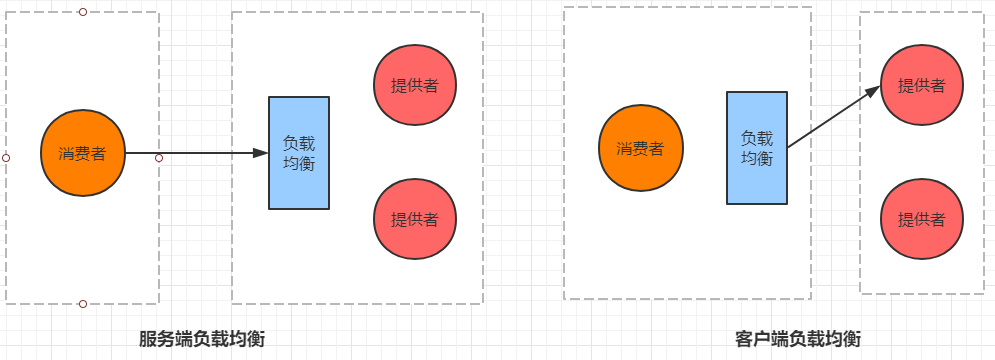
# 实现服务调用的负载均衡

## 什么是负载均衡

通俗的讲， 负载均衡就是将负载（工作任务，访问请求）进行分摊到多个操作单元（服务器,组件）上进行执行。

根据负载均衡发生位置的不同,一般分为**服务端负载均衡**和**客户端负载均衡**。服务端负载均衡指的是发生在服务提供者一方,比如常见的nginx负载均衡

而客户端负载均衡指的是发生在服务请求的一方，也就是在发送请求之前已经选好了由哪个实例处理请 求。

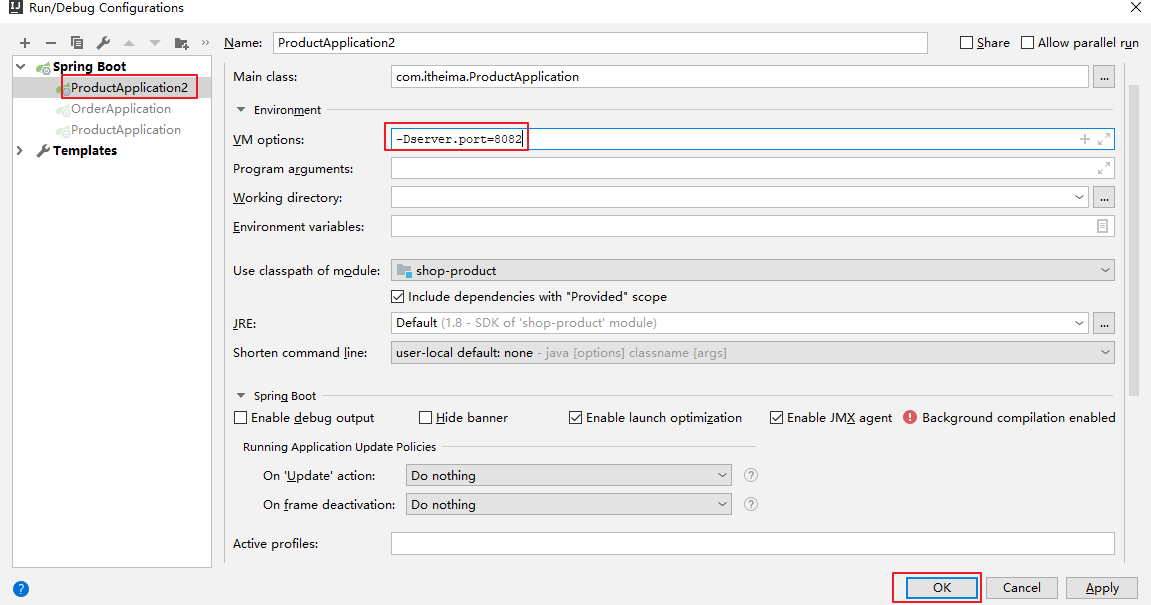


我们在微服务调用关系中一般会选择客户端负载均衡，也就是在服务调用的一方来决定服务由哪个提供 者执行。

# 自定义实现负载均衡

1 通过idea再启动一个shop-product 微服务，设置其端口为8082





2 通过nacos查看微服务的启动情况

3 修改shop-order 的代码，实现负载均衡

@RestController @Slf4j

public class OrderController {

@Autowired

private RestTemplate restTemplate;

@Autowired

private OrderService orderService;

@Autowired

private DiscoveryClient discoveryClient;

//准备买1件商品

@GetMapping("/order/prod/{pid}")

public Order order(@PathVariable("pid") Integer pid) { log.info(">>客户下单，这时候要调用商品微服务查询商品信息");

//从nacos中获取服务地址

//自定义规则实现随机挑选服务



List<ServiceInstance> instances = discoveryClient.getInstances("service- product");

int index = new Random().nextInt(instances.size()); ServiceInstance serviceInstance = instances.get(index);

String url = serviceInstance.getHost() + ":" + serviceInstance.getPort();

log.info(">>从nacos中获取到的微服务地址为:" + url);

//通过restTemplate调用商品微服务

Product product = restTemplate.getForObject("http://" + url + "/product/" + pid, Product.class);

log.info(">>商品信息，查询结果:" + JSON.toJSONString(product));

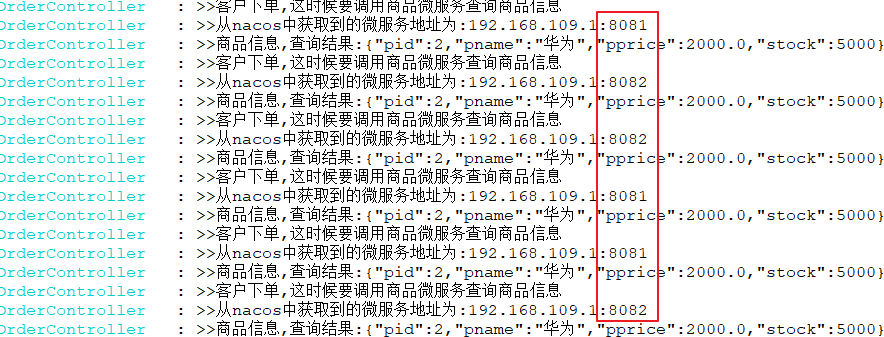
Order order = new Order(); order.setUid(1); order.setUsername(" 测 试 用 户 "); order.setPid(product.getPid()); order.setPname(product.getPname());

order.setPprice(product.getPprice()); order.setNumber(1);

orderService.save(order); return order;

}

}



第3步：启动两个服务提供者和一个服务消费者，多访问几次消费者测试效果

# 基于Ribbon实现负载均衡

## Ribbon是Spring Cloud的一个组件， 它可以让我们使用一个注解就能轻松的搞定负载均衡

第1步：在RestTemplate 的生成方法上添加@LoadBalanced注解

@Bean @LoadBalanced

public RestTemplate restTemplate() { return new RestTemplate();

}

第2步：修改服务调用的方法

## Ribbon支持的负载均衡策略

@RestController @Slf4j

public class OrderController {

@Autowired

private RestTemplate restTemplate;

@Autowired

private OrderService orderService;

//准备买1件商品@GetMapping("/order/prod/{pid}")

public Order order(@PathVariable("pid") Integer pid) { log.info(">>客户下单，这时候要调用商品微服务查询商品信息");

//直接使用微服务名字， 从nacos中获取服务地址

String url = "service-product";

//通过restTemplate调用商品微服务

Product product = restTemplate.getForObject(

"http://" + url + "/product/" + pid, Product.class);

log.info(">>商品信息，查询结果:" + JSON.toJSONString(product));

Order order = new Order(); order.setUid(1); order.setUsername(" 测 试 用 户 "); order.setPid(product.getPid()); order.setPname(product.getPname());

order.setPprice(product.getPprice()); order.setNumber(1);

orderService.save(order); return order;

}

}



Ribbon内置了多种负载均衡策略,内部负载均衡的顶级接口为

com.netflix.loadbalancer.IRule , 具体的负载策略如下图所示:

|  |  |  |
| --- | --- | --- |
| **策略名** | **策略描述** | **实现说明** |
| BestAvailableRule | 选择一个最小的并发请求的server | 逐个考察Server，如果Server被tripped了，则忽略，在选择其中ActiveRequestsCount最小的server |
| AvailabilityFilteringRule | 过滤掉那些因为一直连接失败的被标记为circuit tripped的后端server，并过滤掉那些高并发的的后端server（active connections 超过配置的阈值） | 使用一个AvailabilityPredicate来包含过滤server的逻辑，其实就就是检查status里记录的各个server的运行状态 |
| WeightedResponseTimeRule | 根据相应时间分配一个weight，相应时间越长，weight越小，被选中的可能性越低。 | 一个后台线程定期的从status里面读取评价响应时间，为每个server计算一个weight。Weight的计算也比较简单responsetime 减去每个server自己平均的responsetime是server的权重。当刚开始运行，没有形成statas 时，使用roubine策略选择server。 |
| RetryRule | 对选定的负载均衡策略机上重试机制。 | 在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server |
| RoundRobinRule | 轮询方式轮询选择  server | 轮询index，选择index对应位置的  server |
| RandomRule | 随机选择一个server | 在index上随机，选择index对应位置的server |
| ZoneAvoidanceRule | 复合判断server所在区域的性能和server 的可用性选择server | 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用 的 zone（ 的 所 有 server）， AvailabilityPredicate用于过滤掉连接数过多的Server。 |



我们可以通过修改配置来调整Ribbon的负载均衡策略，具体代码如下

service-product: # 调用的提供者的名称

ribbon:

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule