# SpringCloud02-课堂笔记

单体架构：所有功能、业务、代码全部在一个项目里，all in one
  好处：简单。开发简单、部署简单
  缺点：耦合性强，不能针对性的伸缩
  适合：中小型项目
分布式架构：把完整的系统拆分成多个不同的子系统，多个子系统共同组成完整的系统
  好处：耦合性降低了，可以针对性的伸缩
  缺点：复杂。调用关系复杂，完成功能时还有网络开销
  适合：中大型项目
微服务架构：是一种经过良好设计的分布式架构，有以下特征
  单一职责：一个服务只做一件事。保证服务的颗粒度非常细
  服务自治：每个服务要有独立的技术、团队、数据库
  面向服务：所有服务之间交互时，要遵循相同的协议。SpringCloud使用http协议
  服务保护：服务出错时应该把错误限制在小范围内，防止问题的扩大
  适合：大型项目
微服务核心组件：【演讲内容】
  注册中心：解决了服务治理问题。即：服务的地址可能会变化的问题
  远程调用：解决了服务之间互相调用，进行数据交互的问题
  负载均衡：解决了目标服务搭建了集群时，实现服务器负载的均摊
  服务保护：解决了服务出错时可能导致大规模级联失败，从而引起雪崩的问题
  服务网关：给整个微服务提供一个统一的访问入口
  配置中心：解决了众多微服务的配置文件散乱的问题，不方便管理维护
微服务技术方案：实际开发中通常是混用
  Alibaba的Dubbo
  SpringCloud
  SpringCloudAlibaba
远程调用：OpenFeign 【演讲内容】
  使用步骤：
    1. 添加依赖：先锁定SpringCloud的版本；再导入OpenFeign的依赖坐标
    2. 创建Feign接口。
      类上加注解 @FeignClient(value="目标服务名",url="目标服务地址")
      类里的方法
        返回值：是期望得到的结果值类型
        方法名：随意，见名知意
        方法参数：告诉Feign，发HTTP请求时要传递哪些参数。
          必须加注解@RequestBody，@PathVariable，@RequestParam
        方法上加：加@GetMapping等注解，配置API接口的路径
    3. 在启动类上加@EnableFeignClients("扫描包")
  输出Feign的日志
    1. 修改全局的日志级别为debug
    2. 设置Feign的日志级别：修改配置文件，@Bean
      NONE -> BASIC -> HEADERS -> FULL
  Feign的性能优化
    问题：没有使用连接池，每次HTTP请求时都要创建连接，请求响应完成再关闭连接
    解决：使用httpclient的连接池进行优化
注册中心：Nacos
  使用步骤：

    1. 添加依赖：先锁定SpringCloudAlibaba的版本；再导入nacos-discovery的坐标

2. 修改配置：配置一下Nacos注册中心的地址； 每个微服务都必须有一个名称spring.application.name
3. 修改引导类：@EnableDiscoveryClient
远程调用：
Feign的接口上@FeignClient("目标服务名")，不需要再配置url了
nacos的原理：【演讲内容】
微服务启动时：
服务注册：会自动把自己的地址信息上报给注册中心，注册中心会存储起来
心跳续约：会每隔5s一次向Nacos发送心跳进行续约；Nacos如果15s收不到心跳就标记为不健康；Nacos30s收不到心跳，就剔除服务实例的地址
微服务会拉取地址列表：
默认是在第一次远程调用时，从Nacos里拉取服务地址列表
然后默认每10s重新拉取一次
负载均衡：【演讲内容】
两种：
提供者一方的负载均衡：Nginx。适合于 整个服务端的最前沿，直接面对客户端的请求；客户端是无感知
消费者一方的负载均衡：Ribbon。适合于 微服务之间互相调用时，实现负载均衡
Ribbon默认就有负载均衡，是轮询策略
如果要修改负载均衡策略：
修改配置文件：针对某一个服务单独设置负载均衡策略
使用@Bean：针对全局设置的负载均衡策略
饥饿加载和懒加载
懒加载：当微服务第一次远程调用时，才会从Nacos里拉取地址列表；第一次远程调用通常有点慢
饥饿加载：当微服务启动时就立即从Nacos里拉取地址列表；第一次远程调用也不会慢

# 一、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

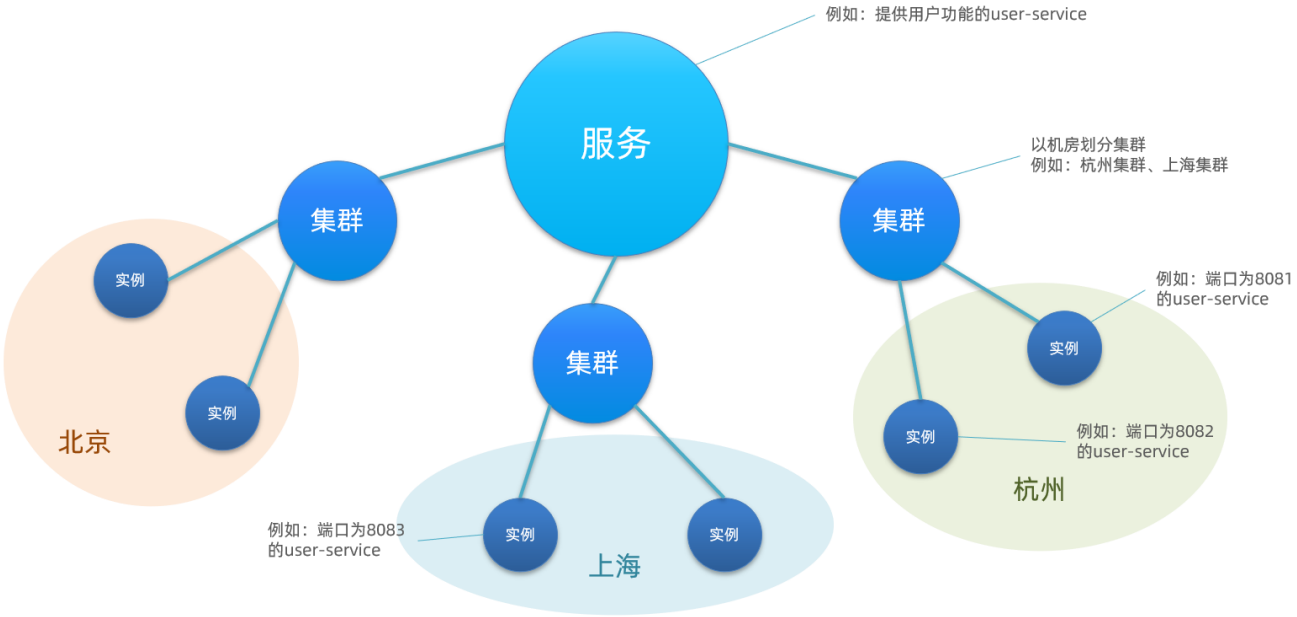
一个服务可以有多个实例，例如我们的user-service，可以有很多实例，例如:

- 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：

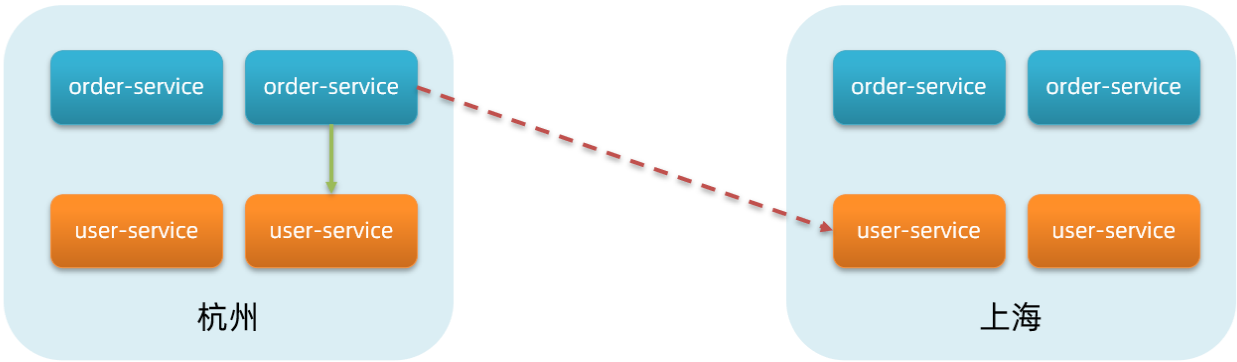
- 127.0.0.1:8081，在上海机房
- 127.0.0.1:8082，在上海机房
- 127.0.0.1:8083，在杭州机房

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

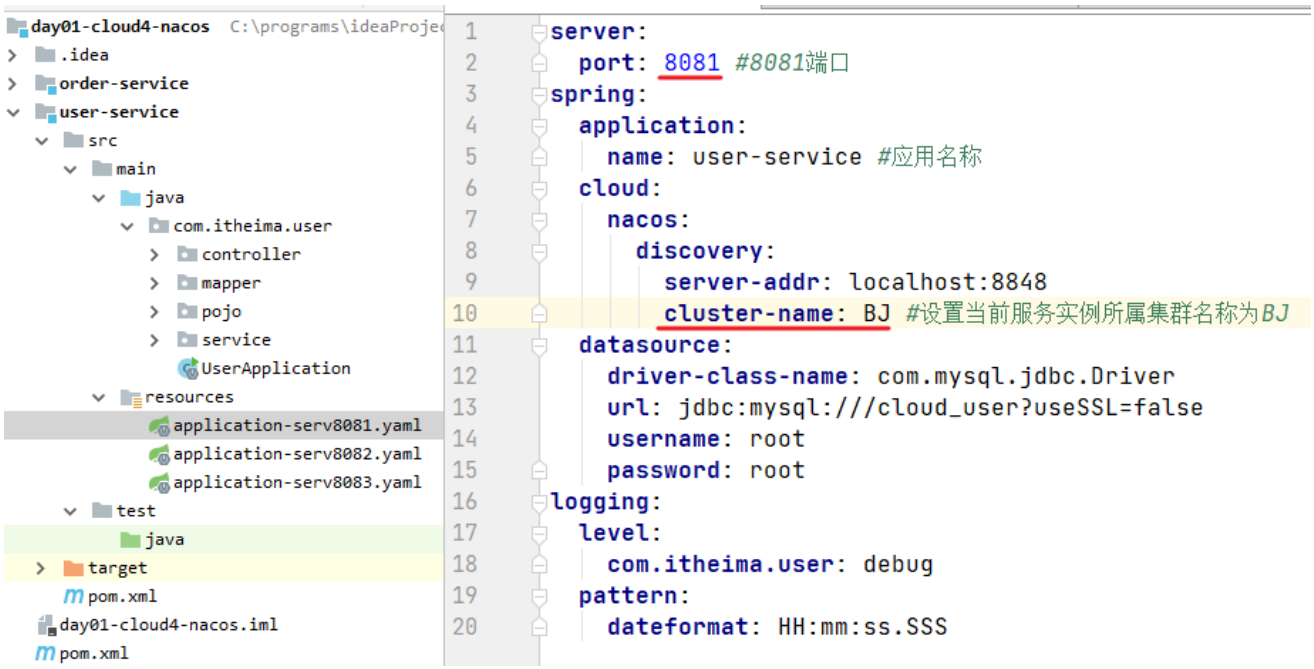
```yaml
spring:
  cloud:
    nacos:
      discovery:
        cluster-name: HZ #设置当前服务实例所属集群名称为HZ
```

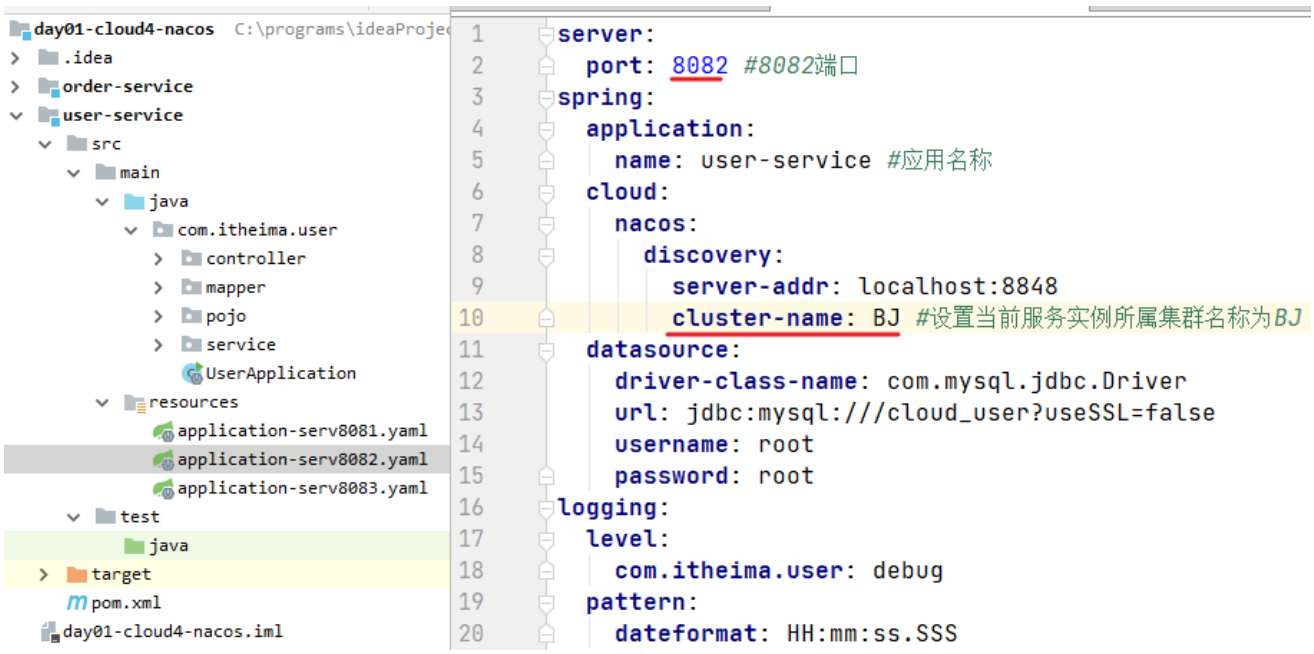
#### 1) 准备配置文件

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



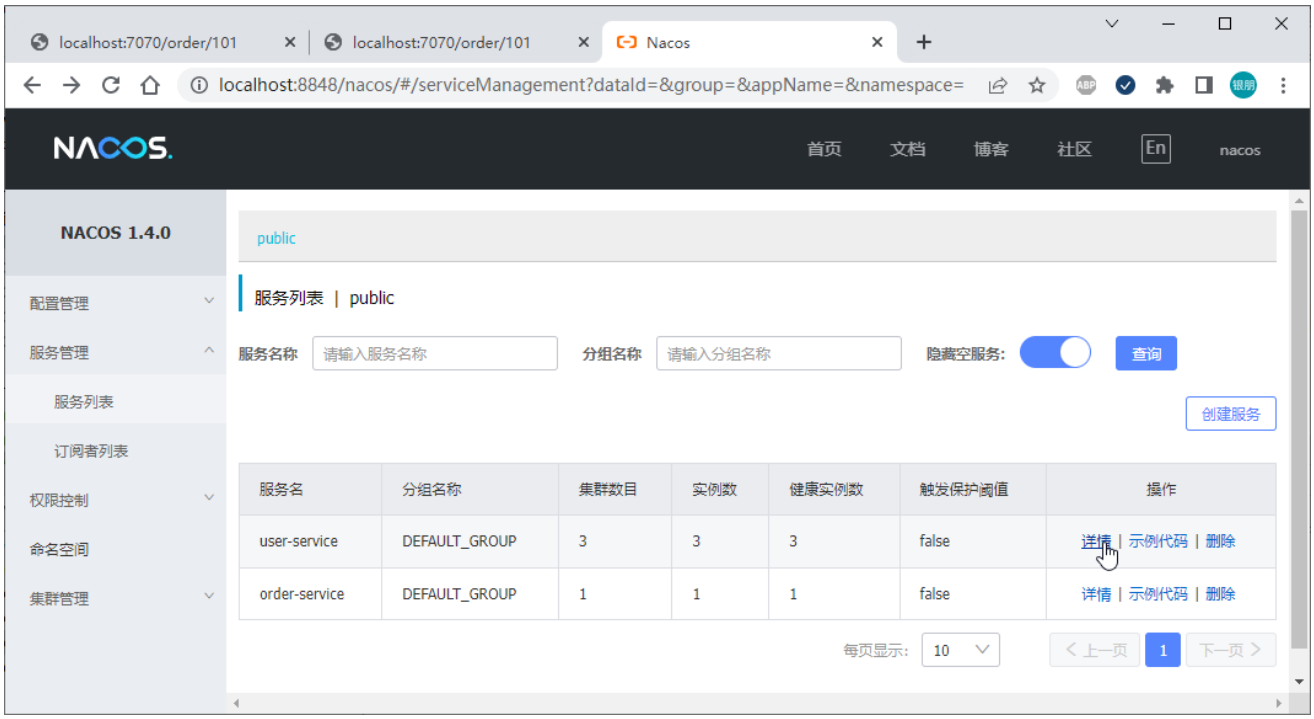

**2) 准备启动链接**

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:
```

```
  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

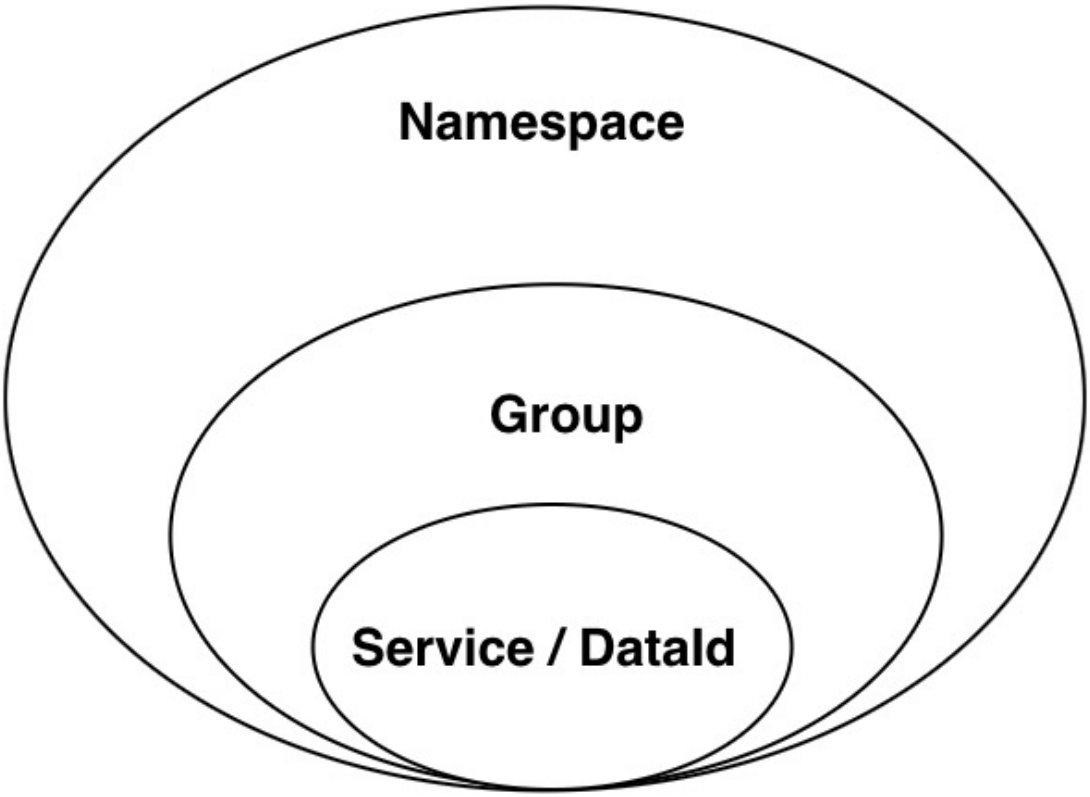
# 2. namespace环境隔离

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

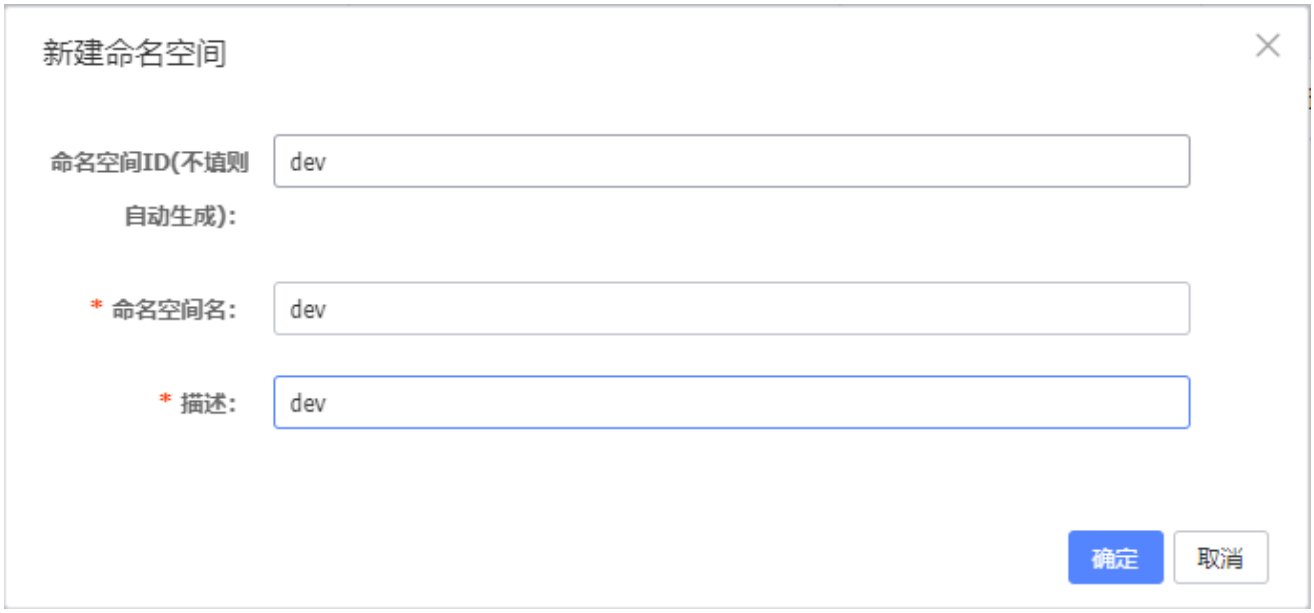
- nacos中可以有多个namespace，namespace下可以有group、service等
- 不同namespace之间相互隔离，例如不同namespace的服务互相不可见




## 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



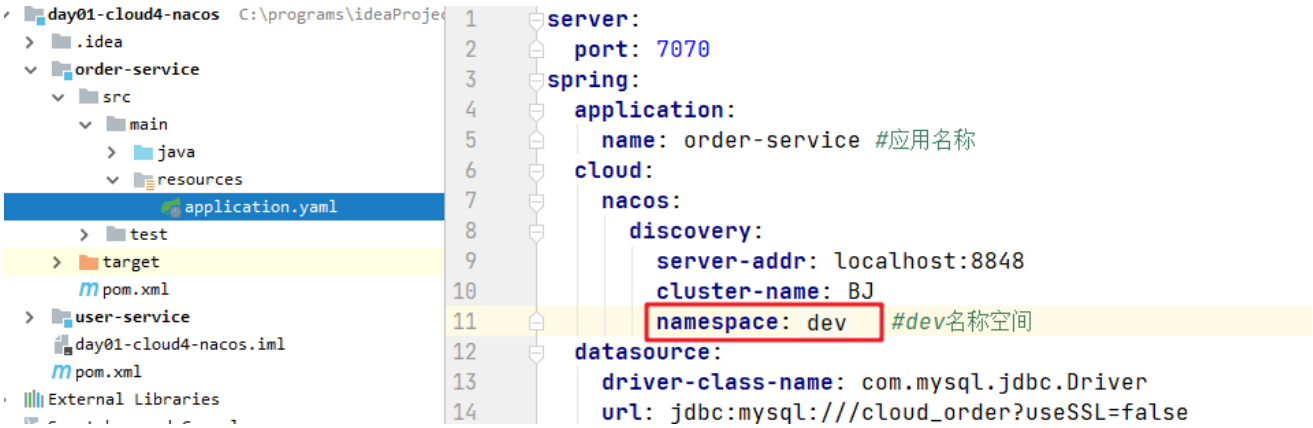

## 2.2 给微服务指定namespace

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

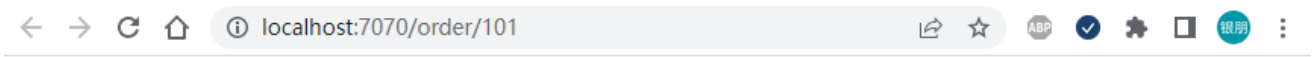
打开Nacos控制台，可以看到

- 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

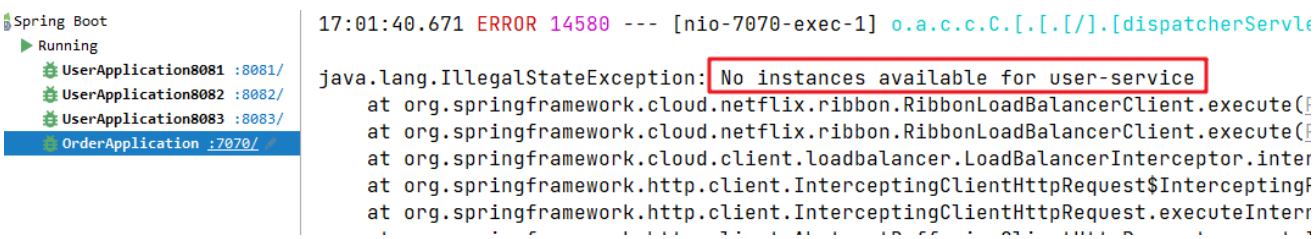



# Whitelabel Error Page

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

Nacos的多级存储模型：把多个服务实例划分成多个不同的子集群。好处：

- 实现异地容灾。如果某个机房出现问题，还有其它机房的服务可用
- 实现同集群优先访问，速度更快。

  北京集群的订单服务，优先访问北京集群的用户服务

  如果北京集群里没有可用的用户服务，就会出现跨集群的访问

```
用户服务user-service
   |---北京集群
   |     |---服务实例1
   |     |---服务实例2
   |---深圳集群
   |     |---服务实例1
   |     |---服务实例2
```

Nacos的多级存储模型具体实现：

1. 给每个服务实例设置所属的集群名称。相同集群名称的服务实例属于同一集群
2. 设置微服务的负载均衡策略为NacosRule

namespace环境隔离：

- 相同namespace的服务之间，可以互相访问
- 不同namespace的服务之间，一定不可能互相访问的
- 实现步骤：
  1. 在Nacos里创建命名空间。设置命名空间的id、名称、描述。如果不设置id，将会自动生成uuid作为id
  2. 给微服务设置所属的命名空间。spring.cloud.nacos.discovery.namespace=命名空间的id

# 二、网关Gateway

本章节学习目标：

- [ ] 了解网关的作用
- [ ] 能够使用Gateway搭建网关
- [ ] 能够使用GatewayFilter
- [ ] 能够自定义GlobalFilter

## 1. 网关简介

### 1.1 网关简介

在微服务架构中，一个系统会被拆分为很多个微服务。那么作为消费者要如何去调用这么多的微服务呢？如果没有网关的存在，我们只能在消费者一端记录每个微服务的地址，然后分别去调用。

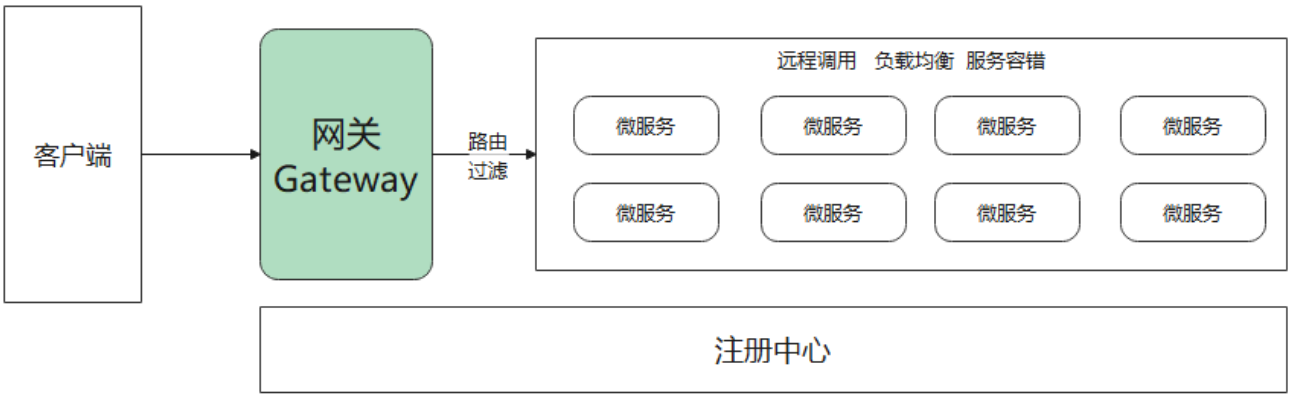
这样的架构，会存在着诸多的问题：

- 客户端多次请求不同的微服务，需要有微服务的地址，增加客户端代码或配置编写的复杂性
- 认证复杂，每个服务都需要独立认证。认证：身份校验，即登录。没有登录，就不允许访问服务
- 存在跨域请求，在一定场景下处理相对复杂

上面的这些问题可以借助**API网关**来解决。所谓的API网关，就是指系统的**统一入口。**它封装了应用程序的内部结构，为客户端提供统一服务。

一些与业务本身功能无关的公共逻辑可以在这里实现，诸如认证、鉴权、监控、路由转发等等。

添加上API网关之后，系统的架构图变成了如下所示：




如果没有网关：
第1个问题：微服务太多，每个微服务有自己的地址。客户端要调用时有很大的麻烦
第2个问题：每个微服务都需要做认证。接收token、校验token、解析token

## 1.2 Gateway简介

zuul 祖鲁：网关技术目前使用已经不多了，因为性能较低

Spring Cloud Gateway是Spring基于Spring5.0、SpringBoot2.0、Project Reactor等技术开发的网关技术

- 旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。
- 它不仅提供统一的路由方式，并且基于Filter链的方式提供了网关基本的功能，例如：安全，监控和限流。
- 它是用于代替NetFlix Zuul的一套解决方案：webflux

Spring Cloud Gateway组件的核心是一系列的过滤器，通过过滤器可以将客户端的请求转发到应用的微服务（这个过程叫路由）。Spring Cloud Gateway是站在整个微服务最前沿的防火墙和代理器，隐藏微服务节点的ip信息、从而加强安全保护。

Spring Cloud Gateway本身也是一个微服务，需要注册到注册中心

Spring Cloud Gatewa的核心功能是：**路由和过滤**

# 2. Gateway入门【重点】

## 2.1 核心概念

### 路由Route

一个路由的配置信息，由一个id、一个目的地url、一组断言工厂、一组过滤器组成。

### 断言Predicate

断言是一种判断规则；如果客户端的请求符合要求的规则，则这次请求将会被路由到目的地

Spring Cloud Gateway的断言函数输入类型是Spring5.0框架中的ServerWebExchange，它允许开发人员自定义匹配来自HTTP请求中任何信息

### 过滤器Filter

Spring Cloud Gateway中的Filter可以对请求和响应进行过滤修改。是一个标准的Spring WebFilter。它分为两类：

- Gateway Filter：局部过滤器(路由过滤器)，应用于单个路由或者一组路由，通常由SpringCloudGateway内置好
- Global Filter：全局过滤器，应用于所有路由

## 2.2 入门示例

### 说明

浏览器通过api网关，将以/user开头的请求转发到用户微服务

### 步骤

1. 创建一个模块：网关模块，导入gateway的依赖
2. 创建引导类：开启服务注册@EnableDiscoveryClient
3. 创建配置文件：
    - 网关的端口
    - 设置服务名称
    - 设置注册中心的地址
    - 配置网关的路由：给每个微服务设置路由信息

### 实现

#### 1) 创建模块导入依赖

```
<dependencies>
  <!-- gateway -->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
  </dependency>

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

#### 2) 创建引导类

```
package com.itheima;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

**3) 编写配置文件**

```
server:
  port: 10000
spring:
  application:
    name: api-gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #注册中心地址
    gateway:
      routes: #路由配置，是个数组
        - id: user-service # 路由id
          uri: lb://user-service #路由目的地的地址： lb 表示从注册中心拉取服务列表，并启用负载均衡
          predicates: #断言，什么样的请求可以到达目标地
            - Path=/user/**
```

### 测试

- 依次启动user-service、api-gateway
- 打开浏览器输入 http://localhost:10000/user/1，发现可以查询到id为1的用户

# 3. Gateway断言【了解】

我们在配置文件中写的`predicates`断言规则只是字符串，这些字符串会被Predicate Factory读取并处理，转变为路由判断的条件。例如Path=/user/**是按照路径匹配，这个规则是由`org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory`类来处理的，像这样的断言工厂在SpringCloudGateway还有十几个，而我们需要掌握的只有`Path`

所有断言工厂的使用方式都是 在网关的路由配置中，使用`predicates`配置的：

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: 路由唯一标识
          uri: lb://user-service #路由目的地的地址
          predicates: #断言，可以配置多个
            - Path=/user/** # - 断言名称=配置值
```

其它断言工厂参考：https://docs.spring.io/spring-cloud-gateway/docs/3.0.4/reference/html/#gateway-request-predicates-factories

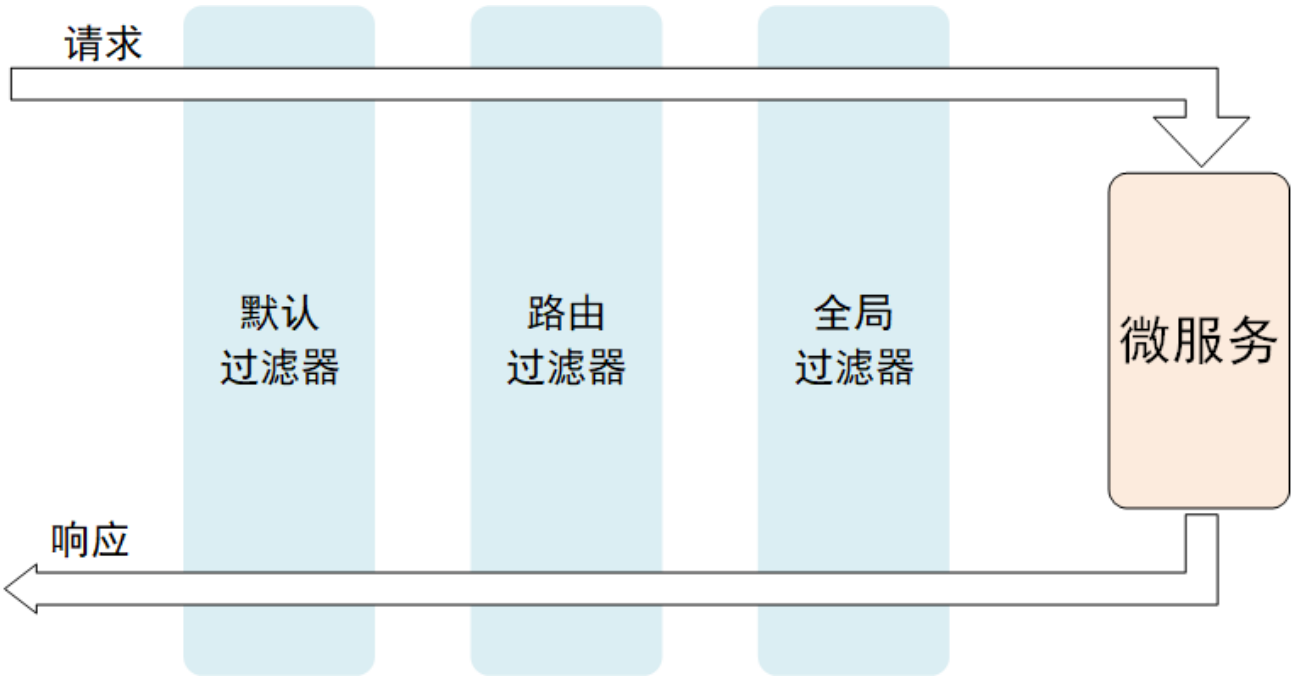
| 名称 | 说明 | 示例 |
| --- | --- | --- |
| After | 是某个时间点后的请求 | - After=2037-01-20T17:42:47.789-07:00[America/Denver] |
| Before | 是某个时间点之前的请求 | - Before=2031-04-13T15:14:47.433+08:00[Asia/Shanghai] |
| Between | 是某两个时间点之前的请求 | - Between=2037-01-20T17:42:47.789-07:00[America/Denver], 2037-01-21T17:42:47.789-07:00[America/Denver] |
| Cookie | 请求必须包含某些cookie | - Cookie=chocolate, ch.p |
| Header | 请求必须包含某些header | - Header=X-Request-Id, \d+ |
| Host | 请求必须是访问某个host（域名） | - Host=**.somehost.org**,.anotherhost.org |
| Method | 请求方式必须是指定方式 | - Method=GET,POST |
| Path | 请求路径必须符合指定规则 | - Path=/red/{segment},/blue/** |
| Query | 请求参数必须包含指定参数 | - Query=name, Jack或者- Query=name |
| RemoteAddr | 请求者的ip必须是指定范围 | - RemoteAddr=192.168.1.1/24 |
| Weight | 权重处理 | |

# 4. Gateway过滤器【重点】

## 4.1 说明

Gateway的过滤器会对请求或响应进行拦截，完成一些通用操作。在Gateway中, Filter的生命周期（**执行时机**）只有两个：

- PRE： 这种过滤器在请求被路由之前调用，可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等
- POST：这种过滤器在路由到微服务以后执行，可用来为响应添加标准的HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等




Gateway的Filter可分为两种

- `GatewayFilter`：应用到单个路由上，是**局部过滤器**，必须要配置到配置文件

  它需要实现GatewayFilterFactory接口，并且需要在配置文件中配置才会生效；

  GatewayFilter也可以配置为默认过滤器，针对所有路由进行过滤
- `GlobalFilter`：应用到所有的路由上，是**全局过滤器**，不需要配置到配置文件

  它不需要在配置文件中配置，只要实现GlobalFilter接口即可。通常用于我们自定义拦截器，实现一些功能的

所有的过滤器都可以参考官方手册 https://docs.spring.io/spring-cloud-gateway/docs/2.2.9.RELEASE/reference/html/#gatewayfilter-factories

## 4.2 GatewayFilter

在SpringCloud Gateway中内置了很多不同类型的网关路由过滤器，

- 这些过滤器如果配置到单个路由下，就只针对这个路由进行过滤
- 如果配置到`default-filters`下，就针对所有路由进行过滤

### 常见的内置局部过滤器

网关过滤器列表如下：

| 过滤器工厂 | 作用 | 参数 |
| --- | --- | --- |

| 过滤器工厂 | 作用 | 参数 |
| --- | --- | --- |
| AddRequestHeader | 为原始请求添加Header | Header的名称及值 |
| AddResponseHeader | 为原始响应添加Header | Header的名称及值 |
| RemoveRequestHeader | 为原始请求删除某个Header | Header名称 |
| RemoveResponseHeader | 为原始响应删除某个Header | Header名称 |
| RequestRateLimiter | 用于对请求限流，限流算法为令牌桶 | keyResolver、rateLimiter、statusCode、denyEmptyKey、emptyKeyStatus |
| ---------- | ---------- | ---------- |
| AddRequestParameter | 为原始请求添加请求参数 | 参数名称及值 |
| DedupeResponseHeader | 剔除响应头中重复的值 | 需要去重的Header名称及去重策略 |
| Hystrix | 为路由引入Hystrix的断路器保护 | `HystrixCommand`的名称 |
| FallbackHeaders | 为fallbackUri的请求头中添加具体的异常信息 | Header的名称 |
| StripPrefix | 用于截断原始请求的路径 | 使用数字表示要截断的路径的数量 |
| PrefixPath | 为原始请求路径添加前缀 | 前缀路径 |
| PreserveHostHeader | 为请求添加一个preserveHostHeader=true的属性，路由过滤器会检查该属性以决定是否要发送原始的Host | 无 |
| RedirectTo | 将原始请求重定向到指定的URL | http状态码及重定向的url |
| RemoveHopByHopHeadersFilter | 为原始请求删除IETF组织规定的一系列Header | 默认就会启用，可以通过配置指定仅删除哪些Header |
| RewritePath | 重写原始的请求路径 | 原始路径正则表达式以及重写后路径的正则表达式 |
| RewriteResponseHeader | 重写原始响应中的某个Header | Header名称，值的正则表达式，重写后的值 |
| SaveSession | 在转发请求之前，强制执行`WebSession::save`操作 | 无 |
| secureHeaders | 为原始响应添加一系列起安全作用的响应头 | 无，支持修改这些安全响应头的值 |
| SetPath | 修改原始的请求路径 | 修改后的路径 |
| SetResponseHeader | 修改原始响应中某个Header的值 | Header名称，修改后的值 |
| SetStatus | 修改原始响应的状态码 | HTTP 状态码，可以是数字，也可以是字符串 |
| Retry | 针对不同的响应进行重试 | retries、statuses、methods、series |
| RequestSize | 设置允许接收最大请求包的大小。如果请求包大小超过设置的值，则返回 `413 Payload Too Large` | 请求包大小，单位为字节，默认值为5M |
| ModifyRequestBody | 在转发请求之前修改原始请求体内容 | 修改后的请求体内容 |
| ModifyResponseBody | 修改原始响应体的内容 | 修改后的响应体内容 |

## 局部过滤器使用示例

在网关中给用户服务和订单服务做了路由配置，要求：

- 当浏览器访问用户服务时，添加一个响应头：abc=user service
- 当浏览器访问订单服务时，添加一个响应头：aaa=order service is strong
- 当浏览器访问任意服务时，添加一个响应头：company=itcast

**配置过滤器**

```yaml
spring:
  application:
    name: api-gateway
  cloud:
    gateway:
      routes:
        - id: user-service #用户服务的路由配置
          uri: lb://user-service
          predicates:
            - Path=/user/**
          filters:
            - AddResponseHeader=abc, user service is strong
        - id: order-service #订单服务的路由配置
          uri: lb://order-service
          predicates:
            - Path=/order/**
          filters:
            - AddResponseHeader=aaa, order service works great
      default-filters: #添加到这里的过滤器，对所有路由都生效
        - AddResponseHeader=company, itcast
```

**测试效果**

- 重启网关服务
- 打开浏览器F12的Network进行抓包
  - 访问网址 http://localhost:10000/order/101，可以看到有两个响应头

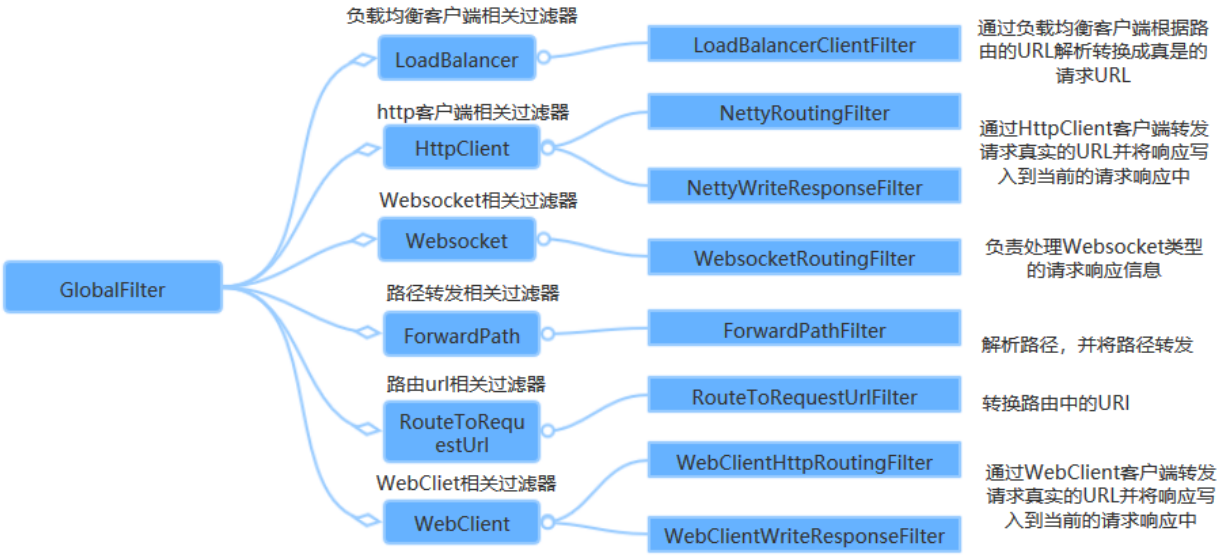

  - 访问网址 http://localhost:10000/user/1，可以看到有两个响应头


## 4.3 GlobalFilter全局过滤器

## 内置全局过滤器

全局过滤器作用于所有路由，而且无需配置。通过全局过滤器可以实现对权限的统一校验，安全性验证等功能。

SpringCloud Gateway内部也是通过一系列的内置全局过滤器对整个路由转发进行处理。如下：




## 自定义全局过滤器【重点】

### 说明

内置的过滤器已经可以完成大部分的功能，但是对于企业开发的一些业务功能处理，还是需要我们自己编写过滤器来实现的。

### 示例

- 全局过滤器类必须实现GlobalFilter接口，重写filter方法，在filter方法里实现过滤逻辑
- 全局过滤器类可以实现Ordered接口，重写getOrder方法，如果需要设置过滤器执行顺序的话
- 类上要加注解`@Component`

```
@Component
public class DemoGlobalFilter implements Ordered, GlobalFilter {
    /**
     * 执行过滤的方法
     */
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        //如果要处理请求，就从exchange里获取request对象
        ServerHttpRequest request = exchange.getRequest();
        // 获取请求路径
        System.out.println("本次请求路径：" + request.getURI());
        // 获取请求头
        System.out.println("请求头Host：" + request.getHeaders().getFirst("Host"));


        //如果要处理响应，就从exchange里获取response对象
        ServerHttpResponse response = exchange.getResponse();
```

```
        // 设置响应状态码
        response.setStatusCode(HttpStatus.UNAUTHORIZED);
        // 设置响应cookie
        response.addCookie(ResponseCookie.from("cookieName", "cookieValue").build());
        // 结束本次请求，并返回响应
        // return response.setComplete();

        //放行
        return chain.filter(exchange);
    }

    /**
     * 设置过滤器的执行顺序，值越小，执行的越早
     */
    @Override
    public int getOrder() {
        return 0;
    }
}
```

**练习：校验token**

下面，我们一起通过代码的形式自定义一个过滤器，实现用户鉴权

- 如果本次请求携带了请求头Authorization（token值），则放行；
- 否则不放行，并且返回状态码401

```
package com.itheima.filter;

import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

import java.util.List;

@Component
public class AuthGlobalFilter implements GlobalFilter , Ordered{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        //获取请求头Authorization
        List<String> authorization = exchange.getRequest().getHeaders().get("Authorization");
        //如果获取不到Authorization
        if (authorization == null || authorization.size() == 0) {
            System.out.println("鉴权失败");
            //设置响应状态码
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            //结束本次调用
            return exchange.getResponse().setComplete();
        }
```

```
        //放行到下一个过滤器
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        //数值越小，执行的优先级越高
        return 0;
    }
}
```

## 4.4 过滤器执行顺序【了解】

当一次请求进入网关后，网关会：

1. 找到所有能拦截本次请求的所有过滤器，包括：GatewayFilter、GlobalFilter
2. 根据所有过滤器的Order排序值进行排序，值越小，优先级越高，执行的越早

   `GlobalFilter`全局过滤器的排序值：通过实现`Ordered`接口或者添加`@Order`注解来指定Order值

   `GatewayFilter`局部过滤器排序值：由框架指定order值，默认按照声明的顺序从1开始递增
3. 如果过滤器的Order值相同，优先级：GlobalFilter > defaultFilter > GatewayFilter

# 5. 跨域问题

## 5.1 浏览器的同源策略

### 什么是同源策略

1995年，同源策略由 Netscape 公司引入浏览器。目前，所有浏览器都实行这个同源策略。它是指：一个页面，只允许访问与页面同源的某些资源。

- 所谓的同源包含：同协议、同域名(同IP)、同端口。假如有一个资源是`http://www.itcast.cn/a.html`，那么：

  `https://www.itcast.cn/user/1`：不同源，因为协议不同

  `http://itcast.cn/user/1`：不同源，因为域名不同

  `http://www.itcast.cn:81/user/1`：不同源，因为端口不同

  `http://www.itcast.cn/user/1`：同源，因为同协议、同域名、同端口
- 被同源限制的资源有：
  - Cookie、LocalStorage 和 IndexDB：只能同源的页面进行访问
  - DOM和js对象：只能同源的页面才能获得
  - AJAX：只能向同源的资源发Ajax请求

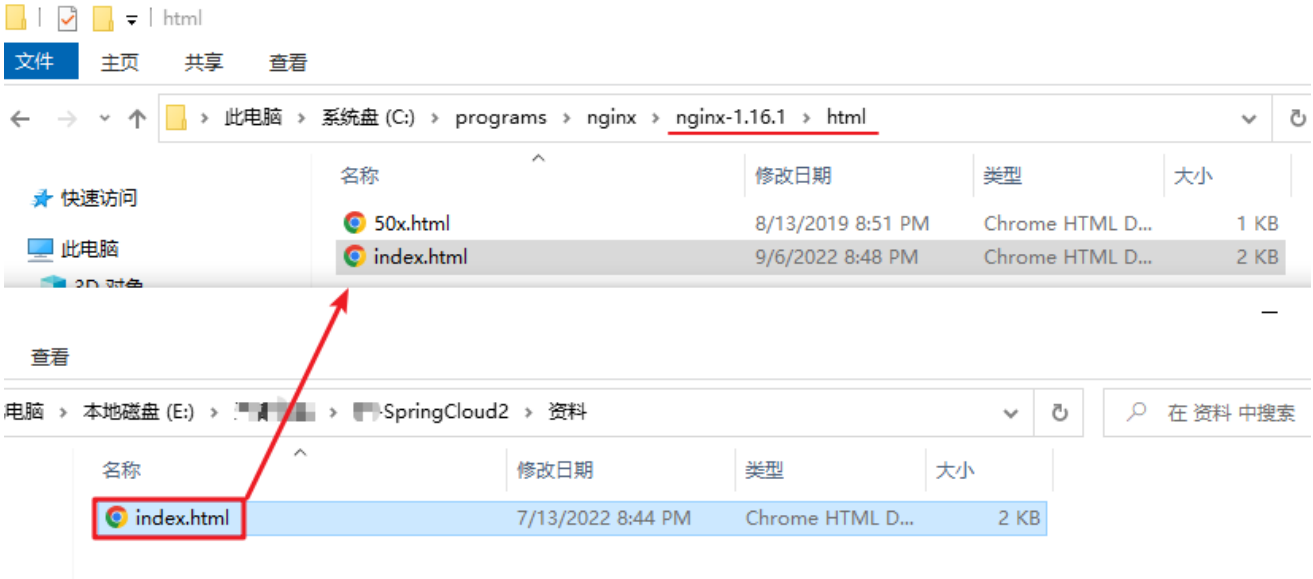
### 什么是跨域问题

如果`http://localhost:80/index.html`页面上要发起一个Ajax请求，请求的目的地是：`http://localhost:8080/user/1`，这就是一个跨域Ajax请求了。受限于浏览器的同源策略，这次请求会失败

但是目前流行的开发方式是前后端分离，即前端资源使用nginx部署到单独的服务器上，服务端项目部署到其它服务器上，这样的情况下，跨域请求就不可避免了。我们该如何规避浏览器的同源策略，允许浏览器跨域发送Ajax请求呢?

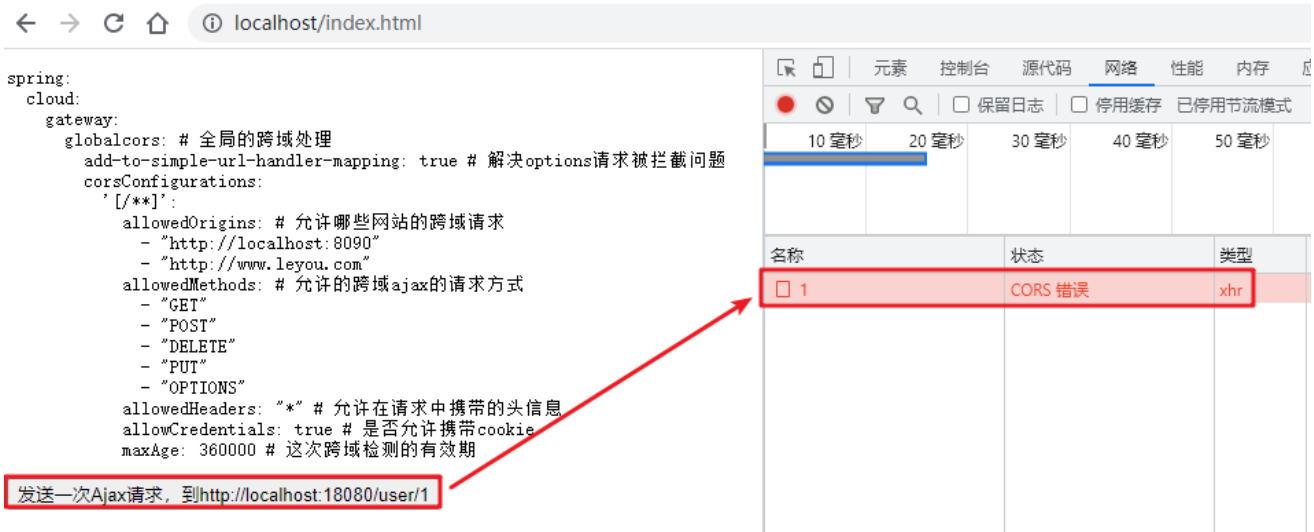
## 5.2 模拟跨域问题

1. 把资料里的`index.html`放到Nginx的html目录里，nginx端口使用80




2. 启动Nginx，打开浏览器访问http://localhost/index.html

   点击页面上的按钮，发送一次Ajax请求。使用抓包工具可以看到报错




## 5.3 解决跨域问题

只需要在网关里添加如下配置即可：

```yaml
spring:
  cloud:
    gateway:
      globalcors: # 全局的跨域处理

        add-to-simple-url-handler-mapping: true # 解决options请求被拦截问题
```

```
    corsConfigurations:
      '[/**]':
        allowedOrigins: "*" # 允许哪些网站的跨域请求。 *表示任意网站
        allowedMethods: # 允许的跨域ajax的请求方式
        - "GET"
        - "POST"
        - "DELETE"
        - "PUT"
        - "OPTIONS"
        allowedHeaders: "*" # 允许在请求中携带的头信息
        allowCredentials: true # 是否允许携带cookie
        maxAge: 360000 # 这次跨域检测的有效期
```

# 6. 小结

服务网关的两大核心功能：路由和过滤

服务网关使用的技术：SpringCloudGateway

SpringCloudGateway的使用步骤：

1. 添加依赖：gateway的依赖，nacos-discovery的依赖
2. 创建配置文件：

   基本配置：端口，应用服务名，nacos的地址

   网关配置：配置各个服务的路由

   ```
   server:
     port: 10010
   spring:
     application:
       name: gateway
     cloud:
       nacos:
         discovery:
           server-addr: 127.0.0.1:8848
       gateway:
         routes:
           - id: user #路由的id。可以不设置这一个参数，如果设置了就不能重复
             uri: lb://user-service #路由的目标地址。写法：lb://目标服务名
             predicates:
               - Path=/user/** #如果请求路径以/user开头的，当前路由会把请求分发到目标uri地址
           - id: order
             uri: lb://order-service
             predicates:
               - Path=/order/**
   ```

3. 创建启动类：@EnableDiscoveryClient

网关过滤器-GatewayFilter

- 由网关服务提供好的过滤器，我们直接配置就可以使用了
- 可以给某个路由配置，也可以作为默认的过滤器配置

```yaml
spring:
  gateway:
    default-filters: #针对所有路由全部生效
      - 过滤器名称=参数值
      - 过滤器名称=参数值
    routes:
      - id: user #路由的id。可以不设置这一个参数，如果设置了就不能重复
        uri: lb://user-service #路由的目标地址。写法：lb://目标服务名
        predicates:
          - Path=/user/** #如果请求路径以/user开头的，当前路由会把请求分发到目标uri地址
        filters:
          - 过滤器名称=参数值 #只针对当前路由生效
          - 过滤器名称=参数值
      - id: order
        uri: lb://order-service
        predicates:
          - Path=/order/**
```

GlobalFilter全局过滤器，使用步骤：

1. 创建类，实现Ordered、GlobalFilter接口，重写 接口的方法

   Ordered里的getOrder方法：返回整数， 整数越小优先级越高，执行的越早

   GlobalFilter里的filter方法：编写过滤逻辑的。有两个参数

   - exhange：用于获取request和response对象的

     exchange.getRequest()：获取请求对象

     - 获取请求路径：request.getURI().getPath()
     - 获取请求方式：request.getMethodValue()
     - 获取请求头：request.getHeaders().getFirst("请求头名称")

     exchange.getResponse()：获取响应对象

     - 设置响应状态码：response.setStatusCode(HttpStatus.枚举项)

   - chain：过滤器链，用于放行的

     如果要放行：return chain.filter(exchange);

     如果不放行：先设置状态码，然后return response.setComplete()

2. 类上要加 @Component

要能判断什么样的请求是跨域：同源要求 必须是同协议、同ip(域名)、同端口

- 假如从http://localhost:8888/abc.html里发起的Ajax请求
- 判断：

  请求到 https://localhost:8888/user/1，跨域了

  请求到 http://127.0.0.1:8888/user/1，跨域了

  请求到 http://localhost:8080/user/1，跨域了

  请求到 http://localhost:8888/user/1，不跨域 ，是同源

# 三、配置中心Nacos

# 1. 配置中心简介

## 1.1 微服务的配置问题

首先我们来看一下,微服务架构下关于配置文件的一些问题：

1. 配置文件相对分散，不利于维护。

   在一个微服务架构下，配置文件会随着微服务的增多变的越来越多，而且分散在各个微服务中，不好统一配置和管理。

2. 配置文件不方便区分环境。

   微服务项目可能会有多个环境，例如：测试环境、预发布环境、生产环境。每一个环境所使用的配置理论上都是不同的，一旦需要修改，就需要我们去各个微服务下手动维护，这比较困难。

3. 配置文件无法实时更新。

   我们修改了配置文件之后，必须重新启动微服务才能使配置生效，这对一个正在运行的项目来说是非常不友好的。

基于上面这些问题，我们就需要**配置中心**的加入来解决这些问题。

1. 可以统一管理配置文件
2. 配置文件可以区分环境。给每个微服务设置多个配置文件，在启动微服务时，可以设置拉取指定的配置文件

   比如：一个微服务，在配置中心提供多个配置文件。一个开发环境的配置文件，一个测试环境的配置文件，一个生产环境的配置文件

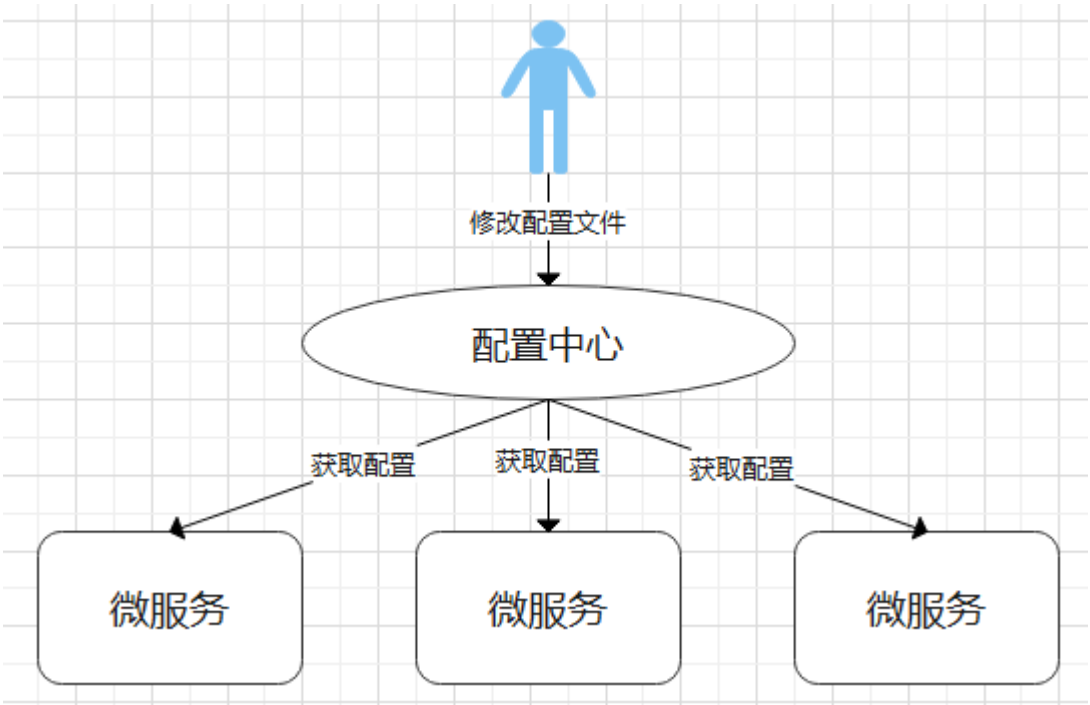
   然后：在启动微服务时，可以指定要拉取哪个环境的配置文件

3. 配置文件可以实时更新，不需要重启微服务

## 1.2 配置问题的解决方案

配置中心的思路是：

- 首先把项目中各种配置全部都放到一个集中的地方进行统一管理。
- 当各个服务需要获取配置的时候，就来配置中心的接口拉取自己的配置。
- 当配置中心中的各种参数有更新的时候，也能通知到各个服务实时的过来同步最新的信息，使之动态更新。




SpringCloudAlibaba Nacos本身就可以管理配置文件。

我们只要把配置文件放到nacos上，微服务就可以从nacos里拉取配置、实现配置的动态刷新了

# 2. 配置中心Nacos入门【重点】

SpringCloud本身提供了一种配置中心：SpringCloudConfig，使用相对麻烦

SpringCloudAlibaba提供了配置中心：Nacos，使用更简单

1. 把配置文件托管到配置中心Nacos
2. 修改微服务

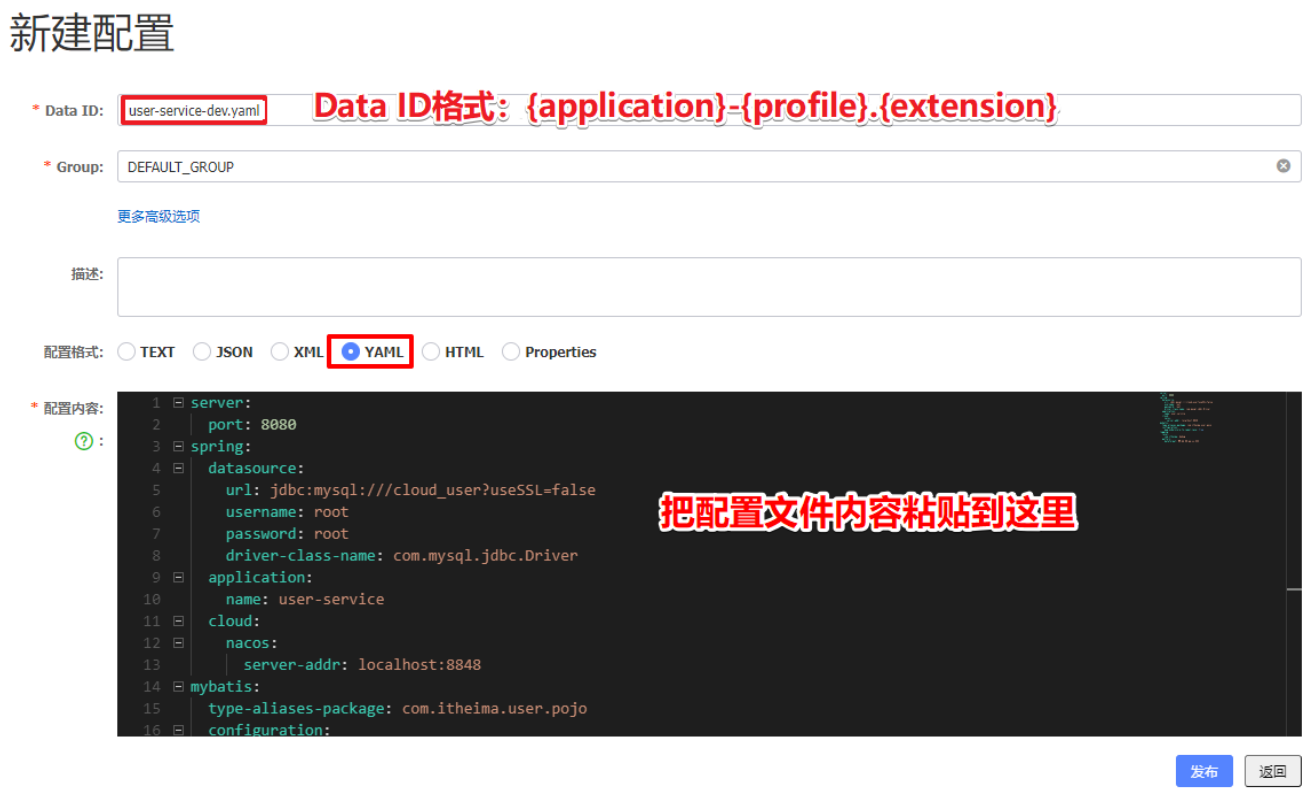
   添加依赖坐标nacos-config

   删除application.yaml，创建bootstrap.yaml， 在bootstrap.yaml里，配置 从哪拉取配置文件
3. 启动微服务，测试功能是否正常

## 2.1 把配置文件托管到nacos

### 1) 新增配置


### 2) 设置配置信息



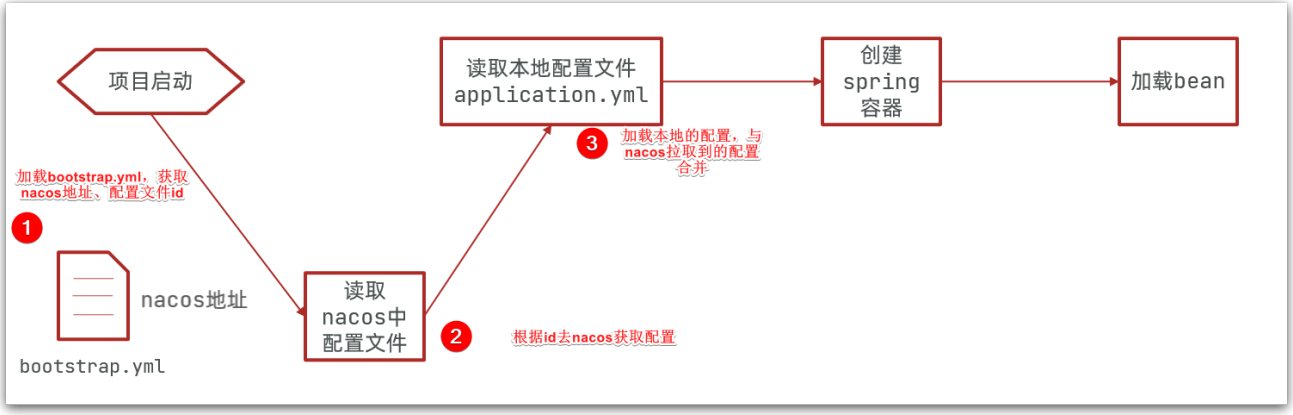

> 注意：项目的核心配置，需要热更新的配置才有放到nacos管理的必要。
>
> 基本不会变更的一些配置还是保存在微服务本地比较好。

## 2.2 从nacos上拉取配置

微服务要拉取nacos中管理的配置，并且与本地的application.yml配置合并，才能完成项目启动。

但如果尚未读取application.yml，又如何得知nacos地址呢?

因此spring引入了一种新的配置文件：bootstrap.yaml文件，会在application.yml之前被读取，流程如下：




### 1) 添加依赖坐标

- 在用户服务的pom.xml中，添加nacos配置中心的坐标
- 注意：一旦引入了nacos配置中心的坐标，就必须有`bootstrap.yaml`配置文件

```
<!--nacos配置-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

### 2) 添加bootstrap.yaml

- 先把用户服务的application.yaml删除掉（配置文件的所有内容，都已经托管到nacos上了）
- 再给用户服务创建`bootstrap.yaml`。注意：名称必须是`bootstrap`+后缀名

```
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848 #配置中心nacos的地址
        prefix: user-service # 要加载的配置文件，{application}部分
        file-extension: yaml # 要加载的配置文件，{extension}后缀名部分
  profiles:
    active: dev # 激活的环境名称，配置文件中{profile}部分
```

### 3) 测试

打开浏览器，所有功能都可以正常使用，说明已经成功拉取到了配置参数

打开浏览器，访问 http://localhost:8080/user/company，可以看到公司名称，也说明成功拉取到了配置参数

## 3. 配置参数的动态刷新

Nacos支持动态刷新配置，也叫热更新：只要在nacos里修改了配置，微服务不需要重启，就会自动拉取最新的配置参数。

有两种实现方案：

- 方案1：使用@Value获取参数值，并在bean对象上添加注解@RefreshScope
- 方案2【推荐】：使用@ConfigurationProperties封装配置参数，会自动拉取最新配置

## 1) 方案1: @Value和@RefreshScope

1. 在UserController里读取参数
    - 在UserController里使用@Value读取了company.name配置参数
    - 在UserController上添加注解`@RefreshScope`

```
@RestController
@RefreshScope
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @Value("${company.name}")
    private String companyName;

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

    @GetMapping("/company")
    public String company(){
        return companyName;
    }
}
```

2. 打开浏览器访问 http://localhost:7070/user/company，先查看一下company.name原始值
3. 在nacos里修改`company.name`的值

    把`company.name`的值修改为“传智教育”
4. 打开浏览器刷新 http://localhost:7070/user/company，可以看到已经得到最新的值了

## 2) 方案2: @ConfigurationProperties

1. 创建Company类，用于封装配置参数

```
@Data
@ConfigurationProperties(prefix = "company")
public class Company {
    private String name;
}
```

2. 在引导类上添加`@EnableConfigurationProperties`，启动Company类

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

3. 修改UserController

   去掉`@RefreshScope`注解

   注入`Company`对象

   添加方法，浏览器访问时，把Company对象返回给客户端 显示到页面上

```
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @Value("${company.name}")
    private String companyName;

    @Autowired
    private Company company;

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

    @GetMapping("/company")
    public String companyStr(){
        return companyName;
    }

    @GetMapping("/companyObj")
    public Company companyObj(){
        return company;
    }
}
```

4. 功能测试

   先打开浏览器访问 http://localhost:8080/user/companyObj，查看原始参数值

   在Nacos里修改company.name的值

   再打开浏览器，直接刷新页面，可以看到参数值已经变成最新的了

# 4. 配置共享【拓展】

在实际开发中，一个服务通常有多个配置文件。在不同环境中，只要激活对应的配置文件即可。例如：

- user-service-dev.yaml：作为用户服务的开发环境配置
- user-service-test.yaml：作为用户服务的测试环境配置
- user-service-prod.yaml：作为用户服务的生产环境配置

但是这多个环境的配置文件中，可能有大部分配置参数都是完全相同的，如果在每个文件里都复制一份的话，修改维护起来会比较麻烦，这时候可以使用共享的配置文件的：

- 再创建一个user-service.yaml：不带环境标识的配置文件，会被所有环境共享

## 4.0 准备多环境配置

### 4.0.1 准备dev和test两个配置文件

在Nacos配置中心里已经有了`user-service-dev.yaml`。

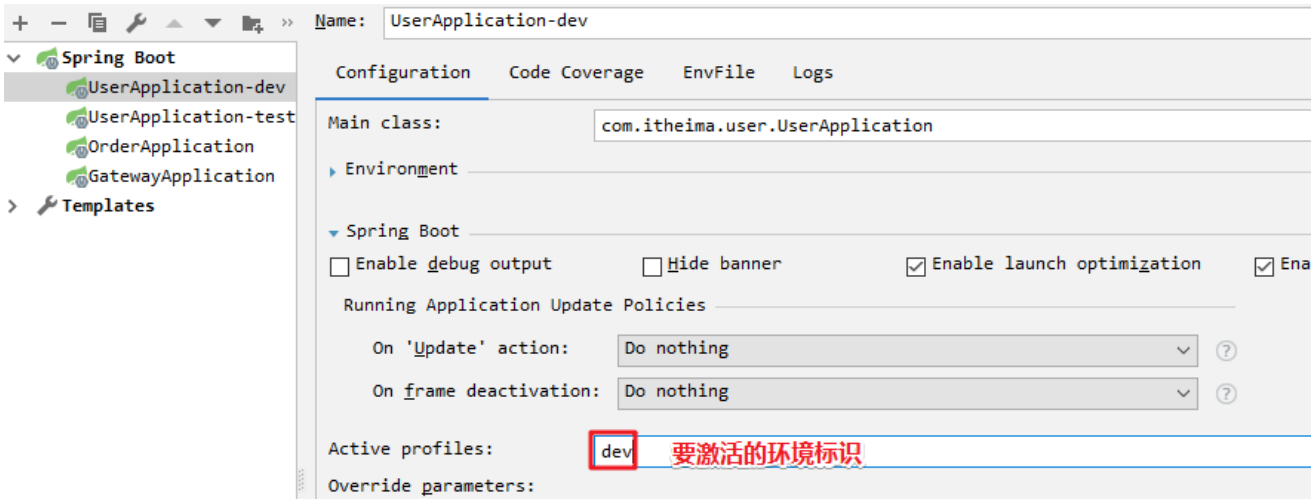
我们再增加一个配置文件`user-service-test.yaml`，内容如下：

```yaml
spring:
  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
company:
  name: 传智教育-test
```

### 4.0.2 准备多环境的启动链接

在idea里创建dev和test的启动链接，创建方式如下：

- dev环境，激活`dev`
- test环境，激活`test`




# 4.1 多环境配置共享

## 4.1.1 创建共享配置文件

在Nacos的配置中心里，创建一个用户服务的共享配置文件


## 4.1.2 读取配置参数值

修改UserController，读取共享配置项`aaa`的值

```
@RestController
@RequestMapping("/user")
public class UserController {
    ...略...;

    @Value("${aaa}")
    private String shareValue;

    @GetMapping("/share")
    public String share(){
        return shareValue;
    }
}
```

### 4.1.3 启动测试

同时启动两个环境的用户服务，

打开浏览器访问：

- 开发环境dev：http://localhost:8080/user/share，页面上可以看到共享的配置项“共享配置值”
- 测试环境test：http://localhost:8081/user/share，页面上可以看到共享的配置项“共享配置值”

## 4.2 配置共享的优先级【了解】

实际开发中，通常是：

- 在项目内使用bootstrap.yaml中配置 不变的、不需要修改的参数
- 在配置中心里配置可能变化的、可能修改的参数

当nacos、服务本地同时出现相同属性时，优先级高的生效。优先级从高到低如下：

1. Nacos里的`服务名-{profile}.yaml`：Nacos里激活的某一环境的配置文件
2. Nacos里的`服务名.yaml`：Nacos里的共享配置文件
3. 本地的`application-{profile}.yaml`
4. 本地的`application.yaml`

注意：实际开发中不建议在不同地方配置相同的参数，不要给自己制造困难

# 5. 小结

配置中心：解决了配置文件太散乱难以管理的问题

配置中心使用步骤：

1. 把配置参数托管到Nacos里

   创建配置时，dataId的写法：`应用服务名-环境标识.后缀名`

2. 微服务里：创建一个bootstrap.yaml，文件里要有

   配置中心的地址，表示要从这里拉取配置文件

   要拉取哪个文件

```yaml
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848 #配置中心的地址。要从这个地址里拉取配置文件
        prefix: user-service #要拉取的配置文件的 “应用服务名”前缀部分
        file-extension: yaml #要拉取的配置文件的 后缀名部分
  profiles:
    active: dev #要激活dev环境的配置文件
```

参数热更新：

- 方式1：使用@Value读取参数，并在bean对象上加 @RefreshScope
- 方式2：直接使用@ConfigurationProperties读取参数，本身就具备热更新的能力