Spring Cloud Ribbon：客户端负载均衡

- 是一个基于 HTTP 和 TCP 的客户端负载均衡工具，它基于 NetflixRibbon实现。
- 通过SpringCloud的封装，可以轻松地将面向服务的REST模板请求自动转换成客户端负载均衡的服务调用。
- 不需要独立部署，几乎存在于每一个Spring Cloud 构建的微服务和基础设施中
- 微服务间的调用，API 网关的请求转发等内容实际上都是通过Ribbon 来实现的
- 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
- 服务端负载均衡
  - 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的地址， 然后进行转发
- 客户端负载均衡
  - 客户端负载均衡和服务端负载均衡最大的不同点在千上面所提到的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， 而这些服务端的清单来自于服务注册中心
  - 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，只是这个步骤 需要与服务注册中心配合完成

使用

- 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

RestTemplate

- 该对象会使用 Ribbon 的自动化配置， 同时通过配置 @LoadBalanced 还能够开启客户端负载均衡
- Get请求
  - getForEntity 函数。该方法返回的是 ResponseEntity, 该对象是 Spring对 HTTP 请求响应的封装

```
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<User> responseEntity =
restTemplate.getForEntity("http://USER• SERVICE/user?name= {1}",
User.class, "didi");
User body= responseEntity.getBody();
```

- getForObject 函数。该方法可以理解为对 getForEntity 的进一步封装，它通过 HttpMessageConverterExtractor 对 HTTP 的请求响应体 body内容进行对象转换， 实现请求直接返回包装好的对象内容

```
RestTemplate restTemplate = new RestTemplate();
User result = restTemplate.getForObject(uri, User.class);
```

- Post请求
  - postForEntity 函数。 该方法同 GET 请求中的 getForEntity 类似， 会在调用后返回 ResponseEn巨ty对象， 其中 T 为请求响应的 body类型
  - postForObject 函数。 该方法也跟getForObject 的类型类似， 它的作 用是简化 postForEntity的后续处理。 通过直接将请求响应的body内容包装成对象来返回使用

```
RestTemplate restTemplate = new RestTemplate();
User user = new User("didi", 20);
String postResult = restTemplate.postForObject("http://USER-
SERVICE/user", user,
String.class);
```

  - postForLocation函数。 该方法实现了以POST请求提交资源， 并返回新资源的URI

```
User user = new User("didi", 40);
URI responseURI = restTemplate.postForLocation("http://USER-
SERVICE/user", user);
```

- Put、Delete...

源码分析

- LoadBalancerClient接口
  - ServiceInstance choose(String 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
  - Texecute(StringserviceidL,oadBalancerRequestrequest)throws IOException: 使用从负载均衡器中挑选出的服务实例来执行请求内容
  - URI reconstructURI(Serviceinstance instance , URI original :) 为系统构建一个合适的host:port形式的URI
  - 在分布式系统中， 我们使用逻辑上的服务名称作为host来构建URI（替代服务实例的host:port形式)进行请求， 比如http://myservice/path/to/service 。 在该操作的定义中，前者 ServiceInstance对象是带有host和port的具体服务实例， 而后者URI对象则是使用逻辑服务名定义为host的URI , 而返回的URI内容则是通过ServiceInstance的服务实例详情拼接出的具体host:post形式的请求地址
- Spring CloudR巾bon中实现客户端负载均衡的基本脉络
  - 通过 LoadBalancerinterceptor拦截器对 RestTemplate 的请求进行拦截， 并利用Spring Cloud的负载均衡器 LoadBalancerClient将以逻辑服 务名为host的 URI转换成具体的服务实例地址

    - LoadBalancerClient的Ribbon实现RibbonLoadBalancerClient, 可以知道在使用Ribbon实现负载均衡器的时候， 实际使用的还是Ribbon中定义的ILoadBalancer接口的实现，自动化配置会采用ZoneAwareLoadBalancer的实例来实现客户端负载均衡
- 负载均衡器
    - 虽然 Spring Cloud中定义了LoadBalancerClient工作为负载均衡器的通用接口， 并且针对Ribbon实现了RibbonLoadBalancerClient， 但是它在具体实现客户端负载均衡时，是通过Ribbon的ILoadBalancer接口实现的。
- 负载均衡策略
    - Random Rule：从服务实例清单中随机选择 一个服务实例的功能
    - RoundRobinRule：实现了按照线性轮询的方式依次选择每个服务实例的功能
    - RetryRule：实现了 一 个具备重试机制的实例选择功能
    - Weighted ResponseTimeRule：是对 RoundRobinRule 的扩展， 增加了根据实例的运行情况来计算权重， 并根据权重来挑选实例， 以达到更优的分配效果

配置详解

- 由于Ribbon中定义的每一个接口都有多种不同的策略实现，同时这些接口之间又有一定的依赖关系，如何选择具体的实现策略以及如何组织它们的关系 ——Spring CloudRibbon中的自动化配置
- 在引入Spring CloudRibbon的 依赖之后， 就能够自动化构建下面这些接口的实现。
    - 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.NoOpPinng 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 ，默认采用ZonePreferenceServerListFilter实现，该策略能够优先过滤出与请求调用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
- 上面这些自动化配置内容仅在没有引入Spring CloudEureka等服务治理框架时如此， 在同时引入Eureka和Ribbon依赖时，自动化配置会有 一 些不同
- 通过自动化配置的实现，可以轻松地实现客户端负载均衡 。 同时，针对一些个性化需求，也可以方便地替换上面的这些默认实现。 只需在SpringBoot应用中创建对应的实现实例就能覆盖这些默认的配置实现
- 另外， 也可以通过使用@RibbonClient 注解来实现更细粒度的客户端配置
- Spring Cloud和Ribbon对 RibbonClient 定义个性化配置的方法做了进一 步优化。 可以直接通过.ribbon.=的形式进行配置
    - 将 hello-service 服务客户端的IPing 接口实现替换为 PingUrl, 只需在 application.properties 配置中增加下面的内容即可`hello-service.ribbon.NF LoadBalancerPingClassName=com.netflix.loadbalancer.PingUrl`
    - 其中 hello-service 为服务名， NFLoadBalancerPingClassName 参数用来指定具体 的 IPing 接口实现类

- 在Camden版本中我们可以通过配置的方式，更加方便地为RibbonClient指定ILoadBalancer、IPing、IRule、ServerList和ServerListFilter的定制化实现
- 参数配置
  - 全局配置
    - 只需使用 ribbon.=格式进行配置即可
  - 指定客户端的配置方式
    - 采用 .ribbon.=的格式
    - 代表了客户端 的名称， 如上文中在@RibbonClient中指定的名称， 也可以将它理解为是一个服务名

与Eureka结合

- 当在Spring Cloud的应用中同时引入Spring CloudRibbon和Spring CloudEureka依赖时，会触发Eureka中实现的对Ribbon的自动化配置。
  - 这时ServerList的维护机制实现将被com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList的实例所覆盖，该实现会将服务清单列表交给Eureka的服务治理机制来进行维护
  - IPing的实现将被com.netflix.niws.loadbalancer.NIWSDiscoveryPing 的实例所覆盖，该实现也将实例检查的任务交给了服务治理框架来进行维护
  - 默认情况下，用于获取实例请求的ServerList接口实现将采用Spring CloudEureka中封装的org.springframework. cloud.netflix.ribbon.eureka.DomainExtractingServerList，其目的是为了让实例维护策略更加通用， 所以将使用物理元数据来进行负载均衡， 而不是使用原生的AWSAMI元数据。
- 在与SpringCloud Eureka结合使用的时候， 我们的配置将会变得更加简单。不再需要通过类似hello-service.ribbon. listOfServers的参数来指定具体的服务实例清单，因为Eureka将会为我们维护所有服务的实例清单。
- 而对于Ribbon的参数配置， 我们依然可以采用之前的两种配置方式来实现， 而指定客户端的配置方式可以直接使用Eureka 中的服务名作为来完成针对各个微服务的个性化配置
- 此外，由于SpringCloudRibbon默认实现了区域亲和策略，所以，我们可以通过Eureka 实例的元数据配置来实现区域化的实例配置方案。 比如，可以将处于不同机房的实例配置成不同的区域值，以作为跨区域的容错机制实现。 而实现的方式非常简单， 只需在服务实例的元数据中增加zone参数来指定自己所在的区域

  `eureka.instance.metadataMap.zone=shanghai`
- 在SpringCloudRibbon与SpringCloud Eureka结合的工程中， 也可以通过参数配置的方式来禁用Eureka对Ribbon服务实例的维护实现`ribbon.eureka.enabled=false`。这时对于服务实例的维护就又将回归到使用 . ribbon.listOfServers参数配置的方式来实现了
- 重试机制
  - 由于SpringCloud Eureka实现的服务治理机制强调了CAP原理中的AP，即可用性与可靠性，它与Zoo Keeper这类强调CP(一致性、可靠性)的服务治理框架最大的区别就是
    - Eureka为了实现更高的服务可用性， 牺牲了一定的一致性， 在极端情况下它宁愿接受故障实例也不要丢掉 “ 健康 ” 实例
    - 比如， 当服务注册中心的网络发生故障断开时， 由于所有的服务实例无法维持续约心跳， 在强调AP的服务治理中将会把所有服务实例都剔除掉，而Eureka则会因为超过

85%的实例丢失心跳而会触发保护机制，注册中心将会保留此时的所有节点， 以实现服务间依然可以进行互相调用的场景， 即使其中有部分故障节点， 但这样做可以继续保障大多数的服务正常消费

- 由于SpringCloud Eureka在可用性与一致性上的取舍， 不论是由于触发了保护机制还是服务剔除的延迟， 引起服务调用到故障实例的时候， 我们还是希望能够增强对这类问题的容错。所以， 我们在实现服务调用的时候通常会加入一些重试机制

```
# 开启重试机制，默认关闭
spring.cloud.loadbalancer.retry.enabled=true
# 断路器的超时时间需要大于Ribbon的超时时间， 不然不会触发重试
hystrix.command.default.execution.isolation.thread.timeoutinMillisecon
ds=lOOOO
# 请求连接的超时时间
hello-service.ribbon.ConnectTimeout= 250
# 请求处理的超时时间。
hello-service.ribbon.ReadTimeout= lOOO
# 对所有操作请求都 进行重试。
hello-service.ribbon.OkToRetryOnAllOperations=true
#切换实例的重试次 数
hello-service.ribbon.MaxAutoRetriesNextServer=2
#对当前实例的重试次数
hello-service.ribbon.MaxAutoRetries=l
```

  - 根据如上配置，当访问到故障请求的时候，它会再尝试访问一次当前实例(次数由MaxAutoRetries配置)， 如果不行，就换一个实例进行访问，如果还是不行，再换一次实例访问(更换次数由MaxAutoRe红iesNextServer配置)， 如果依然不行， 返回失败信息