1. 负载均衡介绍

1.1 问题描述

观察上个章节远程调用的代码

```
1 List<ServiceInstance> instances = discoveryClient.getInstances("product-
  service");
2 //服务可能有多个，获取第一个
3 EurekaServiceInstance instance = (EurekaServiceInstance) instances.get(0);
```

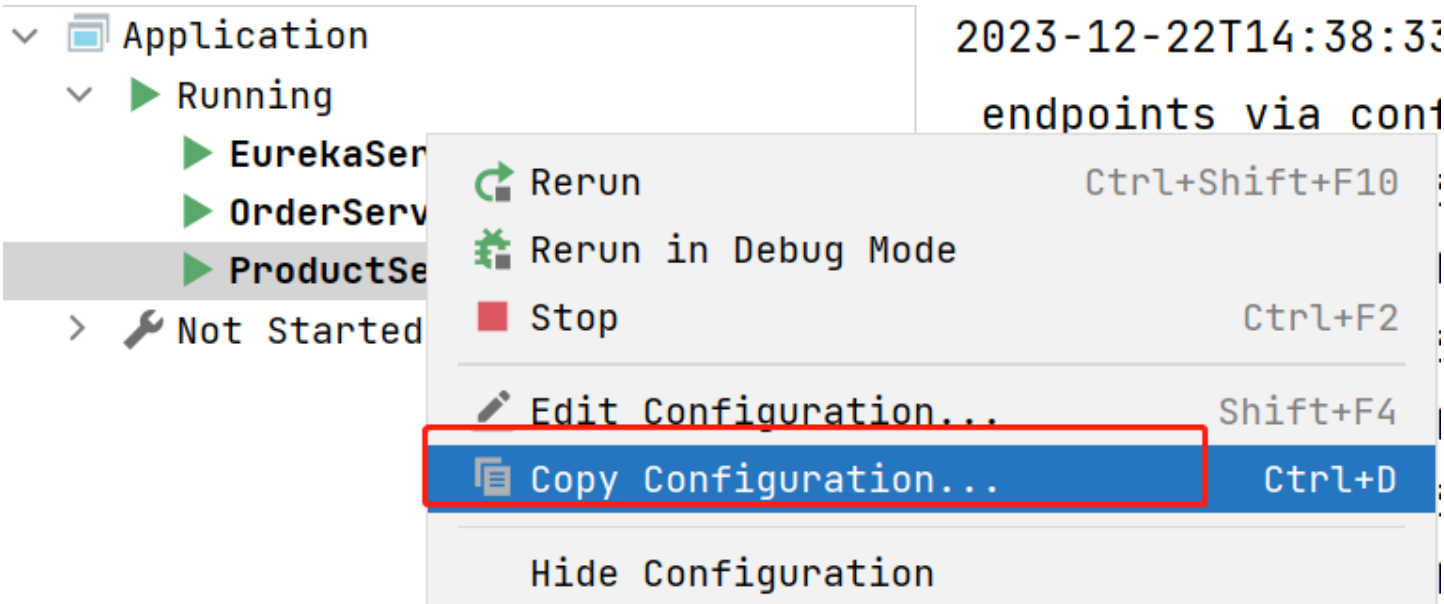
1. 根据应用名称获取了服务实例列表
2. 从列表中选择了一个服务实例

思考: 如果一个服务对应多个实例呢? 流量是否可以合理的分配到多个实例呢?

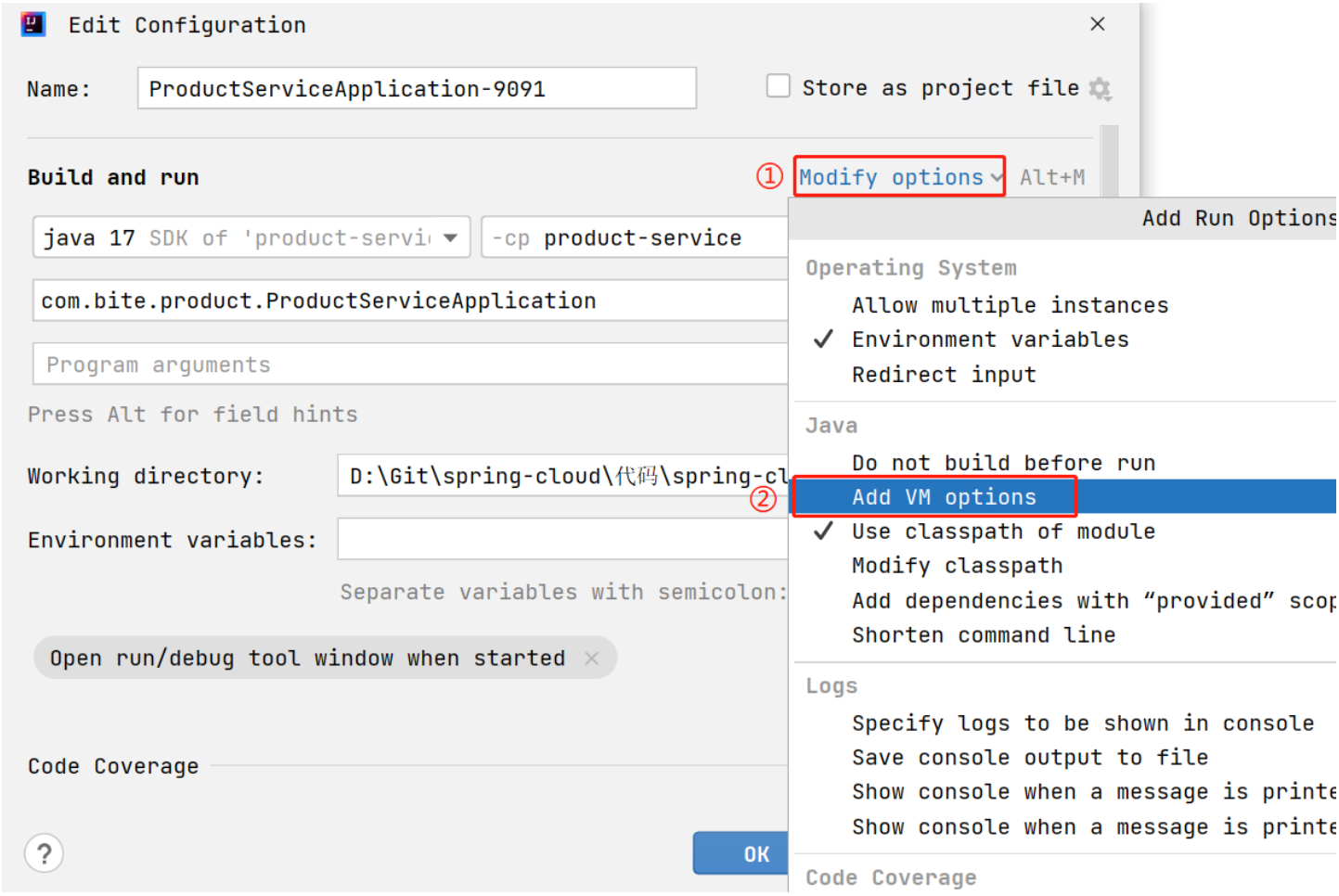
现象观察:

我们再启动2个product-service实例

选中要启动的服务, 右键选择 `Copy Configuration...`

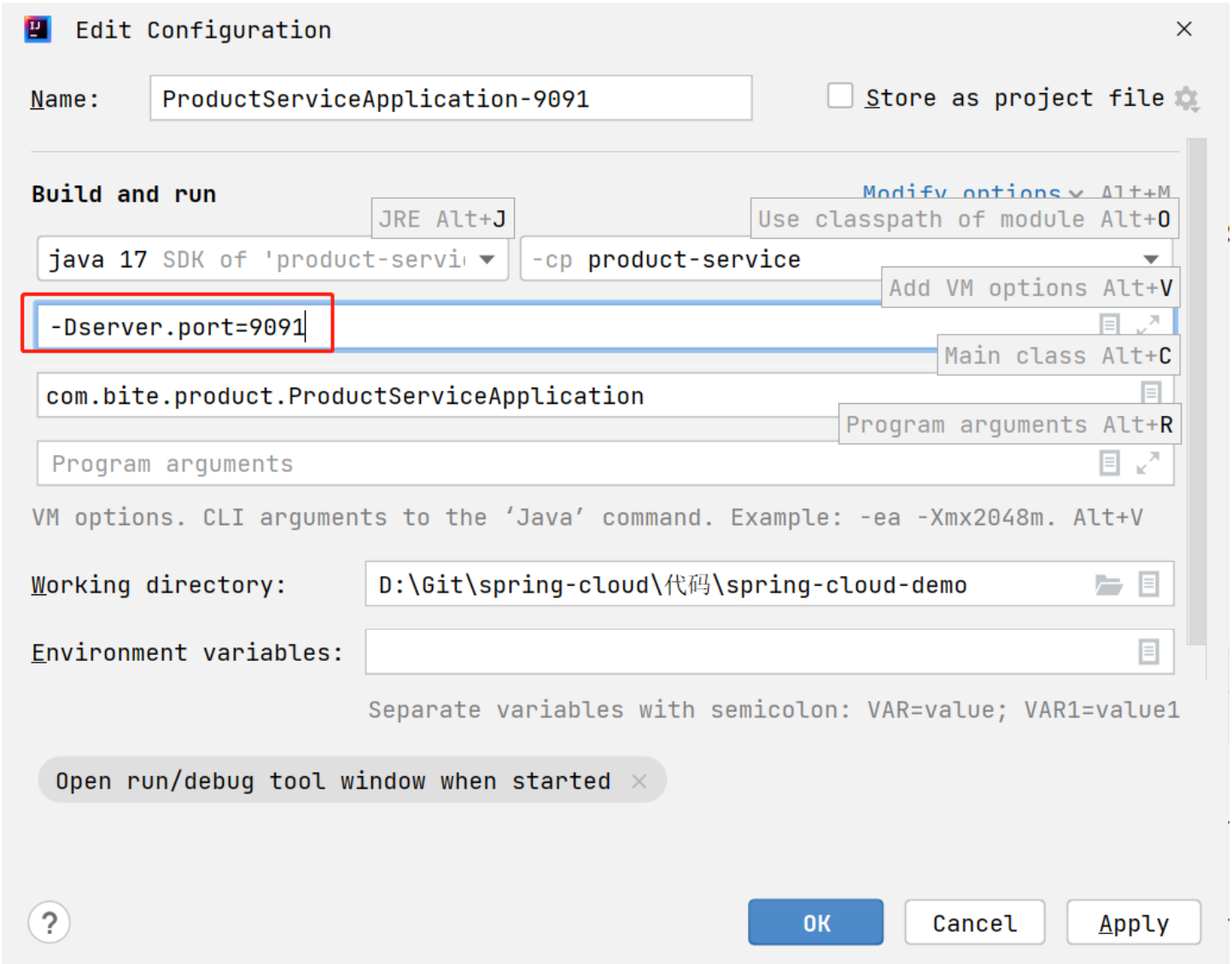


在弹出的框中, 选择 `Modify options` -> `Add VM options`

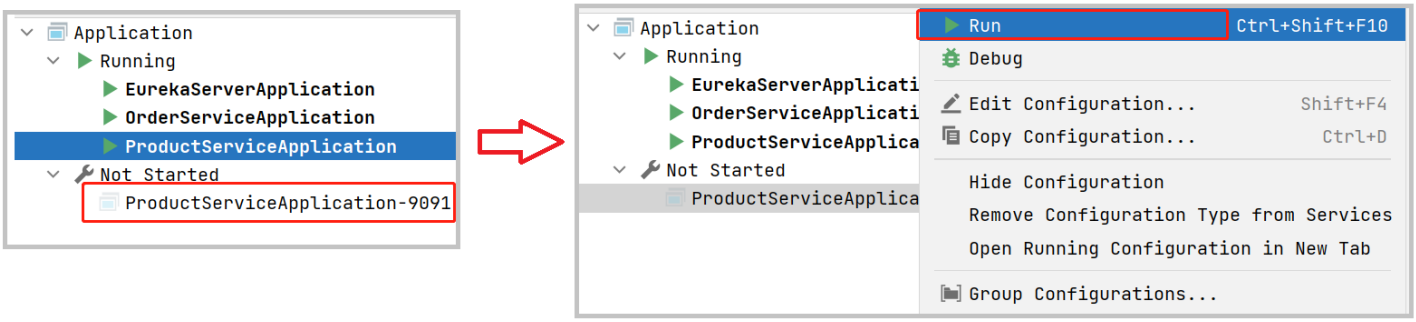


添加 `VM options`：`-Dserver.port=9091`

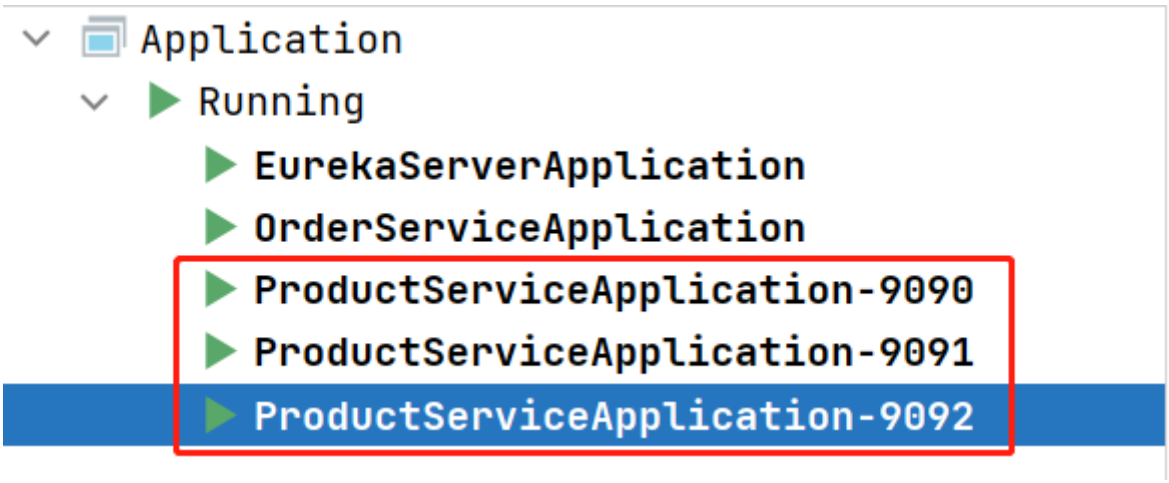
> 9091 为服务启动的端口号，根据自己的情况进行修改



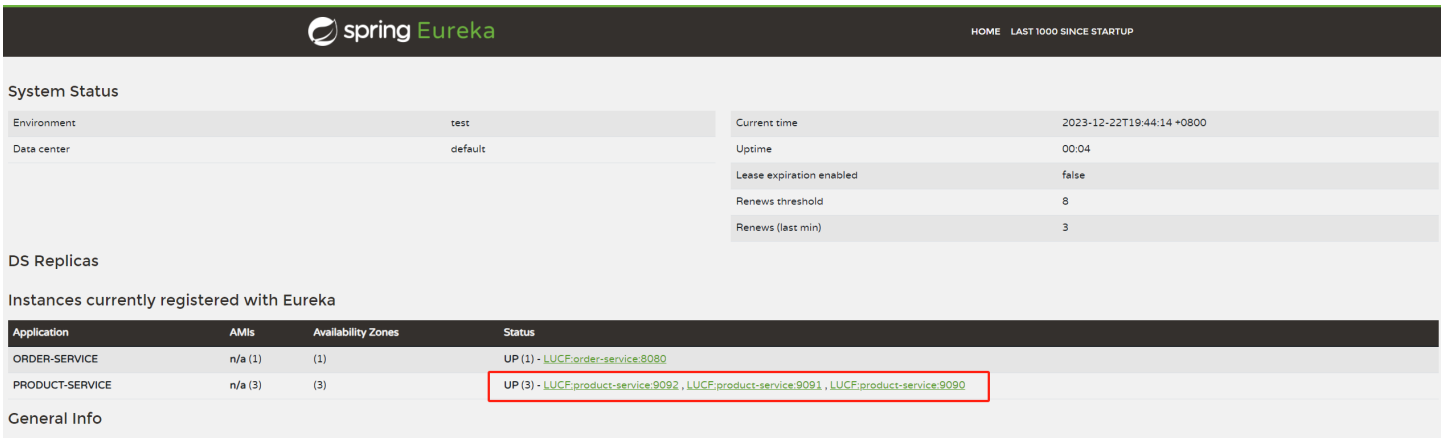
现在IDEA的Service窗口就会多出来一个启动配置, 右键启动服务就可以



同样的操作, 再启动1个实例, 共启动3个服务



观察Eureka, 可以看到product-service下有三个实例:



访问结果:

访问: http://127.0.0.1:8080/order/1

```
11:46:05.684+08:00  INFO 23128 --- [nio-8080-exec-1] com.bite.order.service.OrderService      : LUCF:product-service:9090
11:46:06.435+08:00  INFO 23128 --- [nio-8080-exec-2] com.bite.order.service.OrderService      : LUCF:product-service:9090
11:46:07.081+08:00  INFO 23128 --- [nio-8080-exec-3] com.bite.order.service.OrderService      : LUCF:product-service:9090
```

通过日志可以观察到, 请求多次访问, 都是同一台机器.

这肯定不是我们想要的结果, 我们启动多个实例, 是希望可以分担其他机器的负荷, 那么如何实现呢?

解决方案:

我们可以对上述代码进行简单修改:

```
private static AtomicInteger atomicInteger = new AtomicInteger(1);

private static List<ServiceInstance> instances;
```

```
4
5  @PostConstruct
6  public void init(){
7      //根据应用名称获取服务列表
8      instances = discoveryClient.getInstances("product-service");
9  }
10
11 public OrderInfo selectOrderById(Integer orderId) {
12     OrderInfo orderInfo = orderMapper.selectOrderById(orderId);
13     //String url = "http://127.0.0.1:9090/product/"+ orderInfo.getProductId();
14     //服务可能有多个，轮询获取实例
15     int index = atomicInteger.getAndIncrement() % instances.size();
16     ServiceInstance instance =instances.get(index);
17     log.info(instance.getInstanceId());
18     //拼接url
19     String url = instance.getUri()+"/product/"+ orderInfo.getProductId();
20     ProductInfo productInfo = restTemplate.getForObject(url,
   ProductInfo.class);
21     orderInfo.setProductInfo(productInfo);
22     return orderInfo;
23 }
```

观察日志:

```
1 12:02:13.245+08:00  INFO 1800 --- [nio-8080-exec-1]
  com.bite.order.service.OrderService      : LUCF:product-service:9091
2 12:02:15.723+08:00  INFO 1800 --- [nio-8080-exec-2]
  com.bite.order.service.OrderService      : LUCF:product-service:9090
3 12:02:16.534+08:00  INFO 1800 --- [nio-8080-exec-3]
  com.bite.order.service.OrderService      : LUCF:product-service:9092
4 12:02:16.864+08:00  INFO 1800 --- [nio-8080-exec-4]
  com.bite.order.service.OrderService      : LUCF:product-service:9091
5 12:02:17.078+08:00  INFO 1800 --- [nio-8080-exec-5]
  com.bite.order.service.OrderService      : LUCF:product-service:9090
6 12:02:17.260+08:00  INFO 1800 --- [nio-8080-exec-6]
  com.bite.order.service.OrderService      : LUCF:product-service:9092
7 12:02:17.431+08:00  INFO 1800 --- [nio-8080-exec-7]
  com.bite.order.service.OrderService      : LUCF:product-service:9091
```

通过日志可以看到, **请求被均衡的分配在了不同的实例上, 这就是负载均衡**.

1.2 什么是负载均衡

负载均衡(Load Balance，简称 LB) , 是高并发, 高可用系统必不可少的关键组件.

当服务流量增大时, 通常会采用增加机器的方式进行扩容, 负载均衡就是用来在多个机器或者其他资源中, 按照一定的规则合理分配负载.

> 一个团队最开始只有一个人, 后来随着工作量的增加, 公司又招聘了几个人. 负载均衡就是: 如何把工作量均衡的分配到这几个人身上, 以提高整个团队的效率

1.3 负载均衡的一些实现

上面的例子中, 我们只是简单的对实例进行了轮询, 但真实的业务场景会更加复杂. 比如根据机器的配置进行负载分配, 配置高的分配的流量高, 配置低的分配流量低等.

> 类似企业员工: 能力强的员工可以多承担一些工作.

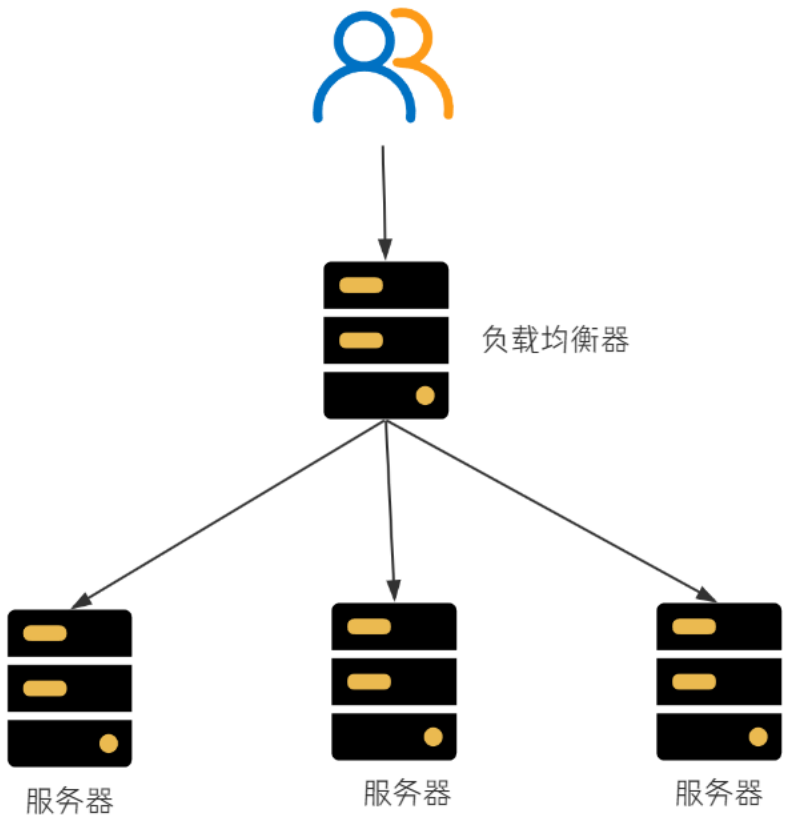
服务多机部署时, 开发人员都需要考虑负载均衡的实现, 所以也出现了一些负载均衡器, 来帮助我们实现负载均衡.

负载均衡分为服务端负载均衡和客户端负载均衡.

服务端负载均衡

在服务端进行负载均衡的算法分配.

比较有名的服务端负载均衡器是Nginx. 请求先到达Nginx负载均衡器, 然后通过负载均衡算法, 在多个服务器之间选择一个进行访问.



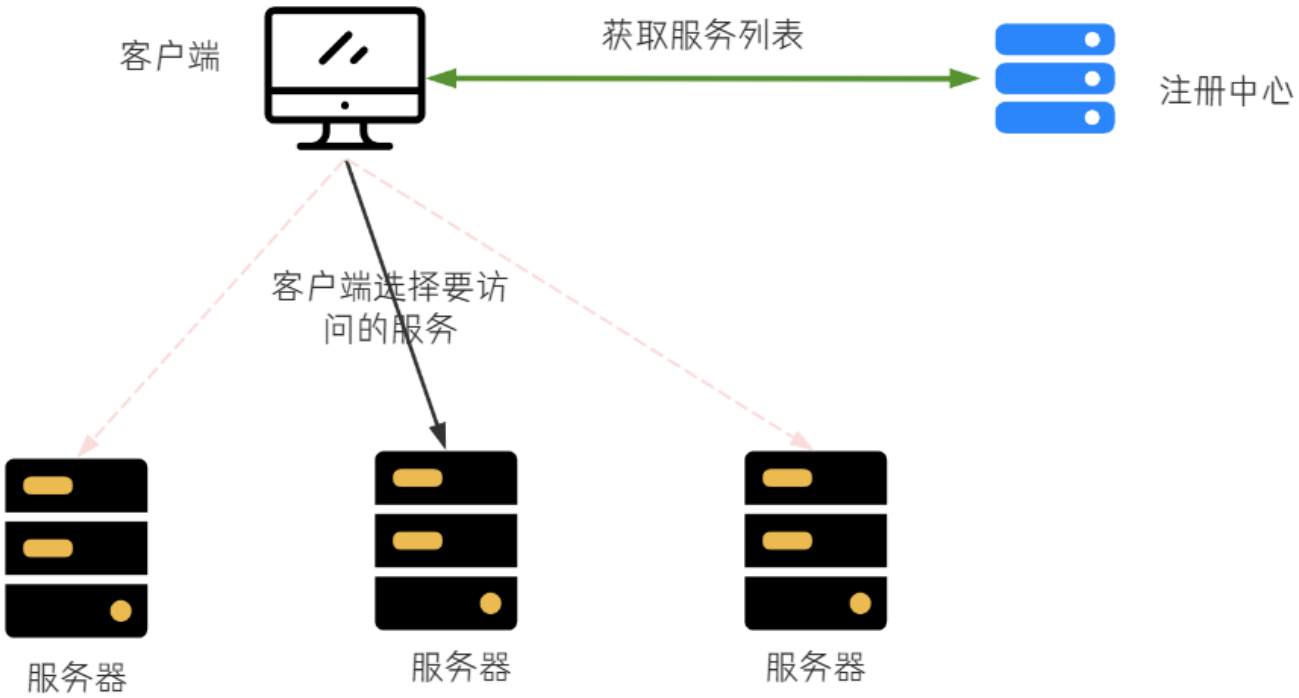
客户端负载均衡

在客户端进行负载均衡的算法分配.

把负载均衡的功能以库的方式集成到客户端, 而不再是由一台指定的负载均衡设备集中提供.

比如Spring Cloud的Ribbon, 请求发送到客户端, 客户端从注册中心(比如Eureka)获取服务列表, 在发送请求前通过负载均衡算法选择一个服务器,然后进行访问.

Ribbon是Spring Cloud早期的默认实现, 由于不维护了, 所以最新版本的Spring Cloud负载均衡集成的是Spring Cloud LoadBalancer(Spring Cloud官方维护)



客户端负载均衡和服务端负载均衡最大的区别在于服务清单所存储的位置

2. Spring Cloud LoadBalancer

2.1 快速上手

SpringCloud 从 2020.0.1 版本开始, 移除了Ribbon 组件，使用Spring Cloud LoadBalancer 组件来代替 Ribbon 实现客户端负载均衡.

2.1.1 使用Spring Cloud LoadBalancer实现负载均衡

1. 给 RestTemplate 这个Bean添加 `@LoadBalanced` 注解就可以

```
@Configuration
public class BeanConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
```

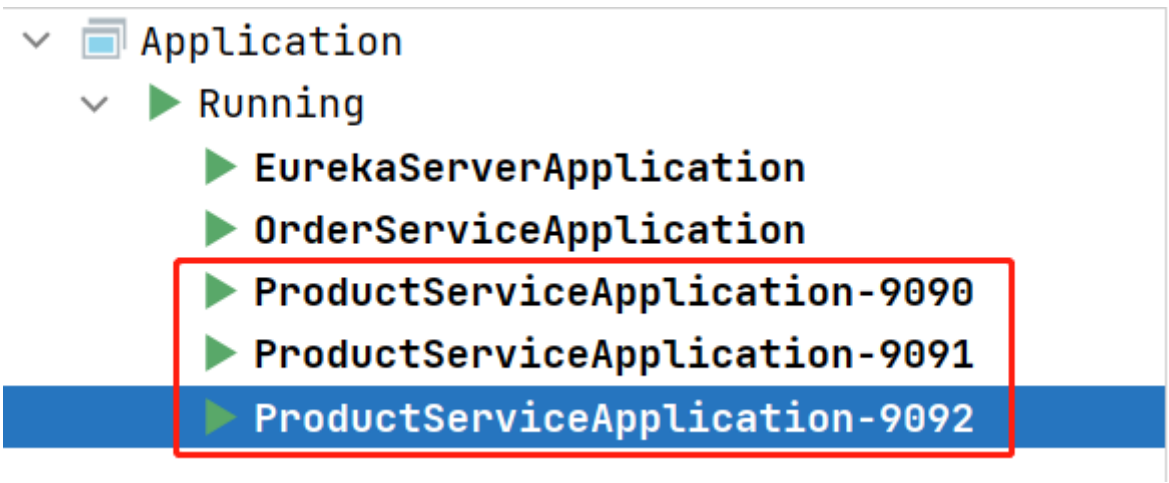
```
8 }
```

2. 修改IP端口号为服务名称

```
1 public OrderInfo selectOrderById(Integer orderId) {
2     OrderInfo orderInfo = orderMapper.selectOrderById(orderId);
3     //String url = "http://127.0.0.1:9090/product/"+ orderInfo.getProductId();
4     String url = "http://product-service/product/"+ orderInfo.getProductId();
5     ProductInfo productInfo = restTemplate.getForObject(url,
  ProductInfo.class);
6     orderInfo.setProductInfo(productInfo);
7     return orderInfo;
8 }
```

2.1.2 启动多个product-service实例

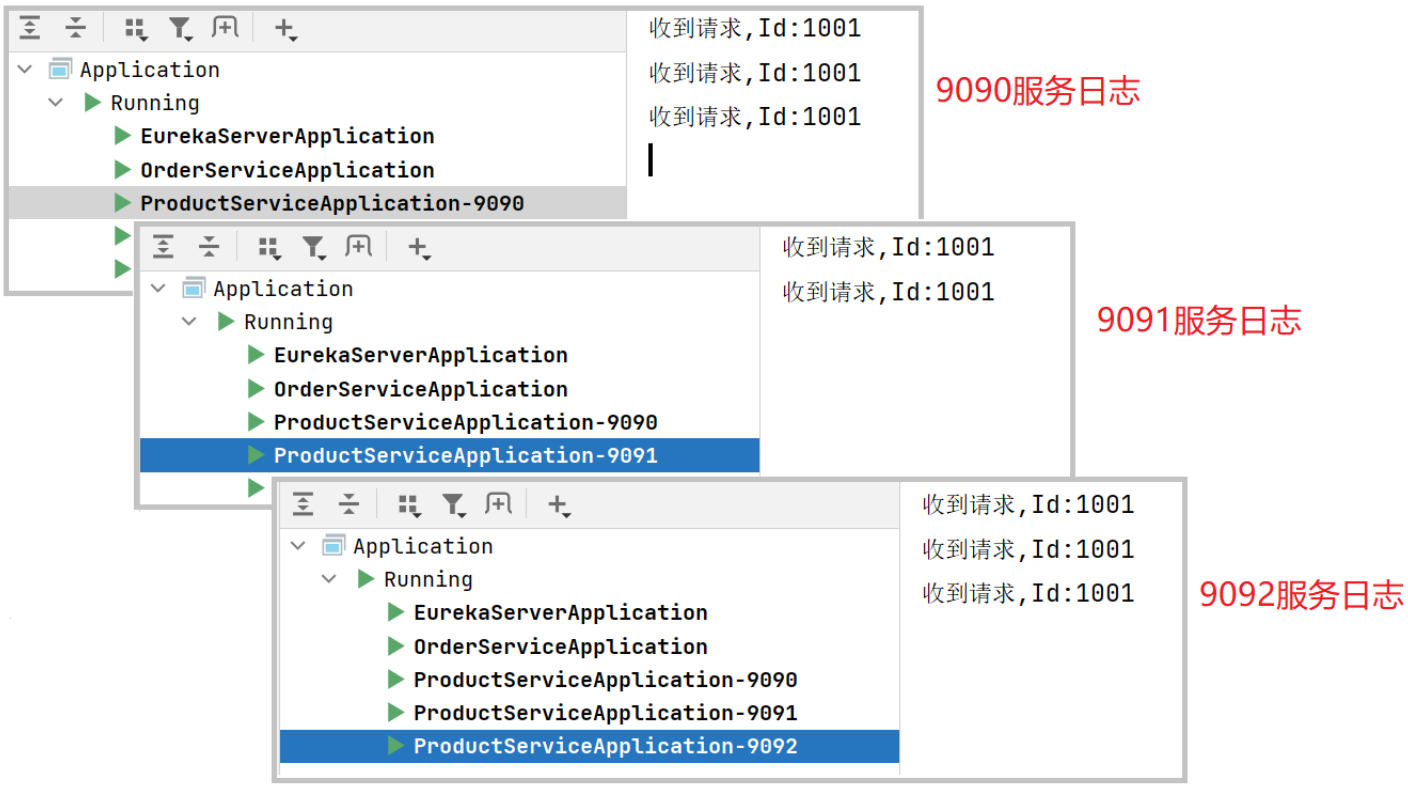
按照上一章节的方式, 启动多个product-service实例



2.1.3 测试负载均衡

连续多次发起请求: http://127.0.0.1:8080/order/1

观察product-service的日志, 会发现请求被分配到这3个实例上了



2.2 负载均衡策略

负载均衡策略是一种思想, 无论是哪种负载均衡器, 它们的负载均衡策略都是相似的. Spring Cloud LoadBalancer 仅支持两种负载均衡策略: 轮询策略 和 随机策略

1. **轮询(Round Robin):** 轮询策略是指服务器轮流处理用户的请求. 这是一种实现最简单, 也最常用的策略. 生活中也有类似的场景, 比如学校轮流值日, 或者轮流打扫卫生.
2. **随机选择（Random):** 随机选择策略是指随机选择一个后端服务器来处理新的请求.

自定义负载均衡策略

Spring Cloud LoadBalancer 默认负载均衡策略是 轮询策略, 实现是 RoundRobinLoadBalancer, 如果服务的消费者如果想采用随机的负载均衡策略, 也非常简单.

> 参考官网地址：Spring Cloud LoadBalancer :: Spring Cloud Commons

1. 定义随机算法对象, 通过 @Bean 将其加载到 Spring 容器中

> 此处使用Spring Cloud LoadBalancer提供的 RandomLoadBalancer

```
1 import org.springframework.cloud.client.ServiceInstance;
2 import org.springframework.cloud.loadbalancer.core.RandomLoadBalancer;
3 import org.springframework.cloud.loadbalancer.core.ReactorLoadBalancer;
4 import org.springframework.cloud.loadbalancer.core.ServiceInstanceListSupplier;
5 import
  org.springframework.cloud.loadbalancer.support.LoadBalancerClientFactory;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.core.env.Environment;
```

```
8
9  public class LoadBalancerConfig {
10      @Bean
11      ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(Environment
   environment,
12
   LoadBalancerClientFactory loadBalancerClientFactory) {
13          String name =
   environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
14          System.out.println("==============="+name);
15          return new
   RandomLoadBalancer(loadBalancerClientFactory.getLazyProvider(name,
   ServiceInstanceListSupplier.class), name);
16      }
17  }
```

💡 注意: 该类需要满足:

1. 不用 `@Configuration` 注释
2. 在组件扫描范围内

2. 使用 `@LoadBalancerClient` 或者 `@LoadBalancerClients` 注解

在 RestTemplate 配置类上方, 使用 @LoadBalancerClient 或 @LoadBalancerClients 注解, 可以对不同的服务提供方配置不同的客户端负载均衡算法策略.

由于咱们项目中只有一个服务提供者, 所以使用@LoadBalancerClient

```
1  @LoadBalancerClient(name = "product-service", configuration =
   LoadBalancerConfig.class)
2  @Configuration
3  public class BeanConfig {
4      @Bean
5      @LoadBalanced
6      public RestTemplate restTemplate(){
7          return new RestTemplate();
8      }
9  }
```

`@LoadBalancerClient` 注解说明

1. name: 该负载均衡策略对哪个服务生效(服务提供方)
2. configuration : 该负载均衡策略 用哪个负载均衡策略实现.

2.3 LoadBalancer 原理

LoadBalancer 的实现, 主要是 `LoadBalancerInterceptor` , 这个类会对 `RestTemplate` 的请求进行拦截, 然后从Eureka根据服务id获取服务列表，随后利用负载均衡算法得到真实的服务地址信息，替换服务id

我们来看看源码实现:

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {
    //...
    public ClientHttpResponse intercept(final HttpRequest request, final
byte[] body, final ClientHttpRequestExecution execution) throws IOException {
        URI originalUri = request.getURI();
        String serviceName = originalUri.getHost();
        Assert.state(serviceName != null, "Request URI does not contain a
valid hostname: " + originalUri);
        return (ClientHttpResponse)this.loadBalancer.execute(serviceName,
this.requestFactory.createRequest(request, body, execution));
    }
}
```

可以看到这里的intercept方法, 拦截了用户的HttpRequest请求，然后做了几件事:

1. `request.getURI()` 从请求中获取uri, 也就是 `http://product-`
2. `originalUri.getHost()` 从uri中获取路径的主机名, 也就是服务id, `product-service`
3. `loadBalancer.execute` 根据服务id, 进行负载均衡, 并处理请求

点进去继续跟踪

```
public class BlockingLoadBalancerClient implements LoadBalancerClient {

    public <T> T execute(String serviceId, LoadBalancerRequest<T> request)
throws IOException {
        String hint = this.getHint(serviceId);
        LoadBalancerRequestAdapter<T, TimedRequestContext> lbRequest = new
LoadBalancerRequestAdapter(request, this.buildRequestContext(request, hint));
        Set<LoadBalancerLifecycle> supportedLifecycleProcessors =
this.getSupportedLifecycleProcessors(serviceId);
        supportedLifecycleProcessors.forEach((lifecycle) -> {
            lifecycle.onStart(lbRequest);
        });
        //根据serviceId,和负载均衡策略, 选择处理的服务
        ServiceInstance serviceInstance = this.choose(serviceId, lbRequest);
```

```
        if (serviceInstance == null) {
            supportedLifecycleProcessors.forEach((lifecycle) -> {
                lifecycle.onComplete(new CompletionContext(Status.DISCARD, lbRequest, new EmptyResponse()));
            });
            throw new IllegalStateException("No instances available for " + serviceId);
        } else {
            return this.execute(serviceId, serviceInstance, lbRequest);
        }
    }


    /**
     * 根据serviceId,和负载均衡策略, 选择处理的服务
     *
     */
    public <T> ServiceInstance choose(String serviceId, Request<T> request) {
        //获取负载均衡器
        ReactiveLoadBalancer<ServiceInstance> loadBalancer = this.loadBalancerClientFactory.getInstance(serviceId);
        if (loadBalancer == null) {
            return null;
        } else {
            //根据负载均衡算法, 在列表中选择一个服务实例
            Response<ServiceInstance> loadBalancerResponse = (Response)Mono.from(loadBalancer.choose(request)).block();
            return loadBalancerResponse == null ? null : (ServiceInstance)loadBalancerResponse.getServer();
        }
    }
}
```

3. 服务部署(Linux)

接下来我们把服务部署在Linux系统上

3.1 准备数据

安装mysql

参考 MySQL安装

数据初始化

参考环境搭建课件-数据准备SQL

修改配置文件

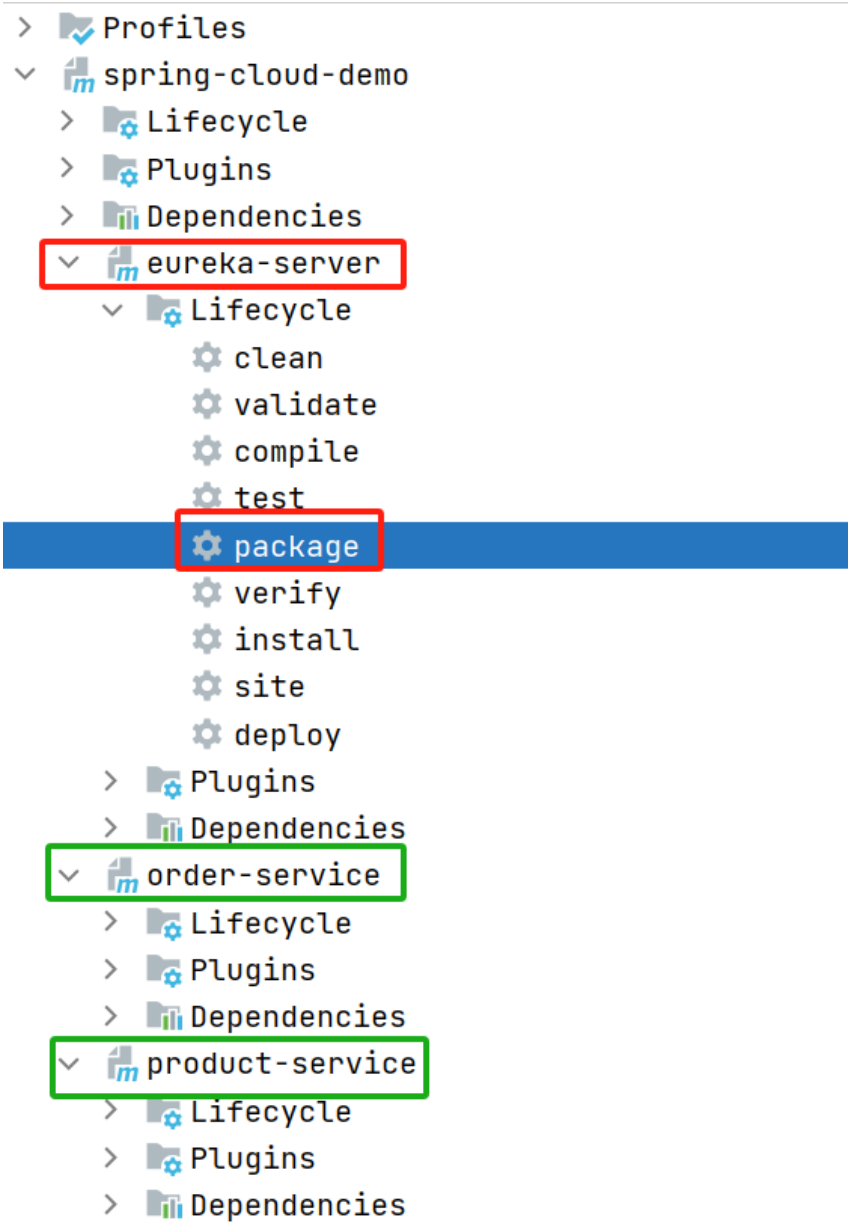
修改配置文件中, 数据库的密码

3.2 服务构建打包

采用Maven打包, 需要对3个服务分别打包:

eureka-server, order-service, product-service

1. 打包方式和SpringBoot项目一致, 依次对三个项目打包即可.



3.3 启动服务

1. 上传Jar包到云服务器

> 第一次上传需要安装lrzsz

```
1 apt install lrzsz
```

直接拖动文件到xshell窗口, 上传成功.

2. 启动服务

```
#后台启动eureka-server, 并设置输出日志到logs/eureka.log
nohup java -jar eureka-server.jar >logs/eureka.log &

#后台启动order-service, 并设置输出日志到logs/order.log
nohup java -jar order-service.jar >logs/order.log &

#后台启动product-service, 并设置输出日志到logs/order.log
nohup java -jar product-service.jar >logs/product-9090.log &
```

再多启动两台product-service实例

```
#启动实例, 指定端口号为9091
nohup java -jar product-service.jar --server.port=9091 >logs/product-9091.log &

#启动实例, 指定端口号为9092
nohup java -jar product-service.jar --server.port=9092 >logs/product-9092.log &
```

3.4 开放端口号

根据自己项目设置的情况, 在云服务器上开放对应的端口号

> 不同的服务器厂商, 开放端口号的入口不同, 需要自行找一找或者咨询对应的客服人员.

以腾讯云服务器举例:

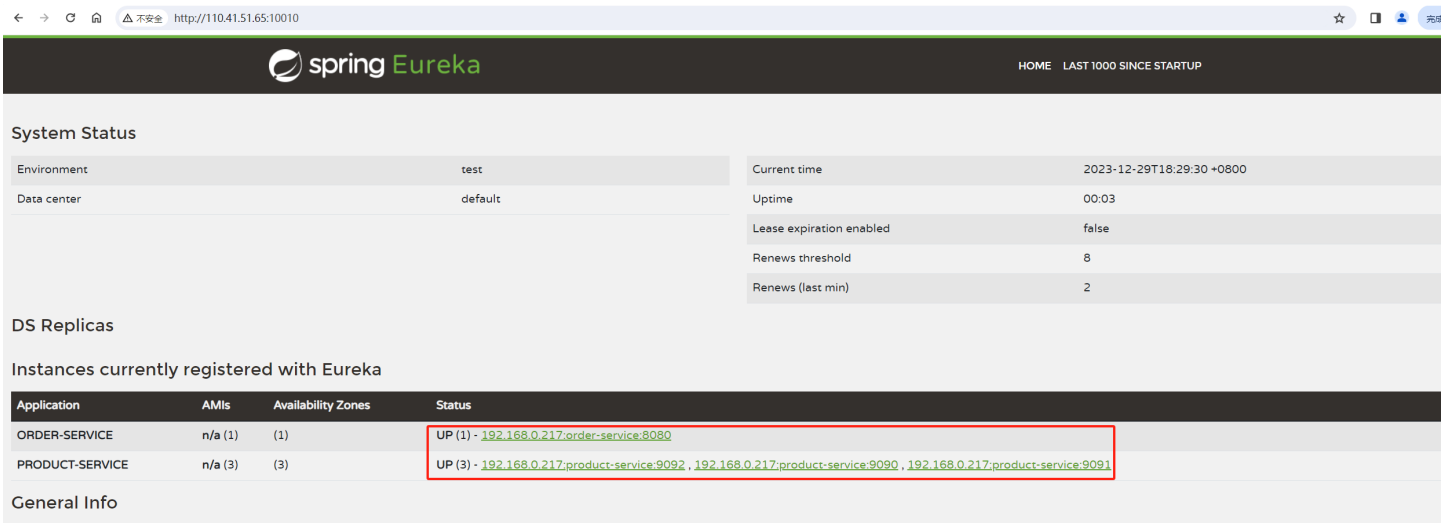
1) 进入防火墙管理页面

2) 添加规则

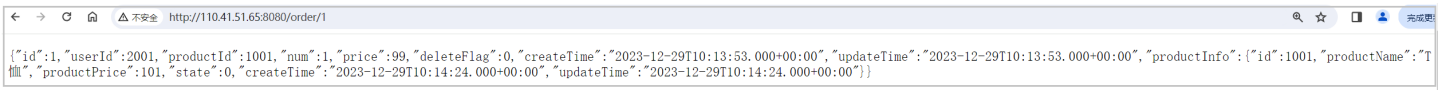
端口号写需要开放的端口号, 多个端口号以逗号分割.

3.5 测试

1. 访问Eureka Server:



2. 访问订单服务接口: http://110.41.51.65:8080/order/1



远程调用成功.