

1. 背景

1.1 问题描述

上个章节的例子中可以看到, 远程调用时, 我们的URL是写死的

```
1 String url = "http://127.0.0.1:9090/product/"+ orderInfo.getProductId();
```

当更换机器, 或者新增机器时, 这个URL就需要跟着变更, 就需要去通知所有的相关服务去修改. 随之而来的就是各个项目的配置文件反复更新, 各个项目的频繁部署. 这种没有具体意义, 但又不得不做的工作, 会让人非常痛苦.

1.2 解决思路

试想生活中的场景:

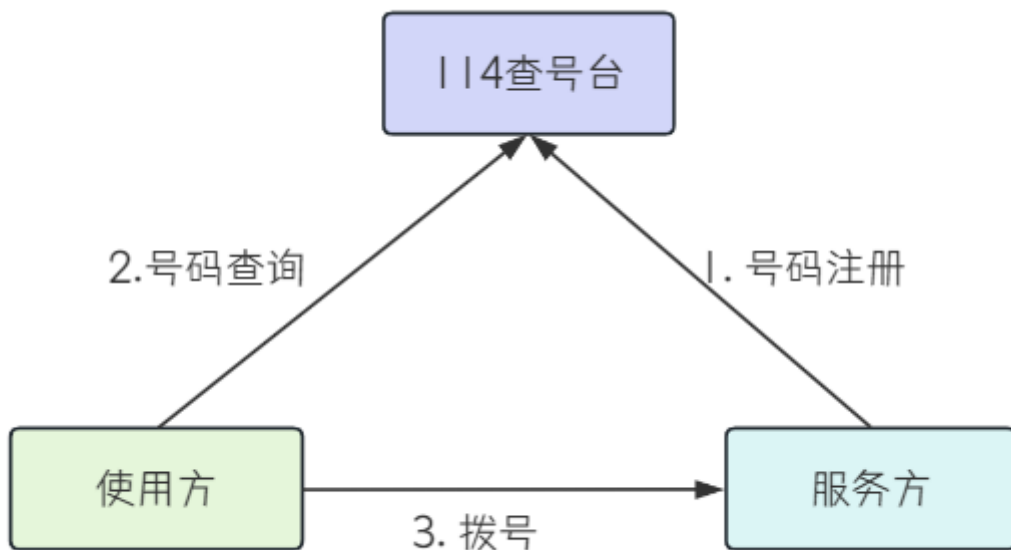
我们生活中, 避免不了和各个机构(医院, 学校, 政府部门等)打交道, 就需要保存各个机构的电话号码. 如果机构换了电话号码, 就需要通知各个使用方, 但是这些机构的使用方群体是巨大的, 没办法做到一一通知, 怎么处理呢?

机构电话如果发生变化, 通知114. 用户需要联系机构时, 先打114查询电话, 然后再联系各个机构.

114查号台的作用主要有两个:

号码注册: 服务方把电话上报给114

号码查询: 使用方通过114可以查到对应的号码



同样的, 微服务开发时, 也可以采用类似的方案.

服务启动/变更时, 向**注册中心**报道. 注册中心记录应用和IP的关系.

调用方调用时, 先去**注册中心**获取服务方的IP, 再去服务方进行调用.

1.3 什么是注册中心

在最初的架构体系中, 集群的概念还不那么流行, 且机器数量也比较少, 此时直接使用DNS+Nginx就可以满足几乎所有服务的发现. 相关的注册信息直接配置在Nginx. 但是随着微服务的流行与流量的激增, 机器规模逐渐变大, 并且机器会有频繁的上下线行为, 这种时候需要运维手动地去维护这个配置信息是一个很麻烦的操作. 所以开发者们开始希望有这么一个东西, 它能维护一个服务列表, 哪个机器上线了, 哪个机器宕机了, 这些信息都会自动更新到服务列表上, 客户端拿到这个列表, 直接进行服务调用即可. 这个就是注册中心.

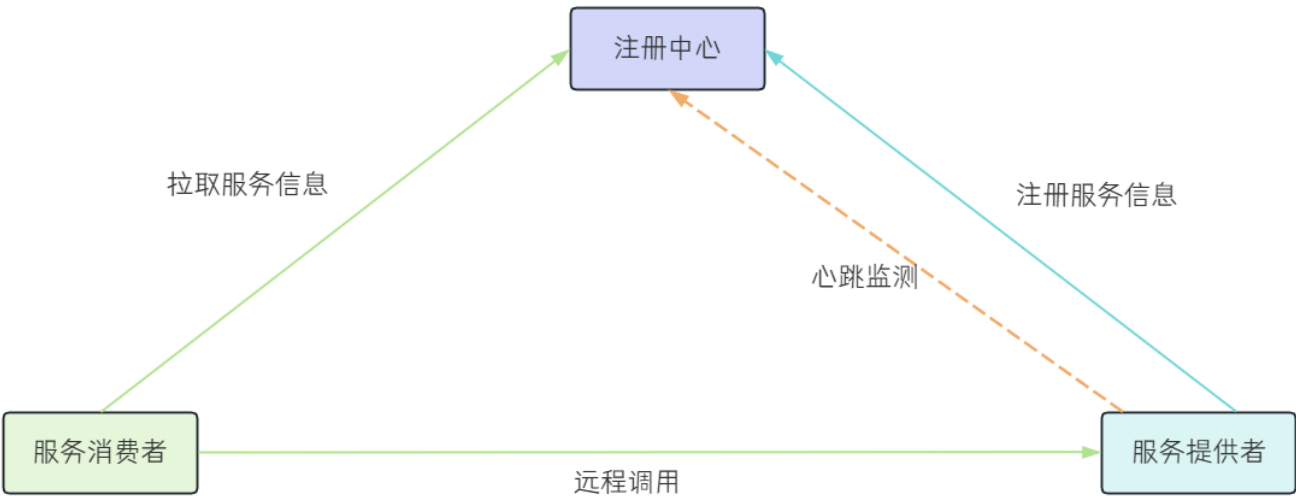
注册中心主要有三种角色:

- **服务提供者(Server)**: 一次业务中, 被其它微服务调用的服务. 也就是提供接口给其它微服务.
- **服务消费者(Client)**: 一次业务中, 调用其它微服务的服务. 也就是调用其它微服务提供的接口.
- **服务注册中心(Registry)**: 用于保存Server 的注册信息, 当Server 节点发生变更时, Registry 会同步变更. 服务与注册中心使用一定机制通信, 如果注册中心与某服务长时间无法通信, 就会注销该实例.

他们之间的关系以及工作内容, 可以通过两个概念来描述:

服务注册: 服务提供者在启动时, 向 Registry 注册自身服务, 并向 Registry 定期发送心跳汇报存活状态.

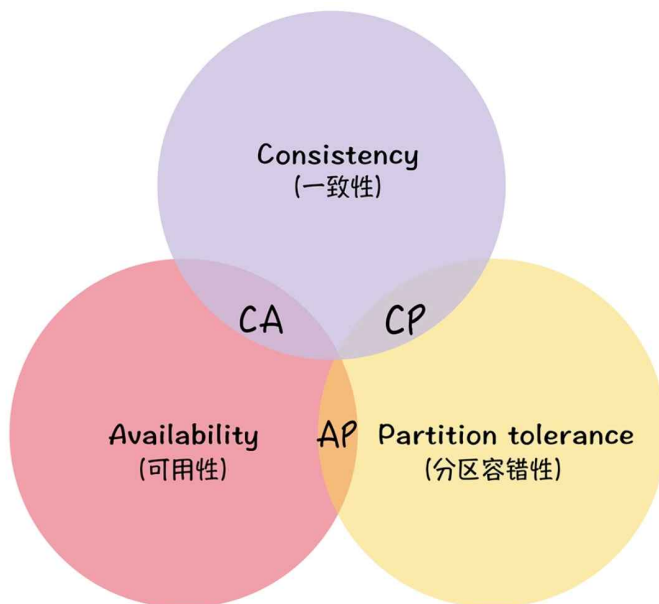
服务发现: 服务消费者从注册中心查询服务提供者的地址, 并通过该地址调用服务提供者的接口. 服务发现的一个重要作用就是提供给服务消费者一个可用的服务列表.



1.4 CAP理论

谈到注册中心, 就避不开CAP理论.

CAP 理论是分布式系统设计中最基础, 也是最为关键的理论.



- **一致性(Consistency)** CAP理论中的一致性, 指的是强一致性. 所有节点在同一时间具有相同的数据
- **可用性(Availability)** 保证每个请求都有响应(响应结果可能不对)
- **分区容错性(Partition Tolerance)** 当出现网络分区后, 系统仍然能够对外提供服务

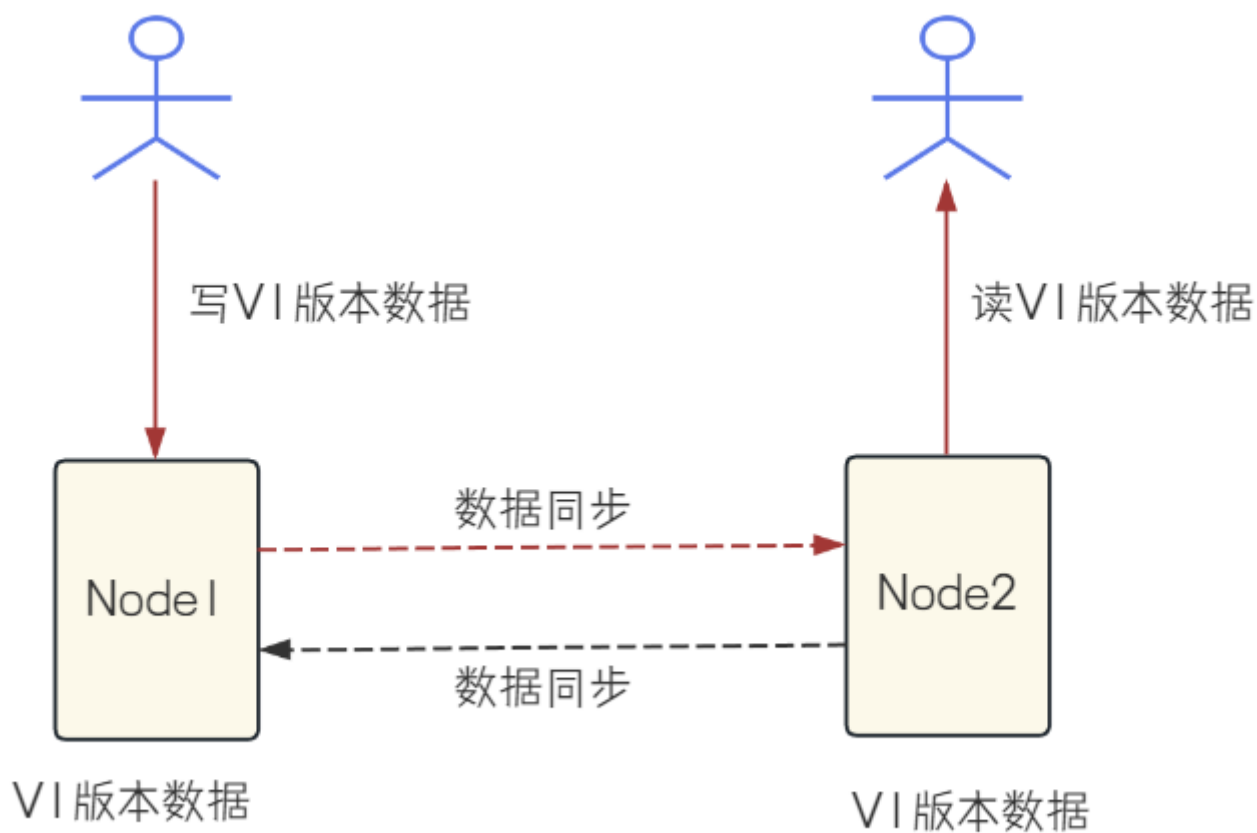
一个部门全国各地都有岗位, 这时候, 总部下发了一个通知, 由于通知需要开会周知全员, 当有客户咨询时:

1. 所有成员对客户的回应结果都是一致的(一致性)
2. 客户咨询时, 一定有回应(可用性)
3. 当其中一个成员休假时, 这个部门的其他成员也可以对客户提供咨询服务(分区容错性)

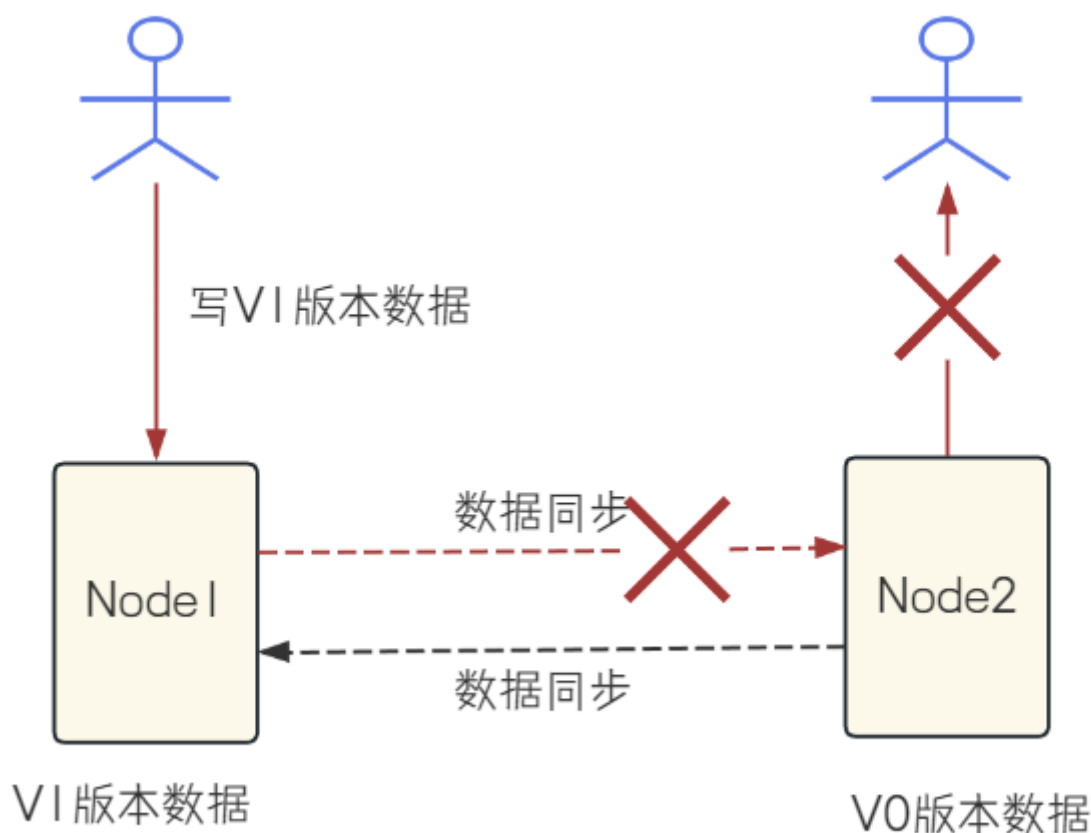
CAP 理论告诉我们: 一个分布式系统不可能同时满足数据一致性, 服务可用性和分区容错性这三个基本需求, 最多只能同时满足其中的两个.

在**分布式系统**中, 系统间的网络不能100%保证健康, 服务又必须对外保证服务. 因此Partition Tolerance不可避免. 那就只能在C和A中选择一个. 也就是CP或者AP架构

正常情况:



网络异常:



CP架构: 为了保证分布式系统对外的数据一致性, 于是选择不返回任何数据

AP架构: 为了保证分布式系统的可用性, 节点2返回V0版本的数据(即使这个数据不正确)

更多参考: <https://cloud.tencent.com/developer/article/1860632>

1.5 常见的注册中心

1. Zookeeper

Zookeeper的官方并没有说它是一个注册中心, 但是国内Java体系, 大部分的集群环境都是依赖Zookeeper来完成注册中心的功能.

2. Eureka

Eureka是Netflix开发的基于REST的服务发现框架, 主要用于服务注册, 管理, 负载均衡和服务故障转移.

官方声明在Eureka2.0版本停止维护, 不建议使用. 但是Eureka是SpringCloud服务注册/发现的默认实现, 所以目前还是有很多公司在使用.

3. Nacos

Nacos是Spring Cloud Alibaba架构中重要的组件, 除了服务注册, 服务发现功能之外, Nacos还支持配置管理, 流量管理, DNS, 动态DNS等多种特性.

CAP理论对比

| | Zookeeper | Eureka | Nacos |
|-------|-----------|--------|---------------|
| CAP理论 | CP | AP | CP或AP 默认AP |

在分布式环境中, 即使拿到一个错误的的数据, 也胜过无法提供实例信息而造成请求失败要好(比如淘宝 11.11, 京东618都是谨遵AP原则)

ka咱们课堂中, 会给大家介绍Eureka和Nacos的使用.

2. Eureka 介绍

Eureka是Netflix OSS套件中关于服务注册和发现的解决方案. Spring Cloud对Eureka进行了集成, 并作为优先推荐方案进行宣传, 虽然目前Eureka 2.0已经停止维护, 新的微服务架构设计中, 也不再建议使用, 但是目前依然有大量公司的微服务系统使用Eureka作为注册中心.

官方文档: <https://github.com/Netflix/eureka/wiki>

Eureka主要分为两个部分:

- Eureka Server: 作为注册中心Server端, 向微服务应用程序提供服务注册, 发现, 健康检查等能力.
- Eureka Client: 服务提供者, 服务启动时, 会向Eureka Server 注册自己的信息(IP,端口,服务信息等),Eureka Server 会存储这些信息

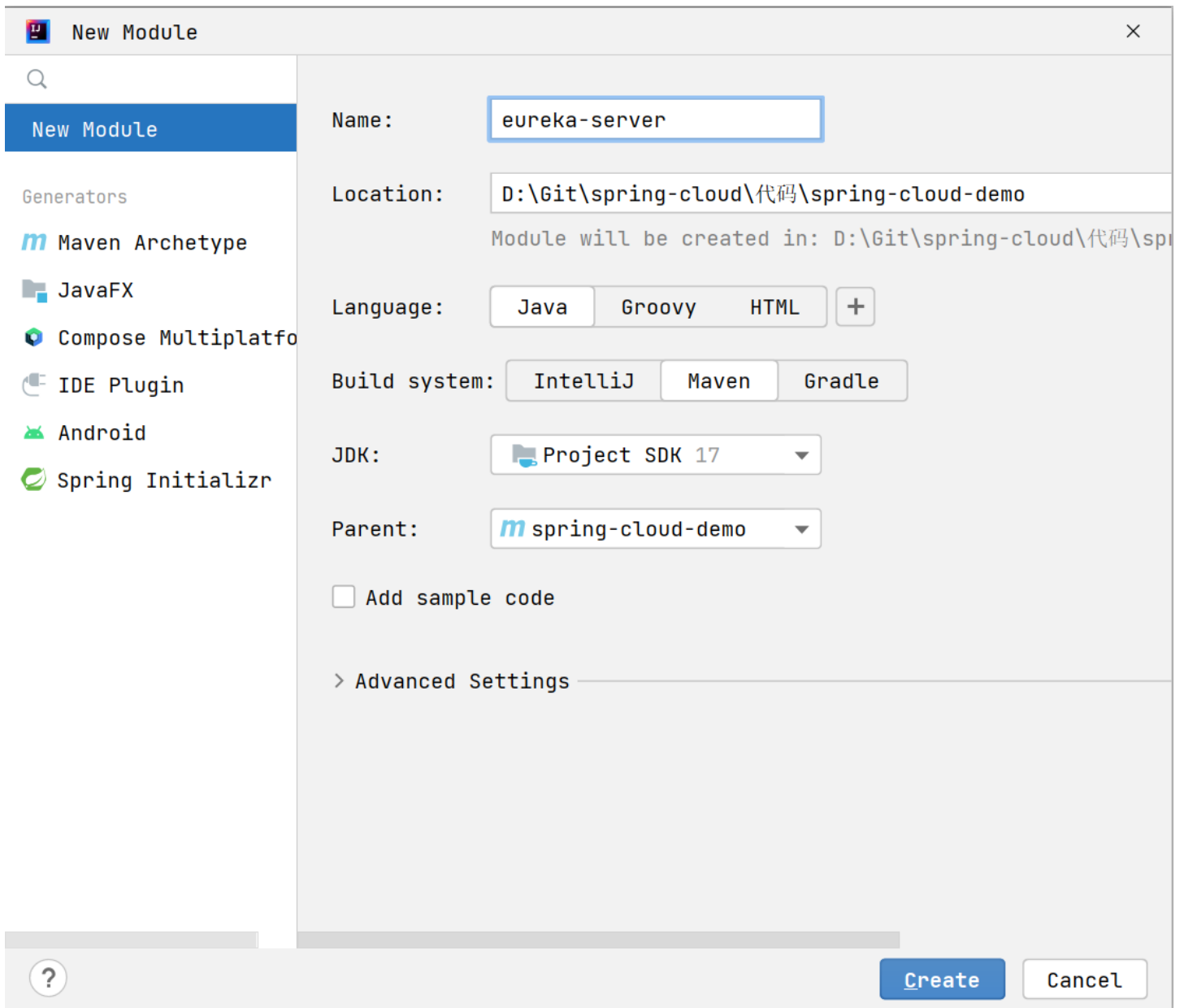
关于Eureka的学习, 主要包含以下三个部分:

1. 搭建Eureka Server
2. 将order-service, product-service 都注册到Eureka
3. order-service远程调用时, 从Eureka中获取product-service的服务列表, 然后进行交互

3. 搭建Eureka Server

Eureka-server 是一个独立的微服务.

3.1 创建Eureka-server 子模块



3.2 引入eureka-server依赖

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
4 </dependency>
```

3.3 项目构建插件

```
1 <build>
2     <plugins>
3         <plugin>
4             <groupId>org.springframework.boot</groupId>
5             <artifactId>spring-boot-maven-plugin</artifactId>
```

```
6         </plugin>
7     </plugins>
8 </build>
```

3.4 完善启动类

给该项目编写一个启动类,并在启动类上添加 `@EnableEurekaServer` 注解,开启eureka注册中心服务

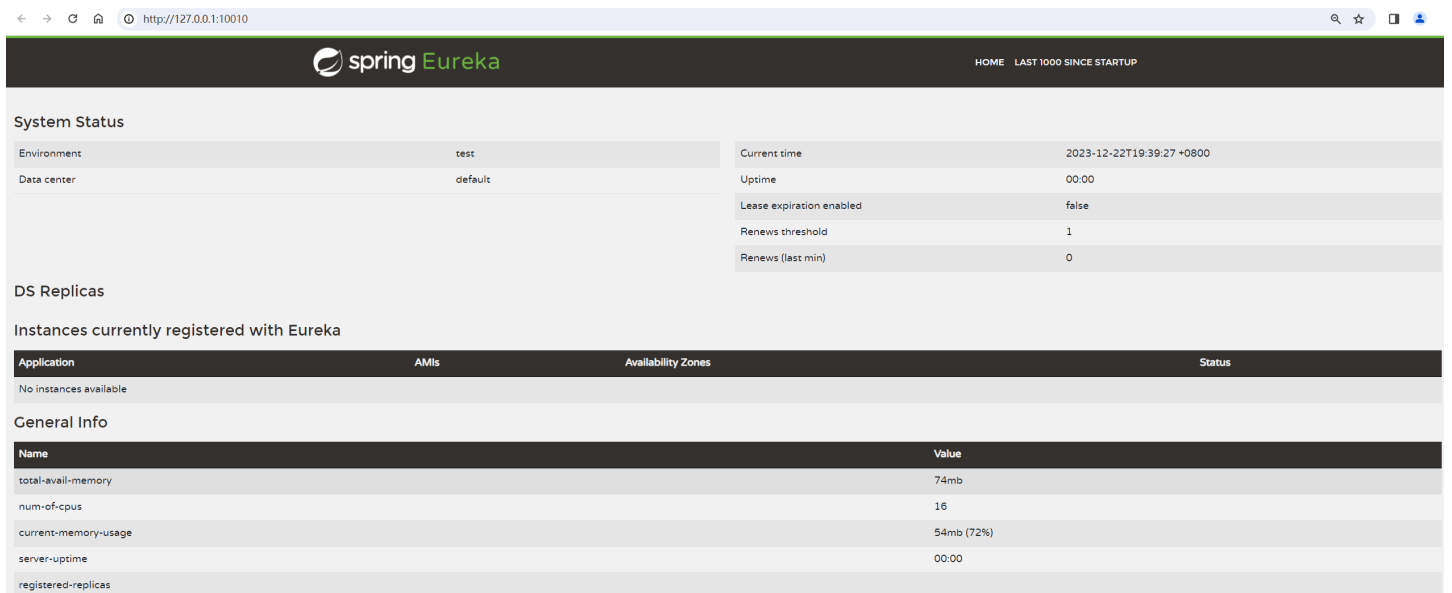
```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3
4 @EnableEurekaServer
5 @SpringBootApplication
6 public class EurekaServerApplication {
7     public static void main(String[] args) {
8         SpringApplication.run(EurekaServerApplication.class, args);
9     }
10 }
```

3.5 编写配置文件

```
1 server:
2   port: 10010
3 spring:
4   application:
5     name: eureka-server
6 eureka:
7   instance:
8     hostname: localhost
9   client:
10    fetch-registry: false # 表示是否从Eureka Server获取注册信息,默认为true.因为这是一个单点的Eureka Server,不需要同步其他的Eureka Server节点的数据,这里设置为false
11    register-with-eureka: false # 表示是否将自己注册到Eureka Server,默认为true.由于当前应用就是Eureka Server,故而设置为false.
12    service-url:
13      # 设置与Eureka Server的地址,查询服务和注册服务都需要依赖这个地址.
14      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

3.6 启动服务

启动服务, 访问注册中心: <http://127.0.0.1:10010/>



The screenshot shows the Spring Eureka web interface. The top navigation bar includes the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details.

| Environment | test |
|-------------|---------|
| Data center | default |
- Current time:** 2023-12-22T19:39:27 +0800
- Uptime:** 00:00
- Lease expiration enabled:** false
- Renews threshold:** 1
- Renews (last min):** 0
- DS Replicas:** (Empty list)
- Instances currently registered with Eureka:** A table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status'. It shows 'No instances available'.
- General Info:** A table with columns 'Name' and 'Value'.

| Name | Value |
|----------------------|------------|
| total-avail-memory | 74mb |
| num-of-cpus | 16 |
| current-memory-usage | 54mb (72%) |
| server-uptime | 00:00 |
| registered-replicas | |

可以看到eureka-server已经启动成功了

4. 服务注册

接下来我们把product-service 注册到eureka-server中

4.1 引入eureka-client依赖

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
```

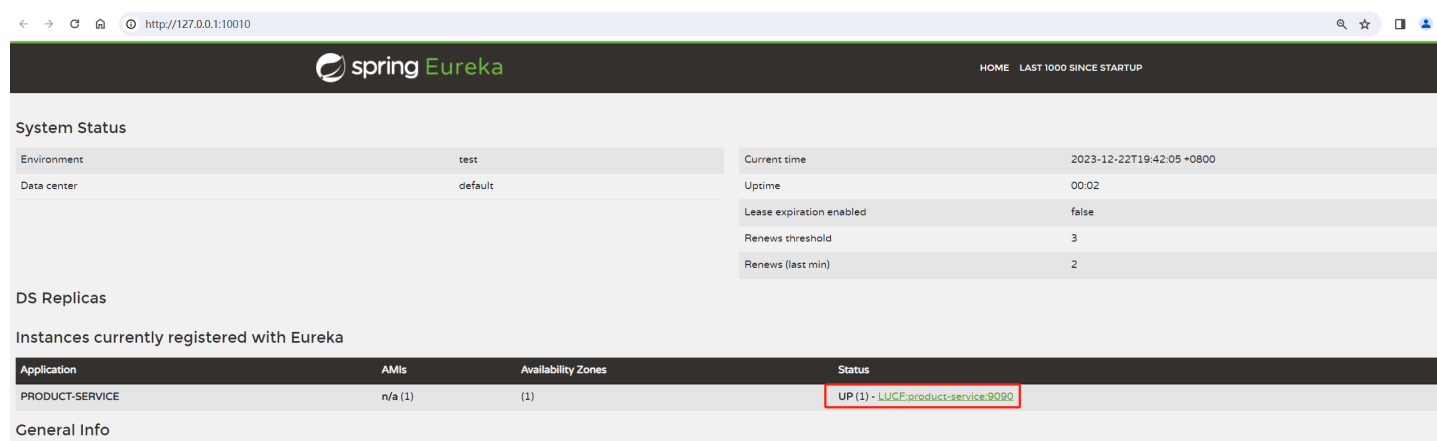
4.2 完善配置文件

添加服务名称和eureka地址

```
1 spring:
2   application:
3     name: product-service
4 eureka:
5   client:
6     service-url:
7       defaultZone: http://127.0.0.1:10010/eureka
```

4.3 启动服务

刷新注册中心: <http://127.0.0.1:10010/>



The screenshot shows the Spring Eureka web interface. At the top, there's a navigation bar with the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section displays various metrics in a table format. To the right, there's a table for 'Instances currently registered with Eureka' with columns for Application, AMIs, Availability Zones, and Status. The 'PRODUCT-SERVICE' instance is listed with a status of 'UP (1) - LUCEproduct-service:9090', which is highlighted with a red box. Below the instances table is a 'General Info' section.

| System Status | |
|--------------------------|---------------------------|
| Environment | test |
| Data center | default |
| Current time | 2023-12-22T19:42:05 +0800 |
| Uptime | 00:02 |
| Lease expiration enabled | false |
| Renews threshold | 3 |
| Renews (last min) | 2 |

| Application | AMIs | Availability Zones | Status |
|-----------------|---------|--------------------|-----------------------------------|
| PRODUCT-SERVICE | n/a (1) | (1) | UP (1) - LUCEproduct-service:9090 |

可以看到product-service已经注册到 eureka上了

5. 服务发现

接下来我们修改order-service, 在远程调用时, 从eureka-server拉取product-service的服务信息, 实现服务发现

5.1 引入依赖

服务注册和服务发现都封装在eureka-client依赖中, 所以服务发现时, 也是引入eureka-client依赖

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
```

5.2 完善配置文件

服务发现也需要知道eureka地址, 因此配置内容依然与服务注册一致, 都是配置eureka信息

```
1 spring:
2   application:
3     name: product-service
4 eureka:
5   client:
6     service-url:
7       defaultZone: http://127.0.0.1:10010/eureka
```

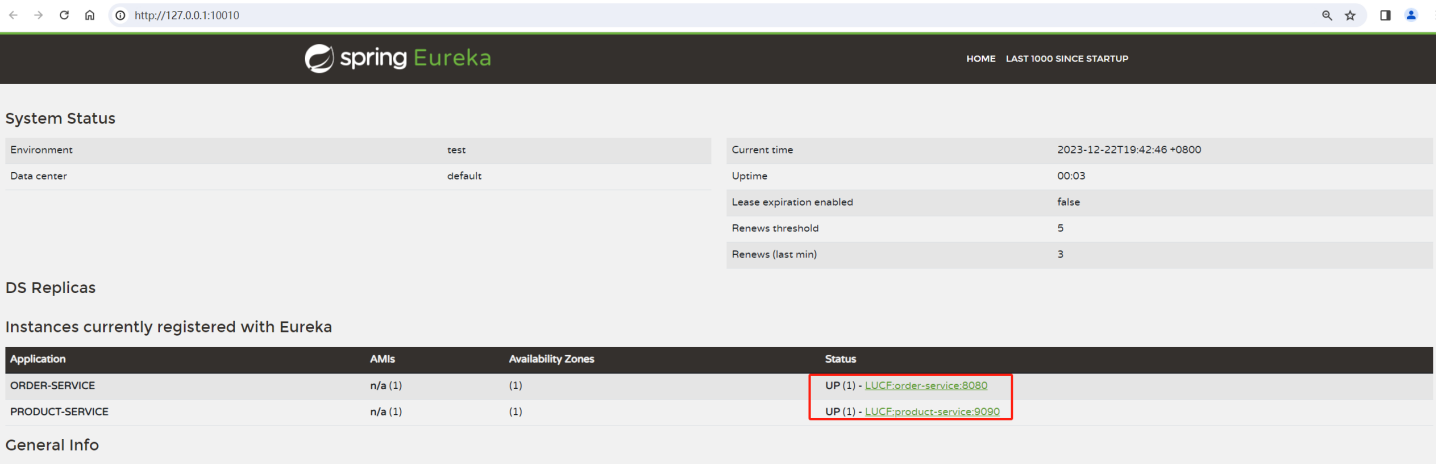
5.3 远程调用

远程调用时, 我们需要从eureka-server中获取product-service的列表(可能存在多个服务), 并选择其中一个进行调用

```
1 import com.bite.order.mapper.OrderMapper;
2 import com.bite.order.model.OrderInfo;
3 import com.bite.order.model.ProductInfo;
4 import org.springframework.cloud.client.ServiceInstance;
5 import org.springframework.cloud.client.discovery.DiscoveryClient;
6 import jakarta.annotation.Resource;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.cloud.netflix.eureka.EurekaServiceInstance;
9 import org.springframework.stereotype.Service;
10 import org.springframework.web.client.RestTemplate;
11
12 import java.util.List;
13
14 @Slf4j
15 @Service
16 public class OrderService {
17     @Autowired
18     private OrderMapper orderMapper;
19
20     @Resource
21     private DiscoveryClient discoveryClient;
22     @Autowired
23     private RestTemplate restTemplate;
24
25     public OrderInfo selectOrderById(Integer orderId) {
26         OrderInfo orderInfo = orderMapper.selectOrderById(orderId);
27         //String url = "http://127.0.0.1:9090/product/"+
        orderInfo.getProductId();
28         //根据应用名称获取服务列表
29         List<ServiceInstance> instances =
        discoveryClient.getInstances("product-service");
30         //服务可能有多个, 获取第一个
31         EurekaServiceInstance instance = (EurekaServiceInstance)
        instances.get(0);
32         log.info(instance.getInstanceId());
33         //拼接url
34         String url = instance.getUri()+"/product/"+ orderInfo.getProductId();
35         ProductInfo productInfo = restTemplate.getForObject(url,
        ProductInfo.class);
36         orderInfo.setProductInfo(productInfo);
37         return orderInfo;
38     }
39 }
```

5.4 启动服务

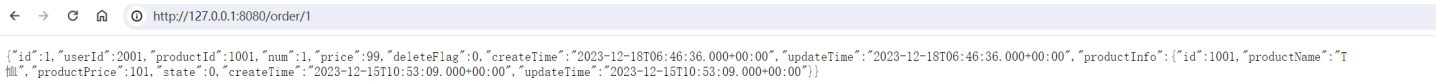
刷新注册中心: <http://127.0.0.1:10010/>



可以看到order-service已经注册到 eureka上了

访问接口: <http://127.0.0.1:8080/order/1>

可以看到, 远程调用也成功了.



6. Eureka 和Zookeeper区别

Eureka和Zookeeper都是用于服务注册和发现的工具,区别如下:

1. Eureka是Netflix开源的项目, 而Zookeeper是Apache开源的项目.
2. Eureka 基于AP原则, 保证高可用, Zookeeper基于CP原则, 保证数据一致性.
3. Eureka 每个节点 都是均等的, Zookeeper的节点区分Leader 和Follower 或 Observer, 也正因为这个原因, 如果Zookeeper的Leader发生故障时, 需要重新选举, 选举过程集群会有短暂时间的不可用.