port来替换这个服务名。
    URI reconstructURI(ServiceInstance var1, URI var2);
}
```

- 父接口ServiceInstanceChooser

```
public interface ServiceInstanceChooser {

    /**
     * Choose a ServiceInstance from the LoadBalancer for the specified service
     * @param serviceId the service id to look up the LoadBalancer
     * @return a ServiceInstance that matches the serviceId
     */
    // 根据传入的服务名serviceId，从负载均衡器中挑选一个对应服务的实例。
    ServiceInstance choose(String serviceId);
}
```

- RibbonLoadBalancerClient 实现类代码

```
@Override
public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws IOException {
    // 获取负载均衡策略
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
    // 根据负载均衡策略，获取一个服务器
    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);
}
```

- ILoadBalancer接口

```
public interface ILoadBalancer {
    //向负载均衡器的实例列表中增加实例
    public void addServers(List<Server> newServers);

    //通过某种策略，从负载均衡器中选择一个具体的实例
    public Server chooseServer(Object key);

    //用来通知和标识负载均衡器中某个具体实例已经停止服务，不然负载均衡器在下一次获取服务实例清单前都会认为服务实例均是正常服务的。
    public void markServerDown(Server server);

   //获取正常服务列表
    public List<Server> getReachableServers();

    //所有已知实例列表
    public List<Server> getAllServers();
}
```

- LoadBalancerContext类中实现

```java
// 转换host:port形式的请求地址。
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);
    }
}
```