SpringCloud01-课堂笔记

一、微服务介绍

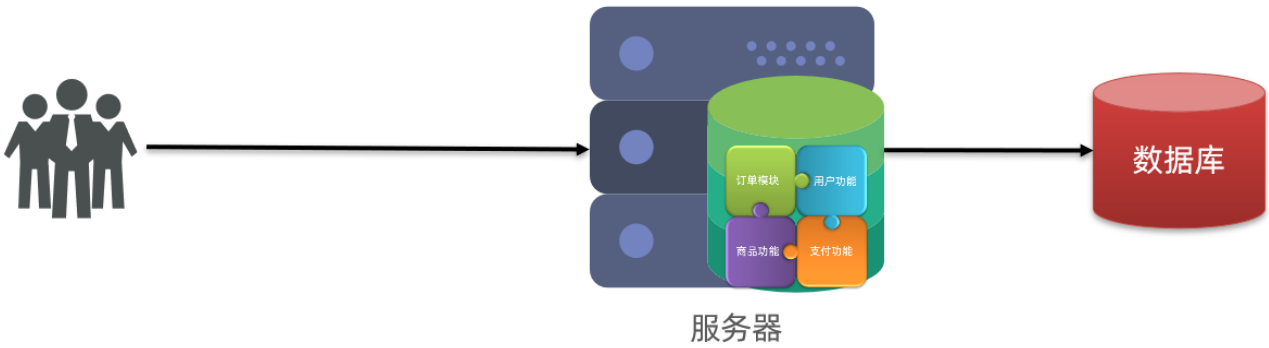
本章节学习目标：

- [ ] 什么是微服务，微服务有哪些特征
- [ ] SpringCloud是什么
- [ ] SpringCloud与Dubbo的区别

1. 系统架构的演变

1.1 单体架构

将业务的所有功能集中在一个项目中开发，打成一个包部署。当网站流量很小时，单体架构非常合适。



单体架构的优缺点如下：

优点：

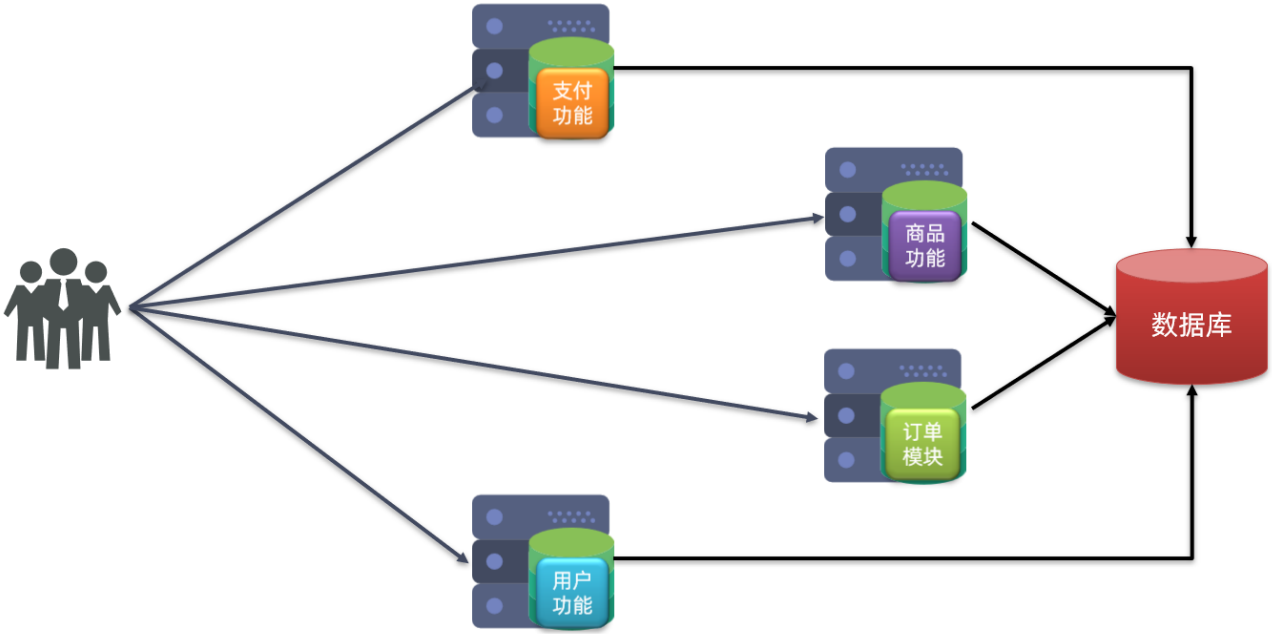
- 架构简单
- 部署成本低

缺点：

- 耦合度高（维护困难、升级困难）

1.2 分布式服务

分布式架构：根据业务功能对系统做拆分，每个业务功能模块作为独立项目开发，称为一个服务。



分布式架构的优缺点：

优点：

- 降低服务耦合
- 有利于服务升级和拓展

缺点：

- 服务调用关系错综复杂
- 服务容错性较差

1.3 微服务

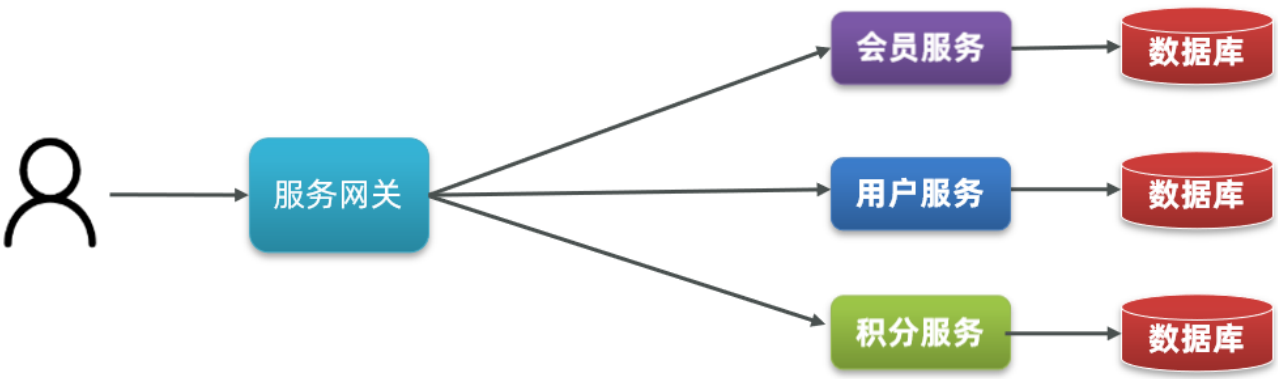
微服务其实是在给分布式架构制定一个标准，进一步降低服务之间的耦合度，提供服务的独立性和灵活性。做到高内聚，低耦合。

因此，可以认为**微服务**是一种经过良好架构设计的**分布式架构方案**，微服务以如下特征：

- 单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责。一个服务只做一件事
- 服务自治：团队独立、技术独立、数据独立，独立部署和交付
- 面向服务：服务对外暴露统一标准的接口，与语言和技术无关。SpringCloud体系里使用HTTP协议的接口
- 隔离性强：服务调用做好隔离、容错、降级，避免出现级联问题

但是使用了微服务，也会带来一些新的问题(相对于单体架构来说)：

- 增加了系统间的通信成本
- 增加了数据一致性问题，分布式事务问题等等
- 服务数量增加，运维压力大



一旦采用微服务系统架构，就势必会遇到这样几个问题：

- 这么多小服务，如何管理他们的地址？**服务治理**的问题，可以使用"注册中心"来解决
- 这么多小服务，他们之间如何通讯？**远程调用**，可以使用httpclient、RestTemplate、**OpenFeign**(优雅的远程调用技术)来解决
- 这么多小服务，客户端怎么访问他们？要使用网关实现统一的对外访问入口，**Gateway**来解决
- 这么多小服务，一旦出现问题了，应该如何自处理，防止问题扩大？要实现服务的隔离防止雪崩，Hystrix或者**Sentinel**等解决
- 这么多小服务，每个服务都有各自的配置文件，如果需要修改配置太麻烦？使用**配置中心**统一管理所有配置文件。SpringCloudConfig或**Nacos**

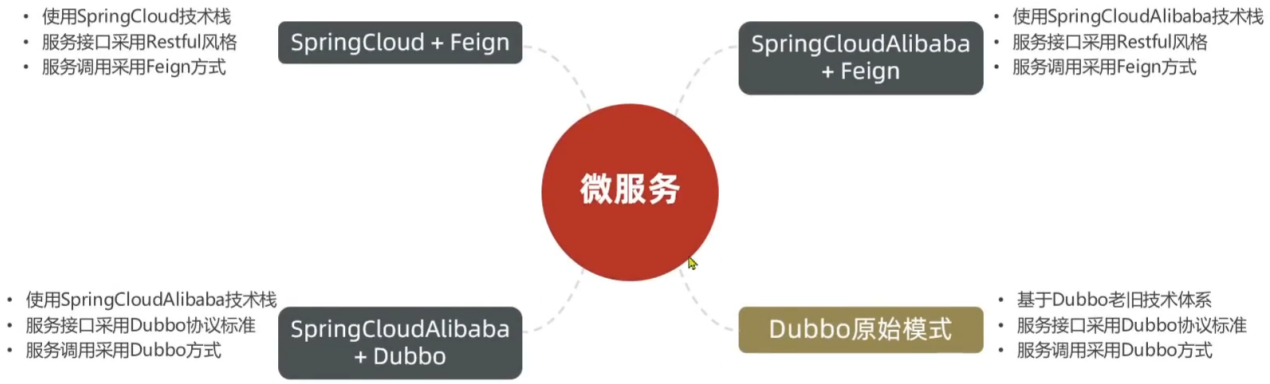
对于上面的问题，是任何一个微服务设计者都不能绕过去的，因此大部分的微服务产品都针对每一个问题提供了相应的组件来解决它们。

2. SpringCloud介绍

微服务的实现方式很多，例如dubbo+zookeeper， SpringCloud等等。那么这两种微服务实现方案有什么区别呢?

微服务技术方案对比

	Dubbo	SpringCloud	SpringCloudAlibaba
注册中心	zookeeper、Redis	Eureka、Consul	Nacos、Eureka
服务远程调用	Dubbo协议	Feign（http协议）	Dubbo、Feign
配置中心	无	SpringCloudConfig	SpringCloudConfig、Nacos
服务网关	无	SpringCloudGateway、Zuul	SpringCloudGateway、Zuul
服务监控和保护	dubbo-admin，功能弱	Hystix	Sentinel



SpringCloud微服务解决方案：不是一个框架，是一系列框架集合，目前包含二十多个框架，还在不断增加中.....

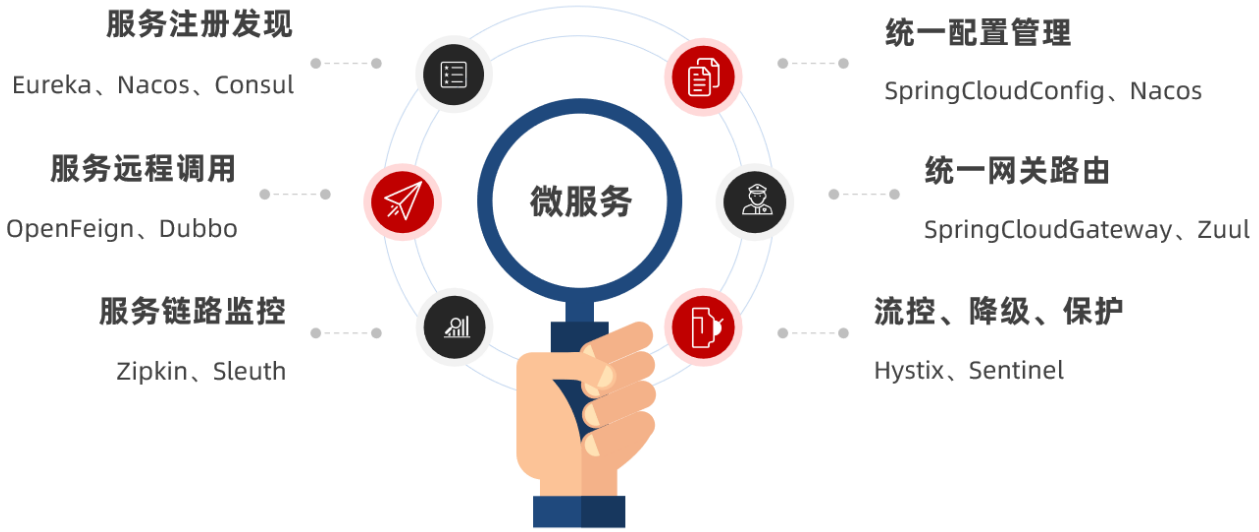
SpringCloud简介

官网地址：https://spring.io/projects/spring-cloud

中文文档(非官方)：https://www.springcloud.cc/

SpringCloud是Pivotal团队提供的、基于SpringBoot开箱即用的、一站式微服务解决方案的框架体系。目前已成为国内外使用最广泛的微服务框架。SpringCloud集成了各种微服务功能组件。

其中常见的组件包括：



SpringCloud版本

SpringCloud的版本命名比较特殊，因为它不是一个组件，而是许多组件的集合，它的命名是以A到Z的为首字母的一些单词（其实是伦敦地铁站的名字）组成：

Release Train	Boot Version
2021.0.x aka Jubilee	2.6.x, 2.7.x (Starting with 2021.0.3)
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

我们在项目中，会使用Hoxton.SR10版本，对应的SpringBoot版本为2.3.x

3. 小结

微服务架构的核心组件有哪些：

- 注册中心：解决服务治理问题，每个服务的地址可能会变，远程调用时就需要调整调用的地址。使用注册中心解决

  Alibaba的Nacos，NetFlix的Eureka，雅虎的zookeeper，.....
- 远程调用：服务之间要进行远程调用，进行数据交互

  httpclient，RestTemplate，NetFlix的OpenFeign
- 服务保护：要防止某个服务出错，导致问题级联扩大；要把问题限制在小范围内。

  NetFlix的Hystrix，**Alibaba的Sentinel**
- 负载均衡：远程调用时，如果目标服务有集群，需要实现负载均衡

  Ribbon，load-balancer
- 服务网关：众多的微服务，需要有一个统一的访问入口

  zuul，SpringCloud Gateway
- 配置中心：众多微服务的配置文件，由配置中心统一进行管理

  SpringCloudConfig，**Alibaba的Nacos**

微服务架构的技术方案：常见的有3套，实际开发中，通常是混搭的

- dubbo
- SpringCloud
- SpringCloudAlibaba

SpringCloud版本：SpringCloud不是一个框架，而是一批组件的集合。这一批组件必须使用配套的版本，所以通常要锁定SpringCloud的版本号。

二、远程调用OpenFeign【重点】

本章节学习目标：

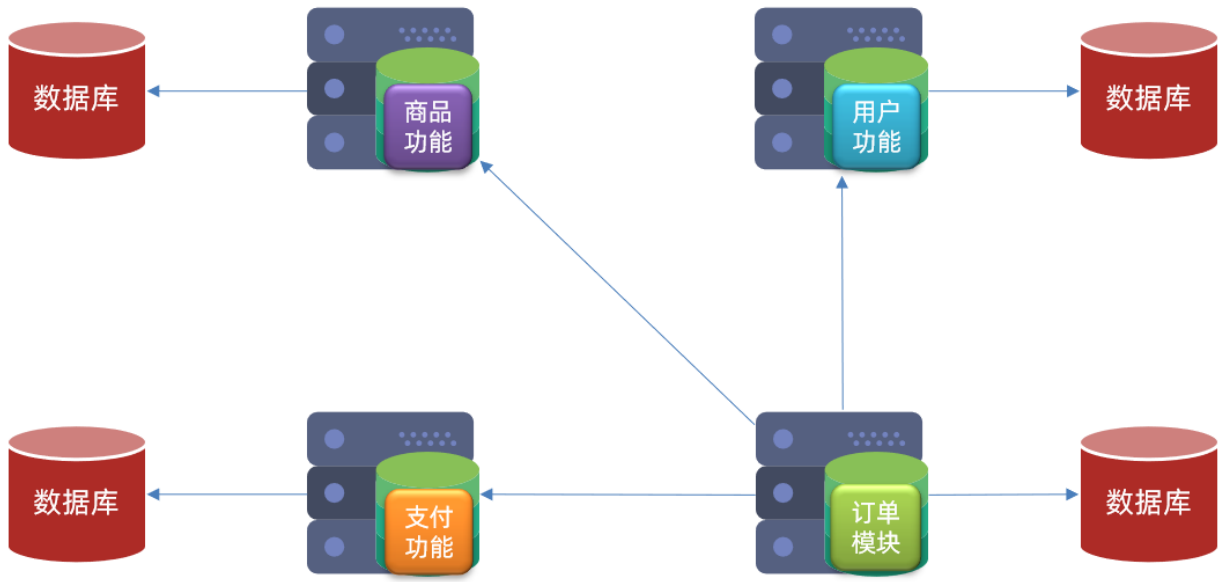
- [ ] 理解服务拆分的原则
- [ ] 使用RestTemplate实现远程调用

1. 服务拆分【了解】

1.1 服务拆分原则

这里我们总结了微服务拆分时的几个原则：

- 职责单一：不同微服务，不要重复开发相同业务
- 服务自治：微服务数据独立，不要访问其它微服务的数据库
- 面向服务：微服务可以将自己的业务暴露为接口，供其它微服务调用：有Controller即可



1.2 服务拆分示例

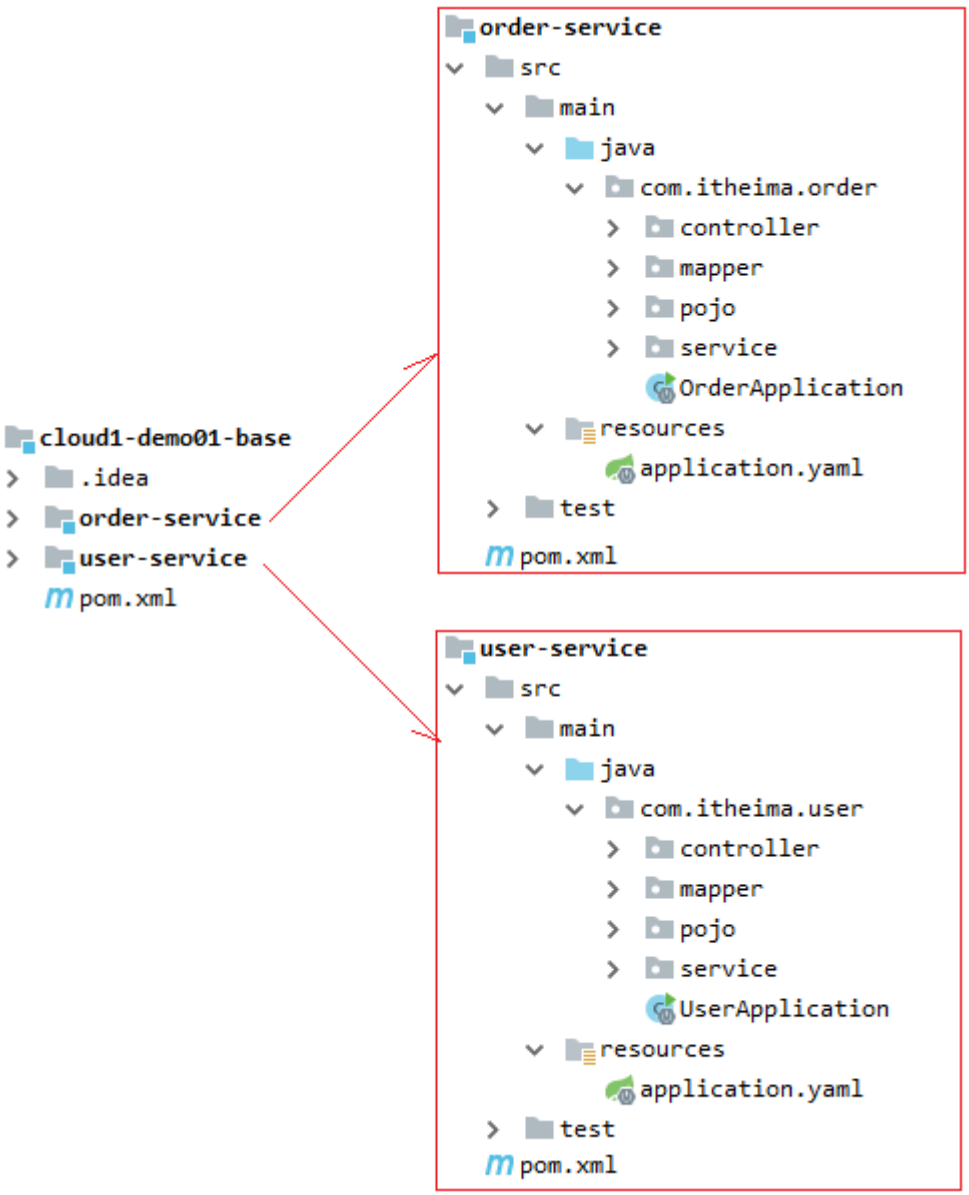
要求：

- 有用户服务，提供“根据id查询用户”功能
- 有订单服务，提供“根据id查询订单”功能

准备：

- 创建数据库cloud_user，执行脚本《cloud-user.sql》
- 创建数据库cloud_order，执行脚本《cloud-order.sql》

项目结构：



1) 创建父工程

注意：在开发项目时，不要随意动依赖坐标。一旦依赖出现问题，就可能导致整个项目出问题

注意：每个服务命名时，以英文字母开头，单词中间用横杠连接。不建议用下划线_连接

注意：每个服务的配置文件里，都必须有应用名称`spring.application.name`

1. 删除src文件夹
2. 修改pom.xml导入依赖

```
<packaging>pom</packaging>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.9.RELEASE</version>
    <relativePath/>
  </parent>
```

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <mysql.version>8.0.31</mysql.version>
  <mybatisplus.version>3.4.1</mybatisplus.version>
</properties>

<dependencyManagement>
  <dependencies>
    <!-- mysql驱动 -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysql.version}</version>
    </dependency>
    <!-- MybatisPlus -->
    <dependency>
      <groupId>com.baomidou</groupId>
      <artifactId>mybatis-plus-boot-starter</artifactId>
      <version>${mybatisplus.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
  </dependency>
</dependencies>
```

2) 准备用户服务

1. 用户服务的基础代码

1) 创建用户模块

在项目上创建Module：`user-service`

2) 导入依赖

修改pom.xml，添加依赖坐标

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
  <dependency>

    <groupId>com.baomidou</groupId>
```

```
        <artifactId>mybatis-plus-boot-starter</artifactId>
    </dependency>
</dependencies>
```

3) 准备配置文件

创建配置文件`application.yaml`

```yaml
server:
  port: 8080 #8080端口
spring:
  application:
    name: user-service #应用名称
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql:///cloud_user
    username: root
    password: root
logging:
  level:
    com.itheima.user: debug
  pattern:
    dateformat: HH:mm:ss.SSS
```

4) 准备引导类

```java
package com.itheima.user;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@MapperScan("com.itheima.user.mapper")
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class, args);
    }
}
```

2. 准备实体类User

```
package com.itheima.user.pojo;

import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;

@Data
@TableName("tb_user")
public class User {
    private Long id;
    private String username;
    private String address;
}
```

3. 创建UserMapper

```
package com.itheima.user.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.itheima.user.pojo.User;

public interface UserMapper extends BaseMapper<User> {
}
```

4. 创建UserService

```
package com.itheima.user.service;

import com.itheima.user.mapper.UserMapper;
import com.itheima.user.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@Service
public class UserService {
    @Autowired
    private UserMapper userMapper;

    public User findById(Long id){
        return userMapper.selectById(id);
    }
}
```

5. 创建UserController

```
package com.itheima.user.controller;
```

```
import com.itheima.user.pojo.User;
import com.itheima.user.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User findById(@PathVariable("id") Long id) {
        return userService.findById(id);
    }
}
```

6. 启动测试

- 启动服务
- 在浏览器上输入地址访问测试：http://localhost:8080/user/1

3) 准备订单服务

1. 订单服务的基础代码

1) 创建订单模块

在项目上右键创建Module：order-service

2) 导入依赖

修改pom.xml，添加依赖坐标

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
        </dependency>
```

```
    </dependencies>
```

3) 配置文件

创建application.yaml

```yaml
server:
  port: 7070
spring:
  application:
    name: order-service
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql:///cloud_order
    username: root
    password: root
logging:
  level:
    com.itheima.order: debug
  pattern:
    dateformat: MM-dd HH:mm:ss.SSS
```

4) 创建引导类

```java
package com.itheima.order;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@SpringBootApplication
@MapperScan("com.itheima.order.mapper")
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

2. 创建实体类Order

- 把User类拷贝进来，稍后会用到
- 创建Order类

```java
package com.itheima.order.pojo;

import com.baomidou.mybatisplus.annotation.TableField;
import com.baomidou.mybatisplus.annotation.TableName;
```

```
import lombok.Data;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@Data
@TableName("tb_order")
public class Order {
    private Long id;
    private Long userId;
    private String name;
    private Long price;
    private Integer num;

    @TableField(exist = false)
    private User user;
}
```

3. 创建OrderMapper

```
package com.itheima.order.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.itheima.order.pojo.Order;

/**
 * @author liuyp
 * @date 2023/05/09
 */
public interface OrderMapper extends BaseMapper<Order> {
}
```

4. 创建OrderService

```
package com.itheima.order.service;

import com.itheima.order.mapper.OrderMapper;
import com.itheima.order.pojo.Order;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@Service
public class OrderService {
    @Autowired
    private OrderMapper orderMapper;

    public Order findById(Long id){
```

```
        Order order = orderMapper.selectById(id);

        return order;
    }
}
```

5. 创建OrderController

```
package com.itheima.order.controller;

import com.itheima.order.pojo.Order;
import com.itheima.order.service.OrderService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * @author liuyp
 * @date 2023/05/09
 */
@RestController
@RequestMapping("/order")
public class OrderController {
    @Autowired
    private OrderService orderService;

    @GetMapping("/{id}")
    public Order findById(@PathVariable("id") Long id) {
        return orderService.findById(id);
    }
}
```

6. 启动测试

- 启动服务
- 打开浏览器输入地址访问测试：http://localhost:7070/order/101

1.3 提供者与消费者

在服务调用关系中，会有两个不同的角色：

- **服务提供者**：一次业务中，被其它微服务调用的服务。
- **服务消费者**：一次业务中，调用其它微服务的服务。

但是，服务提供者与服务消费者的角色并不是绝对的，而是相对于业务而言。

如果服务A调用了服务B，而服务B又调用了服务C，服务B的角色是什么?

- 对于A调用B的业务而言：A是服务消费者，B是服务提供者
- 对于B调用C的业务而言：B是服务消费者，C是服务提供者

因此，服务B既可以是服务提供者，也可以是服务消费者。

2. 远程调用技术方案

这里所谓远程调用，其实就是由Java代码发起HTTP请求、并获取HTTP响应。能实现这样要求的技术方案有很多，比如：

- URLConnection：JDK本身提供的技术
- HttpClients：Apache的HTTP类库
- OkHttp：square公司提供的类库，在Android开发中应用较多

Feign是Netflix公司提供服务调用组件，基于URLConnection提供了Feign。而SpringCloud团队对Feign再度封装成了Open-Feign。

Open-Feign支持SpringMVC注解。是Spring Cloud提供的一个声明式的伪Http客户端，它使得调用远程服务就像调用本地服务一样简单，只需要创建一个接口并添加一个注解即可。

3. OpenFeign使用示例

要求：在查询订单时，把订单关联的用户信息一并查询出来

- 在order-service中，查询一个订单OrderOrder中的user对象为空
- 在user-service中，提供了根据id查询用户的功能。请求路径是 http://localhost:8080/user/{id}

步骤：

1. 添加OpenFeign起步依赖：哪个微服务里要发起远程调用，哪个微服务就要添加OpenFeign依赖
2. 编写Feign接口
3. 在引导类上添加注解扫描Feign接口

3.1 加OpenFeign依赖坐标

3.1.1 锁定SpringCloud依赖版本

使用SpringCloud的任何组件，都必须要先锁定SpringCloud的依赖版本。否则很容易造成不同组件的版本混乱

我们直接在父工程的pom.xml中锁定依赖版本，加到`<dependencyManagement>`下的`<dependencies>`里

```
<!-- springCloud依赖版本锁定 -->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-dependencies</artifactId>
   <version>Hoxton.SR10</version>
   <type>pom</type>
   <scope>import</scope>
</dependency>
```

3.1.2 添加OpenFeign起步依赖

哪个服务里要发起远程调用，就要在哪个服务里添加OpenFeign的起步依赖。所以这里我们需要在order服务里添加OpenFeign起步依赖。加到order-service的pom.xml里

```
<!--OpenFeign起步依赖-->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

3.2 编写Feign接口

因为我们需要向user服务发起远程调用，所以把Feign接口命名为`UserClient`。这不是强制性规范，只是建议这样命名。

编写Feign接口时要求：

- 接口上加`@FeignClient(value="目标服务名", url="目标服务地址")`

  注意：当项目整合了注册中心以后，就不需要再加url了。只保留`value="目标服务名"`即可
- 方法上加SpringMVC的注解。下面代码中的`@GetMapping("/user/{id}")`表示请求的目标路径和请求方式

```
@FeignClient(value = "user-service", url = "http://localhost:8080")
public interface UserClient {

   @GetMapping("/user/{id}")
   User getById(@PathVariable("id") Long id);
}
```

3.3 扫描Feign接口

修改引导类，添加注解`@EnableFeignClients("包名")`，要扫描到刚刚创建的Feign接口

```
@SpringBootApplication
@MapperScan("com.itheima.order.mapper")
@EnableFeignClients("com.itheima.order.feign")
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
}
```

3.4 功能测试

修改OrderService类，注入UserClient对象，通过UserClient对象发起远程调用获取用户信息

```
@Service
public class OrderService {
    @Autowired
    private OrderMapper orderMapper;

    @Autowired
    private UserClient userClient;

    public Order findById(Long id){
        Order order = orderMapper.selectById(id);

        //使用UserClient发起远程调用，根据用户id获取User对象
        User user = userClient.getById(order.getUserId());
        //将User对象封装到order对象里
        order.setUser(user);

        return order;
    }
}
```

测试

- 在浏览器上输入地址：http://localhost:7070/order/101
- 查询的结果里有订单信息，也有订单关联的用户信息

4. OpenFeign输出日志

在之前的开发中，我们通过修改 `logging.level` 来控制日志输出的级别。然后这项配置不会对Feign生效

因为@FeignClient注解的客户端都是接口，我们实际上是通过这个接口的代理对象来进行远程调用的。而每个代理对象都会生成一个新的Feign.Logger实例对象，我们需要额外指定这个日志级别才可以。

步骤：

1. 修改配置文件，设置整体的日志级别
2. 创建Feign配置类，注册Logger.Level用于设置Feign的日志级别

4.1 设置整体的日志级别

修改order-service的配置文件，设置日志级别为debug

```
logging:
  level:
    com.itheima: debug
```

4.2 设置Feign的日志级别

有两种设置方式，用哪种都可以

方式1：@Bean方式

创建一个配置类，在配置类里使用@Bean设置日志级别

```
package com.itheima.order.config;

import feign.Feign;
import feign.Logger;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FeignConfig {
    /**
     * Feign的日志级别，是通过Logger.Level枚举来指定的
     * 它支持4种日志级别：
     *     1. NONE：不输出任何日志，是默认值
     *     2. BASIC：仅输出请求的方式、URL以及响应状态码、执行时间
     *     3. HEADERS：在BASIC基础上，额外输出请求头和响应头信息
     *     4. FULL：输出所有请求和响应的明细，包括头信息、请求体、元数据
     */
    @Bean
    public Logger.Level feignLog(){
        return Logger.Level.FULL;
    }
}
```

方式2：配置文件方式

修改order服务的配置文件application.yaml，添加如下配置：

```
feign:
  client:
    config:
      default:
        loggerLevel: FULL
```

5. OpenFeign性能优化

Feign的底层使用了JDK的URLConnection发起HTTP请求，而URLConnection是不支持连接池的，所以当频繁远程调用时势必会影响性能。

为了优化性能，我们可以使用 Apache的HttpClient代替默认的URLConnection，因为HttpClients具备**连接池**功能。使用步骤如下：

1. 添加httpclient的依赖坐标
2. 配置httpclient连接池

5.1 添加HttpClient依赖坐标

在order服务的pom.xml里添加HttpClient的依赖坐标：

```
<!--httpClient的依赖 -->
<dependency>
  <groupId>io.github.openfeign</groupId>
  <artifactId>feign-httpclient</artifactId>
</dependency>
```

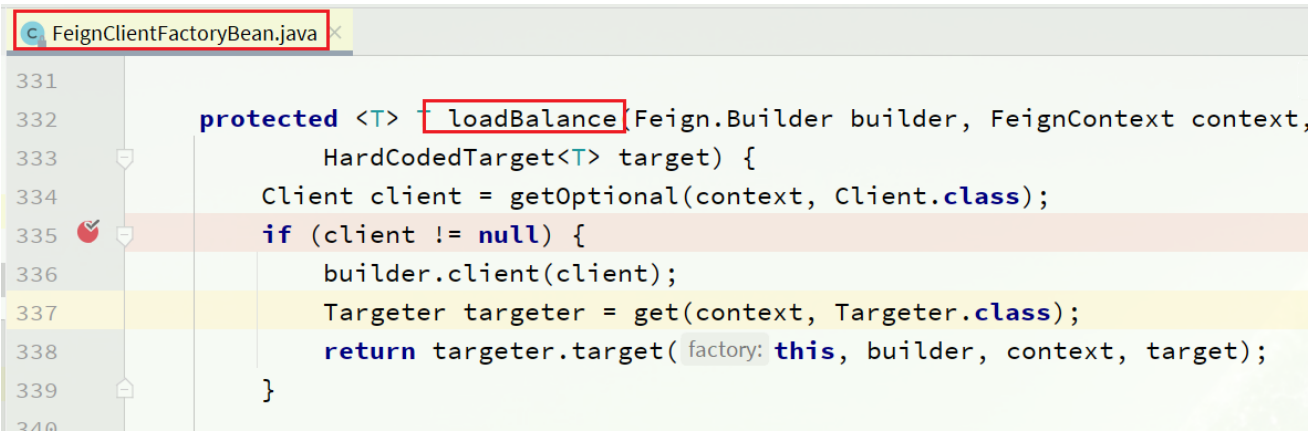
5.2 配置连接池参数

修改order服务的配置文件，添加连接池的参数：

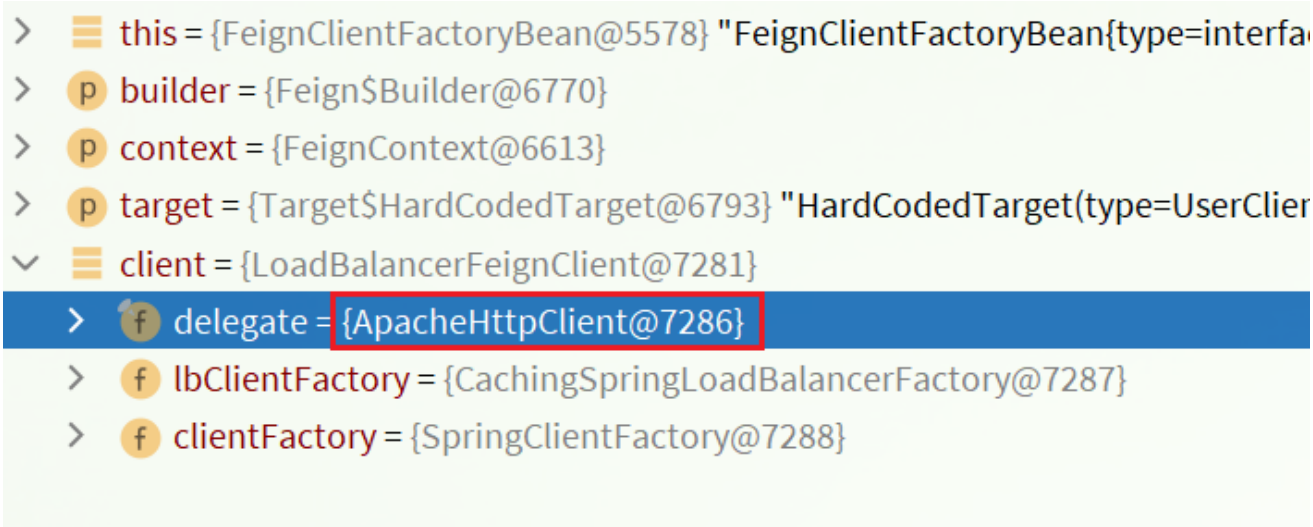
```
feign:
 httpclient:
  enabled: true # 开启feign对HttpClient的支持，默认是true
  max-connections: 200 # 最大的连接数，默认200
  max-connections-per-route: 50 # 每个路径的最大连接数，默认50
```

5.3 测试效果

接下来，在FeignClientFactoryBean中的loadBalance方法中打断点：



Debug方式**重启**order-service服务，可以看到这里的client，底层就是Apache HttpClient：



小结

Feign的使用步骤：

1. 先导入依赖：先锁定SpringCloud的依赖版本，再添加OpenFeign的依赖坐标
2. 创建Feign的接口：调用哪个服务，就创建一个Feign接口专门用于向这个服务发HTTP请求

```
@FeignClient(value="目标服务名", url="目标服务的地址")
public interface UserClient{

   /*
      返回值：期望Feign帮我们把HTTP响应结果转换成什么，返回值就是什么
      方法参数：告诉Feign发HTTP请求时需要的参数
      方法上加注解：SpringMVC的注解。路径是目标API接口的路径，要向这个API接口发请求
   */
   @GetMapping("/user/{id}")
   User findById(@PathVariable("id") Long id);
}
```

3. 在引导类或者配置类上加注解：`@EnableFeignClients("扫描的包名")`，必须扫描到刚刚创建的Feign接口

Feign的一些细节：

- Feign底层用了动态代理。Feign会帮我们生成一个代理类对象，由代理类对象帮我们：

  构造HTTP请求

  发送HTTP请求

  获取HTTP响应

  解析HTTP响应

- 注意事项：方法参数上的注解不能省略。@RequestBody、@PathVariable、@RequestParam

Feign的日志：

1. 先配置整体的日志级别为：debug
2. 再配置feign的日志级别：修改配置文件的

none：无日志

basic：只记录请求行和响应行

headers：只记录请求行和响应行、请求头和响应头

full：记录全部

Feign的优化：

- 什么地方需要优化：每次远程调用都要创建连接，然后释放连接，消耗性能消耗资源
- 优化的方案是：使用HTTP连接池
- 具体的做法是：
  1. 添加feign-httpclient依赖坐标
  2. 修改配置文件，开启feign底层的httpclient的支持

三、注册中心Nacos【重点】

- [ ] 能安装启动Nacos
- [ ] 能够使用Nacos作为注册中心

1. 介绍

1.1 服务治理问题

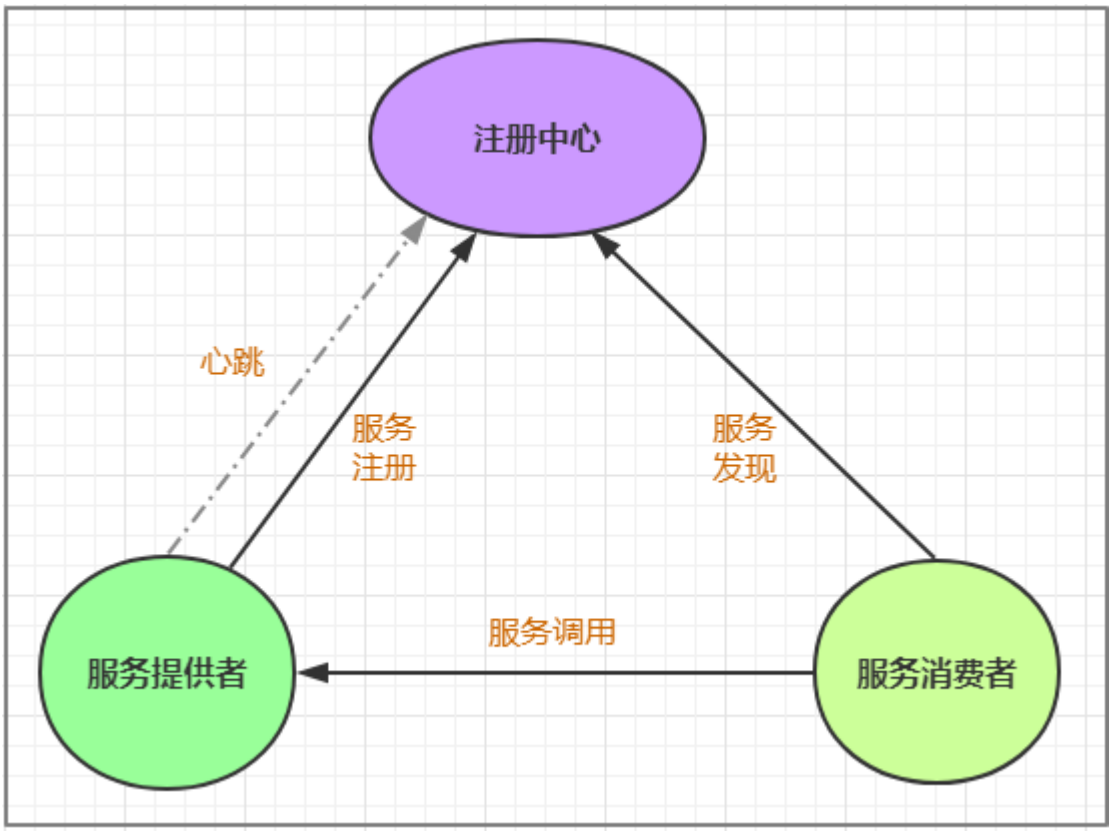
目前已经可以实现微服务之间的调用，但是我们把服务提供者的网络地址（ip，端口）等硬编码到了代码中，这种做法存在许多问题：

- 一旦服务提供者地址变化，就需要手工修改消费者的代码
- 一旦服务变得越来越多，人工维护调用关系困难

这时候就需要通过注册中心动态的实现**服务治理**。

服务治理是微服务架构中最核心最基本的模块。用于实现各个微服务的**自动化注册与发现**。

- **服务注册**：在服务治理框架中，都会构建一个注册中心，每个服务单元向注册中心登记自己提供服务的详细信息。并在注册中心形成一张服务的清单，服务注册中心需要以心跳的方式去监测清单中的服务是否可用，如果不可用，需要在服务清单中剔除不可用的服务。
- **服务发现**：服务调用方向服务注册中心咨询服务，并获取所有服务的实例清单，实现对具体服务实例的访问。



1.2 Nacos简介

国内公司一般都推崇阿里巴巴的技术，比如注册中心，SpringCloudAlibaba也推出了一个名为Nacos的注册中心。

Nacos是阿里巴巴的产品，现在是SpringCloud中的一个组件。相比Eureka功能更加丰富，在国内受欢迎程度较高。

1.3 Nacos安装

1. 下载：https://github.com/alibaba/nacos/releases，下载zip格式的程序包。

   可以直接使用资料里提供的程序包

2. 安装：

   免安装，直接解压到一个不含中文、空格、特殊字符的目录里

3. 启动：

   使用cmd切换到nacos的bin目录里

   执行命令：`startup.cmd -m standalone`，以单机模式启动nacos

4. 进入管理界面

   打开浏览器输入地址 http://localhost:8848/nacos

   默认帐号：nacos，密码：nacos

2. Nacos使用入门

Nacos是SpringCloudAlibaba的组件，而SpringCloudAlibaba也遵循SpringCloud中定义的服务注册、服务发现规范。因此使用Nacos和使用Eureka对于微服务来说，并没有太大区别。

主要差异在于：

- 依赖坐标不同
- 配置参数不同

2.1 添加坐标

1) 锁定SpringCloudAlibaba依赖版本

使用SpringCloud的任意组件，都必须要锁定SpringCloudAlibaba的依赖版本。否则不同的SpringCloudAlibaba组件之间会因为版本混乱导致出错。

在父工程pom.xml的`<dependencyManagement>`下`<dependencies>`中添加SpringCloudAlibaba的版本锁定

```
<!--SpringCloudAlibaba-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>
    <version>2.2.6.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

2) 添加nacos-discovery依赖坐标

在用户服务和订单服务中添加nacos的服务发现包的依赖坐标：

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2.2 配置注册中心地址

每个微服务都要注册到Nacos里，所以每个微服务都要配置Nacos的地址。添加以下内容：

```
spring:
  application:
    name: 应用服务名称 #必须有!
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #Nacos的地址
```

1) 用户服务的配置文件

用户服务的配置文件内容 最终如下：

```
server:
  port: 8080 #8080端口
spring:
  application:
    name: user-service #应用名称，必须有!
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #注册中心的地址
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql:///cloud_user
    username: root
    password: root
logging:
  level:
    com.itheima.user: debug
  pattern:
    dateformat: HH:mm:ss.SSS
```

2) 订单服务的配置文件

订单服务的配置文件内容 最终如下：

```
server:
```

```yaml
  port: 7070 #端口
spring:
  application:
    name: order-service #应用服务名称，必须有！
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #注册中心的地址
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql:///cloud_order
    username: root
    password: root
logging:
  level:
    com.itheima.order: debug
  pattern:
    dateformat: MM-dd HH:mm:ss.SSS
feign:
  httpclient:
    enabled: true
    max-connections: 200
    max-connections-per-route: 50
```

2.3 开启服务发现功能

修改所有微服务的引导类，在引导类上添加注解`@EnableDiscoveryClient`

用户服务最终的引导类

```java
package com.itheima.user;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
@MapperScan("com.itheima.user.mapper")
public class UserApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class, args);
    }
}
```

订单服务最终的引导类

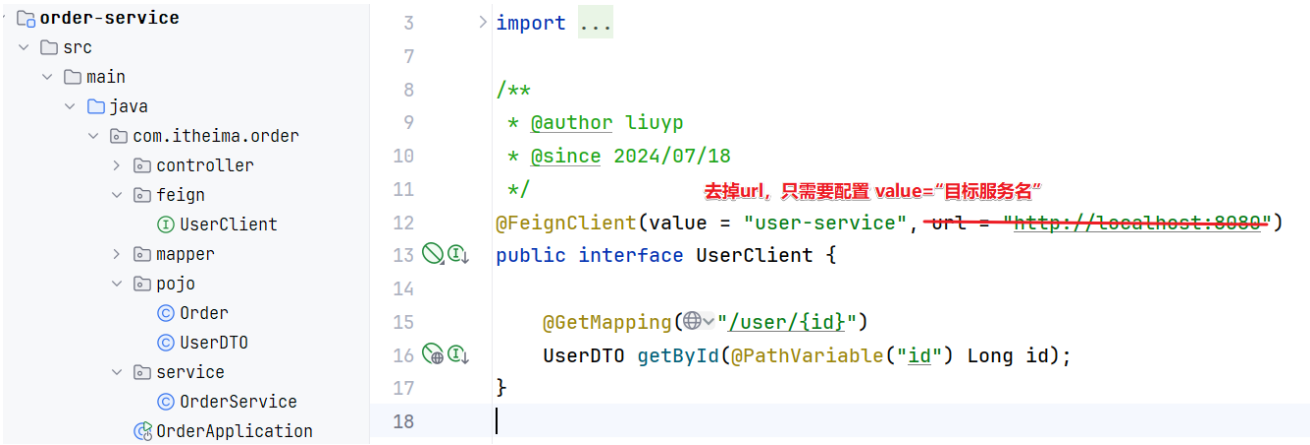
```java
package com.itheima.order;


import org.mybatis.spring.annotation.MapperScan;
```

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@EnableDiscoveryClient
@SpringBootApplication
@MapperScan("com.itheima.order.mapper")
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

2.4 修改Feign接口

因为已经使用了注册中心维护所有服务的地址，所以在远程调用时，不需要再把服务地址写死在代码里了，只要给Feign配置目标服务名即可。

最终修改如下：




```java
@FeignClient(value = "user-service")
public interface UserClient {

    @GetMapping("/user/{id}")
    UserDTO getById(@PathVariable("id") Long id);
}
```

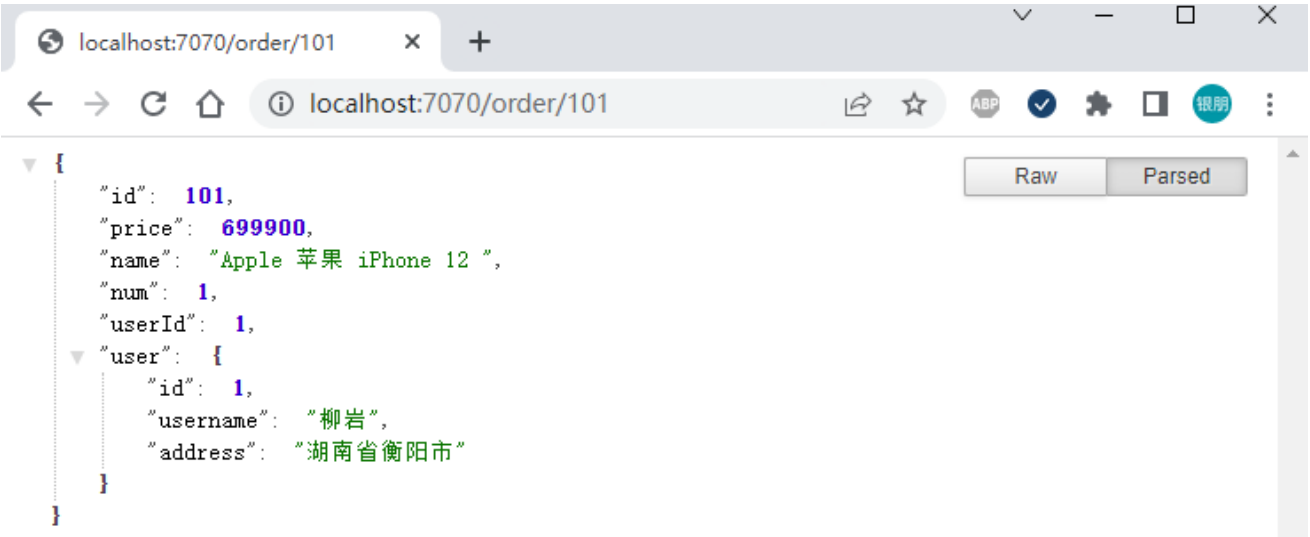
2.5 功能测试

验证服务是否注册到Nacos

1. 先启动Nacos，再启动用户服务和订单服务
2. 打开Nacos控制台，访问http://localhost:8848/nacos，查看注册的服务信息

验证远程调用是否成功

打开浏览器，访问http://localhost:7070/order/101



3. Nacos的原理

3.1 临时实例与非临时实例

从Nacos1.0开始就提供了一个配置参数：`spring.cloud.nacos.discovery.ephemeral`，用于设置微服务实例是临时实例还是非临时实例

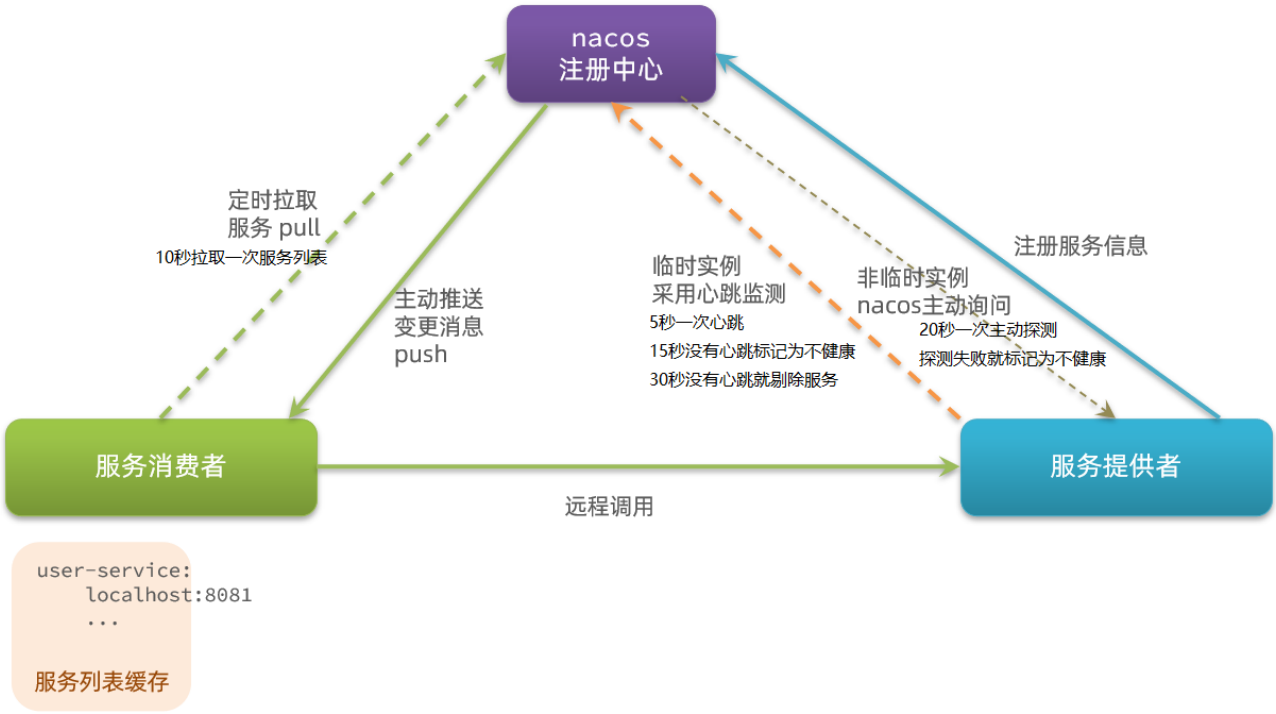
- 值为true：微服务是临时实例。当微服务长时间不能向Nacos续约时，Nacos会剔除微服务实例的信息
- 值为false：微服务是非临时实例。即使微服务长时间不能向Nacos续约，Nacos也仅仅是将服务实例标记为不健康状态，而不会剔除服务实例的信息

两者的作用是：

- 临时实例：适用于应对流量突增的情况，当流量洪峰时，增加服务实例数量；当流量高峰过去以后，服务实例停止，就自动从Nacos里剔除了

- 非临时实例：适用于服务的保护。当Nacos发现短时间有大量微服务不健康时，实际可能是Nacos的网络波动等情况，而微服务实例其实是正常的。 这时Nacos仍然保留服务信息，供消费者进行拉取

3.2 Nacos的原理



4. 常见错误

启动微服务时，可能会报以下错误：

```
10-08 09:45:06.338 ERROR 3836 --- [           main] c.a.c.n.registry.NacosServiceRegistry    : nacos registry, order-se

com.alibaba.nacos.api.exception.NacosException Create breakpoint : failed to req API:/nacos/v1/ns/instance after all serv
    at com.alibaba.nacos.client.naming.net.NamingProxy.reqApi(NamingProxy.java:556) ~[nacos-client-1.4.2.jar:na]
    at com.alibaba.nacos.client.naming.net.NamingProxy.reqApi(NamingProxy.java:498) ~[nacos-client-1.4.2.jar:na]
    at com.alibaba.nacos.client.naming.net.NamingProxy.reqApi(NamingProxy.java:493) ~[nacos-client-1.4.2.jar:na]
```

出错的原因，通常是Nacos缓存问题：

- 在使用Nacos的时候，每次有服务注册到Nacos，Nacos会把服务的信息缓存起来。缓存的数据在Nacos软件的data文件夹里
- 下次再使用Nacos的时候，如果 之前注册服务的ip地址 和 这一次注册服务的ip地址不同，就会报这个错

解决的方法是：

- 关闭Nacos
- 清理掉Nacos里的data文件夹
- 再启动Nacos，启动微服务。 就没有缓存了，就不会报这个错了

5. 小结

使用Nacos，微服务整合Nacos的步骤：

1. 导入依赖：

需要先锁定SpringCloudAlibaba的依赖版本

给所有微服务都导入依赖nacos-discovery

2. 修改配置：

   给所有微服务都配置 Nacos注册中心的地址

3. 修改引导类：

   给所有微服务的引导类上都添加 @EnableDiscoveryClient

微服务整合Nacos以后，使用Feign远程调用时：不需要再写目标服务的url地址了

- `@FeignClient("目标服务名")`，目标服务名一定不要写错了

Nacos的原理：

- Naco作为注册中心，主要职责有：维护所有活跃服务的地址列表；监控所有服务的健康状态

  Nacos维护的数据：

  ```
  服务名：
      实例1：实例1的地址
      实例2：实例2的地址
  服务名：
      实例1：实例1的地址
  ```

- 当微服务启动时，会自动进行服务注册与心跳续约

  服务注册：微服务把自己的地址信息上报给Nacos注册中心。

  服务健康监控：

  - 如果微服务实例是临时实例：由微服务主动发起心跳进行续约。

    每5秒一次心跳；

    如果Nacos15s收不到心跳，就标记为不健康；

    如果Nacos30s收不到心跳，就剔除服务实例的地址
  - 如果微服务实例是永久实例（非临时）：由Nacos进行主动探测

    Nacos会每20s向微服务发起一次探测；不健康的实例地址也不剔除，而是标记不为健康

  服务发现：

  - 微服务会定时从Nacos里拉取服务地址的列表：默认是10s一次
  - 如果Nacos里的地址有变化，也会推送

四、负载均衡Ribbon

本章节学习目标：

- [ ] 理解负载均衡的概念与分类
- [ ] 能够使用ribbon实现负载均衡

1. 负载均衡简介

1.1 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）分摊到多个操作单元上进行执行。

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(提供者)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。
- 客户端(消费者)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。

1.2 Ribbon负载均衡

Ribbon是NetFlix提供的客户端负载均衡工具，它有助于控制HTTP和TCP客户端的行为。

Ribbon提供了多种负载均衡算法，包括轮询、随机等等，同时也支持自定义负载均衡算法。

它不需要独立部署，而是几乎存在于SpringCloud的每个组件中，包括Nacos也已经引入了Ribbon。

- 当使用RestTemplate发起远程调用时，需要给RestTemplate对象添加@LoadBalanced注解
- 当使用OpenFeign发起远程调用时，不需要做任何额外配置，就具备负载均衡效果

2. Ribbon效果演示

在刚刚的案例中，order-service要调用user-service，而user-service只有一个服务实例。但是实际环境中为了保证高可用，通常会搭建集群环境。比如user-service搭建了集群，这就需要实现对用户服务的负载均衡效果

我们以订单中调用用户服务为例，来演示负载均衡的问题

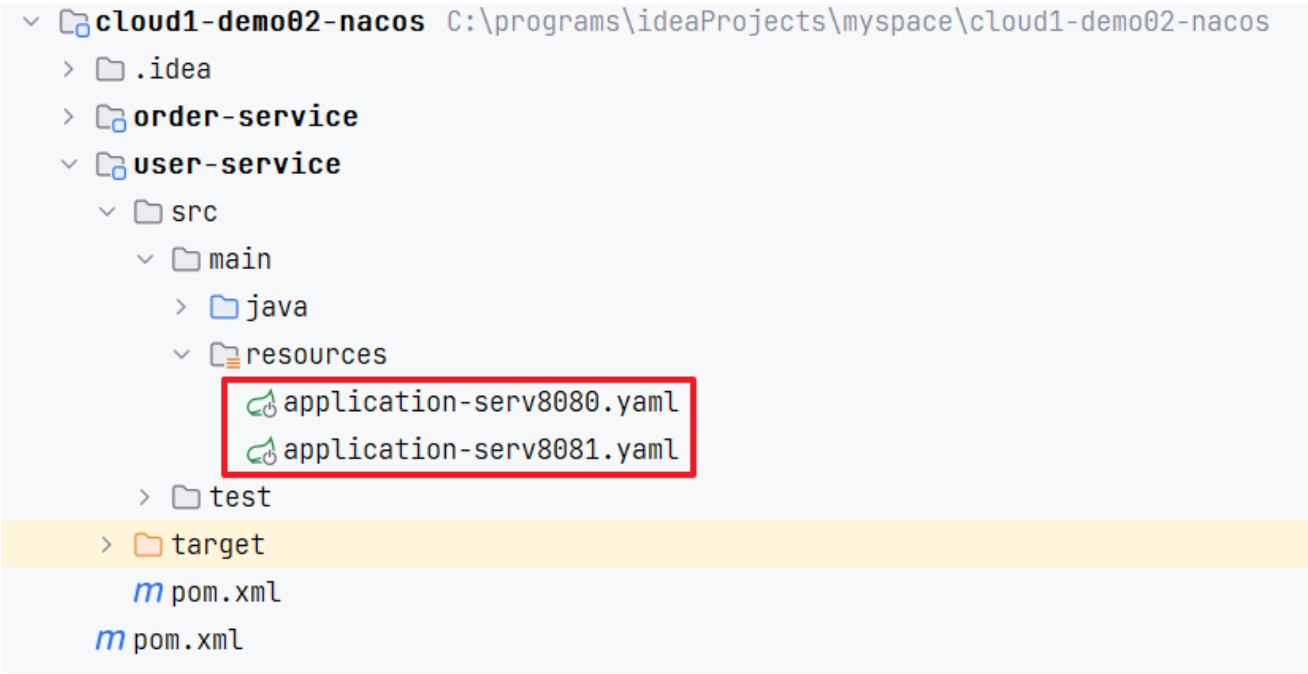
基础服务代码：

- 注册中心：Nacos
- 用户服务：提供findById功能
- 订单服务：提供findById功能，并通过RestTemplate远程调用 用户服务。

2.1 搭建用户服务集群

1) 复制用户服务的配置文件

复制用户服务的配置文件，准备多个不同后缀名的配置环境`application-环境标识.yaml`



- application-8080serv.yaml：设置为8080端口

```yaml
server:
  port: 8080
spring:
  application:
    name: user-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///cloud_user?useSSL=false
    username: root
    password: root
logging:
  level:
    com.itheima.user: debug
  pattern:
    dateformat: HH:mm:ss.SSS
```

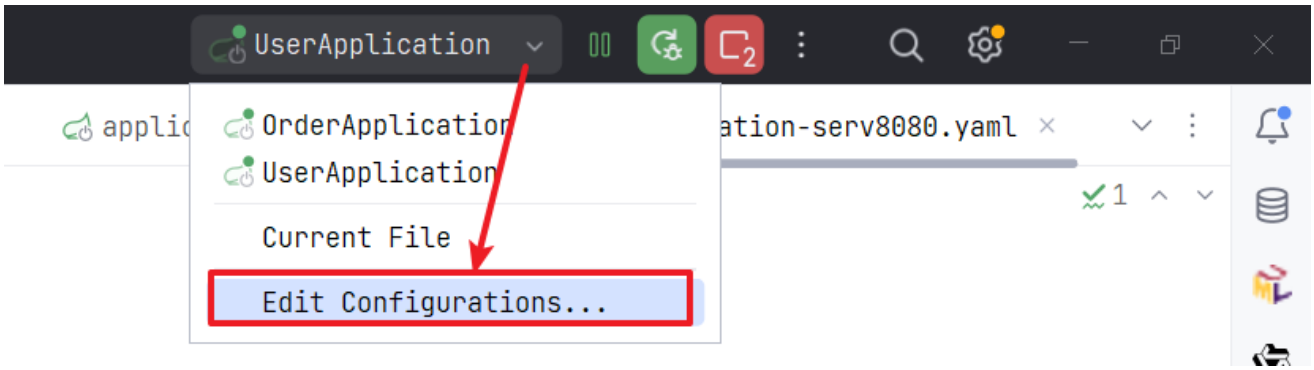
- application-8081serv.yaml：设置为8081端口

```yaml
server:
  port: 8081
spring:
  application:
    name: user-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
```

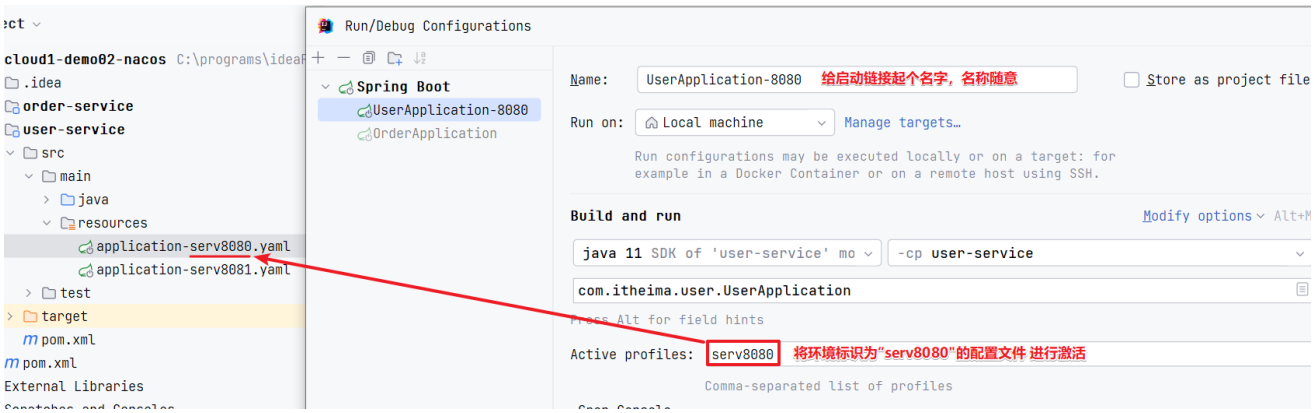
```yaml
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///cloud_user?useSSL=false
    username: root
    password: root

logging:
  level:
    com.itheima.user: debug
  pattern:
    dateformat: HH:mm:ss.SSS
```

2) 设置启动链接

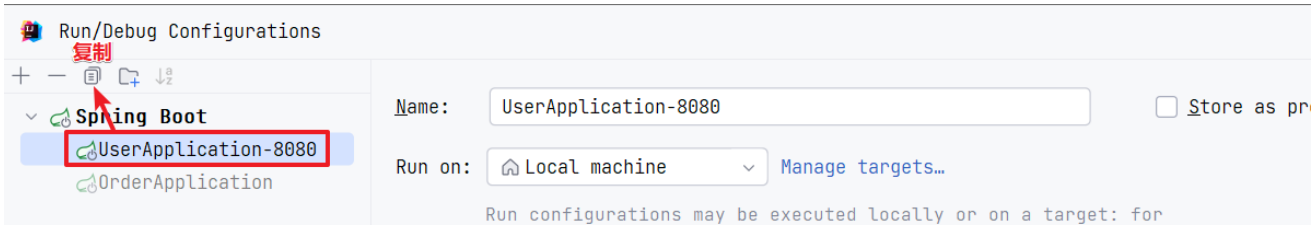
1. 找到启动链接的编辑按钮



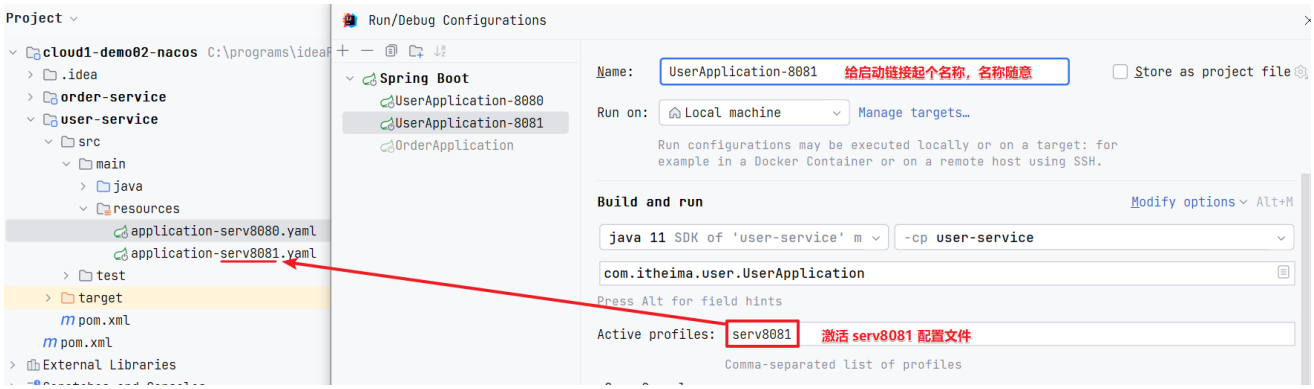
2. 创建启动链接激活serv8080配置文件



3. 复制一个启动链接



4. 修改复制出来的启动链接，激活serv8081



3) 启动服务

1. 分别启动两个启动链接，启动2个用户服务实例。启动后在nacos界面上可看到这两个实例
2. 再启动订单服务

2.2 订单服务调用用户服务

在浏览器上访问http://localhost:7070/order/101，多次刷新访问，发现多个用户服务都被调用到了，而且是轮询的方式。

3. Ribbon实现原理分析【了解】

@LoadBalanced注解

注解的文档中说明了：如果使用@LoadBalanced标记一个RestTemplate或者WebClient对象，表示将会配置使用一个LoadBanalcerClient对象

```java
/**
 * Annotation to mark a RestTemplate or WebClient bean to be configured to use a
 * LoadBalancerClient.
 * @author Spencer Gibb
 */
@Target({ ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Qualifier
public @interface LoadBalanced {

}
```

LoadBalancerClient源码

LoadBalancerClient有一个子类`RibbonLoadBalancerClient`，它使用Ribbon实现了负载均衡调用

```java
public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint)
        throws IOException {
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
    Server server = getServer(loadBalancer, hint);
    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);
}
```

获取负载均衡器ILoadBalancer
而ILoadBalancer会根据serviceId去Eureka里获取服务地址列表并缓存起来

利用内置的负载均衡算法，从服务地址列表中选择一个

IRule源码

在上一步的`getServer`方法代码如下：

```java
protected Server getServer(ILoadBalancer loadBalancer, Object hint) {
    if (loadBalancer == null) {
        return null;
    }
    // Use 'default' on a null hint, or just pass it on?
    return loadBalancer.chooseServer(hint != null ? hint : "default");
}
```

继续跟进去，发现是由rule对象挑选了一个目标服务地址

java × BaseLoadBalancer.java ×

```
    */
    public Server chooseServer(Object key) {
        if (counter == null) {
            counter = createCounter();
        }
        counter.increment();
        if (rule == null) {
            return null;
        } else {
            try {
                return rule.choose(key);
            } catch (Exception e) {
                logger.warn("LoadBalancer [{}]:  Error choosing server for key {}", name, key, e);
                return null;
            }
        }
    }
```

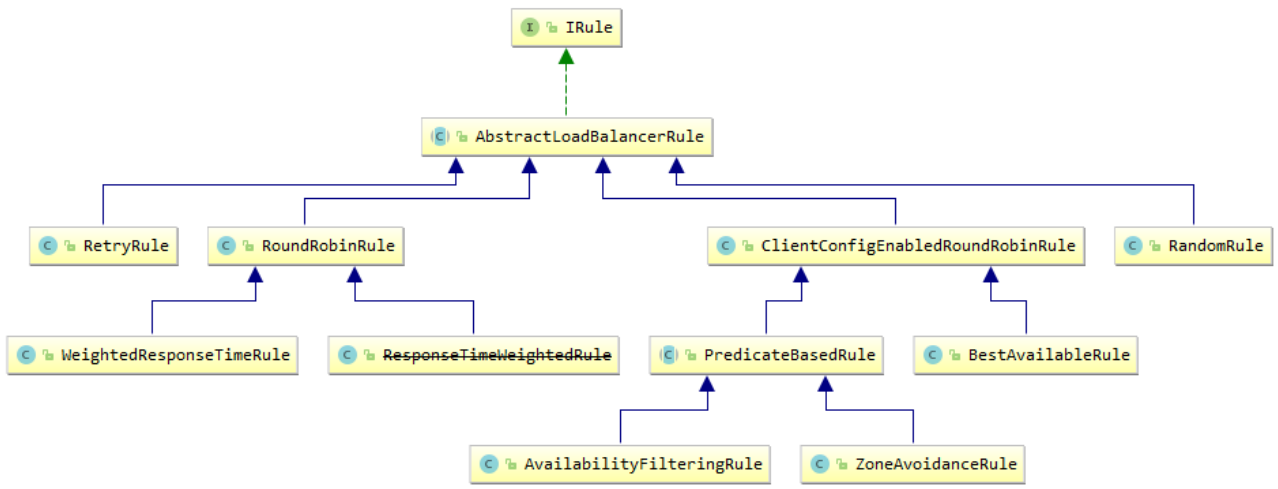
这个IRule是什么呢?

```
    private static Logger logger = LoggerFactory
            .getLogger(BaseLoadBalancer.class);
    private final static IRule DEFAULT_RULE = new RoundRobinRule();
    private final static SerialPingStrategy DEFAULT_PING_STRATEGY = new SerialP
    private static final String DEFAULT_NAME = "default";
    private static final String PREFIX = "LoadBalancer_";

    protected IRule rule = DEFAULT_RULE;
```

4. Ribbon负载均衡策略

1) 负载均衡策略介绍

IRule是负载均衡策略的顶级接口，它的每个实现类就是一个我们可以选择使用的负载均衡策略



Ribbon内置了多种负载均衡策略，这些策略的顶级接口是`com.netflix.loadbalancer.IRule`，它的常用实现有：

策略名	描述	详细说明
RandomRule	随机策略	随机选择一个Server
RoundRobinRule	轮询策略	按顺序依次选择Server（默认的策略）
RetryRule	重试策略	在一个配置时间段内，如果选择的Server不成功，则一直尝试选择一个可用的Server
BestAvailableRule	最低并发策略	逐个考察Server，如果Server断路器打开则忽略掉；再选择其余Server中并发链接最低的
AvailabilityFilteringRule	可用过滤策略	过滤掉一直失败并被标记为circuit tripped的server，过滤掉那些高并发链接的server
WeightedResponseTimeRule	响应时间加权重策略	根据server的响应时间分配权重，响应时间越长权重越低，被选择到的机率越小。响应时间越短被选中的机率越高。刚开始运行时使用robin策略选择Server
ZoneAvoidanceRule	区域权重策略	综合判断server所在区域的性能，和server的可用性，轮询选择server并且判断一个aws zone的运行性能是否可用，剔除不可用的zon中的所有server

2) 修改负载均衡策略

配置文件方式

我们可以通过修改配置文件来调用Ribbon的负载均衡策略，只要修改**<u>调用者的配置文件</u>**即可：

```
服务名:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

例如，在订单服务的`application.yaml`里添加如下配置：

```
#调用用户服务时，使用指定的负载均衡策略
user-service:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

代码@Bean方式

我们修改订单服务的配置类（或引导类），添加：

```
@Bean
public IRule randomRule(){
   return new RandomRule();
}
```

注意事项

1. 使用配置文件方式，只能设置 调用某一服务时指定负载均衡策略，而不是全局的策略；

   使用@Bean方式，配置的是全局的负载均衡策略

2. 实际开发中，通常使用默认的负载均衡策略，不需要修改

5. 饥饿加载

在我们的示例代码里，订单服务里必须要先拉取到用户服务的地址列表，创建LoadBanalcerClient对象，然后才可以发起远程调用。而Ribbon有两种策略：

- 懒加载：即第一次访问目标服务时，才会去创建LoadBalanceClient对象，拉取目标服务地址列表，会导致第一次请求时间比较长
- 饥饿加载：当服务启动时，就立即拉取服务列表，并创建好LoadBalancerClient对象；这样的话，即使第一次远程调用时也不会花费很长时间

Ribbon默认采用懒加载方式，如果我们需要修改成饥饿加载的话，以订单服务为例，可以在订单服务的配置文件里增加如下配置：

```
ribbon:
  eager-load:
    enabled: true #开启饥饿加载
    clients: #要饥饿加载的服务列表
      - user-service
```

6. 小结

负载均衡有两种：

- 提供者一方的负载均衡：nginx。 适合用在 整个服务端的最前沿，直接处理客户端的请求。因为消费者是无感知的
- 消费者一方的负载均衡：ribbon。适合用在 微服务之间互相调用时实现负载均衡

Ribbon负载均衡的使用：默认什么都不用做就有负载均衡，是轮询策略

如果想要修改负载均衡策略：修改消费者一方

- 针对某一服务设置负载均衡策略：修改配置文件
- 设置全局的负载均衡策略：使用@Bean

Ribbon的饥饿加载：

- 什么是饥饿加载：当服务启动时，就立即从注册中心里拉取服务地址列表
- 饥饿加载的好处：第一次访问做远程调用时，速度比较快，直接做负载均衡即可，不需要再拉取地址列表了

- 饥饿加载的实现：修改消费者一方的配置文件

```yaml
ribbon:
  eager-load:
    enabled: true
    clients:
      - 目标服务名称
      - 目标服务名称
```

五、Nacos分级存储与环境隔离

- [ ] 理解Nacos的分级存储
- [ ] 理解Nacos的环境隔离

1. Nacos分级存储模型

1.1 配置实例集群

1.1.1 Nacos里实例集群的概念

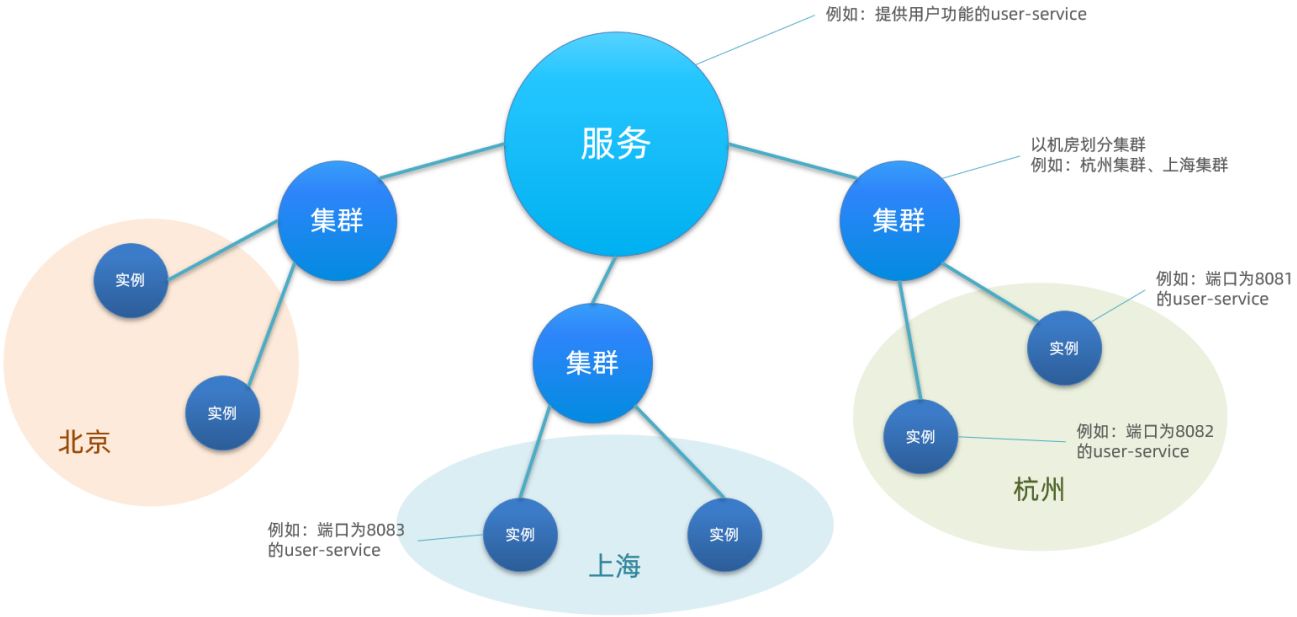
一个服务可以有多个实例，例如我们的user-service，可以有很多实例，例如:

- 127.0.0.1:8081
- 127.0.0.1:8082
- 127.0.0.1:8083

在大型项目里，为了满足异地容灾的需要，通常将这些实例部署在全国各地的不同机房，Nacos将同一机房内的实例称为一个**集群cluster**。例如：

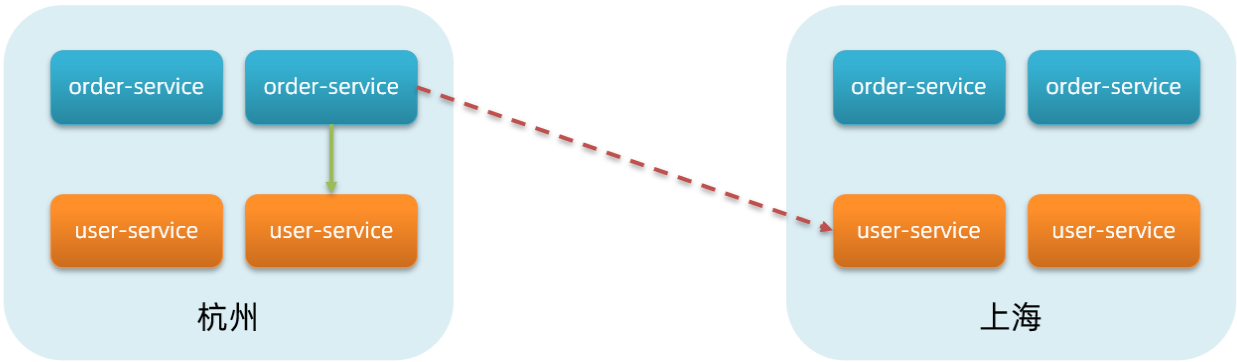
- 127.0.0.1:8081，在上海机房
- 127.0.0.1:8082，在上海机房
- 127.0.0.1:8083，在杭州机房

也就是说，user-service是服务，一个服务可以包含多个集群，如杭州、上海，每个集群下可以有多个实例，形成分级模型，如图：



微服务互相访问时，应该尽可能访问同集群实例，因为本地访问速度更快。

当本集群内不可用时，才访问其它集群。例如：杭州机房内的order-service应该优先访问同机房的user-service。



1.1.2 配置实例集群

配置语法：只要修改配置文件，增加如下设置

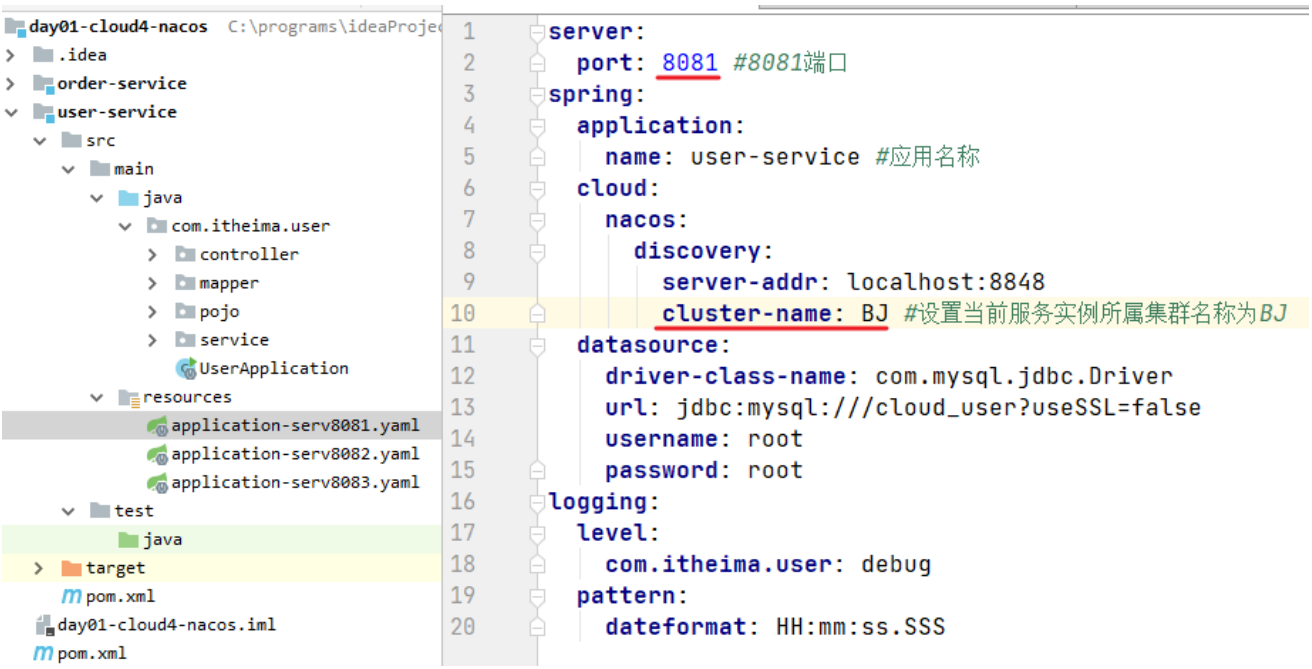
```yaml
spring:
  cloud:
    nacos:
      discovery:
        cluster-name: HZ #设置当前服务实例所属集群名称为HZ
```

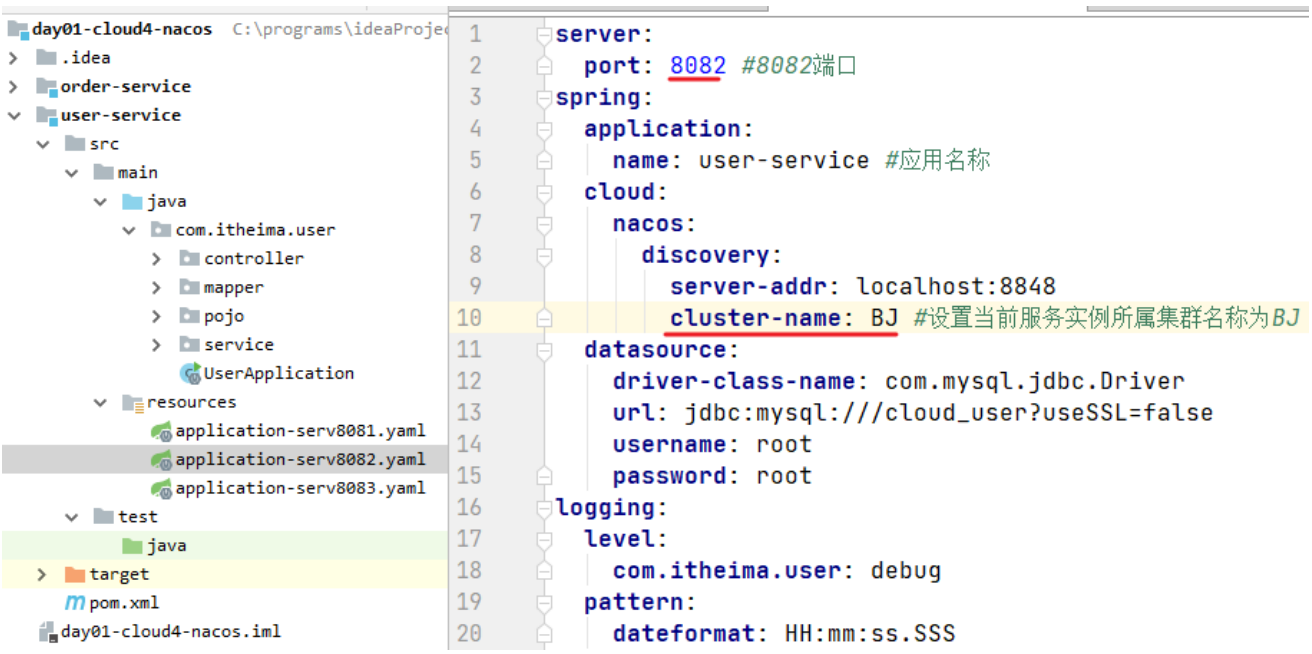
1) 准备配置文件

我们以用户服务为例，启动多个用户服务实例，分别为：

- localhost:8081，属于BJ集群
- localhost:8082，属于BJ集群

- localhost:8083，属于HZ集群





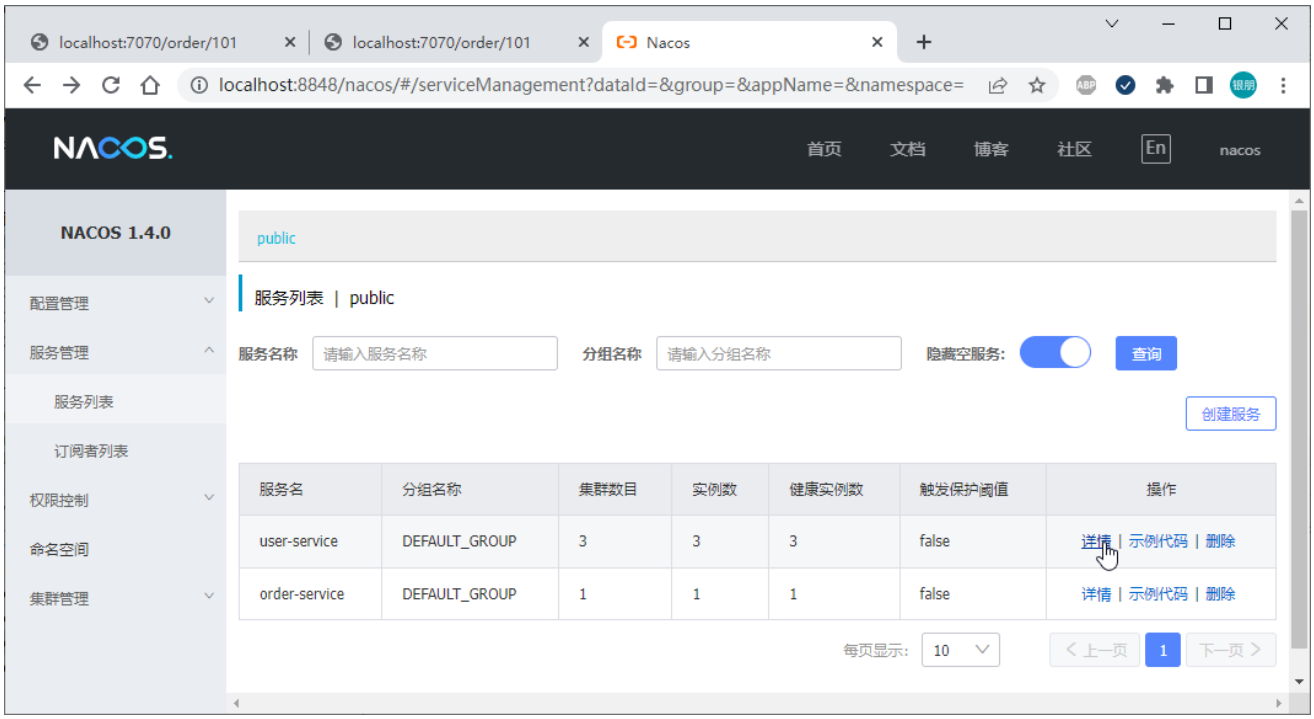
2) 准备启动链接

创建三个启动链接，分别激活这三个配置文件

创建步骤，略

1.1.3 查看配置效果

打开Nacos的控制台，找到user-service服务，查看详情



查看服务实例列表

集群：HZ　　集群配置

IP	端口	临时实例	权重	健康状态	元数据	操作
192.168.1.107	8083	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线

集群：BJ　　集群配置

IP	端口	临时实例	权重	健康状态	元数据	操作
192.168.1.107	8081	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线
192.168.1.107	8082	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线

1.2 同集群优先访问

在划分了实例集群之后，我们期望集群内的服务优先调用集群内部的服务实例，这样会有更高的响应速度。而默认的负载均衡策略并不能实现这种效果，因此Nacos提供了一个新的负载均衡策略：`NacosRule`。

1.2.1 配置负载均衡策略

我们以订单服务为例，假如订单服务属于BJ集群，那么它最好优先调用BJ集群内的用户服务。而实现的方式是：

1. 给订单服务设置集群：BJ
2. 给订单服务设置负载均衡策略：`NacosRule`

最终配置如下：

```yaml
server:
  port: 7070
spring:
  application:
    name: order-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
        cluster-name: BJ #订单服务属于BJ集群
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///cloud_order?useSSL=false
    username: root
    password: root

logging:
```

```yaml
  level:
    com.itheima.user: debug
  pattern:
    dateformat: HH:mm:ss.SSS
mybatis:
  configuration:
    map-underscore-to-camel-case: true
user-service:
  ribbon:
    NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule # 负载均衡规则
```

1.2.2 测试效果

1) 同集群调用

1. 启动所有用户服务和订单服务
2. 打开浏览器多次访问http://localhost:7070/order/101，发现只有8081和8082的用户服务被访问了

2) 跨集群调用

1. 关闭8081和8082用户服务
2. 稍等一会，再访问http://localhost:7070/order/101，发现8083服务被调用到了，但是idea报了一个警告：A cross-cluster call occurs，意思是 出现了一次跨集群调用

```
11:57:47.731  WARN 5024 --- [nio-7070-exec-8] c.alibaba.cloud.nacos.ribbon.NacosRule  : A cross-cluster call occurs，name = user-service, clusterName = BJ, instance = [Instance{instanceId='192.168.1.107#8083#HZ#DEFAULT_GROUP@@user-service', ip='192.168.1.107', port=8083, weight=1.0, healthy=true, enabled=true, ephemeral=true, clusterName='HZ', serviceName='DEFAULT_GROUP@@user-service', metadata={preserved.register.source=SPRING_CLOUD}}]
```

1.3 Nacos的服务实例权重

实际部署中，服务器设备性能往往是有差异的，部分实例所在机器性能较好，另一些较差，我们希望性能好的机器承担更多的用户请求。

但默认情况下NacosRule是同集群内随机挑选，不会考虑机器的性能问题。因此，Nacos提供了权重配置来控制访问频率，权重越大则访问频率越高。

3.3.1 设置服务实例的权重

在nacos控制台，找到user-service的实例列表，点击编辑，即可修改权重：

3.3.2 测试效果

1. 启动所有用户服务和订单服务
2. 打开浏览器访问http://localhost:7070/order/101，发现8081被访问到的频率更高

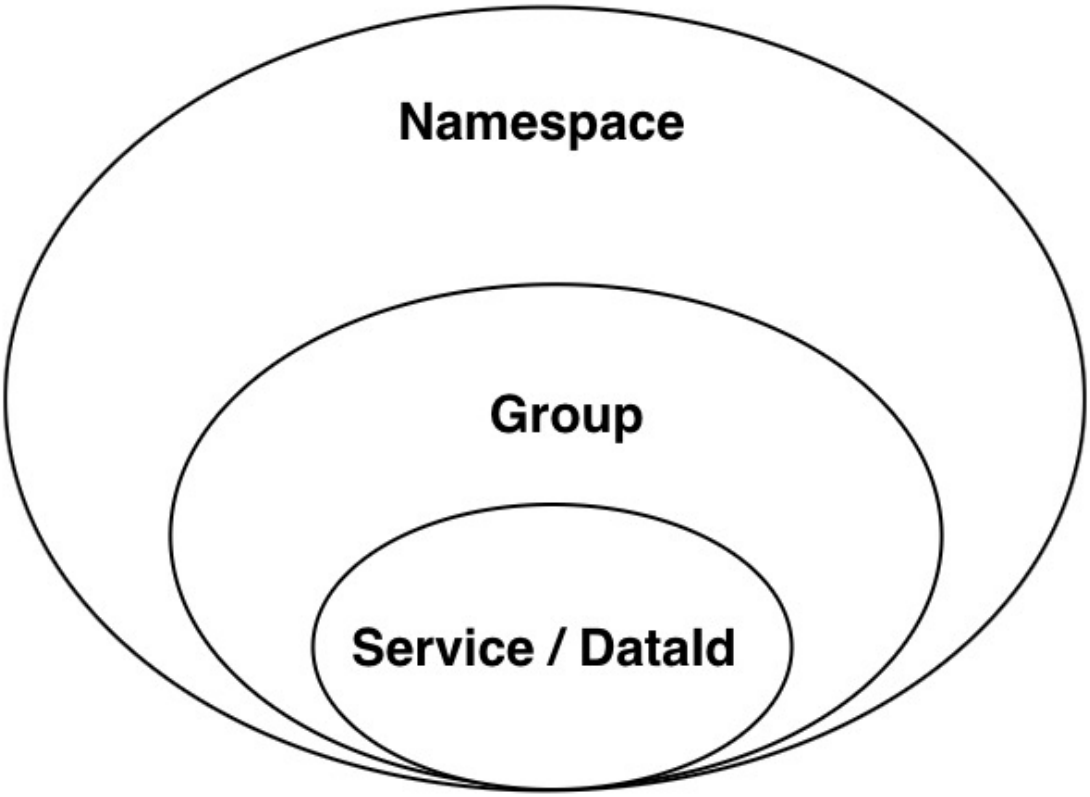
2. namespace环境隔离

一个项目的部署运行，通常需要有多个环境，例如：开发环境、测试环境、生产环境。不同的环境之间的配置不同、部署的代码不同，应当是相互隔离的。

比如：项目已经部署到生产环境、正式运行起来了了，后来开发了新的功能，要部署到测试环境进行测试。那么测试环境的微服务，一定要调用测试环境的目标服务，而不能调用到生产环境上的服务。

Nacos提供了namespace来实现环境隔离功能

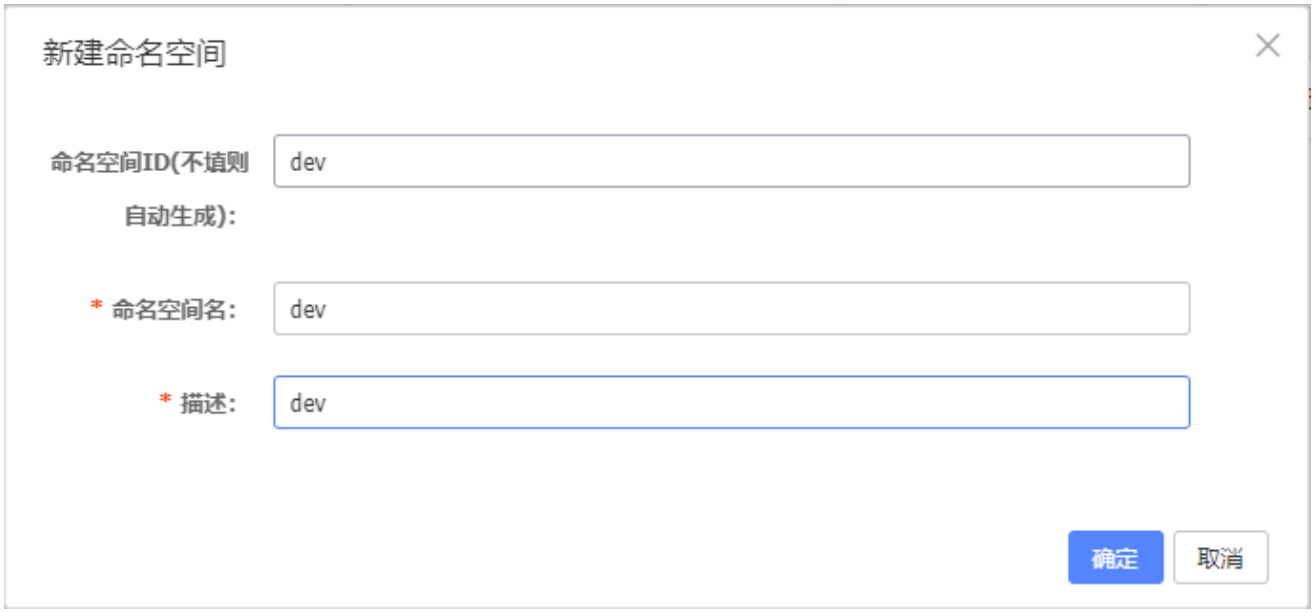
- nacos中可以有多个namespace，namespace下可以有group、service等
- 不同namespace之间相互隔离，例如不同namespace的服务互相不可见



2.1 创建namespace

1. 打开“命名空间”管理界面，点击“创建命名空间”

2. 填写空间的id、名称和描述，点击确定



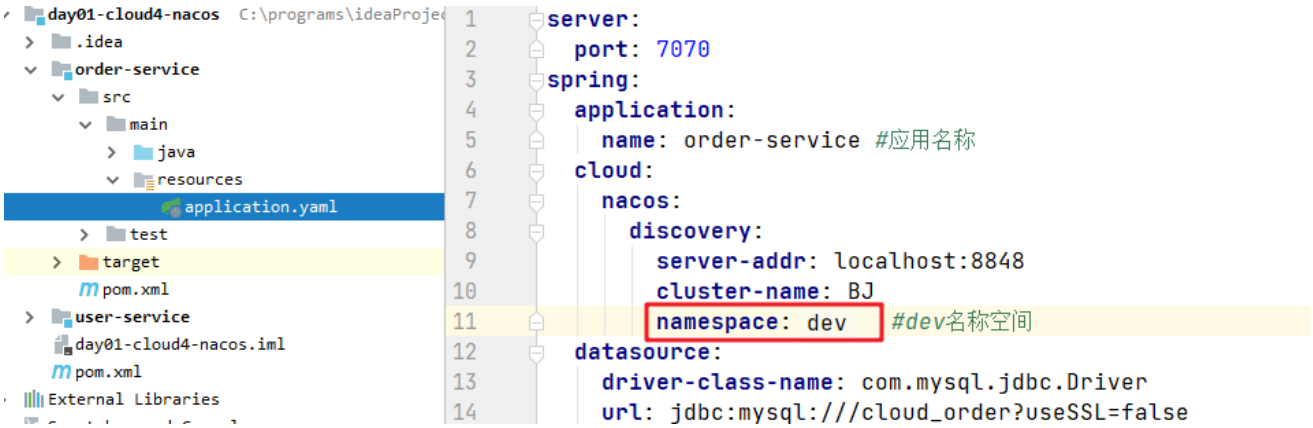
2.2 给微服务指定namespace

目前我们的所有微服务都没有指定namespace，所以默认使用的都是public名称空间。

现在我们把order-service指定名称空间为dev，和user-service不在同一命名空间内。那么订单服务和用户服务之间就是隔离的，不能调用。

- 用户服务user-service：不指定名称空间，还使用默认的public命名空间
- 订单服务order-service：修改配置文件，指定名称空间为dev

然后重启订单服务



2.3 隔离效果

1) 查看nacos控制台

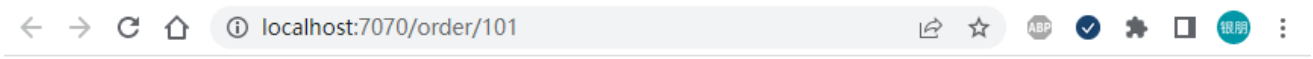
打开Nacos控制台，可以看到

- 用户服务在public命名空间下

- 订单服务在dev命名空间下

2) 隔离效果测试

打开浏览器访问http://localhost:7070/order/101，发现报错了，服务不可用

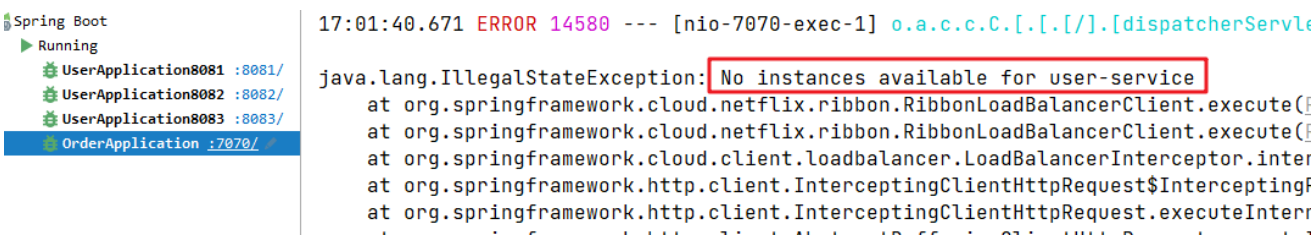


Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Sep 04 17:01:40 CST 2022
There was an unexpected error (type=Internal Server Error, status=500).

idea里报错找不到用户服务，因为命名空间之间是相互隔离、不能访问的



3. 小结