

# Spring Cloud Alibaba

## (2022.0.0.0-RC2 版)

### 课程讲义



主讲：动力哥

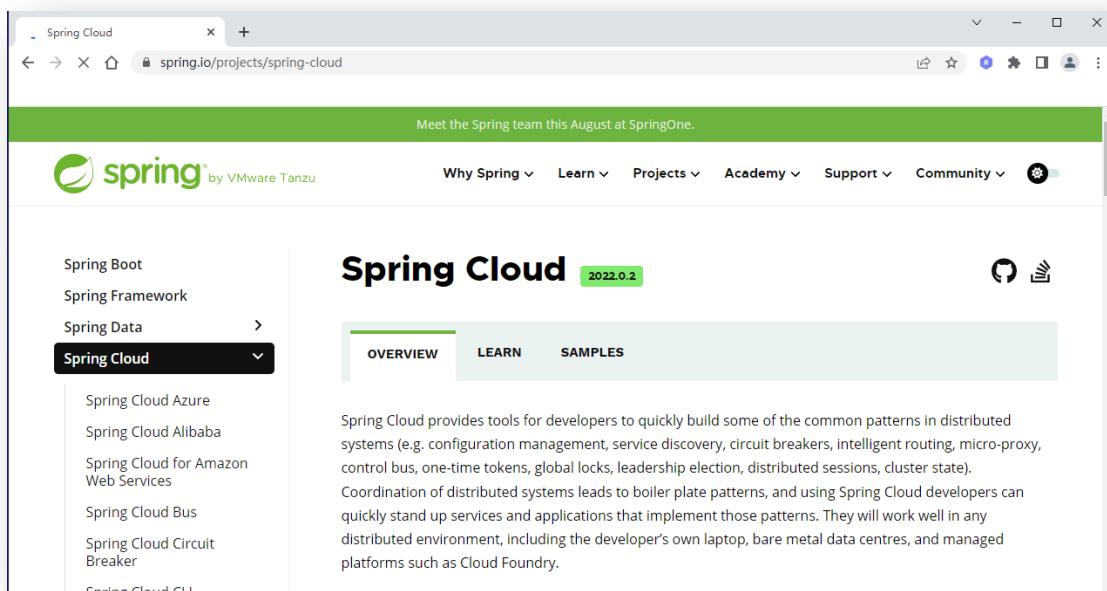
2023

# 第1章 Spring Cloud Alibaba 入门

## 1.1 Spring Cloud 简介

### 1.1.1 官网简介

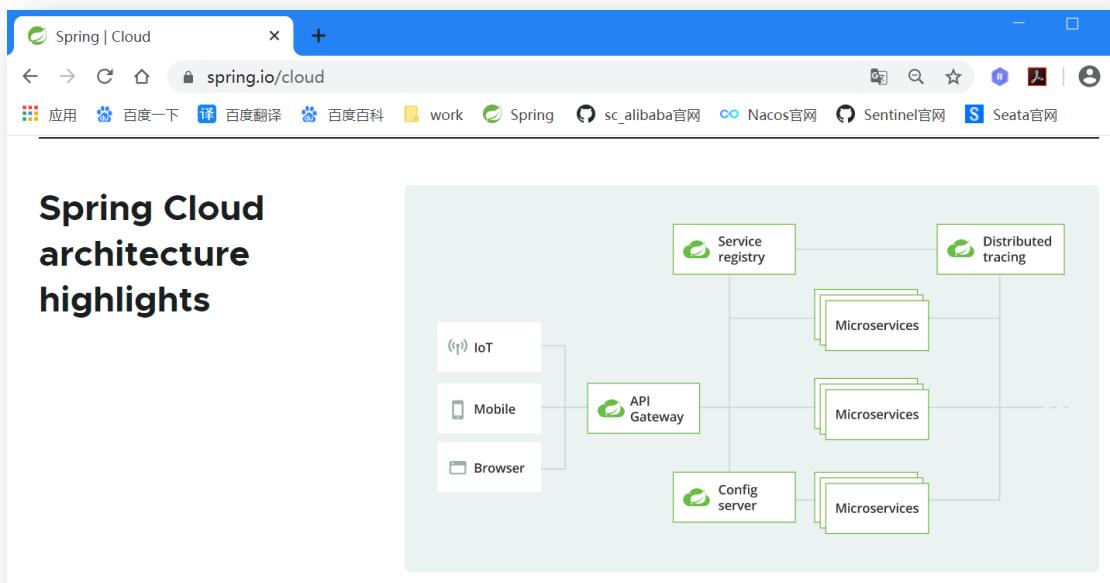
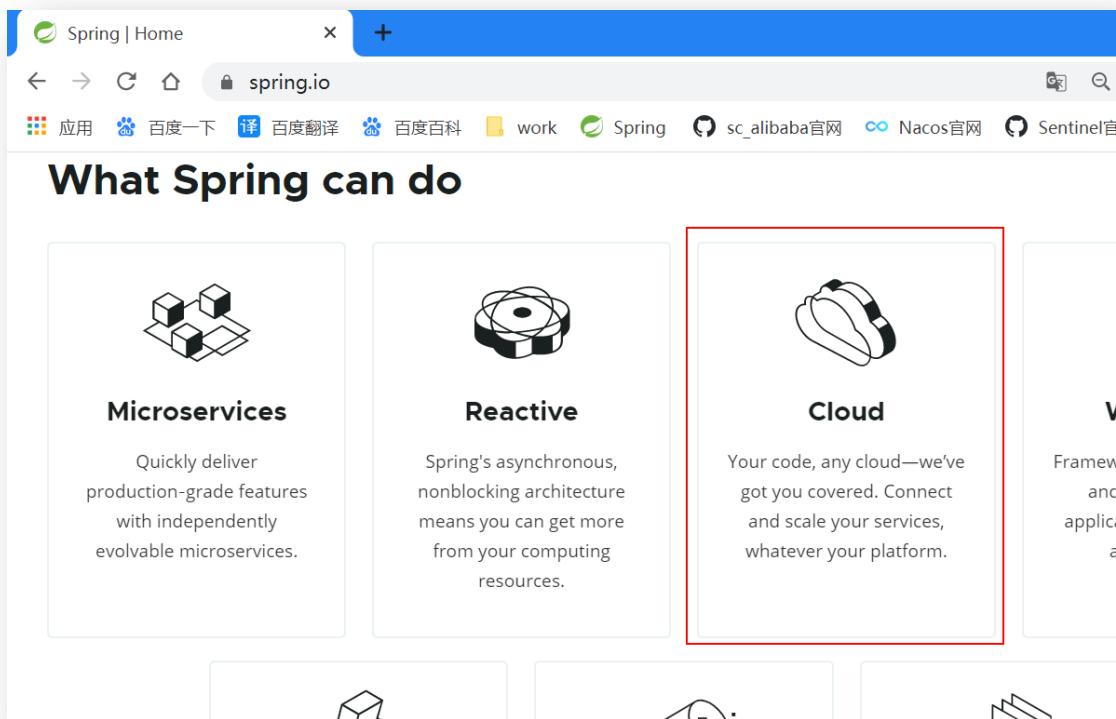
打开 Spring Cloud 官网 <https://spring.io/projects/spring-cloud> 首页，可以看到 Spring Cloud 的简介。



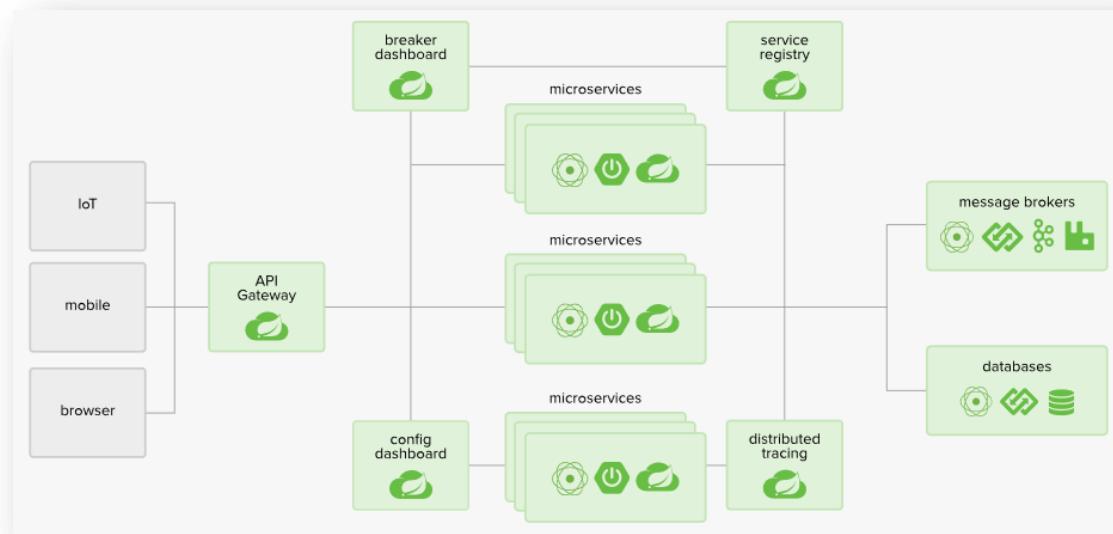
**【原文】**Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

**【翻译】**springcloud 为开发人员提供了在分布式系统中快速构建一些常见模式的工具（例如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性令牌、全局锁、领导选举、分布式会话、集群状态）。分布式系统的协调导致了样板模式，使用 springcloud 的开发人员可以快速地建立实现这些模式的服务和应用程序。它们在任何分布式环境下都能很好地工作，包括开发人员自己的笔记本电脑、裸机数据中心和云计算等托管平台。

从 Spring 官网首页中打开 Cloud 页面，可以看到 Spring Cloud 最重要的系统架构。



下图是原来 Spring Cloud 官网上发布的架构图。



### 1.1.2 百度百科

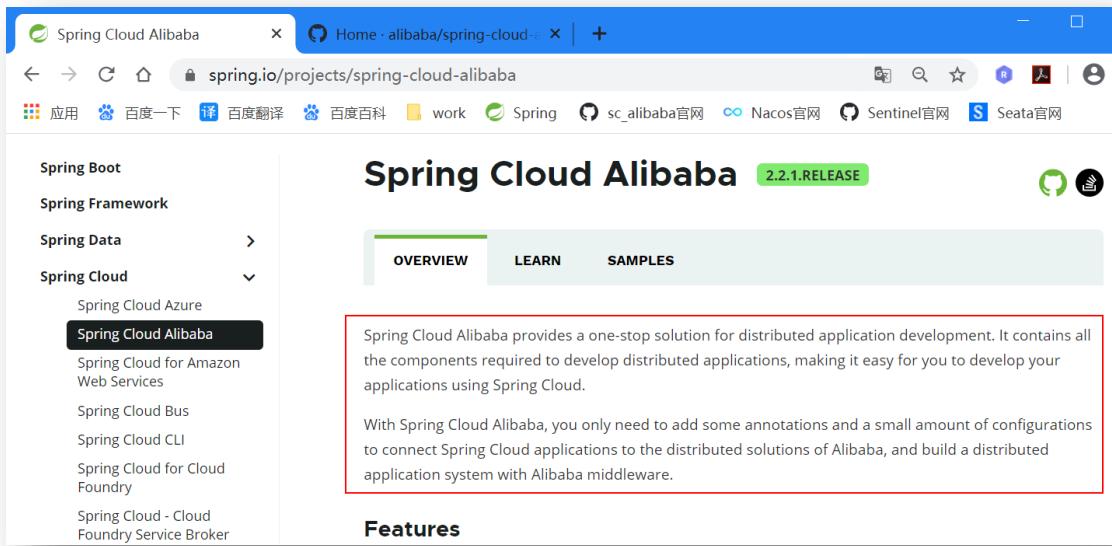
Spring Cloud 是一系列框架的有序集合。它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。Spring Cloud 并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过 Spring Boot 风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者提供了一套简单易懂、易部署和易维护的分布式系统开发工具包。

### 1.1.3 与 Spring Boot 的关系

Spring Boot 为 Spring Cloud 提供了代码实现环境，使用 Spring Boot 将其它组件有机融合到了 Spring Cloud 的体系架构中了。所以说，Spring Cloud 是基于 Spring Boot 的、微服务系统架构的一站式解决方案。

## 1.2 Spring Cloud Alibaba 简介

### 1.2.1 官网简介

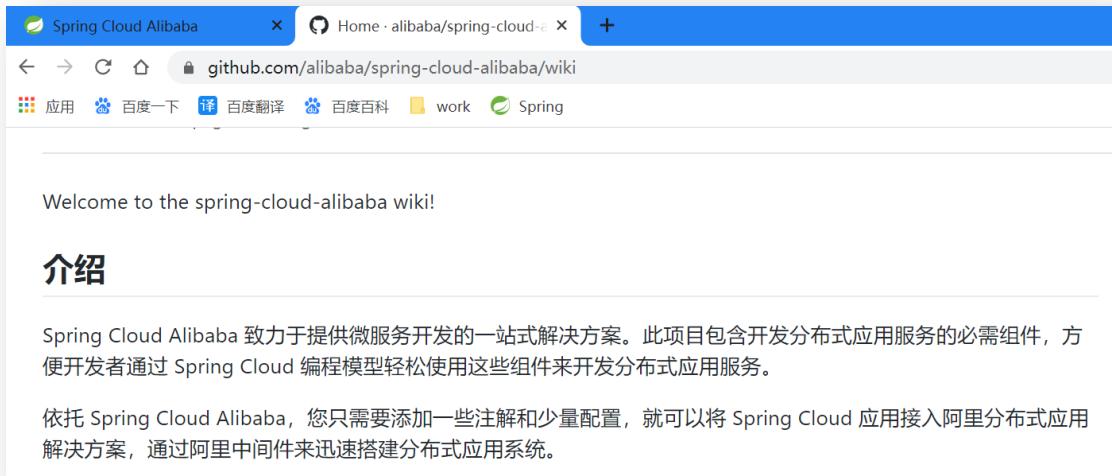


The screenshot shows the official website for Spring Cloud Alibaba at [spring.io/projects/spring-cloud-alibaba](http://spring.io/projects/spring-cloud-alibaba). The page title is "Spring Cloud Alibaba 2.2.1.RELEASE". On the left, there's a sidebar with links to other Spring projects like Spring Boot, Spring Framework, Spring Data, and Spring Cloud. The "Spring Cloud Alibaba" link is highlighted. The main content area has tabs for "OVERVIEW", "LEARN", and "SAMPLES". A red box highlights the following text from the "OVERVIEW" section:

Spring Cloud Alibaba provides a one-stop solution for distributed application development. It contains all the components required to develop distributed applications, making it easy for you to develop your applications using Spring Cloud.

With Spring Cloud Alibaba, you only need to add some annotations and a small amount of configurations to connect Spring Cloud applications to the distributed solutions of Alibaba, and build a distributed application system with Alibaba middleware.

github 中的中文简介就是上面这段文字的翻译。



The screenshot shows the GitHub wiki page for Spring Cloud Alibaba at [github.com/alibaba/spring-cloud-alibaba/wiki](https://github.com/alibaba/spring-cloud-alibaba/wiki). The page title is "Welcome to the spring-cloud-alibaba wiki!". Below it, there's a section titled "介绍" (Introduction) with the following text:

Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。

依托 Spring Cloud Alibaba，您只需要添加一些注解和少量配置，就可以将 Spring Cloud 应用接入阿里分布式应用解决方案，通过阿里中间件来迅速搭建分布式应用系统。

### 1.2.2 总结

Alibaba 的很多开源组件、中间件、框架，在国内外很多公司中早就被使用，并且很多已经是经过了“双 11”的考验与历练的。而 Spring Cloud 的使用便捷性、各组件的完美结合

性，深深吸引和打动了阿里，所以阿里团队研发了 Spring Cloud Alibaba。2018 年 10 月阿里将其捐赠给了 Spring Cloud 并开始孵化。2019 年 8 月正式孵化毕业成为 Spring Cloud 中的重要成员，与 Spring Cloud Netflix 并驾齐驱，并在国内国际市场迅速火爆。

## 1.3 版本兼容关系

由于 Spring Boot 3.0, Spring Boot 2.7~2.4 和 2.4 以下版本之间变化较大，目前企业级客户老项目相关 Spring Boot 版本仍停留在 Spring Boot 2.4 以下，为了同时满足存量用户和新用户不同需求，Spring Cloud Alibaba 社区以 Spring Boot 3.0 和 2.4 分别为分界线，同时维护 2022.x、2021.x、2.2.x 三个分支迭代。

### 1.3.1 Spring Boot 3.x 新变化

#### (1) JDK 与 Spring 的要求

Spring Boot3.x 版本要求 JDK 至少是 17, Spring 6.0。

#### (2) Jakarta 依赖

Spring Boot3.x 已经将所有底层依赖从 JavaEE 迁移到了 JarkartaEE，是基于 JakartaEE9 并尽可能地兼容 JakartaEE10。 java.\* → jakarta.\*

#### (3) IDEa 版本

IntelliJ IDEA 从 2022.2 版本开始完全支持 Spring Boot3.x 与 Spring 6。

### 1.3.2 JDK 免费/收费问题

从 2019 年开始，Oracle 宣布，某些版本开始收费。

- JDK8 之前版本，仍然免费
- JDK8 免费版本到 8u202，从 8u211 版本开始收费。
- JDK9、JDK10，全版本免费
- JDK11，免费版本到 11.0.2，从 11.0.3 版本开始商用收费
- JDK12、JDK13、JDK14、JDK15、JDK16，全版本商用收费
- JDK17、JDK18、JDK19、JDK20，全版本(二进制版本)免费

### 1.3.3 2022.x 分支

适配 Spring Boot 3.0, Spring Cloud 2022.x 版本及以上的 Spring Cloud Alibaba 版本按从新到旧排列如下表（最新版本用\*标记）：（注意，该分支 Spring Cloud Alibaba 版本命名方式进行了调整，未来将对应 Spring Cloud 版本，前三位为 Spring Cloud 版本，最后一位为扩展版本，比如适配 Spring Cloud 2022.0.0 版本对应的 Spring Cloud Alibaba 第一个版本为：2022.0.0.0，第二个版本为：2022.0.0.1，依此类推）。

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2022.0.0.0-RC*	Spring Cloud 2022.0.0	3.0.0

### 1.3.4 2021.x 分支

适配 Spring Boot 2.4, Spring Cloud 2021.x 版本及以上的 Spring Cloud Alibaba 版本按从新到旧排列如下表（最新版本用\*标记）：

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2021.0.4.0*	Spring Cloud 2021.0.4	2.6.11
2021.0.1.0	Spring Cloud 2021.0.1	2.6.3
2021.1	Spring Cloud 2020.0.1	2.4.2

### 1.3.5 2.2.x 分支

适配 Spring Boot 为 2.4, Spring Cloud Hoxton 版本及以下的 Spring Cloud Alibaba 版本按从新到旧排列如下表（最新版本用\*标记）：

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2.2.10-RC1*	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2.2.9.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2.2.8.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.RELEASE

<b>2.2.7.RELEASE</b>	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
<b>2.2.6.RELEASE</b>	Spring Cloud Hoxton.SR9	2.3.2.RELEASE
<b>2.2.1.RELEASE</b>	Spring Cloud Hoxton.SR3	2.2.5.RELEASE
<b>2.2.0.RELEASE</b>	Spring Cloud Hoxton.RELEASE	2.2.X.RELEASE
<b>2.1.4.RELEASE</b>	Spring Cloud Greenwich.SR6	2.1.13.RELEASE
<b>2.1.2.RELEASE</b>	Spring Cloud Greenwich	2.1.X.RELEASE
<b>2.0.4.RELEASE(停止维护, 建议升级)</b>	Spring Cloud Finchley	2.0.X.RELEASE
<b>1.5.1.RELEASE(停止维护, 建议升级)</b>	Spring Cloud Edgware	1.5.X.RELEASE

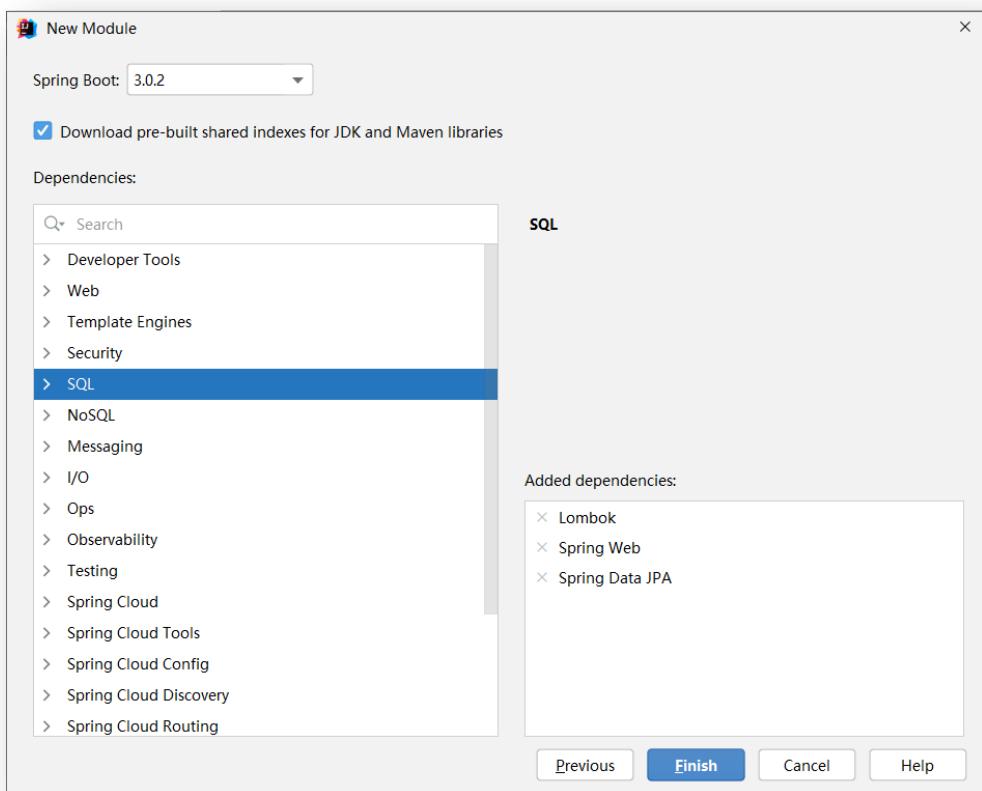
## 1.4 测试环境搭建

本例实现了消费者对提供者的调用，但并未使用到 Spring Cloud Alibaba，而是使用的 Spring 提供的 RestTemplate 实现，对于 MySQL 数据库的访问，使用 Spring Data JPA 作为持久层技术。不过后续 Spring Cloud Alibaba 的运行测试环境就是在此基础上修改出来的。

### 1.4.1 创建提供者工程 01-provider-8081

#### (1) 创建工程

创建一个 Spring Initializr 工程，并命名为 01-provider-8081。导入 Lombok、Web、JPA 依赖。



## (2) 导入 Druid 依赖

注意，默认添加的 MySQL 驱动版本号可能并不适合当前的 MySQL 版本，所以这里最好是指定 MySQL 驱动版本。

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.8</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
    <scope>runtime</scope>
</dependency>
```

### (3) 定义实体类

```
@Data  
@Entity  
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler", "fieldHandler"})  
public class Depart {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
    private String name;  
}
```

### (4) 定义 Repository 接口

```
public interface DepartRepository  
    extends JpaRepository<Depart, Integer> {  
}
```

### (5) 定义 Service 接口

```
public interface DepartService {  
    boolean saveDepart(Depart depart);  
    boolean removeDepartById(int id);  
    boolean modifyDepart(Depart depart);  
    Depart getDepartById(int id);  
    List<Depart> listAllDeparts();  
}
```

## (6) 定义 Service 实现类

## A、添加数据

```
@Service
public class DepartServiceImpl implements DepartService {
    @Autowired
    private DepartRepository repository;

    @Override
    public boolean saveDepart(Depart depart) {
        Depart obj = repository.save(depart);
        if (obj != null) {
            return true;
        }
        return false;
    }
}
```

## B、删除数据

```
@Override
public boolean removeDepartById(int id) {
    // 指定id的实体不存在, deleteById()方法会抛异常
    if(repository.existsById(id)) {
        repository.deleteById(id);
        return true;
    }
    return false;
}
```

## C、修改数据

与添加数据代码相同。

```
@Override
public boolean modifyDepart(Depart depart) {
    Depart obj = repository.save(depart);
    if (obj != null) {
        return true;
    }
    return false;
}
```

#### D、根据 id 查询

```
@Override
public Depart getDepartById(int id) {
    // 指定id的实体不存在, getReferenceById() 方法会抛异常
    if(repository.existsById(id)) {
        return repository.getReferenceById(id);
    }
    Depart depart = new Depart();
    depart.setName("no this depart");
    return depart;
}
```

#### E、查询所有

```
@Override
public List<Depart> listAllDeparts() {
    return repository.findAll();
}
```

## (7) 定义 Controller

```
    @RequestMapping("/provider/depart")
    @RestController
    public class DepartController {
        @Autowired
        private DepartService service;

        @PostMapping("/save")
        public boolean saveHandle(@RequestBody Depart depart) {
            return service.saveDepart(depart);
        }

        @DeleteMapping("/del/{id}")
        public boolean deleteHandle(@PathVariable("id") int id) {
            return service.removeDepartById(id);
        }
    }
```

```
    @PutMapping("/update")
    public boolean updateHandle(@RequestBody Depart depart) {
        return service.modifyDepart(depart);
    }

    @GetMapping("/get/{id}")
    public Depart getHandle(@PathVariable("id") int id) {
        return service.getDepartById(id);
    }

    @GetMapping("/list")
    public List<Depart> listHandle() {
        return service.listAllDeparts();
    }
}
```

## (8) 修改配置文件

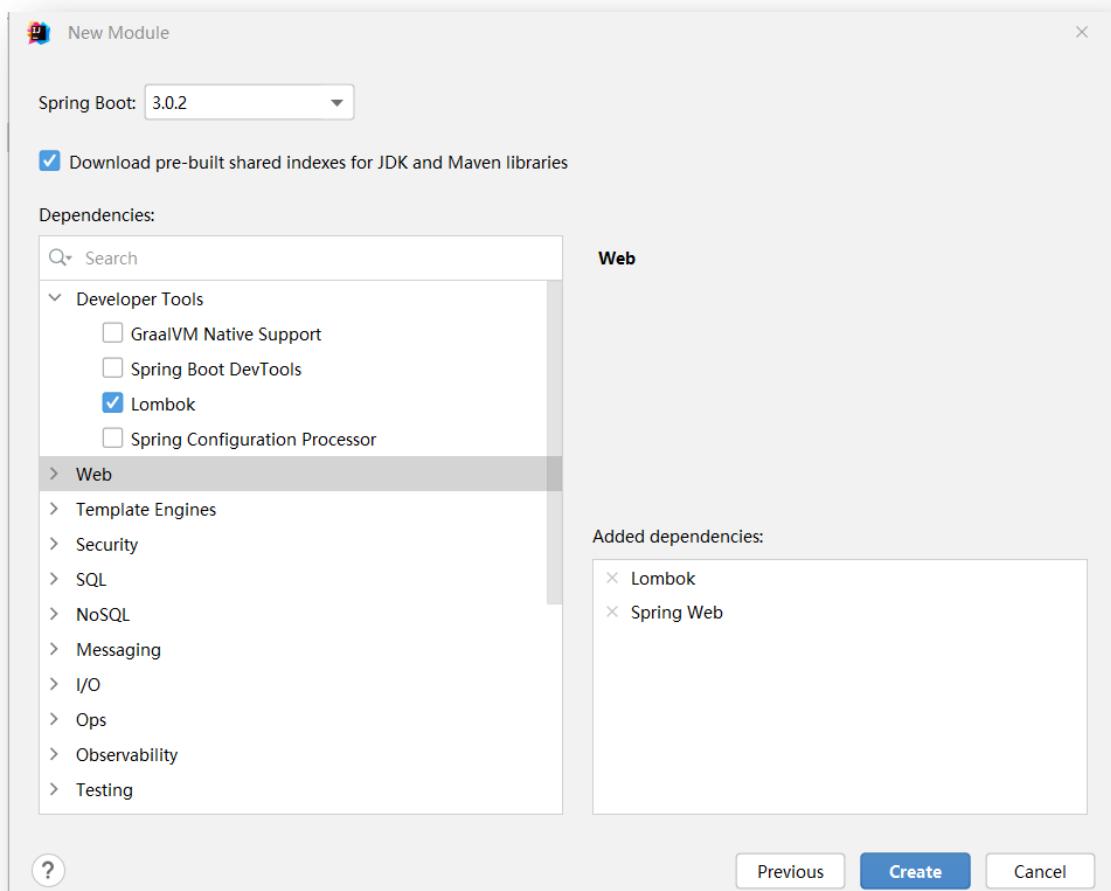
```
application.yml
1  server:
2    port: 8081
3
4  spring:
5    jpa:
6      generate-ddl: true
7      show-sql: true
8      hibernate:
9        ddl-auto: none
10
11  datasource:
12    type: com.alibaba.druid.pool.DruidDataSource
13    driver-class-name: com.mysql.jdbc.Driver
14    url: jdbc:mysql://test?useUnicode=true&useSSL=false&characterEncoding=utf8
15    username: root
16    password: 111
17
```

```
17
18  logging:
19    pattern:
20      console: level-%level %msg%
21    level:
22      root: info
23      org.hibernate: info
24      org.hibernate.type.descriptor.sql.BasicBinder: trace
25      org.hibernate.type.descriptor.sql.BasicExtractor: trace
26      com.abc: debug
27
```

### 1.4.2 创建消费者工程 01-consumer-8080

#### (1) 创建工程

创建一个 Spring Initializr 工程，并命名为 01-consumer-8080，导入 Lombok 与 Web 依赖。



## (2) 定义实体类

```
@Data  
public class Depart {  
    private Integer id;  
    private String name;  
}
```

### (3) 定义 JavaConfig 容器类

```
@Configuration
public class DepartCodeConfig {

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

### (4) 定义 Controller

```
no usages
@RequestMapping("/consumer/depart")
@RestController
public class DepartController {

    5 usages
    @Autowired
    private RestTemplate template;

    5 usages
    private static final String SERVICE_PROVIDER = "http://localhost:8081/provider/depart";
}
```

```
no usages
@PostMapping(PathVariable"/save")
public boolean saveHandle(@RequestBody Depart depart) {
    String url = SERVICE_PROCIER + "/save";
    return template.postForObject(url, depart, Boolean.class);
}

no usages
@DeleteMapping(PathVariable"/del/{id}")
public void deleteHandle(@PathVariable("id") int id) {
    template.delete(SERVICE_PROCIER + "/del/" + id);
}

no usages
@PutMapping(PathVariable"/update")
public void updateHandle(@RequestBody Depart depart) {
    String url = SERVICE_PROCIER + "/update";
    template.put(url, depart);
}
```

```
no usages
@GetMapping(PathVariable"/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = SERVICE_PROCIER + "/get/" + id;
    return template.getForObject(url, Depart.class);
}

no usages
@GetMapping(PathVariable"/list")
public List<Depart> listHandle() {
    String url = SERVICE_PROCIER + "/list";
    return template.getForObject(url, List.class);
}
```

## 第2章 Nacos 服务注册与发现

### 2.1 概述

#### 2.1.1 注册中心简介

所有提供者将自己提供服务的名称及自己主机详情（IP、端口、版本等）写入到另一台主机中的一个列表中，这台主机称为**服务注册中心**，而这个表称为**服务注册表**。

所有消费者需要调用微服务时，其会从注册中心首先将服务注册表下载到本地，然后根据消费者本地设置好的负载均衡策略选择一个服务提供者进行调用。这个过程称为**服务发现**。

可以充当 Spring Cloud 服务注册中心的服务器很多，如 Zookeeper、Eureka、Consul 等。Spring Cloud Alibaba 中使用的注册中心为 Alibaba 的中间件 Nacos。

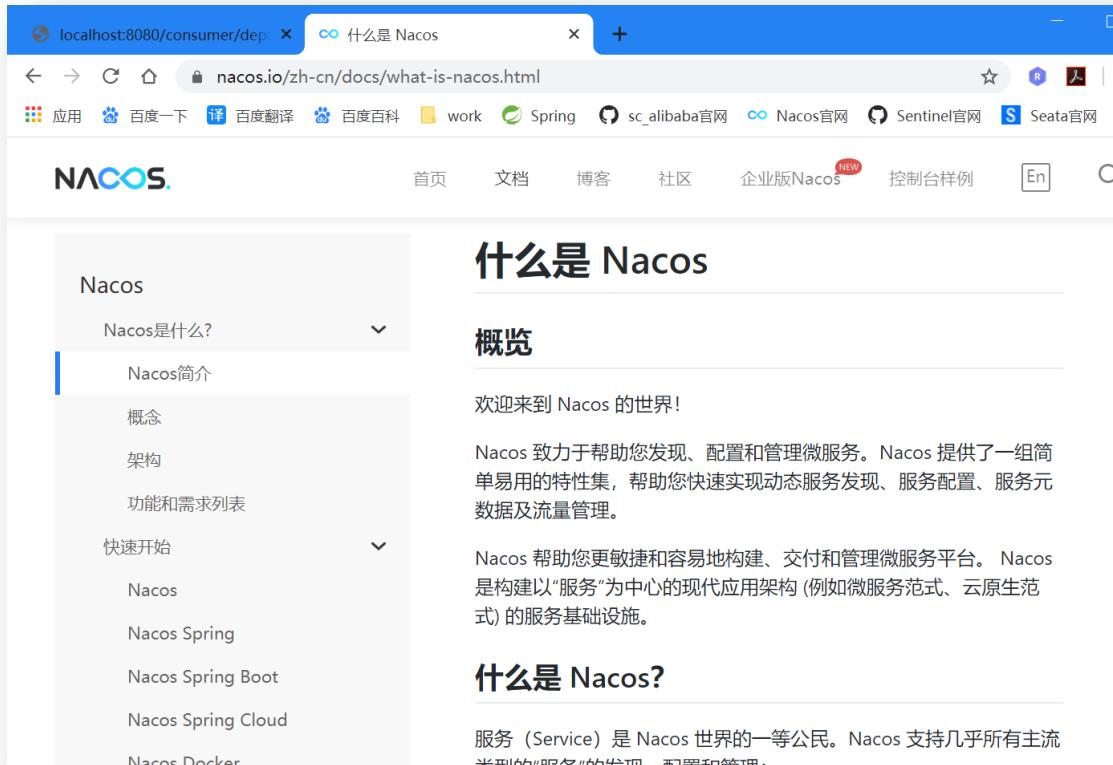
#### 2.1.2 Nacos 简介

Nacos 简介在其官网 <https://nacos.io/> 中描述的很详细。

云原生应用简单来说就是 SaaS，就是跑在 IaaS、PaaS 上的 SaaS。

云原生 = 微服务 + DevOps + CD + 容器化



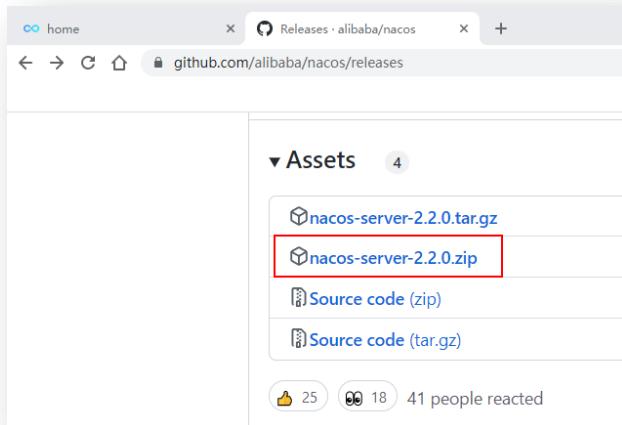


The screenshot shows the Nacos official website at [nacos.io/zh-cn/docs/what-is-nacos.html](https://nacos.io/zh-cn/docs/what-is-nacos.html). The page title is "什么是 Nacos". On the left, there is a sidebar with sections like "Nacos是什么?", "Nacos简介", "概念", "架构", "功能和需求列表", "快速开始", "Nacos", "Nacos Spring", "Nacos Spring Boot", "Nacos Spring Cloud", and "Nacos Docker". The main content area has a section titled "概览" (Overview) with text about Nacos's mission to help discover, configure, and manage microservices. It also features a section titled "什么是 Nacos?" (What is Nacos?) with a note that services are first-class citizens.

## 2.2 Nacos 下载与启动

### 2.2.1 下载

从官网页面可以看出，有两种资源下载方式：源码下载与打过包的工程下载。点击“最新稳定版本”，可以选择性地下载最新版的这两种资源。这里选择编译过的 zip 压缩资源。

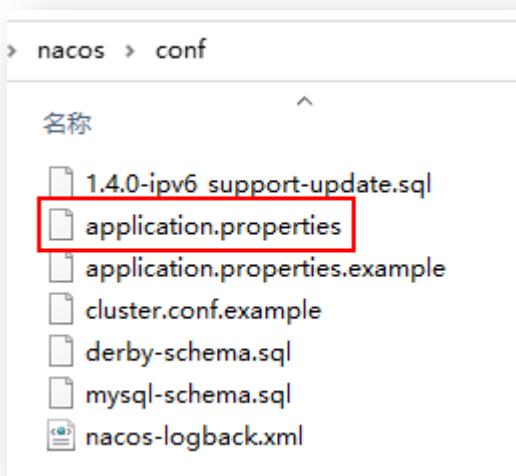


The screenshot shows the GitHub releases page for the Nacos project at [github.com/alibaba/nacos/releases](https://github.com/alibaba/nacos/releases). It lists four assets: "nacos-server-2.2.0.tar.gz", "nacos-server-2.2.0.zip" (which is highlighted with a red box), "Source code (zip)", and "Source code (tar.gz)". Below the assets, there are engagement metrics: 25 likes, 18 comments, and 41 people reacted.

## 2.2.2 安装与配置

### (1) 配置端口号

解压压缩包。在压缩包的 conf 目录中，找到 application.properties 文件。



从配置文件可以看出，默认 Nacos 服务器的端口号为 8848，上下文路径为/nacos。一般都是采用默认值，但也可以修改。

```
16
17 **** Spring Boot Related Configurations
18 ### Default web context path:
19 server.servlet.contextPath=/nacos
20 ### Include message field
21 server.error.include-message=ALWAYS
22 ### Default web server port:
23 server.port=8848
24
```

### (2) 配置鉴权

从 nacos2.2.0.1 版本开始，nacos 配置文件中去掉了默认的鉴权配置，需要用户手工添加，否则无法启动 nacos。

```
136
137     ### If turn on auth system:
138     nacos.core.auth.enabled=true
139
140     ### Turn on/off caching of auth information. By turning on this switch, the update of auth information w
141     nacos.core.auth.caching.enabled=true
142
143     ### Since 1.4.1, Turn on/off white auth for user-agent: nacos-server, only for upgrade from old version.
144     nacos.core.auth.enable.userAgentAuthWhite=false
145
146     ### Since 1.4.1, worked when nacos.core.auth.enabled=true and nacos.core.auth.enable.userAgentAuthWhite=
147     ### The two properties is the white list for auth and used by identity the request from other server.
148     nacos.core.auth.server.identity.key=dfsgdas
149     nacos.core.auth.server.identity.value=dgdgs
150
151     ### worked when nacos.core.auth.system.type=nacos
152     ### The token expiration in seconds:
153     nacos.core.auth.plugin.nacos.token.cache.enable=false
154     nacos.core.auth.plugin.nacos.token.expire.seconds=18000
155     ### The default token (Base64 String):
156     nacos.core.auth.plugin.nacos.token.secret.key=sdfsfsfsfsfsgghdfhfkhjfm58j5i56dfaasghawhegdgd2wtdgds==
157
158     ### worked when nacos.core.auth.system.type=ldap, {0} is Placeholder, replace login username
159     #nacos.core.auth.ldap.url=ldap://localhost:389
160     #nacos.core.auth.ldap.baseDc=dc=example,dc=org
```

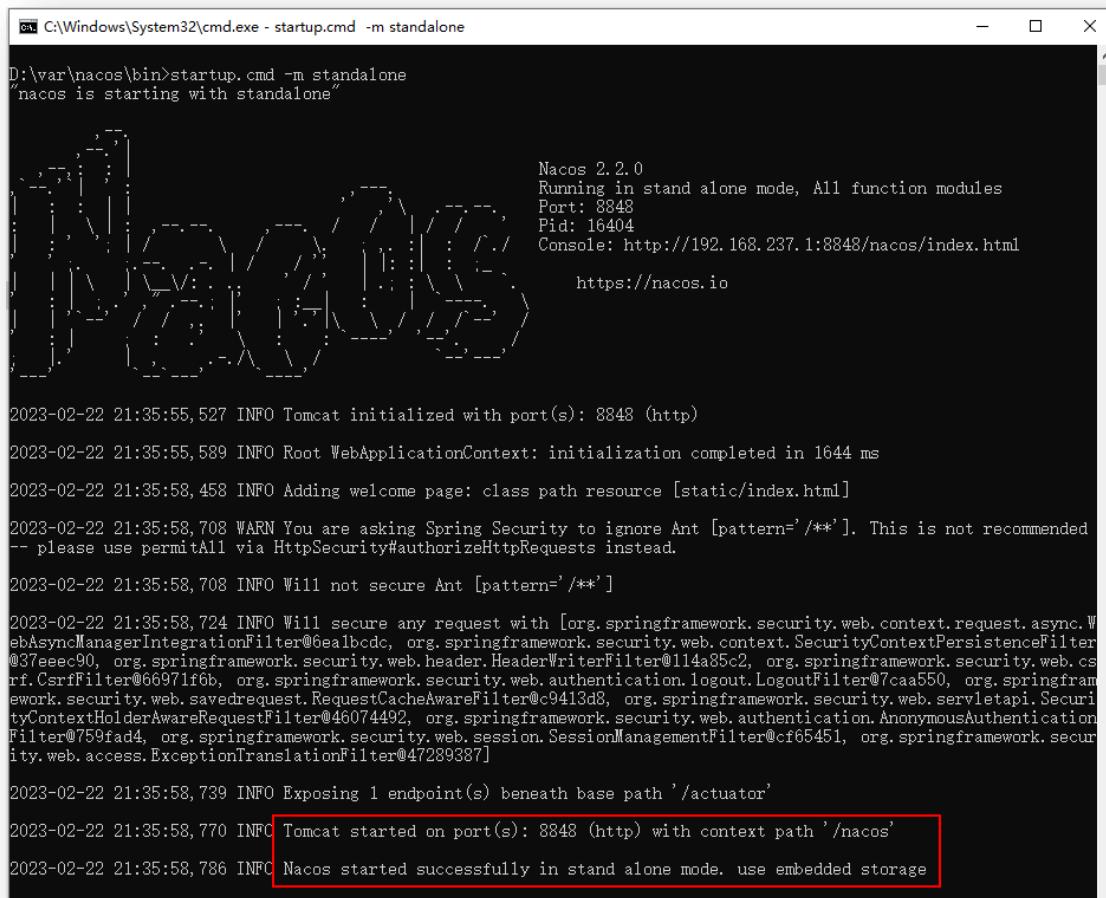
### (3) 启动

在 nacos/bin 目录中有启动命令文件。其中 cmd 是 Windows 系统中的命令，sh 是 Linux 系统中的命令。

由于其默认是集群方式启动，所以若要单机启动，则需要在 cmd 中通过命令启动。



```
C:\Windows\System32\cmd.exe
D:\var\nacos\bin>startup.cmd -m standalone
```



```
C:\Windows\System32\cmd.exe - startup.cmd -m standalone
D:\var\nacos\bin>startup.cmd -m standalone
"nacos is starting with standalone"

Nacos 2.2.0
Running in stand alone mode, All function modules
Port: 8848
Pid: 16404
Console: http://192.168.237.1:8848/nacos/index.html
https://nacos.io

2023-02-22 21:35:55,527 INFO Tomcat initialized with port(s): 8848 (http)
2023-02-22 21:35:55,589 INFO Root WebApplicationContext: initialization completed in 1644 ms
2023-02-22 21:35:58,458 INFO Adding welcome page: class path resource [static/index.html]
2023-02-22 21:35:58,708 WARN You are asking Spring Security to ignore Ant [pattern='/**']. This is not recommended
-- please use permitAll via HttpSecurity#authorizeHttpRequests instead.
2023-02-22 21:35:58,708 INFO Will not secure Ant [pattern='/**']

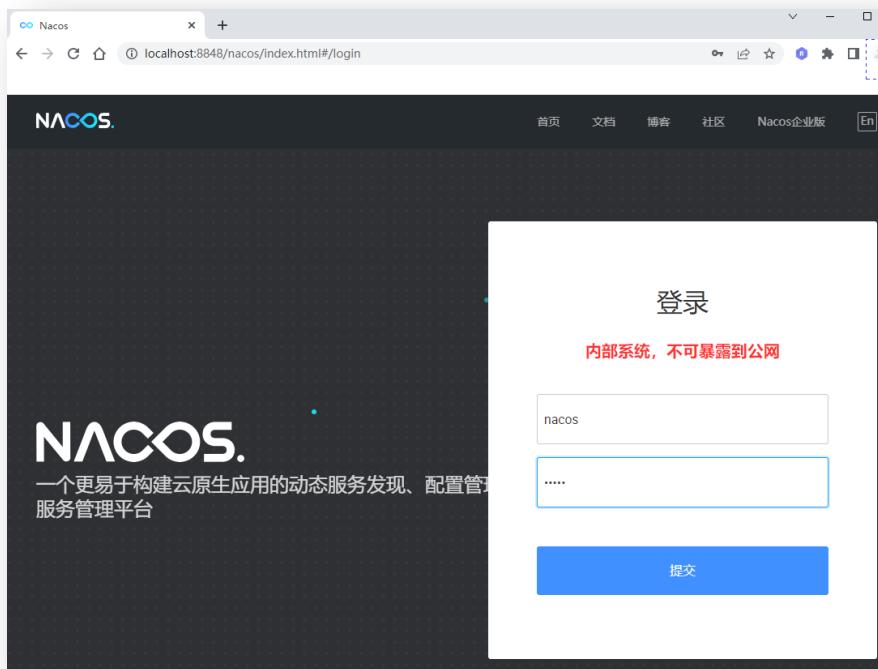
2023-02-22 21:35:58,724 INFO Will secure any request with [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@6ealbcde, org.springframework.security.web.context.SecurityContextPersistenceFilter@37eeec90, org.springframework.security.web.header.HeaderWriterFilter@114a85c2, org.springframework.security.web.csrf.CsrfFilter@66971f6b, org.springframework.security.web.authentication.logout.LogoutFilter@7caa550, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@c9413d8, org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@46074492, org.springframework.security.web.authentication.AnonymousAuthenticationFilter@759fad4, org.springframework.security.web.session.SessionManagementFilter@cf65451, org.springframework.security.web.access.ExceptionTranslationFilter@47289387]

2023-02-22 21:35:58,739 INFO Exposing 1 endpoint(s) beneath base path '/actuator'
2023-02-22 21:35:58,770 INFO Tomcat started on port(s): 8848 (http) with context path '/nacos'
2023-02-22 21:35:58,786 INFO Nacos started successfully in stand alone mode. use embedded storage
```

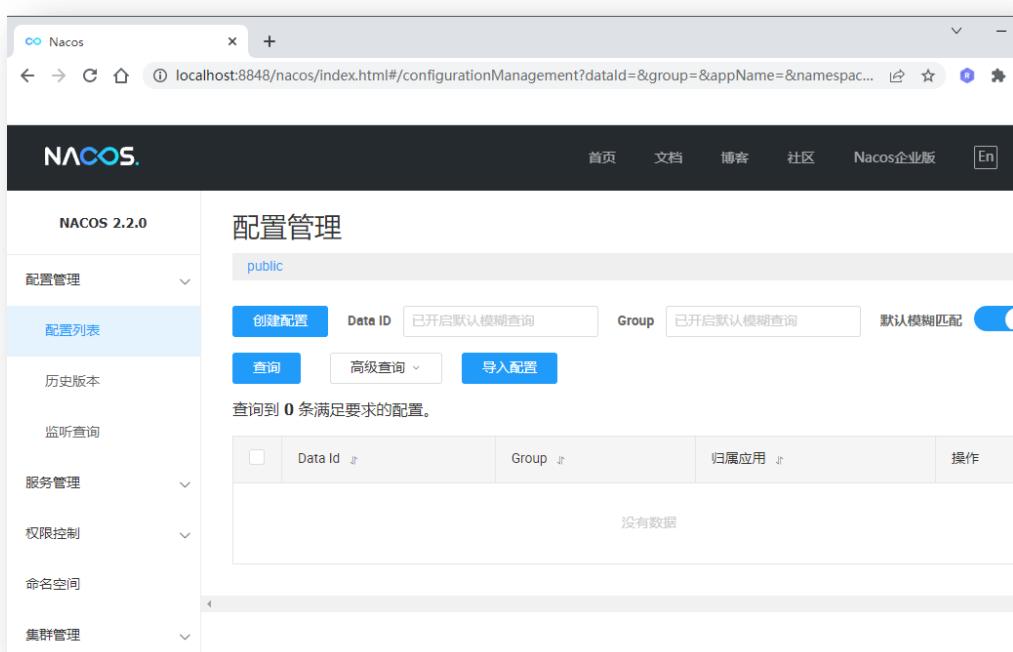
## 2.2.3 访问控制台

### (1) 登录

在浏览器地址栏通过 <http://localhost:8848/nacos/index.html> 可打开 Nacos 控制台的登录页面。



默认账号/密码为 nacos/nacos。不过，该默认账号与密码存放在 nacos 内置 mysql 数据库中的，而非存放在某配置文件中。若要添加账号或修改密码，可在登录后通过页面修改。  
输入账号/密码后即可看到如下界面，说明 nacos server 已经启动成功了。

A screenshot of the Nacos configuration management interface. The URL is localhost:8848/nacos/index.html#/configurationManagement?dataId=&group=&appName=&namespace=. The left sidebar shows "NACOS 2.2.0" and "配置管理" (Configuration Management) selected. The main area is titled "配置管理" (Configuration Management) and shows a search bar with "public" selected. Below the search bar are buttons for "创建配置" (Create Configuration), "Data ID" (Data ID), "已开启默认模糊查询" (Default fuzzy search enabled), "Group" (Group), "已开启默认模糊查询" (Default fuzzy search enabled), and "默认模糊匹配" (Default fuzzy matching). There are also "查询" (Query), "高级查询" (Advanced Query), and "导入配置" (Import Configuration) buttons. A message below the search bar says "查询到 0 条满足要求的配置." (0 configurations found). A table below shows columns for Data Id, Group, Application, and Operation, with a note "没有数据" (No data).

## (2) 修改登录信息

在该页面中可以添加、删除普通用户，修改所有用户的密码。但不能删除管理员用户，也不能将普通用户指定为管理员角色。



The screenshot shows the Nacos 2.2.0 user management interface. On the left, there's a sidebar with options like Configuration Management, Service Management, Permission Control, and the currently selected User List. The main area has a title "User Management" and a search bar with filters for "用户名" (hello/nacos), "密码" (\*\*\*\*\*), and a toggle for "默认模糊匹配". Below the search bar is a table with two rows. Each row contains a user name ("hello" and "nacos"), their password (both shown as \*\*\*\*), and two buttons: "修改" (Modify) and "删除" (Delete). The "nacos" row is highlighted with a blue background.

## 2.3 定义提供者 02-provider-nacos-8081

### 2.3.1 定义工程

复制 01-provider-8081 工程，并重命名为 02-provider-nacos-8081。

### 2.3.2 添加 spring cloud 依赖管理

当前的工程首先是个 Spring Cloud 工程，所以需要在 pom 文件中添加 Spring Cloud 依赖管理模块。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2022.0.0</version>
            <type>pom</type>
            <scope>import</scope>
        
```

```
</dependency>
</dependencies>
</dependencyManagement>
```

### 2.3.3 添加 alibaba 依赖管理

还需要添加上 spring cloud alibaba 的依赖。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2022.0.0-RC1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

### 2.3.4 添加 nacos-discovery 依赖

在<dependencies>标签下添加 Nacos Discovery 依赖。

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

### 2.3.5 pom 内容

修改后的 pom 文件依赖如下：

```
<properties>
  <java.version>19</java.version>
</properties>
```

```
<dependencies>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.2.8</version>
    </dependency>
    <!--修改 MySQL 驱动版本-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

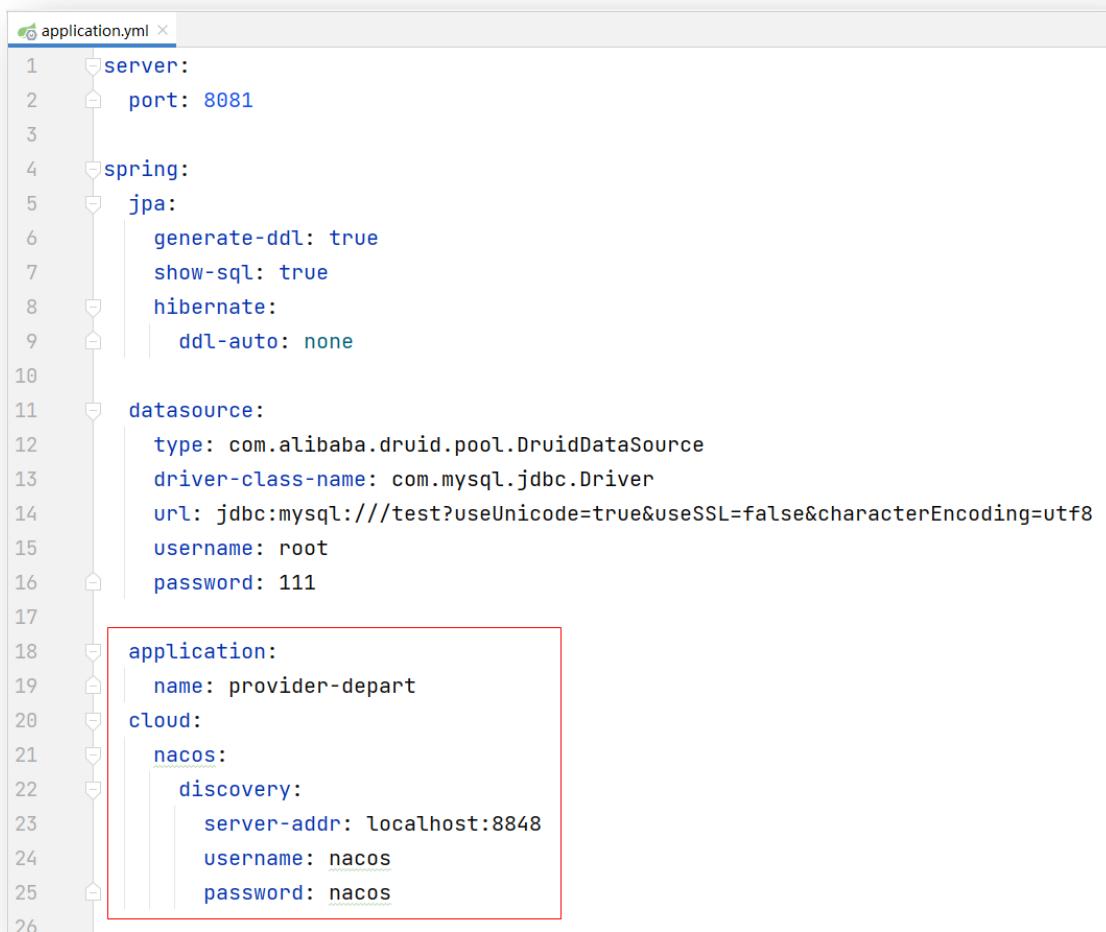
<dependencyManagement>
    <dependencies>
        <dependency>
```

```
<groupId>com.alibaba.cloud</groupId>
<artifactId>spring-cloud-alibaba-dependencies</artifactId>
<version>2022.0.0-RC1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>2022.0.0</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<excludes>
<exclude>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
</exclude>
</excludes>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

### 2.3.6 修改配置文件

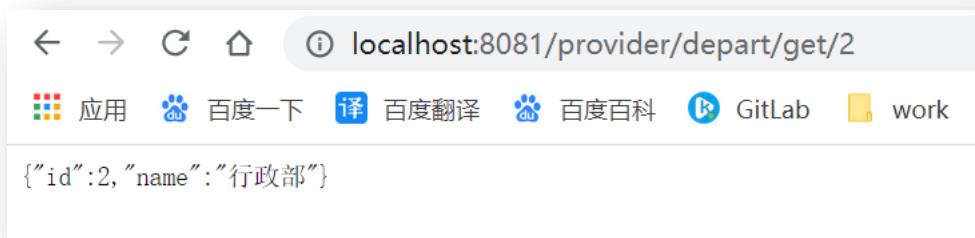
添加如下配置，指定 nacos server 的地址及微服务名称。



```
application.yml
1 server:
2   port: 8081
3
4 spring:
5   jpa:
6     generate-ddl: true
7     show-sql: true
8     hibernate:
9       ddl-auto: none
10
11 datasource:
12   type: com.alibaba.druid.pool.DruidDataSource
13   driver-class-name: com.mysql.jdbc.Driver
14   url: jdbc:mysql:///test?useUnicode=true&useSSL=false&characterEncoding=utf8
15   username: root
16   password: 111
17
18 application:
19   name: provider-depart
20
21 cloud:
22   nacos:
23     discovery:
24       server-addr: localhost:8848
25       username: nacos
26       password: nacos
```

### 2.3.7 查看 nacos 控制台

启动该工程后，在浏览器中直接访问该提供者是没有问题的，说明该提供者已经启动。



在 nacos server 已经启动的情况下，查看 nacos 控制台，就可以看到该提供者。说明该微服务已经被 nacos 发现。

## 2.4 定义消费者 02-consumer-nacos-8080

### 2.4.1 定义工程

复制 01-consumer-8080 工程，并重命名为 02-consumer-nacos-8080。这个消费者是通过 RestTemplate 进行消费的。

### 2.4.2 修改 pom

与定义 provider 时的相同，consumer 的 pom 也做了如下几处的修改：

- 添加 spring cloud 依赖管理
- 添加 spring cloud alibaba 依赖管理
- 添加 nacos-discovery 依赖
- **添加 spring-cloud-starter-loadbalance 依赖**

修改后的 POM 文件中的依赖情况如下：

```
<properties>
    <java.version>19</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

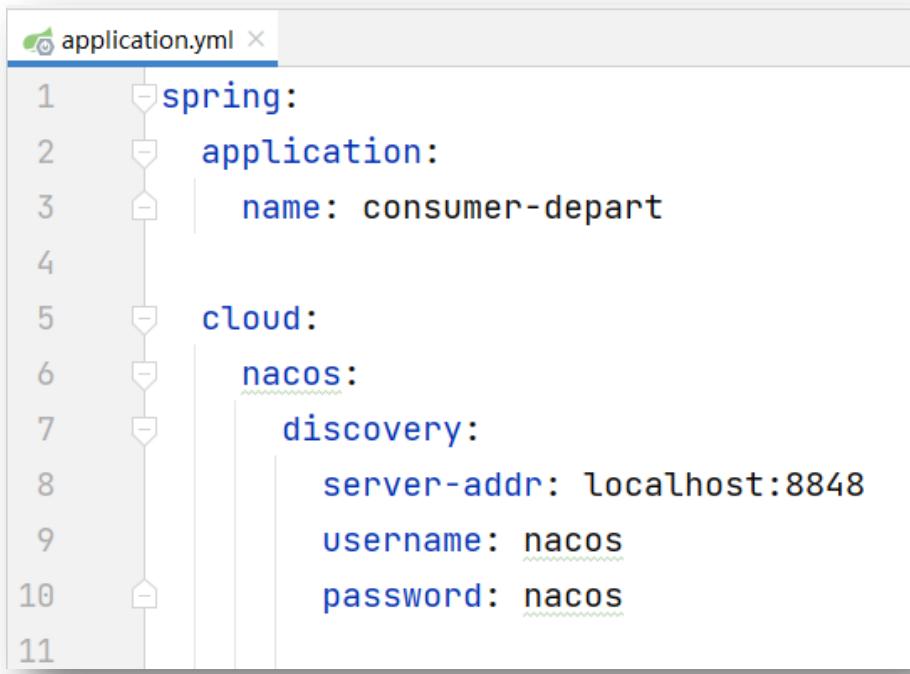
```
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>com.alibaba.cloud</groupId>
<artifactId>spring-cloud-alibaba-dependencies</artifactId>
<version>2022.0.0-RC1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>2022.0.0</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<excludes>
<exclude>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
</exclude>
</excludes>
</configuration>
</plugin>
</plugins>
</build>

</project>
```

### 2.4.3 修改配置文件



```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4
5   cloud:
6     nacos:
7       discovery:
8         server-addr: localhost:8848
9         username: nacos
10        password: nacos
11
```

### 2.4.4 修改控制器类

原来 consumer 中采用的是直连方式访问 provider，现在要根据微服务名称来访问。

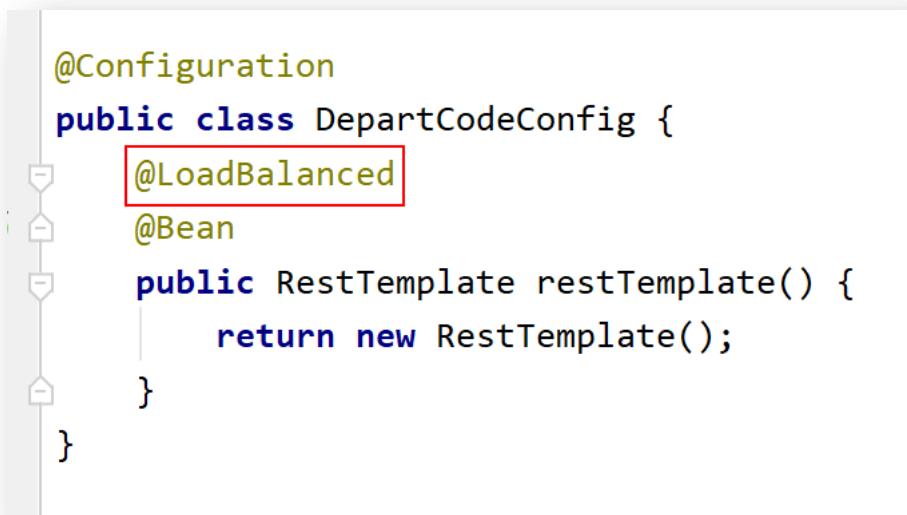


```
DepartController
@RequestMapping("/consumer/depart")
public class DepartController {
    private RestTemplate restTemplate;

    private static final String SERVICE_PROVIDER = "http://localhost:8081";
    private static final String SERVICE_PROVIDER = "http://provider-depart";
```

## 2.4.5 修改 JavaConfig 类

当使用微服务名称来访问 provider 时，其是通过负载均衡方式进行访问的。所以，需要添加如下注解。这里的负载均衡采用的是 spring cloud 自己开发的 spring cloud loadbalancer。



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

## 2.4.6 查看 nacos 控制台



服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值
consumer-depart	DEFAULT_GROUP	1	1	1	false
provider-depart	DEFAULT_GROUP	1	1	1	false

当启动 consumer 后，直接在浏览器访问 consumer，可以看到正确的结果，说明 consumer 已经发现并调用到了 provider。然后查看 nacos 控制台，也可以看到这两个服务。



## 2.5 获取服务列表

除了可以在 Nacos 控制台直观的查看到 Nacos 注册中心中注册的微服务信息外，也可以在代码中通过 `DiscoveryClient API` 获取服务列表。

### 2.5.1 修改 controller

在任何微服务的提供者或消费者处理器中，只要获取到“服务发现 Client”，即可读取到注册中心的微服务列表。本例直接修改 `02-provider-nacos-8081` 中的控制器类。



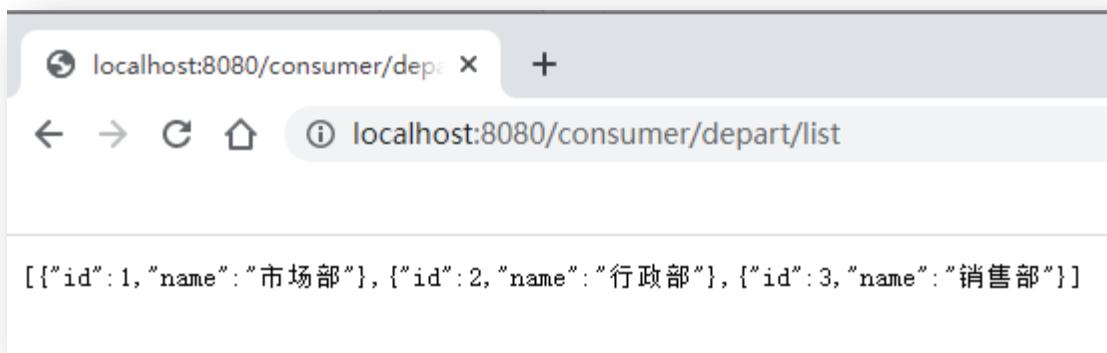
```
@RequestMapping("/provider/depart")
@RestController
public class DepartController {
    5 usages
    @Autowired
    private DepartService service;

    2 usages
    @Autowired
    private DiscoveryClient client;
```

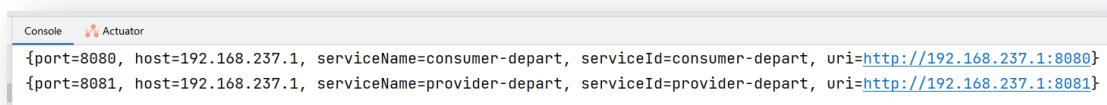
```
@GetMapping("/discovery")
public List<String> discoveryHandle() {
    List<String> services = client.getServices();
    for (String serviceName : services) {
        List<ServiceInstance> instances = client.getInstances(serviceName);
        for (ServiceInstance instance : instances) {
            Map<String, Object> map = new HashMap<>();
            map.put("serviceName", serviceName);
            map.put("serviceId", instance.getServiceId());
            map.put("host", instance.getHost());
            map.put("port", instance.getPort());
            map.put("uri", instance.getUri());
            System.out.println(map);
        }
    }
    return services;
}
```

## 2.5.2 运行效果

启动 02-provider-nacos-8081 与 02-consumer-nacos-8080 两个工程。



在 provider 的控制台，可以看到如下的服务信息。



### 2.5.3 注册表缓存

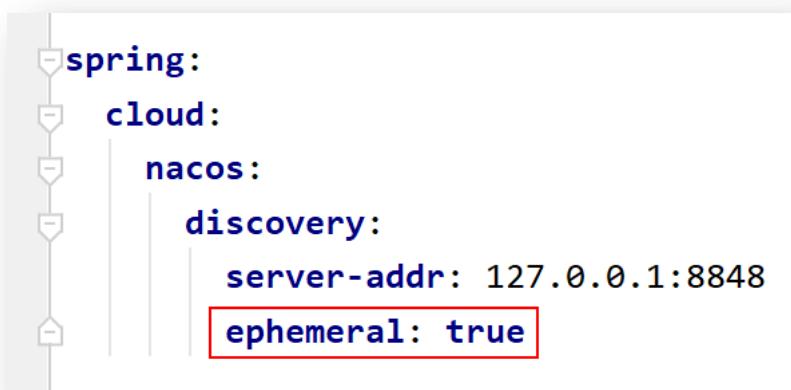
服务在启动后，当发生调用时会自动从 Nacos 注册中心下载并缓存注册表到本地。所以，即使 Nacos 发生宕机，会发现消费者仍然是可以调用到提供者的。只不过此时已经不能再有服务进行注册了，服务中缓存的注册列表信息无法更新。

## 2.6 临时实例与持久实例

Nacos 中的实例分为临时实例与持久实例。

### 2.6.1 配置

在服务注册时有一个属性 `ephemeral` 用于描述当前实例在注册时是否以临时实例出现。为 `true` 则为临时实例，默认值；为 `false` 则为持久实例。



### 2.6.2 区别

- 临时实例与持久实例的实例存储的位置与健康检测机制是不同的。
- **临时实例：**默认情况。服务实例仅会注册在 Nacos 内存，不会持久化到 Nacos 磁盘。其健康检测机制为 Client 模式，即 Client 主动向 Server 上报其健康状态。默认心跳间隔为 5 秒。在 15 秒内 Server 未收到 Client 心跳，则会将其标记为“不健康”状态；在 30 秒内若收到了 Client 心跳，则重新恢复“健康”状态，否则该实例将从 Server 端内存清除。
- **持久实例：**服务实例不仅会注册到 Nacos 内存，同时也会被持久化到 Nacos 磁盘。其健康检测机制为 Server 模式，即 Server 会主动去检测 Client 的健康状态，默认每 20 秒检测一次。健康检测失败后服务实例会被标记为“不健康”状态，但不会被清除，因为其是持久化在磁盘的。

## 2.7 将数据持久化到外置 MySQL

默认情况下，Nacos 使用的是内置 storage，使用内置 storage 存在两个很大的问题：

- 数据是存放在内存中的，无法持久化
- 无法搭建集群。当 Nacos 作为配置中心时，要求必须是外置的 DBMS

### 2.7.1 DBMS 说明

Nacos 对于 DBMS 的要求：

- 安装数据库，版本要求：5.6.5+
- 初始化 mysql 数据库，数据库初始化文件：mysql-schema.sql
- 修改 conf/application.properties 文件，增加支持 mysql 数据源配置（目前只支持 mysql），添加 mysql 数据源的 url、用户名和密码。

### 2.7.2 修改 SQL 脚本文件

若要连接外置 MySQL，则外置 MySQL 中就要有相应的数据库及表。这些表的创建脚本文件在 Nacos 解压目录的 config 子目录中的 mysql-schema.sql。

打开 mysql-schema.sql 文件，发现其中只能表的创建语句，并没有数据库的创建语句，且文件中给出数据库的名称建议使用 nacos\_config。为了方便后面对这个脚本文件的执行，在该文件中添加如下的 DB 创建语句。

```
10 * Unless required by applicable law or agreed to in writing
11 * distributed under the License is distributed on an "AS IS"
12 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
13 * or implied. See the License for the specific language governing per
14 * limitations under the License.
15 */
16 ****
17 /* 数据库全名 = nacos_config */
18 ****
19 CREATE DATABASE IF NOT EXISTS `nacos_config`;
20
21 USE `nacos_config`;
22
23 ****
24 /* 数据库全名 = nacos_config */
25 /* 表名称 = config_info */
26 /* 表名称 = config_info */
27 ****
28 CREATE TABLE `config_info` (
29   `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'id',
30   `data_id` varchar(255) NOT NULL COMMENT 'data_id',
31   `group_id` varchar(255) DEFAULT NULL
```

### 2.7.3 运行脚本文件

这里对于该脚本文件的运行，放在 Idea 中进行。在 Idea 的数据库连接上右击，选择 SQL Scripts/Run SQL Scripts，在弹出的窗口中找到这个 sql 文件运行即可。

### 2.7.4 修改 Nacos 配置

打开 Nacos 安装目录下的 conf/application.properties 文件，把用于注释的#去掉。

	1	2	3	4	5	6
--	---	---	---	---	---	---

```
30
31 ##### Config Module Related Configurations *****
32 ### If use MySQL as datasource:
33 #spring.datasource.platform=mysql
34
35 ### Count of DB:
36 #db.num=1
37
38 ### Connect URL of DB:
39 #db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?characterEnco
40 #db.user=nacos
41 #db.password=nacos
42
```

变为以下内容：

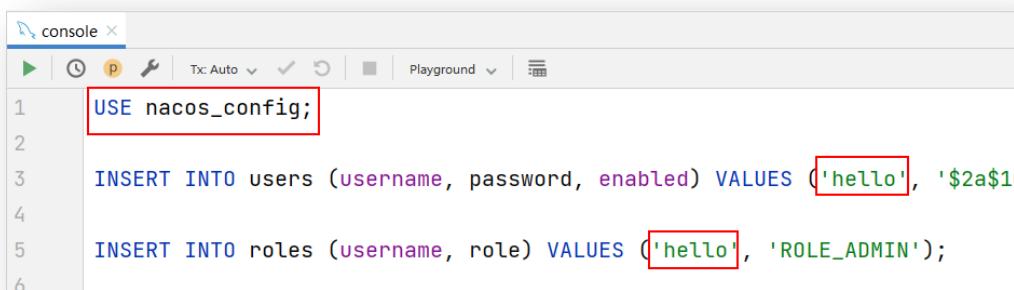
```
32
33 ##### Config Module Related Configurations *****
34 ### If use MySQL as datasource:
35 spring.datasource.platform=mysql
36
37 ### Count of DB:
38 db.num=1
39
40 ### Connect URL of DB:
41 db.url.0=jdbc:mysql://127.0.0.1:3306/nacos_config?characterEn
42 db.user.0=root
43 db.password.0=111
44
45 ### Connection pool configuration: hikariCP
46 #hikari.dataSource.idleTimeout = 60000
```

## 2.7.5 修改 Nacos 平台登录账号

为了验证已经换为了外置 MySQL，现在 MySQL 中添加一个新的 nacos 用户。

### (1) 添加新用户

插入一个新的用户，用户名为 hello，密码仍为 nacos。当然，若要使用新密码，可以通过 MD5 工具加密后，将加密后的字符串放到这里。



```
console >
1 USE nacos_config;
2
3 INSERT INTO users (username, password, enabled) VALUES ('hello', '$2a$10$');
4
5 INSERT INTO roles (username, role) VALUES ('hello', 'ROLE_ADMIN');
```

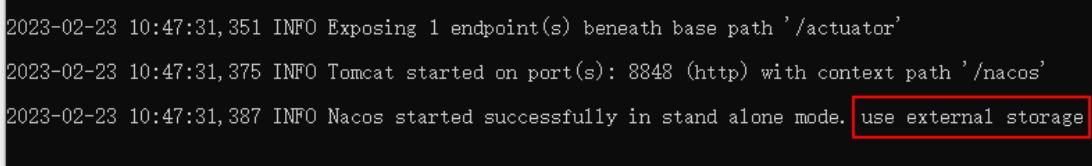
### (2) 重启 Nacos 后再登录

SQL 执行成功后，将原来启动的 Nacos 服务器关闭后重新启动。



```
C:\Windows\System32\cmd.exe
D:\var\nacos\bin>startup.cmd -m standalone
```

在启动日志中可以看到，已经使用外部存储设备了。



```
2023-02-23 10:47:31,351 INFO Exposing 1 endpoint(s) beneath base path '/actuator'
2023-02-23 10:47:31,375 INFO Tomcat started on port(s): 8848 (http) with context path '/nacos'
2023-02-23 10:47:31,387 INFO Nacos started successfully in stand alone mode. use external storage
```

然后再次打开 Nacos 浏览器平台，登出原来的账号后，使用 hello 账号 nacos 密码再次登录，可以正常登录到平台，说明现在已经由内置 MySQL 成功转换到了外置 MySQL 了。

## 2.8 Nacos 集群搭建

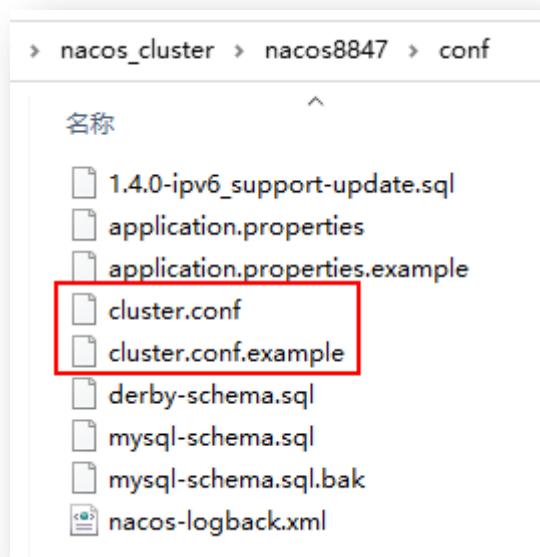
单机版 Nacos 都存在单点问题，所以需要搭建高可用的 Nacos 集群。

### 2.8.1 Nacos 集群搭建

#### (1) 修改配置

这里要搭建的 Nacos 集群中包含三台 Nacos 服务器，由于这些 Nacos 都在同一台主机，所以这里创建的集群实际只有端口号不同，是个伪集群。

首先随意创建一个目录，用于存放三个 Nacos 服务器。例如在 D 盘创建一个 nacos\_cluster 目录。然后再复制原来配置好的单机版的 Nacos 到这个目录，并重命名为 nacos8847。将来要这里要存放三个子目录，分别为 nacos8847、nacos8849、nacos8851。



打开 nacos8847/conf，重命名其中的 cluster.conf.example 为 cluster.conf。然后打开该文件，在其中写入三个 nacos 的 ip:port。注意，不能写为 localhost 与 127.0.0.1，且这三个端口号不能连续。否则会报地址被占用异常。

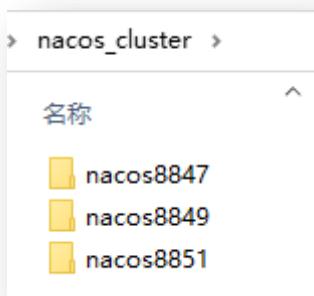
	1	2
1	192.168.3.153:8847	
2	192.168.3.153:8849	
3	192.168.3.153:8851	
4		

然后再打开 nacos8847/conf/application.properties 文件，修改端口号为 8847。

```
10
17 #***** Spring Boot Related Configurations *
18 ### Default web context path:
19 server.servlet.contextPath=/nacos
20 ### Include message field
21 server.error.include-message=ALWAYS
22 ### Default web server port:
23 server.port=8847
24
```

## (2) 复制目录

将 nacos8847 目录复制三份，分别命名为 nacos8849、nacos8851。



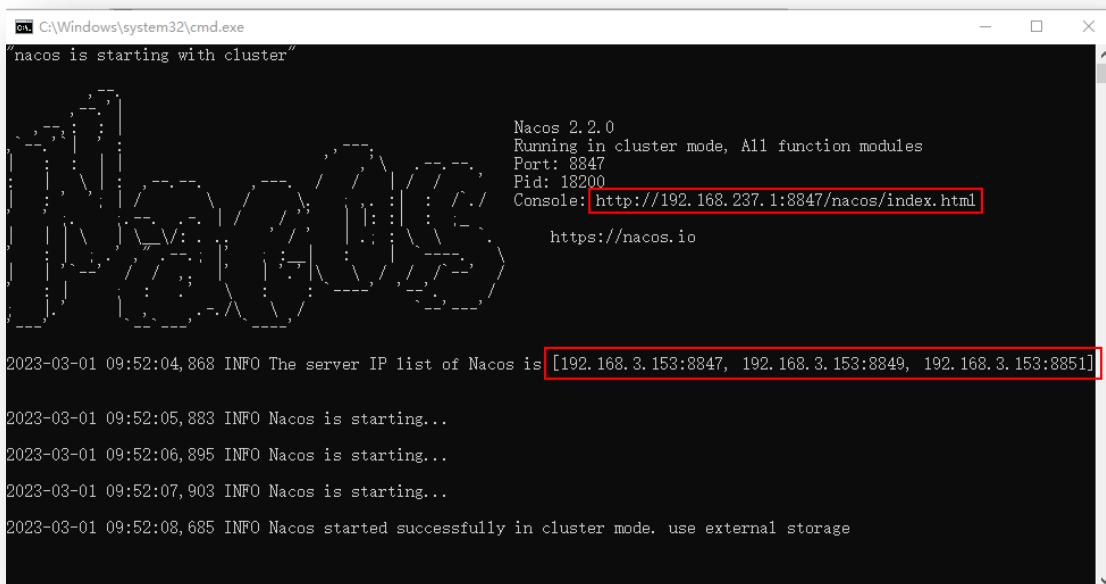
并修改各自目录中 conf/application.properties 文件中 nacos 的端口号为 8849 与 8851。

```
17 **** Spring Boot Related Configurations *
18 ### Default web context path:
19 server.servlet.contextPath=/nacos
20 ### Include message field
21 server.error.include-message=ALWAYS
22 ### Default web server port:
23 server.port=8849
24
```

```
10
17 **** Spring Boot Related Configurations *
18 ### Default web context path:
19 server.servlet.contextPath=/nacos
20 ### Include message field
21 server.error.include-message=ALWAYS
22 ### Default web server port:
23 server.port=8851
24
```

### (3) 启动集群

逐个双击三个 nacos 目录中的 bin/startup.cmd 命令，逐个启动三台 nacos。  
从启动日志上可以看到其提供的 SLB 服务访问地址及三台 Nacos 节点地址。



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window displays the following text:

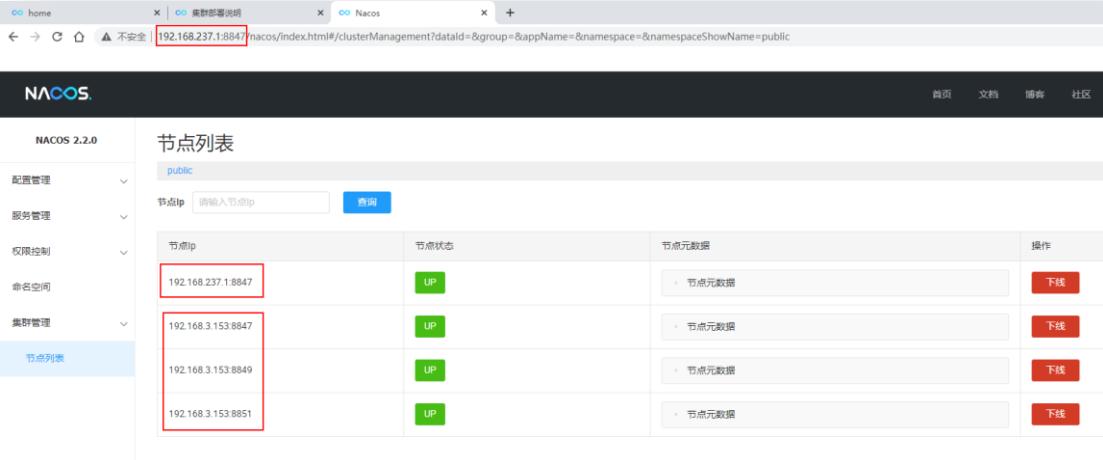
```
"nacos is starting with cluster"
Nacos 2.2.0
Running in cluster mode, All function modules
Port: 8847
Pid: 18200
Console: http://192.168.237.1:8847/nacos/index.html
https://nacos.io

2023-03-01 09:52:04,868 INFO The server IP list of Nacos is [192.168.3.153:8847, 192.168.3.153:8849, 192.168.3.153:8851]

2023-03-01 09:52:05,883 INFO Nacos is starting...
2023-03-01 09:52:06,895 INFO Nacos is starting...
2023-03-01 09:52:07,903 INFO Nacos is starting...
2023-03-01 09:52:08,685 INFO Nacos started successfully in cluster mode. use external storage
```

## (4) 查看 nacos 平台

在浏览器中使用 SLB 的 VIP 访问地址即可打开 nacos 服务器平台，查看到集群节点列表。点击“节点元数据”可查看 nacos 的节点元数据。



节点ip	节点状态	节点元数据	操作
192.168.237.1:8847	UP	节点元数据	下线
192.168.3.153:8847	UP	节点元数据	下线
192.168.3.153:8849	UP	节点元数据	下线
192.168.3.153:8851	UP	节点元数据	下线

### 2.8.2 Client 连接 Nacos 集群

直接将微服务配置文件 application.yml 中的 nacos 地址更换为 Nacos 集群的 VIP 地址。

```
cloud:  
  nacos:  
    discovery:  
      server-addr: 192.168.237.1:8847,192.168.237.1:8849,192.168.237.1:8851
```

微服务重启后便可以在 Nacos 平台上查看到它们的信息了。

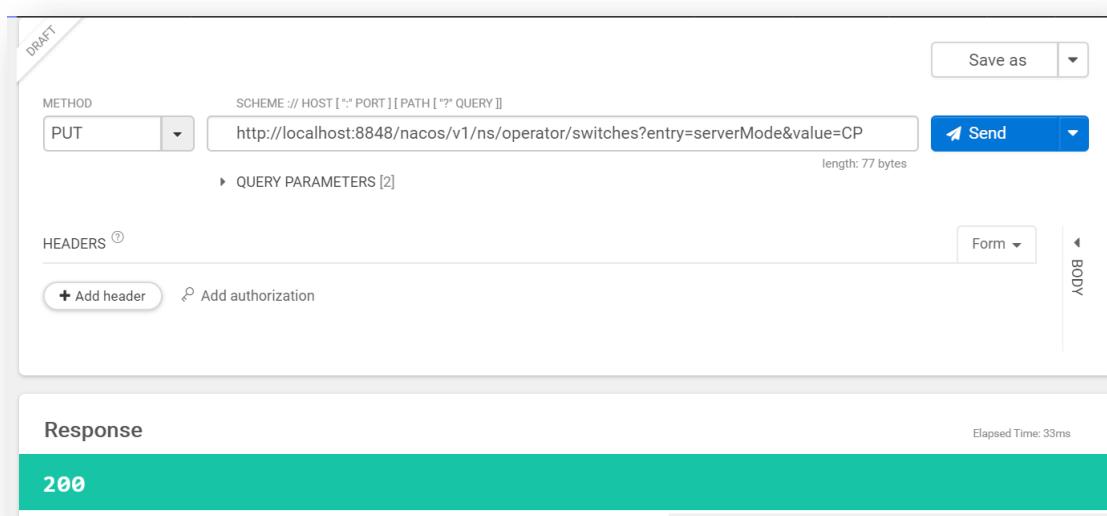


服务名	分组名称	集群数目
student-provider	DEFAULT_GROUP	1
student-consumer	DEFAULT_GROUP	1

### 2.8.3 Nacos 的 CAP 模式

默认情况下，Nacos Discovery 集群的数据一致性采用的是 AP 模式。但其也支持 CP 模式，需要进行转换。若要转换为 CP 的，可以提交如下 PUT 请求，完成 AP 到 CP 的转换。

<http://localhost:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP>



DRAFT Save as ▾

METHOD: PUT SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
http://localhost:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=CP Send length: 77 bytes

QUERY PARAMETERS [2]

HEADERS ⑦ Form ▾

+ Add header ⚒ Add authorization

BODY

Response Elapsed Time: 33ms

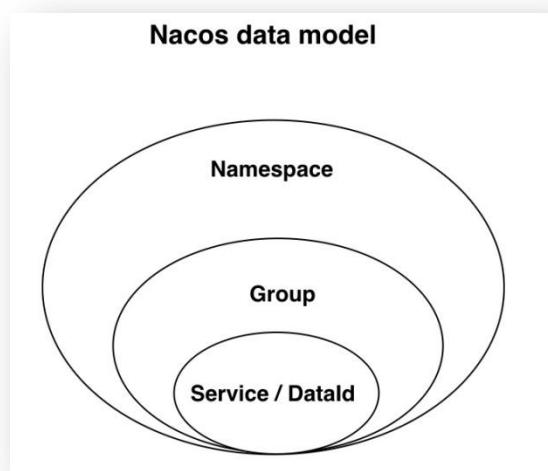
200

## 2.9 服务隔离

### 2.9.1 数据模型

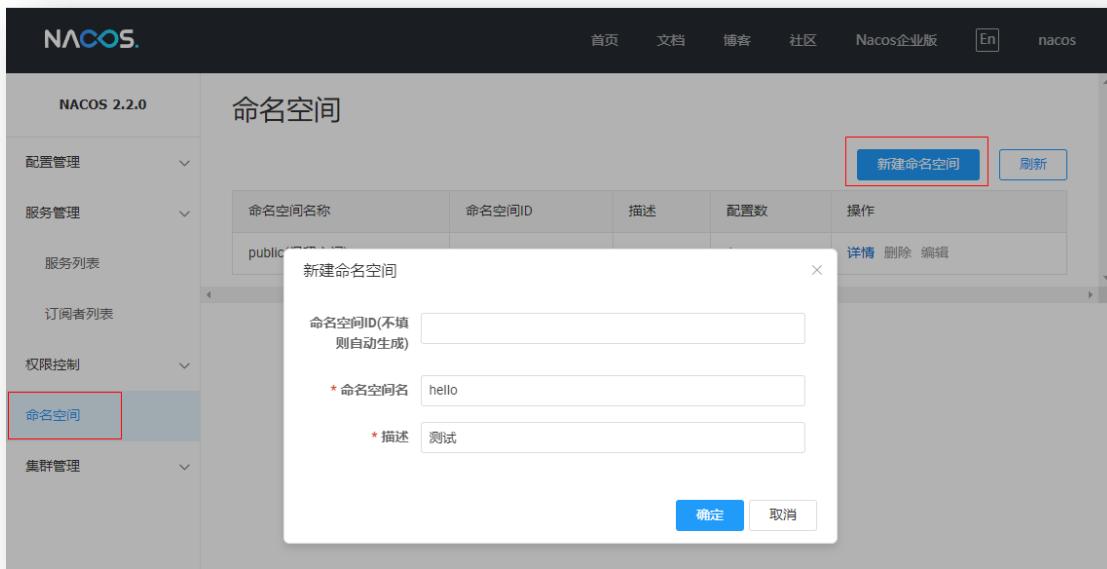
Nacos 中的服务是由三元组唯一确定的：namespace、group 与服务名称 service。 namespace 与 group 的作用是相同的，用于划分不同的区域范围，隔离服务。不同的是， namespace 的范围更大，不同的 namespace 中可以包含相同的 group。不同的 group 中可以包含相同的 service。namespace 的默认值为 public，group 的默认值为 DEFAULT\_GROUP。

它们之间的关系就如官方给出的下图所示。



### 2.9.2 新建命名空间

新建一个命名空间 hello，其会自动生成该命名空间的 ID。



The screenshot shows the Nacos 2.2.0 web interface. On the left sidebar, the '命名空间' (Namespace) option is selected and highlighted with a red box. In the main content area, there is a modal window titled '新建命名空间' (Create Namespace). Inside the modal, there are three input fields: '命名空间ID(不填则自动生成)' (Namespace ID (will be generated if left empty)), '命名空间名' (Namespace Name) with the value 'hello', and '描述' (Description) with the value '测试'. At the bottom of the modal are two buttons: '确定' (Confirm) and '取消' (Cancel). Above the modal, there is a table with columns: 命名空间名称 (Namespace Name), 命名空间ID (Namespace ID), 描述 (Description), 配置数 (Config Count), and 操作 (Operations). One row in the table is visible, showing 'public(保留空间)' as the namespace name and 'd41fb4ce-8667-42cd-ab8c-272c5da92f6c' as the namespace ID.



The screenshot shows the Nacos 2.2.0 web interface displaying the 'Namespace' list. The '命名空间' (Namespace) option is selected in the sidebar. The main area shows a table with the following data:

命名空间名称	命名空间ID	描述	配置数	操作
public(保留空间)			1	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>
hello	d41fb4ce-8667-42cd-ab8c-272c5da92f6c	测试	0	<a href="#">详情</a> <a href="#">删除</a> <a href="#">编辑</a>

### 2.9.3 启动三个 provider

启动三个 02-provider-nacos-8081 实例，它们提供的服务相同，不同的是 namespace、group 与 port：

- public + DEFAULT\_GROUP + 8081
- public + MY\_GROUP + 8082
- hello + MY\_GROUP + 8083

### 2.9.4 查看 Nacos 服务列表

查看 Nacos 服务列表，在 public 命名空间中有两个服务，服务名称相同，但分组不同。



The screenshot shows the Nacos 2.2.0 service list interface. On the left, there's a sidebar with 'NACOS 2.2.0' at the top, followed by '配置管理', '服务管理', '服务列表' (which is highlighted in blue), '订阅者列表', '权限控制', and '命名空间'. The main area has a title '服务列表' and two search input fields: 'public' and 'hello'. Below is a table with columns '服务名', '分组名称', and '集群数目'. It contains two rows: one for 'provider-depart' in 'DEFAULT\_GROUP' with a cluster count of 1, and another for 'provider-depart' in 'MY\_GROUP' with a cluster count of 1. The 'MY\_GROUP' row is highlighted with a red box.

查看 hello 命名空间，其中也有一个服务，服务名称与 public 中的相同，但分组为 MY\_GROUP。



This screenshot shows the same Nacos interface but with a different namespace selected. In the top right, it says '命名空间ID d41fb4ce-8667-42cd-ab8c-272c5da92f6c'. The 'public' tab is selected, while 'hello' is highlighted with a red box. The table below shows a single entry for 'provider-depart' in the 'MY\_GROUP' group, which is also highlighted with a red box.

## 2.9.5 调用 provider

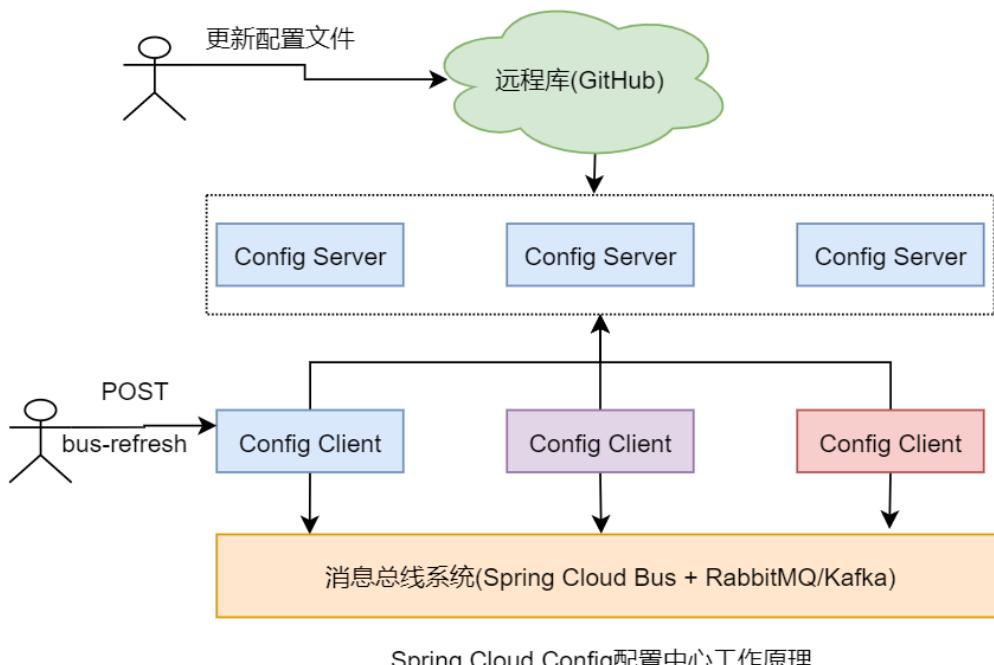
修改 02-consumer-nacos-8080 的配置文件，指定 nacos 服务发现的 group 与 namespace，就只会调用到指定范围中的服务。这就是 namespace+group 的服务隔离。

## 第3章 Nacos Config 服务配置中心

集群中每一台主机的配置文件都是相同的，对配置文件的更新维护就成为了一个棘手的问题。此时就出现了配置中心，将集群中每个节点的配置文件交由配置中心统一管理。相关产品很多，例如，Spring Cloud Config、Zookeeper、Apollo、Disconf(百度的，不再维护)等。但 Spring Cloud Alibaba 官方推荐使用 Nacos 作为微服务的配置中心。

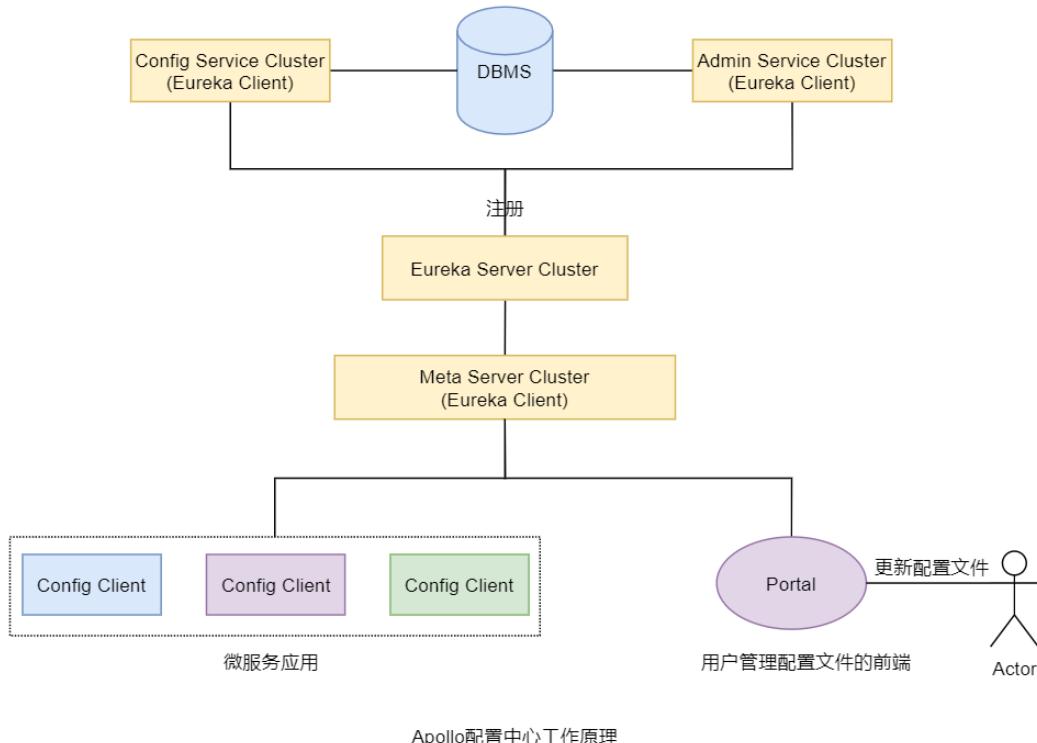
### 3.1 常见配置中心工作原理

#### 3.1.1 Spring Cloud Config



- 其存在三大问题：
- 无法自动感知更新
  - 存在羊群效应
  - 系统架构过于复杂

### 3.1.2 Apollo

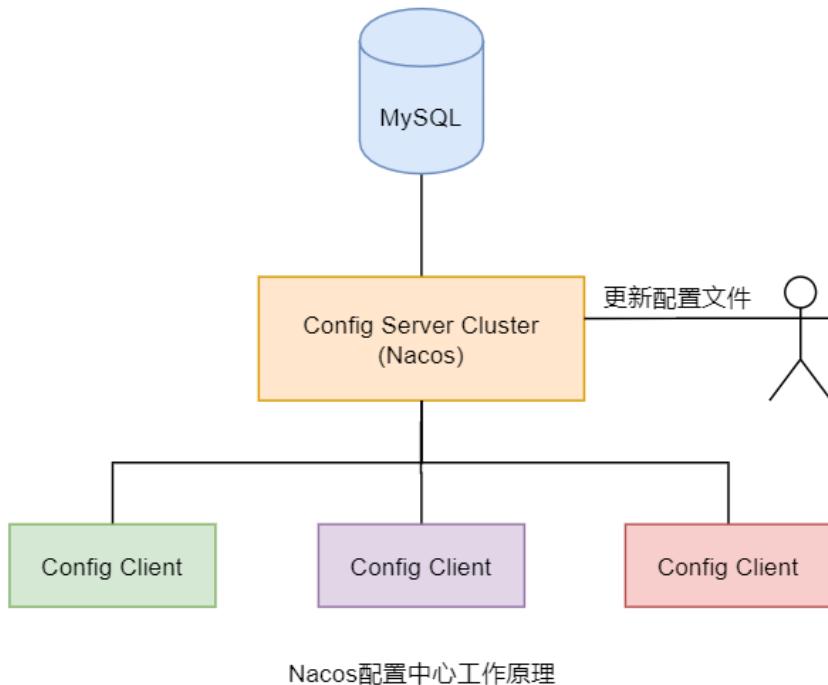


Apollo 配置中心工作原理

其 Config Client 可以自动感知配置文件的更新。但也存在两个不足：

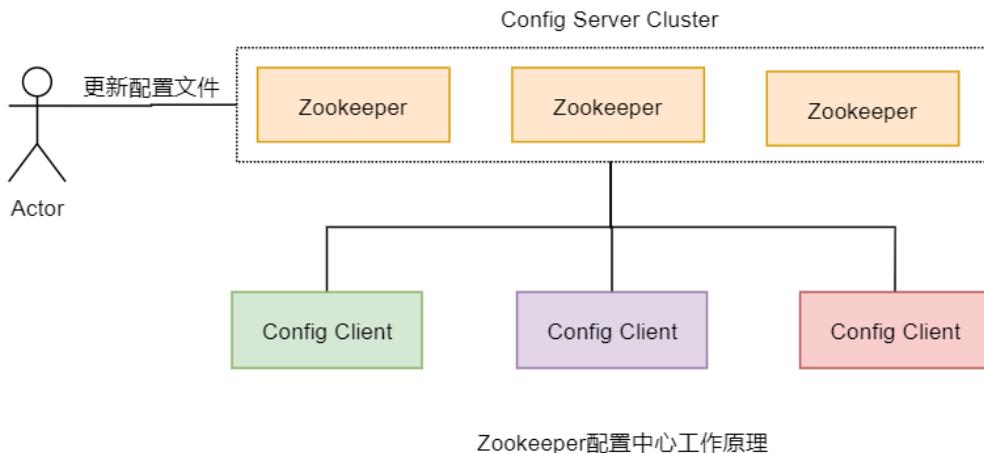
- 系统架构复杂
- 配置文件支持类型较少，其只支持 xml、text、properties，不支持 json、yml。

### 3.1.3 Nacos Config



- Config Client 通知自动感知配置中心中相应配置文件的更新。
- 架构简单
- 支持的配置文件类型较多(支持 JSON 与 YML)

### 3.1.4 Zookeeper



Zookeeper 作为配置中心，其工作原理与前面的三种都不同。其没有第三方服务器去存储配置数据，而是将配置数据存放在自己的 Znode 中了。当配置中心中的配置数据发生了变更，Config Client 也是可以自动感知到的（Watcher 监听机制）。

### 3.1.5 一致性问题

配置中心中的配置数据一般都是持久化在第三方服务器的，例如 DBMS、Git 远程库等。由于这些配置中心 Server 中根本就不存放数据，所以它们的集群中就不存在数据一致性问题。但像 Zookeeper，其作为配置中心，配置数据是存放在自己本地的。所以该集群中的节点是存在数据一致性问题的。Zookeeper 集群对于数据一致性采用的是 CP 模式。

作为注册中心，这些 Server 集群间是存在数据一致性问题的，它们采用的模式是不同的。Zookeeper (CP)、Eureka (AP)、Consul (AP)、Nacos (默认 AP，也支持 CP)。

## 3.2 获取远程配置

这里实现的需求是，应用的配置文件不在本地，而由 Nacos Config 进行管理。

### 3.2.1 Nacos 中完成配置

#### (1) 打开配置列表

启动 Nacos，然后打开 nacos 页面。

## 新建配置

\* Data ID: abcmsc-provider-depart.yml

\* Group: DEFAULT\_GROUP ×

[更多高级选项](#)

描述: abcmsc-provider-depart 的远程配置文件

配置格式:  TEXT  JSON  XML  YAML  HTML  Properties

\* 配置内容:

```
1  depart:
2    name: new-depart
3
4  server:
5    port: 8081
6
7  spring:
8    cloud:
9      # 指定nacos server地址
10   nacos:
11     discovery:
12       server-addr: localhost:8848
13
14  jpa:
15    generate-ddl: true
16    show-sql: true
```

发布 返回

Data ID 用于指定在 nacos config 中保存的配置文件的名称。该文件可以是 yml 或 properties，但名称必须为微服务名称。应用在启动时就是通过微服务名称在 nacos config 中查找相应的配置文件的。

这里要使用的配置格式为 YAML。发布，返回后，就可看到如下清单了。

配置管理 | public    查询结果: 共查询到 1 条满足要求的配置。

	Data Id	Group	归属应用:	操作
<input type="checkbox"/>	abcmsc-provider-depart.yml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>

删除 导出选中的配置 克隆

每页显示: 10 < 上一页

## (2) 填写配置内容

将原来 application.yml 中的全部内容复制到该页面，并将原来的 spring.application.name 属性删除。

```
server:
  port: 8081

spring:
  cloud:
    # 指定 nacos server 地址
    nacos:
      discovery:
        server-addr: localhost:8848

  jpa:
    generate-ddl: true
    show-sql: true
    hibernate:
      ddl-auto: none

  # 配置数据源
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql:///test?useUnicode=true&useSSL=false&characterEncoding=utf8
    username: root
    password: 111

  logging:
    # 设置日志输出格式
    pattern:
      console: level-%level %msg%
    level:
      root: info
      org.hibernate: info
      org.hibernate.type.descriptor.sql.BasicBinder: trace
      org.hibernate.type.descriptor.sql.BasicExtractor: trace
      com.abc.provider: debug
```

### 3.2.2 定义提供者 03-provider-nacos-config-8081

#### (1) 定义工程

复制 02-provider-nacos-8081 工程，并重命名为 03-provider-nacos-config-8081。

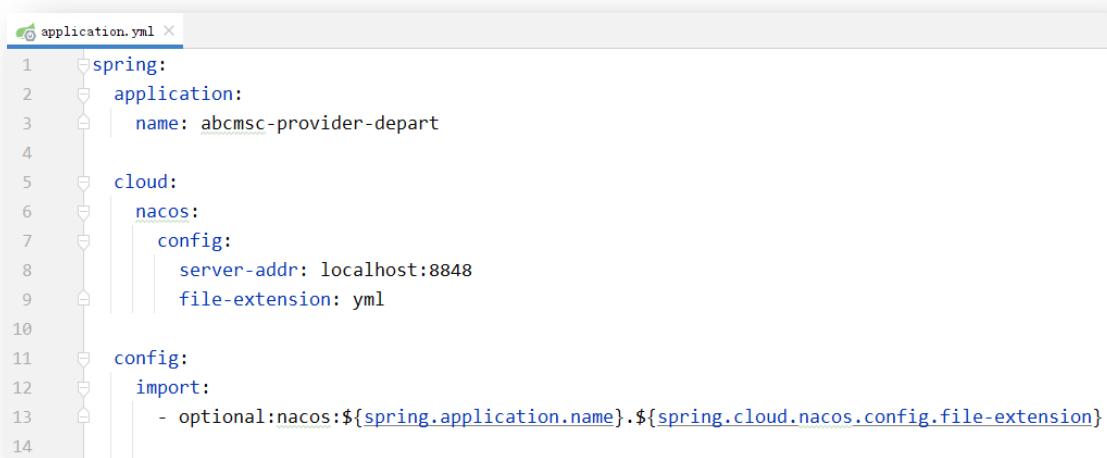
#### (2) 修改 pom

在 pom 中添加如下依赖。

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

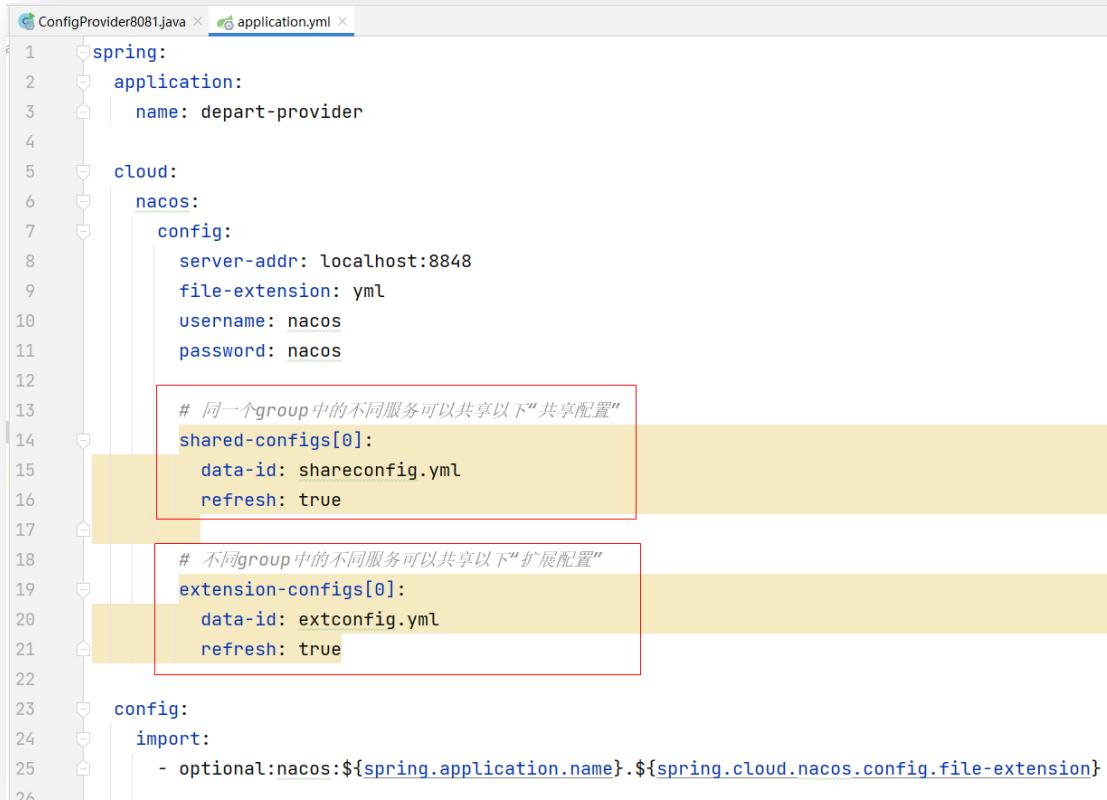
#### (3) 修改 application.yml

删除原有的 application.yml 文件的内容，替换为如下内容。



```
application.yml
1 spring:
2   application:
3     name: abcmsc-provider-depart
4
5   cloud:
6     nacos:
7       config:
8         server-addr: localhost:8848
9         file-extension: yml
10
11   config:
12     import:
13       - optional:nacos:${spring.application.name}.${spring.cloud.nacos.config.file-extension}
```

### 3.2.3 关于配置文件的扩展



```
spring:
  application:
    name: depart-provider

  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml
        username: nacos
        password: nacos

# 同一个group中的不同服务可以共享以下“共享配置”
shared-configs[0]:
  data-id: shareconfig.yml
  refresh: true

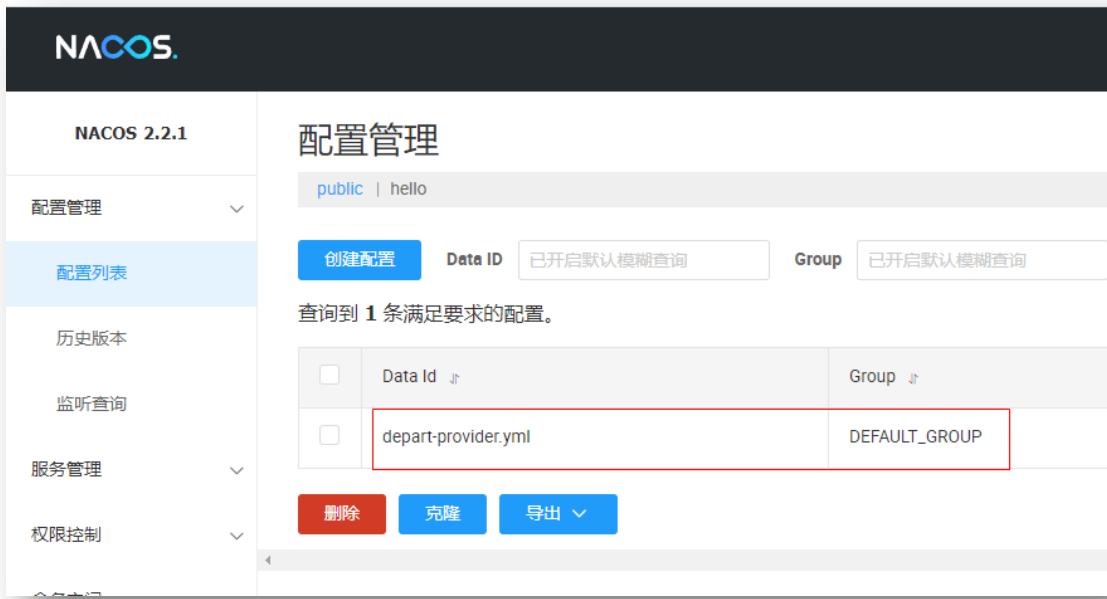
# 不同group中的不同服务可以共享以下“扩展配置”
extension-configs[0]:
  data-id: extconfig.yml
  refresh: true

config:
  import:
    - optional:nacos:${spring.application.name}.${spring.cloud.nacos.config.file-extension}
```

当前服务配置、共享配置与扩展配置的加载顺序为：共享配置，扩展配置，当前服务配置。若在三个配置中具有相同属性设置，但它们具有不同的值，那么，后加载的会将先加载的给覆盖。即这三类配置的优先级由低到高是：共享配置，扩展配置，当前服务配置

当前服务配置可以存在于三个地方：

远程配置文件：(Nacos config 中)



The screenshot shows the Nacos 2.2.1 configuration management interface. On the left, there's a sidebar with 'NACOS 2.2.1' at the top, followed by '配置管理' (Configuration Management), '配置列表' (Configuration List) which is selected, '历史版本' (History Versions), '监听查询' (Monitoring Query), '服务管理' (Service Management), and '权限控制' (Permission Control). Below the sidebar is a search bar with 'public | hello'. There are two tabs: '创建配置' (Create Configuration) and 'Data ID' (selected). Under 'Data ID', there are two buttons: '已开启默认模糊查询' (Default Fuzzy Search Enabled) and 'Group' (selected). A message says '查询到 1 条满足要求的配置' (1 configuration found). A table lists one configuration: 'depart-provider.yml' under 'Data Id' and 'DEFAULT\_GROUP' under 'Group'. At the bottom of the table are three buttons: '删除' (Delete), '克隆' (Clone), and '导出' (Export).

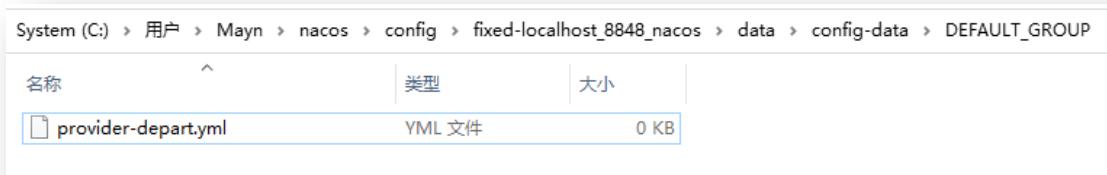
快照文件：



A file explorer window showing the path: System (C:) > 用户 > Mayn > nacos > config > fixed-localhost\_8848\_nacos > snapshot > DEFAULT\_GROUP. The table below lists a single file: depart-provider.yml, which is a YML file and 2 KB in size.

名称	类型	大小
depart-provider.yml	YML 文件	2 KB

本地配置文件：(主动写入的配置文件)



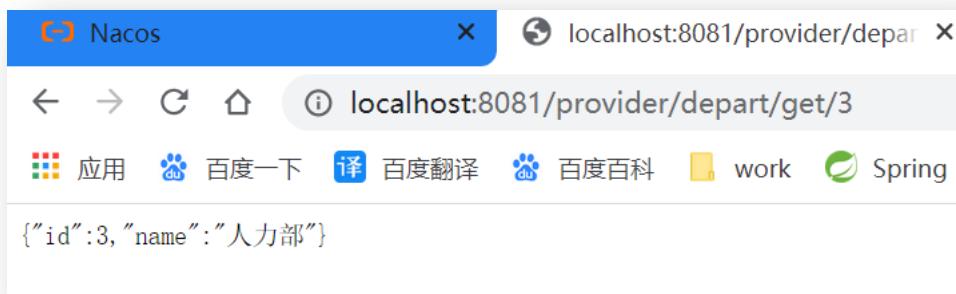
A file explorer window showing the path: System (C:) > 用户 > Mayn > nacos > config > fixed-localhost\_8848\_nacos > data > config-data > DEFAULT\_GROUP. The table below lists a single file: provider-depart.yml, which is a YML file and 0 KB in size.

名称	类型	大小
provider-depart.yml	YML 文件	0 KB

这三个同名文件也存在加载顺序问题，它们的加载顺序为：本地配置文件、远程配置文件、快照配置文件。只要系统加载到了配置文件，那么后面的就不再加载。

### 3.2.4 运行访问

启动 03-provider-nacos-config-8081，页面正常访问即可。



### 3.3 动态更新配置

#### 3.3.1 需求

这里要实现的需求是：从数据库中根据 id 查询到的 depart 的 name 值，在浏览器上显示的并不是 DB 中的 name，而是来自于 nacos 的动态配置 depart.name。

#### 3.3.2 修改 Nacos 配置数据

直接在 nacos config 配置页面修改配置信息，例如添加一个 depart.name 属性。修改完毕后，再次发布即可。



The screenshot shows the Nacos Config management interface. The top navigation bar has 'public' selected. Below it, there's a search bar with 'abcmsc-provider-depart.yml' and a 'Group' dropdown set to 'DEFAULT\_GROUP'. The main table lists one configuration entry:

Data Id	Group	归属应用	操作
abcmsc-provider-depart.yml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a> (highlighted)   <a href="#">删除</a>   <a href="#">更多</a>

At the bottom, there are buttons for 'Delete', 'Export Selected Configuration', 'Clone', and page controls for '每页显示: 10' and '< 上一页'.

配置格式:  TEXT  JSON  XML  YAML

配置内容 [?](#):

```
1 depart:  
2   name: new-depart  
3  
4 server:  
5   port: 8081
```

### 3.3.3 修改提供者工程

直接修改 03-provider-nacos-config-8081 工程中的 DepartServiceImpl 类。

首先在需要动态更新配置数据的类上添加@RefreshScope 注解，然后再添加 departName 属性，该属性值读取自配置文件中的 depart.name 属性。



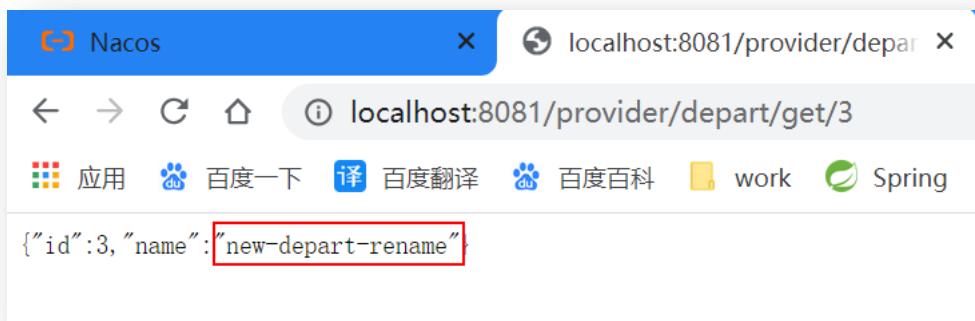
```
@Service  
@RefreshScope  
public class DepartServiceImpl implements DepartService {  
    @Autowired  
    private DepartRepository repository;  
  
    @Value("${depart.name}")  
    private String departName;
```

为了演示配置信息的动态更新效果，再修改如下方法返回值。

```
@Override
public Depart getDepartById(int id) {
    if(repository.existsById(id)) {
        Depart one = repository.getOne(id);
        one.setName(departName);
        return one;
    }
    Depart depart = new Depart();
    depart.setName("no this depart " + departName);
    return depart;
}
```

### 3.3.4 刷新访问页面

重启应用后，刷新页面可以看到数据已经更新。然后再多次更新远程配置文件，无需重启 03-provider-nacos-config-8081 工程，直接刷新访问页面即可获取到更新的数据。



### 3.3.5 长轮询模型

Nacos Config Server 中配置数据的变更，Nacos Config Client 是如何知道的呢？Nacos 采用的是长轮询模型。

长轮询模型整合了 Push 与 Pull 模型的优势。Client 仍定时发起 Pull 请求，查看 Server 端数据是否更新。若发生了更新，则 Server 立即将更新数据以响应的形式 Push 给 Client 端；

若没有发生更新, Server 端并不向 Client 进行 Push, 而是临时性的保持住这个连接一段时间。若在此时间段内, Server 端数据发生了变更, 则立即将变更数据 Push 给 Client。若仍未发生变更, 则放弃这个连接。等待着下一次 Client 的 Pull 请求。

长轮询模型, 是 Push 与 Pull 模型的整合, 既降低了 Push 模型中长连接的维护问题, 又降低了 Push 模型实时性较低的问题。

## 3.4 多环境选择的实现

### 3.4.1 什么是多环境选择

在开发应用时, 通常同一套程序会被运行在多个不同的环境, 例如, 开发、测试、生产环境等。每个环境的数据库地址、服务器端口号等配置都会不同。若在不同环境下运行时将配置文件修改为不同内容, 那么, 这种做法不仅非常繁琐, 而且很容易发生错误。此时就需要定义出不同的配置信息, 在不同的环境中选择不同的配置。

### 3.4.2 新增多环境配置文件

#### (1) 克隆文件

在 Nacos config 服务器中克隆两个配置文件。



The screenshot shows the Nacos Config server interface for the 'public' group. The search results show one configuration entry: 'abcmsc-provider-depart.yml'. This entry is selected, indicated by a red border around its row. At the bottom of the table, there are three buttons: '删除' (Delete), '导出选中的配置' (Export Selected Configuration), and '克隆' (Clone). The '克隆' button is also highlighted with a red border.

	Data Id	Group
<input checked="" type="checkbox"/>	abcmsc-provider-depart.yml	DEFAULT_GROUP



再以相同的方式克隆一个文件，最终可以看到多出了两个配置文件。  
注意，多环境选择配置文件的文件名中，后面必须是-{profile}。



The screenshot shows the 'Configuration Management' interface for the 'public' space. The search results indicate 3 configurations found. The table lists three entries:

	Data Id	Group
<input type="checkbox"/>	abcmsc-provider-depart.yml	DEFAULT_GROUP
<input type="checkbox"/>	abcmsc-provider-depart-dev.yml	DEFAULT_GROUP
<input type="checkbox"/>	abcmsc-provider-depart-test.yml	DEFAULT_GROUP

At the bottom, there are three buttons: '删除' (Delete), '导出选中的配置' (Export Selected Configuration), and '克隆' (Clone). The 'abcmsc-provider-depart-dev.yml' entry in the table is highlighted with a red border.

## (2) 修改配置文件内容

分别修改两个配置文件中的 depart.name 属性。

配置格式:  TEXT  JSON  XML  YAML

配置内容 [?](#):

```
1 depart:  
2   name: new-depart-dev  
3  
4 server:  
5   port: 8081  
6
```

配置格式:  TEXT  JSON  XML  YAML

配置内容 [?](#):

```
1 depart:  
2   name: new-depart-test  
3  
4 server:  
5   port: 8081  
6
```

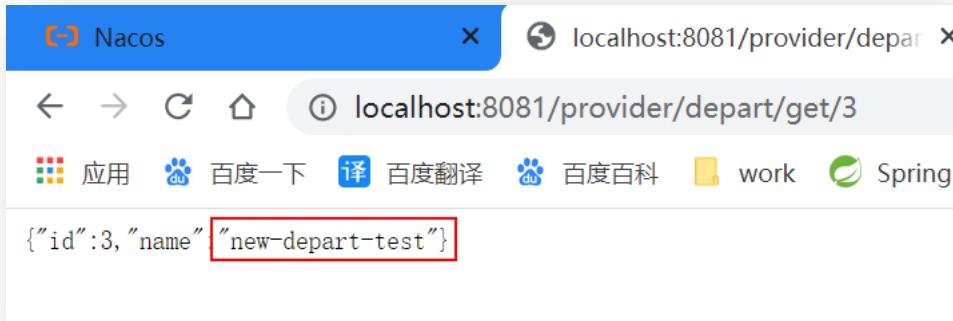
### 3.4.3 修改应用的配置文件

修改 03-provider-nacos-config -8081 工程的 application.yml 文件。在其中添加多环境选择配置，并修改要导入的 nacos 配置中心中配置文件的名称格式。



```
application.yml
1 spring:
2   cloud:
3     nacos:
4       config:
5         server-addr: localhost:8848
6         file-extension: yml
7
8   application:
9     name: provider-depart
10
11 profiles:
12   active: test
13
14 config:
15   import:
16     - optional:nacos:${spring.application.name}-${spring.profiles.active}.${spring.cloud.nacos.config.file-extension}
```

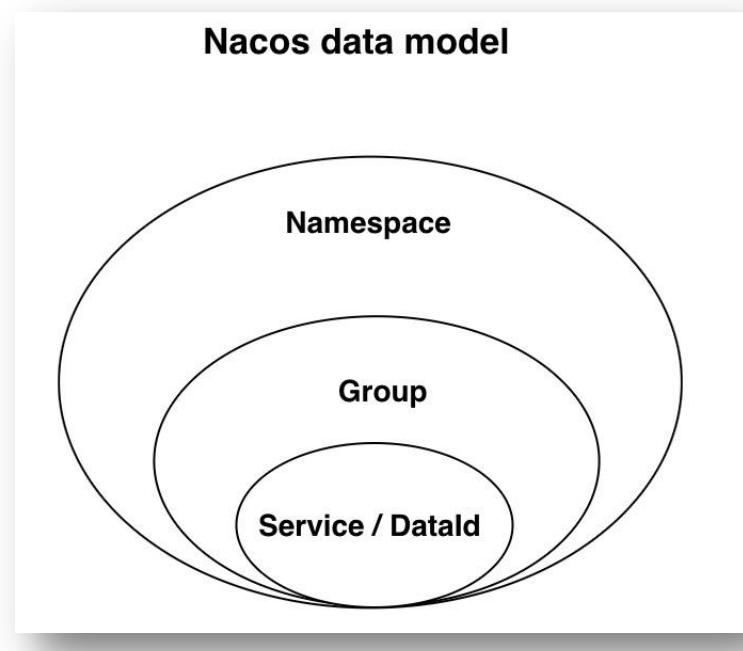
### 3.4.4 访问



## 3.5 配置隔离

### 3.5.1 数据模型

namespace 与 group 除了能够隔离服务外，还可以隔离配置文件。在官方的数据模型图中就可以看到。



### 3.5.2 定义三个 provider 的配置文件

在 Nacos Config 中定义三个 03-provider-config-8081 的配置文件,它们的内容几乎相同,不同的是 namespace、group 与 port:

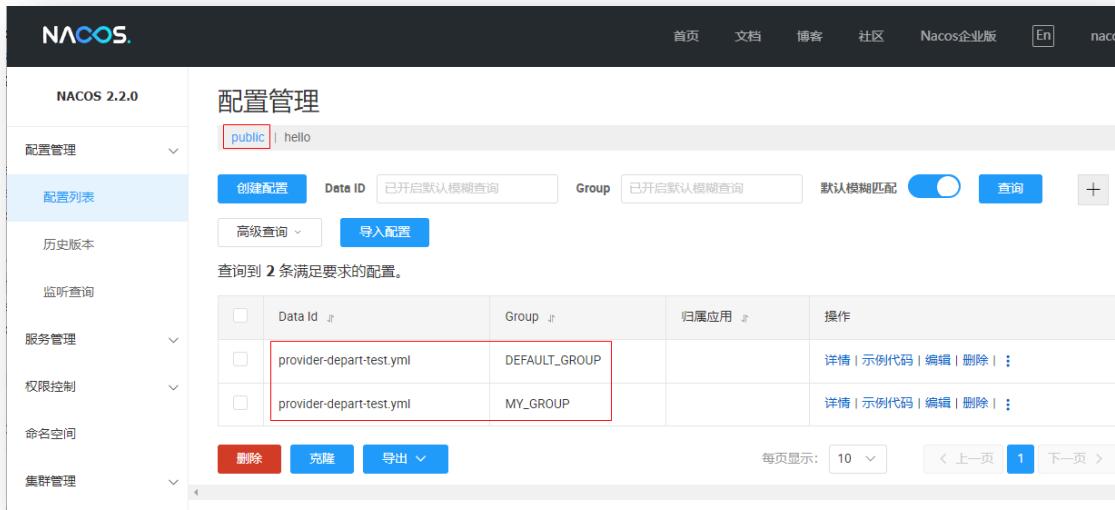
- public + DEFAULT\_GROUP + 8081
- public + MY\_GROUP + 8082
- hello + MY\_GROUP + 8083

### 3.5.3 启动三个 provider

通过对 03-provider-config-8081 的 application.yml 中为 spring.cloud.nacos.config 下的 namespace 与 group 指定不同的值,启动三个 provider,让它们加载不同 namespace 与 group 下的不同的配置文件。

### 3.5.4 查看 Nacos 配置列表

查看 Nacos 配置列表,在 public 命名空间中有两个配置文件,服务名称相同,但分组不同。



The screenshot shows the Nacos 2.2.0 Configuration Management interface. The left sidebar has sections like Configuration Management, Configuration List, History Version, Listener Query, Service Management, Permission Control, Namespace, and Cluster Management. The main area is titled 'Configuration Management' with a search bar containing 'public' and 'hello'. It shows two configuration entries:

Data Id	Group	操作
provider-depart-test.yml	DEFAULT_GROUP	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>
provider-depart-test.yml	MY_GROUP	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>

查看 hello 命名空间，其中也有一个配置文件，名称与 public 中的相同，但分组为 MY\_GROUP。



The screenshot shows the Nacos 2.2.0 Configuration Management interface. The left sidebar has sections like Configuration Management, Configuration List, History Version, Listener Query, Service Management, Permission Control, and Namespace. The main area is titled 'Configuration Management' with a search bar containing 'public' and 'hello'. It shows one configuration entry:

Data Id	Group	操作
provider-depart-test.yml	MY_GROUP	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>

### 3.5.5 查看 Nacos 服务列表

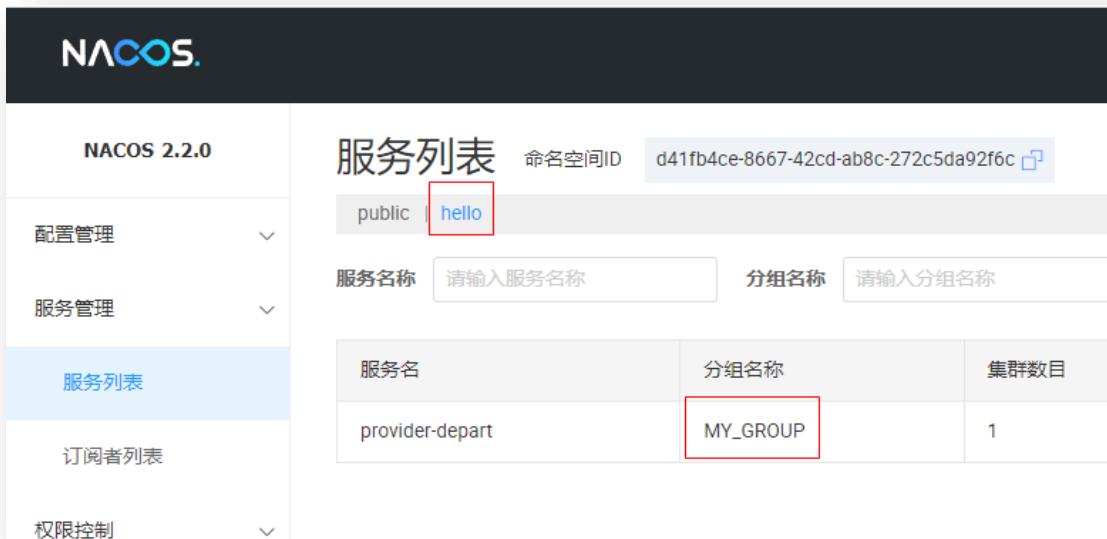
查看 Nacos 服务列表，在 public 命名空间中有两个服务，服务名称相同，但分组不同。



The screenshot shows the Nacos service list interface. On the left, there's a sidebar with 'NACOS 2.2.0' at the top, followed by dropdown menus for '配置管理' (Configuration Management), '服务管理' (Service Management), '服务列表' (Service List) which is selected and highlighted in blue, '订阅者列表' (Subscriber List), '权限控制' (Permission Control), and '命名空间' (Namespace). The main area is titled '服务列表' (Service List) and shows a table with two rows. The first row has 'provider-depart' in the '服务名' (Service Name) column, 'DEFAULT\_GROUP' in the '分组名称' (Group Name) column, and '1' in the '集群数目' (Cluster Count) column. The second row has 'provider-depart' in the '服务名' column, 'MY\_GROUP' in the '分组名称' column, and '1' in the '集群数目' column. A red box highlights the 'public' tab in the top navigation bar.

服务名	分组名称	集群数目
provider-depart	DEFAULT_GROUP	1
provider-depart	MY_GROUP	1

查看 hello 命名空间，其中也有一个服务，服务名称与 public 中的相同，但分组为 MY\_GROUP。



This screenshot shows the same Nacos interface but with a different namespace selected. In the top navigation bar, the 'public' tab is still highlighted in blue, but the 'hello' tab is also visible. The main service list table now shows a single row for 'provider-depart' with 'MY\_GROUP' as its group name and a cluster count of 1. A red box highlights the 'hello' tab in the top navigation bar.

服务名	分组名称	集群数目
provider-depart	MY_GROUP	1

### 3.5.6 调用 provider

此时 consumer 调用这些 provider 的方式，与前面“服务隔离”时的调用方式完全相同。

修改 02-consumer-nacos-8080 的配置文件，指定 nacos 服务发现的 group 与 namespace，就只会调用到指定范围中的服务。这就是 namespace+group 的服务隔离。

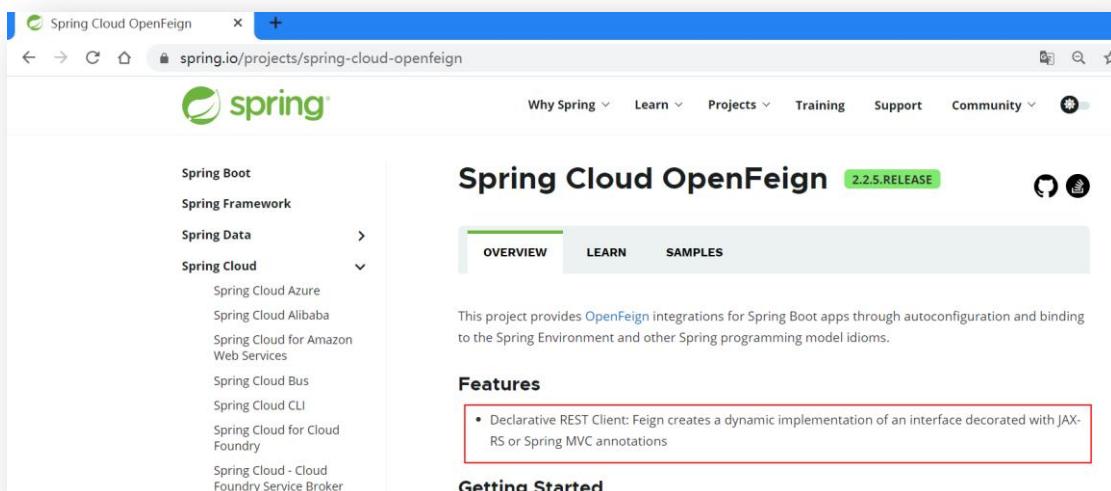
## 第4章 OpenFeign 与负载均衡

前面消费者对于微服务的消费是通过 `RestTemplate` 完成的，这种方式的弊端是很明显的：

- 消费者对提供者的调用无法与业务接口完全吻合。例如，原本 `Service` 接口中的方法是有返回值的，但经过 `RestTemplate` 相关 API 调用后没有了其返回值，最终执行是否成功用户并不清楚。再例如 `RestTemplate` 的对数据的删除与修改操作方法都没有返回值。
- 代码编写不方便，不直观。提供者原本是按照业务接口提供服务的，而经过 `RestTemplate` 一转手，变为了 URL，使得程序员在编写消费者对提供者的调用代码时，变得不直接、不明了。没有直接通过业务接口调用方便、清晰。

### 4.1 概述

#### 4.1.1 OpenFeign 简介



**【翻译】声明式 REST 客户端：**Feign 通过使用 JAX-RS（Java Api eXtensions of RESTful web Services）或 SpringMVC 注解的修饰方式，生成接口的动态实现。

**【解析】**Feign，假装、伪装。OpenFeign 可以将提供者提供的 Restful 服务伪装为接口进行消费，消费者只需使用“feign 接口 + 注解”的方式即可直接调用提供者提供的 Restful 服务，而无需再使用 `RestTemplate`。

**【总结】**对于 OpenFeign，可简单总结为以下几点：

- OpenFeign 只涉及 Consumer 与 Provider 无关。因为其是用于 Consumer 调用 Provider 的
- OpenFeign 仅仅就是一个伪客户端，其不会对请求做任务的处理
- OpenFeign 是通过注解的方式实现 RESTful 请求的

### 4.1.2 OpenFeign 与 Ribbon

OpenFeign 具有负载均衡功能，其可以对指定的微服务采用负载均衡方式进行消费、访问。之前老版本 Spring Cloud 所集成的 OpenFeign 默认采用了 Ribbon 负载均衡器。但由于 Netflix 已不再维护 Ribbon，所以从 Spring Cloud 2021.x 开始集成的 OpenFeign 中已彻底丢弃 Ribbon，而是采用 Spring Cloud 自行研发的 Spring Cloud Loadbalancer 作为负载均衡器。

## 4.2 OpenFeign 用法

### 4.2.1 消费者工程使用

这里无需修改提供者工程，只需修改消费者工程即可。

#### (1) 创建工程

复制 02-consumer-nacos-8080，并重命名为 04-consumer-feign-8080。

#### (2) 添加 openfeign 依赖

注意，这里使用的是 spring-cloud-starter-openfeign 依赖，而非 spring-cloud-starter-feign 依赖。

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

#### (3) 定义 Feign 接口

注意，这里的接口名可以是任意的名称，接口中的方法名也可以是任意的名称。但 @FeignClient 参数指定的提供者服务名称是不能修改的，接口与方法上添加的 @XxxMapping 中的参数是不能修改的，必须与提供者相应的请求 URI 相同。

由于其充当的是业务接口，所以一般其定义在 service 包中。

```
2 usages
@FeignClient(value = "abcmsc-provider-depart", path = "/provider/depart")
public interface DepartService {
    1 usage
    @PostMapping(PathVariable="/save")
    boolean saveDepart(@RequestBody Depart depart);
    1 usage
    @DeleteMapping(PathVariable="/del/{id}")
    boolean removeDepartById(@PathVariable("id") int id);
    1 usage
    @PutMapping(PathVariable="/update")
    boolean modifyDepart(@RequestBody Depart depart);
    1 usage
    @GetMapping(PathVariable="/get/{id}")
    Depart getDepartById(@PathVariable("id") int id);
    1 usage
    @GetMapping(PathVariable="/list")
    List<Depart> listAllDeparts();
}
```

#### (4) 删除 Config 类

由于这里使用的是 Feign，所以无需再定义 RestTemplate 了，所以就将原来的 Config 类删除即可。

```
// @Configuration
public class DepartCodeConfig {

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

## (5) 修改处理器

这里使用 Feign 接口来消费微服务。

```
@RestController
@RequestMapping("/consumer/depart")
public class DepartController {
    @Autowired
    private DepartService service;

    @PostMapping("/save")
    public boolean saveHandle(Depart depart) {
        return service.saveDepart(depart);
    }

    @DeleteMapping("/del/{id}")
    public boolean removeHandle(@PathVariable("id") int id) {
        return service.removeDepartById(id);
    }
}
```

```
    @PutMapping("/update")
    public boolean modifyHandle(Depart depart) {
        return service.modifyDepart(depart);
    }

    @GetMapping("/get/{id}")
    public Depart getDepartHandle(@PathVariable("id") int id) {
        return service.getDepartById(id);
    }

    @GetMapping("/list")
    public List<Depart> listAllDepartsHandle() {
        return service.listAllDeparts();
    }
}
```

## (6) 修改启动类

在启动类上添加@EnableFeignClients 注解。

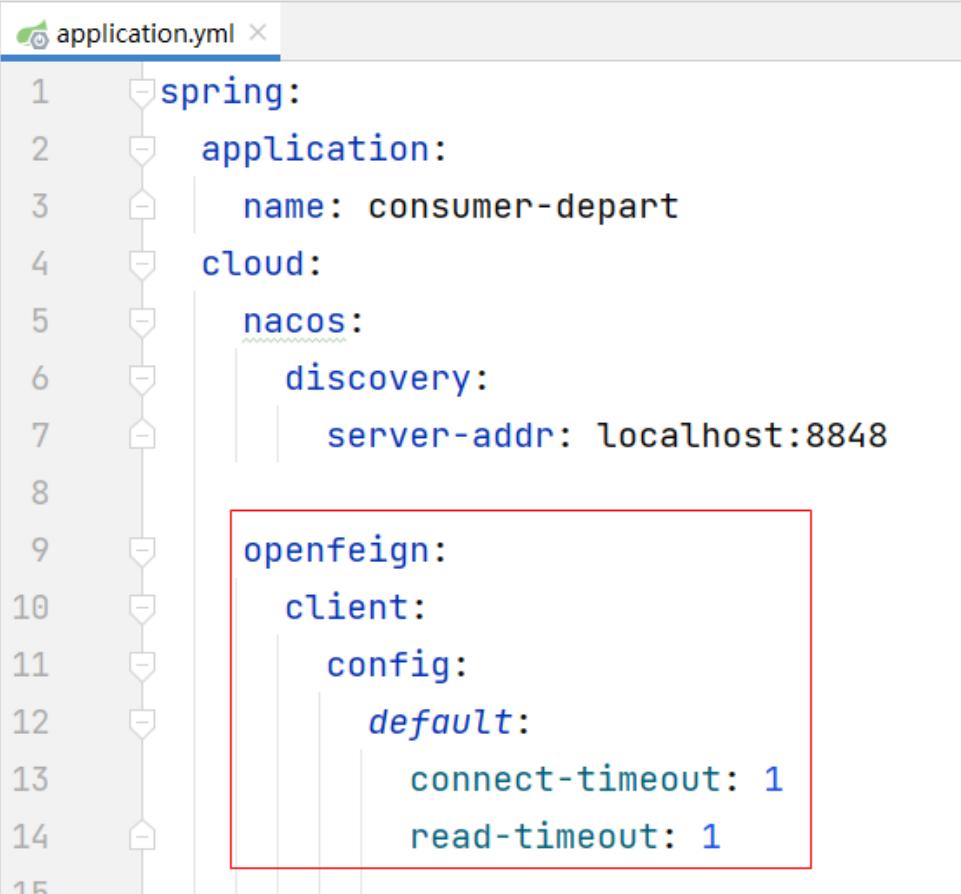
```
@EnableFeignClients
@SpringBootApplication
public class ConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
}
```

### 4.2.2 超时设置

#### (1) 全局超时设置

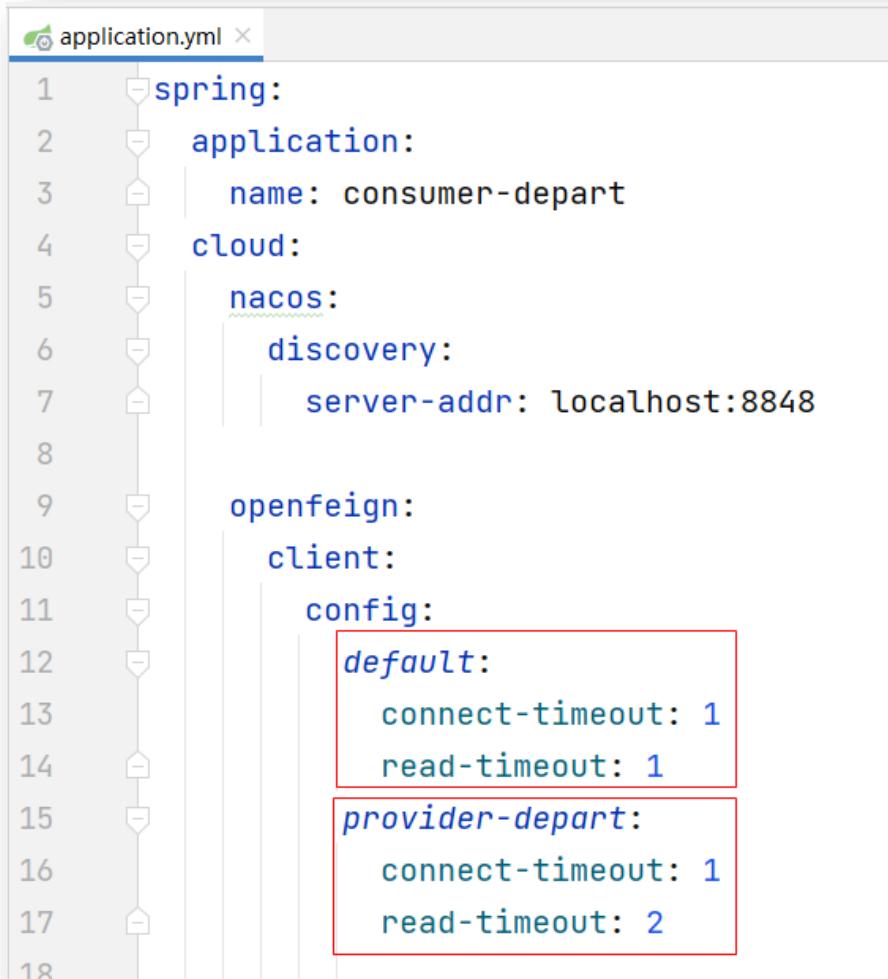
在 04-consumer-feign-8080 工程的配置文件中直接添加如下内容：



```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4   cloud:
5     nacos:
6       discovery:
7         server-addr: localhost:8848
8
9   openfeign:
10    client:
11      config:
12        default:
13          connect-timeout: 1
14          read-timeout: 1
15
```

## (2) 局部超时设置

在全局设置的基础之上，若想单独对某些微服务单独设置超时时间，只需要将前面配置中的 `default` 修改为微服务名称即可。局部设置的优先级要高于全局设置的。



```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4   cloud:
5     nacos:
6       discovery:
7         server-addr: localhost:8848
8
9   openfeign:
10  client:
11    config:
12      default:
13        connect-timeout: 1
14        read-timeout: 1
15      provider-depart:
16        connect-timeout: 1
17        read-timeout: 2
18
```

#### 4.2.3 Gzip 压缩设置

OpenFeign 可对请求与响应进行压缩设置。在 04-consumer-feign-8080 工程的配置文件中直接添加如下内容：

```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4   cloud:
5     nacos:
6       discovery:
7         server-addr: localhost:8848
8
9   openfeign:
10    client:
11      config:
12        default:
13          connect-timeout: 1
14          read-timeout: 1
15        provider-depart:
16          connect-timeout: 1
17          read-timeout: 2
18
19   compression:
20     request:
21       enabled: true
22       mime-types: ["text/xml", "application/xml", "application/json", "video/mp4"]
23       min-request-size: 1024
24     response:
25       enabled: true
26
```

#### 4.2.4 选择远程调用的底层实现技术

##### (1) 理论基础

feign 的远程调用底层实现技术默认采用的是 JDK 的 URLConnection，同时还支持 HttpClient 与 OkHttp。

由于 JDK 的 URLConnection 不支持连接池，通信效率很低，所以生产中是不会使用该默认实现的。所以在 Spring Cloud OpenFeign 中直接将默认实现变为了 HttpClient，同时也支持 OkHttp。

用户可根据业务需求选择要使用的远程调用底层实现技术。

##### (2) 配置说明

在 `spring.cloud.openfeign.httpclient` 下有大量 HttpClient 的相关属性设置。其中可以发现，`spring.cloud.openfeign.httpclient.enabled` 默认为 `true`。

在 `spring.cloud.openfeign.okhttp.enabled` 默认值为 `false`，表明默认没有启动 OkHttp。

OkHttp 的读超时设置共用了 HttpClient 的读超时设置属性。

## 4.3 负载均衡

前面的消费者例子是通过 Feign 接口来消费微服务的，但没体现出其负载均衡的功能。所以，下面将进行 Feign 负载均衡功能展示。

### 4.3.1 需求

下面将构建这样一个系统：一个微服务由三个提供者提供，而消费者对这三个提供者进行负载均衡访问。

### 4.3.2 启动提供者工程

将 02-provider-nacos-8081 工程启动三个实例，它们的端口号分别为 8081、8082 与 8083。

### 4.3.3 默认负载均衡策略

在 nacos 及三个提供者均启动的前提下，直接启动运行 04-consumer-feign-8080 消费者工程即可。然后再访问消费者工程。每刷新一次页面，其显示的部门名称后的端口号就发生变化，说明实现了负载均衡。再仔细查看其变化顺序，一定是 8082、8083、8084，循环变化。那是因为，OpenFeign 的负载均衡器 Ribbon 默认采用的是轮询算法。

### 4.3.4 更换负载均衡策略

在 04-consumer-feign-8080 项目上直接修改。

#### (1) 定义一个 Config 类

```
2 usages
public class MyConfig {
    no usages
    @Bean
    public ReactorLoadBalancer<ServiceInstance> loadBalancer(Environment e, LoadBalancerClientFactory factory){
        String name = e.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new RandomLoadBalancer(factory.getLazyProvider(name, ServiceInstanceListSupplier.class), name);
    }
}
```

## (2) 修改启动类

在启动类上添加@LoadBalancerClients 注解，并指定前面定义的配置类。

```
1 usage
@LoadBalancerClients(defaultConfiguration = {MyConfig.class})
@EnableFeignClients
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
    }

}
```

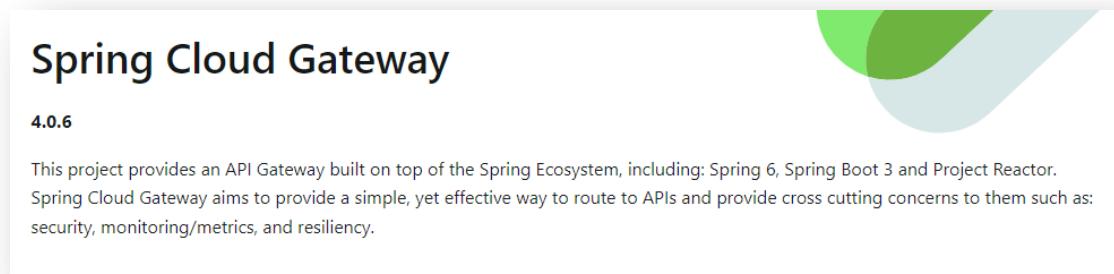
## 第5章 Spring Cloud Gateway 微服务网关

### 5.1 概述

#### 5.1.1 网关简介

网关是系统唯一对外的入口，介于客户端与服务器端之间，用于对请求进行鉴权、限流、路由、监控等功能。

#### 5.1.2 Gateway 简介



这个项目提供了一个建立在 Spring 生态系统之上的 API 网关，包括：Spring 6、Spring Boot 3 和 project Reactor。Spring Cloud Gateway 旨在提供一种简单而有效的方法来路由到 api，并为它们提供跨领域的关注点，例如：安全性、监控/度量和弹性。

#### 5.1.3 Reactor 简介

Reactor 是一种完全基于 Reactive Streams 规范的、全新的库。

##### (1) 响应式编程

响应式编程，Reactive Programming，是一种新的编程范式、编程思想。

响应式编程最早由 .Net 平台上的 Reactive eXtensions(Rx)库来实现。后来被迁移到了 Java 平台，产生了著名的 RxJava。在此之上，后来又产生了 Reactive Streams 规范。

## (2) Reactive Streams

Reactive Streams 是响应式编程的规范，定义了响应式编程的相关接口。只要符合该规范的库，就称为 Reactive 响应式编程库。

## (3) RxJava2

RxJava2 是一个响应式编程库，产生于 Reactive Streams 规范之后。但由于其是在 RxJava 基础之上进行的开发，所以在设计时不仅遵循了 Reactive Streams 规范，同时为了兼容 RxJava，使得 RxJava2 在使用时非常不方便。

## (4) Reactor

Reactor 是一种全新的响应式编程库，完全遵循 Reactive Streams 规范，又与 RxJava 没有任何关系，所以，其使用时非常方便，直观。

### 5.1.4 Zuul

Zuul 是 Netflix 的开源 API 网关。Zuul 是基于 Servlet 的，使用同步阻塞 IO，不支持长连接。Zuul 是 Spring Cloud 生态中的一员。

Zuul2.x 使用 Netty 实现了异步非阻塞 IO，支持长连接。但其未整合到 Spring Cloud。

### 5.1.5 重要概念

在 Spring Cloud Gateway 中有三个非常重要的概念：

#### (1) route 路由

路由是网关的最基本组成，由一个路由 id、一个目标地址 url，一组断言工厂及一组 filter 组成。若断言为 true，则请求将经由 filter 被路由到目标 url。

#### (2) predicate 断言

断言即一个条件判断，根据当前的 http 请求进行指定规则的匹配，比如说 http 请求头，请求时间等。只有当匹配上规则时，断言才为 true，此时请求才会被直接路由到目标地址（目标服务器），或先路由到某过滤器链，经过过滤器链的层层处理后，再路由到相应的目标地址（目标服务器）。

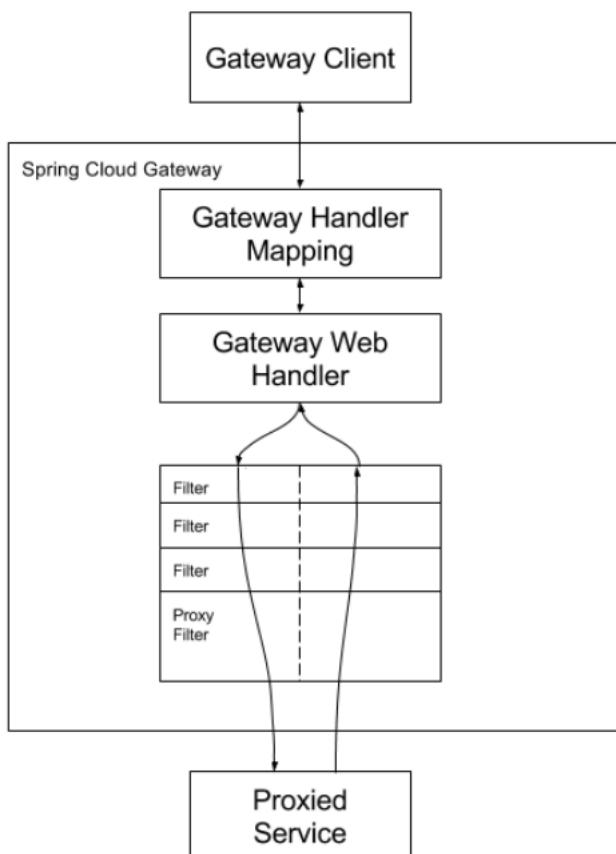
### (3) filter 过滤器

对请求进行处理的逻辑部分。当请求的断言为 `true` 时，会被路由到设置好的过滤器，以对请求或响应进行处理。例如，可以为请求添加一个请求参数，或对请求 URI 进行修改，或为响应添加 header 等。总之，就是对请求或响应进行处理。

#### 5.1.6 工作原理解析

### 3. How It Works

The following diagram provides a high-level overview of how Spring Cloud Gateway works:



## 5.2 牛刀小试-路由到百度

### 5.2.1 需求

用户访问 spring cloud gateway 应用，直接跳转到百度主页。

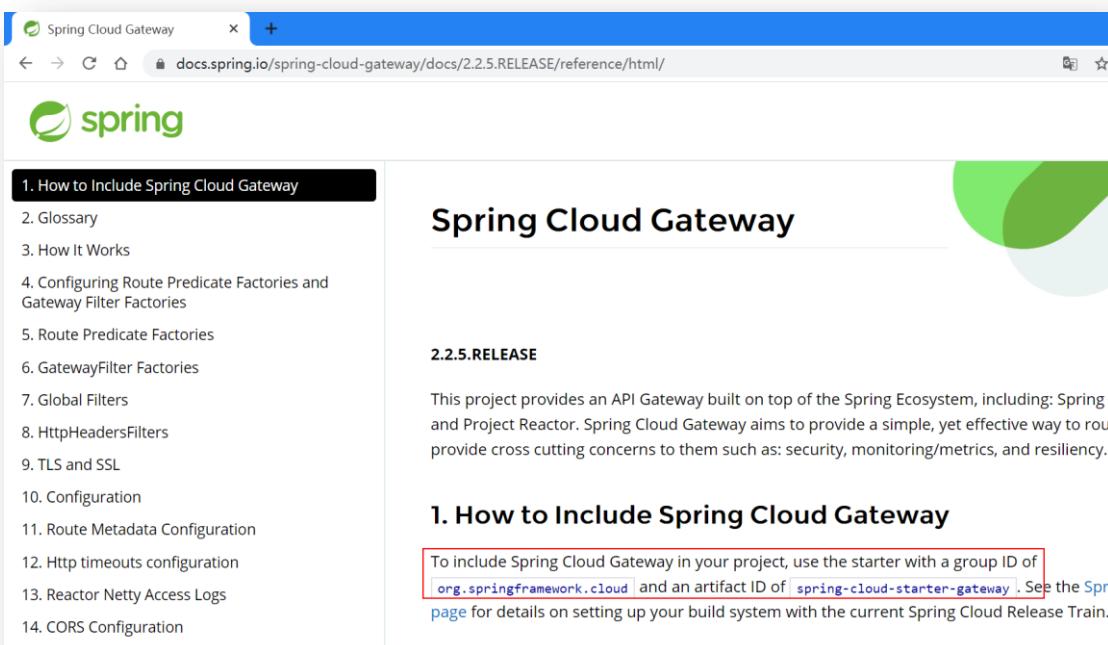
### 5.2.2 配置式路由 05-gateway-config-9000

#### (1) 创建工程

复制工程 02-consumer-nacos-8080，并重命名为 05-gateway-config-9000。

#### (2) 修改 pom 文件

#### A、官网



The screenshot shows a web browser displaying the official Spring Cloud Gateway documentation at [docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/](https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/). The page title is "Spring Cloud Gateway". On the left, there is a sidebar with a navigation menu:

- 1. How to Include Spring Cloud Gateway
- 2. Glossary
- 3. How It Works
- 4. Configuring Route Predicate Factories and Gateway Filter Factories
- 5. Route Predicate Factories
- 6. GatewayFilter Factories
- 7. Global Filters
- 8. HttpHeadersFilters
- 9. TLS and SSL
- 10. Configuration
- 11. Route Metadata Configuration
- 12. Http timeouts configuration
- 13. Reactor Netty Access Logs
- 14. CORS Configuration

The main content area starts with a section titled "2.2.5.RELEASE". Below it, there is a detailed description of the project's purpose and its integration with the Spring ecosystem. A prominent heading "1. How to Include Spring Cloud Gateway" is followed by a callout box containing instructions for adding the dependency to a Maven or Gradle build system.

## B、修改

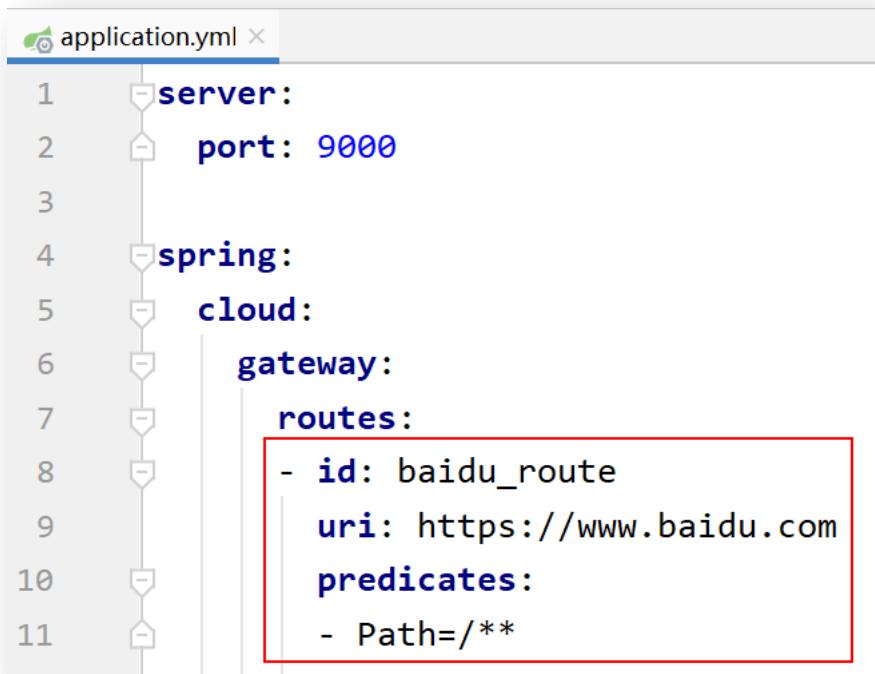
首先要保证其是一个 Spring Cloud Alibaba 工程，所以要保留<dependencyManagement>中的内容。然后将<dependencies>中的其它依赖全部删除，并添加 spring cloiud gateway 依赖。

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

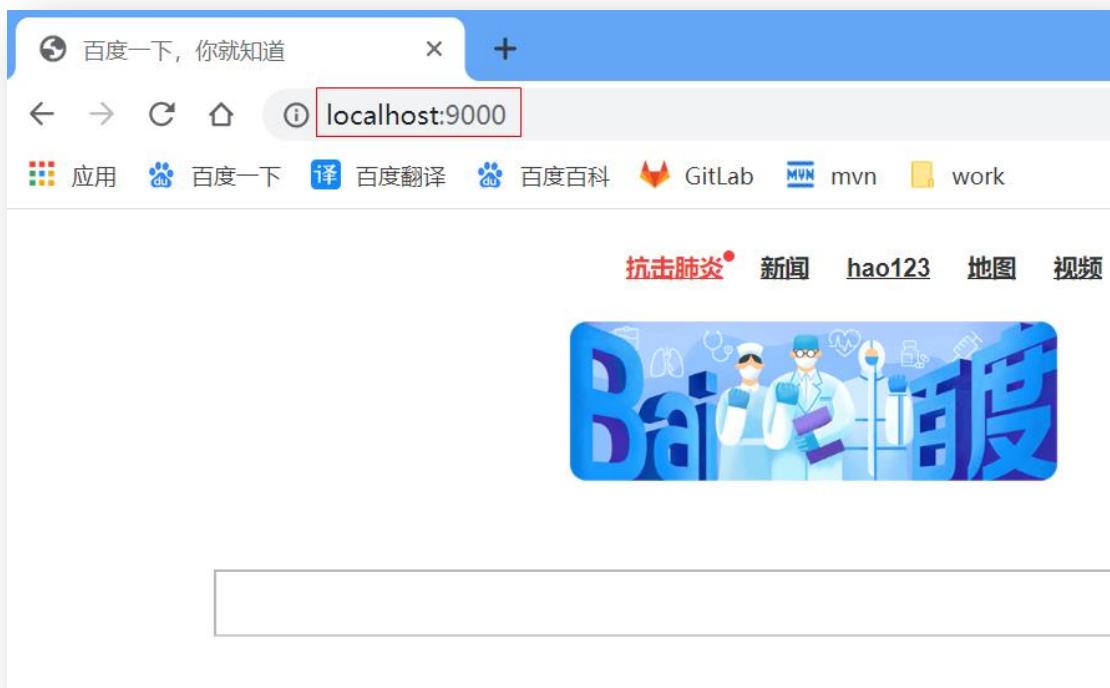
### (3) 修改代码类

代码中，只需一个普通的 Spring Boot 启动类，其它全部删除。

### (4) 修改配置文件



## (5) 访问应用



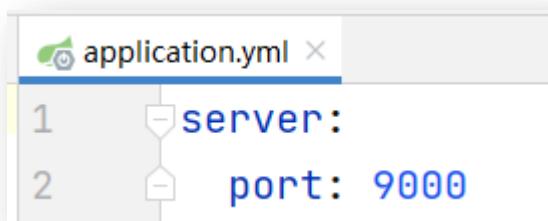
## 5.2.3 API 路由 05-gateway-api-9000

## (1) 创建工程

复制工程 05-gateway-config-9000，并重命名为 05-gateway-api-9000。

## (2) 修改配置文件

去掉原来配置的路由策略，仅剩端口号配置。



```
application.yml
1 server:
2   port: 9000
```

### (3) 修改启动类

在启动类中添加一个@Bean 方法，用于设置路由策略。

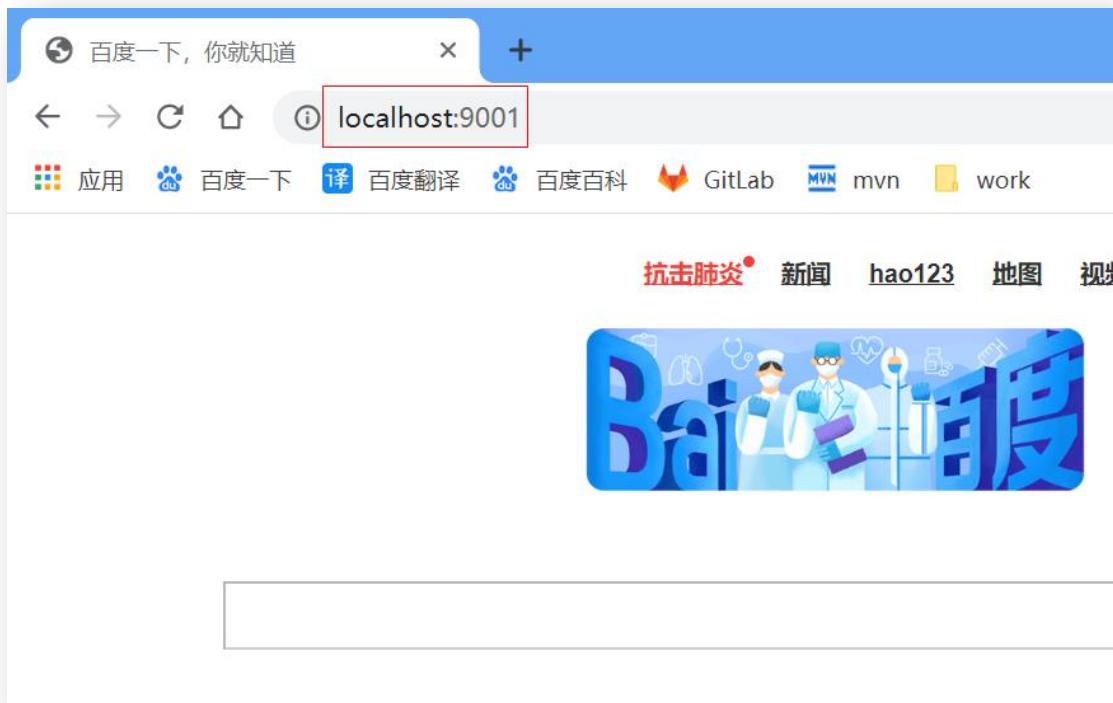
```
1 usage
@SpringBootApplication
public class GatewayApplication9000 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication9000.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("baidu_route",
                ps -> ps.path("/{**}")
                    .uri("https://www.baidu.com"))
            .build();
    }

}
```

## (4) 访问应用



## 5.3 路由断言工厂

### 5.3.1 简介

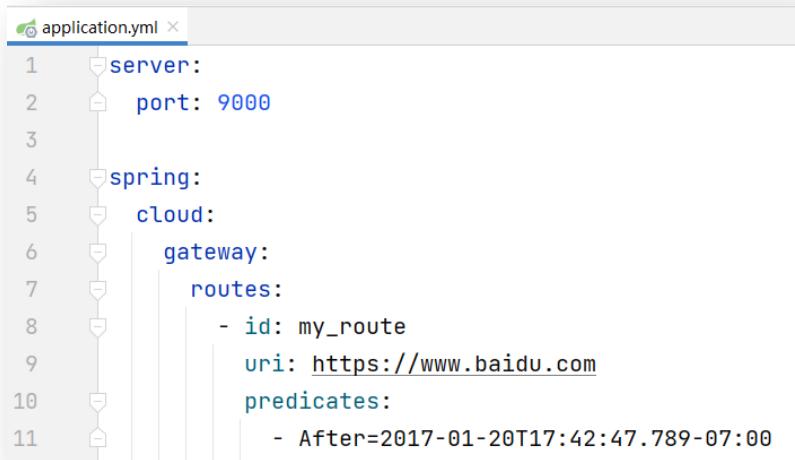
Spring Cloud Gateway 将路由匹配作为最基本的功能。而这个功能是通过路由断言工厂完成的。Spring Cloud Gateway 中包含了很多种内置的路由断言工厂。所有这些断言都可以匹配 HTTP 请求的不同属性，并且可以根据逻辑与状态，将多个路由断言工厂复合使用。

### 5.3.2 After 路由断言工厂

#### (1) 规则

该断言工厂的参数是一个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与该参数时间相比，若请求时间在参数时间之后，则匹配成功，断言为 true。

## (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://www.baidu.com
10          predicates:
11            - After=2017-01-20T17:42:47.789-07:00
```

### (3) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        ZonedDateTime dateTime = LocalDateTime.now().minusDays(5) // 当前时间减5天
            .atZone(ZoneId.systemDefault()); // 将系统的默认时区设置为当前时区

        return builder.routes()
            .route("after_route",
                ps -> ps.after(dateTime)
                    .uri("https://www.baidu.com"))
            .build();
    }

}
```

### (4) 运行效果

可以成功访问到页面。



### 5.3.3 Before 路由断言工厂

#### (1) 规则

该断言工厂的参数是一个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与该参数时间相比，若请求时间在参数时间之前，则匹配成功，断言为 true。

#### (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://www.baidu.com
10          predicates:
11            - Before=2017-01-20T17:42:47.789-07:00
```

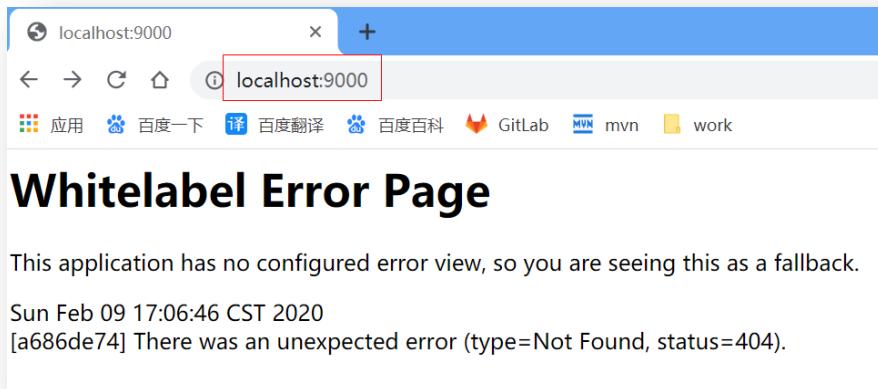
### (3) API 式启动类

```
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    ZonedDateTime dateTime = LocalDateTime.now().minusDays(5)
        .atZone(ZoneId.systemDefault());

    return builder.routes()
        .route("baidu_route", r -> r.before(dateTime)
            .uri("https://www.baidu.com"))
        .build();
}
```

### (4) 运行效果

无法访问到页面。



#### 5.3.4 Between 路由断言工厂

##### (1) 规则

该断言工厂的参数是两个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与这两个参数时间相比，若请求时间在这两个参数时间之间，则匹配成功，断言为 true。

## (2) 配置式配置文件



```
application.yml
1  server:
2    port: 9000
3
4  spring:
5    cloud:
6      gateway:
7        routes:
8          - id: my_route
9            uri: https://www.baidu.com
10           predicates:
11             - Between=2017-01-20T17:42:47.789-07:00,2025-01-20T17:42:47.789-07:00
```

### (3) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicatioinAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicatioinAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        ZonedDateTime minusTime = LocalDateTime.now().minusDays(5) // 当前时间减5天
            .atZone(ZoneId.systemDefault());
        ZonedDateTime plusTime = LocalDateTime.now().plusDays(3) // 当前时间加3天
            .atZone(ZoneId.systemDefault());

        return builder.routes()
            .route("between_route",
                ps -> ps.between(minusTime, plusTime)
                    .uri("https://www.baidu.com"))
            .build();
    }

}
```

### (4) 运行效果

可以成功访问到页面。

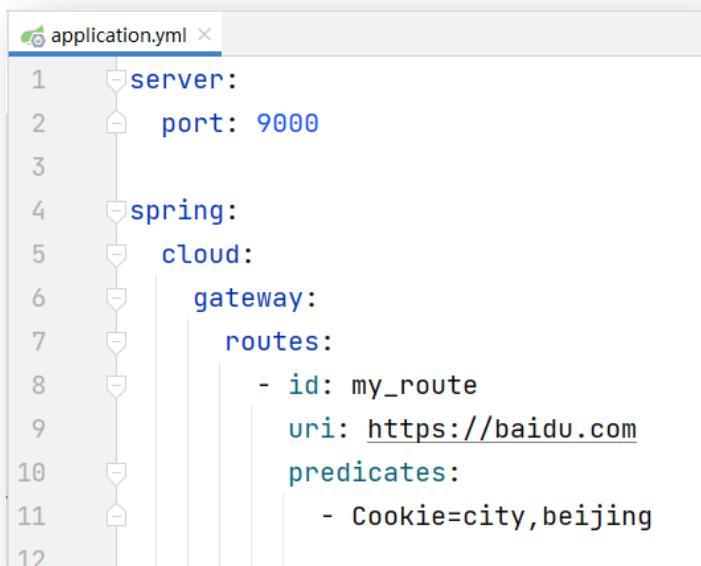


### 5.3.5 Cookie 路由断言工厂

#### (1) 规则

该断言工厂中包含两个参数，分别是 cookie 的 key 与 value。当请求中携带了指定 key 与 value 的 cookie 时，匹配成功，断言为 true。

#### (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Cookie=city,beijing
12
```

### (3) API 式启动类

```
@SpringBootApplication
public class GatewayApplicationAPI {

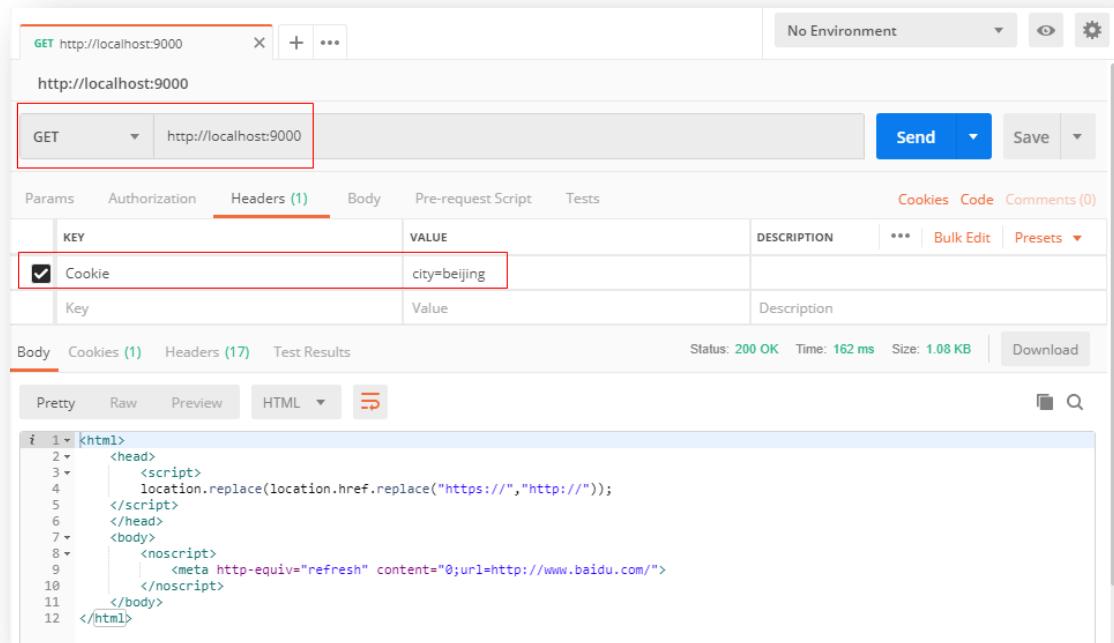
    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                  ps -> ps.cookie("city", "beijing")
                      .uri("https://www.baidu.com"))
            .build();
    }

}
```

### (4) 运行效果

这里使用 Postman 进行测试。在请求中添加指定的 Cookie，则可以成功访问到页面。但若没有设置 Cookie 或设置的 Cookie 的 key 与 value 与指定的不同，则无法访问到。



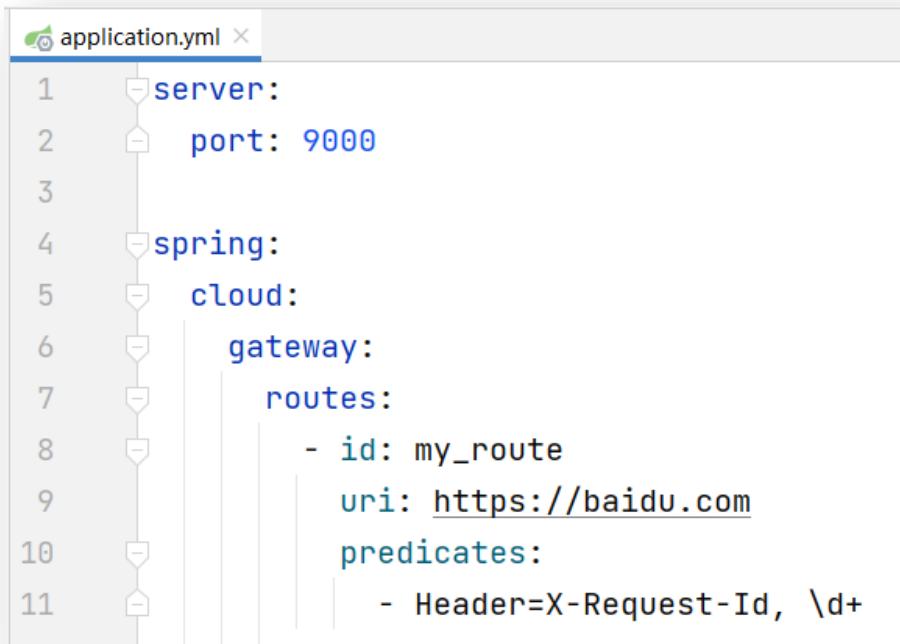
The screenshot shows the Postman application interface. At the top, there is a search bar with 'GET http://localhost:9000' and a 'No Environment' dropdown. Below the search bar, the URL 'http://localhost:9000' is entered again. The 'Headers (1)' tab is selected, showing a single header 'Cookie: city=beijing'. The 'Body' tab is also visible. At the bottom, the response code is shown as 'Status: 200 OK'.

### 5.3.6 Header 路由断言工厂

#### (1) 规则

该断言工厂中包含两个参数，分别是请求头 header 的 key 与 value。当请求中携带了指定 key 与 value 的 header 时，匹配成功，断言为 true。

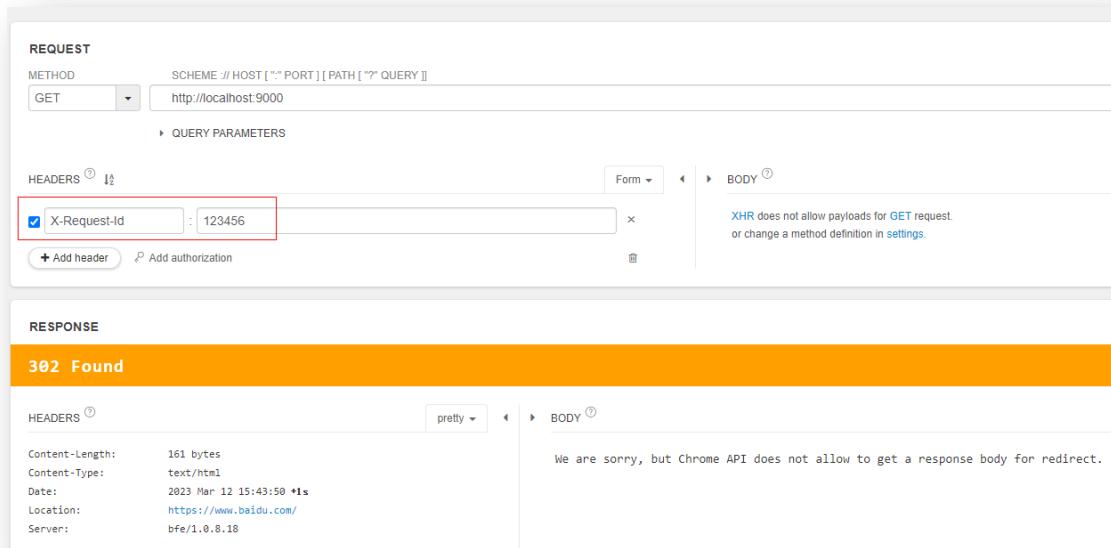
## (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Header=X-Request-Id, \d+
```

## (3) 配置式-运行效果

在请求中添加指定的 header，则可以成功访问到页面。只不过，由于新版 Chrome 不允许这种重定向，所以返回了 302Found。即定位到了指定的 URL 资源。

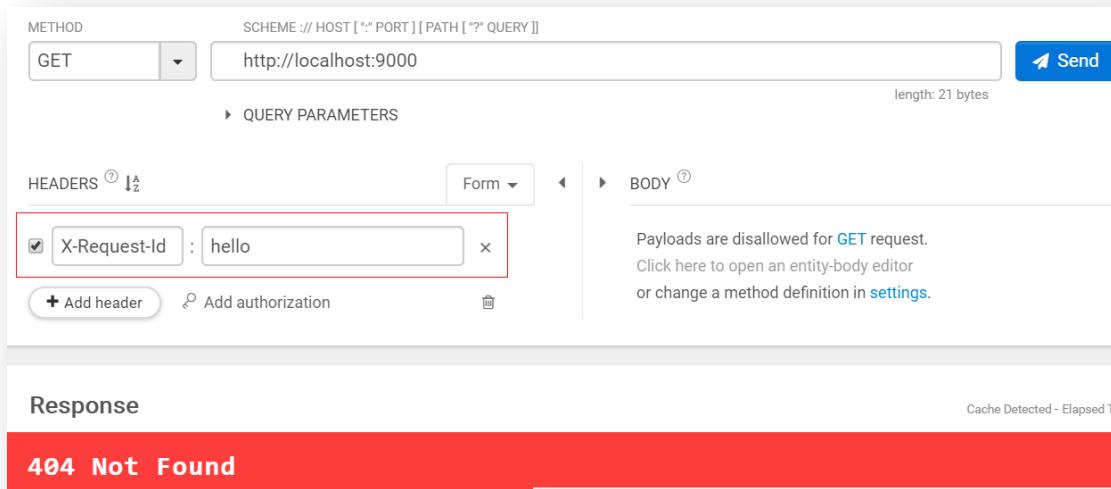


The screenshot shows a POSTMAN interface. In the REQUEST tab, a GET method is selected with the URL `http://localhost:9000`. Under Headers, there is a key-value pair `X-Request-Id : 123456`. In the RESPONSE tab, the status is **302 Found**. The Headers section shows:

Content-Length:	161 bytes
Content-Type:	text/html
Date:	2023 Mar 12 15:43:50 +0s
Location:	<a href="https://www.baidu.com/">https://www.baidu.com/</a>
Server:	bfe/1.0.8.18

The BODY section contains the message: "We are sorry, but Chrome API does not allow to get a response body for redirect."

但若没有设置 header 或设置的 header 的 key 与 value 与指定的不同，则无法访问到。



The screenshot shows a POSTMAN interface. In the REQUEST tab, a GET method is selected with the URL `http://localhost:9000`. Under Headers, there is a key-value pair `X-Request-Id : hello`. In the RESPONSE tab, the status is **404 Not Found**. The BODY section displays the message: "Payloads are disallowed for GET request. Click here to open an entity-body editor or change a method definition in settings."

## (4) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

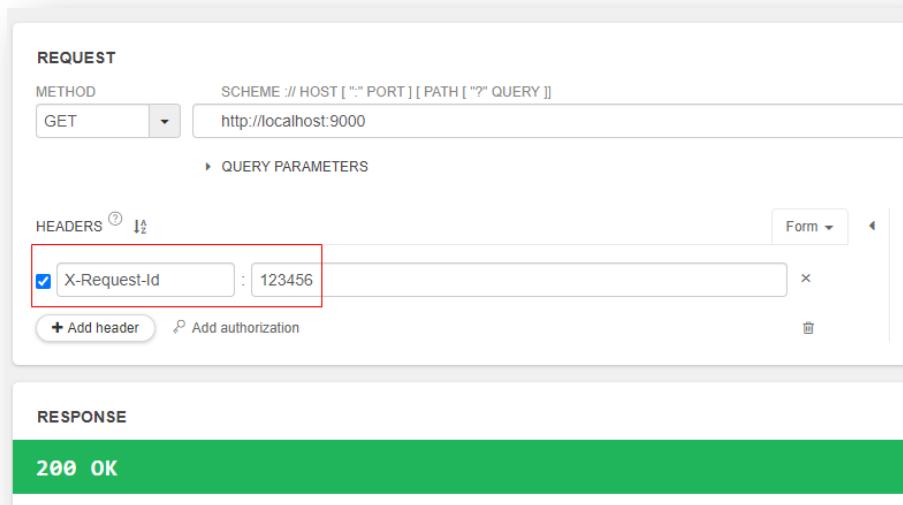
    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.header("X-Request-Id", "\d+")
                .uri("https://www.baidu.com"))
            .build();
    }

}
```

## (5) API 式-运行效果

在请求中添加指定的 header，则可以成功访问到页面。若 header 的值与指定值不匹配，则无法访问到指定 URI。



The screenshot shows a REST client interface with the following details:

- REQUEST** tab is selected.
- METHOD**: GET
- SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]**: http://localhost:9000
- HEADERS**:
  - X-Request-Id : 123456 (This header is highlighted with a red box.)
  - + Add header
  - + Add authorization
- RESPONSE** tab is selected.
- 200 OK** status message is displayed.

### 5.3.7 Host 路由断言工厂

#### (1) 规则

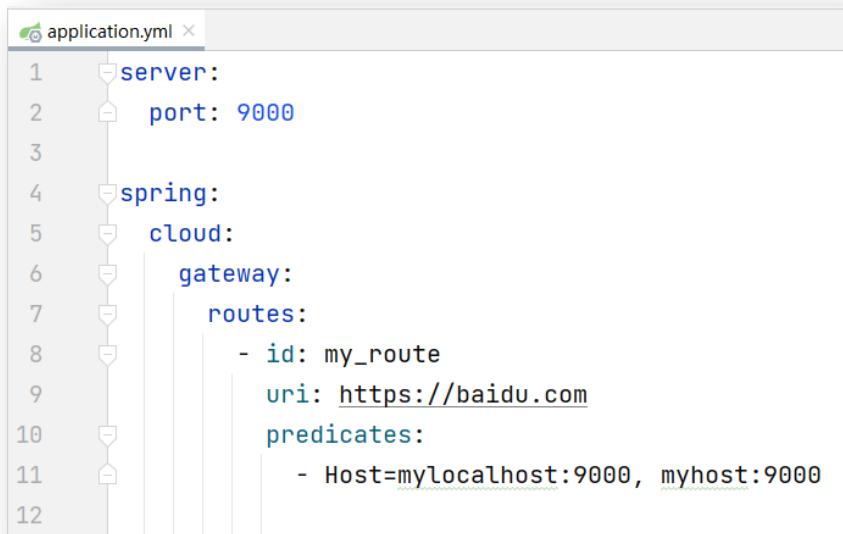
该断言工厂中包含的参数是请求头中的 Host 属性。当请求中携带了指定的 Host 属性值时，匹配成功，断言为 true。

#### (2) 修改 hosts 文件

修改 C:\Windows\System32\drivers\etc 中的 hosts 文件，为 127.0.0.1 这个 ip 指定多个主机名。例如，在该文件中添加如下内容：

```
127.0.0.1 mylocalhost
127.0.0.1 mylocal
127.0.0.1 myhost
```

#### (3) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Host=mylocalhost:9000, myhost:9000
```

## (4) API 式启动类

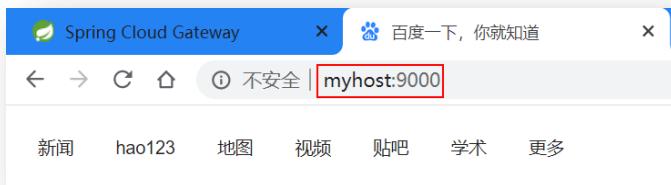
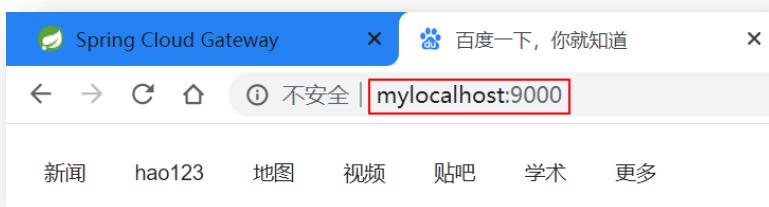
```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

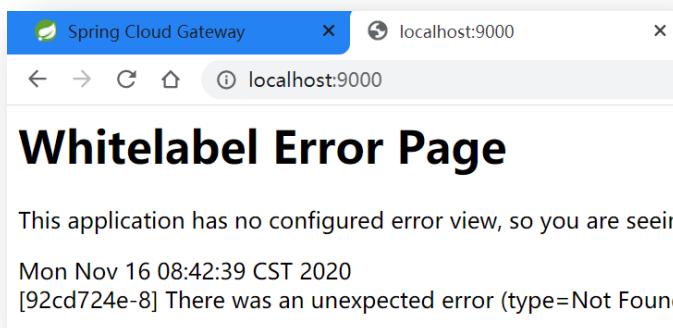
    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.host("mylocalhost:9000", "myhost:9000")
                    .uri("https://www.baidu.com"))
            .build();
    }
}
```

## (5) 运行效果

使用 mylocalhost 与 myhost 是可以成功访问到页面的。



但使用 `localhost`, 则无法访问到。

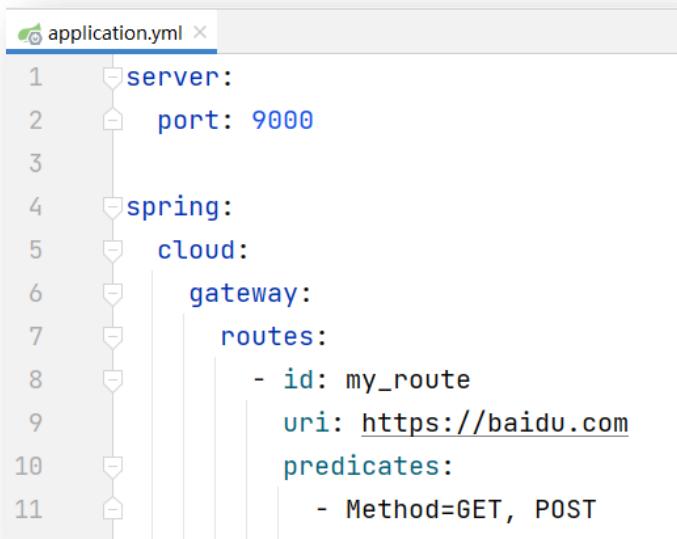


### 5.3.8 Method 路由断言工厂

#### (1) 规则

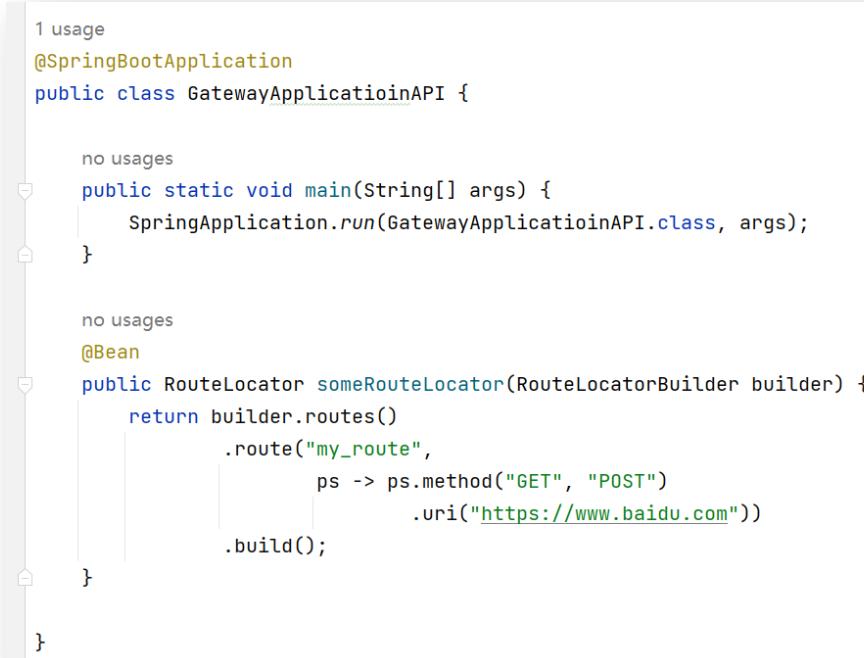
该断言工厂用于判断请求是否使用了指定的请求方法，是 `POST`, 还是 `GET` 等。当请求中使用了指定的请求方法时，匹配成功，断言为 `true`。

## (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Method=GET, POST
```

## (3) API 式启动类



```
usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.method("GET", "POST")
                    .uri("https://www.baidu.com"))
            .build();
    }
}
```

## (4) 运行效果

REQUEST

METHOD: GET SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]  
http://localhost:9000

QUERY PARAMETERS

HEADERS: Form

+ Add header P Add authorization

RESPONSE

302 Found

HEADERS: pretty

Content-Length: 161 bytes  
Content-Type: text/html  
Date: 2023 Mar 12 16:36:53  
Location: https://www.baidu.com/  
Server: bfe/1.0.8.18

BODY: We are sorry, but Chrome

REQUEST

METHOD: POST SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ]]  
http://localhost:9000

QUERY PARAMETERS

HEADERS: Form

+ Add header P Add authorization

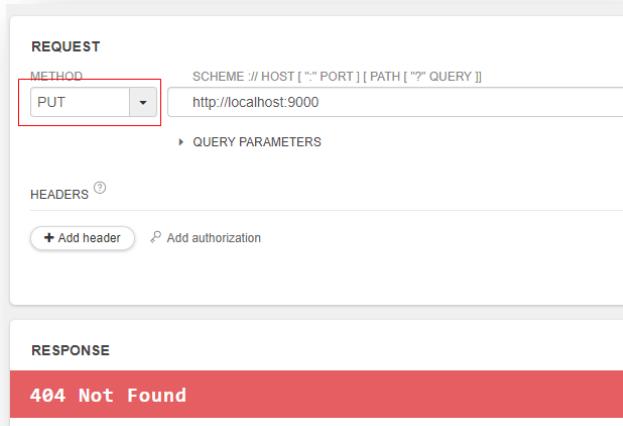
RESPONSE

302 Found

HEADERS: pretty

Content-Length: 161 bytes  
Content-Type: text/html  
Date: 2023 Mar 12 16:38:04  
Location: https://www.baidu.com/  
Server: bfe/1.0.8.18

BODY: We are sorry, but Chrome

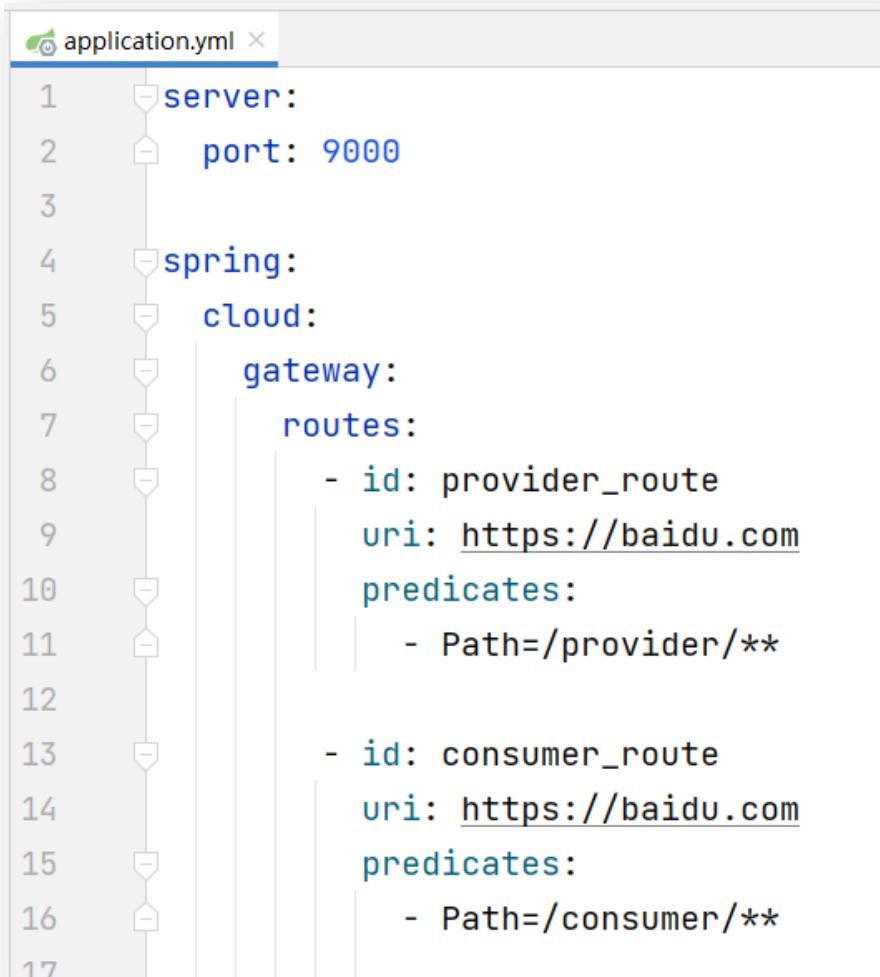


### 5.3.9 Path 路由断言工厂

#### (1) 规则

该断言工厂用于判断请求路径中是否包含指定的 uri。若包含，则匹配成功，断言为 true，此时会将该匹配上的 uri 拼接到要转向的目标 uri 的后面，形成一个统一的 uri。

## (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: provider_route
9           uri: https://baidu.com
10          predicates:
11            - Path=/provider/**
12
13         - id: consumer_route
14           uri: https://baidu.com
15          predicates:
16            - Path=/consumer/**
```

## (3) API 式启动类

直接修改路由方法。添加了两个路由策略。

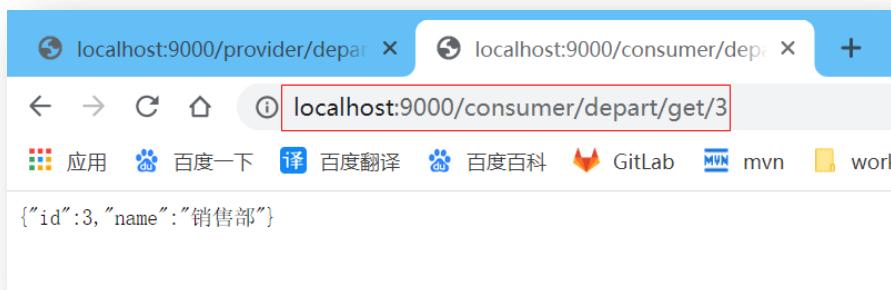
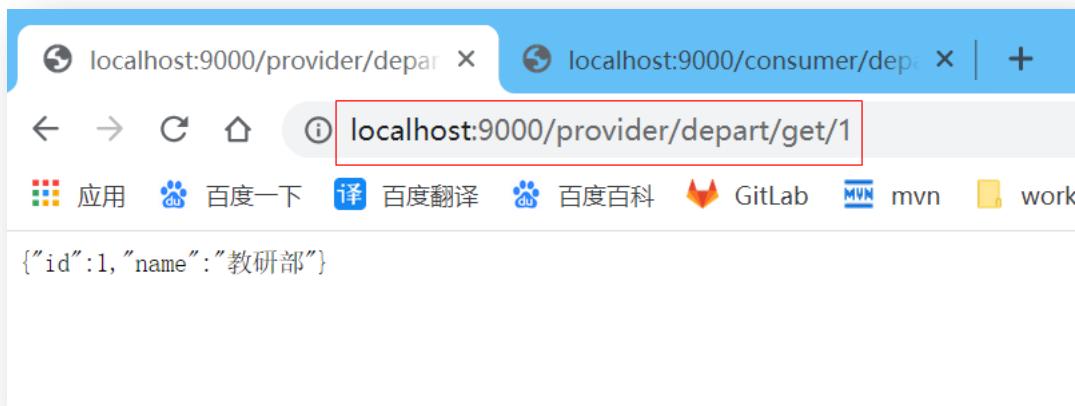
```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("provider_route",
                ps -> ps.path("/provider/**")
                    .uri("http://localhost:8081"))
            .route("consumer_route",
                ps -> ps.path("/consumer/**")
                    .uri("http://localhost:8080"))
            .build();
    }
}
```

#### (4) 配置式-运行效果

首先要启动 01-provider-8081 与 01-consumer-8080 工程（这两个工程与 Nacos 无关），然后再启动当前工程。



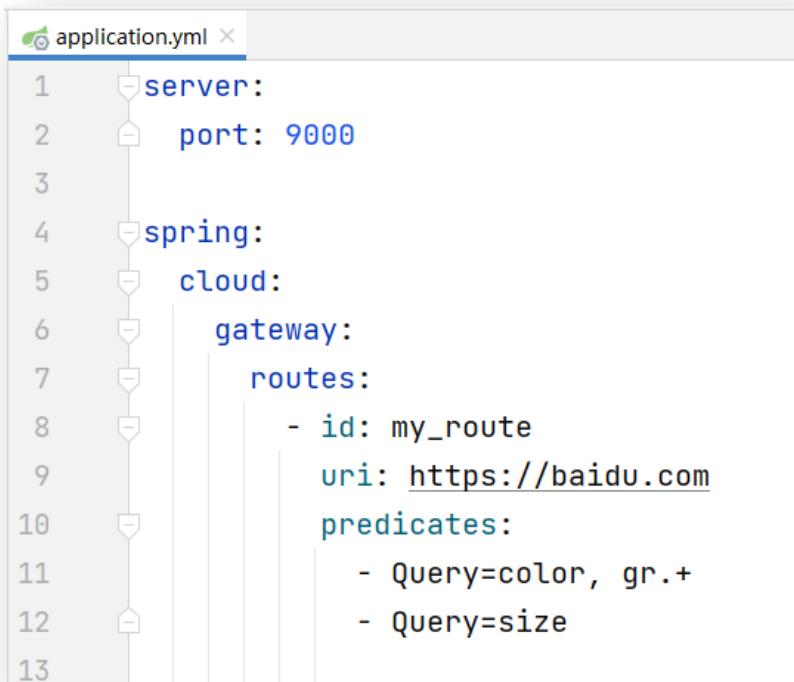
### 5.3.10 Query 路由断言工厂

#### (1) 规则

该断言工厂用于从请求中查找指定的请求参数。其可以只查看参数名称，也可以同时查看参数名与参数值。当请求中包含了指定的参数名，或名值对时，匹配成功，断言为 true。

#### (2) 配置式配置文件

以上两个 Query 断言的关系是“与”，即只有请求中同时包含了 color 与 size 参数，且 color 参数值必须是以 gr 开头的就可以访问到。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Query=color, gr.+
12            - Query=size
13
```

### (3) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

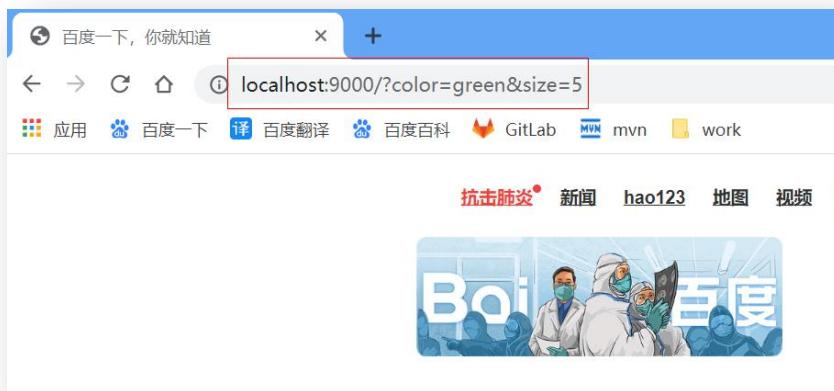
    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.query("color", "gr.+") BooleanSpec
                    .and() BooleanOpSpec
                    .query("size") BooleanSpec
                    .uri("https://baidu.com"))
            .build();
    }

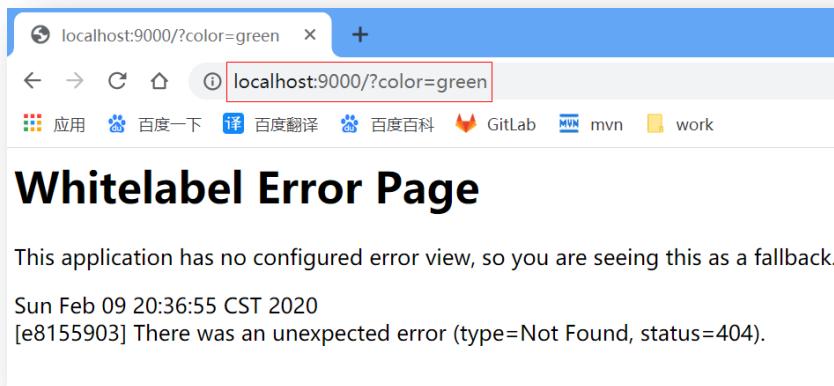
}
```

### (4) 运行效果

只有请求中同时包含了 color 与 size 参数，且 color 的值是以 gr 开头，就可以访问到。



若仅含一个指定的参数，则无法访问到。说明两个路由断言工厂间的关系是与的关系。



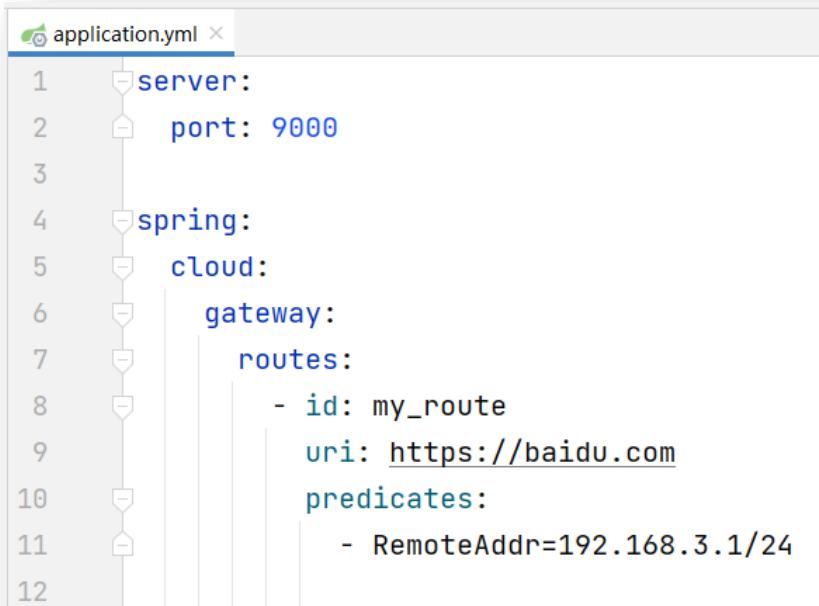
### 5.3.11 RemoteAddr 路由断言工厂

#### (1) 规则

该断言工厂用于判断提交请求的客户端 IP 地址是否在断言中指定的 IP 范围或 IP 列表中。若在，匹配成功，断言为 true。

#### (2) 配置配置文件

下面的路由指定提交请求的客户端 IP 地址若是 192.168.3 网段中的 IP，则是可访问到的。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - RemoteAddr=192.168.3.1/24
12
```

### (3) API 式启动类

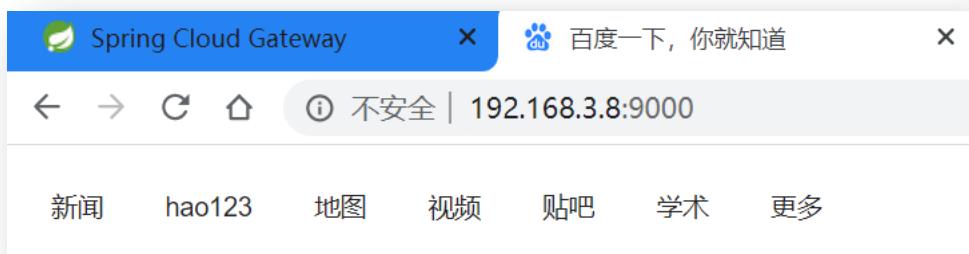
```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                  ps -> ps.remoteAddr("192.168.3.1/24")
                      .uri("https://baidu.com"))
            .build();
    }
}
```

### (4) 运行效果

由于当前浏览器提交请求的主机 IP 是 192.168.3.8，属于 192.168.3 网段，所以是可以访问到。



### 5.3.12 Weight 路由断言工厂

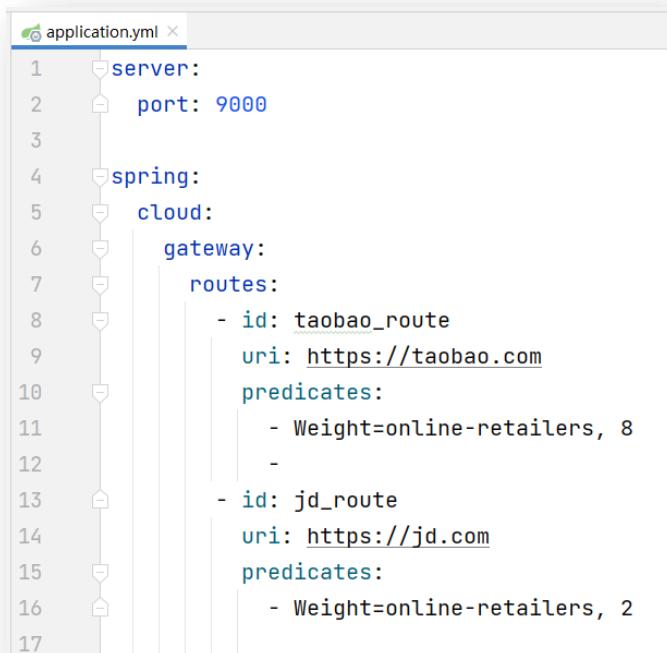
#### (1) 规则

该断言工厂用于实现对同一组中的不同 uri 实现指定权重的负载均衡。路由中包含两个参数，分别是用于表示组 group，与权重 weight。对于同一组中的多个 uri 地址，路由器会根据设置的权重，按比例将请求转发给相应的 uri。

#### (2) 配置式配置文件

下面的路由用于指定对两个电商平台 taobao.com 与 jd.com 的访问权重为 8:2。以下两个路由都属于电商 online-retailers 组。

在大量请求的前提下，对于相同的请求，转发到 tabao 的机率占 8 成，转发到 jd 的机率占 2 成。但这不是一个绝对准确的访问比例，大体差不多。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: taobao_route
9           uri: https://taobao.com
10          predicates:
11            - Weight=online-retailers, 8
12
13         - id: jd_route
14           uri: https://jd.com
15           predicates:
16             - Weight=online-retailers, 2
17
```

### (3) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

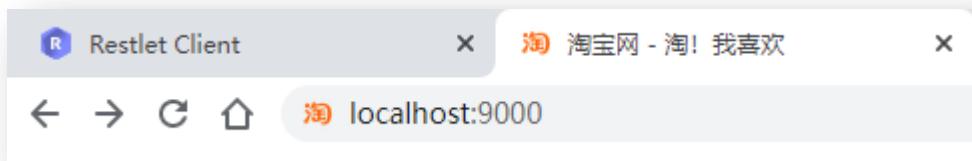
    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("taobao_route",
                ps -> ps.weight("online-retailers", 8)
                    .uri("https://taobao.com"))
            .route("jd_route",
                ps -> ps.weight("online-retailers", 2)
                    .uri("https://jd.com"))
            .build();
    }

}
```

### (4) 运行效果

在地址栏中多次提交 localhost:9000 的请示，可以看到大多数会跳转到 taobao，但偶尔也会跳转到 jd。



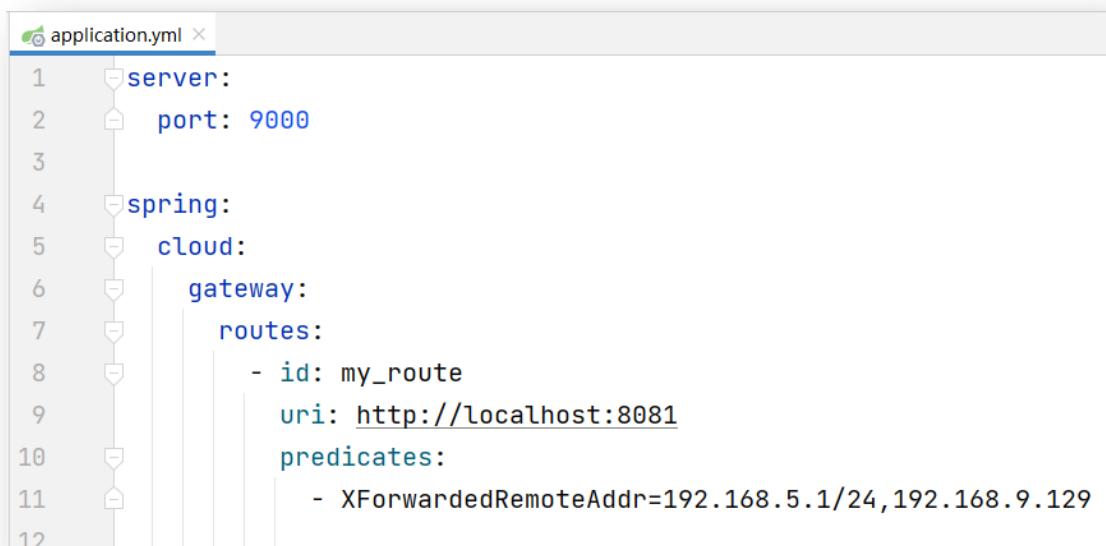
### 5.3.13 XForwardedRemoteAddr 路由断言工厂

#### (1) 规则

只要当前请求头中追加入 X-Forwarded-For 的 IP 出现在路由指定的 IP 列表中，则匹配成功，断言为 true。

#### (2) 配置式配置文件

以下断言的意思是，只要请求头 X-Forwarded-For 中追加的 IP 来自于网段 192.168.5.1/24 或就是具体的 IP 192.168.9.129，则断言为 true。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: http://localhost:8081
10          predicates:
11            - XForwardedRemoteAddr=192.168.5.1/24,192.168.9.129
```

### (3) API 式启动类

```
1 usage
@SpringBootApplication
public class GatewayApplicatioinAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicatioinAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.xForwardedRemoteAddr("192.168.5.1/24", "192.168.9.129")
                    .uri("http://localhost:8081"))
            .build();
    }

}
```

### (4) 运行效果

首先启动一个提供者工程，例如 01-provider-8081 工程。

**REQUEST**

METHOD : SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
 GET http://localhost:9000/provider/depart/get/3

▶ QUERY PARAMETERS

HEADERS ⑦ ↴ ↵

X-Forwarded-For : 192.168.5.15

+ Add header ⚒ Add authorization

**RESPONSE**

**200 OK**

**REQUEST**

METHOD : SCHEME :// HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
 GET http://localhost:9000/provider/depart/get/3 Send  
length: 43 bytes

▶ QUERY PARAMETERS

HEADERS ⑦ ↴ ↵

X-Forwarded-For : 192.168.99.8

+ Add header ⚒ Add authorization

**RESPONSE**

Cache Detected - Elapsed Time: 7ms

**404 Not Found**

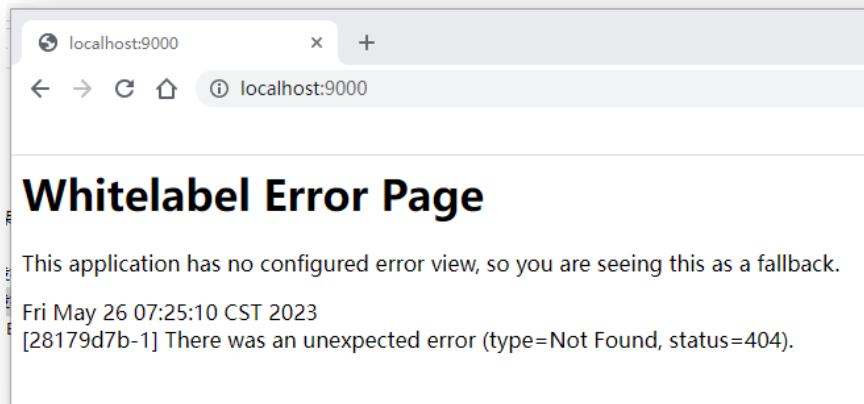
### 5.3.14 优先级

当一个应用中对于相同的路由断言工厂，分别通过配置式与 API 式都进行了设置，且设置的值不同，此时它们的关系有两种：

- 或的关系：例如，Header 路由断言工厂、Query 路由断言工厂
- 优先级关系：例如，After、Before、Between 路由断言工厂。此时配置式的优先级要高于 API 式的。

## 5.4 自定义异常处理器

当路由断言不匹配时会报出 404 异常信息，对用户非常不友好。所以需要自定义异常信息，以更好的形式给出用户提示。



### 5.4.1 自定义异常处理器类



```
① @Component
② @Order(-1)
③ public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

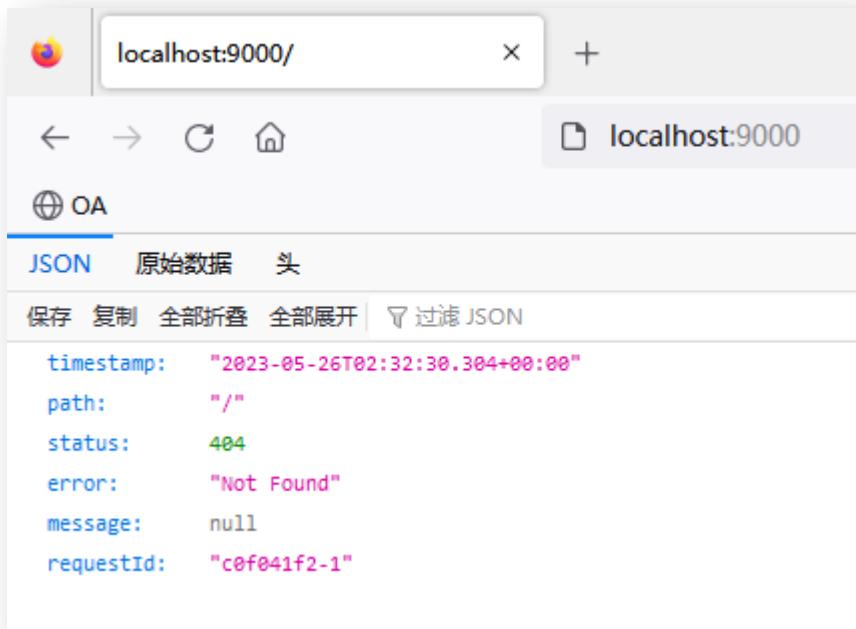
④     no usages
⑤     public CustomErrorWebExceptionHandler(ErrorAttributes errorAttributes,
⑥         ApplicationContext applicationContext,
⑦             ServerCodecConfigurer serverCodecConfigurer) {
⑧     super(errorAttributes, new WebProperties.Resources(), applicationContext);
⑨     super.setMessageWriters(serverCodecConfigurer.getWriters());
⑩     super.setMessageReaders(serverCodecConfigurer.getReaders());
⑪   }
⑫ }
```

```
    @Override
    protected RouterFunction<ServerResponse> getRoutingFunction(final ErrorAttributes ea) {
        return RouterFunctions.route(RequestPredicates.all(), this::renderErrorResponse);
    }

    1 usage
    private Mono<ServerResponse> renderErrorResponse(final ServerRequest request) {
        Map<String, Object> map = getErrorAttributes(request, ErrorAttributeOptions.defaults());

        return ServerResponse.status(HttpStatus.NOT_FOUND)
            .contentType(MediaType.APPLICATION_JSON)
            .body(BodyInserters.fromValue(map));
    }
}
```

## 5.4.2 页面展示



## 5.4.3 更换异常信息

如果想更换系统给出的异常信息，则可定义一个 `DefaultErrorAttributes` 的子类，覆盖其 `getErrorAttributes()` 方法。在该重写的方法中更换亲的信息。

```
@Component
public class CustomErrorAttributes extends DefaultErrorAttributes {
    @Override
    public Map<String, Object> getErrorAttributes(ServerRequest request,
                                                   ErrorAttributeOptions options) {
        Map<String, Object> map = new HashMap<>();
        map.put("status", HttpStatus.NOT_FOUND.value());
        map.put("message", "对不起，没有找到该资源");
        return map;
    }
}
```

#### 5.4.4 页面展示



#### 5.5 自定义路由断言工厂

当官方提供的路由断言工厂无法满足业务需求时，可以根据需求自定义路由断言工厂。下面通过两个例子来学习如何自定义路由断言工厂。

### 5.5.1 Auth 认证

#### (1) 需求

当请求头中携带有用户名与密码的 key-value 对，且其用户名与配置文件中 Auth 路由断言工厂中指定的 username 相同，密码中包含 Auth 路由断言工厂中指定的 password 时才能通过认证，允许访问系统。

#### (2) 定义 Factory

该类类名由两部分构成：后面必须是 RoutePredicateFactory，前面为功能前缀，该前缀将来要用在配置文件中。

```
1 usage
@Component
public class AuthRoutePredicateFactory extends
    AbstractRoutePredicateFactory<AuthRoutePredicateFactory.Config> {

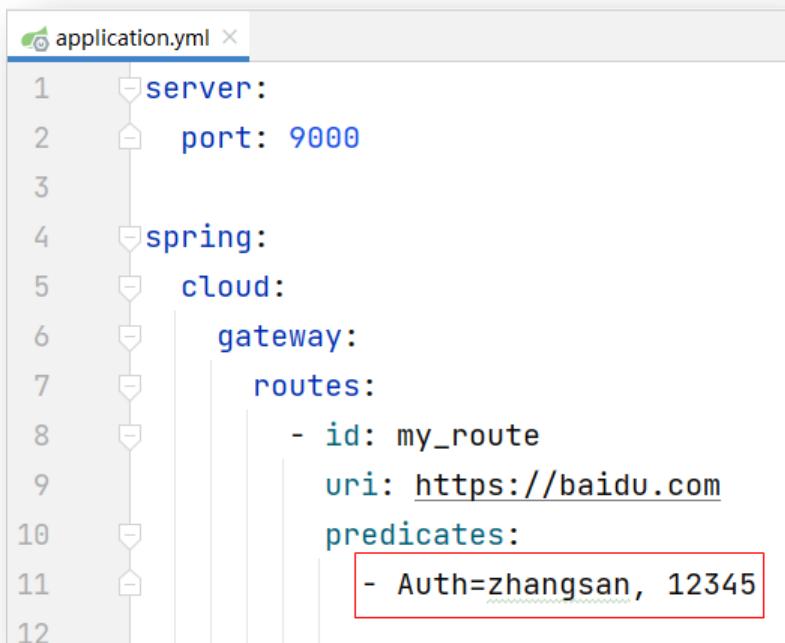
    3 usages
    @Data
    public static class Config {
        no usages
        private String username;
        no usages
        private String password;
    }

    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("username", "password");
    }
}
```

```
no usages
public AuthRoutePredicateFactory() {
    super(Config.class);
}

@Override
public Predicate<ServerWebExchange> apply(Config config) {
    return exchange -> {
        HttpHeaders headers = exchange.getRequest().getHeaders();
        List<String> pwds = headers.get(config.getUsername());
        if (pwds.contains(config.getPassword())) {
            return true;
        }
        return false;
    };
}
```

### (3) 修改配置文件



```
application.yml ×
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Auth=zhangsan, 12345
```

## (4) 访问

**REQUEST**

METHOD: GET    SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
http://localhost:9000    length: 21 bytes

QUERY PARAMETERS

HEADERS: Form  
zhangsan : 12345

+ Add header    Add authorization

**RESPONSE**

Cache Detected - Elapsed Time: 398ms

302 Found

HEADERS: pretty    BODY: pretty

**REQUEST**

METHOD: GET    SCHEME // HOST [ ":" PORT ] [ PATH [ "?" QUERY ] ]  
http://localhost:9000    length: 21 bytes

QUERY PARAMETERS

HEADERS: Form  
zhangsan : 12345678

+ Add header    Add authorization

**RESPONSE**

Cache Detected - Elapsed Time: 45ms

404 Not Found

HEADERS: pretty    BODY: pretty

```
content-len... 60 bytes
content-type... application/json
{
  "message": "对不起，没有找到该资源",
  "status": 404
}
```

## 5.5.2 Token 认证

## (1) 需求

当请求中携带有 token 请求参数，且参数值包含配置文件中 Token 路由断言工厂指

定的 token 值时才能通过认证，允许访问系统。

## (2) 定义 Factory

```
1 usage
@Component
public class TokenRoutePredicateFactory extends
    AbstractRoutePredicateFactory<TokenRoutePredicateFactory.Config> {

    3 usages
    @Data
    public static class Config {
        no usages
        private String token;
    }

    @Override
    public List<String> shortcutFieldOrder() {
        return Collections.singletonList("token");
    }

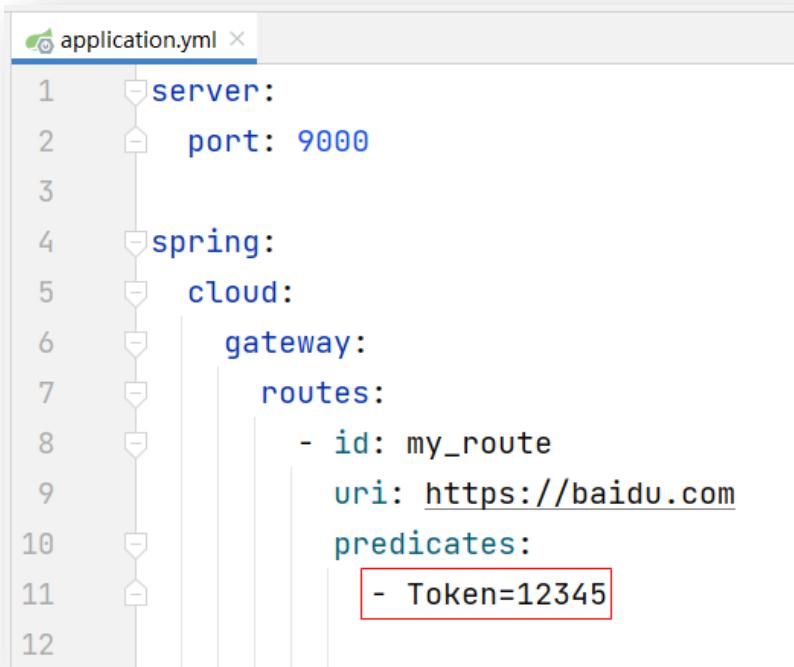
    no usages
    public TokenRoutePredicateFactory() {
        super(Config.class);
    }
}
```

```
@Override
public Predicate<ServerWebExchange> apply(Config config) {
    return exchange -> {
        MultiValueMap<String, String> map = exchange.getRequest().getQueryParams();

        List<String> values = map.get("token");

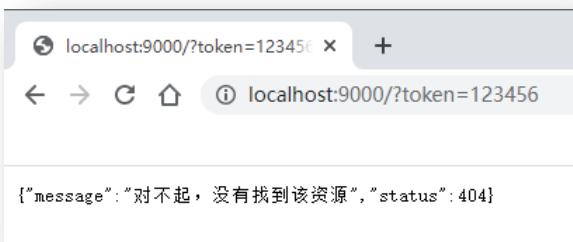
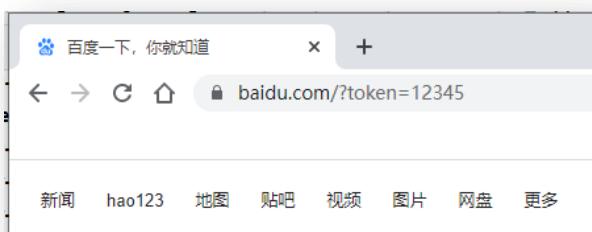
        if (values.contains(config.getToken())) {
            return true;
        }
        return false;
    };
}
```

## (3) 修改配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: https://baidu.com
10          predicates:
11            - Token=12345
```

## (4) 访问



## 5.6 网关过滤工厂

### 5.6.1 简介

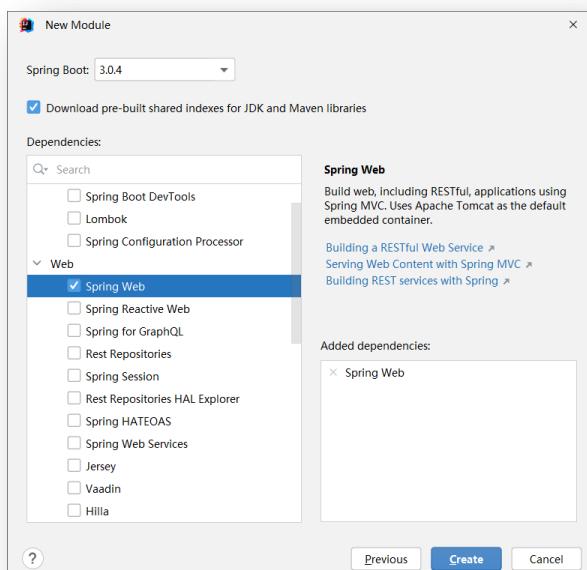
网关过滤器工厂 `GatewayFilterFactory` 允许以某种方式修改传入的 HTTP 请求或返回的 HTTP 响应。其作用域是某些特定路由。Spring Cloud Gateway 包括很多种内置的网关过滤器工厂。下面会学习较常用的几种。

### 5.6.2 定义一个工程

该工程将用于显示网关过滤器工厂对请求处理后的结果。

#### (1) 定义工程

定义一个普通的 Spring Boot 工程，命名为 05-showinfo-8080。该工程仅导入一个 Spring Web 依赖即可。



## (2) 定义处理器

```
@RestController
@RequestMapping("/info")
public class SomeController {

    // 暂时还没有定义任务处理器方法

}
```

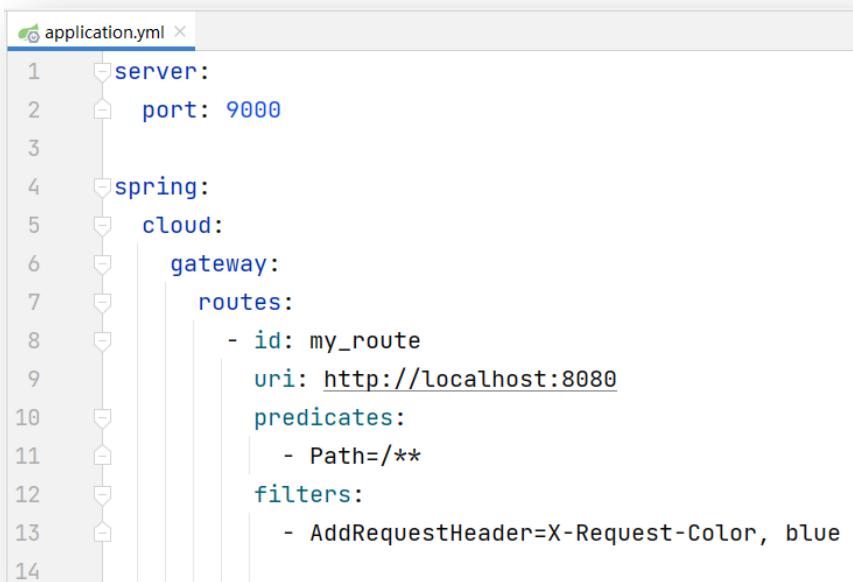
### 5.6.3 AddRequestHeader 过滤工厂

#### (1) 规则

该过滤器工厂会对匹配上的请求添加指定的 header。

#### (2) 配置式配置文件

这里修改的是 05-gateway-config-9000 工程中的配置文件。向请求添加了一个请求头信息 X-Request-Color=blue。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: http://localhost:8080
10          predicates:
11            - Path=/*
12          filters:
13            - AddRequestHeader=X-Request-Color, blue
14
```

### (3) 修改 showinfo 处理器

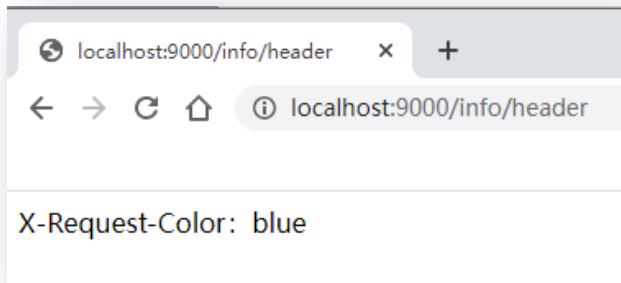
在 SomeController 处理器中添加如下处理器方法。



```
no usages
1 @RequestMapping("/info")
2 @RestController
3 public class SomeController {
4
5   no usages
6   @RequestMapping("/header")
7   public String headerHandle(HttpServletRequest request) {
8     String header = request.getHeader("X-Request-Color");
9     return "X-Request-Color: " + header;
10  }
11}
```

## (4) 运行效果

首先要启动 05-showinfo-8080 工程，然后再启动 05-gateway-config-9000 工程。然后就可以看到已经添加了指定的 header 信息了。



## (5) API 式启动类

这里修改的是 05-gateway-api-9000 工程中的启动类。

```
1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.path("/**") BooleanSpec
                    .filters(fs -> fs.addRequestHeader("X-Request-Color", "blue"))
                    .uri("http://localhost:8080"))
            .build();
    }
}
```

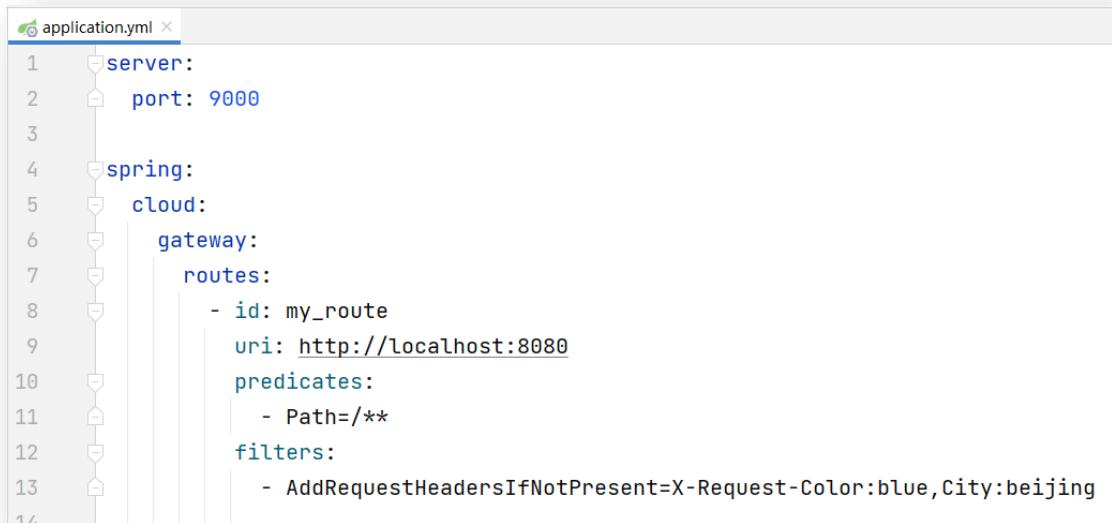
## 5.6.4 AddRequestHeadersIfNotPresent 过滤工厂

### (1) 规则

首先要清楚，请求中允许出现多个同名的请求头。即同一名称的请求头可以被多次添加到请求中。但有些场景下为了保证请求中对于某请求头只有一个值，就可以使用该过滤工厂。

该过滤器工厂会对匹配上的请求添加指定的 header，前提是该 header 在当前请求中尚未出现过。

### (2) 配置式配置文件



```
application.yml
1  server:
2    port: 9000
3
4  spring:
5    cloud:
6      gateway:
7        routes:
8          - id: my_route
9            uri: http://localhost:8080
10           predicates:
11             - Path=/**
12           filters:
13             - AddRequestHeadersIfNotPresent=X-Request-Color:blue,City:beijing
14
```

### (3) 修改 showinfo 处理器

在 SomeController 处理器中添加如下处理器方法，该方法遍历了当前请求中的所有 header。

```
no usages
@RequestMapping("allheaders")
public String allHeadersHandle(HttpServletRequest request) {
    // 获取请求中所有的请求头
    Enumeration<String> headerNames = request.getHeaderNames();

    // 遍历所有请求头
    StringBuilder sb = new StringBuilder();
    while (headerNames.hasMoreElements()) {
        String name = headerNames.nextElement();
        sb.append(name + ":");
        // 获取指定key的header的所有值
        Enumeration<String> headers = request.getHeaders(name);
        // 遍历该枚举实例
        while (headers.hasMoreElements()) {
            sb.append(headers.nextElement() + " ");
        }
        sb.append("<br>");
    }

    return sb.toString();
}
```

## (4) API 式启动类

```

1 usage
@SpringBootApplication
public class GatewayApplicationAPI {

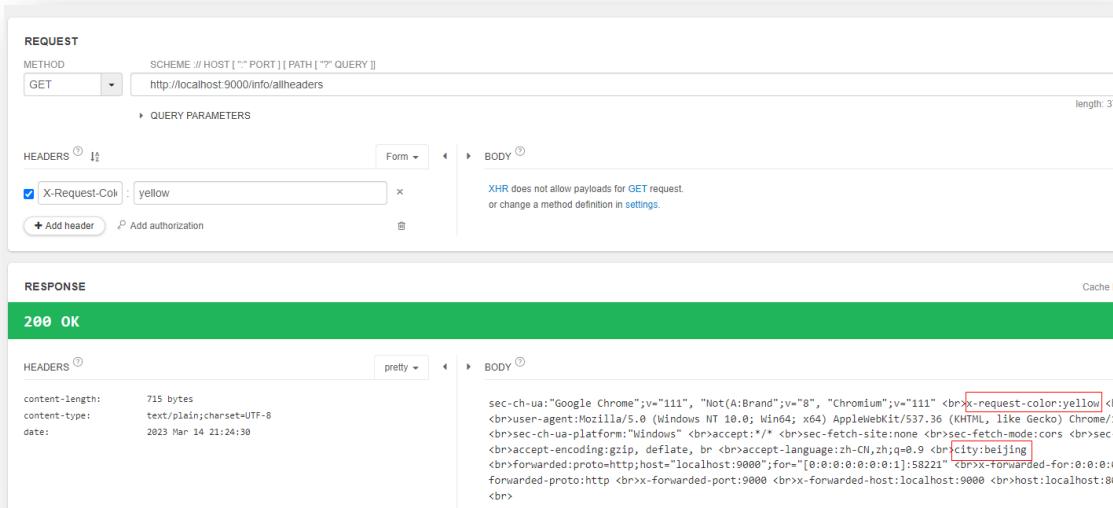
    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationAPI.class, args);
    }

    no usages
    @Bean
    public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("my_route",
                ps -> ps.path(@`/{**}`).BooleanSpec
                    .filters(fs -> fs.addRequestHeadersIfNotPresent("X-Request-Color:blue", "City:bj"))
                    .uri("http://localhost:8080"))
            .build();
    }
}

```

## (5) 运行效果

可以看到，指定要添加的 `x-request-color` header 没有被添加到请求，因为请求中已经存在了。但 `City` 却被写入到了请求，因为之前没有。



The screenshot shows a browser-based REST client interface. In the REQUEST section, a GET request is made to `http://localhost:9000/info/allheaders`. Under the HEADERS tab, there is a checked checkbox labeled `X-Request-Color` with the value `yellow`. In the RESPONSE section, the status is `200 OK`. The BODY tab displays the response content, which includes the following forwarded request headers:

```

sec-ch-ua:"Google Chrome";v="111", "Not(A:Brand";v="8", "Chromium";v="111" <br><b>x-request-color:yellow</b>
sec-ch-ua-platform:"Windows" <br>accept:"/*" <br>sec-fetch-site:none <br>sec-fetch-mode:cors <br>sec-
accept-encoding:gzip, deflate, br <br>accept-language:zh-CN,zh;q=0.9 <br>[city:beijing]
forwarded:proto=http;host="localhost:9000";for="[0:0:0:0:0:1]:58221" <br><br>xx-forwarded-for:[0:0:0:
forwarded-proto:http <br>x-forwarded-port:9000 <br>x-forwarded-host:localhost:9000 <br>host:localhost:8080
<br>

```

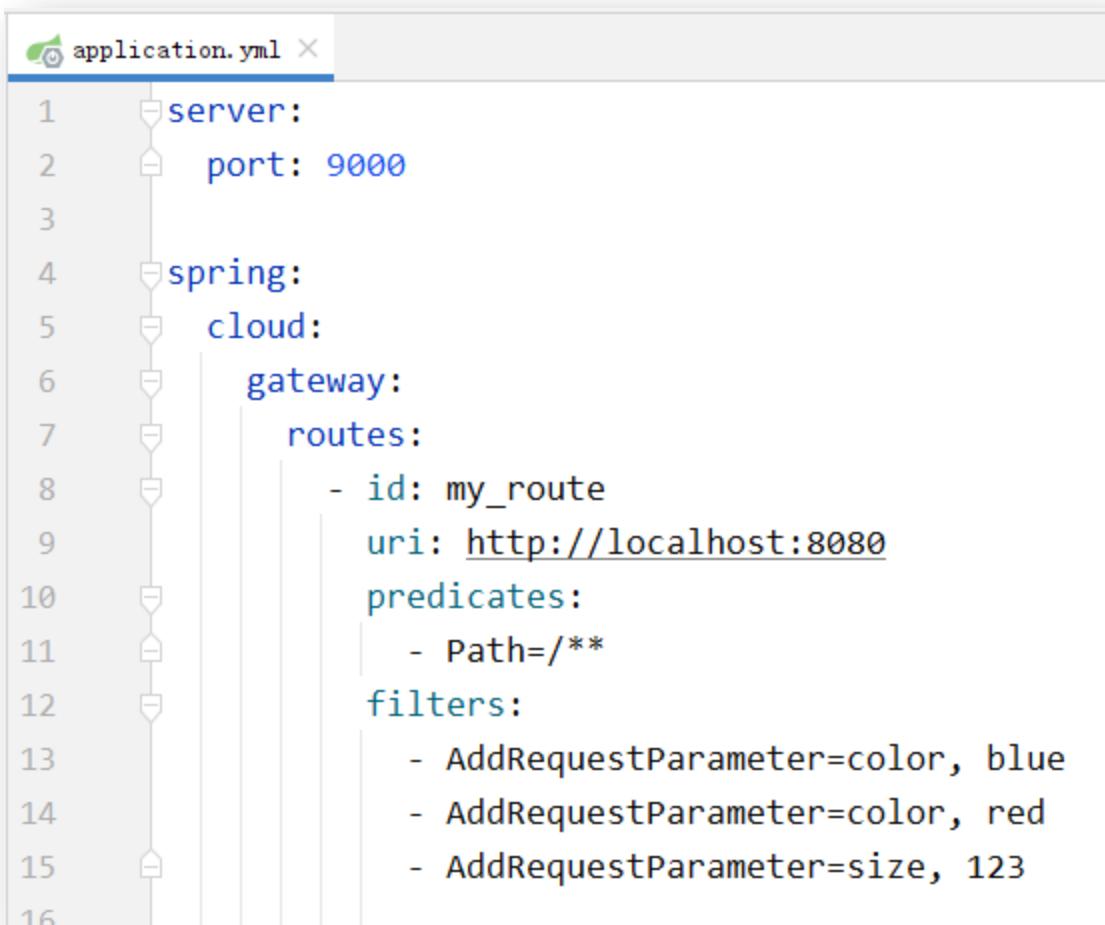
## 5.6.5 AddRequestParameter 过滤工厂

### (1) 规则

该过滤器工厂会对匹配上的请求添加指定的请求参数。

### (2) 配置式配置文件

向请求添加了一个两个 param，其中 color 还有两个值。



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: http://localhost:8080
10          predicates:
11            - Path=/**
12          filters:
13            - AddRequestParameter=color, blue
14            - AddRequestParameter=color, red
15            - AddRequestParameter=size, 123
16
```

### (3) 修改 showinfo 处理器

在处理器中添加如下处理器方法。

```
no usages
@GetMapping("/params")
public String paramsHandle(HttpServletRequest request) {
    StringBuilder sb = new StringBuilder();
    // 获取请求中的所有请求参数及其值
    Map<String, String[]> map = request.getParameterMap();
    // 遍历所有请求参数
    for (String key : map.keySet()) {
        sb.append(key + ": ");
        // 遍历当前请求参数的所有值
        for (String value : map.get(key)) {
            sb.append(value + " ");
        }
        sb.append("<br>");
    }
    return sb.toString();
}
```

## (4) API 式启动类

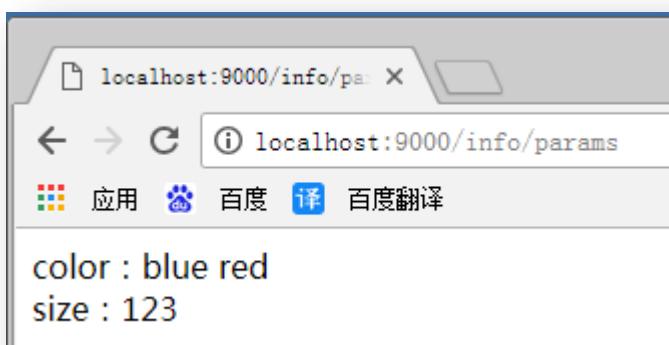
```
1 usage
@SpringBootApplication
public class GatewayAPIApplication {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayAPIApplication.class, args);
    }

    no usages
    @Bean
    public RouteLocator routeLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route(id: "my_route",
                ps -> ps.path(/\**/) BooleanSpec
                    .filters(fs -> fs.addRequestParameter(param: "color", value: "red")
                        .addRequestParameter(param: "color", value: "blue")
                        .addRequestParameter(param: "size", value: "123")) Uri
                .build());
    }
}
```

## (5) 运行效果

首先要启动 09-showheader-8080 工程，然后再启动 09-gateway-config-9000 工程。  
请求 URL 中并没有携带请求参数，但最终显示的确实有了请求参数。

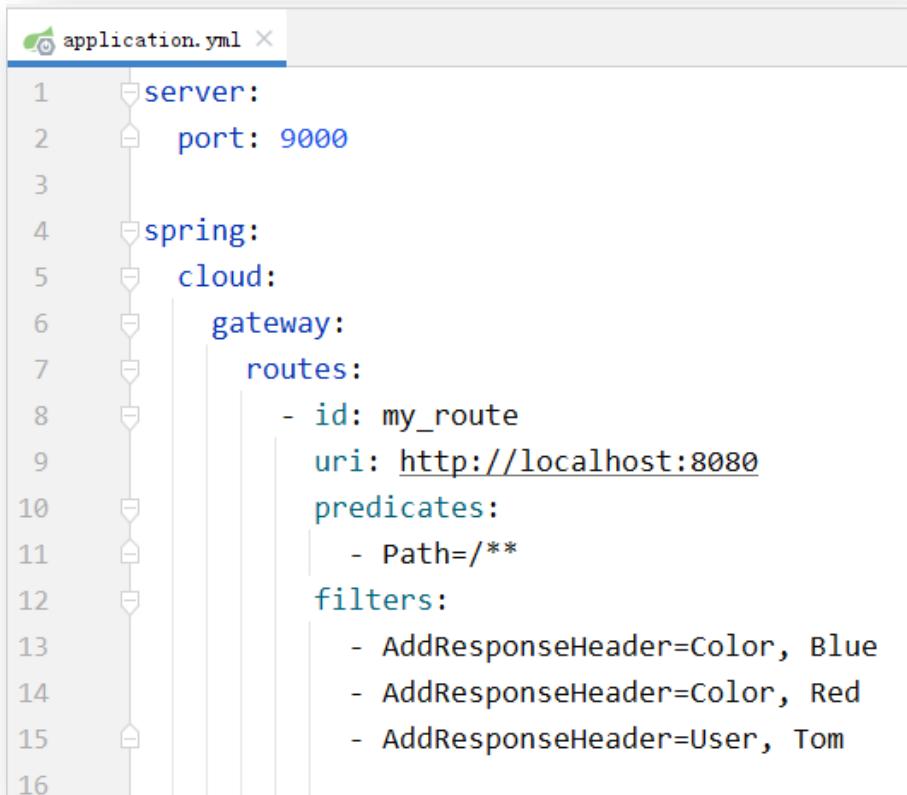


## 5.6.6 AddResponseHeader 过滤工厂

### (1) 规则

该过滤器工厂会给从网关返回的响应添加上指定的 header。

### (2) 配置式配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: http://localhost:8080
10          predicates:
11            - Path=/*
12          filters:
13            - AddResponseHeader=Color, Blue
14            - AddResponseHeader=Color, Red
15            - AddResponseHeader=User, Tom
16
```

### (3) API 式启动类

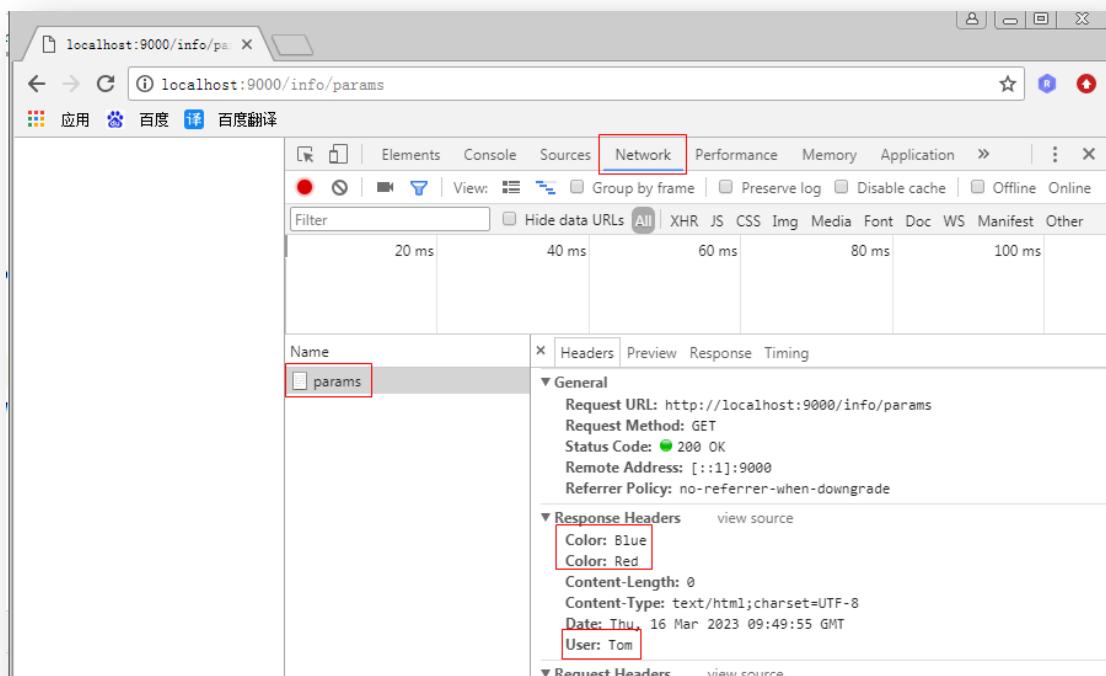
```
 1 usage
@SpringBootApplication
public class GatewayAPIApplication {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayAPIApplication.class, args);
    }

    no usages
    @Bean
    public RouteLocator routelocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route( id: "my_route",
                    ps -> ps.path(/***) BooleanSpec
                        .filters(fs -> fs.addResponseHeader( headerName: "Color", headerValue: "Red")
                                .addResponseHeader( headerName: "Color", headerValue: "Blue")
                                .addResponseHeader( headerName: "User", headerValue: "Tom"))
                        .uri("http://localhost:8080"))
            .build();
    }

}
```

### (4) 运行效果



## 5.6.7 CircuitBreaker 过滤工厂

### (1) 规则

该过滤器工厂完成网关层的服务熔断与降级。

### (2) 添加依赖

05-gateway-filter-config-9000 与 05-gateway-filter-api-9001 两个工程中均需要添加上 resilience4j 依赖。

```
<!--resilience4j 依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>

    <artifactId>spring-cloud-starter-circuitbreaker-reactor-resilience4j</artifactId>
</dependency>
```

### (3) 需求

我们下面的例子实现的需求是，浏览器访问 05-showinfo-8080 的处理器不成功，从而服务降级到 Gateway 工程中定义的降级处理器。

通过 05-showinfo-8080 工程不启动来模拟宕机场景。

### (4) 定义降级处理器

05-gateway-config-9000 与 05-gateway-api-9000 两个工程中均需要添加上该降级处理器。

```
no usages
@RestController
public class FallbackController {

    no usages
    @RequestMapping("/fb")
    public String fallbackHandle() {
        return "This is the Gateway Fallback";
    }
}
```

## (5) 配置式配置文件

```
spring:
  cloud:
    gateway:
      routes:
        - id: my_route
          uri: http://localhost:8081
          predicates:
            - Path=/info/**
          filters:
            - name: CircuitBreaker
              args:
                name: myCircuitBreaker
                fallbackUri: forward:/fb
```

- 第一个 name 属性用于指定要使用的 filter 类型，不能随意写。
- args 是对指定 filter 的配置参数。
- 第二个 name 属性用于指定当前所使用断路器的名称，可以随意。

- fallbackUri 属性用于指定发生断路后要提交的降级 URI。

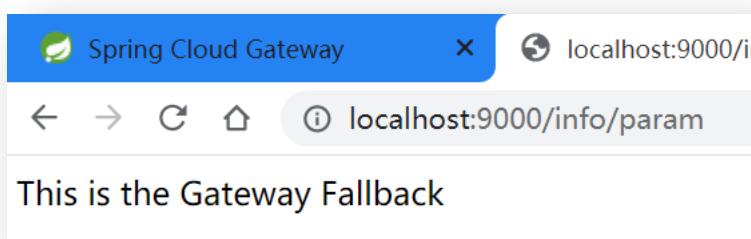
## (6) API 式启动类

直接修改路由方法。

```
no usages
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("my_route",
            ps -> ps.path("/info/**") BooleanSpec
                .filters(fs -> fs.circuitBreaker(config -> {
                    config.setName("myCircuitBreaker");
                    config.setFallbackUri("forward:/fb");
                })) UriSpec
                .uri("http://localhost:8081"))
        .build();
}
```

## (7) 运行效果

注意仅需启动当前 Gateway 工程即可，不用启动其它任何工程。这样用于模拟 localhost:8080 主机宕机，无法提供服务的情况。



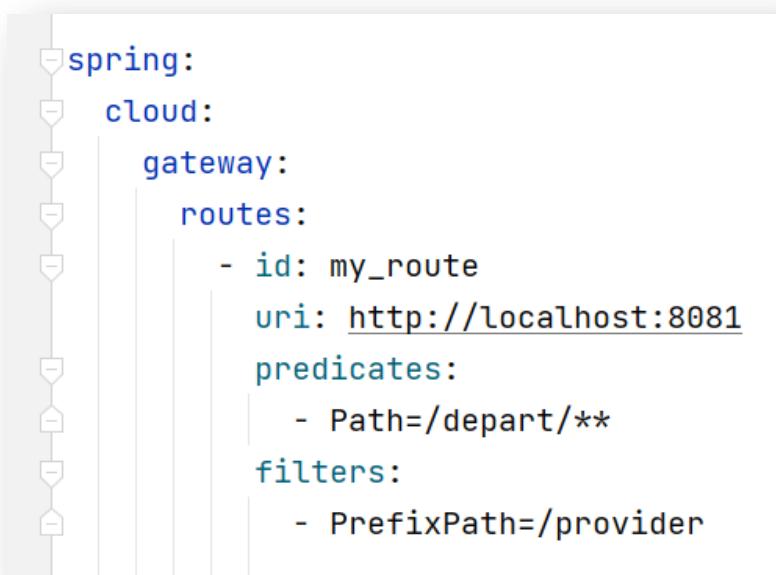
## 5.6.8 PrefixPath 过滤工厂

### (1) 规则

该过滤器工厂会为指定的 URI 自动添加上一个指定的 URI 前缀。

### (2) 配置式配置文件

为 URI 为 /depart 开头的请求添加上 /provider 前缀。



### (3) API 式启动类

直接修改路由方法。

```
no usages
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("my_route",
            ps -> ps.path("/depart/**") BooleanSpec
                .filters(fs -> fs.prefixPath("/provider"))
                .uri("http://localhost:8081"))
        .build();
}
```

#### (4) 运行效果



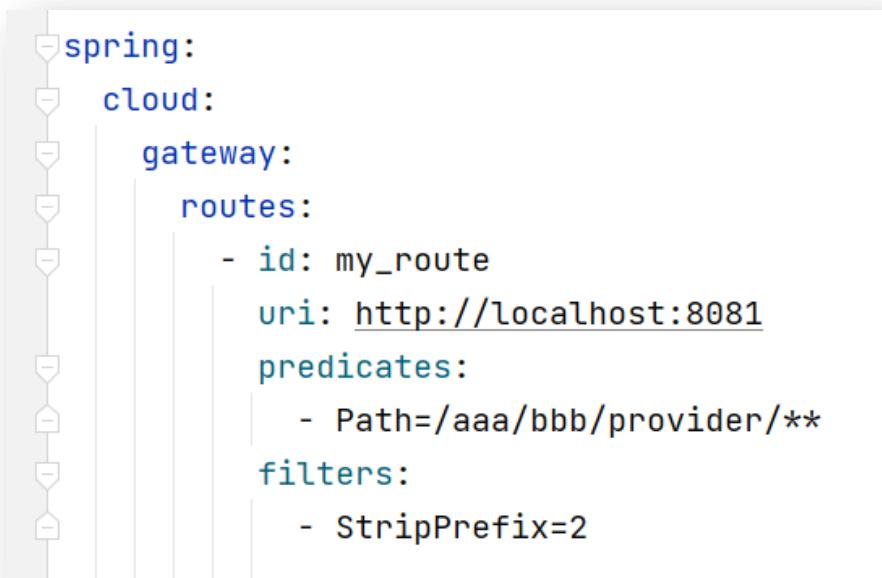
### 5.6.9 StripPrefix 过滤工厂

#### (1) 规则

该过滤器工厂会为指定的 URI 去掉指定长度的前缀。

#### (2) 配置式配置文件

指定去掉 2 个前缀。



### (3) API 式启动类

直接修改路由方法。



### (4) 运行效果

我们可以看到端口号 9000 后又跟了/aaa/bbb，但其仍是可以访问到相应的处理器。因为过滤器自动为这个请求 URL 去掉了 2 个前缀，即/aaa/bbb，使请求路径变为了 http://localhost:9000 /provider/depart/list。



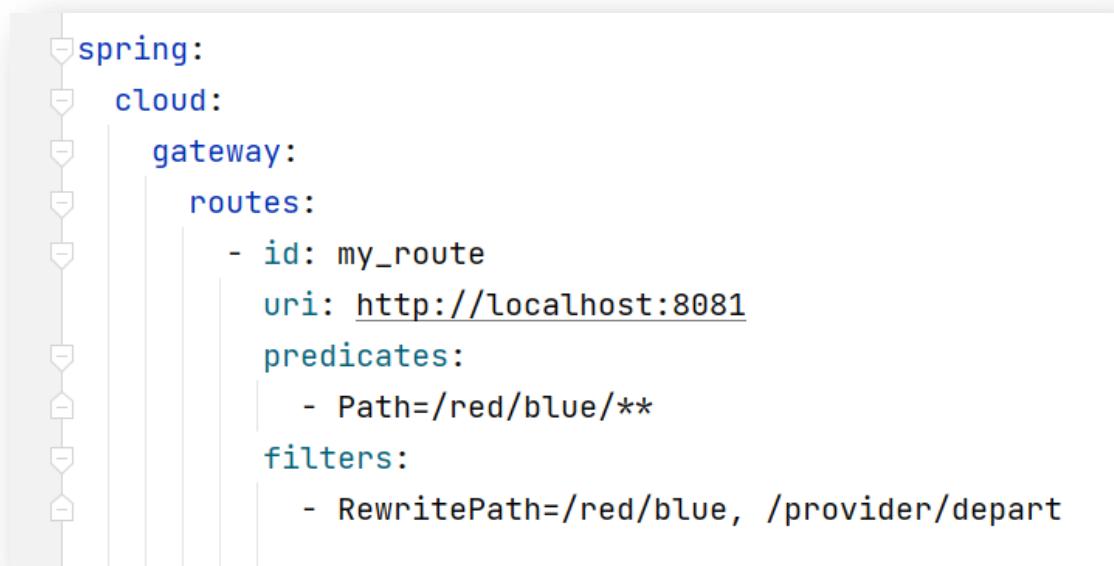
### 5.6.10 RewritePath 过滤工厂

#### (1) 规则

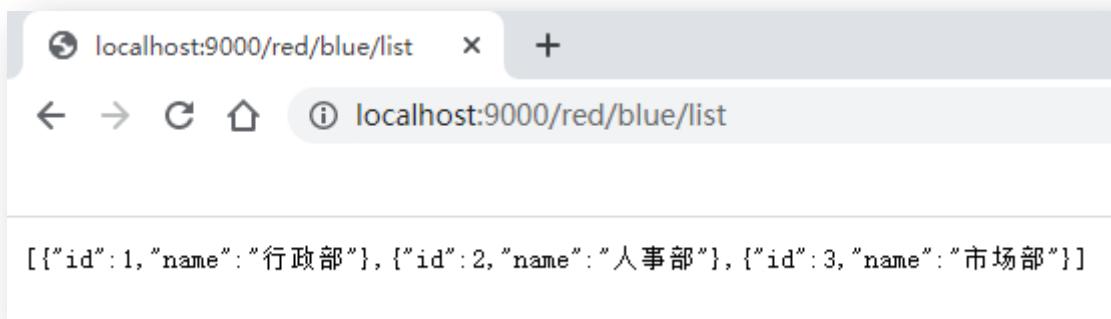
该过滤器工厂会将请求 URI 替换为另一个指定的 URI 进行访问。RewritePath 有两个参数，第一个是正则表达式，第二个是要替换为的目标表达式。

#### (2) 配置式配置文件-无正则

将/red/blue 替换为/provider/depart。



## (3) 运行效果

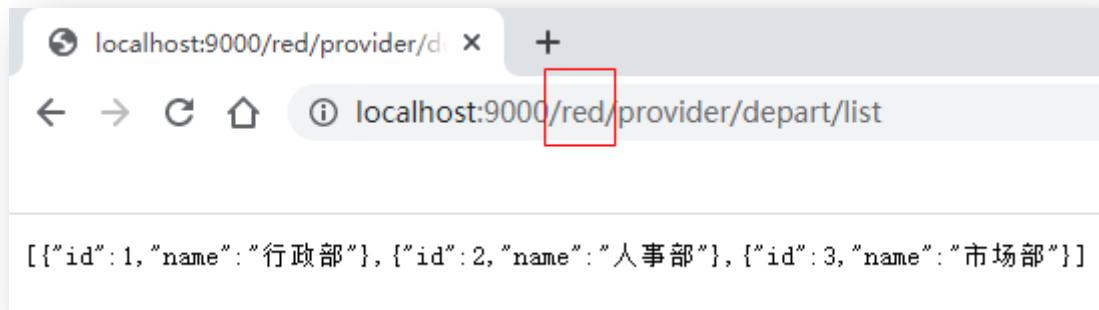


## (4) 配置式配置文件-正则



```
spring:
  cloud:
    gateway:
      routes:
        - id: my_route
          uri: http://localhost:8081
          predicates:
            - Path=/**
          filters:
            - RewritePath=/red/?(<segment>.*), /$\\{segment}
```

## (5) 运行效果



## (6) API 式启动类

直接修改路由方法。

```
no usages
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("my_route",
            ps -> ps.path("/{**}) BooleanSpec
                .filters(fs -> fs.rewritePath("/red/blue", "/provider/depart"))
                .uri("http://localhost:8081"))
        .build();
}
```

## (7) 应用场景(学员无)

与前面的场景类似。例如，在某子模块中的某功能，通过其它子模块也可以访问，此时就可以通过 `uri` 替换方式让不同的子模块路径跳转到同一个路径。

### 5.6.11 RequestRateLimiterGateway 过滤工厂

#### (1) 规则

该过滤器工厂通过令牌桶算法对进来的请求进行限流。

## (2) 导入依赖

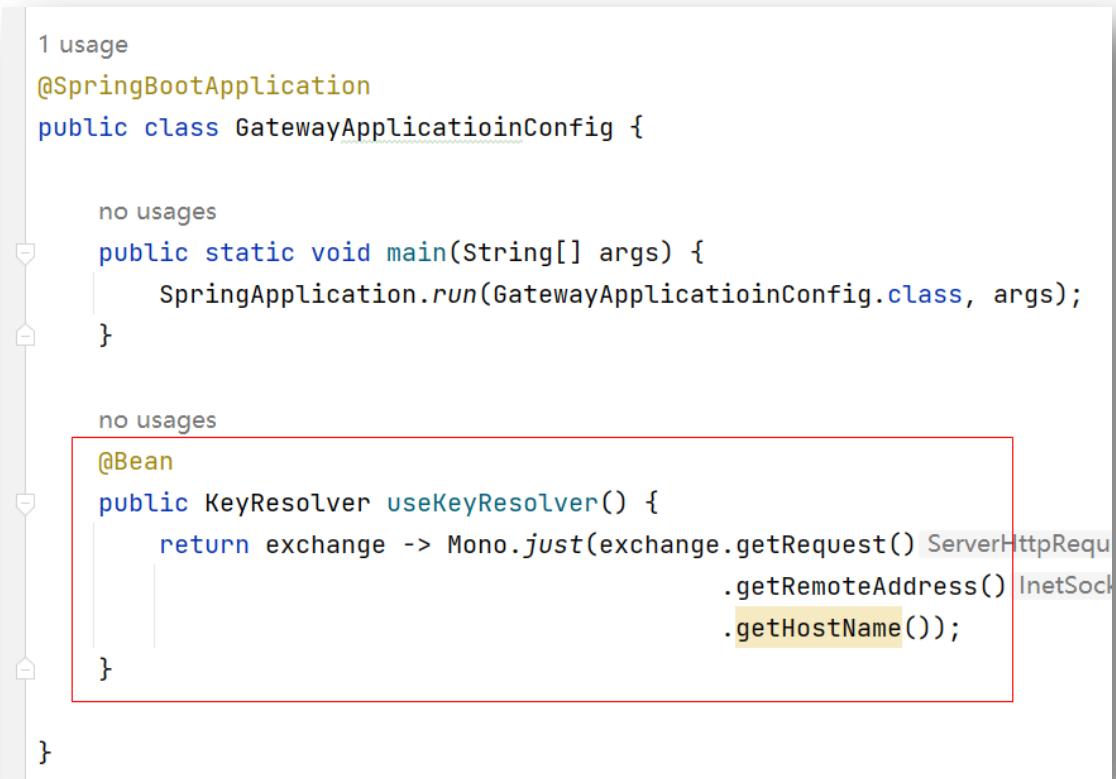
这个限流器是基于 Redis 的，所以需要导入 Redis 的 Starter 依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

## (3) 添加限流键解析器

在启动类中添加一个限流键解析器，其用于从请求中解析出需要限流的 key。限流 key 可以是用户、URI，也可以是目标服务器的 host 或 IP。

本例指定的是根据请求要访问的目标服务器的 host 或 ip 进行限流。



```
1 usage
@SpringBootApplication
public class GatewayApplicationConfig {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplicationConfig.class, args);
    }

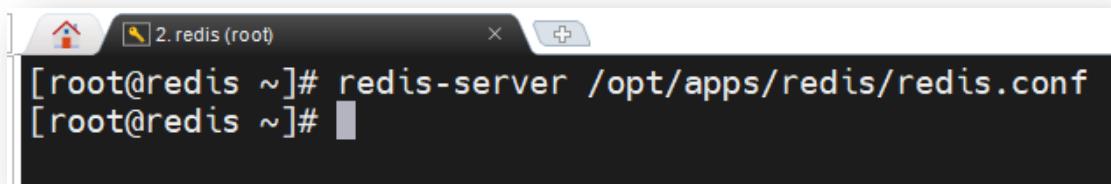
    no usages
    @Bean
    public KeyResolver useKeyResolver() {
        return exchange -> Mono.just(exchange.getRequest() ServerHttpRequest
            .getRemoteAddress() InetSocketAddress
            .getHostName());
    }
}
```

## (4) 修改配置文件

```
4   spring:
5     data:
6       redis:
7         host: 192.168.192.102
8         port: 6379
9
10    cloud:
11      gateway:
12        routes:
13          - id: my_route
14            uri: http://localhost:8081
15            predicates:
16              - Path=/provider/depart/**
17            filters:
18              - name: RequestRateLimiter
19                args:
20                  key-resolver: "#{@useKeyResolver}"
21                  redis-rate-limiter.replenishRate: 2
22                  redis-rate-limiter.burstCapacity: 5
23                  redis-rate-limiter.requestedTokens: 1
```

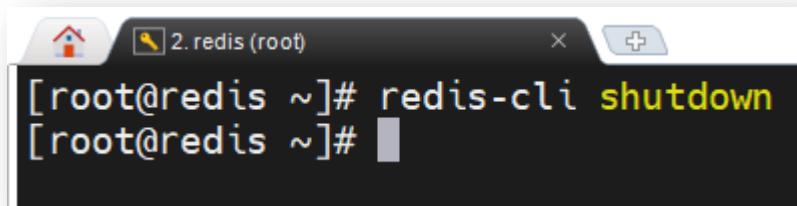
## (5) 运行效果

### A、启动 Redis



```
[root@redis ~]# redis-server /opt/apps/redis/redis.conf
[root@redis ~]#
```

以下是 Redis 的关闭。



```
[root@redis ~]# redis-cli shutdown
[root@redis ~]#
```

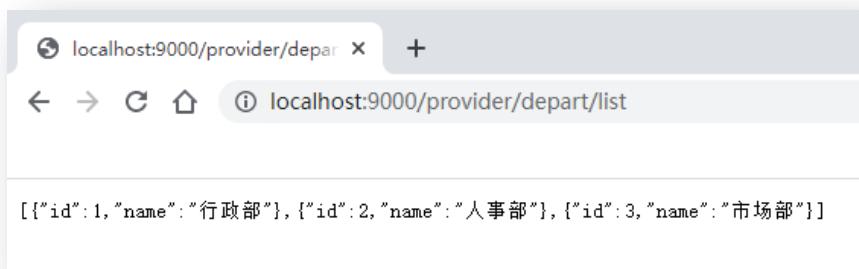
### B、启动应用

启动如下应用：

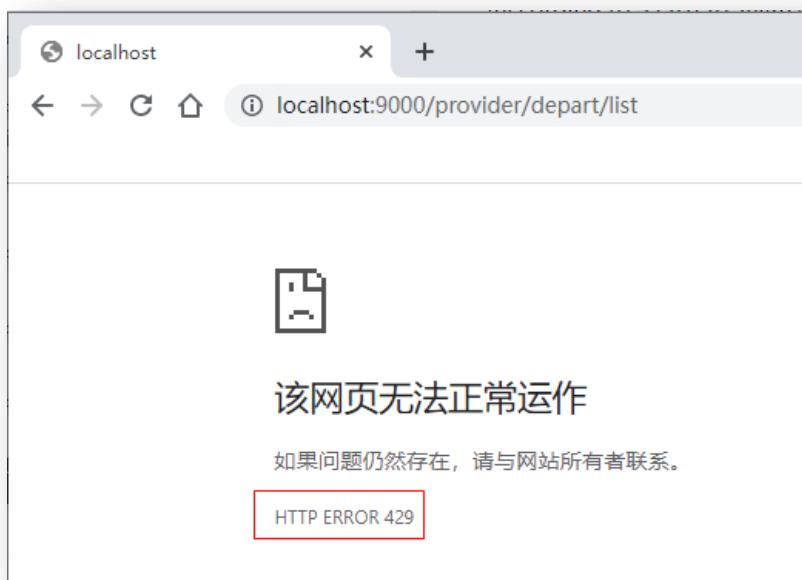
- 01-provider-8081
- 05-gateway-config-9000

### C、访问

正常访问时是没有问题的。



但如果快速连续刷新页面，则会看到如下异常。这是限流后的结果。



### 5.6.12 默认 Filters 工厂

前面的 `GatewayFilter` 工厂是在某一特定路由策略中设置的，仅对这一种路由生效。若要使某些过滤效果应用到所有路由策略中，就可以将该 `GatewayFilter` 工厂定义在默认 `Filters` 中。

## (1) 修改配置文件

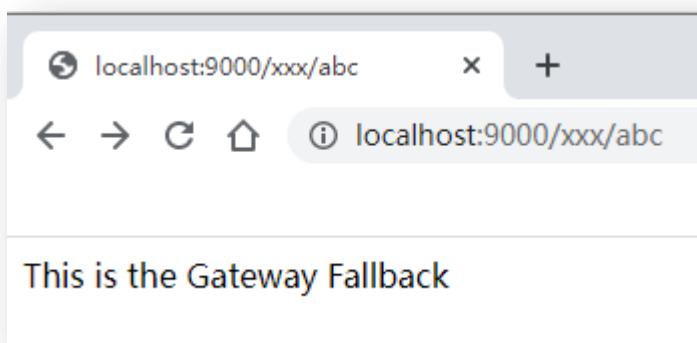
```
spring:
  cloud:
    gateway:
      default-filters:
        - name: CircuitBreaker
          args:
            name: myCircuitBreaker
            fallbackUri: forward:/fb

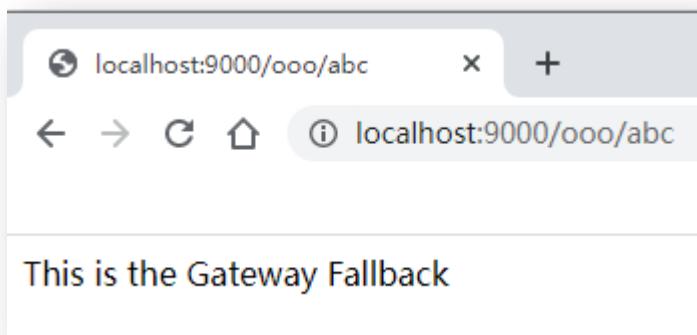
      routes:
        - id: my_route
          uri: http://localhost:8081
          predicates:
            - Path=xxx/**

        - id: my_route_2
          uri: http://localhost:8081
          predicates:
            - Path=ooo/**
```

default-filters 下定义的路由为默认路由，而 routes 的 filters 下定义的路由则为局部路由。

## (2) 运行效果





### 5.6.13 优先级

对于相同 filter 工厂，在不同位置设置了不同的值，则优先级为：

- 局部 filter 的优先级高于默认 filter 的
- API 式的 filter 优先级高于配置式 filter 的

## 5.7 自定义网关过滤工厂

### 5.7.1 添加请求头

这里要实现的需求是，在自定义的 Filter 中为请求添加指定的请求头。

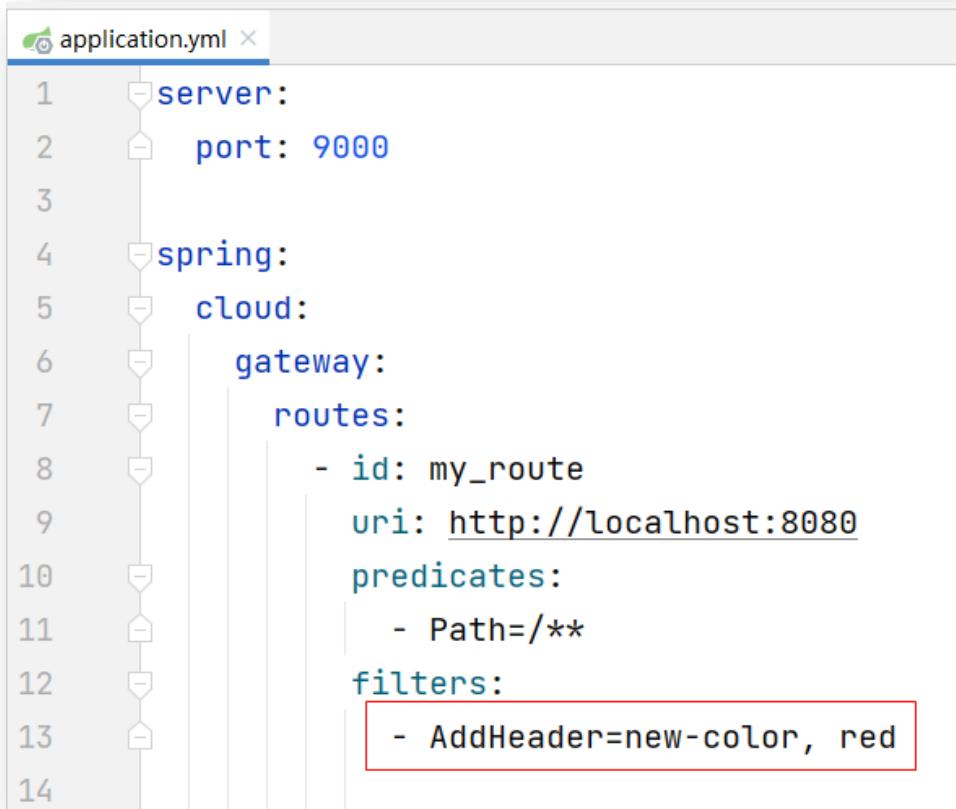
#### (1) 自定义 Factory

在 05-gateway-config-9000 工程中定义 AddHeaderGatewayFilterFactory 类。

该类类名由两部分构成：后面必须是 GatewayFilterFactory，前面为功能前缀，该前缀将来要用在配置文件中。

```
no usages
@Component
public class AddHeaderGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return (exchange, chain) -> {
            ServerHttpRequest changedRequest = exchange.getRequest() ServerHttpRequest
                .mutate() Builder
                .header(config.getName(), config.getValue())
                .build();
            ServerWebExchange webExchange = exchange.mutate()
                .request(changedRequest)
                .build();
            return chain.filter(webExchange);
        };
    }
}
```

## (2) 修改配置文件



The screenshot shows the `application.yml` configuration file with the following content:

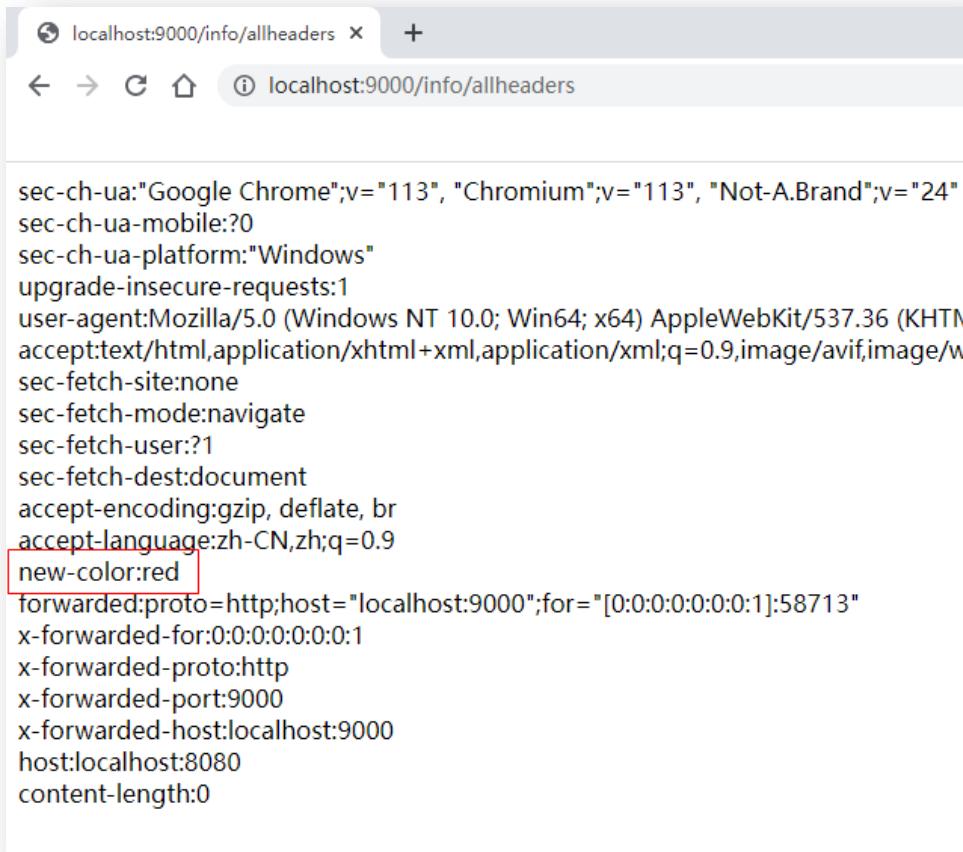
```
server:
  port: 9000

spring:
  cloud:
    gateway:
      routes:
        - id: my_route
          uri: http://localhost:8080
          predicates:
            - Path=/**
          filters:
            - AddHeader=new-color, red
```

The `filters` section is highlighted with a red border.

### (3) 运行效果

分别启动 05-showinfo-8080 与 05-gateway-config-9000 工程，然后在浏览器进行访问。这里直接访问 showInfo 工程中原来的 allheaders 处理器方法。我们可以看到，请求中的触新增了请求头。



```
localhost:9000/info/allheaders +  
← → ⌂ ⌂ localhost:9000/info/allheaders  
  
sec-ch-ua:"Google Chrome";v="113", "Chromium";v="113", "Not-A.Brand";v="24"  
sec-ch-ua-mobile?:0  
sec-ch-ua-platform:"Windows"  
upgrade-insecure-requests:1  
user-agent:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)  
accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*  
sec-fetch-site:none  
sec-fetch-mode:navigate  
sec-fetch-user:?1  
sec-fetch-dest:document  
accept-encoding:gzip, deflate, br  
accept-language:zh-CN,zh;q=0.9  
new-color:red  
forwarded:proto=http;host="localhost:9000";for="[0:0:0:0:0:0:1]:58713"  
x-forwarded-for:0:0:0:0:0:1  
x-forwarded-proto:http  
x-forwarded-port:9000  
x-forwarded-host:localhost:9000  
host:localhost:8080  
content-length:0
```

#### 5.7.2 GatewayFilter 的 pre 与 post

下面我们要定义出多个 Filter，每个 Filter 都具有 pre 与 post 两部分。将所有 Filter 注册到路由中，以查看它们执行的顺序。

### (1) pre 与 post

Spring Cloud Gateway 同 zuul 类似，有“pre”和“post”两种方式的 filter。客户端的请求先按照 filter 的优先级顺序（优先级相同，则按注册顺序），执行 filter 的“pre”部分。然后将请求转发到相应的目标服务器，收到目标服务器的响应之后，再按照 filter 的优先级顺序的逆序（优先级相同，则按注册顺序逆序），执行 filter 的“post”部分，最后返回响应到客户端。

filter 的“pre”部分指的是 chain.filter()方法执行之前的代码，而“post”部分，则是定义在 chain.filter().then()方法中的代码。

## (2) 定义第一个 FilterFactory

```
no usages
@Component
@Slf4j
public class OneGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return (exchange, chain) -> {
            long start = System.currentTimeMillis();
            log.info(config.getName() + " - " + config.getValue() + "开始时间: " + start);
            exchange.getAttributes().put("startTime", start);
            return chain.filter(exchange).then(
                Mono.fromRunnable(() -> {
                    Long startTime = (Long) exchange.getAttributes().get("startTime");
                    long endTime = System.currentTimeMillis();
                    long elapsedTime = endTime - startTime;
                    log.info(config.getName() + " - " + config.getValue() + "结束时间: " + endTime);
                    log.info("该filter执行用时(毫秒): " + elapsedTime);
                })
            );
        };
    }
}
```

### (3) 定义第二个 FilterFactory

```
no usages
@Component
@Slf4j
public class TwoGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return (exchange, chain) -> {
            log.info(config.getName() + "-" + config.getValue());
            return chain.filter(exchange).then(
                Mono.fromRunnable(() -> {
                    log.info(config.getName() + "-" + config.getValue());
                })
            );
        };
    }
}
```

### (4) 定义第三个 FilterFactory

第三个 FilterFactory 的代码与第二个的完全相同。

```
no usages
@Component
@Slf4j
public class ThreeGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return (exchange, chain) -> {
            log.info(config.getName() + "-" + config.getValue());
            return chain.filter(exchange).then(
                Mono.fromRunnable(() -> {
                    log.info(config.getName() + "-" + config.getValue());
                })
            );
        };
    }
}
```

## (5) 修改配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   cloud:
6     gateway:
7       routes:
8         - id: my_route
9           uri: http://localhost:8080
10          predicates:
11            - Path=/*
12          filters:
13            - One=onefilter, 111
14            - Two=twofilter, 222
15            - Three=threefilter, 333
16
```

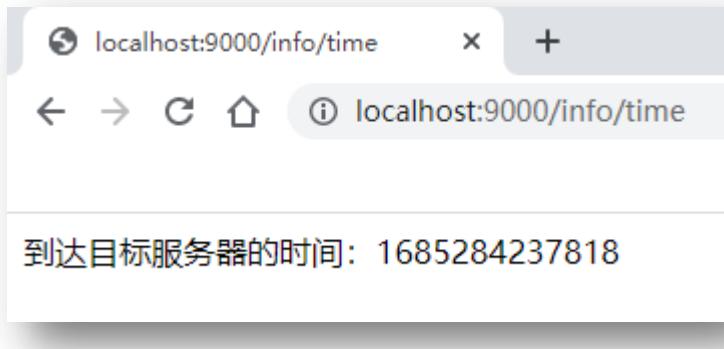
## (6) 修改 ShowInfo 工程的处理器

在 05-showinfo-8080 工程中添加如下处理器。

```
no usages
@RequestMapping("/time")
public String timeHandle() {
    return "到达目标服务器的时间: " + System.currentTimeMillis();
}
```

## (7) 运行效果

分别启动 05-showinfo-8080 与 05-gateway-config-9000 工程，然后在浏览器进行访问。然后再观察两个工程的控制台输出。



查看 05-gateway-config-9000 工程控制台。



```
: onefilter-111开始时间: 1685284237816
: twofilter-222
: threefilter-333
: threefilter-333
: twofilter-222
: onefilter-111结束时间: 1685284237819
: 该filter执行用时(毫秒): 3
```

## 5.8 全局过滤器

### 5.8.1 简介

全局过滤器 Global Filter 是应用于所有路由策略上的 Filter。Spring Cloud Gateway 中已经定义好了很多 GlobalFilter，但这些 GlobalFilter 无需任何的配置与声明，在符合应用条件时就会自动生效。

## 5.8.2 负载均衡全局过滤器

### (1) 需求

根据微服务名称进行负载均衡。有一个微服务，有三个提供者。该微服务被另一个微服务消费，其具有三个消费者。这里要定义一个 Spring Cloud Gateway 网关，实现对该微服务消费者的负载均衡访问。

### (2) 配置式路由

直接在 05-gateway-config-9000 工程中修改。

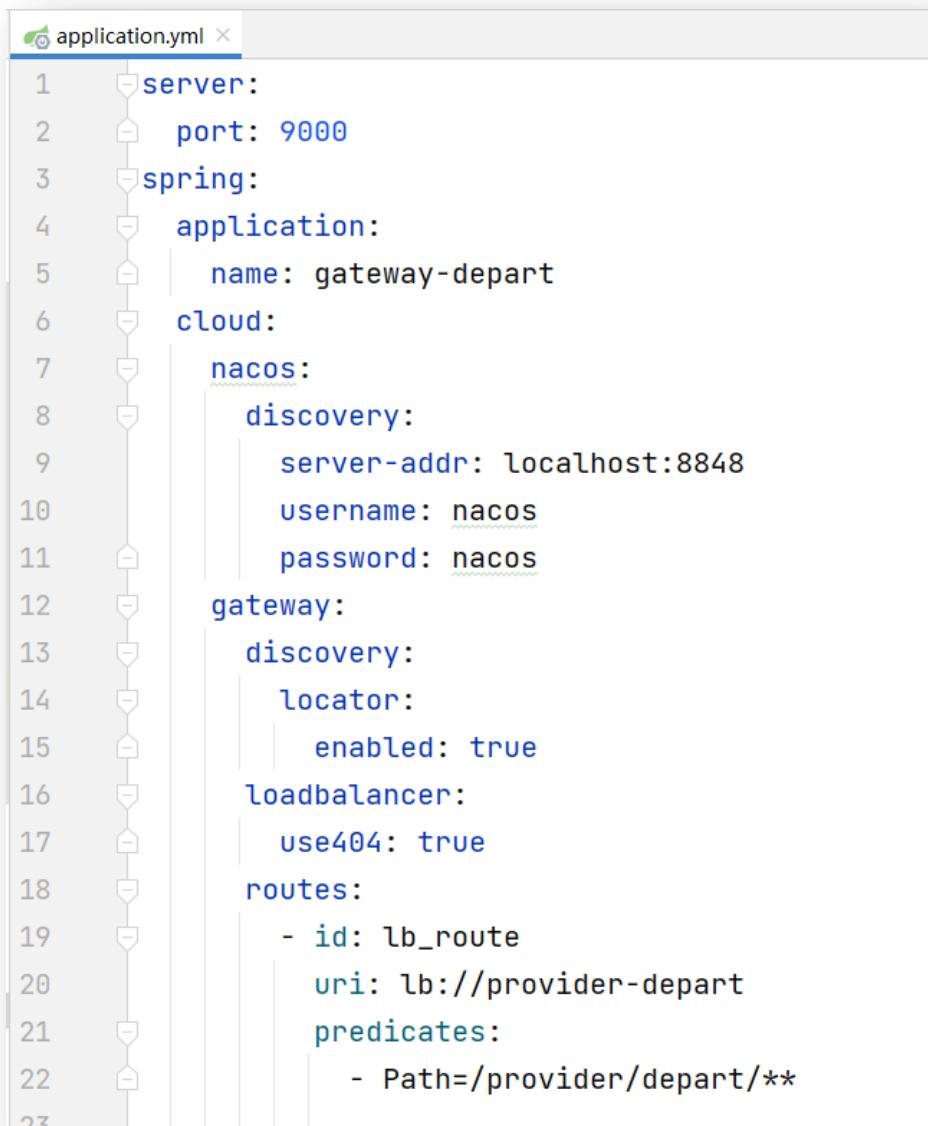
#### A、添加依赖

需要导入 loadbalancer 与 Nacos Discovery 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>

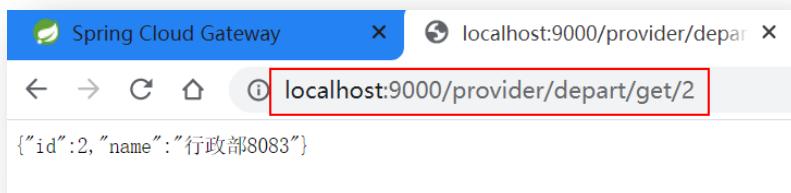
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

## B、修改配置文件



```
application.yml
1 server:
2   port: 9000
3
4 spring:
5   application:
6     name: gateway-depart
7
8   cloud:
9     nacos:
10       discovery:
11         server-addr: localhost:8848
12         username: nacos
13         password: nacos
14
15       gateway:
16         discovery:
17           locator:
18             enabled: true
19             loadbalancer:
20               use404: true
21
22             routes:
23               - id: lb_route
                 uri: lb://provider-depart
                 predicates:
                   - Path=/provider/depart/**
```

## C、启动并访问



localhost:9000 为网关地址, /provider/depart/get/2 会在网关中进行断言, 匹配上后会找到其要访问的微服务名称, 通过负载均衡方式找到一个提供者主机, 然后再访问 /provider/depart/get/2 指定的资源。

不断刷新页面, 可以看到负载均衡的结果。

### (3) API 式路由

直接在 05-gateway-api-9001 工程中修改。

#### A、添加依赖

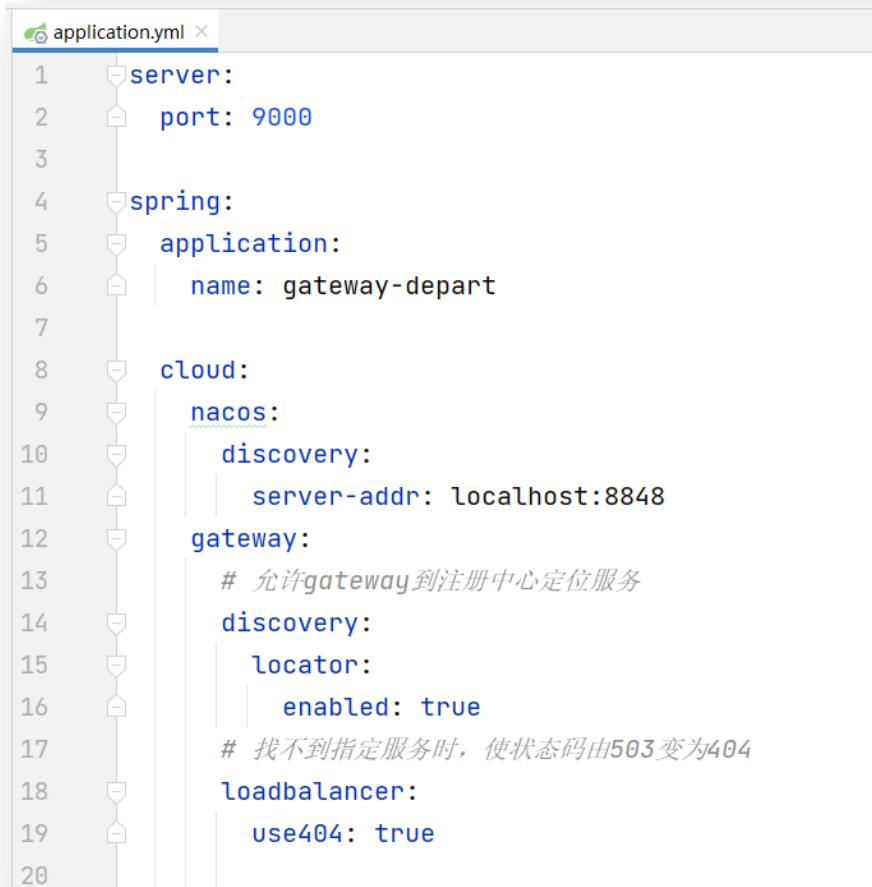
需要导入 loadbalancer 与 Nacos Discovery 依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>

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

#### B、修改配置文件

删除配置文件中的 routes 相关配置, 保留其它。



```
application.yml ×
1  server:
2    port: 9000
3
4  spring:
5    application:
6      name: gateway-depart
7
8  cloud:
9    nacos:
10      discovery:
11        server-addr: localhost:8848
12
13  gateway:
14    # 允许gateway到注册中心定位服务
15    discovery:
16      locator:
17        enabled: true
18    # 找不到指定服务时，使状态码由503变为404
19    loadbalancer:
20      use404: true
```

## C、修改启动类

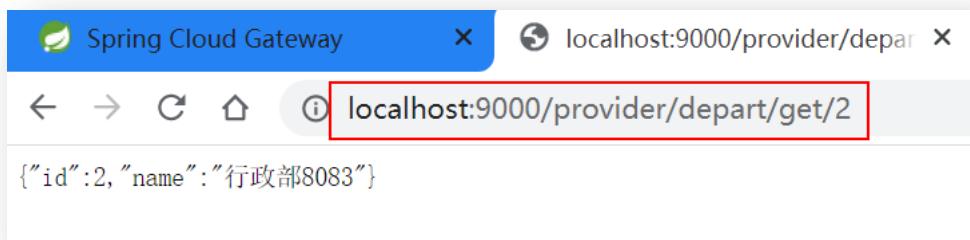
```
1 usage
@SpringBootApplication
public class GatewayApplication9000 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication9000.class, args);
    }

    no usages
    @Bean
    public RouteLocator routeLocator(RouteLocatorBuilder builder) {
        return builder.routes()
            .route("lb_route", ps -> ps.path("/provider/depart/**"))
            .uri("lb://provider-depart")
        .build();
    }

}
```

## D、启动并访问



### 5.8.3 自定义 Global Filter

Global Filter 不需要在任何具体的路由规则中进行注册，只需在类上添加@Compoment 注解，将其生命周期交给 Spring 容器来管理即可。

## (1) 需求

这里要实现的需求是，访问当前系统的任意模块的 URL 都需要是合法的 URL。这里所谓合法的 URL 指的是请求中携带了 token 请求参数。

由于是对所有请求的 URL 都要进行验证，所以这里就需要定义一个 Global Filter，可以应用到所有路由中。

## (2) 定义 URLValidateGlobalFilter

直接在 05-gateway-config-9000 工程中修改。

需要注意，GlobalFilter 不需要在具体的路由规则中进行注册，只需将其生命周期交给 Spring 容器来管理即可。

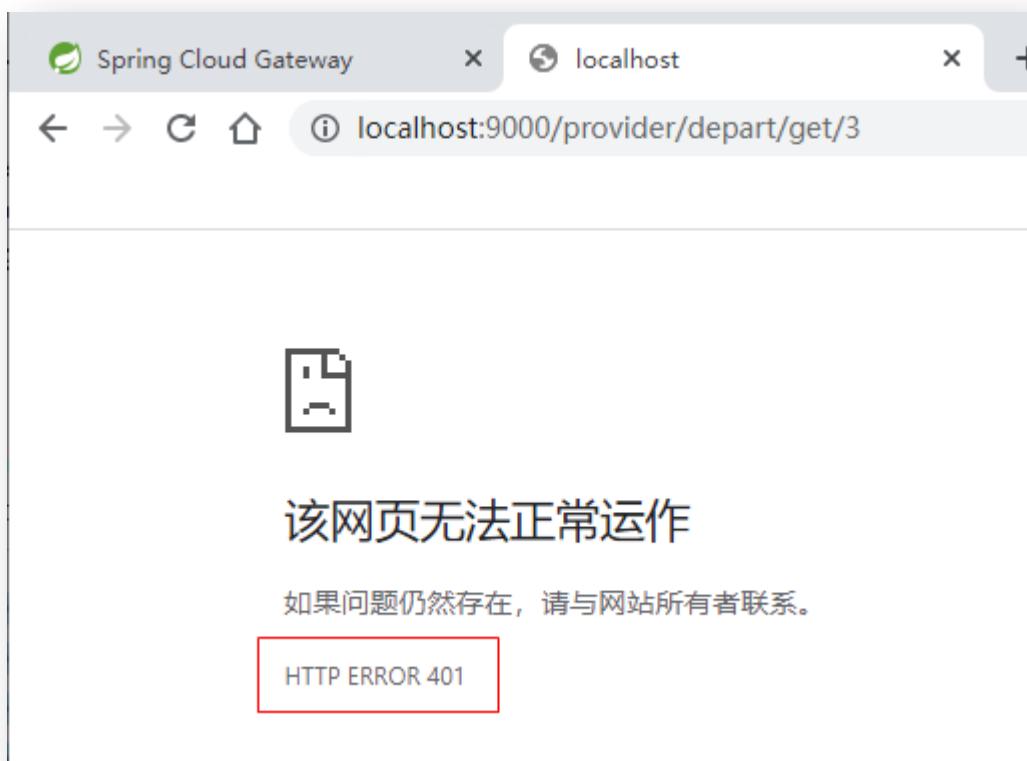


```
@Component
public class URLValidateFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        // 从请求中获取请求参数token
        String token = exchange.getRequest().getQueryParams().getFirst("token");
        // 若token为空，则响应客户端状态码401, 未授权; 否则通过过滤
        if (!StringUtils.hasText(token)) {
            exchange.getResponse().setStatus(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
        return chain.filter(exchange);
    }

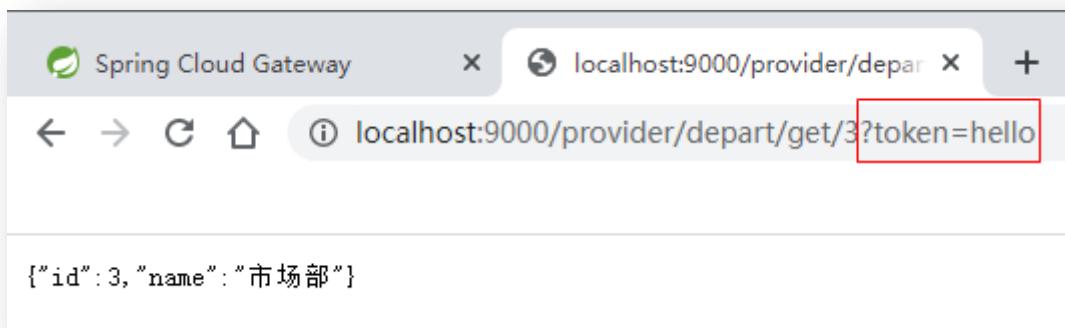
    @Override
    public int getOrder() {
        return Ordered.HIGHEST_PRECEDENCE;
    }
}
```

## (3) 运行效果

分别启动 05-showinfo-8080 与 05-gateway-config-9000 工程，然后在浏览器进行访问。这里可以访问 showInfo 工程中原来的任意处理器方法。只不过，若请求中没有携带 token 请求参数，则验证失败。



请求中携带了 token 参数，无论其值为什么均可通过验证。



## 5.9 跨域配置

### 5.9.1 跨域概述

#### (1) 跨域与同源

为了安全，浏览器中启用了一种称为**同源策略**的安全机制，禁止从一个域名页面中请求

另一个域名下的资源。

当两个请求的访问协议、域名与端口号三者都相同时，才称它们是同源的。只要有一个不同，就称为跨源请求。

源: <http://sports.abc.com/content/kb>

跨源: <https://sports.abc.com/content/kb>

跨源: <http://news.abc.com/content/kb>

跨源: <http://sports.abc.com:8080/content/kb>

同源: <http://sports.abc.com/content/abc>

## (2) CORS

CORS, Cross-Origin Resource Sharing, 跨域资源共享，是一种允许当前域的资源(例如，html、js、web service)被其他域的脚本请求访问的机制。

### 5.9.2 跨域问题模拟

#### (1) 定义 html 页面

在任意工程中定义一个 html 页面，该页面要通过 JS 提交跨域访问请求。

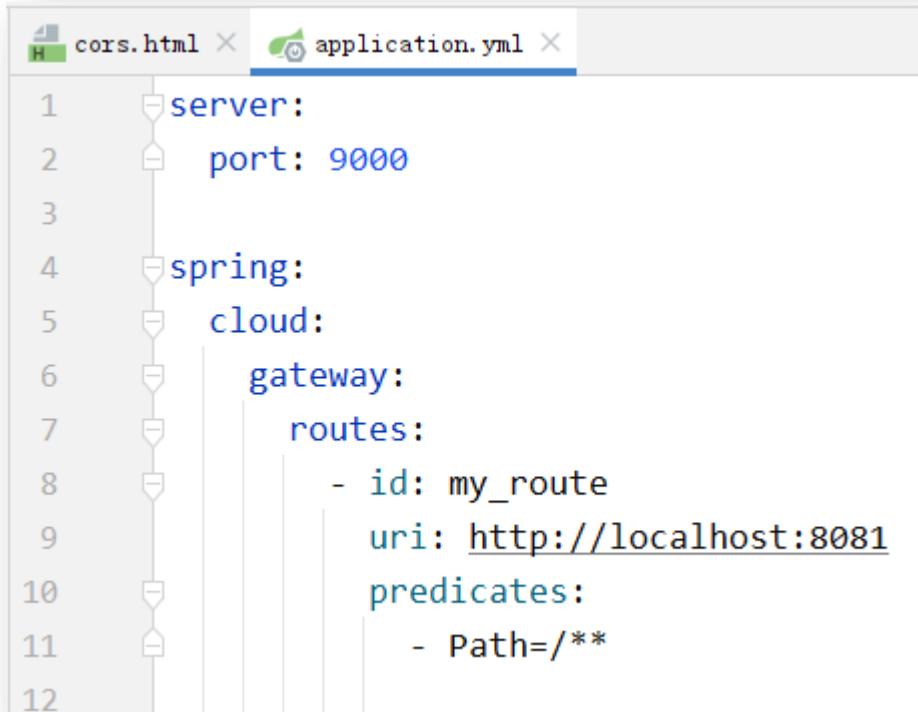


```
cors.html
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8">
5          <title>cors</title>
6          <script src="https://s3.pstatp.com/cdn/expire-1-M/jquery/3.3.1/jquery.min.js"></script>
7          <script>
8              1 usage
9              function accessDepart() {
10                  $.get("http://localhost:9000/provider/depart/get/3", function (data) {
11                      alert(data.name);
12                  });
13              }
14          </script>
15      </head>
16
17      <body>
18          <input type="button" value="访问Depart" onclick="accessDepart()">
19      </body>
20  </html>
```

<https://s3.pstatp.com/cdn/expire-1-M/jquery/3.3.1/jquery.min.js>

## (2) 修改 gateway 配置文件

修改 05-gateway-config-9000 工程中的配置文件。要求其仅包含一个 Path 断言即可。

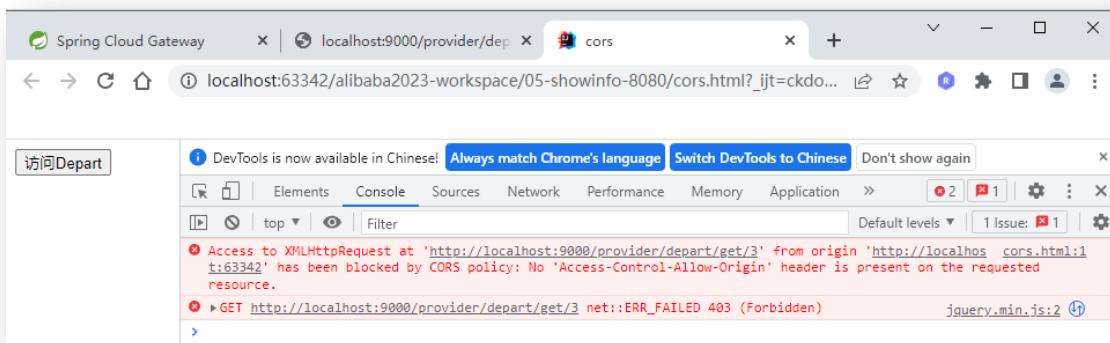


```
cors.html × application.yml ×
1   server:
2     port: 9000
3
4   spring:
5     cloud:
6       gateway:
7         routes:
8           - id: my_route
9             uri: http://localhost:8081
10            predicates:
11              - Path=/**
```

## (3) 页面访问

在访问页面之前，先将一个 provider 工程及 05-gateway-config-9000 工程启动。然后右击 cors.html，在浏览器中打开。

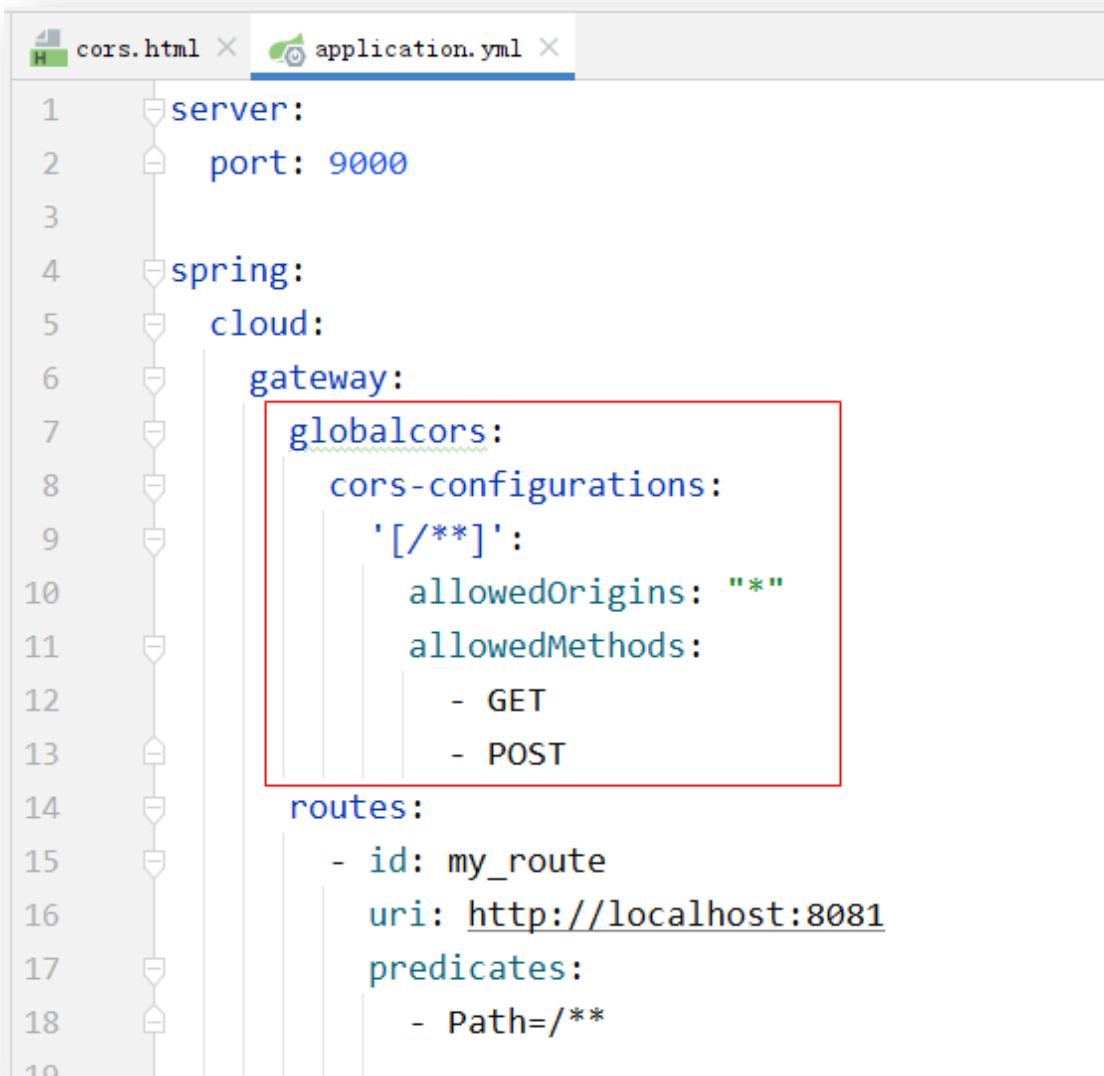
在浏览器中 F12 打开开发者模式，点击按钮后即可看到有跨域访问的异常提示。



### 5.9.3 Gateway 解决跨域问题

#### (1) 全局解决方案

修改 05-gateway-config-9000 工程中的配置文件，在其中添加全局 cors 配置。该解决方案对当前配置文件中的所有路由均起作用。



```
cors.html × application.yml ×
1   server:
2     port: 9000
3
4   spring:
5     cloud:
6       gateway:
7         globalcors:
8           cors-configurations:
9             '[/**]':
10              allowedOrigins: "*"
11              allowedMethods:
12                - GET
13                - POST
14
15             routes:
16               - id: my_route
17                 uri: http://localhost:8081
18                 predicates:
19                   - Path=/**
```

此时在浏览器上刷新页面后，再点击按钮即可看到 alert 提示框。

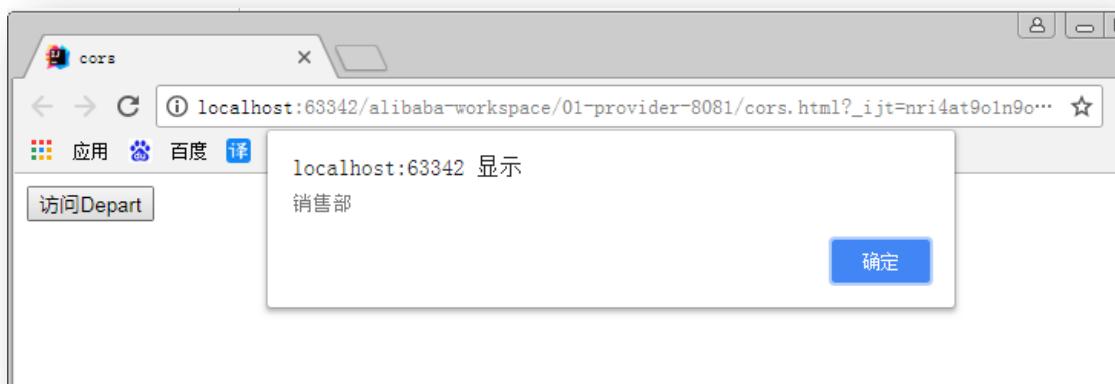


## (2) 局部解决方案

修改 05-gateway-config-9000 工程中的配置文件，在某个具体的路由中添加局部 cors 配置。该解决方案仅对当前路由起作用。



此时在浏览器上刷新页面后，再点击按钮即可看到 alert 提示框。



## 第6章 Sentinel 流量防卫兵

### 6.1 Sentinel 简介

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 是面向分布式、多语言异构化服务架构的流量治理组件，主要以流量为切入点，从流量路由、流量控制、流量整形、熔断降级、系统自适应过载保护、热点流量防护等多个维度来帮助开发者保障微服务的稳定性。

Sentinel 是分布式系统的防御系统。

官网: <https://sentinelguard.io/zh-cn>

github 上的官网: <https://github.com/alibaba/Sentinel>

### 6.2 Sentinel 控制台

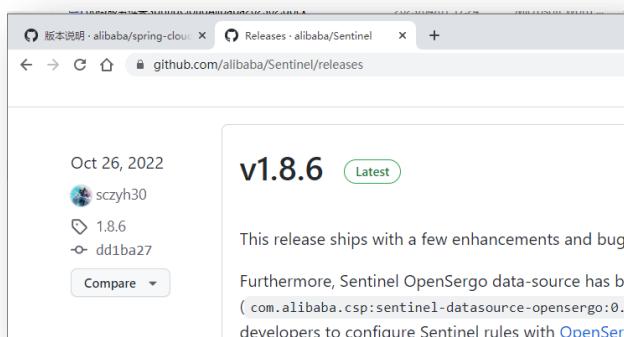
无论是服务熔断、服务流控、服务鉴权等，都首先会涉及到相关规则的可视化管理，即通过 Sentinel 控制台来管理相关规则。所以这里首先来学习 Sentinel 控制台。

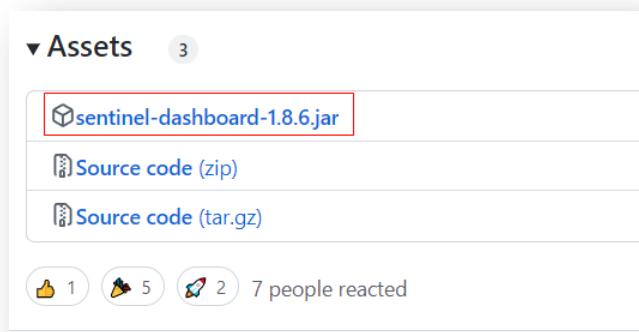
#### 6.2.1 简介

Sentinel 控制台是 Sentinel 的一个轻量级开源 GUI 控制台，可以提供对 Sentinel 主机(Sentinel 应用)的发现及健康管理、动态配置服务流控、熔断、路由规则的配置与管理。

#### 6.2.2 下载

直接从官方下载打包好的 Sentinel Dashboard 启动运行。下载链接在 Sentinel Dashboard 官网。不过，在下载时要先查清楚当前的 Spring Cloud Alibaba 版本兼容的是哪个 Sentinel 版本，然后再下载相应 Sentinel 版本的控制台。





### 6.2.3 启动

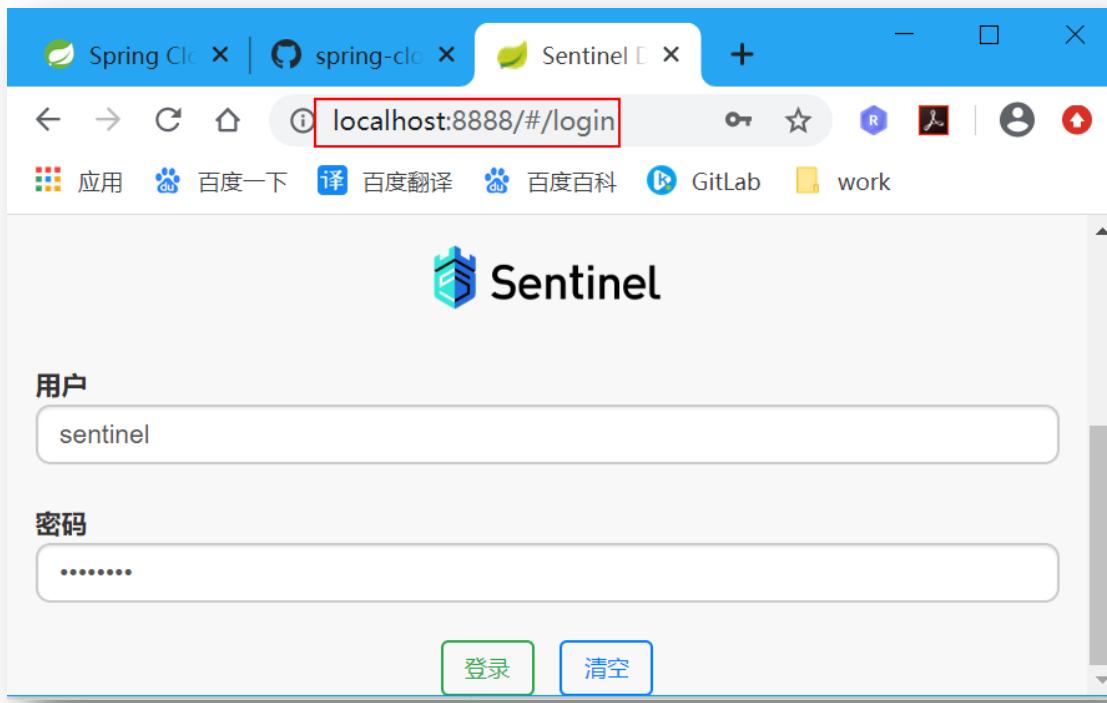
启动 Sentinel 控制台需要 JDK 版本为 1.8 及以上版本。在 Sentinel 控制 jar 包所在目录中打开 cmd 窗口，执行启动命令。



```
C:\Windows\System32\cmd.exe - java -Dserver.port=8888 -Dcsp.sentinel.dashboard.server=localhost:8888 -Dproject.name=sentinel-dashboard -Dsentry.dashboard.auth.username=sentinel -Dsentry.dashboard.auth.password=111 -jar sentinel-dashboard-1.8.6.jar
```

```
java -Dserver.port=8888 ^  
-Dsentry.dashboard.auth.username=sentinel ^  
-Dsentry.dashboard.auth.password=111 ^  
-jar sentinel-dashboard-1.8.6.jar
```

## 6.2.4 访问



由于还没有其它应用，所以这里仅可以看到当前 dashboard 应用本身。

## 6.3 服务降级

由于马上要学习的服务熔断会涉及到服务降级，所以这里先学习服务降级。

### 6.3.1 概述

服务降级是一种增强用户体验的方式。当用户的请求由于各种原因被拒后，系统返回一个事先设定好的、用户可以接受的，但又令用户并不满意的结果。这种请求处理方式称为服务降级。

### 6.3.2 Sentinel 式方法级降级

#### (1) 定义工程

复制 02-consumer-nacos-8080 工程，重命名为 06-consumer-degrade-sentinel-method-8080。

## (2) 添加依赖

除了 spring cloud alibaba 依赖、Nacos 依赖外，还需要再导入 Sentinel 依赖。由于其它依赖在工程中已经有了，所以这里仅添加 Sentinel 依赖。

```
<!--sentinel 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

## (3) 修改处理器类

这里要为控制器类 DepartController 的 getHandle()方法添加服务降级，所以在该方法上添加 @SentinelResource 注解，并在 DepartController 类中定义降级方法。

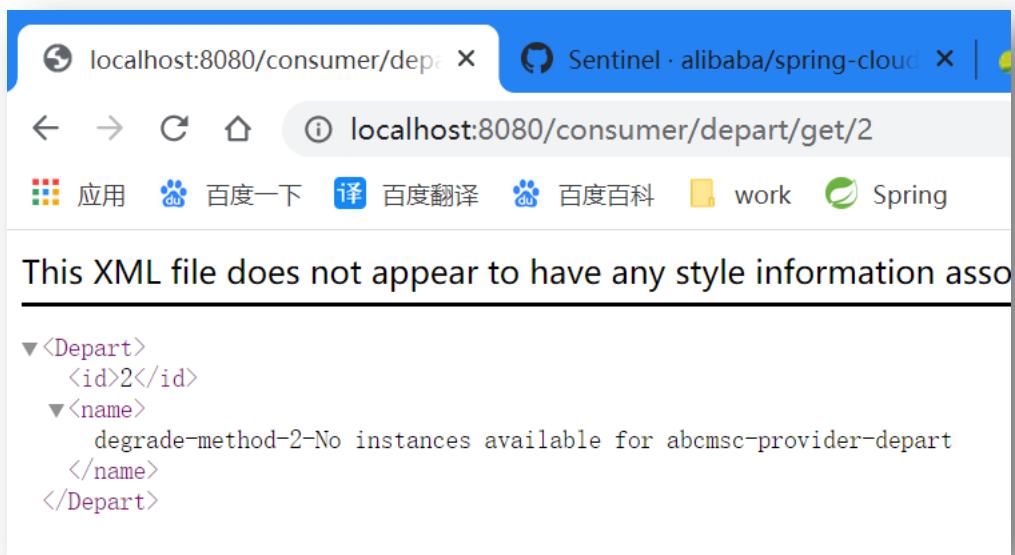
该方法返回值与参数类型必须与原方法的相同，方法名可以随意。不过，也可以在降级方法中增加一个 Throwable 类型的参数，用于获取到降级时相应异常对象。

```
// fallback属性用于指定降级方法名称
@SentinelResource(fallback = "getHandlerFallback")
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = SERVICE_PROVIDER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}

// 定义降级处理方法
public Depart getHandlerFallback(int id, Throwable e) {
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("degrade-method-" + id + "-" + e.getMessage());
    return depart;
}
```

## (4) 运行访问

不用启动 Provider，直接运行 Consumer，就可以看到降级的结果。



### 6.3.3 Sentinel 式类级降级

前面例子中所有控制器方法都需要添加降级方法，若都使用前面的方式，将会导致控制器类非常复杂，不便于后期维护。所以可以将这些降级方法全部定义到一个类中。

#### (1) 定义工程

复制 02-consumer-nacos-8080 工程，重命名为 06-consumer-degrade-sentinel-class-8080。

#### (2) 添加依赖

除了 spring cloud alibaba 依赖、Nacos 依赖外，还需要再导入 Sentinel 依赖。由于其它依赖在工程中已经有了，所以这里仅添加 Sentinel 依赖。

```
<!--sentinel 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

### (3) 定义降级类

定义的降级类所在包没有要求，类名没有要求。在该类中定义的方法均为业务需要的降级方法。降级类中的方法必须是静态方法，方法返回值与参数类型必须与原方法的相同，方法名可以随意。不过，也可以在降级方法中增加一个 `Throwable` 类型的参数，用于获取到降级时相应异常对象。

```
// 自定义降级类
public class DepartServiceFallback {
    public static Depart getFallback(int id, Throwable e) {
        System.out.println("getHandle()执行异常 " + id);
        Depart depart = new Depart();
        depart.setId(id);
        depart.setName("degrade-class-" + id + "-" + e.getMessage());
        return depart;
    }
    public static List<Depart> listFallback() {
        System.out.println("listHandle()执行异常");
        List<Depart> list = new ArrayList();
        Depart depart = new Depart();
        depart.setName("no any depart");
        list.add(depart);
        return list;
    }
}
```

### (4) 修改处理器类

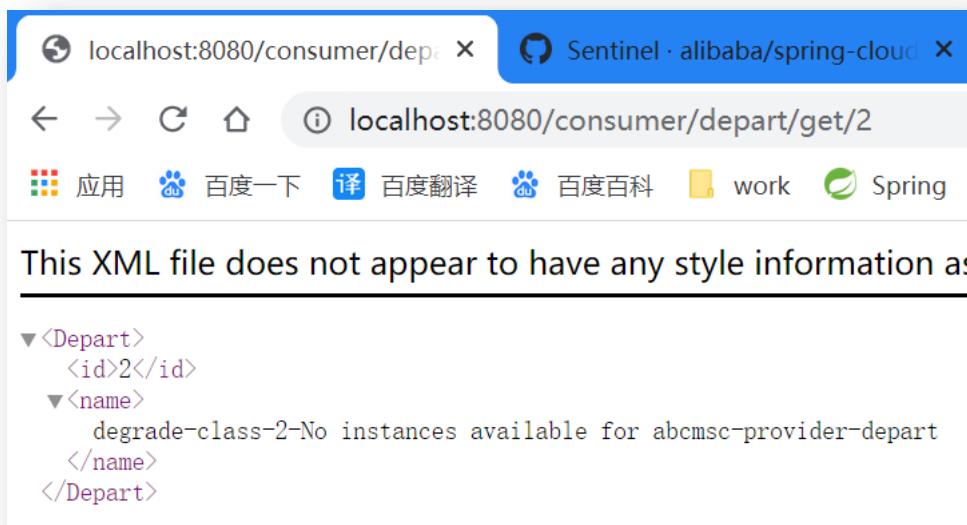
为处理器 `DepartController` 中的 `getHandle()` 与 `list()` 方法添加 `@SentinelResource` 注解，指定具体的降级类的降级方法。

```
@SentinelResource(fallback = "getFallback",
    fallbackClass = DepartServiceFallback.class)
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = SERVICE_PROVIDER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}

@SentinelResource(fallback = "listFallback",
    fallbackClass = DepartServiceFallback.class)
@GetMapping("/list")
public List<Depart> listHandle() {
    String url = SERVICE_PROVIDER + "/provider/depart/list/";
    return restTemplate.getForObject(url, List.class);
}
```

## (5) 运行访问

不用启动 Provider，直接运行 Consumer，就可以看到降级的结果。



### 6.3.4 Feign 式类级降级

#### (1) 定义工程

复制 04-consumer-feign-8080 工程，并重命名为 06-consumer-degrade-feign-8080。这个消费者是通过 Feign 接口进行消费的。

#### (2) 添加依赖

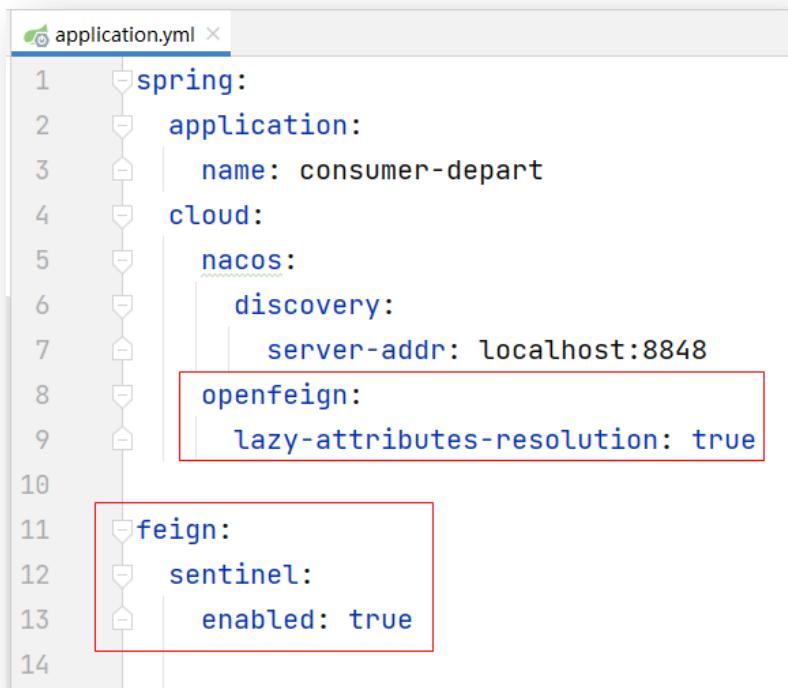
除了 spring cloud alibaba 依赖、openFeign 依赖、Nacos 依赖外，还需要再导入 Sentinel 依赖。由于其它依赖在工程中已经有了，所以这里仅添加 Sentinel 依赖。

```
<!--sentinel 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

#### (3) 修改配置文件

配置文件中除了要添加 sentinel dashboard 的配置外，还要开启 Feign 对 Sentinel 的支持，默認為 false。

另外，spring-cloud-starter-alibaba-sentinel 依赖与 spring-cloud-starter-openfeign 依赖在 Spring Boot3.0 环境下有冲突，此时需要开启 openfeign 的一个懒加载功能即可，即设置属性 spring.cloud.openfeign.lazy-attributes-resolution 的值为 true。



```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4   cloud:
5     nacos:
6       discovery:
7         server-addr: localhost:8848
8       openfeign:
9         lazy-attributes-resolution: true
10
11      feign:
12        sentinel:
13          enabled: true
14
```

#### (4) 定义降级类

降级类定义的包与类名没有要求。但要求必须实现 `Feign` 接口，且其生命周期要交由 `Spring` 容器管理，类上的`@RequestMapping` 的内容要与 `Controller` 处理器上的相同，但在最前必须要添加`/fallback` 的 URI。

不过，这些要求与学习 `Hystrix` 时的要求相同，因为这些要求是 `Feign` 的要求，并不是 `Hystrix` 或 `Sentinel` 本身的要求。

```
@Component
@RequestMapping("/fallback/consumer/depart")
public class DepartServiceFallback implements DepartService {
    @Override
    public boolean saveDepart(Depart depart) {
        System.out.println("执行saveDepart()的服务降级处理方法");
        return false;
    }

    @Override
    public boolean removeDepartById(int id) {
        System.out.println("执行removeDepartById()的服务降级处理方法");
        return false;
    }

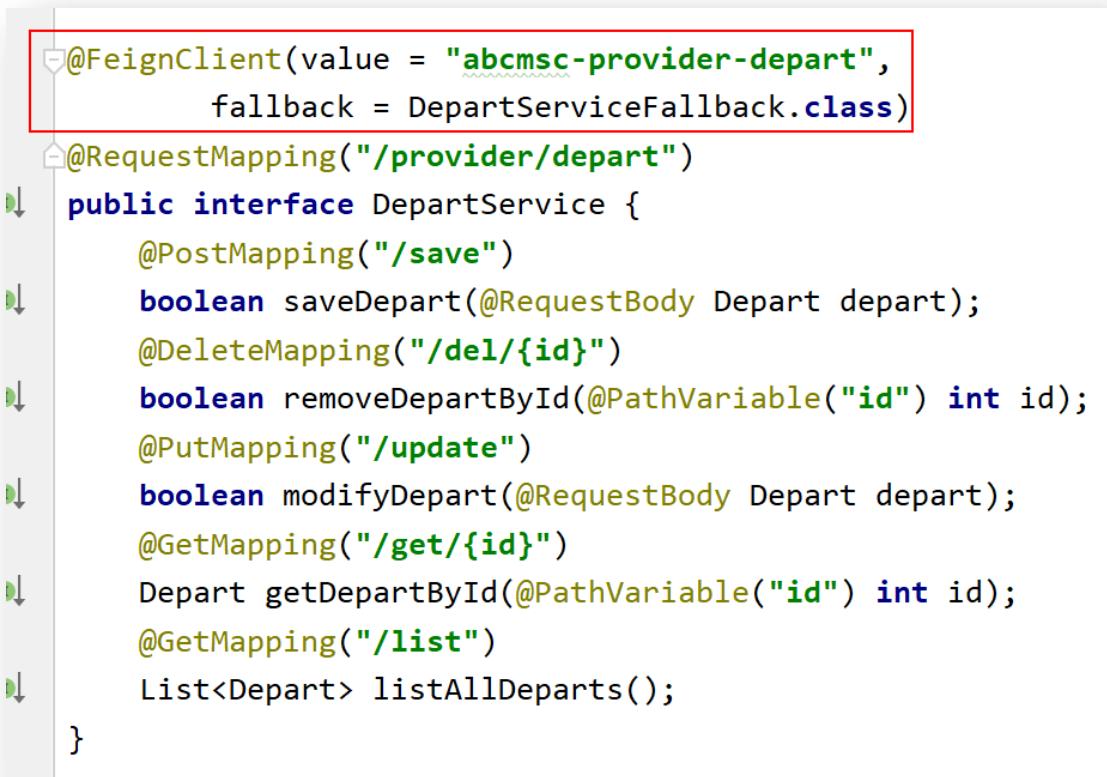
    @Override
    public boolean modifyDepart(Depart depart) {
        System.out.println("执行modifyDepart()的服务降级处理方法");
        return false;
    }
}
```

```
@Override
public Depart getDepartById(int id) {
    System.out.println("执行getDepartById()的服务降级处理方法");
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("degrade-feign");
    return depart;
}

@Override
public List<Depart> listAllDeparts() {
    System.out.println("执行listAllDeparts()服务降级处理方法");
    return null;
}
}
```

## (5) 修改 Feign 接口

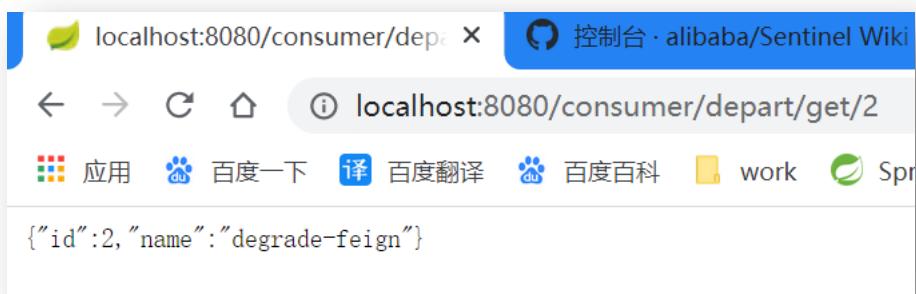
指定要使得的服务降级类。



```
@FeignClient(value = "abcmsc-provider-depart",
    fallback = DepartServiceFallback.class)
@RequestMapping("/provider/depart")
public interface DepartService {
    @PostMapping("/save")
    boolean saveDepart(@RequestBody Depart depart);
    @DeleteMapping("/del/{id}")
    boolean removeDepartById(@PathVariable("id") int id);
    @PutMapping("/update")
    boolean modifyDepart(@RequestBody Depart depart);
    @GetMapping("/get/{id}")
    Depart getDepartById(@PathVariable("id") int id);
    @GetMapping("/list")
    List<Depart> listAllDeparts();
}
```

## (6) 运行访问

不用启动 Provider，直接运行 Consumer，就可以看到降级的结果。



## 6.4 熔断规则

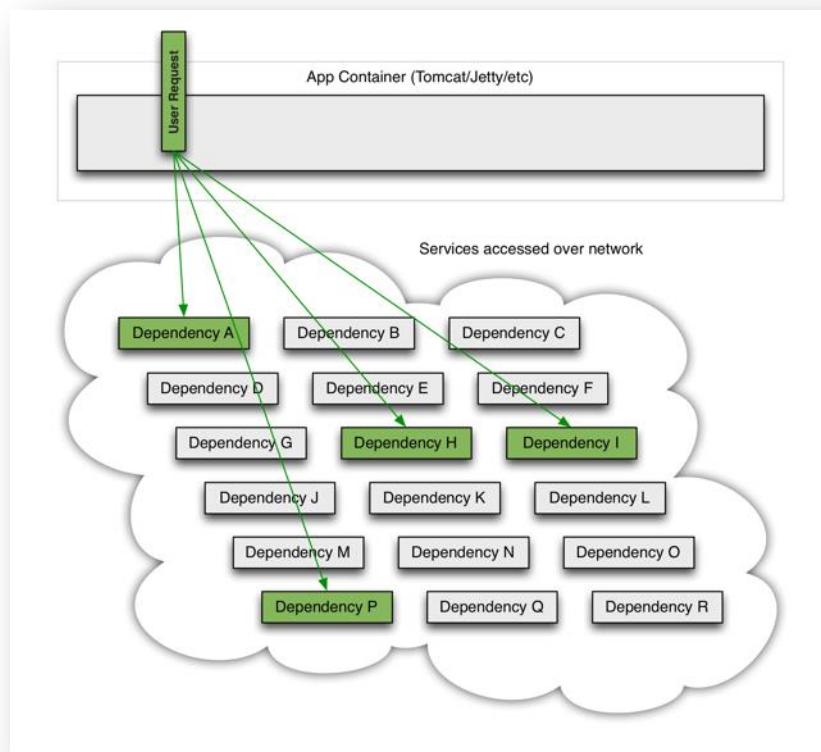
熔断规则用于完成服务熔断。

### 6.4.1 熔断概念

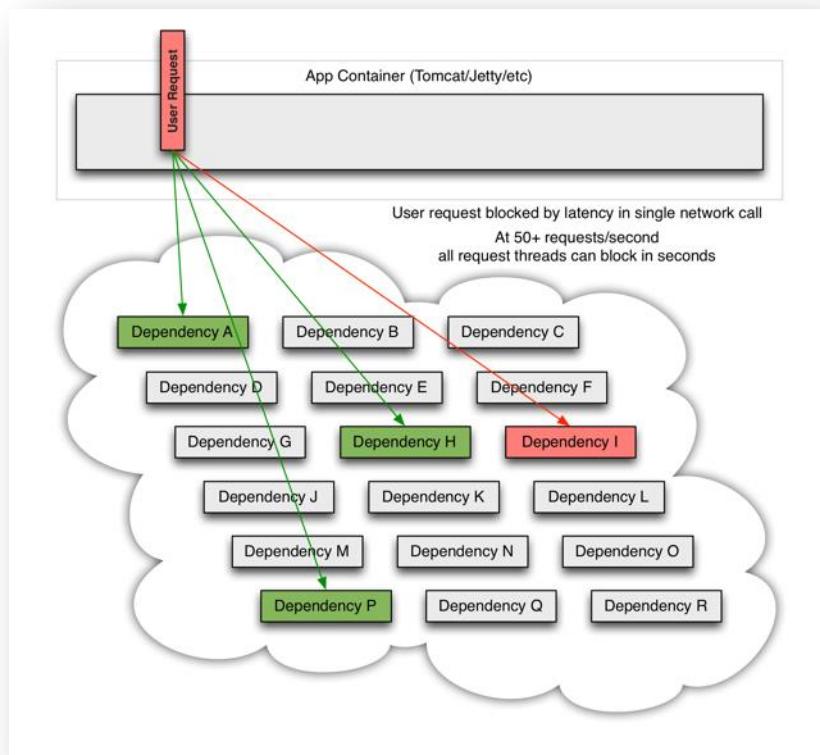
#### (1) 服务雪崩

下面的图都来自于 Hystrix 官网的 wiki 中。

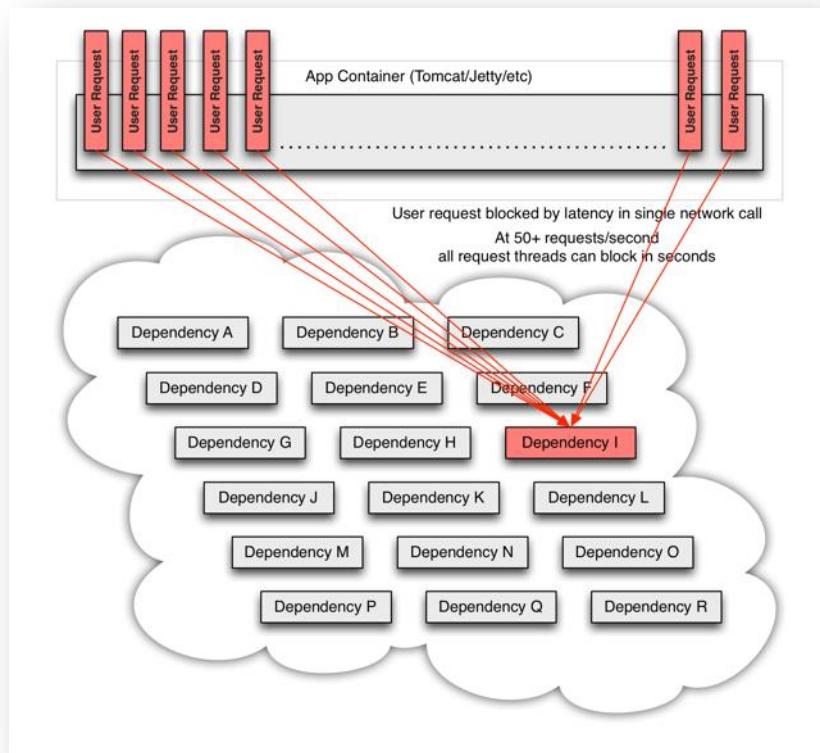
<https://github.com/netflix/hystrix>



上图是用户请求的多个服务（A,H,I,P）均能正常访问并返回的情况。



上图为请求服务 I 出现问题时，一个用户请求被阻塞的情况。



上图为大量用户请求服务 I 出现异常，每个阻塞的请求都会占用一个线程。当并发访问量非常大时，这些请求会迅速用完所有线程，从而导致对于其它正常服务的访问请求无法获取系统资源而被拒绝，发生系统崩溃，即发生服务雪崩。

## (2) 服务熔断

为了防止服务雪崩的发生，在发现了对某些资源请求的响应缓慢或调用异常较多时，直接将对这些资源的请求掐断一段时间。而在这段时间内的请求将不再等待超时，而是直接返回事先设定好的降级结果。这些请求将不占用系统资源，从而避免了服务雪崩的发生。这就是服务熔断。

### 6.4.2 动态熔断配置--默认资源名称

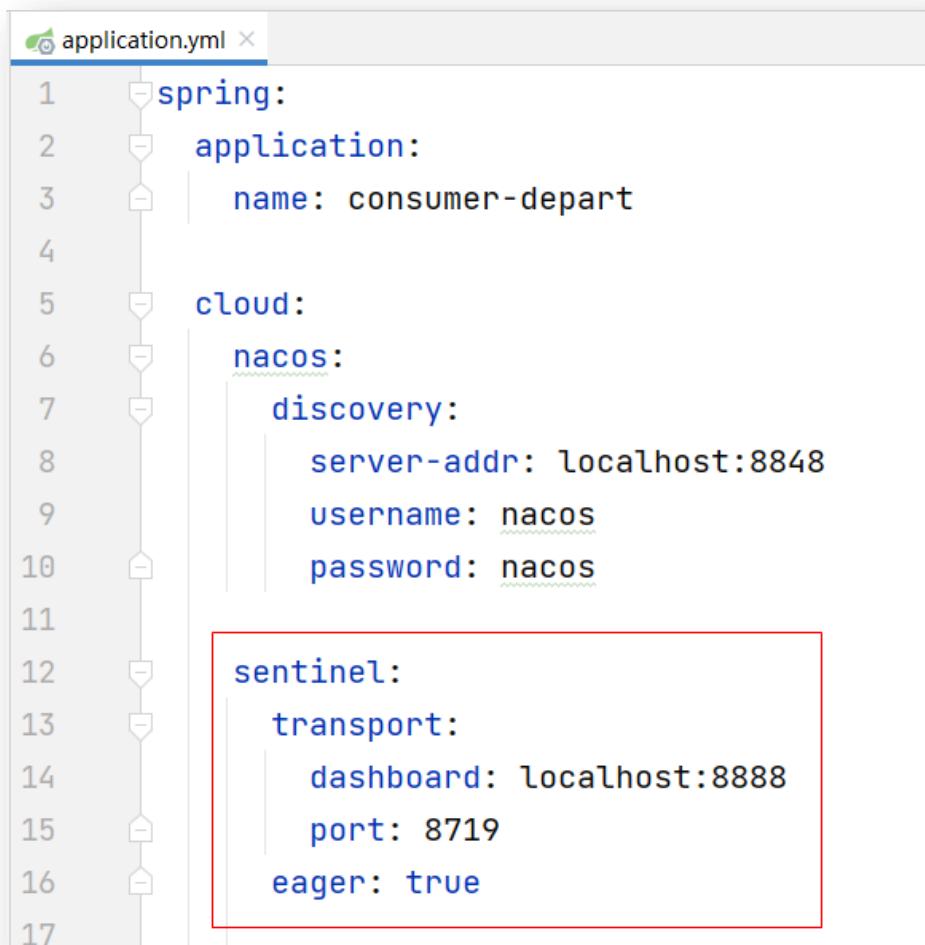
对于服务熔断规则，可以通过 Sentinel Dashboard 进行动态配置。

#### (1) 创建工程

复制 06-consumer-degrade-sentinel-method-8080 工程并重命名为 06-consumer-circuitbreaking-8080。

#### (2) 修改配置文件

在配置文件中添加 Sentinel 控制台的相关配置。



```
application.yml
1 spring:
2   application:
3     name: consumer-depart
4
5   cloud:
6     nacos:
7       discovery:
8         server-addr: localhost:8848
9         username: nacos
10        password: nacos
11
12      sentinel:
13        transport:
14          dashboard: localhost:8888
15          port: 8719
16          eager: true
17
```

### (3) 修改处理器

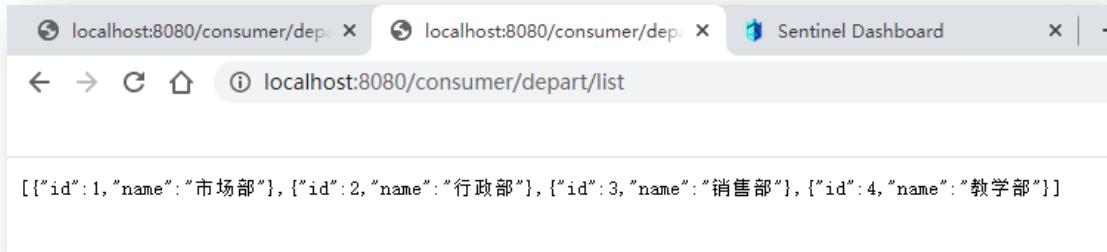
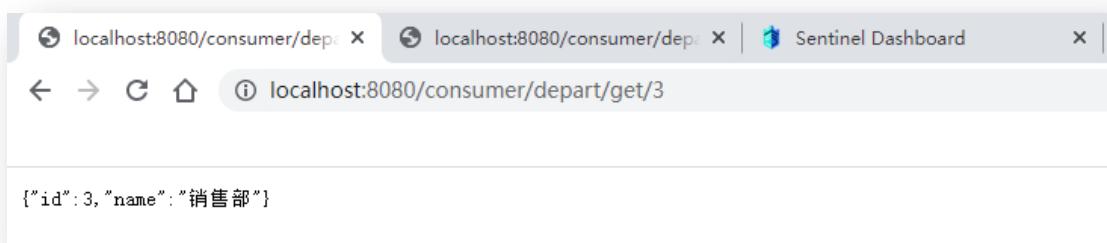
将消费者工程中的处理器中所有的@SentinelResource 注解注释掉。

```
// @SentinelResource(fallback = "getHandleFallback")
no usages
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}

no usages
public Depart getHandleFallback(int id, Throwable t) {
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("sentinel-method-" + id + "-" + t.getMessage());
    return depart;
}
```

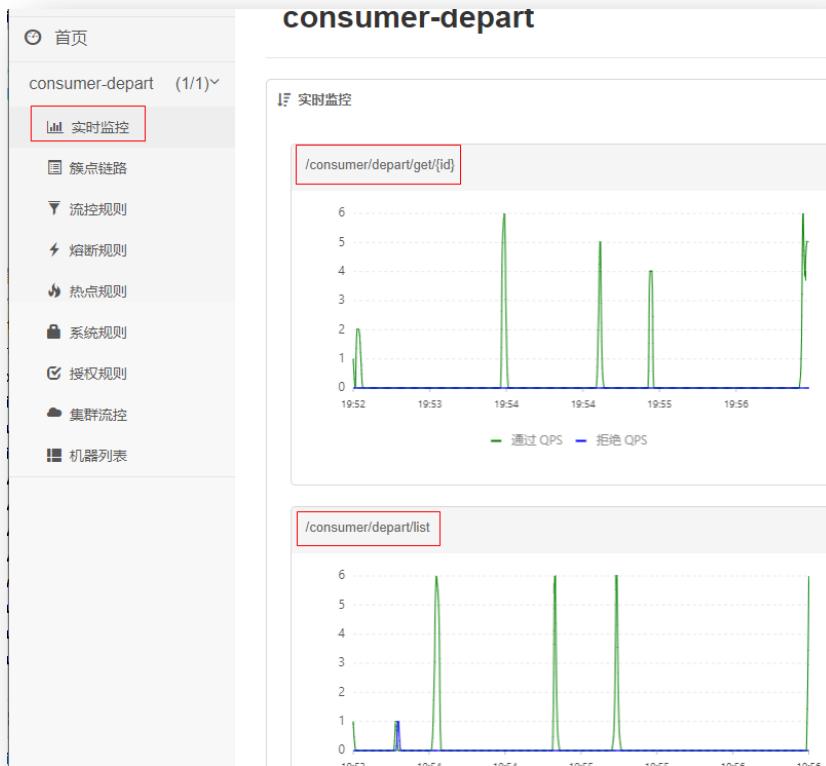
#### (4) 访问

启动 02-provider-nacos-8081 提供者工程与该 06-consumer-circuitbreaking-8080 工程。然后快速访问根据 id 访问与 list 访问。



## (5) dashboard 设置规则

在控制台的“实时监控”处就可以看到对这两个接口进行访问的流量实时数据。



同时在“簇点链路”处可以看到这两个访问接口名（即 URI）成为默认的“资源名”。即默认情况下，对该微服务的访问请求 URI 将作为由 Sentinel 管理的 URI 名称出现。点击 /consumer/depart/get/{id} 中的“熔断”，新增熔断规则。



点击“新增”后，会立即跳转到“熔断规则”页面，并显示出刚新增的规则。



资源名	熔断策略	阈值	熔断时长(s)	操作
/consumer/depart/get/{id}	慢调用比例	2	10s	<button>编辑</button> <button>删除</button>

以同样的方式再为`/consumer/depart/list`新增熔断规则。



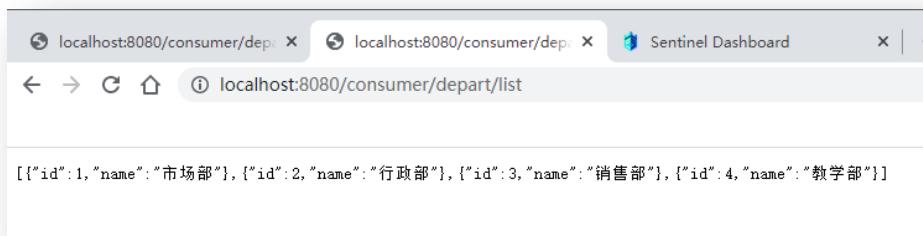
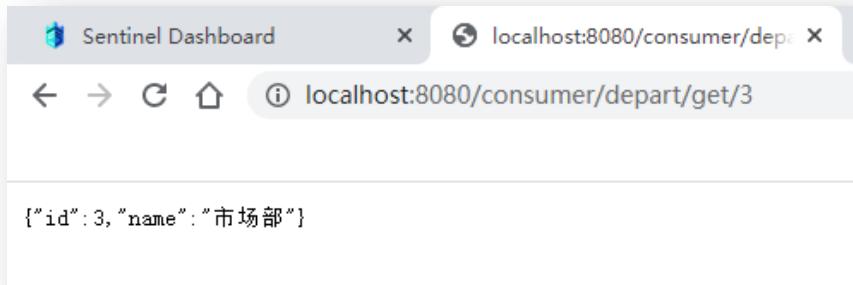
操作
+ 流控 + 熔断 + 热点 + 授权
+ 流控 + 熔断 + 热点 + 授权
+ 流控 + 熔断 + 热点 + 授权
+ 流控 + 熔断 + 热点 + 授权



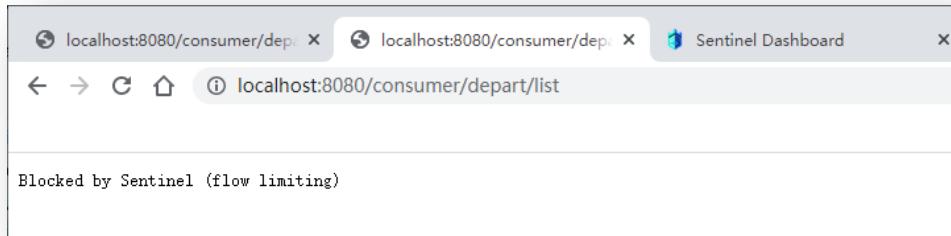
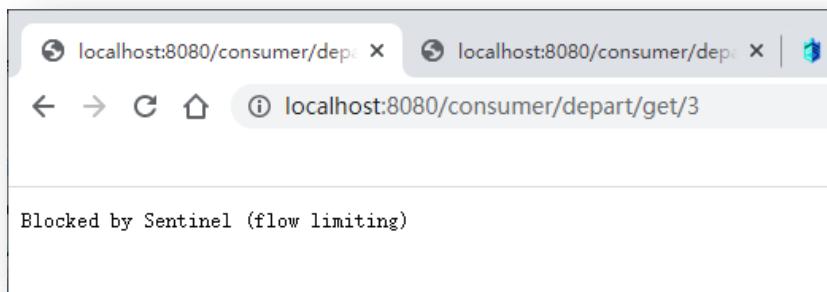
资源名	熔断策略	阈值	熔断时长(s)	操作
/consumer/depart/list	慢调用比例	2	10s	<button>编辑</button> <button>删除</button>
/consumer/depart/get/{id}	慢调用比例	2	10s	<button>编辑</button> <button>删除</button>

## (6) 访问 2

仅提交一次请求正常访问是没有问题的。



但按 F5，快速刷新页面，可以看到其出现了降级。注意要达到每秒 3 次以上的访问才可看到效果。



而且一旦这个降级结果出现，再刷新一次页面也是看不到正常结果的。因为发生了熔断，熔断时长设置的为 30 秒。30 秒后，再刷新一次页面，仍可看到正常结果。

## (7) 再修改处理器

将前面注释掉的@SentinelResource 的注释去掉。

```
no usages
@SentinelResource(fallback = "getHandleFallback")
@GetMapping(PathVariable("/{id}"))
public Depart getHandle(@PathVariable("id") int id) {
    String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}

no usages
public Depart getHandleFallback(int id, Throwable t) {
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("sentinel-method-" + id + "-" + t.getMessage());
    return depart;
}
```

## (8) dashboard 设置规则

再次访问…/get/3 请求后，查看控制台的簇点链路。发现新增了一个新的“资源名”：getHandle()方法的全限定方法签名。即只要在处理器方法上添加了@SentinelDashboar 注解，那么该处理器方法就具有了默认的 Sentinel 资源名称：该方法的全限定性方法签名。

Sentinel 控制台 1.8.6

注销

用户名 搜索

首页 consumer-depart (1/1) 实时监控 线点链路 **线点链路** 流控规则 熔断规则 热点规则 系统规则 授权规则 集群流控 机器列表

### consumer-depart

树状视图 列表视图

簇点链路	资源名	通过QPS	拒绝QPS	并发数	平均RT	分钟通过	分钟拒绝	操作
sentinel_default_context	0 0 0 0 0 0	+ 流控 + 熔断 + 热点 + 授权						
sentinel_spring_web_context	0 0 0 0 0 0	+ 流控 + 熔断 + 热点 + 授权						
/consumer/depart/get/{id}	0 0 0 0 0 0	+ 流控 + 熔断 + 热点 + 授权						
com.abc.controller.DepartController.getHandle(int)	0 0 0 0 0 0	+ 流控 + 熔断 + 热点 + 授权						

共 4 条记录, 每页 16 条记录

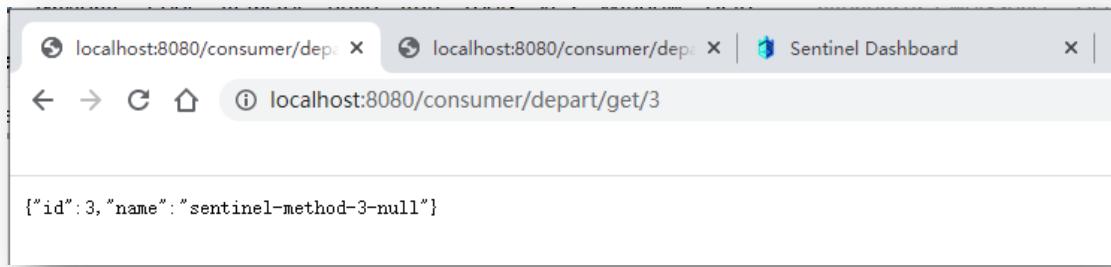
当然，点击其对应的“熔断”按钮同样也可以为其设置熔断规则。

新增熔断规则

资源名	com.abc.controller.DepartController:getHandle(int)		
熔断策略	<input checked="" type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input type="radio"/> 异常数		
最大 RT	2	比例阈值	0.2
熔断时长	10 s	最小请求数	2
统计时长	1000 ms		
<input type="button" value="新增并继续添加"/> <input type="button" value="新增"/> <input type="button" value="取消"/>			

## (9) 访问 3

仅提交一次请求正常访问是没有问题的。但按 F5，快速刷新页面，可以看到其出现了降级。但这次的降级结果则是降级方法执行的结果。



### 6.4.3 动态熔断配置--指定资源名称

除了默认的名称外，还可以为处理器方法指定名称。

直接在 06-consumer-circuitbreaking-8080 工程中进行修改。

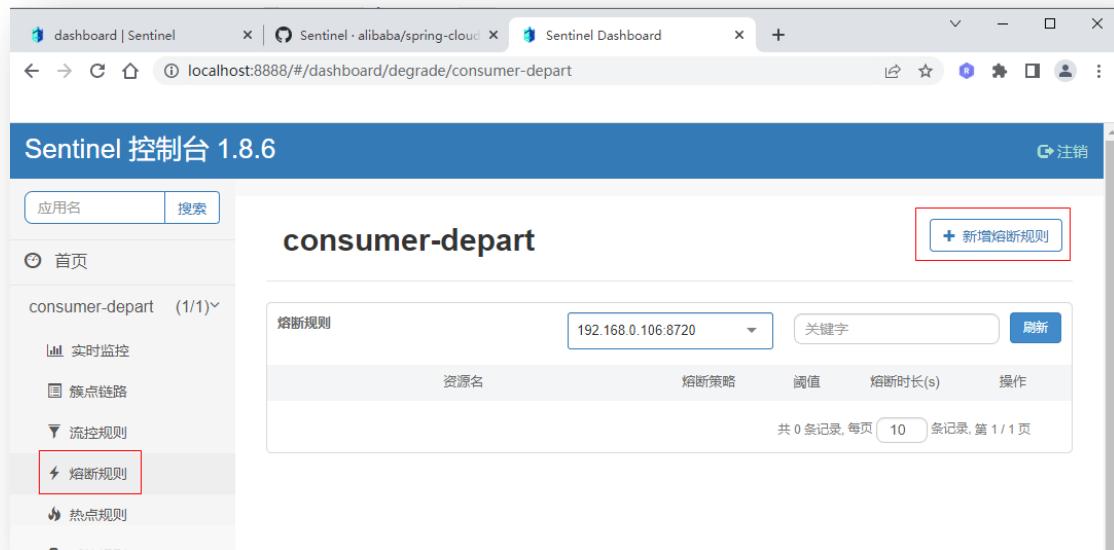
#### (1) 修改控制器类

修改 DepartController 中的 getHandle() 方法上的 @SentinelResource 注解，通过 value 属性指定当前 getHandler() 方法作为 Sentinel 资源时的名称。注意，此时并没有指定降级方法。

```
no usages
@SentinelResource("getHandle")
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}
```

#### (2) dashboard 设置规则

在启动了消费者工程后，再设置 Sentinel 控制台。



Sentinel 控制台 1.8.6

consumer-depart (1/1)

+ 新增熔断规则



新增熔断规则

资源名: getHandle

熔断策略: 慢调用比例

最大 RT: 2

比例阈值: 0.2

熔断时长: 10 s

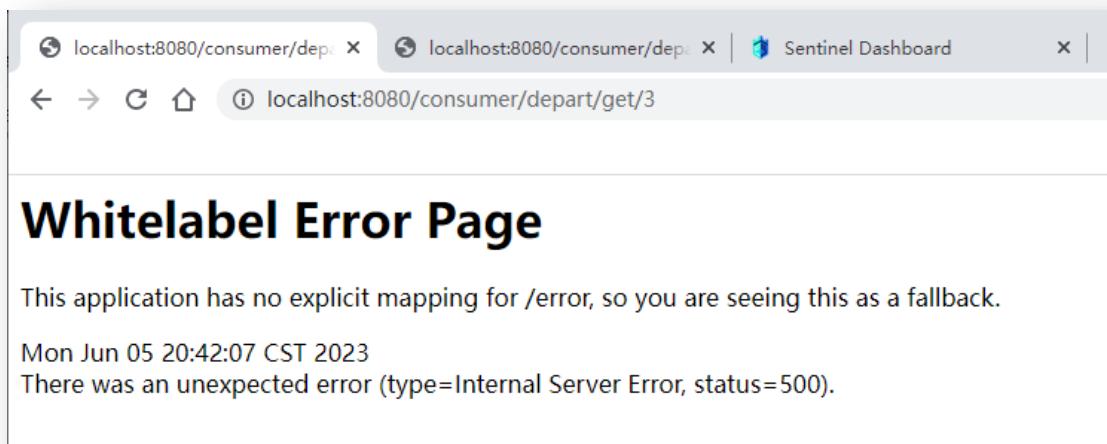
最小请求数: 2

统计时长: 1000 ms

新增 取消

### (3) 访问

仅提交一次请求正常访问是没有问题的。但按 F5 快速刷新页面可以看到发生熔断。只不过，返回的页面是 500，不是原来给出的“Blocked By Sentinel”。



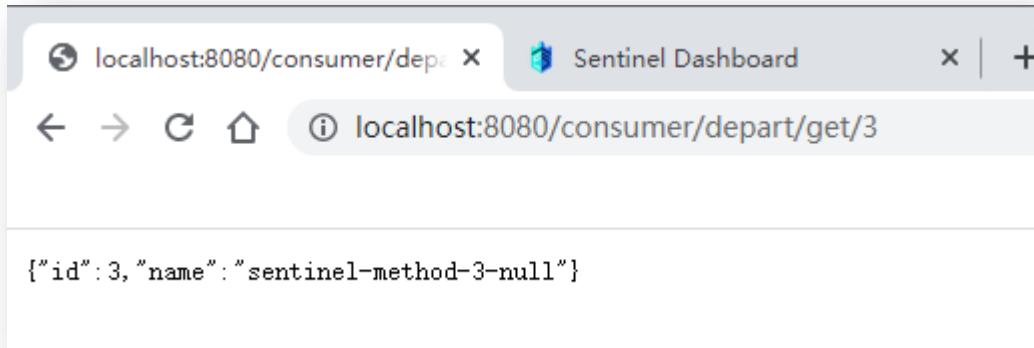
#### (4) 再修改控制器类

为@SentinelResource 指定其要使用的降级方法。

```
no usages
@SentinelResource(value = "getHandle", fallback = "getHandleFallback")
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}
```

#### (5) 访问 2

按 F5 快速刷新页面可以看到发生降级，且降级结果为降级方法执行结果。



#### 6.4.4 API 熔断设置

通过 API 设置熔断规则，即直接将熔断规则定义在代码中的，在应用启动时完成熔断规则的创建与初始化。

为了保证项目运行的安全性，一般会在代码中通过 API 方式设置默认的熔断规则，以应对平时对系统平台的保护要求。不过，API 熔断规则在 Dashboard 中也是可以查看到并且进行编辑的，编辑后以动态编辑的规则为准。

##### (1) 修改启动类

直接在 06-consumer-circuitbreaking-8080 工程中进行修改。在启动类中添加熔断规则定义。

```
2 usages
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
        initRule();
    }

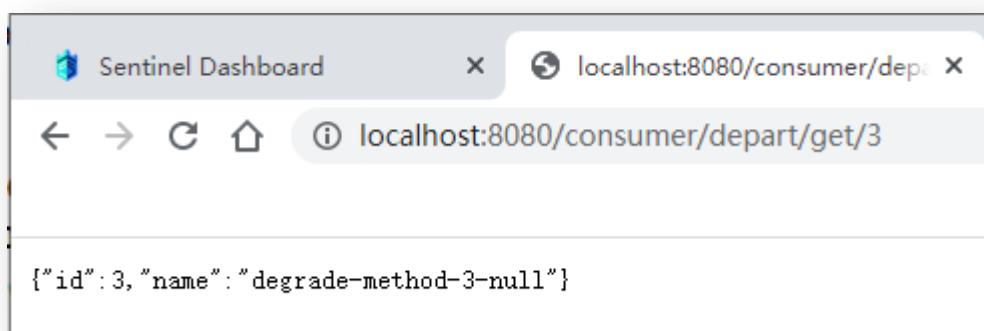
    1 usage
    private static void initRule() {
        List<DegradeRule> degradeRules = new ArrayList<>();
        DegradeRule degradeRule = Consumer8080.configSlowDegradeRule();
        degradeRules.add(degradeRule);
        DegradeRuleManager.loadRules(degradeRules);
    }
}
```

```
1 usage
private static DegradeRule configSlowDegradeRule() {
    DegradeRule degradeRule = new DegradeRule();
    degradeRule.setResource("getHandle");
    degradeRule.setGrade(RuleConstant.DEGRADE_GRADE_RT);
    degradeRule.setCount(200);
    degradeRule.setSlowRatioThreshold(0.3);
    degradeRule.setTimeWindow(10);
    degradeRule.setStatIntervalMs(1000);
    degradeRule.setMinRequestAmount(3);
    return degradeRule;
}

no usages
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

## (2) 访问

启动 06-consumer-circuitbreaking-8080 工程，在浏览器中按 F5 快速访问，发现同样也是可以实现熔断的。



### (3) 修改控制台熔断规则

刷新 Sentinel 控制台页面，发现熔断规则又出现了。这次出现是因为代码中的 API 熔断规则被加载到了控制台。点击该规则的“编辑”按钮，对熔断规则进行修改，使其不可能在通过刷新页面这种速度情况下达到熔断。以说明控制台的动态配置优先级要高于 API 设置的。



保存后再 F5 刷新页面，发现已经不会出现熔断了。说明控制台的熔断规则动态配置优先级的确高于 API 设置的。

## 6.5 自定义异常处理器

当采用 URI 作为默认 Sentinel 资源名称，且发生打破规则阈值的异常情况时，Sentinel 会抛出一个 BlockExcetion 异常，而该异常会被 BlockedExceptionHandler 异常处理器处理。

## 6.5.1 返回响应流

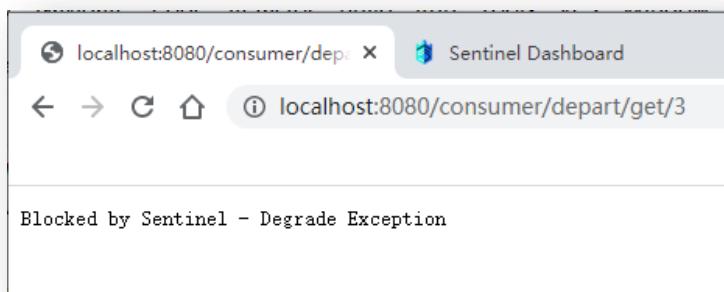
### (1) 定义处理器

```
@Component
public class CustomBlockExceptionHandler implements BlockExceptionHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
                        BlockException e) throws Exception {

        response.setStatus(429);
        PrintWriter out = response.getWriter();
        String msg = "Blocked by Sentinel - ";
        if(e instanceof FlowException) {
            msg += "Flow Exception";
        } else if(e instanceof DegradeException) {
            msg += "Degrade Exception";
        } else if(e instanceof SystemBlockException) {
            msg += "System Block Exception";
        } else if(e instanceof ParamFlowException) {
            msg += "Param Flow Exception";
        } else if(e instanceof AuthorityException) {
            msg += "Authority Exception";
            response.setStatus(401);
        }
        out.print(msg);
        out.flush();
        out.close();
    }
}
```

### (2) 访问结果

当发生相应规则异常时，会由自定义的异常处理器来处理。例如，对于熔断异常的处理结果显示为：

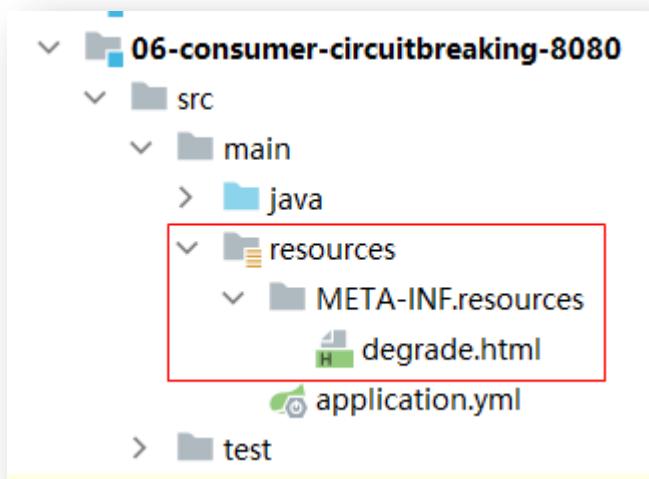


## 6.5.2 跳转到 html 页面

若感觉返回响应流的内容过于简单，也可将其跳转到相应的指定页面。

### (1) 定义页面

在工程的 resources 目录下新建 META-INF/resources 目录，在其中新建一个 degrade.html 文件。



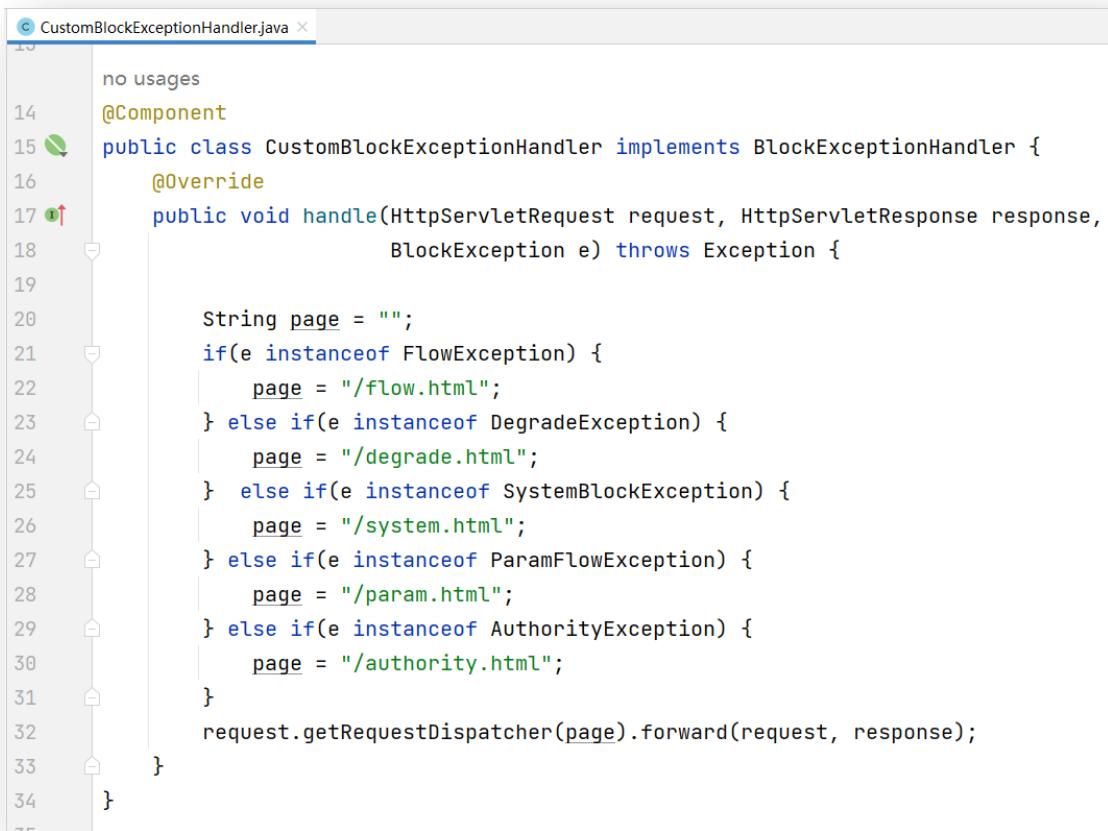
degrade.html 文件内容如下：



```
degrade.html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>degrade</title>
6 </head>
7 <body>
8     发生熔断异常
9 </body>
10 </html>
```

再如法炮制定义出 flow.html、system.html、param.html 与 authority.html。

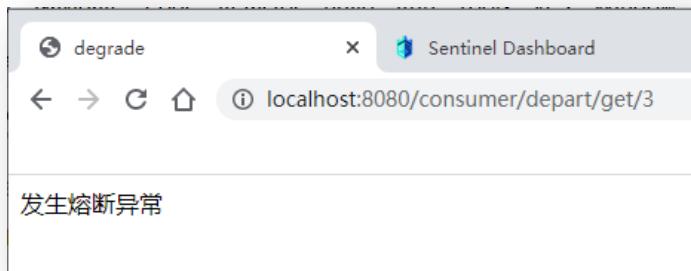
## (2) 定义处理器



```
CustomBlockExceptionHandler.java
13 no usages
14
15 @Component
16 public class CustomBlockExceptionHandler implements BlockExceptionHandler {
17     @Override
18     public void handle(HttpServletRequest request, HttpServletResponse response,
19                         BlockException e) throws Exception {
20
21         String page = "";
22         if(e instanceof FlowException) {
23             page = "/flow.html";
24         } else if(e instanceof DegradeException) {
25             page = "/degrade.html";
26         } else if(e instanceof SystemBlockException) {
27             page = "/system.html";
28         } else if(e instanceof ParamFlowException) {
29             page = "/param.html";
30         } else if(e instanceof AuthorityException) {
31             page = "/authority.html";
32         }
33         request.getRequestDispatcher(page).forward(request, response);
34     }
35 }
```

### (3) 访问结果

当发生相应规则异常时，会由自定义的异常处理器来处理。例如，对于熔断异常的处理结果显示为：



#### 6.5.3 重定向到 URL

当系统采用前后端分离模式时使用页面跳转是不行的，此时可以使用重定向来完成。

### (1) 定义处理器

```
@Component
public class CustomBlockExceptionHandler implements BlockExceptionHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
                       BlockException e) throws Exception {

        String url = "";
        if(e instanceof FlowException) {
            url = "https://jd.com";
        } else if(e instanceof DegradeException) {
            url = "https://taobao.com";
        } else if(e instanceof SystemBlockException) {
            url = "https://baidu.com";
        } else if(e instanceof ParamFlowException) {
            url = "https://oracle.com";
        } else if(e instanceof AuthorityException) {
            url = "https://sentinelguard.io";
        }
        response.sendRedirect(url);
    }
}
```

## (2) 访问结果

当发生相应规则异常时，会由自定义的异常处理器来处理。例如，对于熔断异常的处理结果是，直接跳转到了淘宝官网。

## 6.6 流控规则

### 6.6.1 简介

流控规则是用于完成服务流控的。服务流控即对访问流量的控制，也称为服务限流。Sentinel 实现流控的原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对再到来的请求进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。

### 6.6.2 动态流控

流控规则直接通过 Sentinel Dashboard 定义，该规则可以随时修改而不需要重启应用。所以这种流控是一种动态流控。

#### (1) 定义工程

复制 06-consumer-circuitbreaking-8080 工程，重命名为 06-consumer-flowcontrol-8080。

#### (2) dashboard 设置流控规则

在启动了消费者工程后，再设置 Sentinel 控制台。

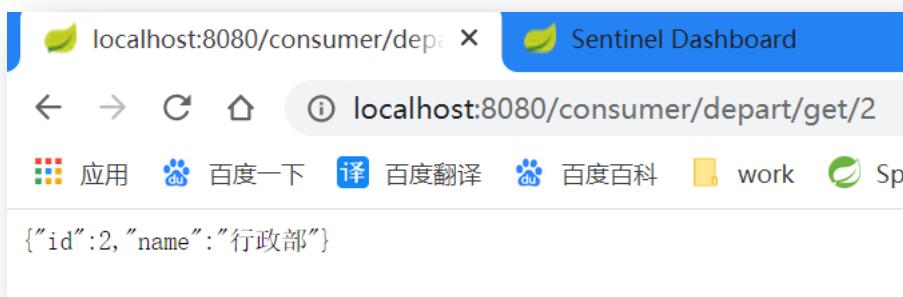


The screenshot shows the Sentinel Control Panel interface. At the top, it says "Sentinel 控制台 1.8.6". On the left, there's a sidebar with "应用名" (Application Name) input, a "搜索" (Search) button, and navigation links: "首页" (Home), "consumer-depart (1/1)" (selected), "实时监控" (Real-time Monitoring), "簇点链路" (Cluster Point Links), "流控规则" (Flow Control Rules) (which is expanded), and "熔断规则" (Circuit Breaker Rules). In the center, the title is "consumer-depart". Below it, there's a search bar with "流控规则" (Flow Control Rules) and a dropdown set to "192.168.0.106:8720". To the right of the search bar is a "关键字" (Keyword) input and a "刷新" (Refresh) button. A large red box highlights the "新增流控规则" (Add Flow Control Rule) button. At the bottom, there's a table header for "资源名" (Resource Name), "来源应用" (Source Application), "流控模式" (Flow Control Mode), "阈值类型" (Threshold Type), "阈值" (Threshold), "阈值模式" (Threshold Mode), "流控效果" (Flow Control Effect), and "操作" (Operation). Below the table, it says "共 0 条记录, 每页 10 条记录".

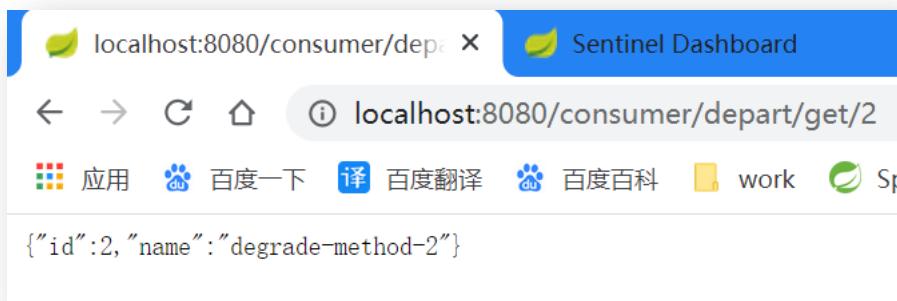


### (3) 访问 1

仅刷新一下浏览器，可以看到正常的结果。



多次刷新页面，可以看到降级的结果。



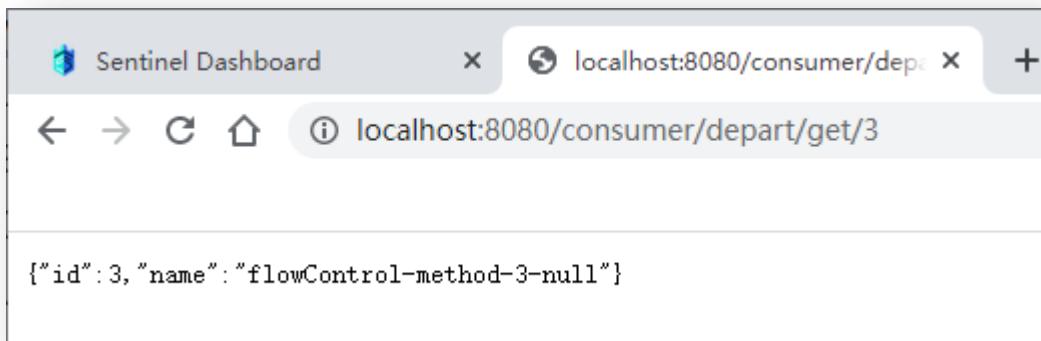
#### (4) 修改处理器

在 @SentinelResource 中添加 blockHandler 属性，用于指定流控处理器方法，并在当前控制器类中定义流控处理器方法。

```
no usages
@SentinelResource(value = "getHandle",
    blockHandler = "getHandleFlowControl",
    fallback = "getHandleFallback")
@GetMapping("/get/{id}")
public Depart getHandle(@PathVariable("id") int id) {
    String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}

// 定义getHandle()的流控降级方法
no usages
public Depart getHandleFlowControl(int id, BlockException be) {
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("flowControl-method-" + id + "-" + be.getMessage());
    return depart;
}
```

## (5) 访问 2



## 6.6.3 API 流控

流控规则直接定义在代码中，当应用启动时完成流控规则的创建与初始化。这个流控规则在 **Dashboard** 中也是可以查看到并且进行编辑的，编辑后以动态编辑的规则为准。

## (1) 修改启动类

直接在 06-consumer-flowcontrol-8080 工程中进行修改。在启动类中初始化流控规则。



```
2 usages
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
        initFlowRule();
    }

    1 usage
    private static void initFlowRule() {
        List<FlowRule> rules = new ArrayList<>();

        FlowRule rule = Consumer8080.configFlowRule();
        rules.add(rule);
        FlowRuleManager.loadRules(rules);
    }
}
```

```

1 usage
private static FlowRule configFlowRule() {
    FlowRule rule = new FlowRule();
    rule.setResource("getHandle");
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setCount(3);
    rule.setLimitApp("default");
    return rule;
}

no usages
@LoadBalanced
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
}

```

## (2) 访问

启动应用后，刷新 dashboard 可以看到代码中定义的流控规则。



The screenshot shows the 'consumer-depart' application's flow control rules interface. On the left sidebar, under the 'consumer-depart' section, the '流控规则' (Flow Control Rules) option is selected. The main area displays a table of flow control rules:

资源名	来源应用	流控模式	阈值类型	阈值	流控效果	操作
getHandle	default	直接	QPS	3	单机 快速失败	<button>编辑</button> <button>删除</button>

At the top right of the main area, there is a search bar with placeholder '关键字' (Keyword) and a '刷新' (Refresh) button. The top left corner has a '用户名' (Username) input field and a '搜索' (Search) button.

## (3) 修改控制台流控规则

将 QPS 的阈值调整为手动无法完成的刷新频率。保存后再 F5 刷新页面，发现已经不会出现流控降级了。说明控制台的流控规则动态配置优先级的确高于 API 设置的。

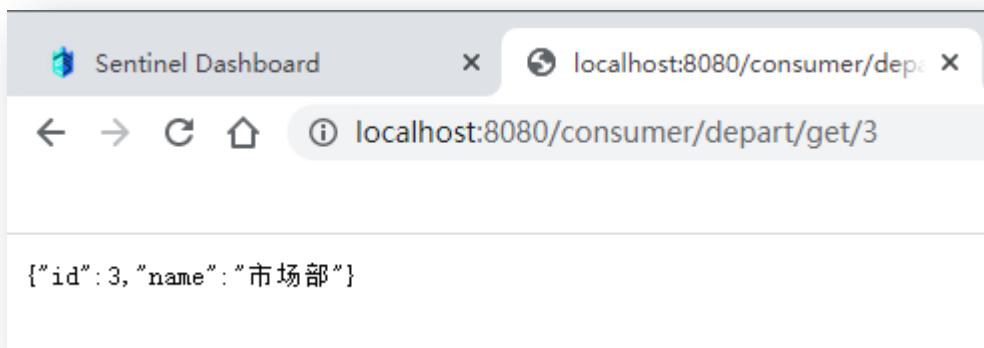
流控规则		192.168.0.106:8720		关键字		刷新	
资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
getHandle	default	直接	QPS	3	单机	快速失败	<span style="border: 1px solid red; padding: 2px;">编辑</span> <span style="border: 1px solid red; padding: 2px;">删除</span>
共 1 条记录, 每页 10 条记录							

编辑流控规则

资源名	getHandle
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
是否集群	<input type="checkbox"/>
高级选项	
<span style="border: 1px solid green; padding: 2px;">保存</span> <span style="border: 1px solid red; padding: 2px;">取消</span>	

#### (4) 访问 2

在浏览器 F5 快速刷新，其也是正常运行结果。因为手工刷新速度达不到设定的每秒 100 次，无法启动流控。



#### 6.6.4 资源实体指定流控规则

当一个处理器方法在使用为其自身打造的各种流控规则的同时，还想使用应用到另一个资源的所有规则时，可以使用资源实体来指定。

##### (1) 定义工程

复制 06-consumer-flowcontrol-8080 工程，重命名为 06-consumer-flowcontrol2-8080。

##### (2) 修改启动类

将启动类中所有规则相关的代码全部删除，这里使用动态规则配置方式。当然，使用 API 方式是最好的。

##### (3) 修改处理器

首先在处理器中为 `listHandle()` 方法添加流控阻断方法，并将 `listHandle()` 方法指定为 `Sentinel` 资源。

```
no usages
@SentinelResource(value = "listHandle",
                   blockHandler = "listHandleFlowControl")
@GetMapping("/list")
public List<Depart> listHandle() {
    String url = PROVIDER_SERVER + "/provider/depart/list";
    return restTemplate.getForObject(url, List.class);
}

// listHandle() 的流控阻断方法
no usages
public List<Depart> listHandleFlowControl(BlockException be) {
    List<Depart> list = new ArrayList<>();
    Depart depart = new Depart();
    depart.setName("listHandle-flowControl");
    list.add(depart);
    return list;
}
```

然后在 `getHandle()` 方法中使用资源实体方式指定其要应用 `listHandle` 资源的规则。

```
no usages
@SentinelResource(value = "getHandle",
                   fallback = "getHandleFallback")
@GetMapping(@RequestMapping("/get/{id}"))
public Depart getHandle(@PathVariable("id") int id) {
    Entry entry = null;
    try {
        // 加载指定资源实体的流控规则
        entry = SphU.entry("listHandle");
        // 未被流控阻断时执行的代码
        String url = PROVIDER_SERVER + "/provider/depart/get/" + id;
        return restTemplate.getForObject(url, Depart.class);
    } catch (BlockException e) {
        // 发生流控阻断时，捕获阻断异常
        Depart depart = new Depart();
        depart.setId(id);
        depart.setName("flowControl-entry-" + id);
        return depart;
    } finally {
        // 结束资源实体规则
        if (entry != null) {
            entry.exit();
        }
    }
}
```

```
// getHandle()熔断降级方法
no usages
public Depart getHandleFallback(int id) {
    Depart depart = new Depart();
    depart.setId(id);
    depart.setName("sentinel-method-" + id);
    return depart;
}
```

#### (4) 控制台添加降级规则

在控制台为 getHandle 资源添加熔断规则。



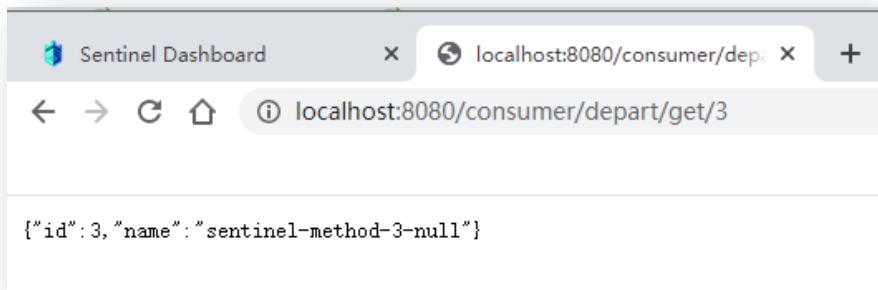
#### (5) 控制台添加流控规则

在控制台为 listHandle 资源添加流控规则。



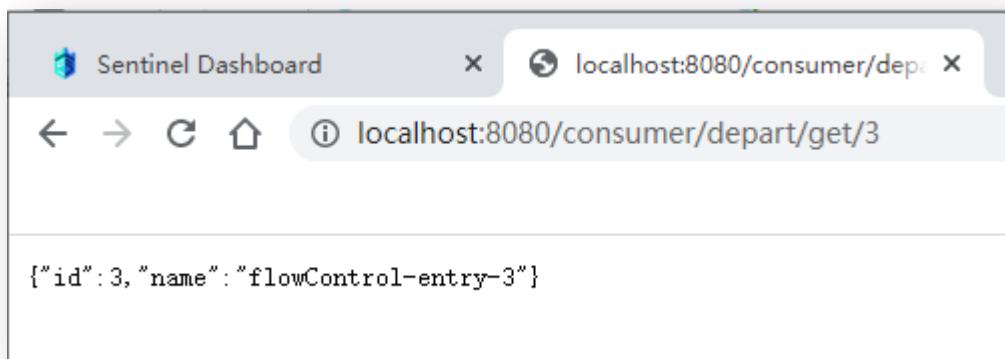
## (6) 访问 1

启动 04-provider-nacos-8081 工程，因为该工程中的 `DepartServiceImpl.getDepartById()` 方法中具有 `sleep()` 方法，会出现慢调用问题。快速刷新页面，可以看到请求被熔断降级的情况。但不会出现被限流的情况，这主要是因为熔断规则中设置的熔断时长引起的。



## (7) 访问 2

启动 02-provider-nacos-8081 工程，因为该工程中的 `DepartServiceImpl.getDepartById()` 方法中没有 `sleep()` 方法，不会出现慢调用问题。快速刷新页面，可以看到请求被阻断的情况。当然，这里不会出现熔断降级的情况，因为不存在慢调用。

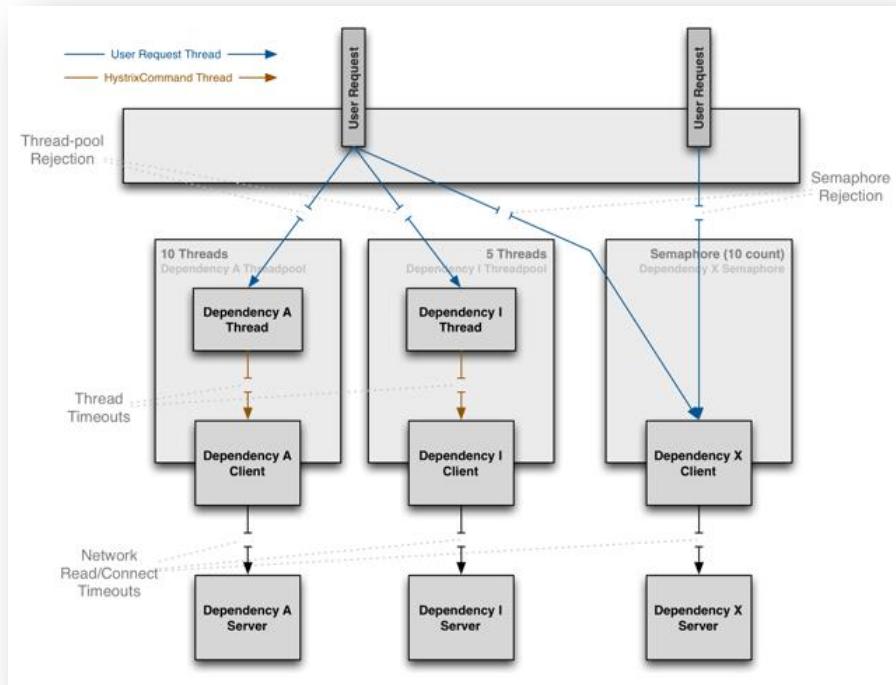


## 6.6.5 并发线程数流控方案

流量控制有两种阈值统计类型，除了 QPS 外，另一种是统计并发线程数。

### (1) 线程隔离方案

并发线程数流控方案通常是对消费者端的配置。是为了避免由于慢调用而将消费者端线程耗尽情况的发生，业内会使用线程隔离方案。一般来说可以分为两种：



## A、线程池隔离

系统为不同的提供者资源设置不同的线程池来隔离业务自身之间的资源争抢。该方案隔离性较好，但需要创建的线程池及线程数量太多，系统消耗较大。当请求线程到达后，会从线程池中获取到一个新的执行线程去完成提供者的调用。由请求线程到执行线程的上下文切换时间开销较大。特别是对低延时的调用有比较大的影响。

## B、信号量隔离

系统为不同的提供者资源设置不同的计数器。每增加一个该资源的调用请求，计数器就变化一次。当达到该计数器阈值时，再来的请求将被限流。该方式的执行线程与请求线程是同一个线程，不存在线程上下文切换的问题，更不存在很多的线程池创建与线程创建问题。也正因为请求线程与执行线程没有分离，所以，其对于提供者的调用无法实现异步，执行效率降低，且对于依赖资源的执行超时控制不方便。

## (2) Sentinel 线程隔离方案



Sentinel 并发线程数控制也属于隔离方案，但不同于以上两种隔离方式，是对以上两种方案的综合与改进，或者说更像是线程池隔离。

其也将请求线程与执行线程进行了分离，但 Sentinel 不负责创建和管理线程池，而仅仅是简单统计当前资源请求占用的线程数目。如果对该资源的请求占用的线程数量超出了阈值，则可以立即拒绝再新进入的请求。

### 6.6.6 直接流控模式

#### (1) 运行原理



当对“资源名”指定资源的请求达到了设置阈值时，再新进入的请求将被直接执行指定的“流控效果”。

#### (2) 任务描述

前面的例子默认采用的就是直接流控模式，所以这里就不再举例了。

### 6.6.7 关联流控模式

#### (1) 运行原理

该模式比较特殊：当对别人的访问达到自己设置的阈值时，将开启对自己的限流。确切

地说是，当对“关联资源”的访问达到了“单机阈值”指定的阈值时，会对当前的资源访问进行限流。

## (2) 修改处理器

直接修改 06-consumer-flowcontrol-8080 工程的 DepartController 类，为 listHandle()方法添加 @SentinelResource 注解，将该方法指定为 Sentinel 管理的资源。

```
no usages
@SentinelResource("listHandle")
@GetMapping("/list")
public List<Depart> listHandle() {
    String url = PROVIDER_SERVER + "/provider/depart/list";
    return restTemplate.getForObject(url, List.class);
}
```

## (3) 修改流控规则

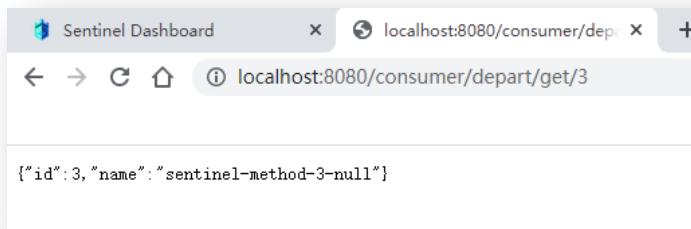
由于在项目代码中通过 API 方式设置了 getHandle 流控规则，所以在 Sentinel 控制台中是可以看到有 getHandle 规则的。将该规则的流控模式修改为“关联”，并将前面新增的 listHandle 流控规则指定为“关联资源”。保持阈值仍为 3。该阈值的意义为：当对关联资源 listHandle 的访问 QPS 超过 3 时，就会对 getHandle 资源进行限流。

编辑流控规则

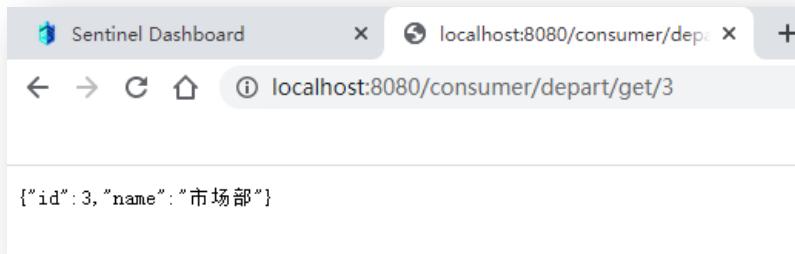
资源名	getHandle
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数      单机阈值 <input type="text" value="3"/>
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路
关联资源	listHandle
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待
<a href="#">关闭高级选项</a>	
<a href="#">保存</a> <a href="#">取消</a>	

#### (4) 浏览器访问

在 Postman 中提交 `http://localhost:8080/consumer/depart/list` 请求，运行期间，马上访问 `getHandle` 资源，可以看到其被限流了。



当 Postman 执行完毕，其又可正常访问了。



## 6.6.8 链路流控模式

### (1) 运行原理

当对一个资源有多种访问路径时，可以对某一路径的访问进行限流，而其它访问路径不限流。

### (2) 创建工程 06-provider-flowcontrol-8081

复制 02-provider-nacos-8081 工程，重命名为 06-provider-flowcontrol-8081。

### (3) 添加依赖

添加 sentinel 依赖。

```
<!--sentinel 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

### (4) 修改业务接口实现类

这次将@SentinelResource 添加到了 Service 接口实现类的 listAllDeparts()方法上，并定义一个降级方法。

2 usages

```
@SentinelResource(value = "listAllDeparts",
                   fallback = "listAllDepartsFallback")  
  
{@Override  
public List<Depart> listAllDeparts() {  
    return repository.findAll();  
}}
```

no usages

```
public List<Depart> listAllDepartsFallback() {  
    List<Depart> list = new ArrayList<>();  
    Depart depart = new Depart();  
    depart.setName("no any depart");  
    list.add(depart);  
    return list;  
}
```

## (5) 修改 Controller

在 Controller 中添加一个方法，保证有两个接口可以访问到 service.listAllDeparts() 方法。

```
no usages
@GetMapping("list")
public List<Depart> listHandle() {
    return service.listAllDeparts();
}

no usages
@GetMapping("/all")
public List<Depart> allHandle() {
    return service.listAllDeparts();
}
```

## (6) 修改配置文件

在配置文件中添加 sentinel-dashboard 配置，并关闭链路收敛功能。

```
17
18     application:
19         name: provider-depart
20     cloud:
21         nacos:
22             discovery:
23                 server-addr: localhost:8848
24
25         sentinel:
26             transport:
27                 dashboard: localhost:8888
28                 port: 8719
29                 eager: true
30                 web-context-unify: false
31
```

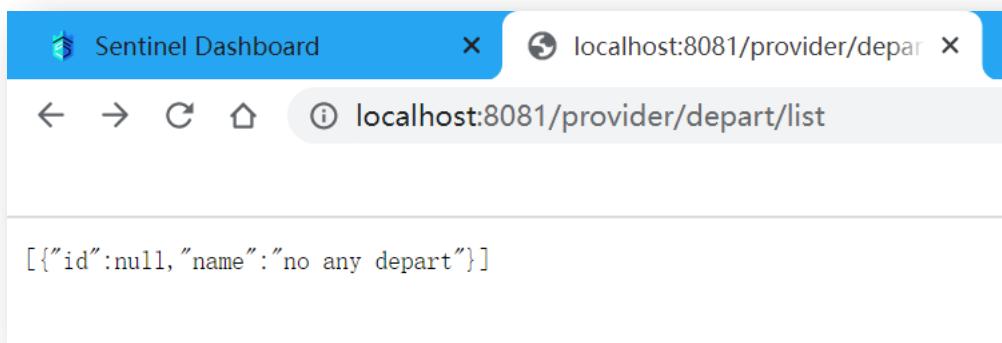
## (7) 新增流控规则

在 Sentinel 控制台的 provider 工程中新增流控规则。注意，“入口资源”必须是要限流的完整的 URI，即必须是 URL 中 port 后的全部内容。以下规则中仅对/provider/depart/list 进行限流，而对/provider/depart/all 不限流。



## (8) 访问

对于请求 <http://localhost:8081/provider/depart/list>，快速刷新出现限流。



对于请求 <http://localhost:8081/provider/depart/all>, 不会限流。



## 6.6.9 流控效果分类

### (1) 快速失败



快速失败，也称为直接拒绝，是默认的 QPS 流控超值处理方式。当 QPS 超过设置的阈值后，再来的请求将被直接拒绝或降级。

## (2) Warm Up



当系统中某 service 的 QPS 长期处于低水位运行状态时，系统为该 service 所分配的各种软硬件资源都会很少，例如，缓存空间、线程数量等资源。若 QPS 陡然增加可能会将系统一下压垮。为了避免这种情况的发生，我们希望 QPS 缓步增加到设定的阈值。这种应急情况的处理方式称为 **warm up**，即预热，也称为冷启动。在达到 QPS 高水位后，再超出阈值设定的 QPS 值时，将对请求执行“快速失败”，即进行降级处理。

Sentinel 的 Warm Up 流控算法是在 Guava 的 SmoothWarmingUP 算法基础上改进的 (SmoothWarmingUP 继承自 SmoothRateLimiter 算法)。但与 Sentinel 的 Warm UP 算法实现思路是不同的。

- SmoothWarmingUP 算法是通过不断缩短请求间的间隔来达到逐步提高访问量的目的的。
- Sentinel 的 Warm UP 算法是通过逐步提升 QPS 来到达提高访问量的目的的。

## (3) 排队等待



排队等待，也称为匀速排队。该方式会严格控制请求通过的间隔时间，让请求以均匀的速度通过。它是漏斗算法的改进。不同的是，当流量超过设定阈值时，漏斗算法会直接将再来的请求丢弃，而排队等待算法则是将请求缓存起来，后面慢慢处理。不过，该算法目前暂不支持 QPS 超过 1000 的场景。其适合处理 **间隔性** 突发流量场景(削峰填谷)。

## 6.7 来源流控

### 6.7.1 概述

#### (1) 简介

来源流控是针对请求发出者的来源名称所进行的流控。在流控规则中可以直接指定该规则仅用于限制指定来源的请求，一条规则仅可以指定一个限流的来源。

## (2) 原理

来源名称可以在请求参数、请求头或 Cookie 中通过 key-value 形式指定，而 sentinel 提供了一个 RequestOriginParser 的请求解析器，sentinel 可以从该解析器中获取到请求中携带的来源，然后将流控规则应用于获取到的请求来源之上。

### 6.7.2 动态流控中指定来源

#### (1) 定义工程

复制 06-consumer-degrade-sentinel-method-8080 工程，重命名为 06-consumer-reqsource-8080。

#### (2) 定义请求解析器

在工程的任意包下定义如下解析器类。本例定义在 parser 包下，且请求来源的指定方式是，在请求中通过名称为 source 的参数指定来源名称。若没有指定则默认来源名称为 serviceA。

```
no usages
@Component
public class DepartRequestOriginParser implements RequestOriginParser {

    // 方法返回值将作为Sentinel获取的请求来源
    no usages
    @Override
    public String parseOrigin(HttpServletRequest request) {
        String source = request.getParameter("source");
        if (!StringUtils.hasText(source)) {
            source = "serviceA";
        }
        return source;
    }
}
```

#### (3) 修改处理器

指定 DepartController 处理器中 getHandle()方法为 Sentinel 资源。

```
no usages
@GetMapping(value = "/get/{id}")
@SentinelResource(value = "getHandle", fallback = "getHandleFallback")
public Depart getHandle(@PathVariable("id") int id) {
    String url = SERVICE_PROVIDER + "/provider/depart/get/" + id;
    return restTemplate.getForObject(url, Depart.class);
}
```

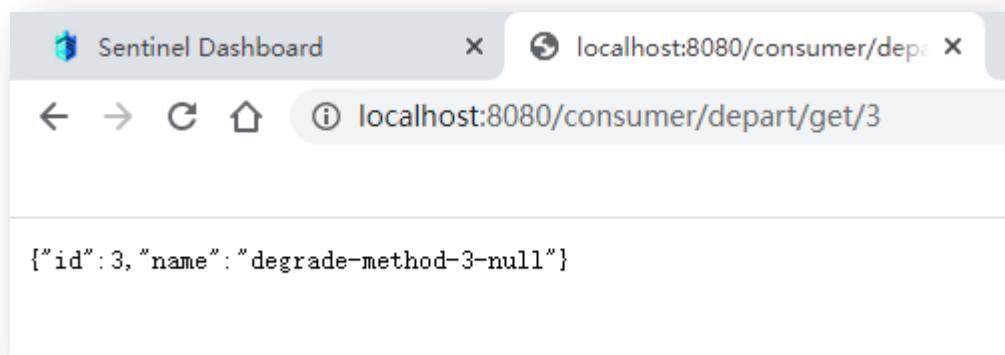
#### (4) dashboard 指定来源

在流控规则中新增一个流控规则，指定该规则仅针对 serviceA 的请求来源起作用。

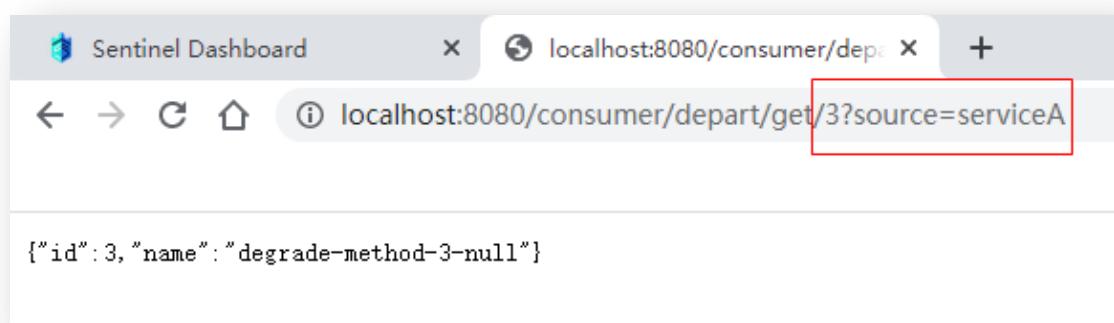


#### (5) 访问 1

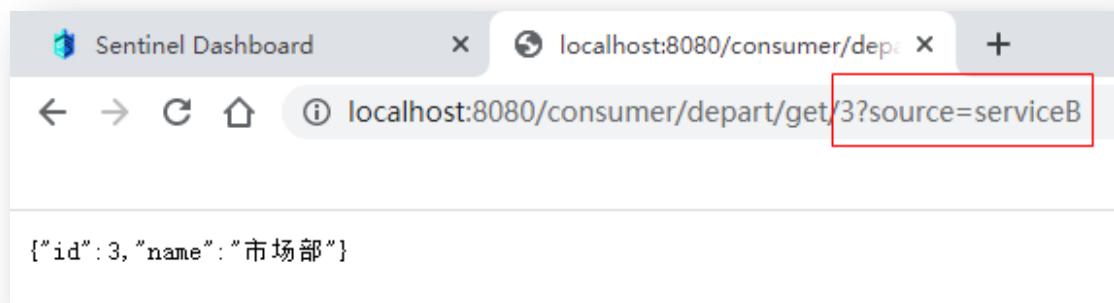
以下两种访问方式都会发生降级。



这种方式是没有携带来源名称请求参数，则使用默认的 serviceA 名称。



而下面的访问方式不会降级，因为该请求来源为 serviceB，流控规则对其不起作用。



## (6) 流控多个来源

若想对同一个资源的多个来源进行流控如何做到呢？同一个资源名称可以配置多条规则。在 dashboard 中再新增两条同名规则。

### consumer-depart

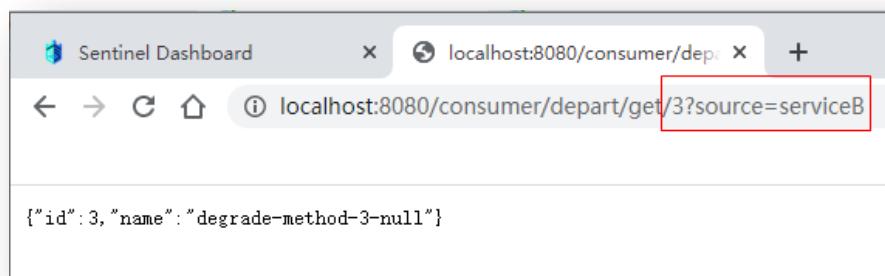
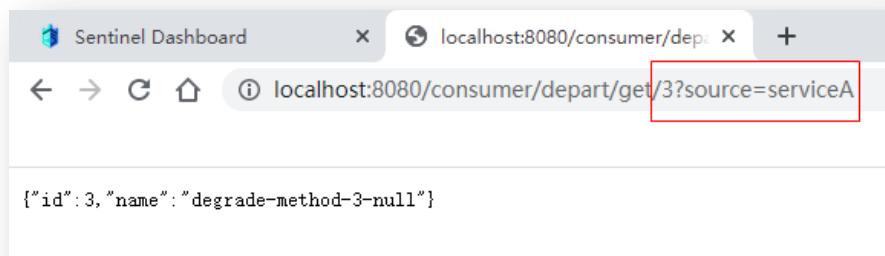
+ 新增流控规则

流控规则		IP	关键字	刷新			
资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
getHandle	serviceB	直接	QPS	4	单机	快速失败	<button>编辑</button> <button>删除</button>
getHandle	other	直接	QPS	5	单机	快速失败	<button>编辑</button> <button>删除</button>
getHandle	serviceA	直接	QPS	3	单机	快速失败	<button>编辑</button> <button>删除</button>

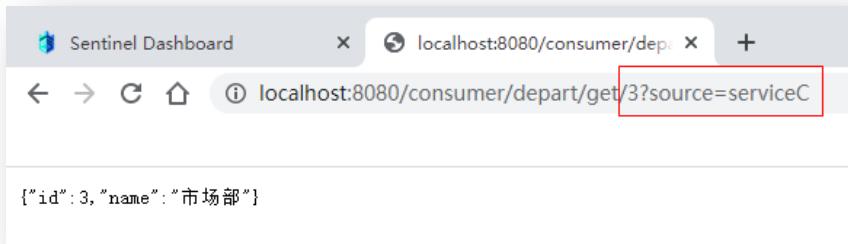
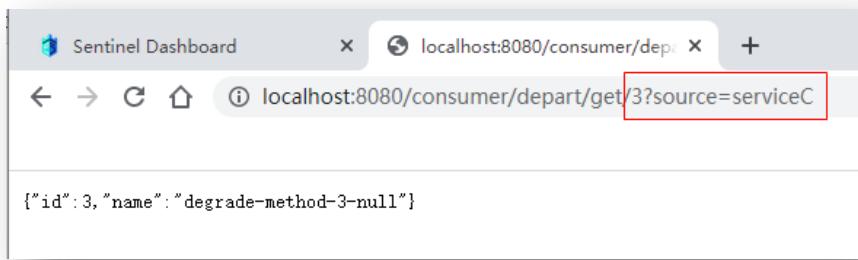
共 3 条记录, 每页 10 条记录

## (7) 访问 2

在手动刷新时 serviceA 与 serviceB 来源的请求都有可能会降级的。



对于其它来源的请求一般是不会降级的，但偶尔刷新的手速快时也可以看到降级结果的。



### 6.7.3 API 流控指定来源

直接在 06-consumer-flowcontrol-8080 工程的启动类中修改。启动工程后，在控制台中就可看到两条流控规则，分别对两个来源进行流控。

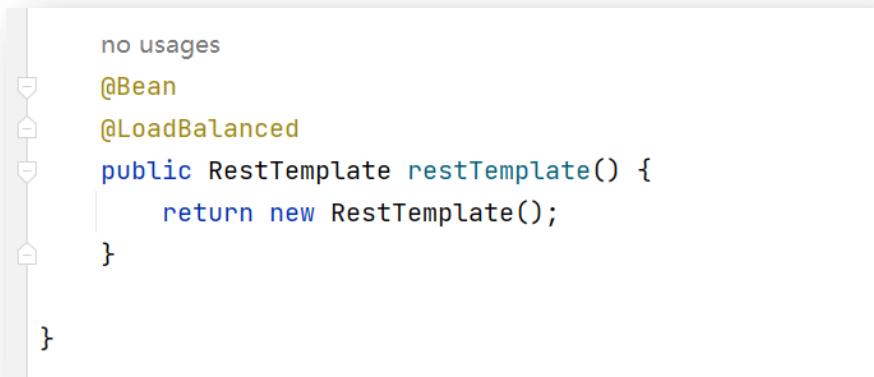
```
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
        initRule();
    }

    1 usage
    private static void initRule() {
        List<FlowRule> flowRules = new ArrayList<>();
        FlowRule flowRule = Consumer8080.configQpsFlowRule();
        FlowRule flowRule2 = Consumer8080.configQpsFlowRule2();
        flowRules.add(flowRule);
        flowRules.add(flowRule2);
        FlowRuleManager.loadRules(flowRules);
    }
}
```

```
1 usage
private static FlowRule configQpsFlowRule() {
    FlowRule flowRule = new FlowRule();
    flowRule.setResource("getHandle");
    flowRule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    flowRule.setCount(3);
    // 指定要限制的请求来源为serviceA
    flowRule.setLimitApp("serviceA");
    return flowRule;
}

1 usage
private static FlowRule configQpsFlowRule2() {
    FlowRule flowRule = new FlowRule();
    flowRule.setResource("getHandle");
    flowRule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    flowRule.setCount(3);
    // 指定要限制的请求来源为serviceB
    flowRule.setLimitApp("serviceB");
    return flowRule;
}
```



## 6.8 授权规则

### 6.8.1 简介

授权规则是一种通过对请求来源进行甄别的鉴权规则。规则规定了哪些请求可以通过访问，而哪些请求则是被拒绝访问的。而这些请求的设置是通过**黑白名单**来完成的。

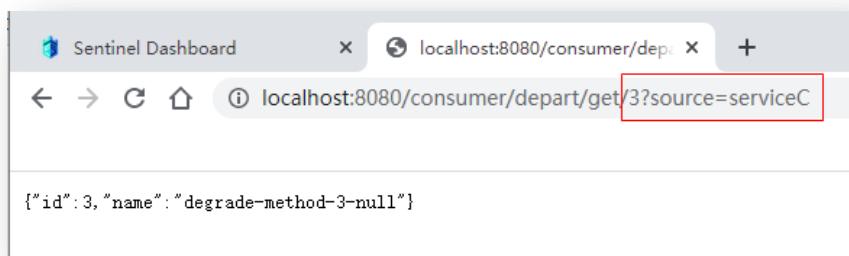
无论是黑名单还是白名单，其实就是一个**请求来源名称列表**。出现在来源黑名单中的请求将被拒绝访问，而其它来源的请求则可以正常访问；出现在来源白名单中的请求是可以正常访问的，而其它来源的请求则将被拒绝。

### 6.8.2 动态设置

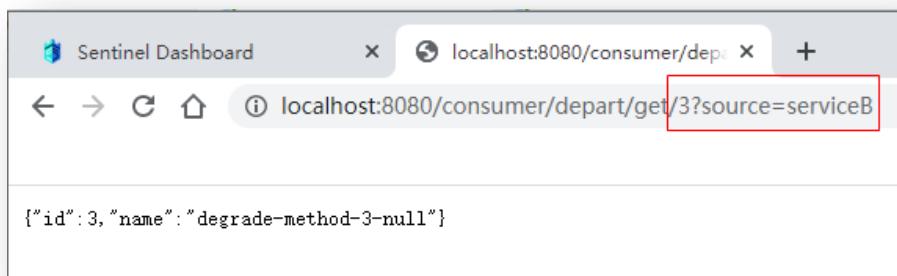
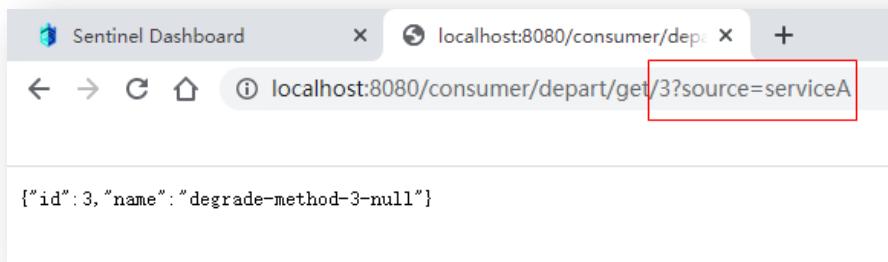
添加一条“授权规则”。



serviceA 与 serviceB 来源的请求均可实现正常访问，但其它来源的请求均被拒绝。说明白名单起了作用。



但快速刷新 serviceA 与 serviceB 来源的请求页面，均会出现降级，说明 QPS 流控规则起了作用。



### 6.8.3 API 设置

直接在 06-consumer-reqsource-8080 工程的启动类中修改。

```
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
        initRule();
    }

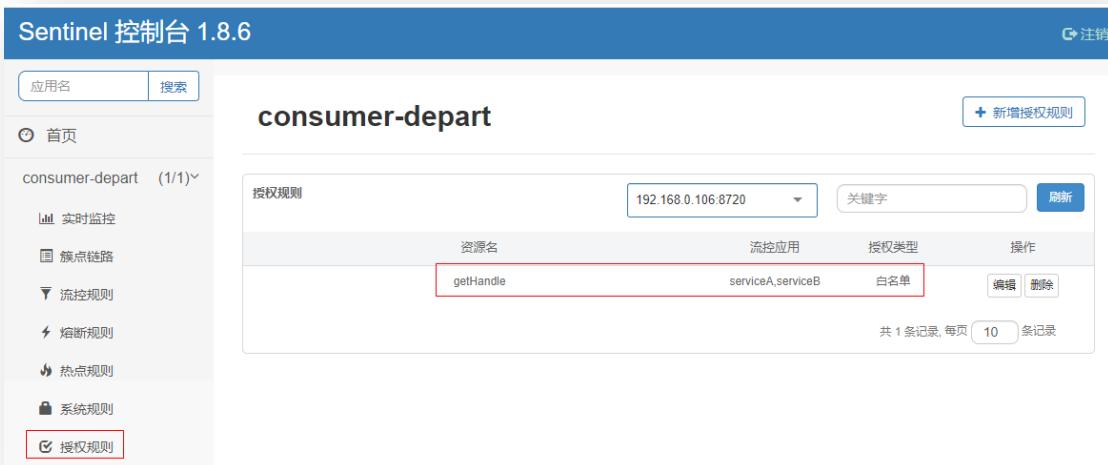
    1 usage
    private static void initRule() {
        List<AuthorityRule> degradeRules = new ArrayList<>();
        AuthorityRule authorityRule = Consumer8080.configAuthorityRule();
        degradeRules.add(authorityRule);
        AuthorityRuleManager.loadRules(degradeRules);
    }
}
```

```
1 usage
private static AuthorityRule configAuthorityRule() {
    AuthorityRule rule = new AuthorityRule();
    rule.setResource("getHandle");
    rule.setStrategy(RuleConstant.AUTHORITY_WHITE);
    rule.setLimitApp("serviceA,serviceB");
    return rule;
}
```

```
no usages
@LoadBalanced
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}

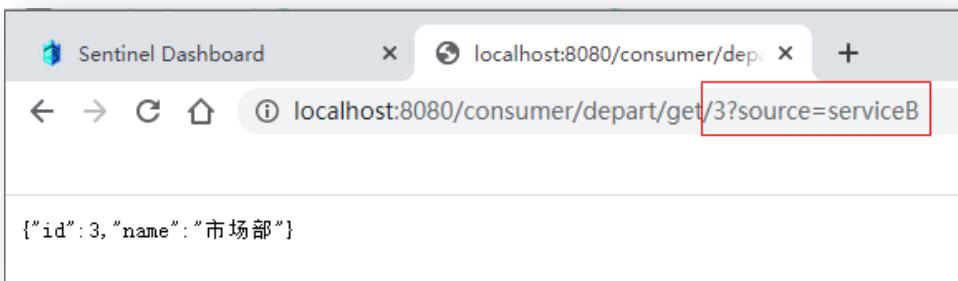
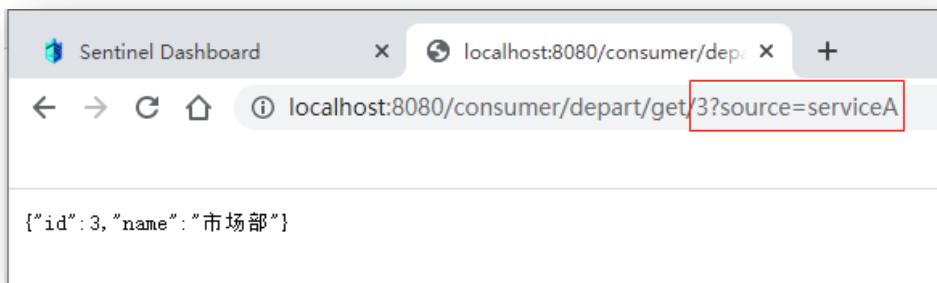
}
```

重启 06-consumer-resource-8080 工程后在控制台可以看到有授权规则。

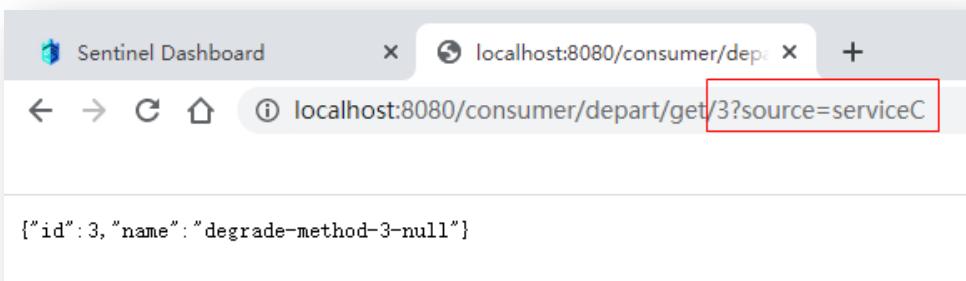


The screenshot shows the Sentinel Control Console interface. On the left, there's a sidebar with navigation links: 首页, 实时监控, 简点链路, 流控规则, 限流规则, 热点规则, 系统规则, and 授权规则. The '授权规则' link is highlighted with a red border. The main content area has a title 'consumer-depart'. Below it, there's a search bar with '授权规则' and dropdowns for 'IP' (192.168.0.106:8720) and '关键字'. A button '+ 新增授权规则' is also present. A table lists the current rule: 资源名 'getHandle', 流控应用 'serviceA,serviceB', 授权类型 '白名单', and an '操作' column with '编辑' and '删除' buttons. At the bottom right of the table, it says '共 1 条记录, 每页 10 条记录'.

在访问时，serviceA 与 serviceB 来源的请求均可实现正常访问。



但 serviceC 来源的请求则直接降级，说明白名单起了作用。



## 6.9 热点规则

### 6.9.1 简介

热点规则是用于实现热点参数限流的规则。热点参数限流指的是，在流控规则中指定对某方法参数的 QPS 限流后，当所有对该资源的请求 URL 中携带有该指定参数的请求 QPS 达到了阈值，则发生限流。

## 6.9.2 动态设置

### (1) 定义工程

复制 06-consumer-degrade-sentinel-method-8080 工程，重命名为 06-consumer-paramflow-8080。

### (2) 修改控制器类

由于这里的测试需要至少两个参数，而 DepartController 的处理器方法不满足条件，所以就在 DepartController 处理器中添加如下两个方法。

```
no usages
@SentinelResource(value = "compluxHandle",
                   fallback = "compluxHandleFallback")
@GetMapping("/complux")
public String compluxHandle(Integer id, String name) {
    return "complux: " + id + ", " + name;
}

no usages
public String compluxHandleFallback(Integer id, String name) {
    return "complux fallback: " + id + ", " + name;
}
```

### (3) dashboard 设置规则

在启动了消费者工程后，再设置 Sentinel 控制台中的“热点规则”。

Sentinel 控制台 1.8.6

注销

用户名 搜索

首页 consumer-depart (1/1)

实时监控 热点链路 流控规则 延时规则 热点规则 系统规则 授权规则 集群流控

新增热点限流规则

热点参数限流规则 192.168.0.106:8720 关键字 刷新

资源名	参数索引	流控模式	阈值	是否集群	例外项数目	操作

共 0 条记录, 每页 10 条记录

新增热点规则

资源名: compluxHandle

限流模式: QPS 模式

参数索引: 0

单机阈值: 3      统计窗口时长: 10 秒

是否集群:

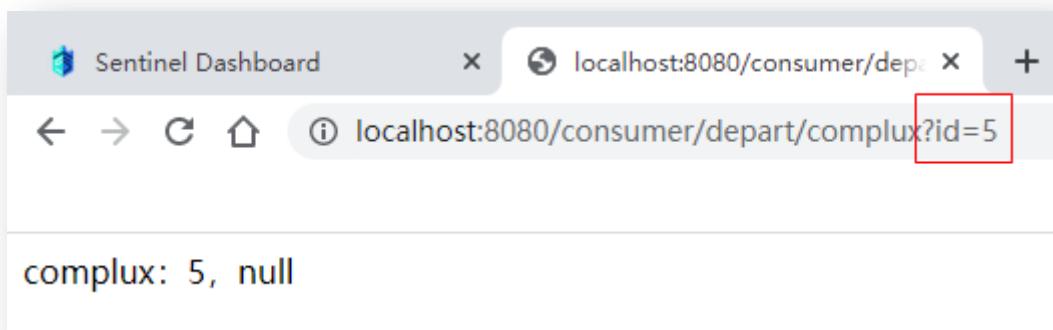
高级选项

新增 取消

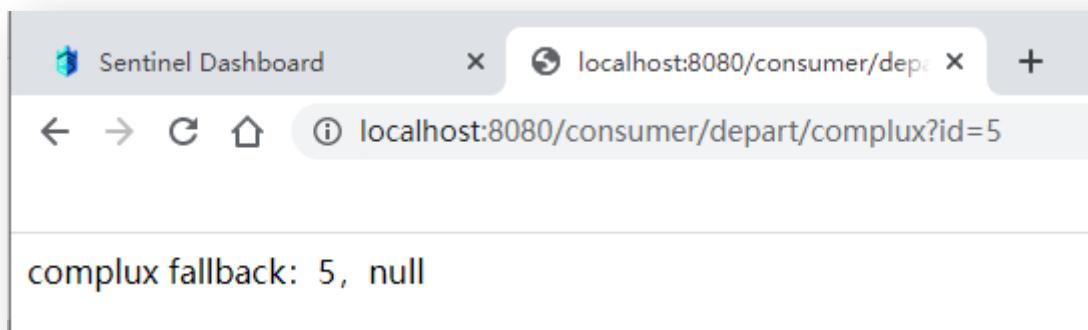
其默认采用的就是 QPS 流控。这里指定热点参数为第 0 个参数，本例就是 Integer 类型的 id，当对该参数的访问 QPS 超过 3 次时会触发熔断。

#### (4) 访问 1

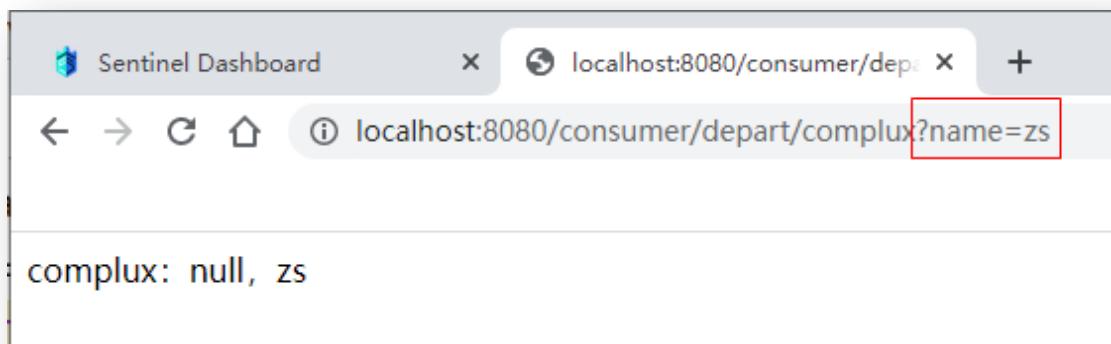
仅提交一次请求，可以获取到正常结果。



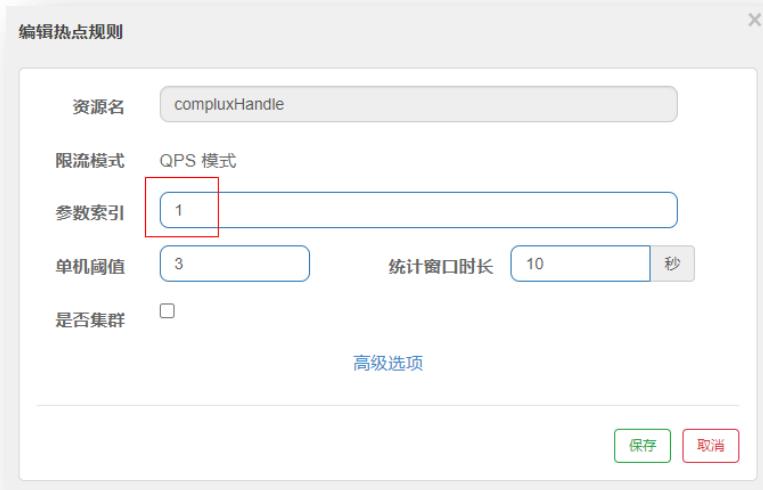
快速按下 F5 刷新页面，看到的则是降级结果。



但以下请求页面中，快速按下 F5 也不会是降级页面，因为该请求没有对热点参数 id 属性进行访问。



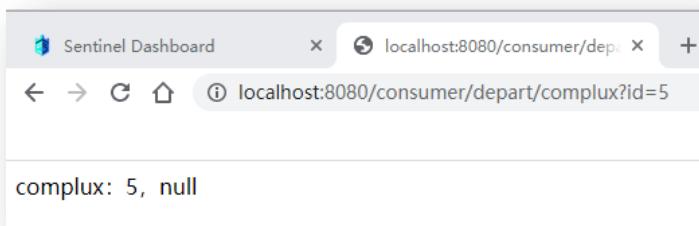
## (5) 修改 dashboard 中规则



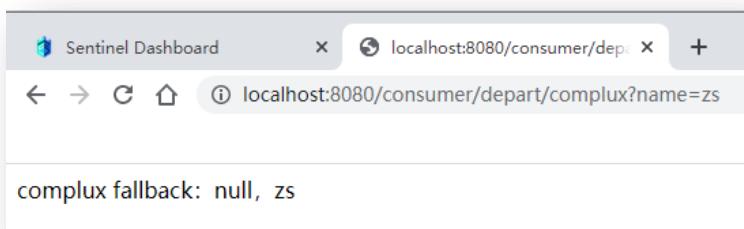
仅修改参数索引，即修改热点参数。本例即指定 name 属性为热点参数。

## (6) 访问 2

此时快速刷新如下请求页面也不会出现降级结果。



但快速刷新如下页面，则会显示降级结果。



### 6.9.3 参数例外项

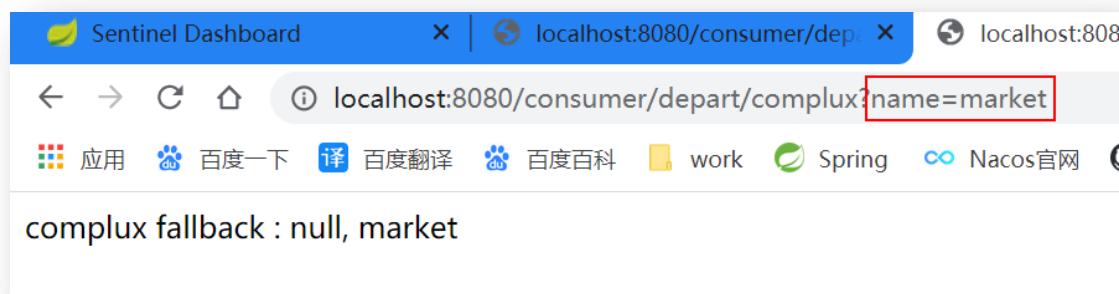
参数例外项是指，对于热点参数中某个或某些特殊值单独设置 QPS 流控阈值。参数类型仅支持基本数据类型或其对应的包装类型，及 String 类型。

#### (1) 修改 dashboard 中规则

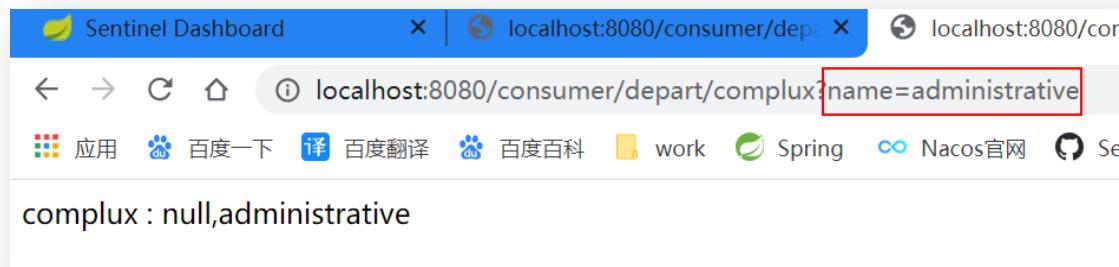


#### (2) 访问

但快速刷新如下页面，则会显示降级结果。注意提交的 name 属性值为 market。



若将 name 属性值修改为 administrative 或 human 后，再快速刷新也不会降级，因为刷新速度达不到每秒 100 次。



#### 6.9.4 API 设置

直接在工程 06-consumer-paramflow-8080 上进行修改。

##### (1) 修改启动类

在启动类中添加如下代码。

```
@SpringBootApplication
public class Consumer8080 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Consumer8080.class, args);
        initRule();
    }

    1 usage
    private static void initRule() {
        List<ParamFlowRule> rules = new ArrayList<>();
        ParamFlowRule rule = Consumer8080.configParamFlowRule();
        rules.add(rule);
        ParamFlowRuleManager.loadRules(rules);
    }
}
```

```
1 usage
private static ParamFlowRule configParamFlowRule() {
    ParamFlowRule rule = new ParamFlowRule();
    rule.setResource("compluxHandle");
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setCount(3);
    rule.setParamIdx(1);
    rule.setDurationInSec(10);

    List<ParamFlowItem> items = new ArrayList<>();
    items.add(nameParamItem("admin", 100));
    items.add(nameParamItem("administrator", 100));
    rule.setParamFlowItemList(items);
    return rule;
}
```

```
2 usages
private static ParamFlowItem nameParamItem(String value, int count) {
    ParamFlowItem item = new ParamFlowItem();
    item.setClassType(String.class.getName());
    item.setObject(String.valueOf(value));
    item.setCount(count);
    return item;
}

no usages
@LoadBalanced
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

## (2) dashboard 查看

06-consumer-paramflow-8080 工程重启后，在 dashboard 上可以直接看到设置的该规则。



The screenshot shows the Sentinel Control Console interface. The main title is "Sentinel 控制台 1.8.6". On the left, there's a sidebar with navigation links: 首页, consumer-depart (1/1), 实时监控, 热点链路, 流控规则, 热断规则, and 热点规则. The main content area is titled "consumer-depart". It displays a table for "热点参数限流规则" (Hotspot Parameter Flow Control Rules). The table has columns: 资源名 (Resource Name), 参数索引 (Parameter Index), 流控模式 (Flow Control Mode), 阈值 (Threshold), 是否集群 (Is Clustered), 例外项数目 (Exception Item Number), and 操作 (Operations). One row is shown: "compluxHandle" with index 1, QPS mode, threshold 3, not clustered, 2 exception items, and buttons for 编辑 (Edit) and 删除 (Delete). There are also buttons for 新增热点限流规则 (Add New Hotspot Flow Control Rule) and 刷新 (Refresh).

## (3) 访问

与前面的访问测试方式相同。

## 6.10 系统规则

### 6.10.1 简介

系统规则是用于实现系统自适应限流的。系统自适应限流，即对应用级别入口流量进行整体控制，结合应用的 Load、CPU 使用率、平均 RT、入口 QPS 和入口并发线程数等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

### 6.10.2 规则模式

系统规则目前支持五种模式：



#### (1) 系统负载 Load

该模式仅对 Linux/Unix-like 系统生效。系统的 `load1`(就是系统的负载) 作为启发指标，进行自适应系统保护。当系统 `load1` 超过设定的启发值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护（BBR 阶段）。系统容量可以由  $\text{maxQps} * \text{minRt}$  估算得出。不过，也可以通过  $\text{CPU cores} * 2.5$  计算出其参考数值。

## (2) CPU 使用率

当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0），比较灵敏。

## (3) 平均响应时间 RT

当对当前应用上所有入口流量的平均 RT 达到阈值时触发系统保护，单位是毫秒。

## (4) 并发线程数

当对当前应用的所有入口流量进行处理的所有线程数量达到阈值时触发系统保护。

## (5) 入口 QPS

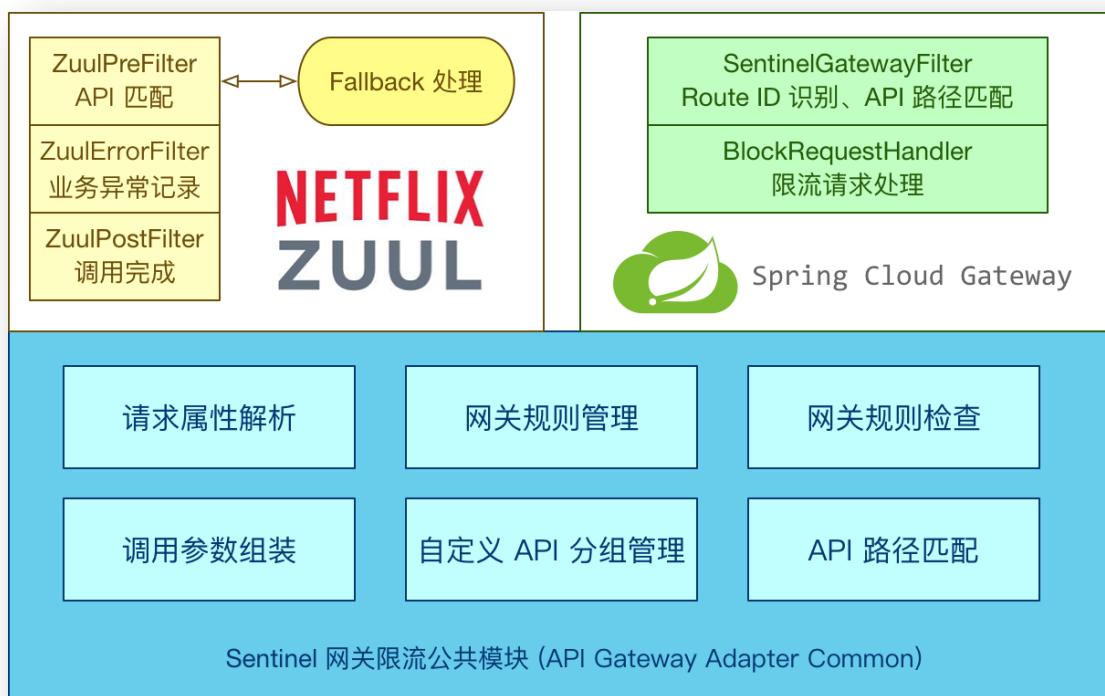
当对当前应用的所有入口流量的总 QPS 达到阈值时触发系统保护。

## 6.11 网关流控

### 6.11.1 概述

#### (1) 公共适配器

从 Sentinel1.6.0 开始，Sentinel 对于主流网关限流的实现，是通过 Sentinel API Gateway Adapter Common 这个公共适配器模块实现的。



## (2) 限流维度

这个公共适配器模块提供了两种维度的限流：

- **Route 维度：**根据网关路由中指定的路由 id 进行路由
- **API 维度：**使用 Sentinel 提供的 API 自定义分组进行限流

### 6.11.2 动态设置--Route 维度

#### (1) 定义工程

复制 05-gateway-lb-config-9000 工程，重命名为 06-gateway-sentinel-route-9000。

#### (2) 修改 pom

需要导入 sentinel 依赖，及 sentinel 与 spring cloud gateway 整合依赖。

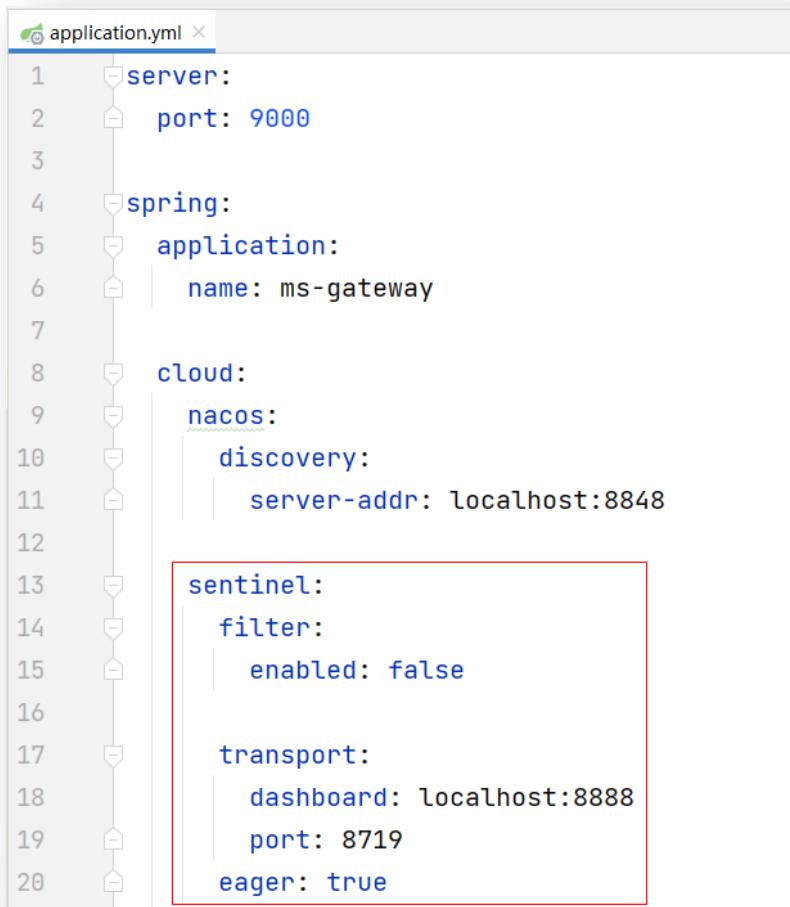
```
<!--sentinel 与 spring cloud gateway 整合依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
```

```
</dependency>

<!--sentinel 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

### (3) 修改配置文件

在配置文件中需要添加上当前网关的微服务名称、nacos server 的地址、开启 gateway 从注册中心进行服务发现的功能、关闭 sentinel 中的 filter 功能，及 Sentinel 控制台配置。



```
application.yml
1  server:
2      port: 9000
3
4  spring:
5      application:
6          name: ms-gateway
7
8  cloud:
9      nacos:
10         discovery:
11             server-addr: localhost:8848
12
13     sentinel:
14         filter:
15             enabled: false
16
17         transport:
18             dashboard: localhost:8888
19             port: 8719
20             eager: true
```

```
21
22     gateway:
23         discovery:
24             locator:
25                 enabled: true
26
27         routes:
28             - id: get_route
29                 uri: lb://consumer-depart
30                 predicates:
31                     - Path=/consumer/depart/get/**

32
33             - id: list_route
34                 uri: lb://consumer-depart
35                 predicates:
36                     - Path=/consumer/depart/list
37
```

#### (4) dashboard 设置规则

由于网关流控具有其特殊性，所以从 Sentinel 1.6.3 开始，为网关流控专门设计了控制台，用户可以直接在 Sentinel 控制台上查看 API Gateway 实时的 route 和自定义 API 分组监控，管理网关规则和 API 分组配置。

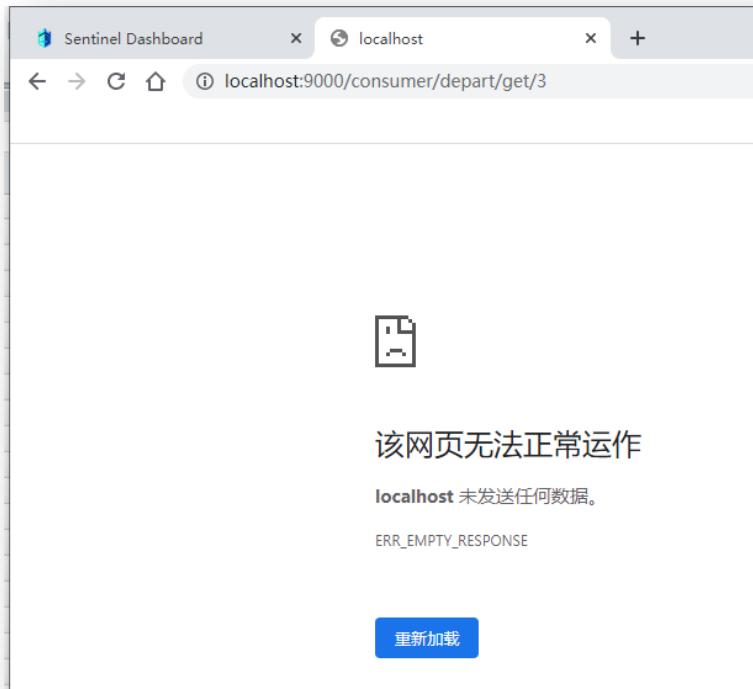


The screenshot shows the Sentinel Control Console interface. On the left sidebar, under the 'ms-gateway' section, the '流控规则' (Flow Control Rules) option is selected. A modal window titled '新增网关流控规则' (Add New Gateway Flow Control Rule) is open. Inside the modal, the 'API 名称' (API Name) field contains 'get\_route'. Other configuration fields include 'API 类型' (Route ID), 'QPS 阀值' (3), '间隔' (1 秒), '流控方式' (Quick Failure), and 'Burst size' (0). At the bottom right of the modal are two buttons: a green '新增' (Add) button and a red '取消' (Cancel) button.

## (5) 访问

对于 <http://localhost:9000/consumer/depart/list> 的快速访问也不会出现问题。

对于 <http://localhost:9000/consumer/depart/get/3> 的正常访问是没有问题的，但快速访问会出现如下一闪而过的错误页面。



### 6.11.3 修改阻断异常结果

前面的例子中被限流后的结果非常不友好。可以修改发生流控阻断异常的结果。

#### (1) API 式阻断异常结果

##### A、重定向异常结果

直接在 06-gateway-sentinel-route-9000 工程启动类中添加如下内容。

```
@SpringBootApplication
public class Gateway9000 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Gateway9000.class, args);
        initBlockHandlers();
    }

    1 usage
    private static void initBlockHandlers() {
        GatewayCallbackManager.setBlockHandler(new RedirectBlockRequestHandler("https://baidu.com"));
    }
}
```

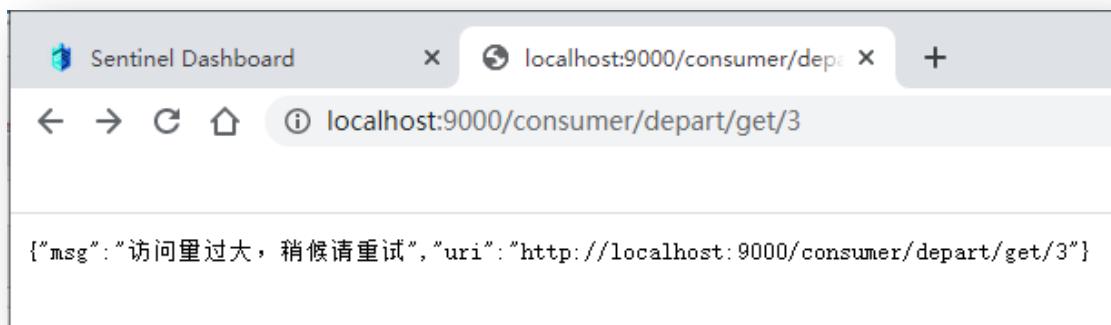
当发生限流阻断时，重定向到百度。

## B、自定义异常结果

直接修改 06-gateway-sentinel-route-9000 工程启动类中的 initBlockHandlers()方法，将原来该方法中的内容删除，写入新的内容。

```
1 usage
private static void initBlockHandlers() {
    GatewayCallbackManager.setBlockHandler((exchange, th) -> {
        URI uri = exchange.getRequest().getURI();
        Map<String, Object> map = new HashMap<>();
        map.put("uri", uri);
        map.put("msg", "访问量过大，稍候请重试");
        return ServerResponse.status(HttpStatus.TOO_MANY_REQUESTS)
            .contentType(MediaType.APPLICATION_JSON)
            .body(BodyInserters.fromValue(map));
    });
    // GatewayCallbackManager.setBlockHandler(new RedirectBlockRequestHandler("https://baidu.com"));
}
```

重启 06-gateway-sentinel-9000 工程后，快速刷新/get/\*\*访问页面，可以看到限流阻断结果。



## (2) 配置式阻断异常结果

前面都是通过代码方式来修改限流阻断异常返回结果的。也可通过在配置文件中添加相应的配置属性来修改限流阻断异常返回结果。

### A、重定向到百度



```
spring:
  application:
    name: ms-gateway

  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848

  sentinel:
    scg:
      fallback:
        mode: redirect
        redirect: https://baidu.com
```

## B、自定义异常结果

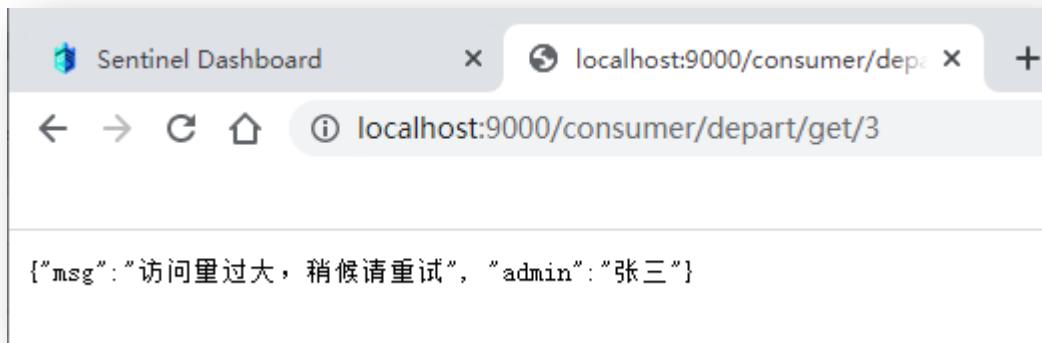


```
spring:
  application:
    name: ms-gateway

  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848

  sentinel:
    scg:
      fallback:
        mode: response
        response-status: 429
        content-type: 'application/json'
        response-body: '{"msg": "访问量过大，稍候请重试", "admin": "张三"}'
```

其限流后的结果如下：



### (3) 优先级

当代码中与配置文件中都设置了限流阻断异常结果时，代码中的优先级更高。

## 6.11.4 针对路由断言的流控

### (1) 三种匹配模式

- 精确：请求中的某个属性值与设置的“匹配串”值完全相同时才会应用当前的流控规则
- 子串：请求中的某个属性值包含设置的“匹配串”时才会应用当前的流控规则
- 正则：请求中的某个属性值符合设置的“匹配器”中的正则表达式时才会应用当前的流控规则

### (2) Client IP

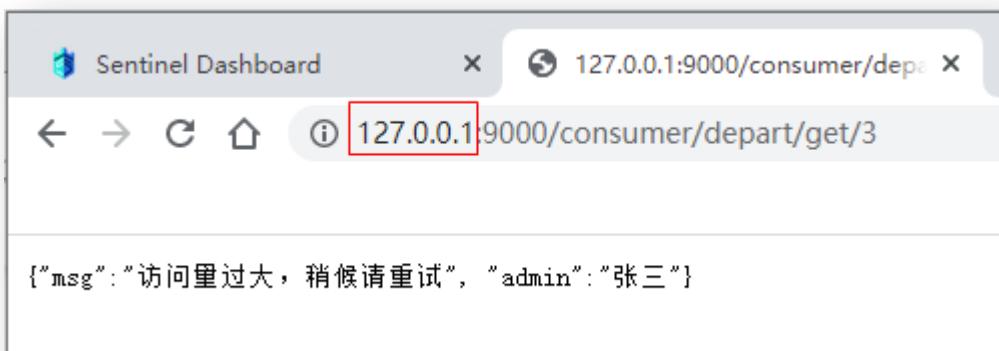
这里并不是针对提交请求的客户端的 IP 进行流控，而是针对网关的 IP 进行流控。因为对于微服务来说，网关就相当于它们的 Client。那么，该设置有什么用呢？生产中的 Spring Cloud Gateway 网关一定是集群，集群的每个节点服务器都会具有不同的 IP。每个用户请求都会通过 Nginx 被路由到不同的网关服务器。而该配置就可以针对用户提交请求所经过的不同的网关集群节点服务器进行流控。

## A、精确

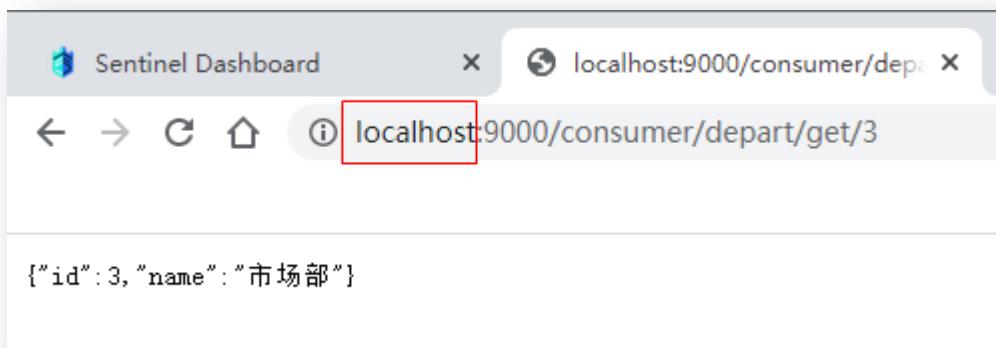
编辑网关流控规则

API 类型	<input checked="" type="radio"/> Route ID <input type="radio"/> API 分组		
API 名称	get_route		
针对请求属性	<input checked="" type="checkbox"/>		
参数属性	<input checked="" type="radio"/> Client IP <input type="radio"/> Remote Host <input type="radio"/> Header <input type="radio"/> URL 参数 <input type="radio"/> Cookie		
属性值匹配	<input checked="" type="checkbox"/>		
匹配模式	<input checked="" type="radio"/> 精确 <input type="radio"/> 子串 <input type="radio"/> 正则	匹配串	127.0.0.1
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数		
QPS 阈值	3		
间隔	1	秒	<input type="button" value="▼"/>
流控方式	<input checked="" type="radio"/> 快速失败 <input type="radio"/> 匀速排队		
Burst size	0		
<input type="button" value="保存"/> <input type="button" value="取消"/>			

以下请求会发生限流。



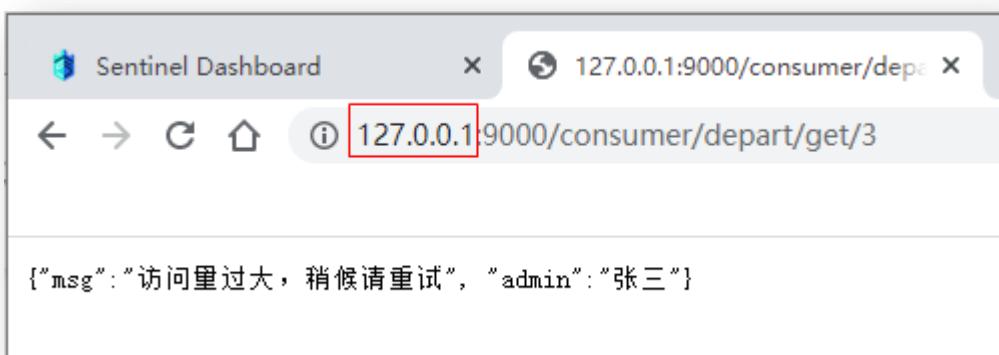
以下请求不会被限流。



## B、子串

如果设置为子串，则只要是包含指定的子串“0.0”的访问 IP，都将被限流。





## C、正则



\d\* 表示 0-n 个数字。不过，这里不识别+号。



### (3) Remote Host



这里指定的 Remote Host 与路由断言工厂中的 RemoteAddr 是相同的意义，都是指的访问当前网关的客户端的地址。这里的“匹配串”就是客户端 IP 的白名单。

## (4) Header

编辑网关流控规则

API 类型	<input checked="" type="radio"/> Route ID <input type="radio"/> API 分组
API 名称	get_route
针对请求属性	<input checked="" type="checkbox"/>
参数属性	<input type="radio"/> Client IP <input type="radio"/> Remote Host <input checked="" type="radio"/> Header <input type="radio"/> URL 参数 <input type="radio"/> Cookie
Header 名称	x-request-color
属性值匹配	<input checked="" type="checkbox"/>
匹配模式	<input type="radio"/> 精确 <input type="radio"/> 子串 <input checked="" type="radio"/> 正则
	匹配串 gr.*
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
QPS 阈值	3
间隔	1 秒
流控方式	<input checked="" type="radio"/> 快速失败 <input type="radio"/> 匀速排队
Burst size	0
<input type="button" value="保存"/> <input type="button" value="取消"/>	

REQUEST

METHOD: GET SCHEME // HOST [":] [ PATH "?" QUERY ]  
http://localhost:9000/consumer/depart/get/3

HEADERS: x-request-color: green

RESPONSE

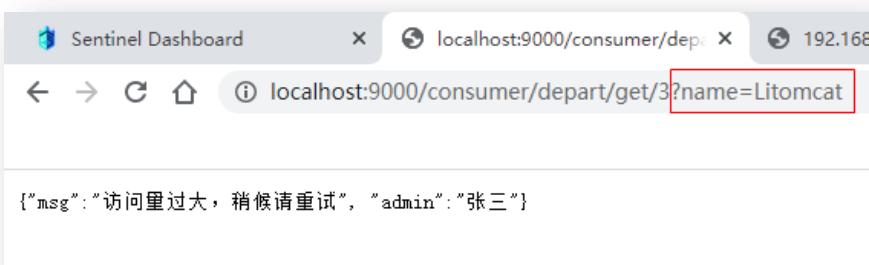
429 Too Many Requests

HEADERS: pretty  
BODY:

```
content-length: 61 bytes
content-type: application/json
{
    "msg": "访问量过大，稍候请重试",
    "admin": "张三"
}
```

## (5) URL 参数

URL 参数，即请求参数。



## (6) Cookie

与前两个类似，都是 key=value 形式的键值对。

## 6.11.5 动态设置--API 维度

### (1) 定义 API 分组



Sentinel 控制台 1.8.6

ms-gateway (1/1)

+ 新增 API 分组

API 名称	匹配模式	匹配串	操作

共 0 条记录, 每页 10 条记录



新增自定义 API

API 名称	depart_group			
匹配模式	<input checked="" type="radio"/> 精确 <input type="radio"/> 前缀 <input type="radio"/> 正则	匹配串	/consumer/depart/save	X
匹配模式	<input checked="" type="radio"/> 精确 <input type="radio"/> 前缀 <input type="radio"/> 正则	匹配串	/consumer/depart/update	X
匹配模式	<input type="radio"/> 精确 <input type="radio"/> 前缀 <input checked="" type="radio"/> 正则	匹配串	/consumer/depart/get/*	X

+ 新增匹配规则

新增 取消

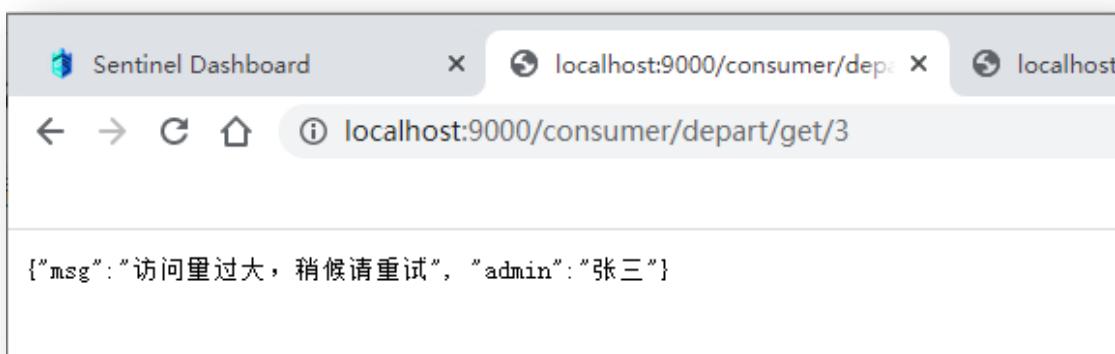
## (2) 定义 API 流控规则

新增网关流控规则

API 类型	<input type="radio"/> Route ID <input checked="" type="radio"/> API 分组
API 名称	depart_group
针对请求属性	<input type="checkbox"/>
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
QPS 阈值	3
间隔	1 秒
流控方式	<input checked="" type="radio"/> 快速失败 <input type="radio"/> 匀速排队
Burst size	0
<span style="float: right;">新增 取消</span>	

## (3) 访问

对 <http://localhost:9000/consumer/depart/get/3> 的请求会发生限流。



但对 <http://localhost:9000/consumer/depart/list> 的访问是不会被限流的。

## 6.11.6 API 设置--Route 维度

就像之前代码实现其它限流时一样，该限流规则同样也需要在启动类中添加。  
这里指定要对 RouteId 为 order\_route 的路由进行限流。

### (1) initRule()



```
-----  
@SpringBootApplication  
public class Gateway9000 {  
  
    no usages  
    public static void main(String[] args) {  
        SpringApplication.run(Gateway9000.class, args);  
        initBlockHandlers();  
        initRule();  
    }  
  
    1 usage  
    private static void initRule() {  
        Set<GatewayFlowRule> rules = new HashSet<>();  
        GatewayFlowRule rule = Gateway9000.configFlowRule();  
        rules.add(rule);  
        GatewayRuleManager.loadRules(rules);  
    }  
}
```

## (2) configFlowRule()

```
1 usage
private static GatewayFlowRule configFlowRule() {
    GatewayFlowRule rule = new GatewayFlowRule();
    rule.setResourceMode(SentinelGatewayConstants.RESOURCE_MODE_ROUTE_ID);
    rule.setResource("get_route");
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setCount(3);
    rule.setIntervalSec(1);
    rule.setBurst(5);
    rule.setControlBehavior(RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER);
    rule.setMaxQueueingTimeOutMs(500);

    GatewayParamFlowItem item = new GatewayParamFlowItem();
    item.setParseStrategy(SentinelGatewayConstants.PARAM_PARSE_STRATEGY_URL_PARAM);
    item.setMatchStrategy(SentinelGatewayConstants.PARAM_MATCH_STRATEGY_CONTAINS);
    item.setFieldName("name");
    item.setPattern("ang");

    rule.setParamItem(item);
    return rule;
}
```

## 6.11.7 API 设置--API 维度

### (1) 定义工程

复制 06-gateway-sentinel-route-9000 工程，重命名为 06-gateway-sentinel-api-9000。

### (2) 修改启动类

```
@SpringBootApplication
public class Gateway9000 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Gateway9000.class, args);
        initCustomizedApis();
        initRule();
    }
}
```

```
1 usage
private static void initCustomizedApis() {
    Set<ApiDefinition> definitions = new HashSet<>();
    ApiDefinition api1 = new ApiDefinition("some_customized_api")
        .setPredicateItems(new HashSet<ApiPredicateItem>() {{
            add(new ApiPathPredicateItem().setPattern("/consumer/depart/save"));
            add(new ApiPathPredicateItem().setPattern("/consumer/depart/get/3"))
                .setMatchStrategy(SentinelGatewayConstants.URL_MATCH_STRATEGY_EXACT);
        }});
    ApiDefinition api2 = new ApiDefinition("another_customized_api")
        .setPredicateItems(new HashSet<ApiPredicateItem>() {{
            add(new ApiPathPredicateItem().setPattern("/consumer/depart/get/**"))
                .setMatchStrategy(SentinelGatewayConstants.URL_MATCH_STRATEGY_PREFIX);
        }});
    definitions.add(api1);
    definitions.add(api2);
    GatewayApiDefinitionManager.loadApiDefinitions(definitions);
}
```

```
1 usage
private static void initRule() {
    Set<GatewayFlowRule> rules = new HashSet<>();
    GatewayFlowRule rule = Gateway9000.configFlowRule();
    rules.add(rule);
    GatewayRuleManager.loadRules(rules);
}

1 usage
private static GatewayFlowRule configFlowRule() {
    GatewayFlowRule rule = new GatewayFlowRule();
    rule.setResourceMode(SentinelGatewayConstants.RESOURCE_MODE_CUSTOM_API_NAME);
    rule.setResource("another_customized_api");
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setCount(3);
    rule.setIntervalSec(1);
    return rule;
}
```

### (3) 访问

由于代码中指定的是对 another\_customized\_api 进行流控，所以，只要是 http://localhost:9000/consumer/depart/get/..... 请求，都有可能会发生限流。



如果代码中指定对 some\_customized\_api 进行流控，则只会对 <http://localhost:9000/consumer/depart/get/3> 请求进行限流。

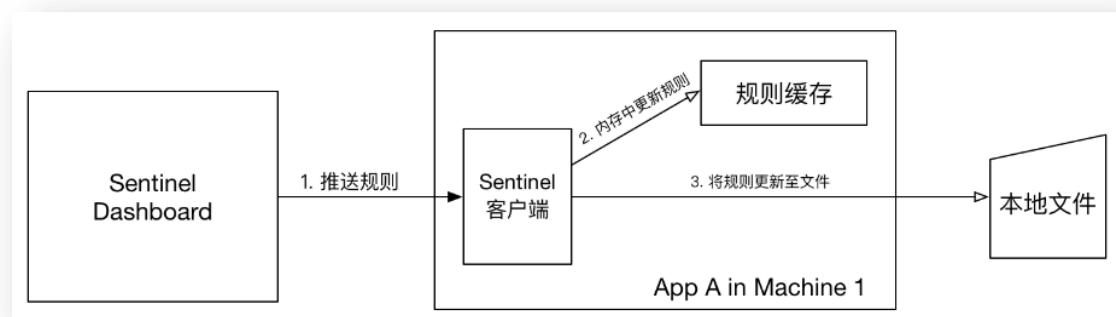


## 6.12 规则持久化

通过 Sentinel Dashboard 动态地设置规则非常的方便，但存在一个问题：一旦应用重启，则规则消失。因为这些规则是存在于 Sentinel Dashboard 主机的内存中的，微服务应用一旦重启，Sentinel Dashboard 就会失去与应用的通信，该内存中的数据就会消失。此时就需要将这些动态规则持久化，Sentinel 可以通过 DataSource 扩展实现持久化。

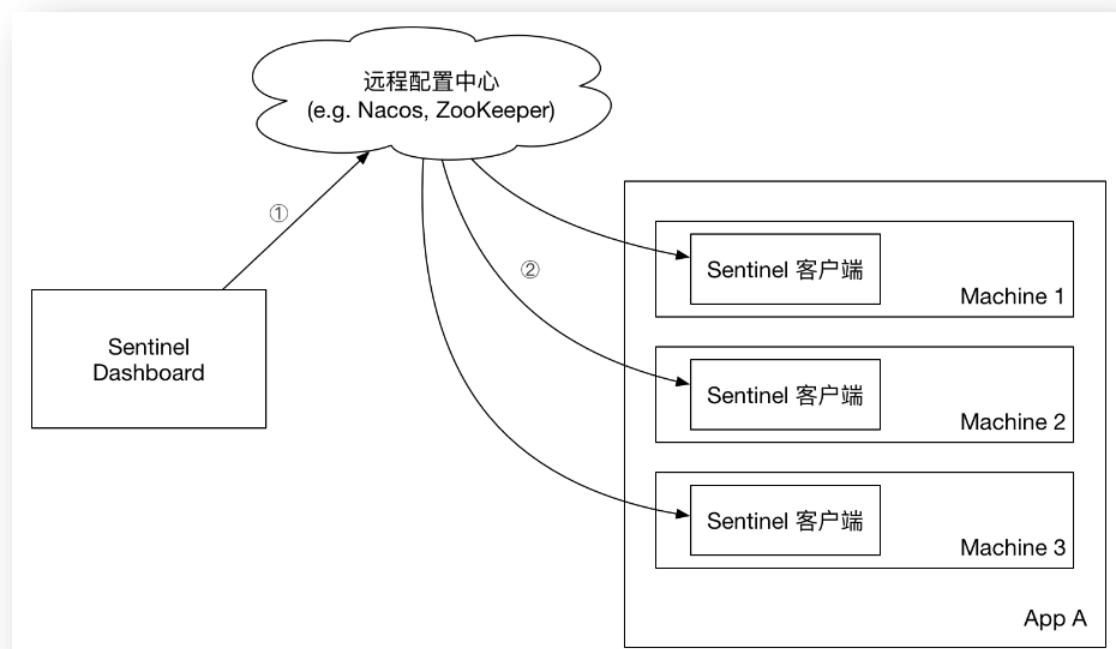
### 6.12.1 DataSource 扩展

#### (1) 拉模式



客户端主动向某个规则管理中心定期轮询拉取规则，这个规则中心可以是 RDBMS、文件，甚至是 VCS 等。这样做的方式是简单，缺点是无法及时获取变更。

#### (2) 推模式



规则中心统一推送，客户端通过注册监听器的方式时刻监听变化。Sentinel 目前支持的基于推模式的数据源有：Nacos、Zookeeper、Redis、Apollo、etcd。这种方式有更好的实时性。

性和一致性保证。

### 6.12.2 拉模式持久化

#### (1) 准备工程

直接在 06-consumer-flowcontrol-8080 工程中进行修改即可。无需导入任何依赖。

#### (2) 定义 FileDataSourceInit 类

该类完成了对 FlowRule、DegradeRule、AuthorityRule、SystemRule 及 ParamFlowRule 五种规则的持久化。

```
1 usage
public class FileDataSourceInit implements InitFunc {

    @Override
    public void init() throws Exception {
        File ruleFileDir = new File(System.getProperty("user.dir") + "/rules");
        if (!ruleFileDir.exists()) {
            ruleFileDir.mkdirs();
        }

        readWriteRuleFileFlow(ruleFileDir.getPath());
        readWriteRuleFileDegrade(ruleFileDir.getPath());
        readWriteRuleFileAuthority(ruleFileDir.getPath());
        readWriteRuleFileSystem(ruleFileDir.getPath());
        readWriteRuleFileParam(ruleFileDir.getPath());
    }
}
```

```
// FlowRule
1 usage
private void readWriteRuleFileFlow(String ruleFilePath) throws IOException {
    String ruleFile = ruleFilePath + "/flow-rule.json";
    createRuleFile(ruleFile);
    ReadableDataSource<String, List<FlowRule>> ds = new FileRefreshableDataSource<>(
        ruleFile, source -> JSON.parseObject(source, new TypeReference<List<FlowRule>>() {})
    );
    // 将可读数据源注册至 FlowRuleManager.
    FlowRuleManager.register2Property(ds.getProperty());

    WritableDataSource<List<FlowRule>> wds = new FileWritableDataSource<>(ruleFile, this::encodeJson);
    // 将可写数据源注册至 transport 模块的 WritableDataSourceRegistry 中.
    // 这样收到控制台推送的规则时, Sentinel会先更新到内存, 然后将规则写入到文件中.
    WritableDataSourceRegistry.registerFlowDataSource(wds);
}
```

```
// DegradeRule
1 usage
private void readWriteRuleFileDegrade(String ruleFilePath) throws IOException {
    String ruleFile = ruleFilePath + "/degrade-rule.json";
    createRuleFile(ruleFile);
    ReadableDataSource<String, List<DegradeRule>> ds = new FileRefreshableDataSource<>(
        ruleFile, source -> JSON.parseObject(source, new TypeReference<List<DegradeRule>>() {})
    );
    DegradeRuleManager.register2Property(ds.getProperty());

    WritableDataSource<List<DegradeRule>> wds = new FileWritableDataSource<>(ruleFile, this::encodeJson);
    WritableDataSourceRegistry.registerDegradeDataSource(wds);
}
```

```
// AuthorityRule
1 usage
private void readWriteRuleFileAuthority(String ruleFilePath) throws IOException {
    String ruleFile = ruleFilePath + "/authority-rule.json";
    createRuleFile(ruleFile);
    ReadableDataSource<String, List<AuthorityRule>> ds = new FileRefreshableDataSource<>(
        ruleFile, source -> JSON.parseObject(source, new TypeReference<List<AuthorityRule>>() {})
    );
    AuthorityRuleManager.register2Property(ds.getProperty());

    WritableDataSource<List<AuthorityRule>> wds = new FileWritableDataSource<>(ruleFile, this::encodeJson);
    WritableDataSourceRegistry.registerAuthorityDataSource(wds);
}
```

```
// SystemRule
1 usage
private void readWriteRuleFileSystem(String ruleFilePath) throws IOException {
    String ruleFile = ruleFilePath + "/system-rule.json";
    createRuleFile(ruleFile);
    ReadableDataSource<String, List<SystemRule>> ds = new FileRefreshableDataSource<>(
        ruleFile, source -> JSON.parseObject(source, new TypeReference<List<SystemRule>>() {})
    );
    SystemRuleManager.register2Property(ds.getProperty());

    WritableDataSource<List<SystemRule>> wds = new FileWritableDataSource<>(ruleFile, this::encodeJson);
    WritableDataSourceRegistry.registerSystemDataSource(wds);
}
```

```
// ParamFlowRule
1 usage
private void readWriteRuleFileParam(String ruleFilePath) throws IOException {
    String ruleFile = ruleFilePath + "/param-rule.json";
    createRuleFile(ruleFile);
    ReadableDataSource<String, List<ParamFlowRule>> ds = new FileRefreshableDataSource<>(
        ruleFile, source -> JSON.parseObject(source, new TypeReference<List<ParamFlowRule>>() {})
    );
    ParamFlowRuleManager.register2Property(ds.getProperty());

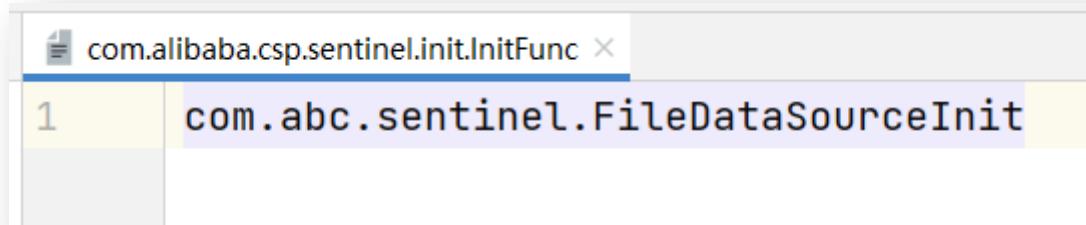
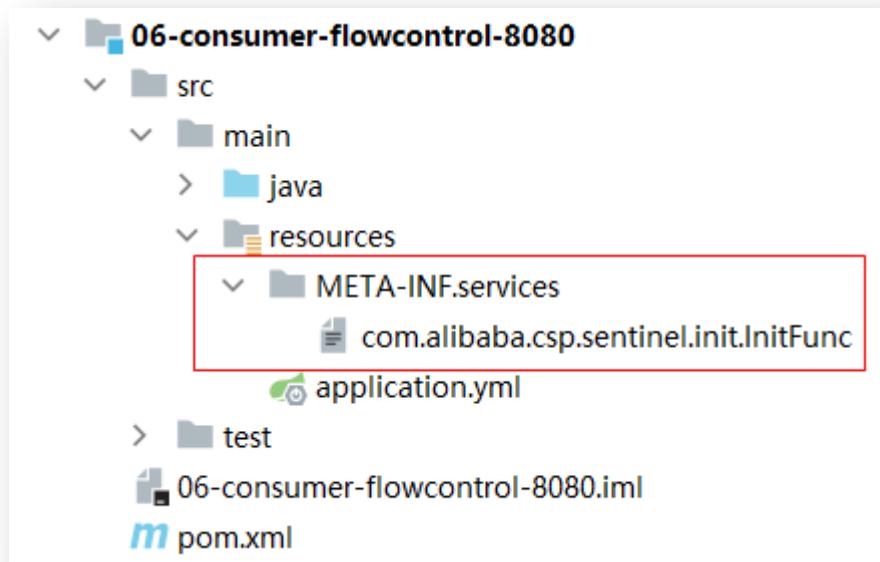
    WritableDataSource<List<ParamFlowRule>> wds = new FileWritableDataSource<>(ruleFile, this::encodeJson);
    ModifyParamFlowRulesCommandHandler.setWritableDataSource(wds);
}
```

```
5 usages
private void createRuleFile(String fileName) throws IOException {
    File ruleFile = new File(fileName);
    if (!ruleFile.exists()) {
        ruleFile.createNewFile();
    }
}

5 usages
private <T> String encodeJson(T t) {
    return JSON.toJSONString(t);
}
```

### (3) 定义 SPI 接口文件

拉模式的 Datasource 扩展采用的是 SPI 服务发现机制。按照 SPI 规范，需要在/resources 目录下新建 META-INF/services 目录，然后在该目录下新建一个文本文件。文件名为扩展类实现接口的全限定性接口名，文件内容则为该接口的实现类的全限定性类名。



### 6.12.3 推模式持久化

#### (1) 复制工程

复制 06-consumer-flowcontrol-8080 工程，并重命名为 06-consumer-persist-8080。

## (2) 删除代码

删除原来定义的 `FileDataSourceInit` 类，删除启动类中所有规则相关的代码。删除 `resources` 目录中 `META-INF/services` 目录及其下的文件。

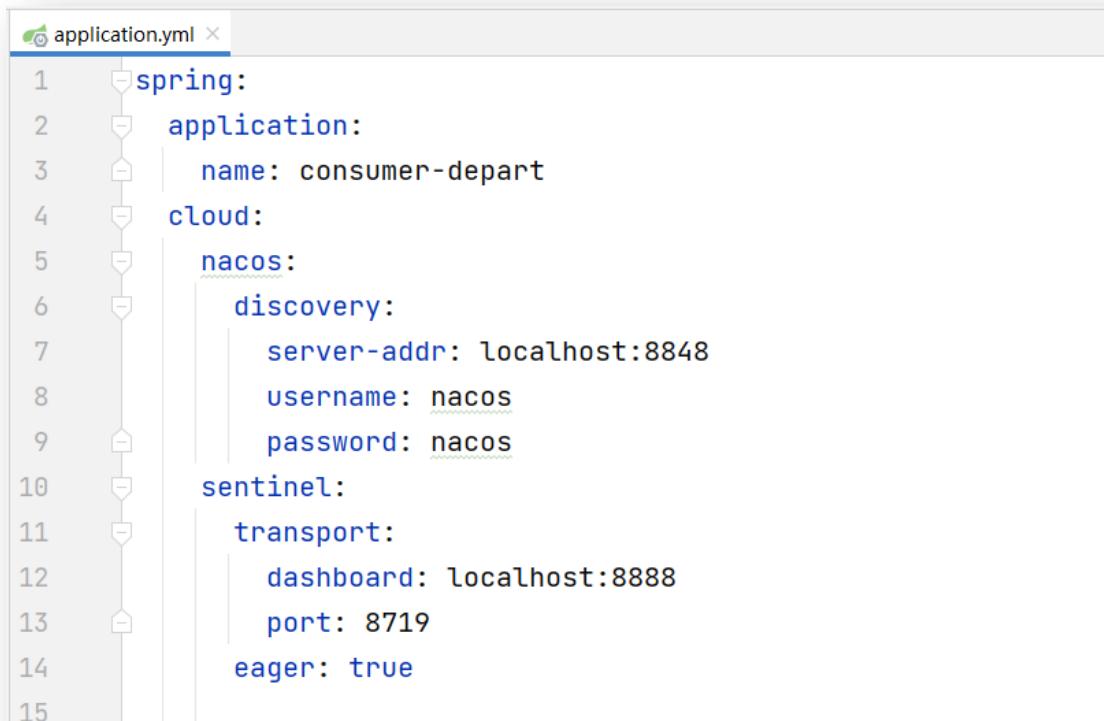
## (3) 添加依赖

要保证有以下几个依赖。

- sentinel 依赖
- sentinel 的 nacos 数据源依赖
- nacos-discovery 依赖

```
<!--sentinel 的nacos 数据源依赖-->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

## (4) 修改配置文件



```
application.yml
1   spring:
2     application:
3       name: consumer-depart
4     cloud:
5       nacos:
6         discovery:
7           server-addr: localhost:8848
8           username: nacos
9           password: nacos
10        sentinel:
11          transport:
12            dashboard: localhost:8888
13            port: 8719
14            eager: true
15
```

```
16
17     datasource:
18         flows:
19             nacos:
20                 server-addr: localhost:8848
21                 username: nacos
22                 password: nacos
23                 rule-type: flow
24                 data-id: ${spring.application.name}-flow-rules
25                 data-type: json
26
27         degrades:
28             nacos:
29                 server-addr: localhost:8848
30                 username: nacos
31                 password: nacos
32                 rule-type: degrade
33                 data-id: ${spring.application.name}-degrade-rules
34                 data-type: json
35
```

## (5) Nacos 中新增规则

### A、添加流控规则文件

在 Nacos Config 中添加流控规则的配置文件。



The screenshot shows the Nacos 2.2.0 configuration management interface. The left sidebar has 'NACOS 2.2.0' at the top, followed by '配置管理' (Configuration Management), '配置列表' (Configuration List) which is selected, '历史版本' (History Versions), '监听查询' (Listener Query), '服务管理' (Service Management), and '权限控制' (Permission Control). The main area is titled '配置管理' (Configuration Management) with a 'public' filter. A red box highlights the '创建配置' (Create Configuration) button. Below it are tabs for 'Data ID' (已开启默认模糊查询), 'Group' (已开启默认模糊查询), and '默认'. A message says '查询到 0 条满足要求的配置' (0 configurations found). A table below shows columns for checked status, Data ID, and Group.

### 新建配置

\* Data ID: consumer-depart-flow-rules.json

\* Group: DEFAULT\_GROUP

[更多高级选项](#)

描述:

配置格式:  TEXT  JSON  XML  YAML  HTML  Properties

\* 配置内容 :

```
1  [ [ { "resource": "getHandle", "limitApp": "default", "grade": 1, "count": 2 } ] ]
```

## B、添加熔断规则文件

再添加熔断规则的配置文件。

## 新建配置

\* Data ID: consumer-depart-degrade-rules.json

\* Group: DEFAULT\_GROUP

[更多高级选项](#)

描述:

配置格式:  TEXT  JSON  XML  YAML  HTML  Properties

\* 配置内容 :

```
1 [ {  
2   "count":200,  
3   "grade":0,  
4   "limitApp":"default",  
5   "minRequestAmount":3,  
6   "resource":"getHandle",  
7   "slowRatioThreshold":0.5,  
8   "statIntervalMs":1000,  
9   "timeWindow":10  
10 } ]  
11 }  
12 ]
```

## 6.13 改造 Sentinel 控制台

基于 Sentinel 控制台的源码改造目的的不同，有两种不同的改造方式。

### 6.13.1 下载源码并导入 Idea

Sentinel Dashboard 的源码是包含在 Sentinel 框架整个源码中的，所以需要从 Sentinel 官网下载整个 Sentinel 源码。不过，下载源码时需要注意下载的版本为 1.8.6。

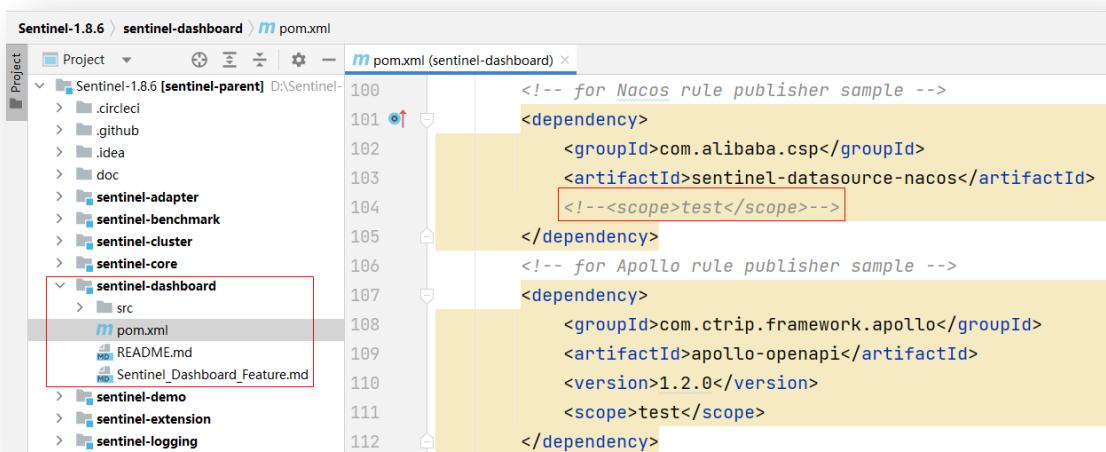
源码下载解压后，可以直接导入到 Idea 中。

## 6.13.2 测试目的改造

该改造方式仅供程序员自己在开发时测试使用，不能用于生产中。

### (1) 修改 pom

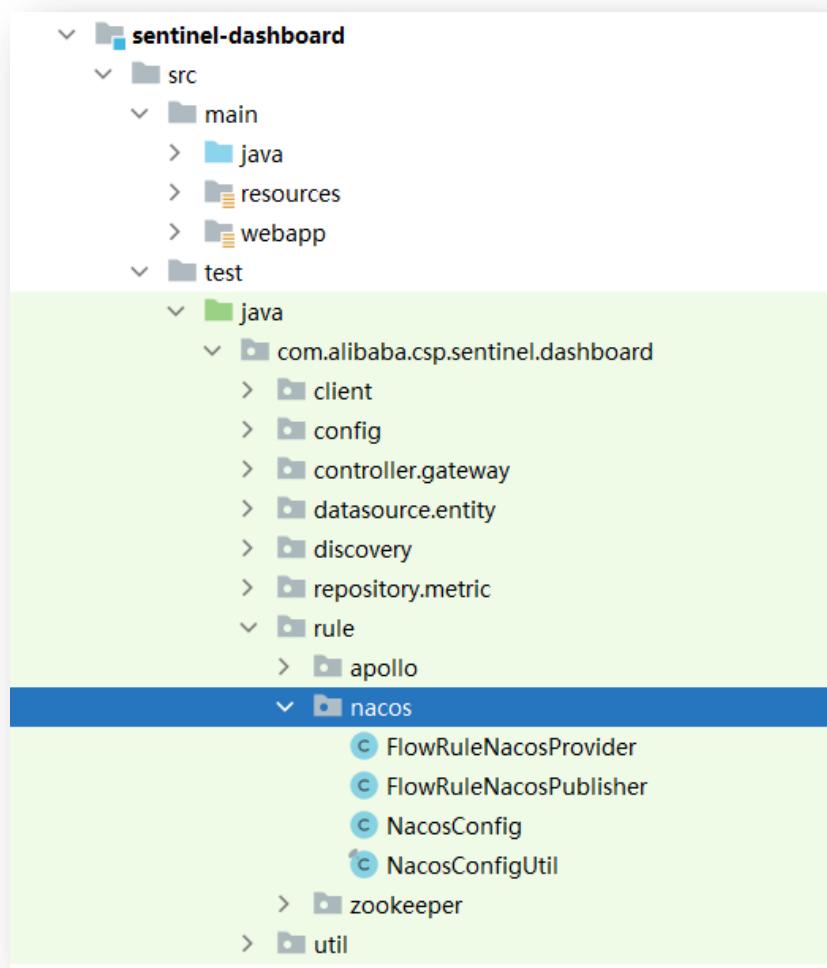
直接在 Idea 中就可以打开该下载的 Sentinel 源码工程。在工程中找到 sentinel-dashboard 模块中的 pom 文件，将其中的 sentinel-datasource-nacos 依赖中的 test 依赖范围注释掉，以保证打包时可以将该依赖打包到工程 jar 包中。

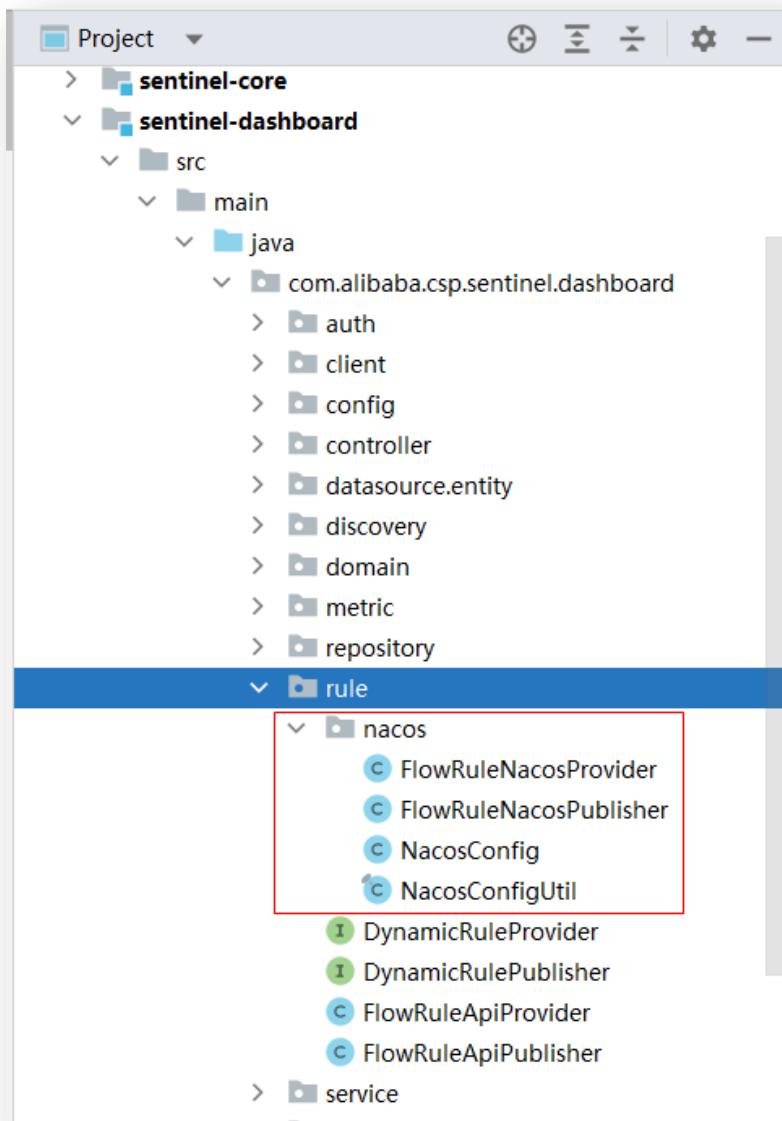


```
<!-- for Nacos rule publisher sample -->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
    <!--<scope>test</scope>-->
</dependency>
<!-- for Apollo rule publisher sample -->
<dependency>
    <groupId>com.ctrip.framework.apollo</groupId>
    <artifactId>apollo-openapi</artifactId>
    <version>1.2.0</version>
    <scope>test</scope>
</dependency>
```

### (2) 目录复制

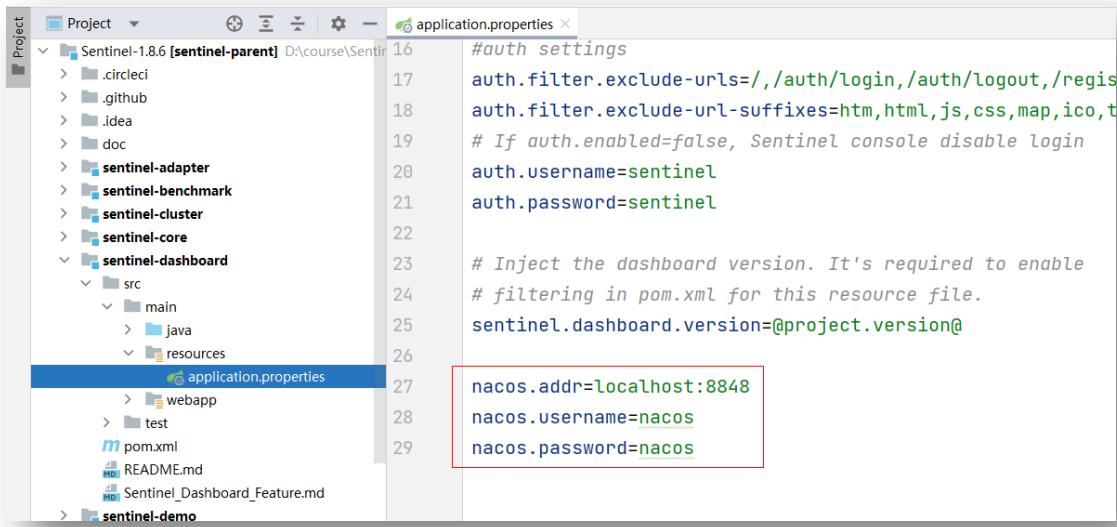
将 sentinel-dashboard 模块的 test 下的 rule/nacos 目录整体复制到 main 下的如下 rule 目录中。





### (3) 修改配置文件

在 sentinel-dashboard 模块的配置文件 application.properties 中添加 nacos 服务器的地址、username 与 password。



```

#auth settings
auth.filter.exclude-urls=/, /auth/login, /auth/logout, /register
auth.filter.exclude-url-suffixes=htm, html, js, css, map, ico, t
# If auth.enabled=false, Sentinel console disable login
auth.username=sentinel
auth.password=sentinel

# Inject the dashboard version. It's required to enable
# filtering in pom.xml for this resource file.
sentinel.dashboard.version=@project.version@

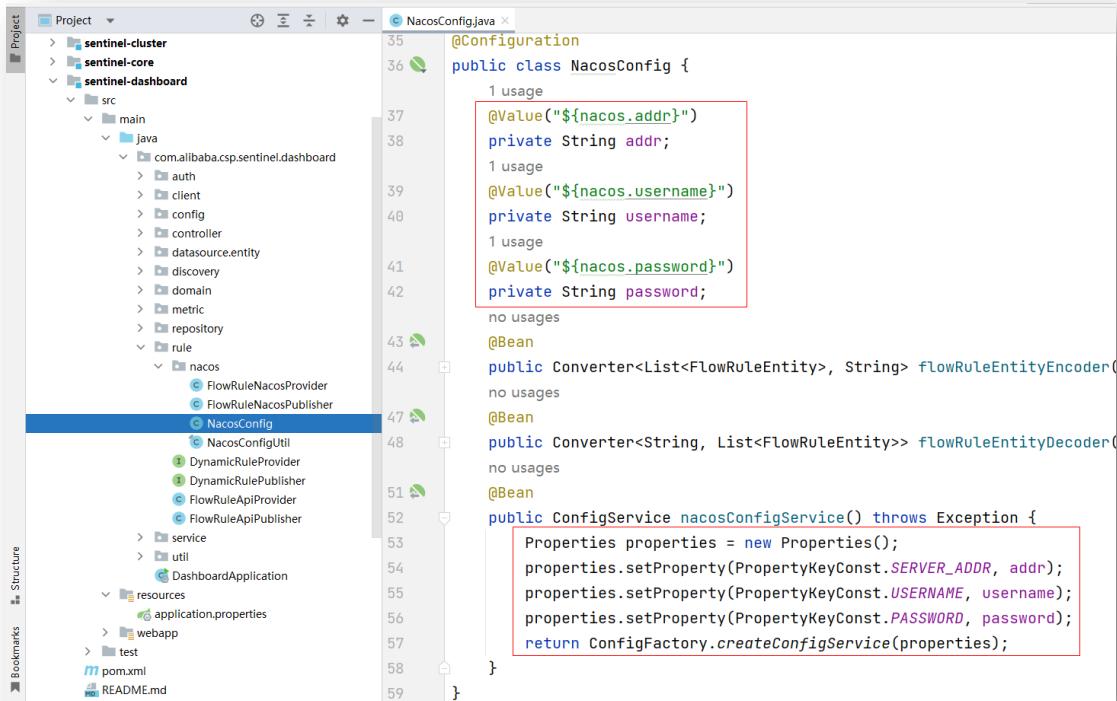
nacos.addr=localhost:8848
nacos.username=nacos
nacos.password=nacos

```

## (4) 修改 NacosConfig 类

找到前面刚添加的 rule.nacos 包中的 NacosConfig 类，该类用于获取 Nacos 配置中心中的数据。对其进行修改，指定 Nacos 配置中心的地址。

找到前面刚添加的 rule.nacos 包中的 NacosConfig 类，该类用于获取 Nacos 配置中心中的数据。对其进行修改，指定 Nacos 配置中心的地址、username 与 password。



```

@Configuration
public class NacosConfig {
    @Value("${nacos.addr}")
    private String addr;
    @Value("${nacos.username}")
    private String username;
    @Value("${nacos.password}")
    private String password;
}

@Bean
public Converter<List<FlowRuleEntity>, String> flowRuleEntityEncoder() {
    return new FlowRuleNacosEncoder();
}

@Bean
public Converter<String, List<FlowRuleEntity>> flowRuleEntityDecoder() {
    return new FlowRuleNacosDecoder();
}

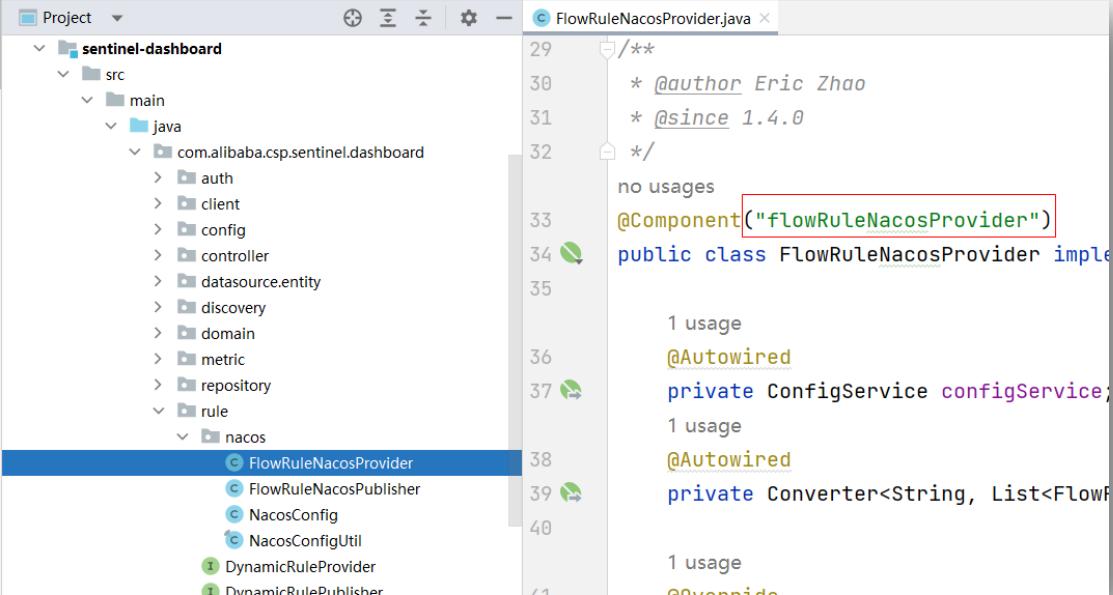
@Bean
public ConfigService nacosConfigService() throws Exception {
    Properties properties = new Properties();
    properties.setProperty(PropertyKeyConst.SERVER_ADDR, addr);
    properties.setProperty(PropertyKeyConst.USERNAME, username);
    properties.setProperty(PropertyKeyConst.PASSWORD, password);
    return ConfigFactory.createConfigService(properties);
}

```

## (5) 修改控制器类

### A、找到实例名称

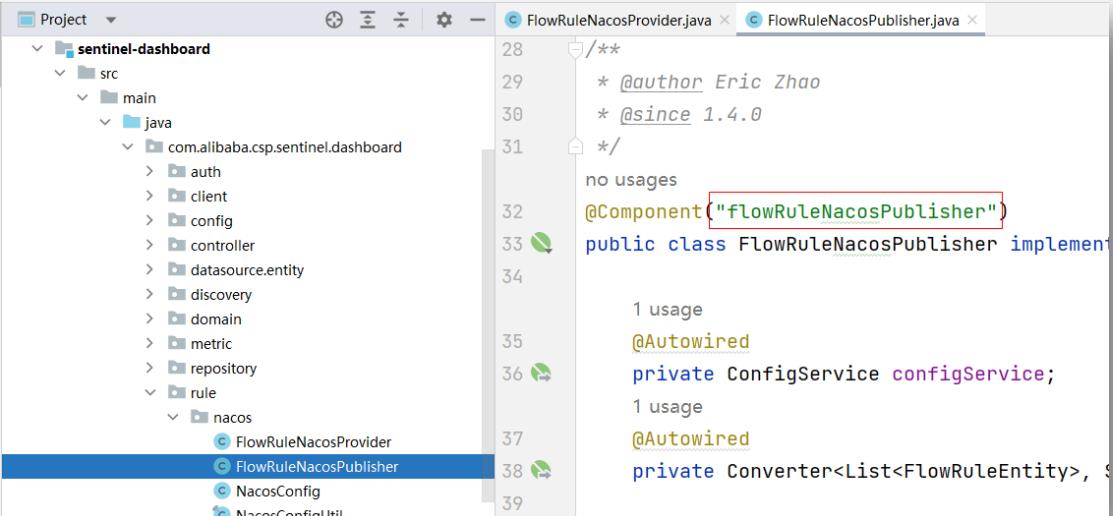
找到前面刚添加的 rule.nacos 包中的 FlowRuleNacosProvider 与 FlowRuleNacosPublisher 类，在其中复制出其实例名称。



```

29 /**
30 * @author Eric Zhao
31 * @since 1.4.0
32 */
33 no usages
34 @Component("flowRuleNacosProvider")
35 public class FlowRuleNacosProvider implements
36
37 1 usage
38 @Autowired
39 private ConfigService configService;
40
41 1 usage
42 @Autowired
43 private Converter<String, List<FlowR
44
45 1 usage
46 @Override
47
48

```



```

28 /**
29 * @author Eric Zhao
30 * @since 1.4.0
31 */
32 no usages
33 @Component("flowRuleNacosPublisher")
34 public class FlowRuleNacosPublisher implements
35
36 1 usage
37 @Autowired
38 private ConfigService configService;
39
40 1 usage
41 @Autowired
42 private Converter<List<FlowRuleEntity>, S
43
44

```

## B、修改实例名称

直接在 controller.v2.FlowControllerV2 类中修改第 52 与 55 行中要注入实例的名称为前面复制的两个名称。



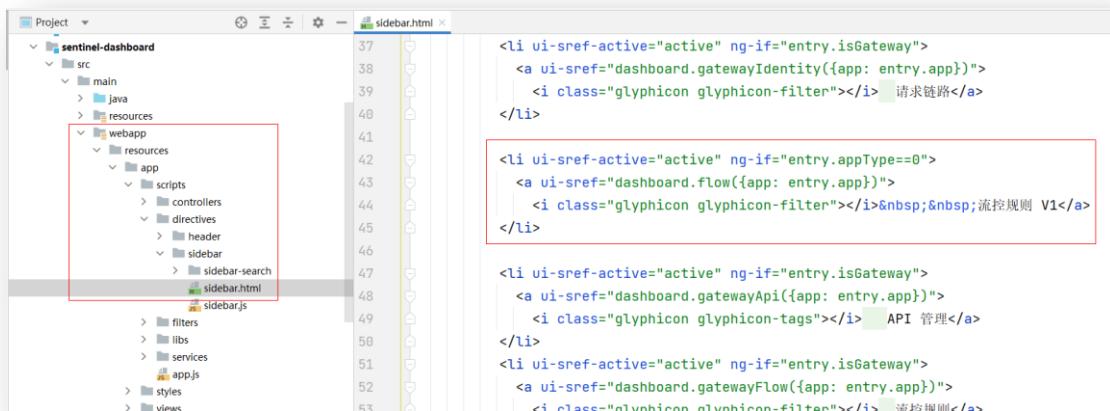
```

1 usage
@.Autowired
@Qualifier("flowRuleNacosProvider")
private DynamicRuleProvider<List<FlowRuleEntity>> ruleProvider;
1 usage
@.Autowired
@Qualifier("flowRuleNacosPublisher")
private DynamicRulePublisher<List<FlowRuleEntity>> rulePublisher;
no usages
@GetMapping("/rules")
@AuthAction(PrivilegeType.READ_RULE)

```

## (6) 修改页面

修改 src/main/webapp/resources/app/scripts/directives/sidebar/sidebar.html 中的代码：将原本注释掉的第 42-45 行的注释去掉。



```

<li ui-sref-active="active" ng-if="entry.isGateway">
    <a ui-sref="dashboard.gatewayIdentity({app: entry.app})">
        <i class="glyphicon glyphicon-filter"></i> 请求链路</a>
    </li>

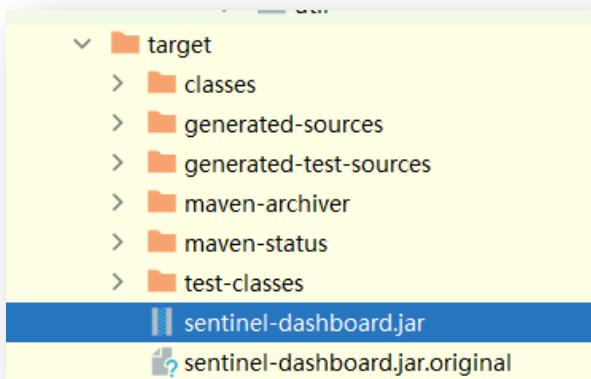
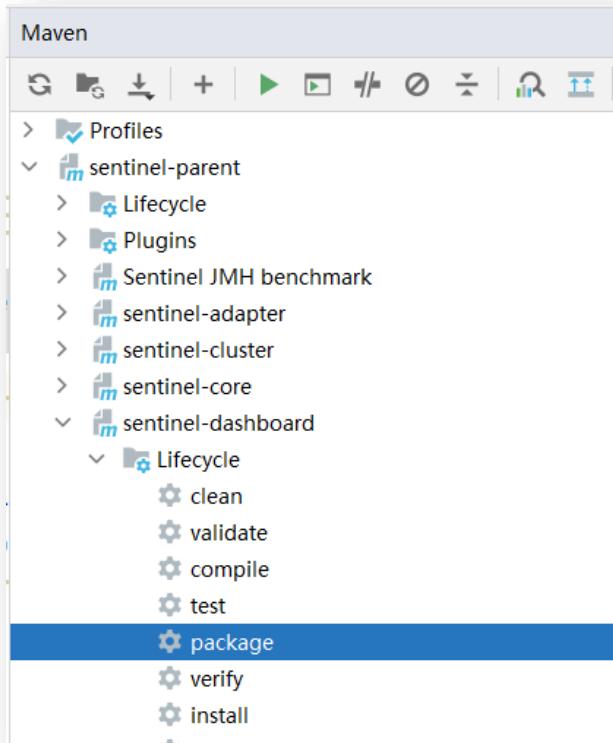
<li ui-sref-active="active" ng-if="entry.appType==0">
    <a ui-sref="dashboard.flow({app: entry.app})">
        <i class="glyphicon glyphicon-filter"></i> 流控规则 V1</a>
    </li>

<li ui-sref-active="active" ng-if="entry.isGateway">
    <a ui-sref="dashboard.gatewayApi({app: entry.app})">
        <i class="glyphicon glyphicon-tags"></i> API 管理</a>
    </li>
<li ui-sref-active="active" ng-if="entry.isGateway">
    <a ui-sref="dashboard.gatewayFlow({app: entry.app})">
        <i class="glyphicon glyphicon-filter"></i> 流控脚本</a>
    </li>

```

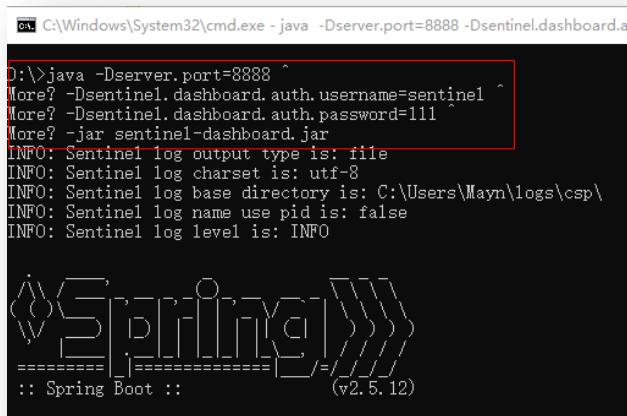
## (7) 重新打包 dashboard

对 Sentinel Dashboard 进行重新打包。



## (8) 启动新的 dashboard

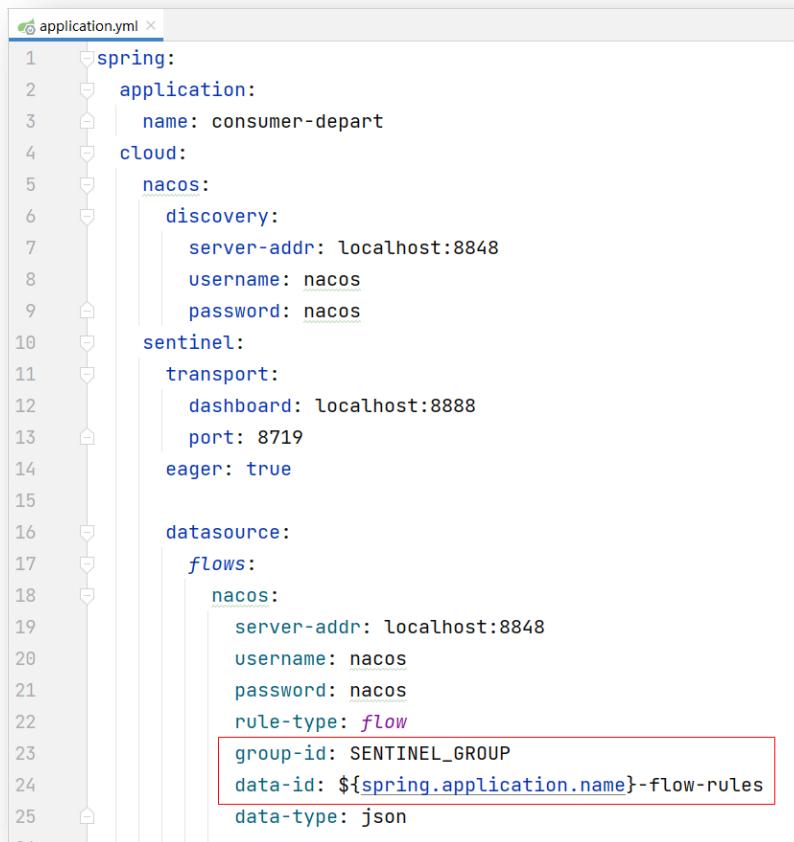
重新打包修改过的 sentinel-dashboard 模块，然后重新启动新的 dashboard 的 jar 包。



```
D:\>java -Dserver.port=8888 ^  
More? -Dsentry.dashboard.auth.username=sentinel ^  
More? -Dsentry.dashboard.auth.password=111 ^  
More? -jar sentinel-dashboard.jar  
INFO: Sentry log output type is: file  
INFO: Sentry log charset is: utf-8  
INFO: Sentry log base directory is: C:\Users\Mayn\logs\csp\  
INFO: Sentry log name use pid is: false  
INFO: Sentry log level is: INFO  
  
:: Spring Boot ::  
(v2.5.12)
```

## (9) 修改微服务应用

通过前面的官网文档可以看出，其要求 groupId 必须是 SENTINEL\_GROUP，流控规则的数据 id 名称必须为 \${spring.application.name}-flow-rules，所以需要修改应用中相应的配置项。直接修改 06-consumer-persist-8080 工程的配置文件即可。

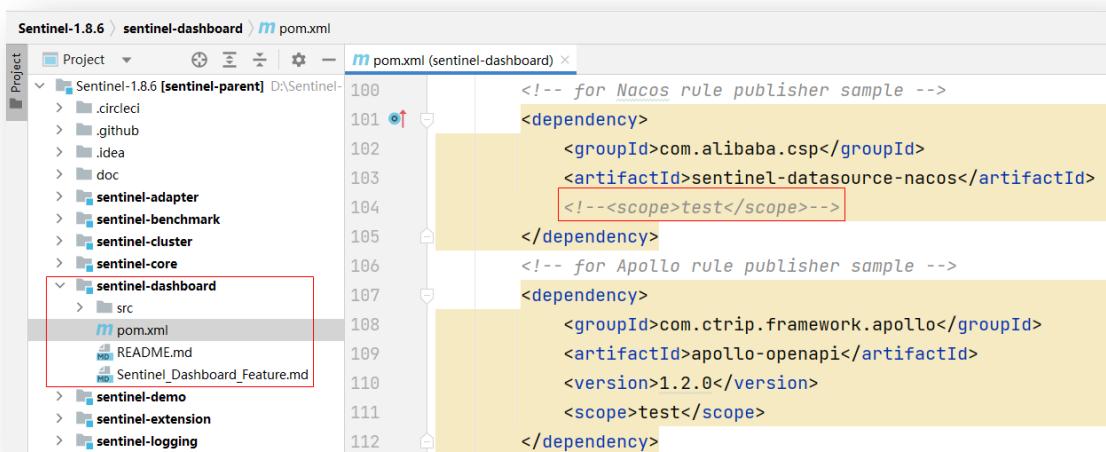


### 6.13.3 生产目的改造

该改造方式可应用于 FlowRule、DegradeRule、SystemRule、AuthorityRule、ParamFlowRule。

#### (1) 修改 pom

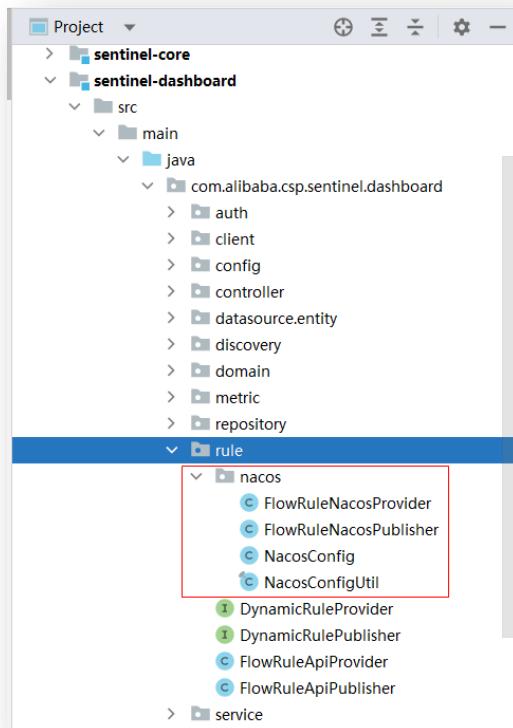
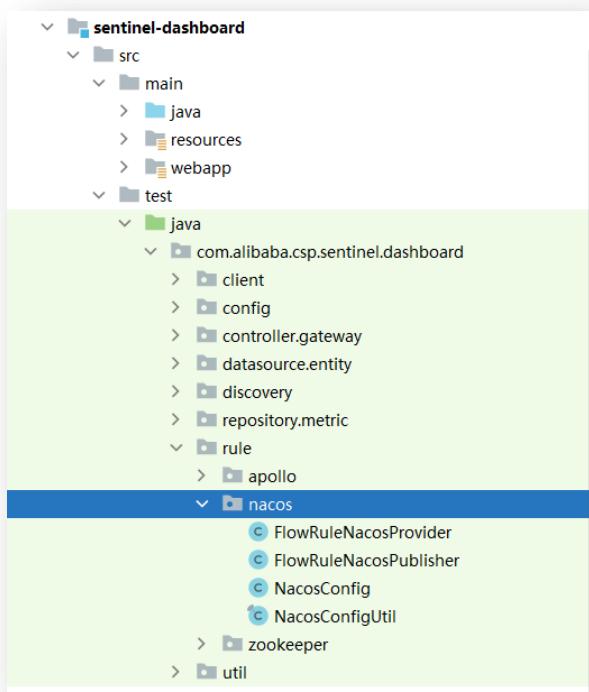
直接在 Idea 中就可以打开该下载的 Sentinel 源码工程。在工程中找到 sentinel-dashboard 模块中的 pom 文件，将其中的 sentinel-datasource-nacos 依赖中的 test 依赖范围注释掉，以保证打包时可以将该依赖打包到工程 jar 包中。



#### (2) 类复制/移动

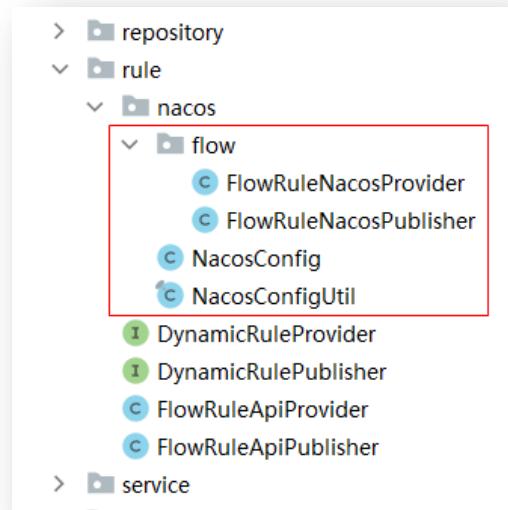
##### A、从 test 到 main 复制

将 sentinel-dashboard 模块的 test 下的 rule/nacos 目录整体复制到 main 下的如下 rule 目录中。



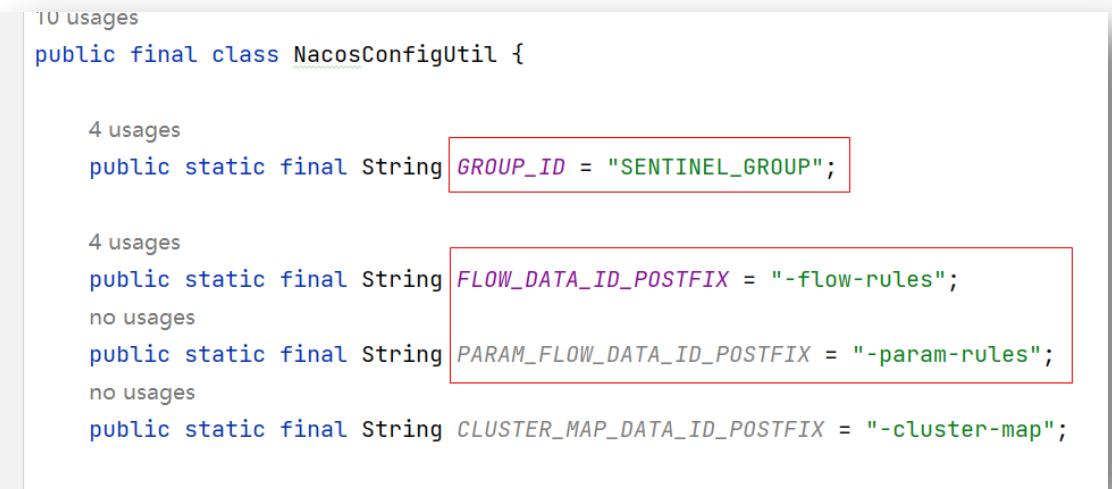
## B、移动读写类

在 main 下的 rule.nacos 包下新建子包 flow，并将 rule.nacos 包下的 FlowRuleNacosProvider 类与 FlowRuleNacosPublisher 类移动到 rule.nacos.flow 包下。



### (3) 修改 NacosConfigUtil 类

打开 NacosConfigUtil 类，除了 GROUP\_ID 外，只能找到我们需要的 5 种规则中的 2 种规则文件后缀。



10 usages

```
public final class NacosConfigUtil {
```

4 usages

```
    public static final String GROUP_ID = "SENTINEL_GROUP";
```

4 usages

```
    public static final String FLOW_DATA_ID_POSTFIX = "-flow-rules";
```

no usages

```
    public static final String PARAM_FLOW_DATA_ID_POSTFIX = "-param-rules";
```

no usages

```
    public static final String CLUSTER_MAP_DATA_ID_POSTFIX = "-cluster-map";
```

添加另外三种规则文件的后缀。

```
10 usages
public final class NacosConfigUtil {

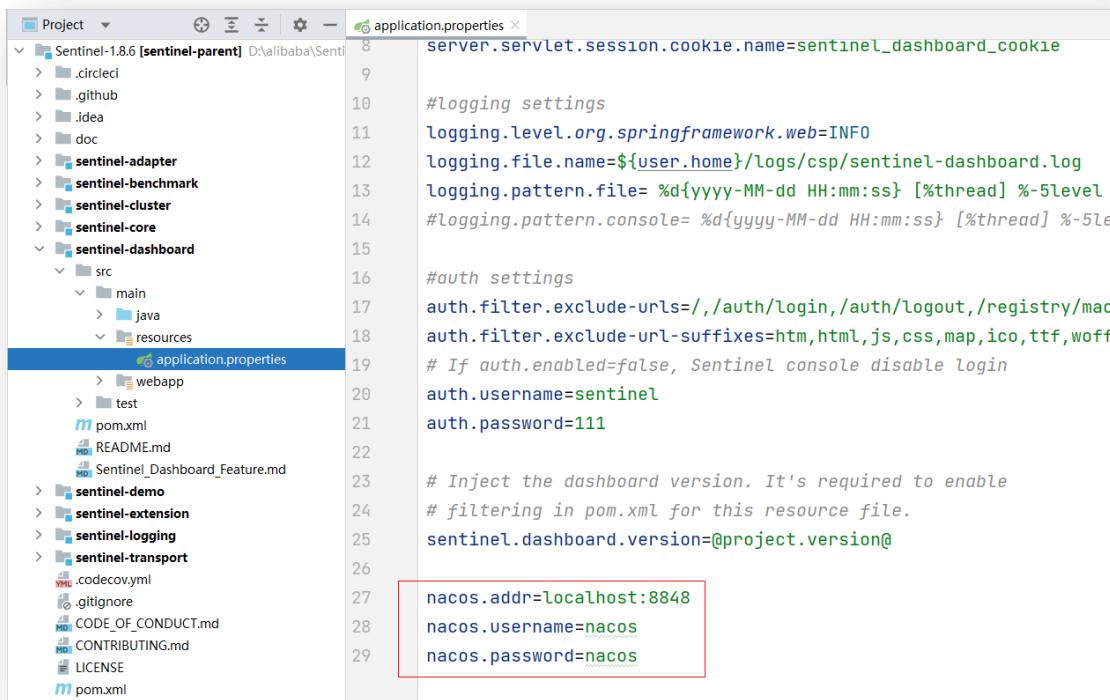
    4 usages
    public static final String GROUP_ID = "SENTINEL_GROUP";

    4 usages
    public static final String FLOW_DATA_ID_POSTFIX = "-flow-rules";
    no usages
    public static final String PARAM_FLOW_DATA_ID_POSTFIX = "-param-rules";
    no usages
    public static final String DEGRADE_DATA_ID_POSTFIX = "-degrade-rules";
    no usages
    public static final String SYSTEM_DATA_ID_POSTFIX = "-system-rules";
    no usages
    public static final String AUTHORITY_DATA_ID_POSTFIX = "-authority-rules";
    no usages
    public static final String CLUSTER_MAP_DATA_ID_POSTFIX = "-cluster-map";

    ...
}
```

#### (4) 修改配置文件

在 sentinel-dashboard 模块的配置文件 application.properties 中添加 nacos 服务器的地址。



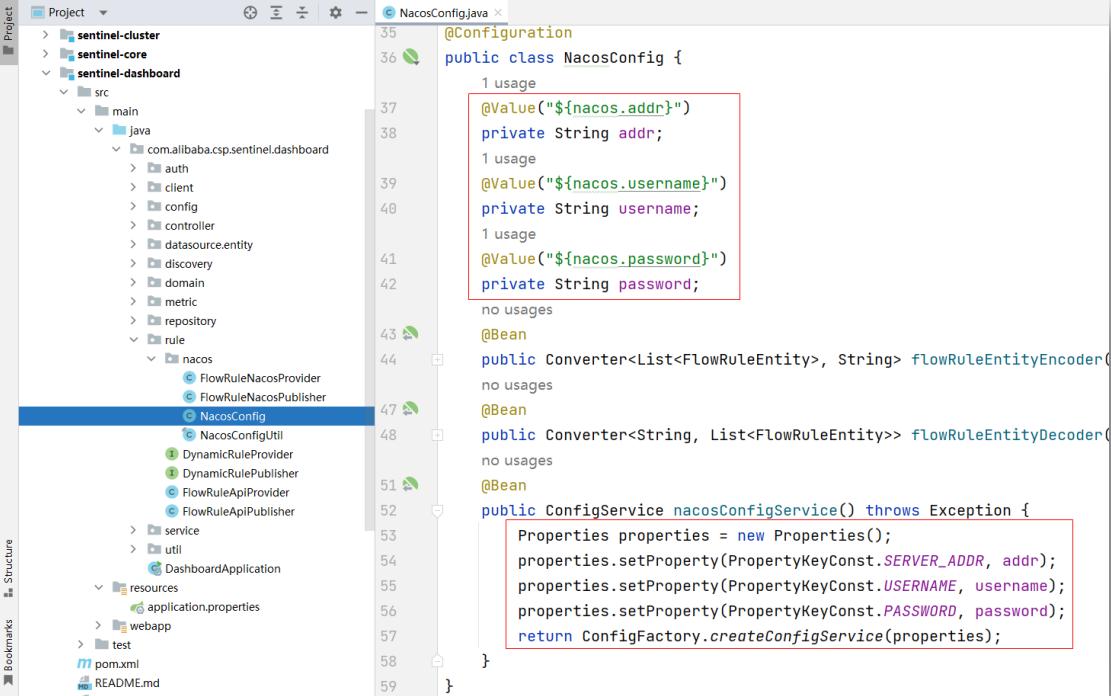
```
server.servlet.session.cookie.name=sentinel_dashboard_cookie
#logging settings
logging.level.org.springframework.web=INFO
logging.file.name=${user.home}/logs/csp/sentinel-dashboard.log
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %
#logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5le
#auth settings
auth.filter.exclude-urls=/,/auth/login,/auth/logout,/registry/mac
auth.filter.exclude-url-suffixes=htm,html,js,css,map,ico,ttf,woff
# If auth.enabled=false, Sentinel console disable login
auth.username=sentinel
auth.password=111
# Inject the dashboard version. It's required to enable
# filtering in pom.xml for this resource file.
sentinel.dashboard.version=@project.version@

nacos.addr=localhost:8848
nacos.username=nacos
nacos.password=nacos
```

## (5) 修改 NacosConfig 类

对于 NacosConfig 类的修改，主要有两类：Nacos 服务器相关属性，添加另外 4 种规则的编码转换器与解码转换器。

## A、Nacos 属性相关



```

@Configuration
public class NacosConfig {
    private String addr;
    private String username;
    private String password;

    @Bean
    public Converter<List<FlowRuleEntity>, String> flowRuleEntityEncoder() {
        return JSON::toJSONString;
    }

    @Bean
    public Converter<String, List<FlowRuleEntity>> flowRuleEntityDecoder() {
        return s -> JSON.parseArray(s, FlowRuleEntity.class);
    }

    public ConfigService nacosConfigService() throws Exception {
        Properties properties = new Properties();
        properties.setProperty(PropertyKeyConst.SERVER_ADDR, addr);
        properties.setProperty(PropertyKeyConst.USERNAME, username);
        properties.setProperty(PropertyKeyConst.PASSWORD, password);
        return ConfigFactory.createConfigService(properties);
    }
}

```

## B、添加 Authority 规则转换器

复制自带的 Flow 规则的两个转换器方法，在此基础上修改。

```

// authority
no usages
@Bean
public Converter<List<AuthorityRuleEntity>, String> authorityRuleEntityEncoder() {
    return JSON::toJSONString;
}

no usages
@Bean
public Converter<String, List<AuthorityRuleEntity>> authorityRuleEntityDecoder() {
    return s -> JSON.parseArray(s, AuthorityRuleEntity.class);
}

```

## C、添加 Degrade 规则转换器

```
// degrade
no usages
@Bean
public Converter<List<DegradeRuleEntity>, String> degradeRuleEntityEncoder() {
    return JSON::toJSONString;
}

no usages
@Bean
public Converter<String, List<DegradeRuleEntity>> degradeRuleEntityDecoder() {
    return s -> JSON.parseArray(s, DegradeRuleEntity.class);
}
```

## D、添加 System 规则转换器

```
// system
no usages
@Bean
public Converter<List<SystemRuleEntity>, String> systemRuleEntityEncoder() {
    return JSON::toJSONString;
}

no usages
@Bean
public Converter<String, List<SystemRuleEntity>> systemRuleEntityDecoder() {
    return s -> JSON.parseArray(s, SystemRuleEntity.class);
}
```

## E、添加 ParamFlow 规则转换器

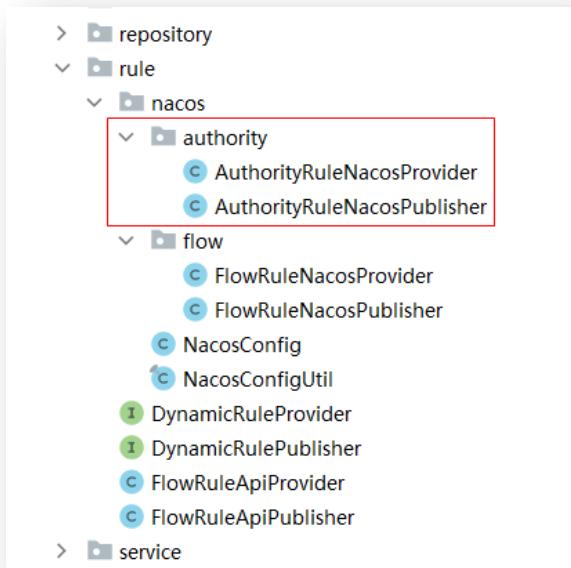
```
// paramFlow
no usages
@Bean
public Converter<List<ParamFlowRuleEntity>, String> paramRuleEntityEncoder() {
    return JSON::toString;
}

no usages
@Bean
public Converter<String, List<ParamFlowRuleEntity>> paramRuleEntityDecoder() {
    return s -> JSON.parseArray(s, ParamFlowRuleEntity.class);
}
```

## (6) 添加读写类

### A、添加 Authority 规则读写类

在 rule.nacos 包下新建一个子包 authority，然后将 rule.nacos.flow 包中的两个类复制到该子包中，再分别重命名为 AuthorityRuleNacosProvider 与 AuthorityRuleNacosPublisher。



### a、修改 AuthorityRuleNacosProvider 类

将原来所有的 Flow 全部替换为 Authrotiry 即可。该类用于将 Nacos 中 JSON 格式的规则文件转换为规则实体对象。即用于“读取” Nacos 中的数据的。



```
110 usages
@Component("authorityRuleNacosProvider")
public class AuthorityRuleNacosProvider implements DynamicRuleProvider<List<AuthorityRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<String, List<AuthorityRuleEntity>> converter;

    1 usage
    @Override
    public List<AuthorityRuleEntity> getRules(String appName) throws Exception {
        String rules = configService.getConfig(appName + NacosConfigUtil.AUTHORITY_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, 3000);
        if (StringUtil.isEmpty(rules)) {
            return new ArrayList<>();
        }
        return converter.convert(rules);
    }
}
```

### b、修改 AuthorityRuleNacosPublisher 类

将原来所有的 Flow 全部替换为 Authrotiry 即可。该类用于将规则实体对象转换为 JSON 字符串后写入到 Nacos 中相应的规则文件中。即，其充当的是规则的“发布者”。

```

no usages
@Component("authorityRuleNacosPublisher")
public class AuthorityRuleNacosPublisher implements DynamicRulePublisher<List<AuthorityRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<List<AuthorityRuleEntity>, String> converter;

    @Override
    public void publish(String app, List<AuthorityRuleEntity> rules) throws Exception {
        AssertUtil.notEmpty(app, "app name cannot be empty");
        if (rules == null) {
            return;
        }
        configService.publishConfig(app + NacosConfigUtil.AUTHORITY_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, converter.convert(rules));
    }
}

```

## B、添加 Degrade 规则读写类

在 rule.nacos 包下新建一个子包 degrade, 然后将 rule.nacos.flow 包中的两个类复制到该子包中，再分别重命名为 DegradeRuleNacosProvider 与 DegradeRuleNacosPublisher。

### a、修改 DegradeRuleNacosProvider 类

```

no usages
@Component("degradeRuleNacosProvider")
public class DegradeRuleNacosProvider implements DynamicRuleProvider<List<DegradeRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<String, List<DegradeRuleEntity>> converter;

    @Override
    public List<DegradeRuleEntity> getRules(String appName) throws Exception {
        String rules = configService.getConfig(appName + NacosConfigUtil.DEGRADE_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, 3000);
        if (StringUtil.isEmpty(rules)) {
            return new ArrayList<>();
        }
        return converter.convert(rules);
    }
}

```

## b、修改 DegradeRuleNacosPublisher 类

```
no usages
@Component("dgradeRuleNacosPublisher")
public class DegradeRuleNacosPublisher implements DynamicRulePublisher<List<DegradeRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<List<DegradeRuleEntity>, String> converter;

    @Override
    public void publish(String app, List<DegradeRuleEntity> rules) throws Exception {
        AssertUtil.notEmpty(app, "app name cannot be empty");
        if (rules == null) {
            return;
        }
        configService.publishConfig(app + NacosConfigUtil.DEGRADE_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, converter.convert(rules));
    }
}
```

## c、添加 System 规则读写类

在 rule.nacos 包下新建一个子包 system，然后将 rule.nacos.flow 包中的两个类复制到该子包中，再分别重命名为 SystemRuleNacosProvider 与 SystemRuleNacosPublisher。

## a、修改 SystemRuleNacosProvider 类

```
no usages
@Component("systemRuleNacosProvider")
public class SystemRuleNacosProvider implements DynamicRuleProvider<List<SystemRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<String, List<SystemRuleEntity>> converter;

    @Override
    public List<SystemRuleEntity> getRules(String appName) throws Exception {
        String rules = configService.getConfig(appName + NacosConfigUtil.SYSTEM_DATA_ID_POSTFIX,
                                                NacosConfigUtil.GROUP_ID, 3000);
        if (StringUtil.isEmpty(rules)) {
            return new ArrayList<>();
        }
        return converter.convert(rules);
    }
}
```

## b、修改 SystemRuleNacosPublisher 类

```
no usages
@Component("systemRuleNacosPublisher")
public class SystemRuleNacosPublisher implements DynamicRulePublisher<List<SystemRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<List<SystemRuleEntity>, String> converter;

    @Override
    public void publish(String app, List<SystemRuleEntity> rules) throws Exception {
        AssertUtil.notEmpty(app, "app name cannot be empty");
        if (rules == null) {
            return;
        }
        configService.publishConfig(app + NacosConfigUtil.SYSTEM_DATA_ID_POSTFIX,
                                   NacosConfigUtil.GROUP_ID, converter.convert(rules));
    }
}
```

## D、添加 ParamFlow 规则读写类

在 rule.nacos 包下新建一个子包 param，然后将 rule.nacos.flow 包中的两个类复制到该子包中，再分别重命名为 ParamRuleNacosProvider 与 ParamRuleNacosPublisher。

### a、修改 ParamRuleNacosProvider 类

```
no usages
@Component("paramRuleNacosProvider")
public class ParamRuleNacosProvider implements DynamicRuleProvider<List<ParamFlowRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<String, List<ParamFlowRuleEntity>> converter;

    @Override
    public List<ParamFlowRuleEntity> getRules(String appName) throws Exception {
        String rules = configService.getConfig(appName + NacosConfigUtil.PARAM_FLOW_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, 3000);
        if (StringUtil.isEmpty(rules)) {
            return new ArrayList<>();
        }
        return converter.convert(rules);
    }
}
```

## b、修改 ParamRuleNacosPublisher 类

```

no usages
@Component("paramRuleNacosPublisher")
public class ParamRuleNacosPublisher implements DynamicRulePublisher<List<ParamFlowRuleEntity>> {

    1 usage
    @Autowired
    private ConfigService configService;
    1 usage
    @Autowired
    private Converter<List<ParamFlowRuleEntity>, String> converter;

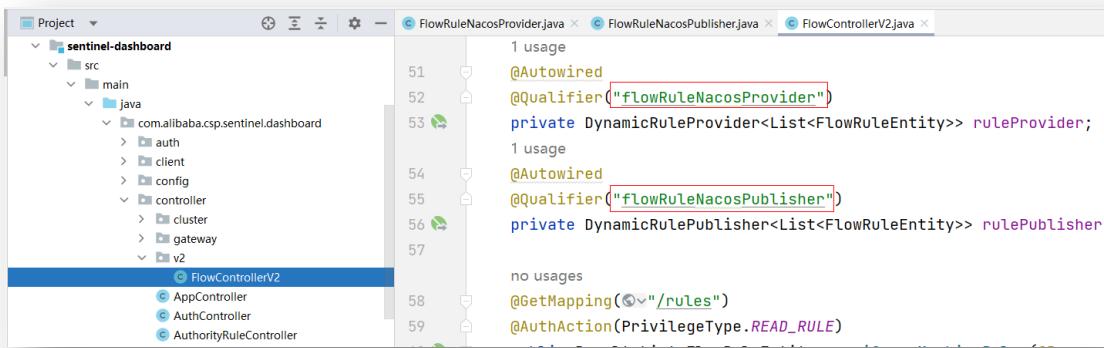
    @Override
    public void publish(String app, List<ParamFlowRuleEntity> rules) throws Exception {
        AssertUtil.notEmpty(app, "app name cannot be empty");
        if (rules == null) {
            return;
        }
        configService.publishConfig(app + NacosConfigUtil.PARAM_FLOW_DATA_ID_POSTFIX,
            NacosConfigUtil.GROUP_ID, converter.convert(rules));
    }
}

```

## (7) 修改处理器类

### A、修改 FlowControllerV2 类

直接在 controller.v2.FlowControllerV2 类中修改第 52 与 55 行中要注入的实例的名称为 Nacos 的 provider 与 publisher。



```

1 usage
@.Autowired
@Qualifier("flowRuleNacosProvider")
private DynamicRuleProvider<List<FlowRuleEntity>> ruleProvider;
1 usage
@.Autowired
@Qualifier("flowRuleNacosPublisher")
private DynamicRulePublisher<List<FlowRuleEntity>> rulePublisher;

no usages
@GetMapping(PathVariable("/rules"))
@AuthAction(PrivilegeType.READ_RULE)

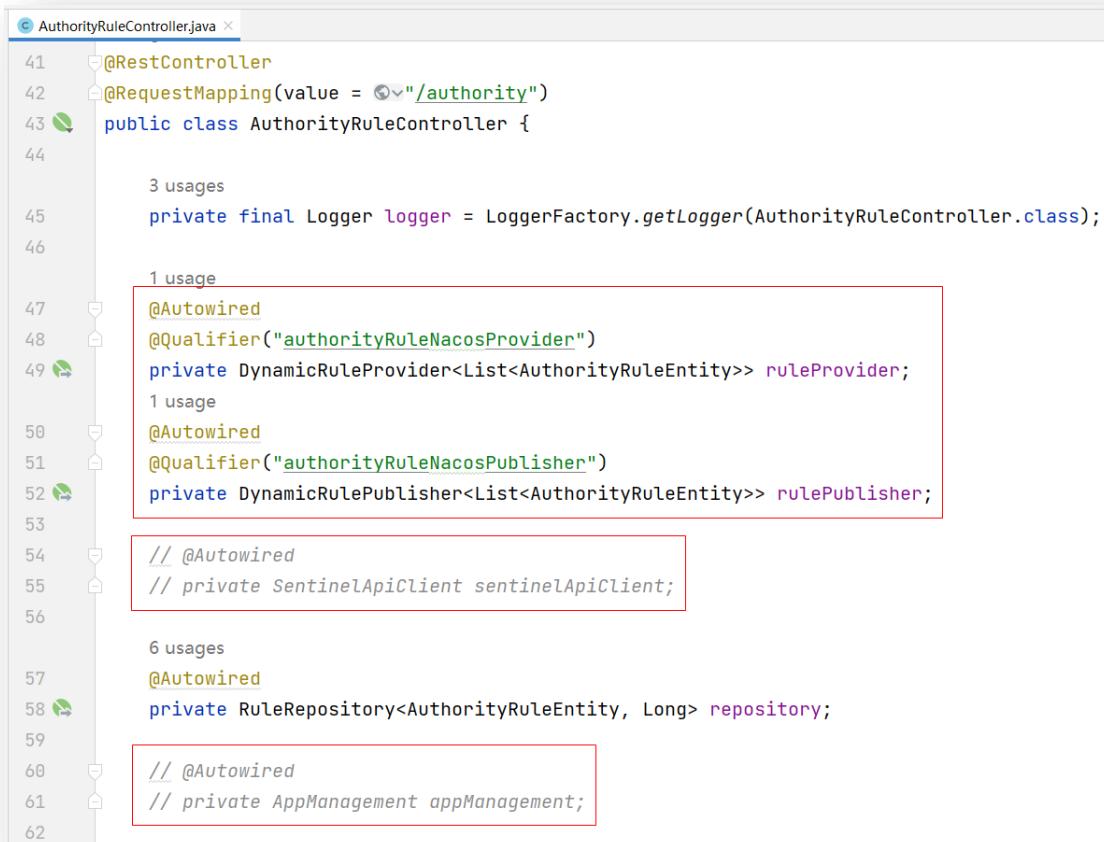
```

## B、修改 AuthorityRuleController 类

修改 controller 包中的 AuthorityRuleController 类。

### a、添加/删除成员变量

在类声明中添加自动注入的 ruleProvider 与 rulePublisher，并将 sentinelApiClient 删除。



```
④ AuthorityRuleController.java ×
41  @RestController
42  @RequestMapping(value = "/authority")
43  public class AuthorityRuleController {
44
45      3 usages
46      private final Logger logger = LoggerFactory.getLogger(AuthorityRuleController.class);
47
48      1 usage
49      @Autowired
50      @Qualifier("authorityRuleNacosProvider")
51      private DynamicRuleProvider<List<AuthorityRuleEntity>> ruleProvider;
52
53      1 usage
54      @Autowired
55      @Qualifier("authorityRuleNacosPublisher")
56      private DynamicRulePublisher<List<AuthorityRuleEntity>> rulePublisher;
57
58      // @Autowired
59      // private SentinelApiClient sentinelApiClient;
60
61      // @Autowired
62      // private AppManagement appManagement;
```

### b、修改 GET 方法

将 sentinelApiClient 删除后报错的用于读取规则的语句删除，然后替换为 ruleProvider 的 getRules()方法，用于从 Nacos 配置中心读取规则配置文件，并转换为规则实体。

```

no usages
63     @GetMapping("rules")
64     @AuthAction(PrivilegeType.READ_RULE)
65     public Result<List<AuthorityRuleEntity>> apiQueryAllRulesForMachine(@RequestParam String app,
66                                         @RequestParam String ip,
67                                         @RequestParam Integer port) {
68         if (StringUtil.isEmpty(app)) {
69             return Result.ofFail(-1, "app cannot be null or empty");
70         }
71         if (StringUtil.isEmpty(ip)) {
72             return Result.ofFail(-1, "ip cannot be null or empty");
73         }
74         if (port == null || port <= 0) {
75             return Result.ofFail(-1, "Invalid parameter: port");
76         }
77         // if (appManagement.isValidMachineOfApp(app, ip)) {
78         //     return Result.ofFail(-1, "given ip does not belong to given app");
79         //}
80         try {
81             // List<AuthorityRuleEntity> rules = sentinelApiClient.fetchAuthorityRulesOfMachine(app, ip);
82             List<AuthorityRuleEntity> rules = ruleProvider.getRules(app);
83             rules = repository.saveAll(rules);
84             return Result.ofSuccess(rules);
85         } catch (Throwable throwable) {

```

### c、修改 publishRules()方法

对 sentinelApiClient 删除后报错的 publishRules()方法进行修改。将报错的用于进行规则实体持久化的语句删除，替换为 rulePublisher 的 publish()方法，用于将规则持久化到 Nacos 配置中心。

不过需要注意的是，rulePublisher.publish()方法没有返回值，且会抛出异常，所以需要将 publishRules()方法的返回值修改为 void，并抛出 Exception 异常。

```

3 usages
194     private void publishRules(String app, String ip, Integer port) throws Exception {
195         List<AuthorityRuleEntity> rules = repository.findAllByMachine(MachineInfo.of(app, ip, port));
196         // return sentinelApiClient.setAuthorityRuleOfMachine(app, ip, port, rules);
197         rulePublisher.publish(app, rules);
198     }
199

```

### d、修改 POST 方法

publishRules()方法的修改会引发 POST、PUT 与 DELETE 方法中对 publishRules()方法引用处的报错。将这些报错语句修改为变更后的 publishRules()方法引用。注意，这些引用所在方法需要抛出异常。

```
no usages
120  @PostMapping(value="/rule")
121  @AuthAction(PrivilegeType.WRITE_RULE)
122  public Result<AuthorityRuleEntity> apiAddAuthorityRule(@RequestBody AuthorityRuleEntity entity) throws Exception {
123      Result<AuthorityRuleEntity> checkResult = checkEntityInternal(entity);
124      if (checkResult != null) {
125          return checkResult;
126      }
127      entity.setId(null);
128      Date date = new Date();
129      entity.setGmtCreate(date);
130      entity.setGmtModified(date);
131      try {
132          entity = repository.save(entity);
133      } catch (Throwable throwable) {
134          logger.error("Failed to add authority rule", throwable);
135          return Result.ofThrowable(-1, throwable);
136      }
137      // if (!publishRules(entity.getApp(), entity.getIp(), entity.getPort())) {
138      //     logger.info("Publish authority rules failed after rule add");
139      // }
140      publishRules(entity.getApp(), entity.getIp(), entity.getPort());
141      return Result.ofSuccess(entity);
142 }
```

## e、修改 PUT 方法

```
no usages
144  @PutMapping(value="/rule/{id}")
145  @AuthAction(PrivilegeType.WRITE_RULE)
146  public Result<AuthorityRuleEntity> apiUpdateParamFlowRule(@PathVariable("id") Long id,
147                                                               @RequestBody AuthorityRuleEntity entity) throws Exception {
148      if (id == null || id <= 0) {
149          return Result.ofFail(-1, "Invalid id");
150      }
151      Result<AuthorityRuleEntity> checkResult = checkEntityInternal(entity);
152      if (checkResult != null) {
153          return checkResult;
154      }
155      try {
156          entity = repository.save(entity);
157      } catch (Throwable throwable) {
158          logger.error("Failed to save authority rule", throwable);
159          return Result.ofThrowable(-1, throwable);
160      }
161      // if (!publishRules(entity.getApp(), entity.getIp(), entity.getPort())) {
162      //     logger.info("Publish authority rules failed after rule update");
163      // }
164      publishRules(entity.getApp(), entity.getIp(), entity.getPort());
165      return Result.ofSuccess(entity);
166  }
```

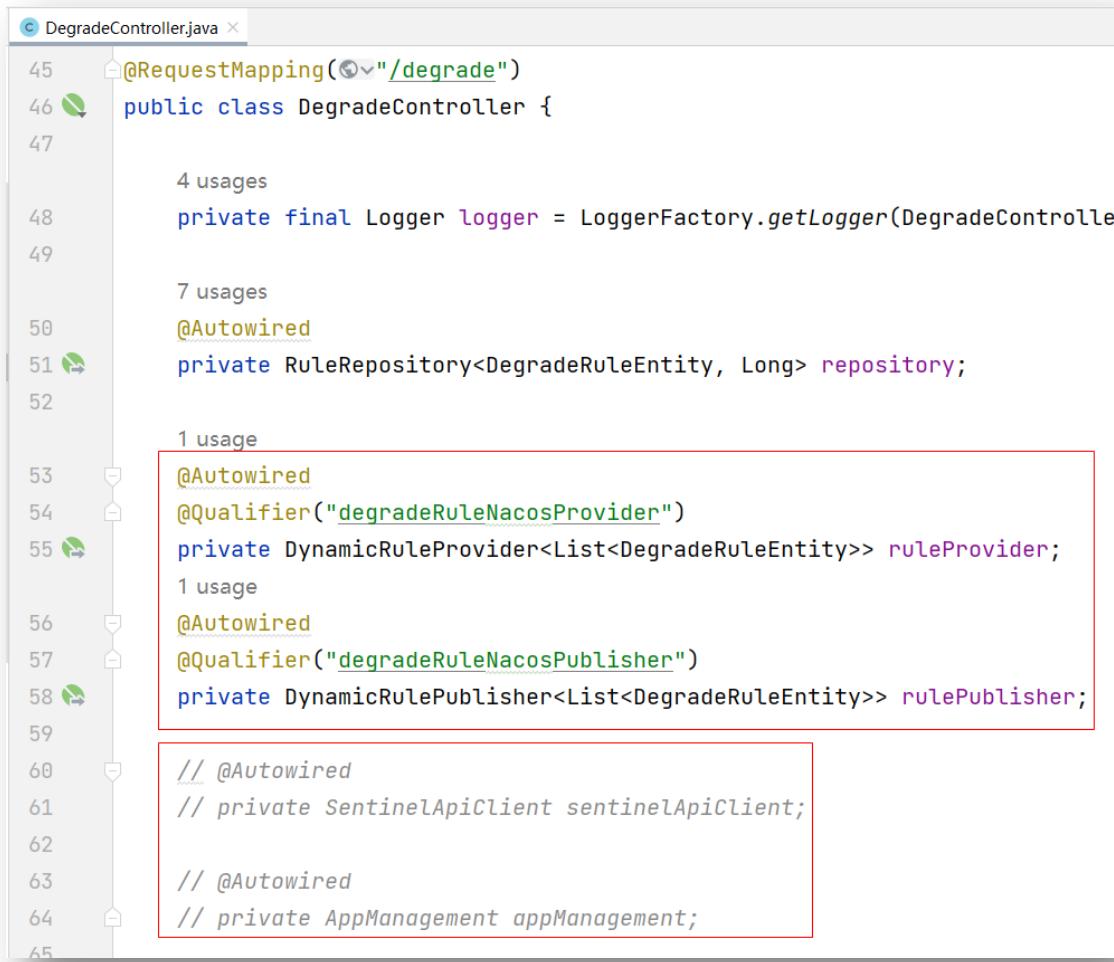
## f、修改 DELETE 方法

```
no usages
175  @DeleteMapping("/rule/{id}")
176  @AuthAction(PrivilegeType.DELETE_RULE)
177  public Result<Long> apiDeleteRule(@PathVariable("id") Long id) throws Exception {
178      if (id == null) {
179          return Result.ofFail(-1, "id cannot be null");
180      }
181      AuthorityRuleEntity oldEntity = repository.findById(id);
182      if (oldEntity == null) {
183          return Result.ofSuccess(null);
184      }
185      try {
186          repository.delete(id);
187      } catch (Exception e) {
188          return Result.ofFail(-1, e.getMessage());
189      }
190      // if (!publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort())) {
191      //     logger.error("Publish authority rules failed after rule delete");
192      // }
193      publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort());
194      return Result.ofSuccess(id);
195  }
```

## C、修改 DegradeRuleController 类

修改 controller 包中的 DegradeRuleController 类。

## a、添加/删除成员变量



```
C DegradeController.java x
45  @RequestMapping("/degrade")
46  public class DegradeController {
47
48      4 usages
49      private final Logger logger = LoggerFactory.getLogger(DegradeController.class);
50
51      7 usages
52      @Autowired
53      private RuleRepository<DegradeRuleEntity, Long> repository;
54
55      1 usage
56      @Autowired
57      @Qualifier("degradeRuleNacosProvider")
58      private DynamicRuleProvider<List<DegradeRuleEntity>> ruleProvider;
59
60      1 usage
61      @Autowired
62      @Qualifier("degradeRuleNacosPublisher")
63      private DynamicRulePublisher<List<DegradeRuleEntity>> rulePublisher;
64
65      // @Autowired
66      // private SentinelApiClient sentinelApiClient;
67
68      // @Autowired
69      // private AppManagement appManagement;
```

## b、修改 GET 方法

```
66     @GetMapping("/rules.json")
67     @AuthAction(PrivilegeType.READ_RULE)
68     public Result<List<DegradeRuleEntity>> apiQueryMachineRules(String app, String
69         if (StringUtil.isEmpty(app)) {
70             return Result.ofFail(-1, "app can't be null or empty");
71         }
72         if (StringUtil.isEmpty(ip)) {
73             return Result.ofFail(-1, "ip can't be null or empty");
74         }
75         if (port == null) {
76             return Result.ofFail(-1, "port can't be null");
77         }
78         // if (!appManagement.isValidMachineOfApp(app, ip)) {
79         //     return Result.ofFail(-1, "given ip does not belong to given app");
80         // }
81     try {
82         // List<DegradeRuleEntity> rules = sentinelApiClient.fetchDegradeRuleO
83         List<DegradeRuleEntity> rules = ruleProvider.getRules(app);
84         rules = repository.saveAll(rules);
85         return Result.ofSuccess(rules);
86     } catch (Throwable throwable) {
87         logger.error("queryApps error:", throwable);
88         return Result.ofThrowable(-1, throwable);
89     }
90 }
```

## c、修改 publishRules()方法

```
3 usages
175     private void publishRules(String app, String ip, Integer port) throws Exception {
176         List<DegradeRuleEntity> rules = repository.findAllByMachine(MachineInfo.of(app,
177             // return sentinelApiClient.setDegradeRuleOfMachine(app, ip, port, rules);
178             rulePublisher.publish(app, rules);
179     }
180 }
```

## d、修改 POST 方法

```
no usages
92  @PostMapping(value="/rule")
93  @AuthAction(PrivilegeType.WRITE_RULE)
94  public Result<DegradeRuleEntity> apiAddRule(@RequestBody DegradeRuleEntity entity) throws Exception {
95      Result<DegradeRuleEntity> checkResult = checkEntityInternal(entity);
96      if (checkResult != null) {
97          return checkResult;
98      }
99      Date date = new Date();
100     entity.setGmtCreate(date);
101     entity.setGmtModified(date);
102     try {
103         entity = repository.save(entity);
104     } catch (Throwable t) {
105         logger.error("Failed to add new degrade rule, app={}, ip={}, entity.getApp(), entity.getIp(), entity.getPort()", entity.getApp(), entity.getIp(), entity.getPort());
106         return Result.ofThrowable(-1, t);
107     }
108     // if (!publishRules(entity.getApp(), entity.getIp(), entity.getPort())) {
109     //     logger.warn("Publish degrade rules failed, app={}", entity.getApp());
110     // }
111     publishRules(entity.getApp(), entity.getIp(), entity.getPort());
112     return Result.ofSuccess(entity);
113 }
```

## e、修改 PUT 方法

```
no usages
115  @PutMapping(value="/rule/{id}")
116  @AuthAction(PrivilegeType.WRITE_RULE)
117  public Result<DegradeRuleEntity> apiUpdateRule(@PathVariable("id") Long id,
118                                                 @RequestBody DegradeRuleEntity entity) throws Exception {
119      if (id == null || id <= 0) {
120          return Result.ofFail(-1, "id can't be null or negative");
121      }
122      DegradeRuleEntity oldEntity = repository.findById(id);
123
124      entity.setGmtModified(new Date());
125      entity.setGmtCreate(oldEntity.getGmtCreate());
126
127      try {
128          repository.save(entity);
129      } catch (Throwable t) {
130          logger.error("Failed to save degrade rule, id={}, rule={}, id, entity, t", id, entity, t);
131          return Result.ofThrowable(-1, t);
132      }
133      // if (!publishRules(entity.getApp(), entity.getIp(), entity.getPort())) {
134      //     logger.warn("Publish degrade rules failed, app={}", entity.getApp());
135      // }
136      publishRules(entity.getApp(), entity.getIp(), entity.getPort());
137      return Result.ofSuccess(entity);
138 }
```

## f、修改 DELETE 方法

```
150     @DeleteMapping("/rule/{id}")
151     @AuthAction(PrivilegeType.DELETE_RULE)
152     public Result<Long> delete(@PathVariable("id") Long id) throws Exception {
153         if (id == null) {
154             return Result.ofFail(-1, "id can't be null");
155         }
156
157         DegradeRuleEntity oldEntity = repository.findById(id);
158         if (oldEntity == null) {
159             return Result.ofSuccess(null);
160         }
161
162         try {
163             repository.delete(id);
164         } catch (Throwable throwable) {
165             logger.error("Failed to delete degrade rule, id={}", id, throwable);
166             return Result.ofThrowable(-1, throwable);
167         }
168         // if (!publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort())) {
169         //     logger.warn("Publish degrade rules failed, app={}", oldEntity.getApp());
170         // }
171         publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort());
172         return Result.ofSuccess(id);
173     }
```

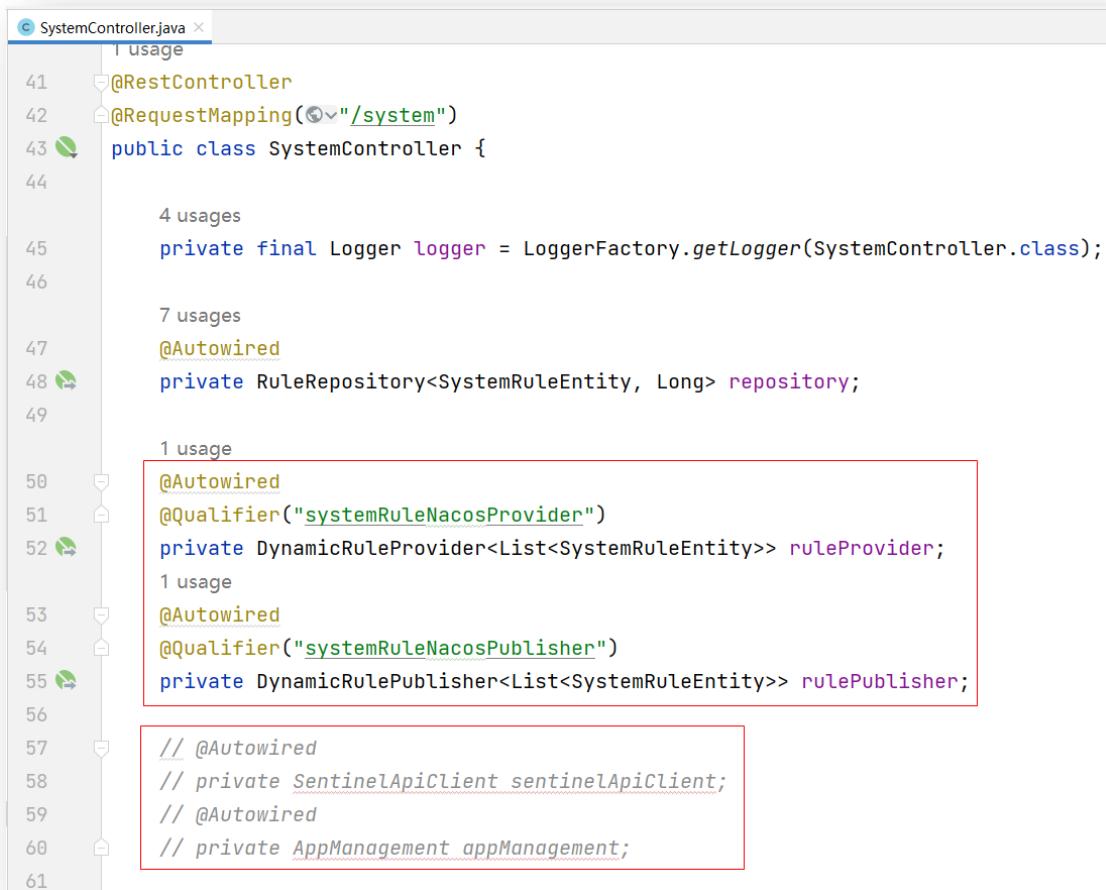
## g、修改 checkEntityInternal()

```
2 usages
181     @
182     private <R> Result<R> checkEntityInternal(DegradeRuleEntity entity) {
183         if (StringUtil.isBlank(entity.getApp())) {
184             return Result.ofFail(-1, "app can't be blank");
185         }
186         if (StringUtil.isBlank(entity.getIp())) {
187             return Result.ofFail(-1, "ip can't be null or empty");
188         }
189         // if (!appManagement.isValidMachineOfApp(entity.getApp(), entity.getIp())) {
190         //     return Result.ofFail(-1, "given ip does not belong to given app");
191         // }
192         if (entity.getPort() == null || entity.getPort() <= 0) {
193             return Result.ofFail(-1, "invalid port: " + entity.getPort());
194         }
195         if (StringUtil.isBlank(entity.getLimitApp())) {
```

## D、修改 SystemRuleController 类

修改 controller 包中的 SystemRuleController 类。

### a、添加/删除成员变量



```
c SystemController.java x
  1 usage
41  @RestController
42  @RequestMapping("/system")
43  public class SystemController {
44
45    4 usages
46    private final Logger logger = LoggerFactory.getLogger(SystemController.class);
47
48    7 usages
49    @Autowired
50    private RuleRepository<SystemRuleEntity, Long> repository;
51
52    1 usage
53    @Autowired
54    @Qualifier("systemRuleNacosProvider")
55    private DynamicRuleProvider<List<SystemRuleEntity>> ruleProvider;
56
57    // @Autowired
58    // private SentinelApiClient sentinelApiClient;
59    // @Autowired
60    // private AppManagement appManagement;
```

## b、修改 checkBasicParams()方法

```
2 usages
62 @ private <R> Result<R> checkBasicParams(String app, String ip, Integer port) {
63     if (StringUtil.isEmpty(app)) {
64         return Result.ofFail(-1, "app can't be null or empty");
65     }
66     if (StringUtil.isEmpty(ip)) {
67         return Result.ofFail(-1, "ip can't be null or empty");
68     }
69     if (port == null) {
70         return Result.ofFail(-1, "port can't be null");
71     }
72     // if (!appManagement.isValidMachineOfApp(app, ip)) {
73     //     return Result.ofFail(-1, "given ip does not belong to given app");
74     // }
75     if (port <= 0 || port > 65535) {
76         return Result.ofFail(-1, "port should be in (0, 65535)");
77     }
78     return null;
79 }
```

## c、修改 query 方法

```
no usages
81 @GetMapping("/rules.json")
82 @AuthAction(PrivilegeType.READ_RULE)
83 public Result<List<SystemRuleEntity>> apiQueryMachineRules(String app, String ip,
84                                                               Integer port) {
85     Result<List<SystemRuleEntity>> checkResult = checkBasicParams(app, ip, port);
86     if (checkResult != null) {
87         return checkResult;
88     }
89     try {
90         // List<SystemRuleEntity> rules = sentinelApiClient.fetchSystemRuleOfMachine();
91         List<SystemRuleEntity> rules = ruleProvider.getRules(app);
92         rules = repository.saveAll(rules);
93         return Result.ofSuccess(rules);
94     } catch (Throwable throwable) {
95         logger.error("Query machine system rules error", throwable);
96         return Result.ofThrowable(-1, throwable);
97     }
98 }
```

## d、修改 publishRules()方法

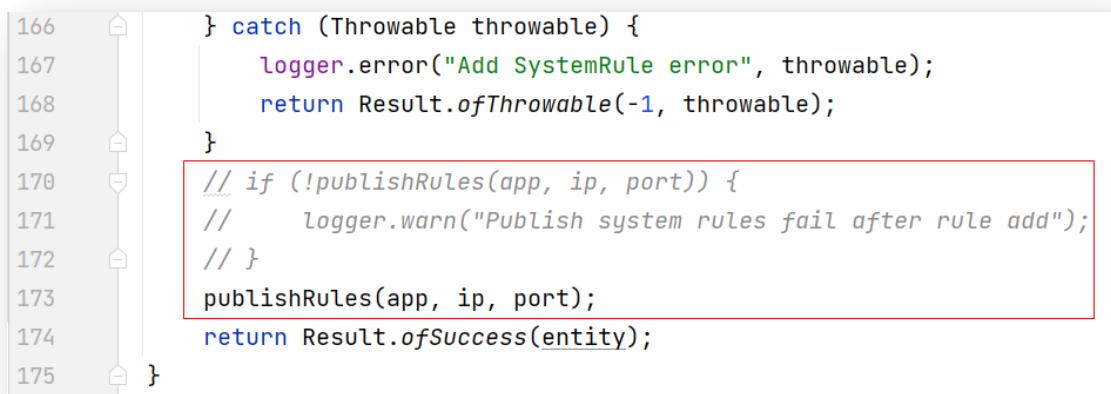


```
263     private void publishRules(String app, String ip, Integer port) throws Exception {
264         List<SystemRuleEntity> rules = repository.findAllByMachine(MachineInfo.of(app,
265             // return sentinelApiClient.setSystemRuleOfMachine(app, ip, port, rules);
266             rulePublisher.publish(app, rules));
267     }
```

## e、修改 add 方法



```
110     @RequestMapping(value="/new.json")
111     @AuthAction(PrivilegeType.WRITE_RULE)
112     public Result<SystemRuleEntity> apiAdd(String app, String ip, Integer port,
113         Double highestSystemLoad, Double highestCpuUsage, Long avgRt,
114         Long maxThread, Double qps) throws Exception {
115
116         Result<SystemRuleEntity> checkResult = checkBasicParams(app, ip, port);
117         if (checkResult != null) {
118             return checkResult;
119         }
120     }
```



```
166     } catch (Throwable throwable) {
167         logger.error("Add SystemRule error", throwable);
168         return Result.ofThrowable(-1, throwable);
169     }
170     // if (!publishRules(app, ip, port)) {
171     //     logger.warn("Publish system rules fail after rule add");
172     // }
173     publishRules(app, ip, port);
174     return Result.ofSuccess(entity);
175 }
```

## f、修改 update 方法

```
no usages
177  @GetMapping("/save.json")
178  @AuthAction(PrivilegeType.WRITE_RULE)
179  public Result<SystemRuleEntity> apiUpdateIfNotNull(Long id, String app, Double highestSystemLoad,
180      Double highestCpuUsage, Long avgRt, Long maxThread, Double qps) throws Exception {
181      if (id == null) {
182          return Result.ofFail(-1, "id can't be null");
183      }
184      SystemRuleEntity entity = repository.findById(id);
185      if (entity == null) {
186
187
188
189
190          logger.error("save error:", throwable);
191          return Result.ofThrowable(-1, throwable);
192      }
193      // if (!publishRules(entity.getApp(), entity.getIp(), entity.getPort())) {
194      //     logger.info("publish system rules fail after rule update");
195      // }
196      publishRules(entity.getApp(), entity.getIp(), entity.getPort());
197      return Result.ofSuccess(entity);
198  }
```

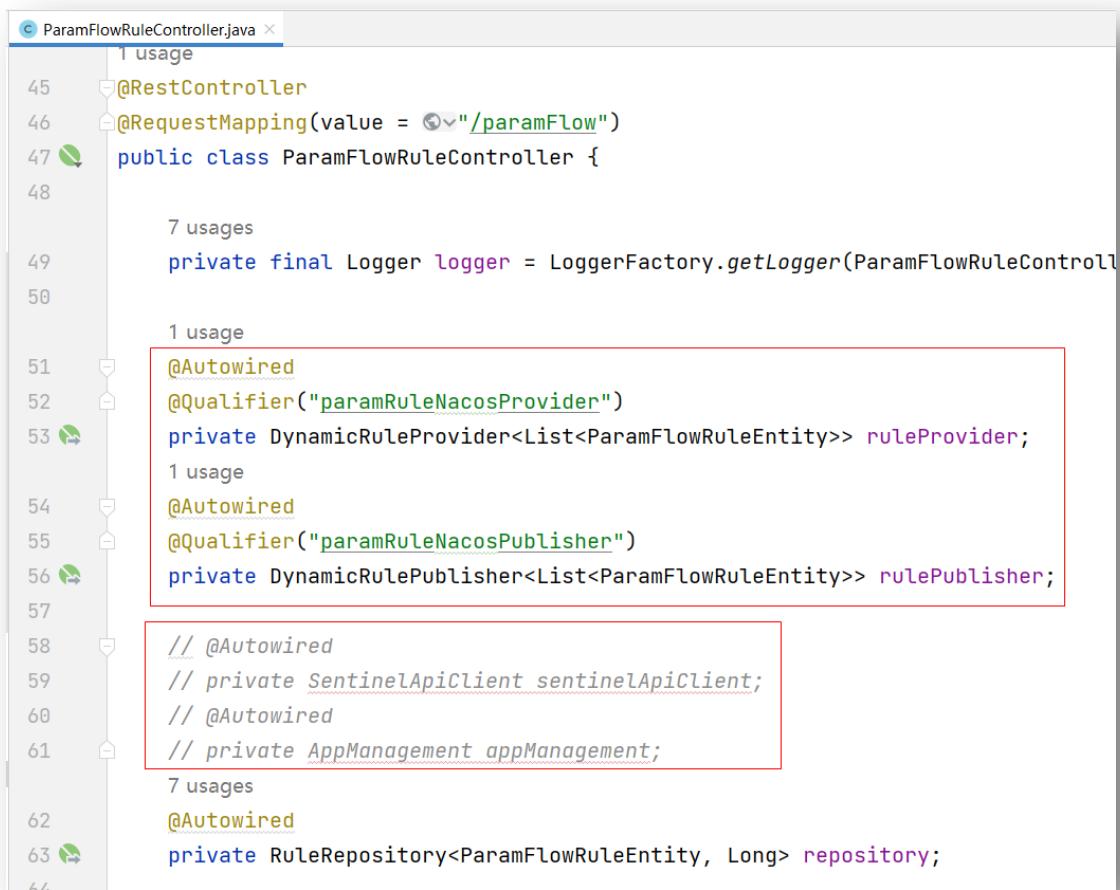
## g、修改 delete 方法

```
240     @RequestMapping(value="/delete.json")
241     @AuthAction(PrivilegeType.DELETE_RULE)
242     public Result<?> delete(Long id) throws Exception {
243         if (id == null) {
244             return Result.ofFail(-1, "id can't be null");
245         }
246         SystemRuleEntity oldEntity = repository.findById(id);
247         if (oldEntity == null) {
248             return Result.ofSuccess(null);
249         }
250         try {
251             repository.delete(id);
252         } catch (Throwable throwable) {
253             logger.error("delete error:", throwable);
254             return Result.ofThrowable(-1, throwable);
255         }
256         // if (!publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort())) {
257         //     logger.info("publish system rules fail after rule delete");
258         // }
259         publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort());
260         return Result.ofSuccess(id);
261     }
```

## E、修改 ParamFlowRuleController 类

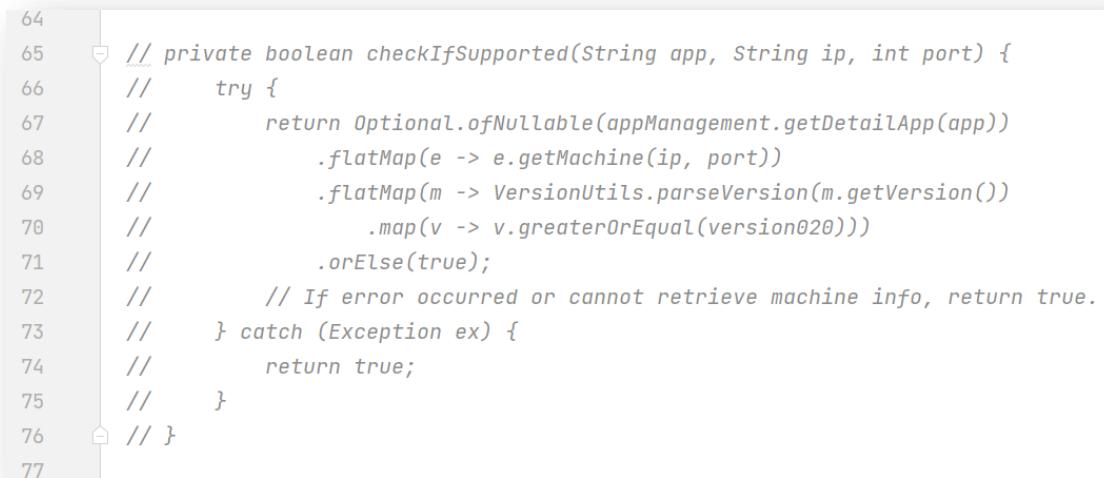
修改 controller 包中的 ParamFlowRuleController 类。

## a、添加/删除成员变量



```
ParamFlowRuleController.java
1 usage
45  @RestController
46  @RequestMapping(value = "/paramFlow")
47  public class ParamFlowRuleController {
48
49      7 usages
50      private final Logger logger = LoggerFactory.getLogger(ParamFlowRuleController.class);
51      1 usage
52      @Autowired
53      @Qualifier("paramRuleNacosProvider")
54      private DynamicRuleProvider<List<ParamFlowRuleEntity>> ruleProvider;
55      1 usage
56      @Autowired
57      @Qualifier("paramRuleNacosPublisher")
58      private DynamicRulePublisher<List<ParamFlowRuleEntity>> rulePublisher;
59
60      // @Autowired
61      // private SentinelApiClient sentinelApiClient;
62      // @Autowired
63      // private AppManagement appManagement;
64
65      7 usages
66      @Autowired
67      private RuleRepository<ParamFlowRuleEntity, Long> repository;
```

## b、删除 checkIfSupported() 方法



```
// private boolean checkIfSupported(String app, String ip, int port) {
//     try {
//         return Optional.ofNullable(appManagement.getDetailApp(app))
//             .flatMap(e -> e.getMachine(ip, port))
//             .flatMap(m -> VersionUtils.parseVersion(m.getVersion()))
//             .map(v -> v.greaterOrEqual(version020))
//             .orElse(true);
//     } catch (Exception ex) {
//         return true;
//     }
// }
```

## c、修改 GET 方法

```
no usages
78     @GetMapping("/rules")
79     @AuthAction(PrivilegeType.READ_RULE)
80     public Result<List<ParamFlowRuleEntity>> apiQueryAllRulesForMachine(@RequestParam String app,
81                                         @RequestParam String ip,
82                                         @RequestParam Integer port) {
83
84         if (StringUtil.isEmpty(app)) {
85             return Result.ofFail(-1, "app cannot be null or empty");
86         }
87         if (StringUtil.isEmpty(ip)) {
88             return Result.ofFail(-1, "ip cannot be null or empty");
89         }
90         if (port == null || port <= 0) {
91             return Result.ofFail(-1, "Invalid parameter: port");
92         }
93
94         // if (!appManagement.isValidMachineOfApp(app, ip)) {
95         //     return Result.ofFail(-1, "given ip does not belong to given app");
96         // }
97     }
```

```
98     try {
99
100        // return sentinelApiClient.fetchParamFlowRulesOfMachine(app, ip, port)
101        //     .thenApply(repository::saveAll)
102        //     .thenApply(Result::ofSuccess)
103        //     .get();
104        List<ParamFlowRuleEntity> rules = ruleProvider.getRules(app);
105        repository.saveAll(rules);
106        return Result.ofSuccess(rules);
107    } catch (ExecutionException ex) {
108        logger.error("Error when querying parameter flow rules", ex.getCause());
109        if (isNotSupported(ex.getCause())) {
110            return unsupportedVersion();
111        } else {
112            return Result.ofThrowable(-1, ex.getCause());
113        }
114    } catch (Throwable throwable) {
115        logger.error("Error when querying parameter flow rules", throwable);
116        return Result.ofFail(-1, throwable.getMessage());
117    }
118 }
```

## d、修改 publishRules()方法

```
262 // private CompletableFuture<Void> publishRules(String app, String ip, Integer port) {  
263 //     List<ParamFlowRuleEntity> rules = repository.findAllByMachine(MachineInfo.of(app, ip, port));  
264 //     return sentinelApiClient.setParamFlowRuleOfMachine(app, ip, port, rules);  
265 // }  
266 // 3 usages  
267 private void publishRules(String app, String ip, Integer port) throws Exception {  
268     List<ParamFlowRuleEntity> rules = repository.findAllByMachine(MachineInfo.of(app, ip, port));  
269     rulePublisher.publish(app, rules);  
270 }  
271
```

## e、修改 POST 方法

```
123 @PostMapping("/{rule}")  
124 @AuthAction(AuthService.PrivilegeType.WRITE_RULE)  
125 public Result<ParamFlowRuleEntity> apiAddParamFlowRule(@RequestBody ParamFlowRuleEntity entity) {  
126     Result<ParamFlowRuleEntity> checkResult = checkEntityInternal(entity);  
127     if (checkResult != null) {  
128         return checkResult;  
129     }  
130     // if (!checkIfSupported(entity.getApp(), entity.getIp(), entity.getPort())) {  
131     //     return unsupportedVersion();  
132     // }  
133     entity.setId(null);  
134     entity.getRule().setResource(entity.getResource().trim());  
135     Date date = new Date();  
136     entity.setGmtCreate(date);  
137     entity.setGmtModified(date);  
138     try {  
139         entity = repository.save(entity);  
140         // publishRules(entity.getApp(), entity.getIp(), entity.getPort()).get();  
141         publishRules(entity.getApp(), entity.getIp(), entity.getPort());  
142         return Result.ofSuccess(entity);  
143     } catch (ExecutionException ex) {  
144         logger.error("Error when adding new parameter flow rules", ex.getCause());  
145         if (isNotSupported(ex.getCause())) {  
146             return unsupportedVersion();  
147         }  
148     }  
149 }
```

## f、修改 PUT 方法

```
202     return Result.ofFail(-1, "id " + id + " does not exist");
203 }
204
205 Result<ParamFlowRuleEntity> checkResult = checkEntityInternal(entity);
206 if (checkResult != null) {
207     return checkResult;
208 }
209 // if (!checkIfSupported(entity.getApp(), entity.getIp(), entity.getPort())) {
210 //     return unsupportedVersion();
211 // }
212 entity.setId(id);
213 Date date = new Date();
214 entity.setGmtCreate(oldEntity.getGmtCreate());
215 entity.setGmtModified(date);
216 try {
217     entity = repository.save(entity);
218     // publishRules(entity.getApp(), entity.getIp(), entity.getPort()).get();
219     publishRules(entity.getApp(), entity.getIp(), entity.getPort());
220     return Result.ofSuccess(entity);
221 } catch (ExecutionException ex) {
222     logger.error("Error when updating parameter flow rules, id=" + id, ex.getCause());
223     if (isNotSupported(ex.getCause())) {
224         return unsupportedVersion();
225     }
226 }
```

## g、修改 DELETE 方法

```
no usages
234 @DeleteMapping("/rule/{id}")
235 @AuthAction(PrivilegeType.DELETE_RULE)
236 public Result<Long> apiDeleteRule(@PathVariable("id") Long id) {
237     if (id == null) {
238         return Result.ofFail(-1, "id cannot be null");
239     }
240     ParamFlowRuleEntity oldEntity = repository.findById(id);
241     if (oldEntity == null) {
242         return Result.ofSuccess(null);
243     }
244
245     try {
246         repository.delete(id);
247         // publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort()).get();
248         publishRules(oldEntity.getApp(), oldEntity.getIp(), oldEntity.getPort());
249         return Result.ofSuccess(id);
250     } catch (ExecutionException ex) {
251         logger.error("Error when deleting parameter flow rules", ex.getCause());
252         if (isNotSupported(ex.getCause())) {
253             return unsupportedVersion();
254         }
255     }
256 }
```

## (8) 修改页面

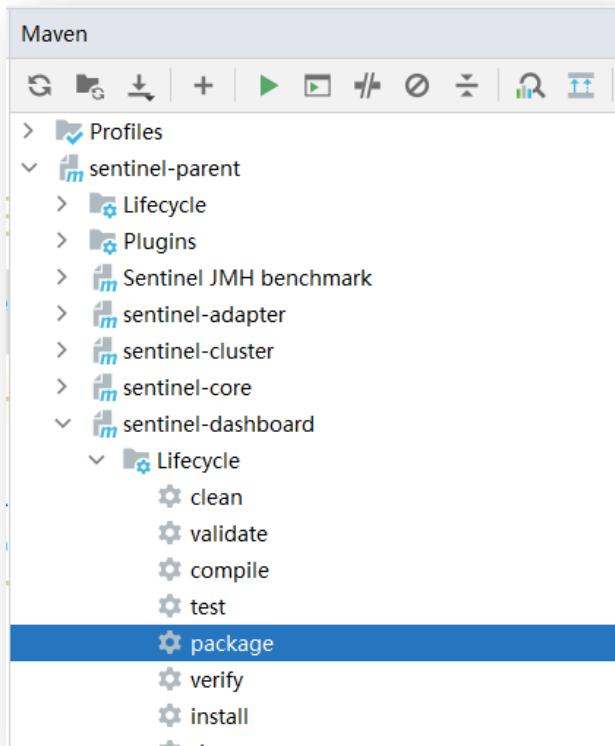
修改 src/main/webapp/resources/app/scripts/directives/sidebar/sidebar.html 中的代码：将 57 行中的 flowV1() 修改为 flow()。

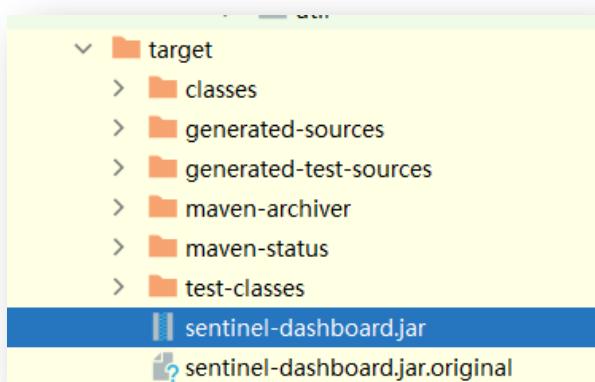


```
55
56      <!--
57      <a ui-sref="dashboard.flowV1({app: entry.app})">-->
58      <a ui-sref="dashboard.flow({app: entry.app})">
59          <i class="glyphicon glyphicon-filter"></i> 流控规则</a>
60    </li>
```

## (9) 重新打包 dashboard

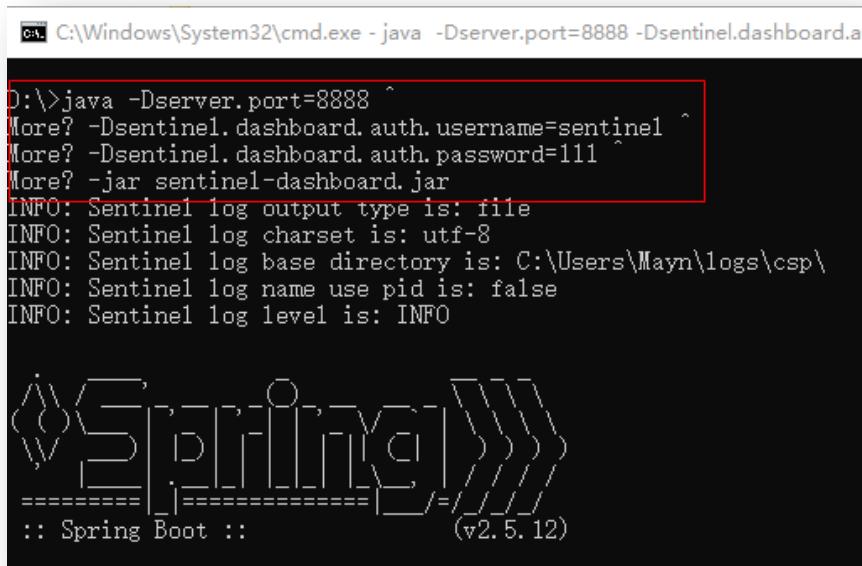
对 Sentinel Dashboard 进行重新打包。





## (10) 启动新的 dashboard

重新打包修改过的 sentinel-dashboard 模块，然后重新启动新的 dashboard 的 jar 包。



```
C:\Windows\System32\cmd.exe - java -Dserver.port=8888 -Dsentry.dashboard.auth.username=sentry1 -Dsentry.dashboard.auth.password=111 -jar sentinel-dashboard.jar
D:\>java -Dserver.port=8888
More? -Dsentry.dashboard.auth.username=sentry1
More? -Dsentry.dashboard.auth.password=111
More? -jar sentinel-dashboard.jar
INFO: Sentinel log output type is: file
INFO: Sentinel log charset is: utf-8
INFO: Sentinel log base directory is: C:\Users\Mayn\logs\csp\
INFO: Sentinel log name use pid is: false
INFO: Sentinel log level is: INFO

Spring Boot :: (v2.5.12)
```

## (11) 运行应用

运行 06-consumer-persist-8080 工程应用。此时就可以实现 Sentinel Dashboard 与 Nacos 配置中心间修改的互通了。

## 6.14 集群流控

### 6.14.1 概述

#### (1) 为什么需要集群流控

有这样一种场景：某微服务由 10 台主机构成的集群提供，现需要控制该微服务的某 API 总 QPS 为 50，如何实现？为每台主机分配单机 QPS 阈值为 5 可以实现吗？不可以。因为对各个主机的访问可能并不均衡，可能有些尚未到达阈值，而有些由于已经超过了阈值而出现了限流。

这种场景下就需要有一个专门的主机，用于统计、分配集群中 QPS，以达到最大限度的降低由于流控而拒绝的请求数量。这就是集群流控。

#### (2) 集群流控构成

集群流控系统由两种角色构成：

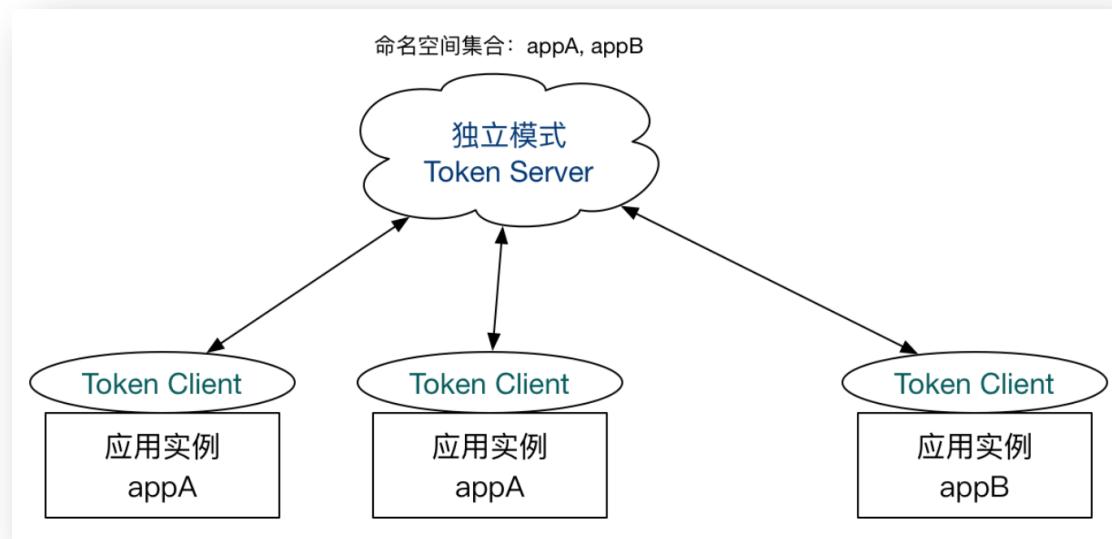
- **Token Client**: 集群流控客户端，用于向所属 Token Server 通信，以请求 token。集群限流服务端会返回给客户端结果，决定是否限流。
- **Token Server**: 集群流控服务端，处理来自 Token Client 的令牌请求，根据配置的集群规则判断是否应该发放 Token（是否允许通过）。

#### (3) Token Server 启动模式

根据 Token Server 启动方式的不同，可以分为两种：

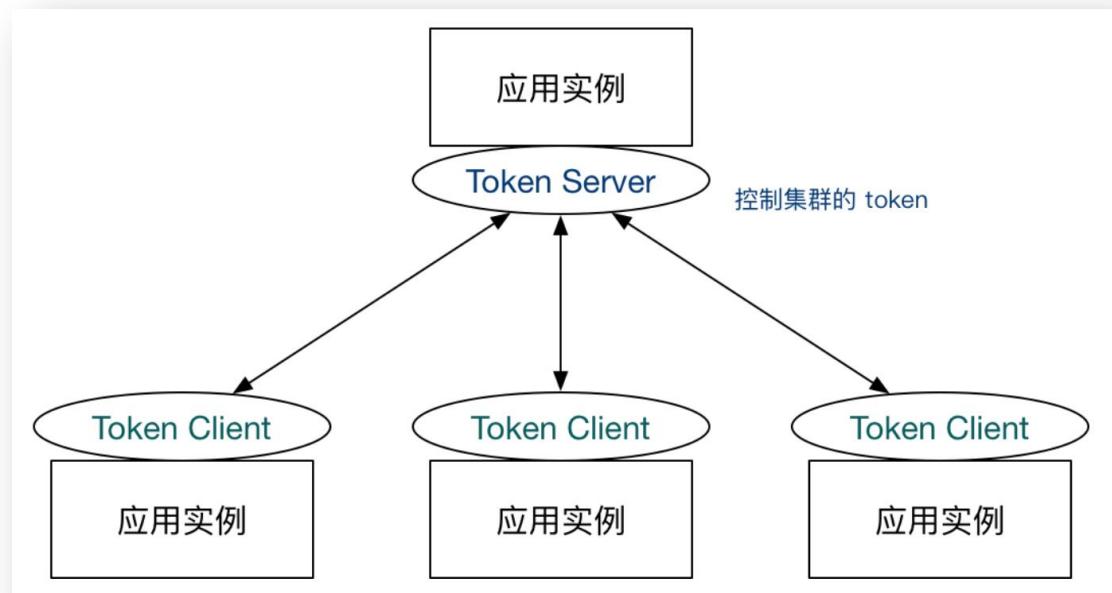
##### A、独立模式

Token Server 作为独立的进程启动，独立部署，隔离性好，但是需要额外的部署操作。



## B、嵌入模式

Token Client 中的某个被指定为 Token Server，该主机同时具备 Server 与 Client 两种身份。该模式下集群中各个主机都是对等的，Token Server 和 Client 可以随时通过提交一个 HTTP 请求进行转变，无需单独部署，灵活性比较好。但隔离性不好，可能会对充当 Server 的 Client 产生影响，且 Token Server 的性能也会由于访问量的增加而受影响。



## 6.14.2 独立模式

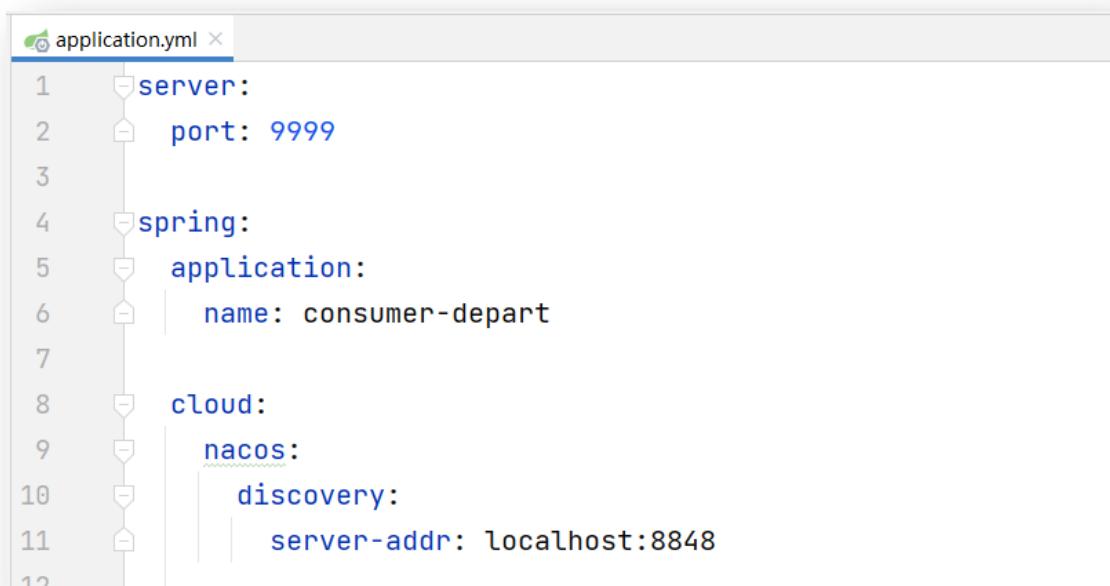
### (1) 定义 Token Server

这里采用独立模式来启动 Token Server。至于嵌入模式，就是将下面的操作直接应用到一个 Token Client 即可。

#### A、定义工程

复制 06-consumer-persist-8080 工程，重命名为 06-cluster-token-server。

#### B、修改配置文件



```
application.yml
1 server:
2   port: 9999
3
4 spring:
5   application:
6     name: consumer-depart
7
8   cloud:
9     nacos:
10    discovery:
11      server-addr: localhost:8848
12
```

```
12
13     sentinel:
14         transport:
15             dashboard: localhost:8888
16             port: 8719
17             eager: true
18
19     datasource:
20         flows:
21             nacos:
22                 server-addr: localhost:8848
23                 rule-type: flow
24                 data-id: ${spring.application.name}-flow-rules
25                 data-type: json
26
```

## C、删除代码

将启动类之外的所有 java 代码全部删除。因为采用的是 Token Server 的独立启动模式，该工程仅是一个 Token Server。

## D、修改启动类代码

```
1 usage
@SpringBootApplication
public class TokenServer9999 {
    no usages
    public static void main(String[] args) throws Exception {
        SpringApplication.run(TokenServer9999.class, args);
        startTokenServer();
    }

    1 usage
    private static void startTokenServer() throws Exception {
        ClusterServerConfigManager.loadGlobalTransportConfig(new ServerTransportConfig()
            .setIdleSeconds(600)
            .setPort(11111));
        String appName = "consumer-depart";
        ClusterServerConfigManager.loadServerNamespaceSet(Collections.singleton(appName));

        ClusterTokenServer tokenServer = new SentinelDefaultTokenServer();
        // Start the server.
        tokenServer.start();
    }
}
```

## (2) 定义 Token Client

### A、定义工程

复制 06-consumer-persist-8080 工程，重命名为 06-cluster-token-client。

## B、修改配置文件



```
# 通过修改该端口号, 启动多个client
server:
  port: 9090

spring:
  application:
    name: abcmsc-depart-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848

sentinel:
  transport:
    dashboard: localhost:8888
  eager: true
  datasource:
    my-flow-rule:
      nacos:
        server-add: localhost:8848
        data-id: consumer_rule
        rule-type: flow
        data-type: json
```

## C、修改启动类

```
@SpringBootApplication
public class TokenClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(TokenClientApplication.class, args);
        // 加载Token Client配置
        LoadClusterClientConfig();
    }
}
```

```
private static void loadClusterClientConfig(){
    // 指定当前应用为Token Client
    ClusterStateManager.applyState(ClusterStateManager.CLUSTER_CLIENT);

    // 为Token Client分配Token Server, assign, 分配
    ClusterClientAssignConfig assignConfig = new ClusterClientAssignConfig();
    assignConfig.setServerHost("localhost");
    assignConfig.setServerPort(9999);
    ClusterClientConfigManager.applyNewAssignConfig(assignConfig);

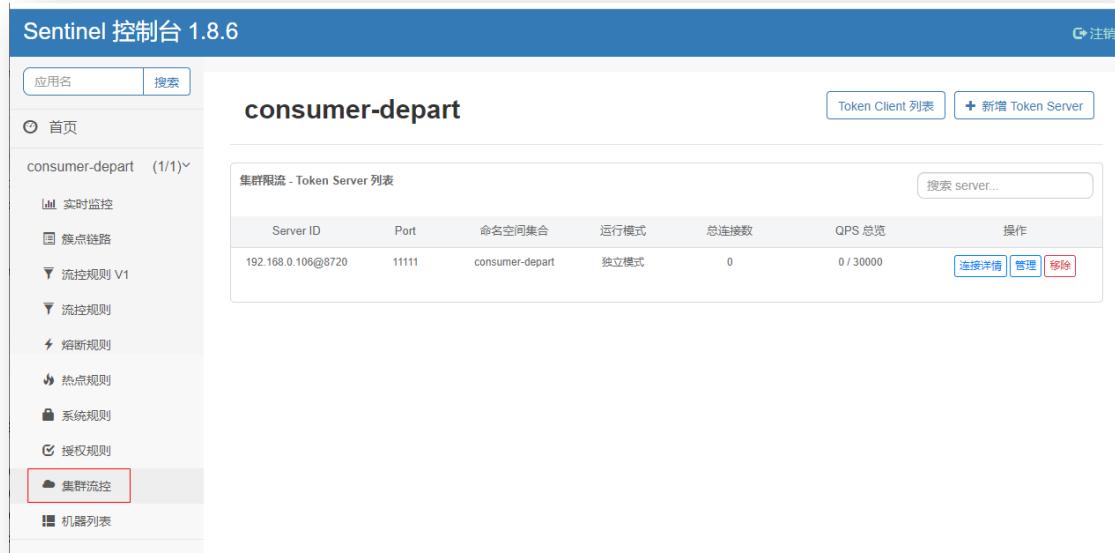
    ClusterClientConfig clientConfig = new ClusterClientConfig();
    // Token Client请求Token Server的超时时限
    clientConfig.setRequestTimeout(200);
    ClusterClientConfigManager.applyNewConfig(clientConfig);
}

}
```

### (3) 启动 Token Server

首先要启动 Nacos 与 Sentinel Dashboard。注意，为了方便与 Nacos 间的数据同步，这里启动前面修改过的 Sentinel Dashboard。然后再启动 Token Server。

Token Server 启动后，在 Sentinel 控制台的“集群流控”的 Token Server 列表中即可立即看到该 Server。只不过，该 Server 目录的连接数为 0。



The screenshot shows the Sentinel Control Console interface. On the left, there's a sidebar with navigation items like '首页', 'consumer-depart (1/1)', and '集群流控' (highlighted with a red box). The main area is titled 'consumer-depart' and shows a table for '集群限流 - Token Server 列表'. The table has columns: Server ID, Port, 命名空间集合 (Namespace Collection), 运行模式 (Run Mode), 总连接数 (Total Connections), QPS 总览 (QPS Overview), and 操作 (Operations). One entry is listed: 192.168.0.106@8720, Port 11111, Namespace consumer-depart, Run Mode 独立模式 (Independent Mode), Total Connections 0, QPS 0 / 30000. Operations buttons include '连接详情' (Connection Details), '管理' (Manage), and '移除' (Remove).

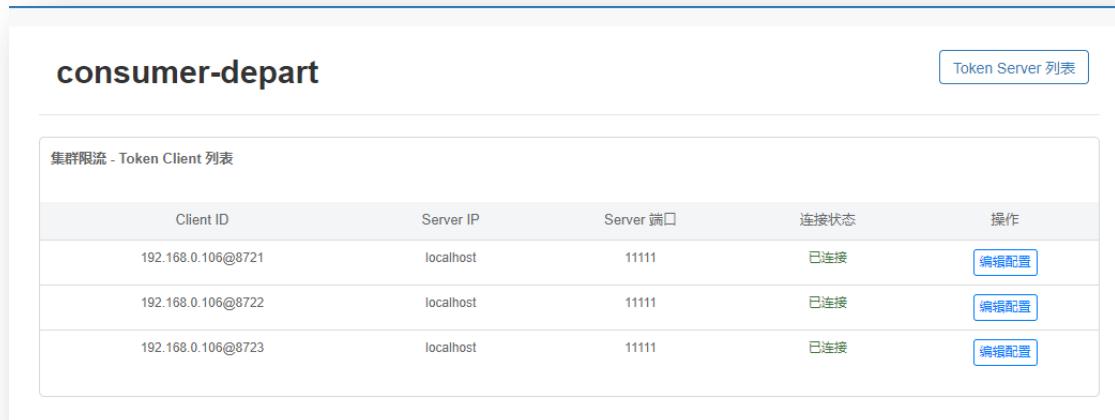
## (4) 启动 Token Client

启动三个 Token Client，修改端口号分别为 8080、7070、6060。再启动一个提供者工程 02-provider-nacos-8081。

## (5) 查看 Sentinel 控制台

### A、查看 Token Client 列表

查看 Sentinel 控制台中的“集群流控”，点击“Token Client 列表”，可以查看到当前所有的 Client 信息。并显示，它们已经连接到了 localhost:11111 的 Server。



The screenshot shows the Sentinel Control Console interface. The main area is titled 'consumer-depart' and shows a table for '集群限流 - Token Client 列表'. The table has columns: Client ID, Server IP, Server 端口 (Port), 连接状态 (Connection Status), and 操作 (Operations). Three entries are listed: 192.168.0.106@8721, Server IP localhost, Port 11111, 已连接 (Connected), 操作 button '编辑配置'; 192.168.0.106@8722, Server IP localhost, Port 11111, 已连接 (Connected), 操作 button '编辑配置'; 192.168.0.106@8723, Server IP localhost, Port 11111, 已连接 (Connected), 操作 button '编辑配置'.

## B、查看 Token Server 列表

点击“Token Server 列表”，在 Token Server 列表中可以看到新增了一个 ServerID 为“localhost:11111（自主指定）”的 Token Server。该 Token Server 是一个临时 Server，是由三个 Token Client 在启动时“自主指定”的，即三个 consumer 在应用中指定的要连接的那个 Token Server。该 Token Server 除了 port 外，其它都是“未知”。



Server ID	Port	命名空间集合	运行模式	总连接数	QPS 总览	操作
192.168.0.106@8720	11111	consumer-depart	独立模式	3	0 / 30000	<button>连接详情</button> <button>管理</button> <button>移除</button>
localhost:11111 (自主指定)	11111	未知	未知	未知	未知	<button>管理</button> <button>移除</button>

虽然在 ID 为 192.168.0.106@8720 的 Server 中可以看到“总连接数”为 3，但点击“管理”按钮却发现其没有一个 client。原因是，启动的 3 个 client 连接在了临时 Server 上。



Token Server 分配编辑

Token Server: 192.168.0.106@8720      Server 端口: 11111

最大允许 QPS: 30000

请从中选取 client:

已选取的 client 列表

保存 取消

点击临时 Server 的“管理”可以看到该 Token Server 中包含了三个 client。



由于该 Token Server 是一个临时 Server，没有用，所以，在该窗口中选中这三个 client，将它们从中“移出(左箭头按钮)”。



点击“保存”后自动返回 Token Server 列表，发现该临时 Token Server 已经自动消失了。



Server ID	Port	命名空间集合	运行模式	总连接数	QPS 总览	操作
192.168.0.106@8720	11111	consumer-depart	独立模式	0	0 / 30000	<a href="#">连接详情</a> <a href="#">管理</a> <a href="#">删除</a>

### C、为 Token Server 分配 Client

点击 Server 的“管理”按钮，可以看到在 client 选取区中已经具有了 client。将它们移动到“已选取区”。



此时 Server 中的总连接数发生了变化。

**consumer-depart**

Token Client 列表 + 新增 Token Server

集群限流 - Token Server 列表						
Server ID	Port	命名空间集合	运行模式	总连接数	QPS 总览	操作
192.168.0.106@8720	11111	consumer-depart	独立模式	3	0 / 30000	<span>连接详情</span> <span>管理</span> <span>移除</span>

## D、新建流控规则

**Sentinel 控制台 1.8.6**

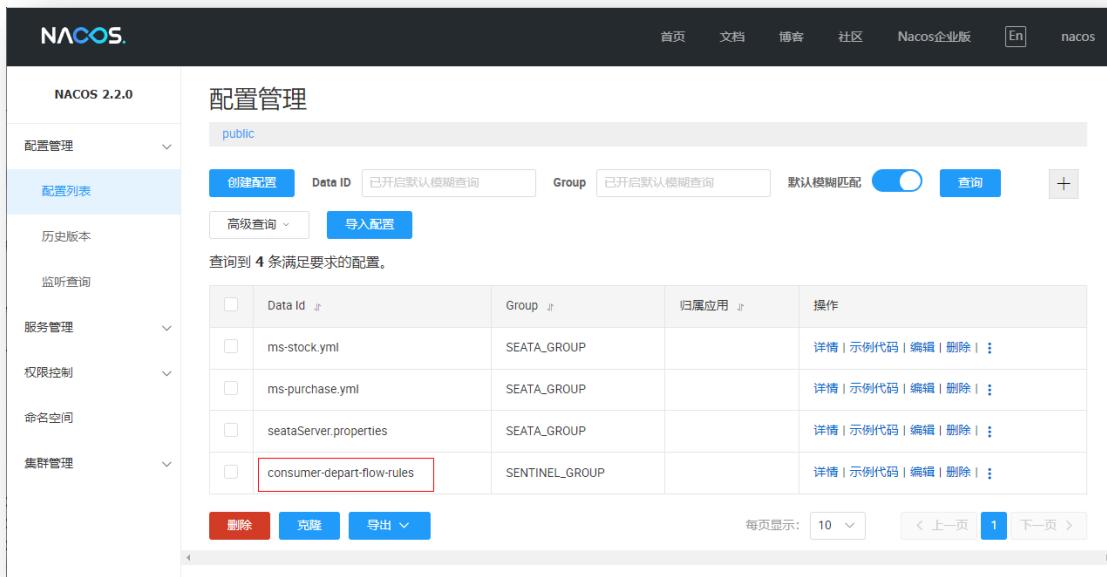
回到单机页面 + 新增流控规则

consumer-depart	
关键字	操作
10 条记录, 每页 10 条记录	<span>重置</span>

**新增流控规则**

资源名: getHandle  
 针对来源: default  
 阀值类型:  QPS  并发线程数  
 集群阀值: 3  
 是否集群:   
 集群阀值模式:  单机均摊  总体阀值  
 失败退化:  如果 Token Server 不可用是否退化到单机限流  
新增 取消

此时打开 nacos 查看其配置列表，发现自动增加了一个配置文件。



The screenshot shows the Nacos 2.2.0 configuration management interface. The left sidebar has sections like 'NACOS 2.2.0', '配置管理' (Configuration Management) (selected), '配置列表' (Configuration List) (highlighted in blue), '历史版本' (History Versions), '监听查询' (Listen Query), '服务管理' (Service Management), '权限控制' (Permission Control), '命名空间' (Namespace), and '集群管理' (Cluster Management). The main area is titled '配置管理' (Configuration Management) and shows a search bar with 'public'. Below it is a table with columns: Data ID, Group, 归属应用 (Belonging Application), and 操作 (Operations). The table contains four rows:

Data ID	Group	归属应用	操作
ms-stock.yml	SEATA_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>
ms-purchase.yml	SEATA_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>
seataServer.properties	SEATA_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>
consumer-depart-flow-rules	SENTINEL_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>

At the bottom of the table are buttons for '删除' (Delete), '克隆' (Clone), and '导出' (Export). To the right, there's a page navigation section with '每页显示: 10' (Items per page: 10), a current page indicator '1', and '上一页' (Previous page) and '下一页' (Next page) buttons.

## E、为 Token Server 分配 Client

此时发现一个奇怪的情况：真正 Token Server 的连接数变为了 0（原来是 3）。再打开“连接详情”，发现没有了连接的 Client。



The screenshot shows the '连接详情' (Connection Details) interface for a Token Server at 192.168.3.8@8719. The main table lists connected clients by namespace and connection count. One row for 'abcmsc-depart-consumer' is highlighted with a red box. The table has columns: 命名空间 (Namespace), 连接数 (Connection Count), and 连接详情 (Connection Details). The toolbar on the right includes a search bar '搜索 server...', and buttons for '操作' (Operations), '连接详情' (Connection Details) (highlighted with a red box), '管理' (Management), and '移除' (Remove).

命名空间	连接数	连接详情
default	0	□
abcmsc-depart-consumer	0	□

这是为什么？打开“管理”就发现了奥妙。



原来，这三个 Client 已经出现在了真正的 Token Server 中，但还没有被选取。



将这三个 Client 选取后保存，即可看到正常的如下所示的连接数。



Server ID	Port	命名空间集合	运行模式	总连接数	QPS 总览	操作
192.168.3.8@8719	9999	abcmsc-depart-consumer	独立模式	3	0 / 30000	<a href="#">连接详情</a>

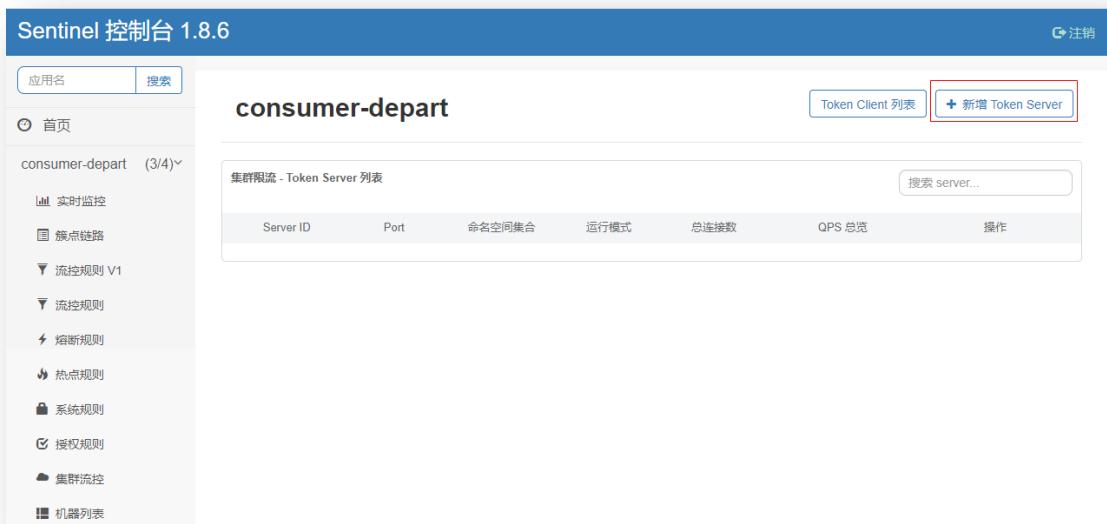
### 6.14.3 嵌入模式

在独立模式基础上可直接建立嵌入模式的集群。无需修改任何代码，只需修改 Sentinel 控制台的配置即可。

#### (1) 停掉 Token Server

将原来独立模式时的 Token Server 停掉。

#### (2) 新增 Token Server



The screenshot shows the Sentinel Control Panel interface. On the left, there's a sidebar with navigation items like '首页', '实时监控', '链路追踪', '流控规则 V1', '流控规则', '熔断规则', '热点规则', '系统规则', '授权规则', '集群流控', and '机器列表'. The main area is titled 'consumer-depart' and shows a table for '集群限流 - Token Server 列表'. The table has columns for 'Server ID', 'Port', '命名空间集合', '运行模式', '总连接数', 'QPS 总览', and '操作'. At the top right of the main area, there are buttons for 'Token Client 列表' and '+ 新增 Token Server', with the latter being highlighted by a red box.

选择“应用内机器”，从“选择机器”中任意选择一个 Token Client 同时作为 Token Server。



The screenshot shows the '新增 Token Server' dialog box. It has fields for '机器类型' (with '应用内机器' selected and highlighted by a red box), '选择机器' (a dropdown menu containing '192.168.0.106@8720', '192.168.0.106@8721', and '192.168.0.106@8722'), 'Server 端口' (set to '18730'), and buttons for '保存' and '取消' at the bottom.

再将剩余的两个 client 移动到“已选取区”。



此时的嵌入模式的集群已经搭建完成。

consumer-depart

Token Client 列表 + 新增 Token Server

集群限流 - Token Server 列表 搜索 server...

Server ID	Port	命名空间集合	运行模式	总连接数	QPS 总览	操作
192.168.0.106@8720	18730	consumer-depart	嵌入模式	3	0 / 20000	<span>连接详情</span> <span>管理</span> <span>移除</span>

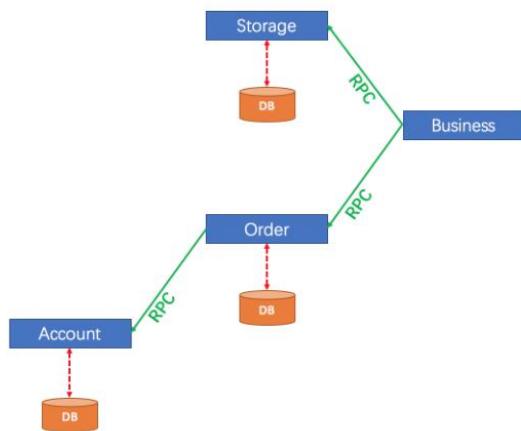
### (3) 修改流控规则

仍可保持原来的流控规则，当然，也可修改。

## 第7章 分布式事务解决方案 Seata

### 7.1 Seata 概述

#### 7.1.1 分布式事务简介



对于分布式事务，通俗地说就是，一次操作由若干分支操作组成，这些分支操作分属不同应用，分布在不同服务器上。分布式事务需要保证这些分支操作要么全部成功，要么全部失败。

能够实现分布式事务的解决方案很多，但阿里 Seata 则是一个非常成熟的分布式事务产品，Spring Cloud Alibaba 推荐使用 Seata 作为分布式事务解决方案。

#### 7.1.2 Seata 简介

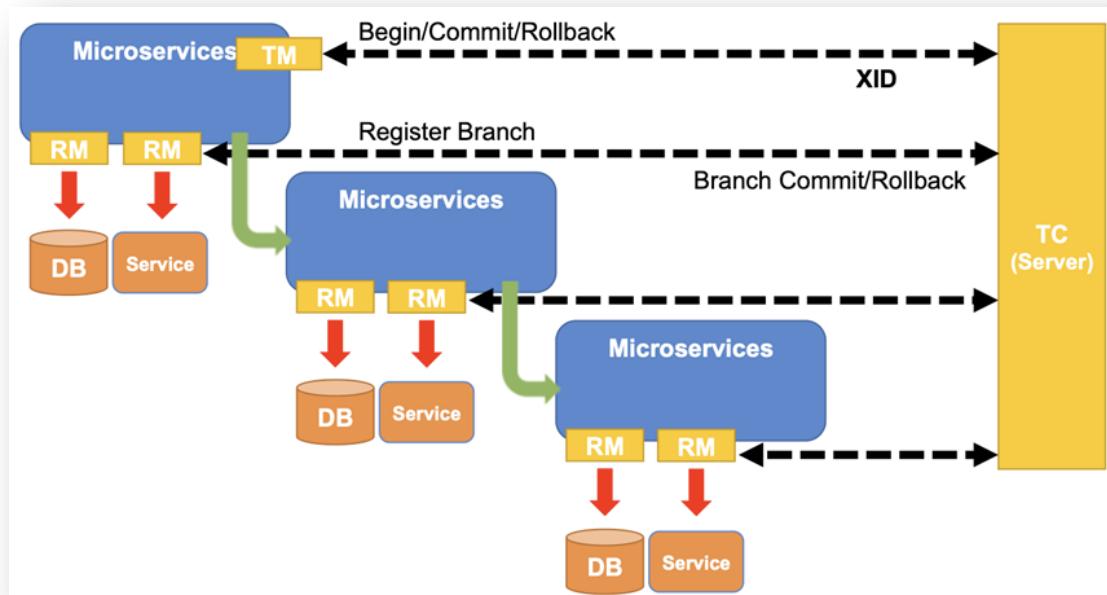
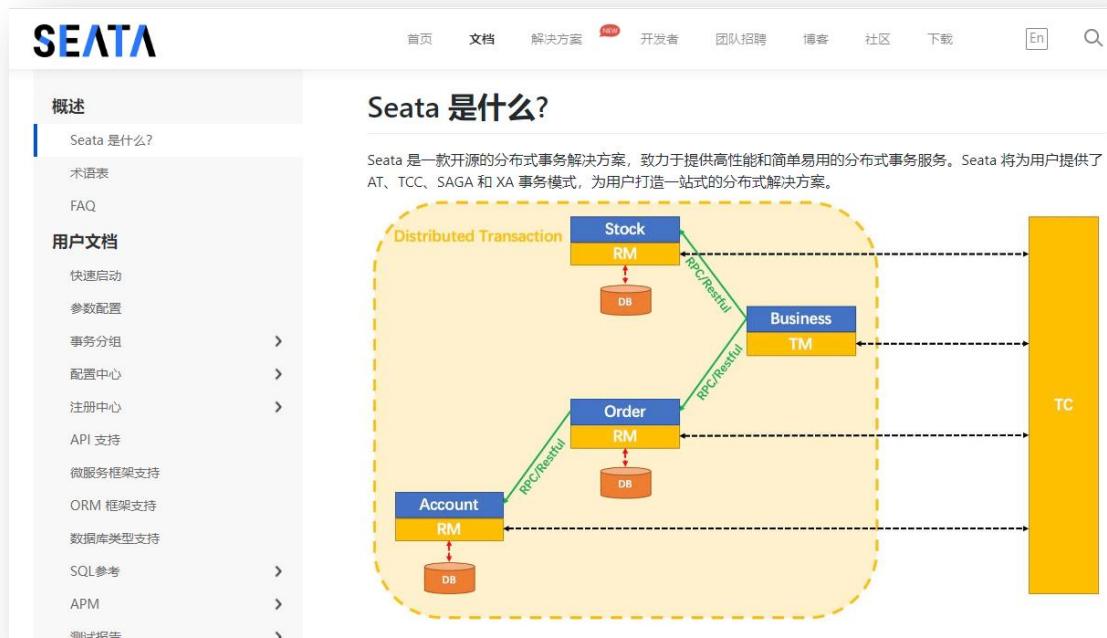
以下是来自于 Seata 官网的介绍：Seata 官网：<http://seata.io/zh-cn/>

Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。

在 Seata 开源之前，Seata 对应的内部版本在阿里经济体内部一直扮演着分布式一致性中间件的角色，帮助经济体平稳地度过历年的双 11，对各 BU 业务进行了有力的支撑。经过多年沉淀与积累，商业化产品先后在阿里云、金融云进行售卖。2019 年 1 月为了打造更加完善的技术生态和普惠技术成果，Seata 正式宣布对外开源，未来 Seata 将以社区共建的形式帮助其技术更加可靠与完备。

### 7.1.3 Seata 架构

#### (1) 官方架构图





Seata 中有三个常用术语：TC、TM 与 RM。

### (2) TC

Transaction Coordinator，事务协调者。维护全局和分支事务的状态，驱动全局事务提交或回滚。

### (3) TM

Transaction Manager，事务管理器。定义全局事务的范围：开始全局事务、提交或回滚全局事务。

### (4) RM

Resource Manager，资源管理器。管理分支事务处理的资源，与 TC 交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

## 7.2 分布式事务模式

Seata 提供了 XA、AT、TCC 与 SAGA 四种分布式事务模式。

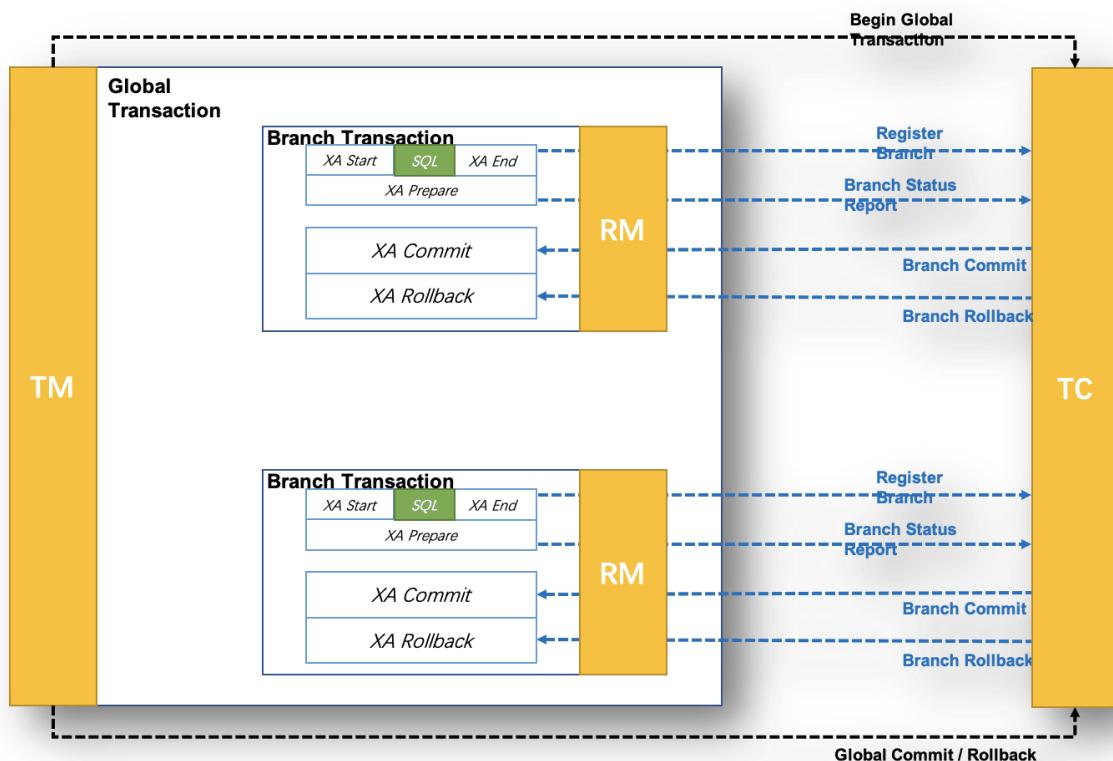
### 7.2.1 XA 模式

#### (1) 使用前提

XA 模式使用的前提条件有：

- 基于 XA 事务的数据库。
- Java 应用，通过 JDBC 访问数据库。

#### (2) 模式架构



XA (UniX TransAction) 是一种分布式事务解决方案，一种分布式事务处理模式，是基于 XA 协议的。XA 协议由 Tuxedo (Transaction for Unix has been Extended for Distributed Operation, 分布式操作扩展之后的 Unix 事务系统) 首先提出的，并交给 X/Open 组织，作为资源管理器与事务管理器的接口标准。

#### (3) 存在的问题

XA 模式存在两个问题：

- 回滚日志无法自动清理，需要手工清理。

- 
- 多线程下对同一个 RM 中的数据进行修改，存在 ABA 问题。

## 7.2.2 AT 模式

### (1) 使用前提

AT 模式使用的前提条件为：

- 基于支持本地 ACID 事务的关系型数据库。
- Java 应用，通过 JDBC 访问数据库。

### (2) 模式架构

AT，Automatic Transaction。AT 模式是 Seata 默认的分布式事务模型，是由 XA 模式演变而来的，通过全局锁对 XA 模式中的问题（ABA 问题）进行了改进，并实现了回滚日志的自动清理。

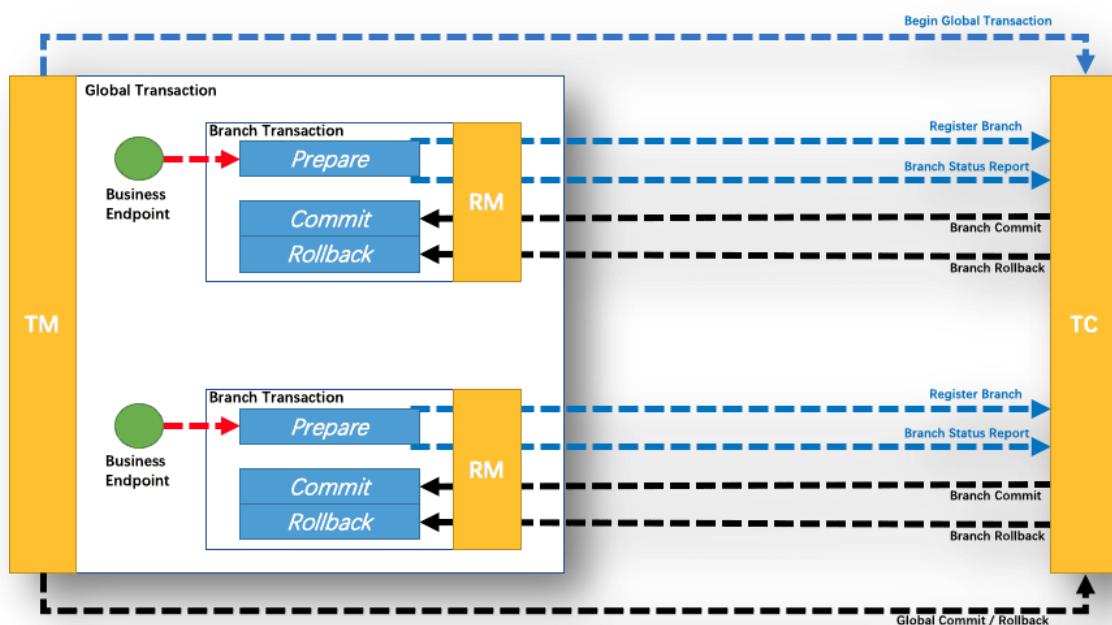
### (3) 存在的问题

AT 模式也存在两个问题：

- 不支持 NoSQL。
- 全局 commit/rollback 阶段及回滚日志的清除过程，完全“自动化”无法实现定制化过程。

### 7.2.3 TCC 模式

#### (1) 模式架构



TCC, Try Confirm/Cancel, 同样也是 2PC 的, 其与 AT 的重要区别是, 支持将自定义的分支事务纳入到全局事务管理中, 即可以实现定制化的日志清理与回滚过程。当然, 该模式对业务逻辑的侵入性是较大的。

#### (2) 官方解释

什么意思呢? 查看官方给出的解释:

根据两阶段行为模式的不同，我们将分支事务划分为 Automatic (Branch) Transaction Mode 和 TCC (Branch) Transaction Mode.

AT 模式（[参考链接 TBD](#)）基于 支持本地 ACID 事务 的 关系型数据库：

- 一阶段 prepare 行为：在本地事务中，一并提交业务数据更新和相应回滚日志记录。
- 二阶段 commit 行为：马上成功结束，**自动** 异步批量清理回滚日志。
- 二阶段 rollback 行为：通过回滚日志，**自动** 生成补偿操作，完成数据回滚。

相应的，TCC 模式，不依赖于底层数据资源的事务支持：

- 一阶段 prepare 行为：调用 **自定义** 的 prepare 逻辑。
- 二阶段 commit 行为：调用 **自定义** 的 commit 逻辑。
- 二阶段 rollback 行为：调用 **自定义** 的 rollback 逻辑。

所谓 TCC 模式，是指支持把 **自定义** 的分支事务纳入到全局事务的管理中。

第一段（红框上的内容）的意思是，对于 2PC 模式，分支事务分为 AT 模式与 TCC 模式。

下面的两部分则是给出了 AT 模式与 TCC 模式的区别：我们可以看出，它们的区别就体现在“自动”“自定义”上：TCC 的 prepare、commit、rollback 逻辑都是自定义的，而 AT 中的都是自动的，执行系统定义好的逻辑。

#### 7.2.4 2PC 先天缺陷

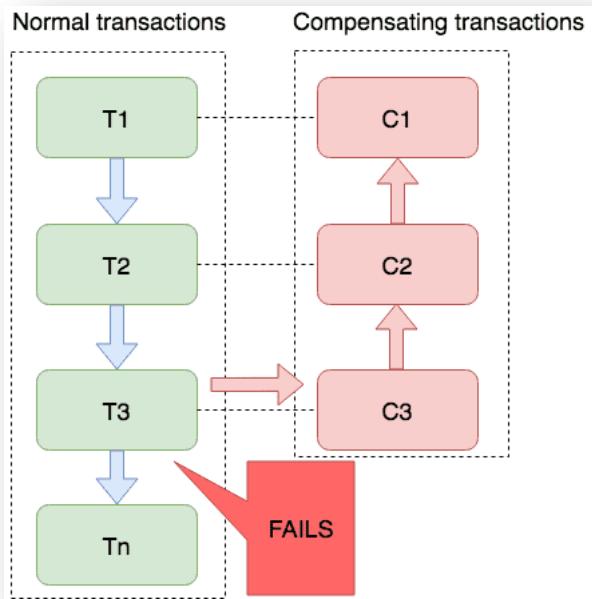
2PC 最大的特点是简单（原理简单、实现简单），但也存在着不少的先天缺陷。例如，同步阻塞、中心化问题、数据不一致、太过保守……。

#### 7.2.5 Saga 模式

对于架构复杂，且业务流程较多较长的系统，一般不适合使用 2PC 的分布式事务模式。因为这种系统一般无法提供 TM、TC、RM 三种接口。此时，我们可以尝试着选择 Saga 模式。

##### （1）模式架构

Saga 模式是 1987 年由普林斯顿大学的 Hector Garcia-Molina 和 Kenneth Salem 共同提出的。该模式不同于前面的模式，它不是 2PC 的，其是一种长事务解决方案。



其应用场景是：在无法提供 TC、TM、RM 接口的情况下，对于一个流程很长的复杂业务，其会包含很多的子流程（事务）。每个子流程都让它们真实提交它们真正的执行结果。只有当前子流程执行成功后才能执行下一个子流程。若整个流程中所有子流程全部执行成功，则整个业务流程成功；只要有一个子流程执行失败，则可采用两种补偿方式：

- 向后恢复：对于其前面所有成功的子流程，其执行结果全部撤销
- 向前恢复：重试失败的子流程直到其成功。当然这个前提是确保每个分支事务都能够成功。

如何保证不发生脏读呢？当子流程在提交了执行结果后，可根据业务场景为业务逻辑加锁或为资源加锁。

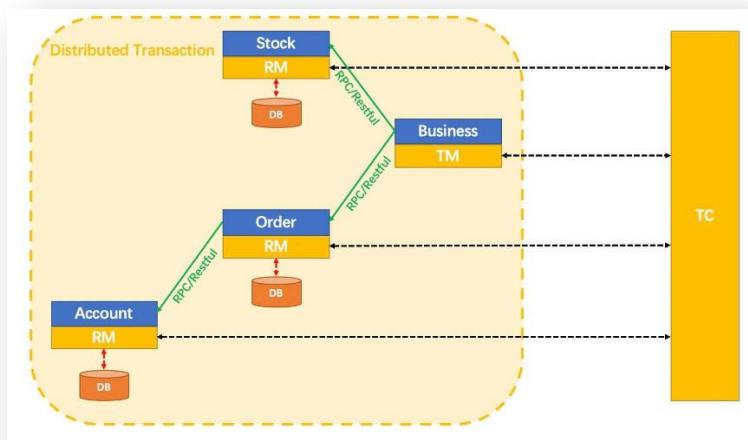
## (2) 与 2PC 模式区别

- Saga 模式的所有分支事务是串行执行的，而 2PC 的则是并行执行的。
- Saga 模式没有 TC，其是通过子流程间的消息传递来完成全局事务管理的，而 2PC 则具有 TC，其是通过 TC 完成全局事务管理的。

## 7.3 测试环境搭建

对于 Seata 的用法，通过官方给出的场景来举例说明。

### 7.3.1 需求



按照官方架构示例先搭建一下后续要进行 Seata 用法测试的代码环境。

这里要定义 Stock(id, goodsId, total)、Order(id, userId, goodsId, goodsPrice, count)、Account(id, userId, balance)与 Business 等至少四个工程。Business 消费 Stock 与 Order 服务，Order 同时也消费 Account 服务。另外，Stock、Order 与 Account 分别连接着 mysql1、mysql2 与 mysql3 三个 DBMS，且均采用 JPA 作为 ORM。

### 7.3.2 Stock 工程 07-ec-stock-8081

#### (1) 定义工程

复制 03-provider-nacos-config-8081 工程，并重命名为 07-ec-stock-8081。将原来工程中除了启动类外所有类删除，保留原有的 POM 与配置文件。

## (2) 定义实体类 Stock

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler", "fieldHandler"})
public class Stock {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    no usages
    private Integer goodsId; // 商品id
    no usages
    private int total; // 库存量
}
```

## (3) 定义 StockRepository 接口

```
2 usages
public interface StockRepository extends JpaRepository<Stock, Integer> {
    1 usage
    Stock findStockByGoodsId(Integer goodsId);
}
```

#### (4) 定义 StockService 接口

```
1 implementation
public interface StockService {
    1 usage 1 implementation
    Stock deductStock(Integer goodsId, Integer count);
}
```

#### (5) 定义 StockServiceImpl 类

```
no usages
@Service
public class StockServiceImpl implements StockService {

    2 usages
    @Autowired
    private StockRepository repository;

    1 usage
    @Override
    public Stock deductStock(Integer goodsId, Integer count) {
        Stock stock = repository.findStockByGoodsId(goodsId);

        if (stock == null || stock.getTotal() < count) {
            return null; // 返回值为null, 表示扣减库存失败
        }
        stock.setTotal(stock.getTotal() - count);
        Stock s = repository.save(stock);
        return s;
    }
}
```

## (6) 定义 StockController 类

```
@RestController
public class StockController {
    1 usage
    @Autowired
    private StockService service;

    no usages
    @PutMapping("stock/deduct")
    public Stock deductHandle(@RequestParam("goodsId") Integer goodsId,
                               @RequestParam("count") Integer count) {
        // 扣减库存
        return service.deductStock(goodsId, count);
    }
}
```

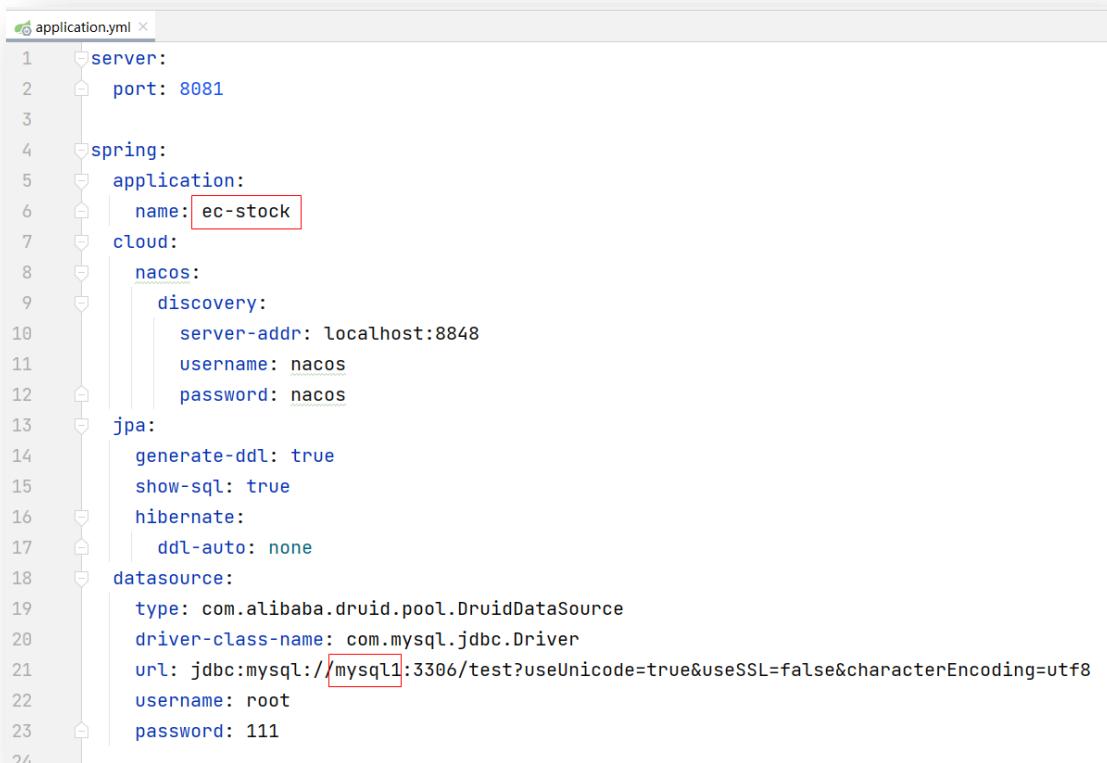
## (7) 定义启动类

```
1 usage
@SpringBootApplication
public class Stock8081 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Stock8081.class, args);
    }
}
```

## (8) 修改配置文件

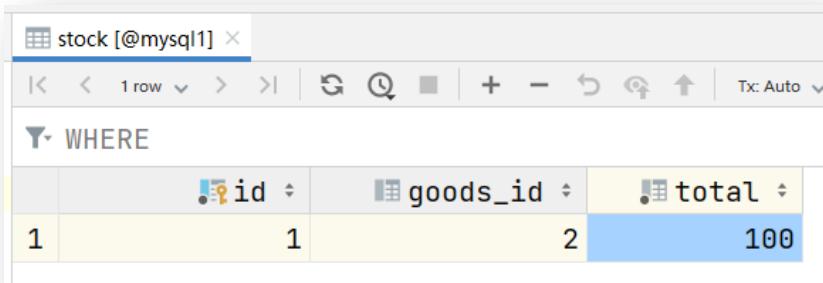
这里仅修改微服务名称与其连接的数据库，指定其连接的为 mysql1 主机。



```
application.yml
1 server:
2   port: 8081
3
4 spring:
5   application:
6     name: ec-stock
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848
11      username: nacos
12      password: nacos
13   jpa:
14     generate-ddl: true
15     show-sql: true
16     hibernate:
17       ddl-auto: none
18   datasource:
19     type: com.alibaba.druid.pool.DruidDataSource
20     driver-class-name: com.mysql.jdbc.Driver
21     url: jdbc:mysql://mysql1:3306/test?useUnicode=true&useSSL=false&characterEncoding=utf8
22     username: root
23     password: 111
24
```

## (9) DB 中添加数据

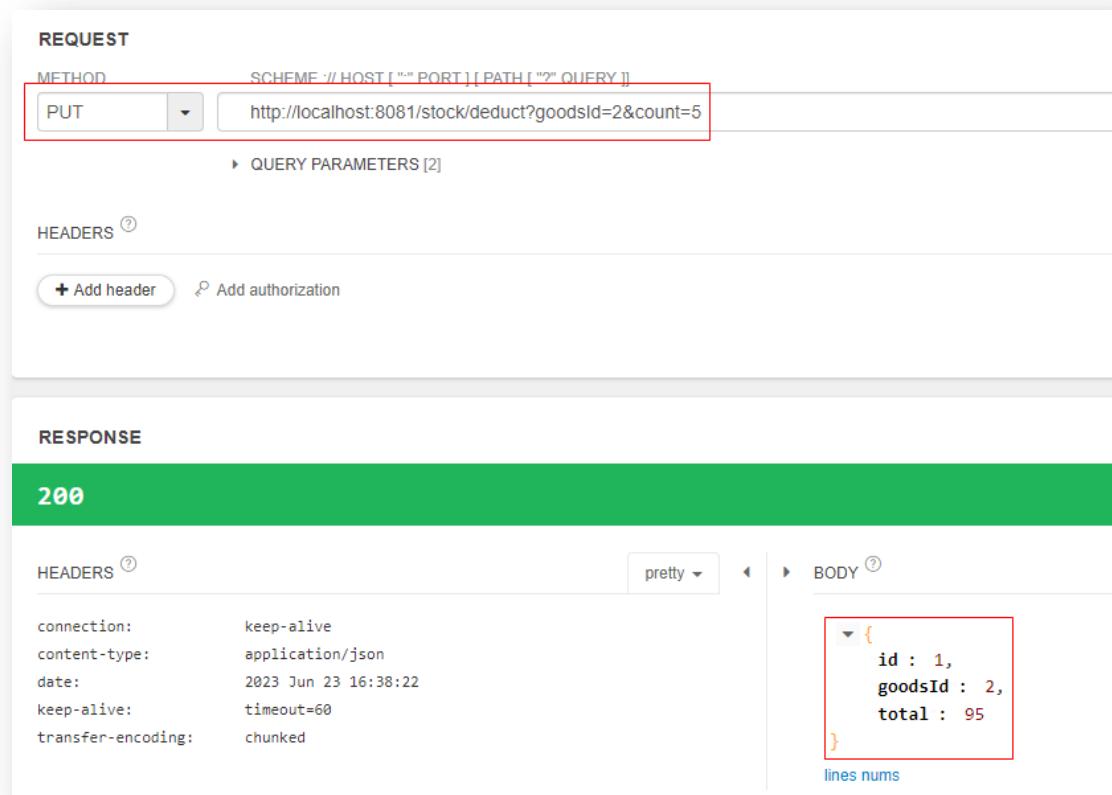
在 Nacos 启动的前提下，直接启动该工程即可。启动后在 mysql 主机的 test 数据库中就可看到 stock 表。在表中插入一条记录。



	 id	 goods_id	 total
1		1	2

## (10) 访问

在浏览器提交 PUT 请求，对 goodsId 为 2 的商品消费 count 为 5 个后，的确使库存少了 5 个。



**REQUEST**

METHOD: PUT    URL: `http://localhost:8081/stock/deduct?goodsId=2&count=5`

**HEADERS**

+ Add header    Add authorization

**RESPONSE**

200

**HEADERS**

connection: keep-alive  
content-type: application/json  
date: 2023 Jun 23 16:38:22  
keep-alive: timeout=60  
transfer-encoding: chunked

**BODY**

```
{  
  "id": 1,  
  "goodsId": 2,  
  "total": 95  
}
```

lines nums

### 7.3.3 Account 工程 07-ec-account-8083

#### (1) 定义工程

复制 07-ec-stock-8081 工程，并重命名为 07-ec-account-8083。将原来工程中除了启动类外所有类删除，保留原有的 POM 与配置文件。

## (2) 定义实体类 Account

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler", "fieldHandler"})
public class Account {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    no usages
    private Integer userId; // 用户id
    no usages
    private Double balance; // 账户余额
}
```

## (3) 定义 AccountRepository 接口

```
public interface AccountRepository extends JpaRepository<Account, Integer> {
    1 usage
    Account findAccountByUserId(Integer userId);
}
```

#### (4) 定义 AccountService 接口

```
public interface AccountService {  
    // 支出账户  
    1 usage 1 implementation  
    Account debitAccount(Integer userId, Double amount);  
}
```

#### (5) 定义 AccountServiceImpl 类

```
@Service  
public class AccountServiceImpl implements AccountService {  
  
    2 usages  
    @Autowired  
    private AccountRepository repository;  
  
    1 usage  
    @Override  
    public Account debitAccount(Integer userId, Double amount) {  
        Account account = repository.findAccountByUserId(userId);  
        if (account == null || account.getBalance() < amount) {  
            // 返回null, 表示支出账户失败  
            return null;  
        }  
        account.setBalance(account.getBalance() - amount);  
        Account at = repository.save(account);  
        return at;  
    }  
}
```

## (6) 定义 AccountController 类

```
@RestController
public class AccountController {
    1 usage
    @Autowired
    private AccountService service;

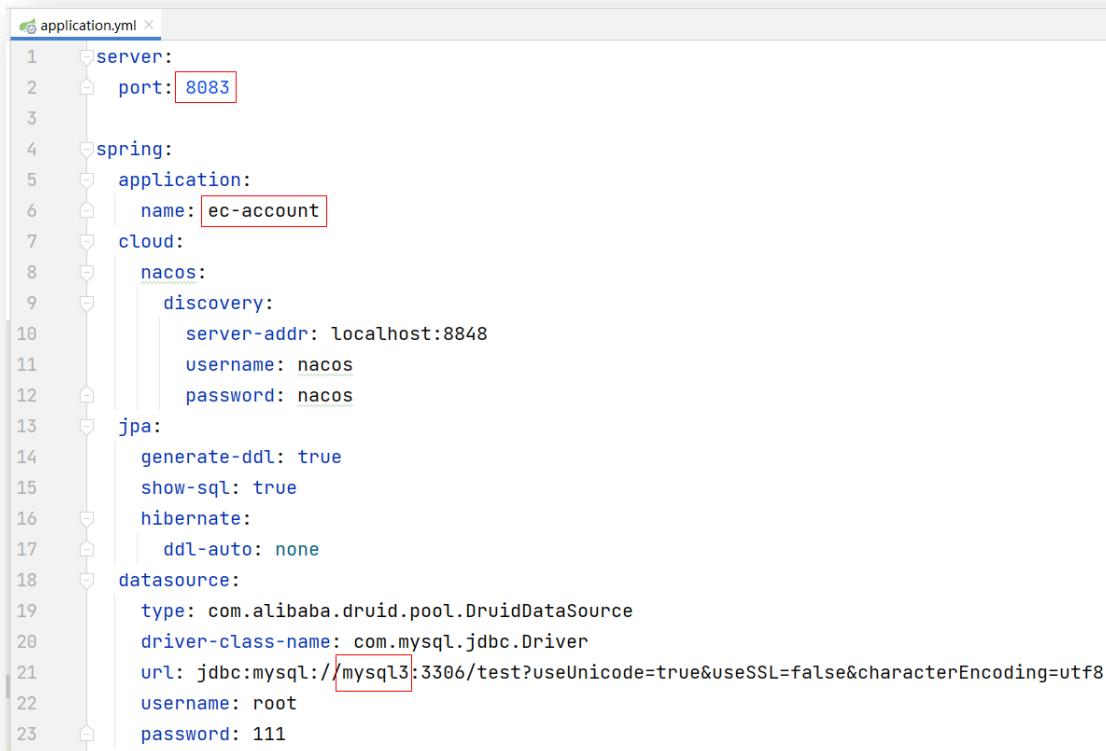
    no usages
    @PutMapping("account/debit")
    public Account debitHandle(@RequestParam("userId") Integer userId,
                               @RequestParam("amount") Double amount) {
        // 支出账户
        return service.debitAccount(userId, amount);
    }
}
```

## (7) 定义启动类

```
@SpringBootApplication
public class Account8083 {

    no usages
    public static void main(String[] args) {
        SpringApplication.run(Account8083.class, args);
    }
}
```

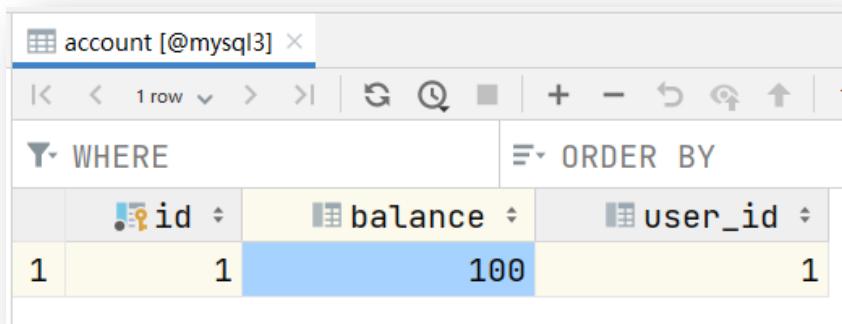
## (8) 修改配置文件



```
application.yml
1 server:
2   port: 8083
3
4 spring:
5   application:
6     name: ec-account
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848
11      username: nacos
12      password: nacos
13   jpa:
14     generate-ddl: true
15     show-sql: true
16     hibernate:
17       ddl-auto: none
18   datasource:
19     type: com.alibaba.druid.pool.DruidDataSource
20     driver-class-name: com.mysql.jdbc.Driver
21     url: jdbc:mysql://mysql3:3306/test?useUnicode=true&useSSL=false&characterEncoding=utf8
22     username: root
23     password: 111
```

## (9) DB 中添加数据

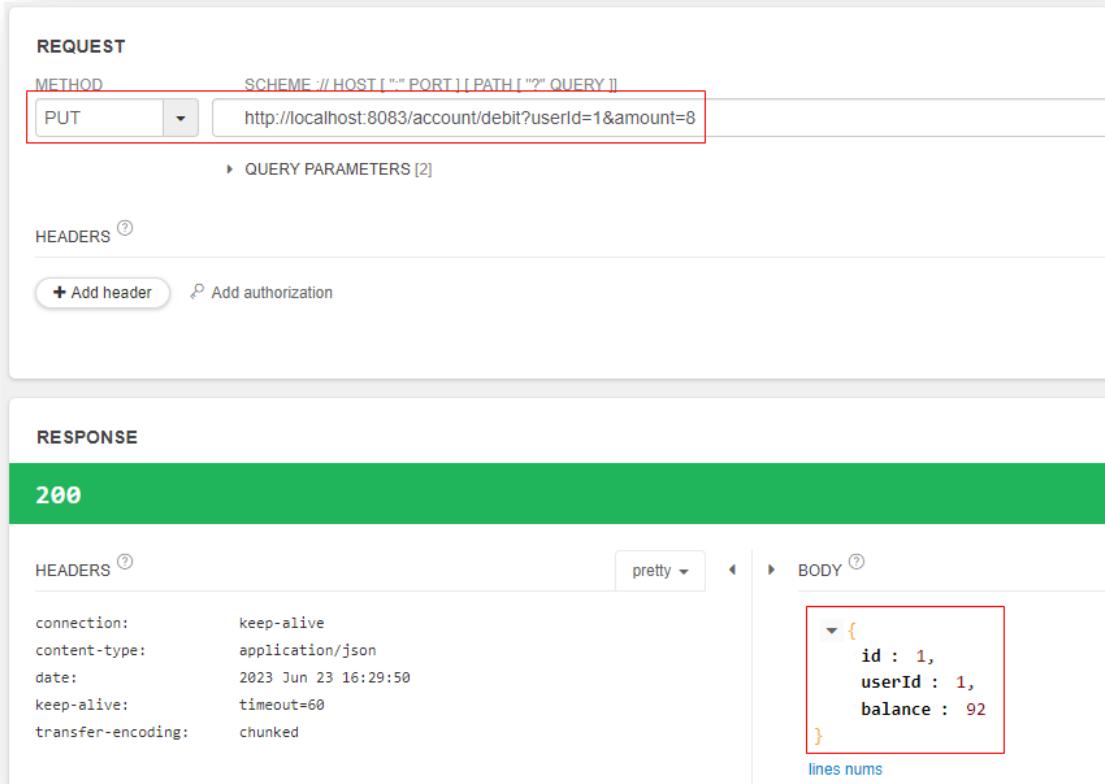
在 Nacos 启动的前提下，直接启动该工程即可。启动后在 mysql3 主机的 test 数据库中就可看到 account 表。在表中插入一条记录。



account [@mysql3]			
WHERE		ORDER BY	
	id	balance	user_id
	1	1	100

## (10) 访问

在浏览器中提交 PUT 请求，userId 为 1 的账户，在消费了 amount 8 元后，查看 account 表中相应记录发现，其账户余额的确少了 8 元。



The screenshot shows the Postman interface. In the REQUEST section, the method is set to PUT and the URL is http://localhost:8083/account/debit?userId=1&amount=8. In the RESPONSE section, the status code is 200. The Headers tab shows standard HTTP headers like Connection, Content-Type, Date, Keep-Alive, and Transfer-Encoding. The BODY tab displays a JSON object with id: 1, userId: 1, and balance: 92.

```
POST /account/debit?userId=1&amount=8 HTTP/1.1
Host: localhost:8083
Content-Type: application/json
Content-Length: 10
Connection: keep-alive
Accept: */*
User-Agent: PostmanRuntime/7.29.0
Accept-Encoding: gzip, deflate
Host: localhost:8083
Cookie: JSESSIONID=0000uJLjwvXWzqfB:1gkqy4n

{
  "id": 1,
  "userId": 1,
  "balance": 92
}
```

### 7.3.4 定义公共类 07-ec-common

由于后面要定义的 goodsorder 工程需要通过 OpenFeign 调用 account 工程，business 工程需要通过 OpenFeign 调用 stock 工程与 goodsorder 工程，所以这些消费者类中都需要用到 OpenFeign，用到生产者实体 Bean，所以这里就定义一个公共类 common，让消费者直接将其作为依赖导入即可。

## (1) 定义工程

直接定义一个 quickstart 的普通 Maven 工程即可。该工程无需启动类，只需定义需要的 OpenFeign 接口与三个实体类。

## (2) 添加依赖

由于该类需要使用 Lombok 与 OpenFeign，所以需要添加以下三个依赖。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
        <version>4.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
        <version>4.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.26</version>
    </dependency>
</dependencies>
```

### (3) 定义实体类

#### A、 Stock

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Stock {
    no usages
    private Integer id;
    no usages
    private Integer goodsId; // 商品id
    no usages
    private int total; // 库存量
}
```

#### B、 Account

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Account {
    no usages
    private Integer id;
    no usages
    private Integer userId; // 用户id
    no usages
    private Double balance; // 账户余额
}
```

### C、 Goodsorder

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Goodsorder {
    no usages
    private Integer id;
    no usages
    private Integer userId; // 用户id
    no usages
    private Integer goodsId; // 商品id
    no usages
    private Double goodsPrice; // 商品单价
    no usages
    private Integer count; // 订货数量
}
```

### (4) 定义 OpenFeign 接口

### A、 StockServie

```
@FeignClient("ms-stock")
public interface StockService {

    1 usage
    @PutMapping("/stock/deduct")
    Stock deductStock(@RequestParam("goodsId") Integer goodsId,
                      @RequestParam("count") Integer count);
}
```

## B、 AccountService

```
@FeignClient("ms-account")
public interface AccountService {

    1 usage
    @PutMapping("/account/debit")
    Account debitAccount(@RequestParam("userId") Integer userId,
                          @RequestParam("amount") Double amount);
}
```

## C、 GoodsorderService

```
@FeignClient("ms-goodsorder")
public interface GoodsorderService {

    1 usage
    @PostMapping("/goodsorder/create")
    Goodsorder createOrder(@RequestBody Goodsorder goodsOrder);
}
```

### 7.3.5 goodsorder 工程 07-ec-goodsorder-8082

#### (1) 定义工程

复制 07-ec-stock-8081 工程，并重命名为 07-ec-goodsorder-8082。将原来工程中的所有类删除，保留原有的 POM 与配置文件。

## (2) 添加依赖

由于 goodsorder 工程要采用负载均衡方式来调用 account 工程, 所以需要引入 OpenFeign 依赖 loadbalancer 依赖及 07-ec-common 依赖。

```
<dependency>
    <groupId>com.abc</groupId>
    <artifactId>07-ec-common</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

### (3) 定义实体类 Goodsorder

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler", "fieldHandler"})
public class Goodsorder {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    no usages
    private Integer userId; // 用户id
    no usages
    private Integer goodsId; // 商品id
    no usages
    private Double goodsPrice; // 商品单价
    no usages
    private Integer count; // 订货数量
}
```

### (4) 定义 GoodsorderRepository 接口

```
2 usages
public interface GoodsorderRepository extends JpaRepository<Goodsorder, Integer> { }
```

## (5) 定义 GoodsorderServiceLocal 接口

```
public interface GoodsorderServiceLocal {  
    1 usage 1 implementation  
    Goodsorder createGoodsorder(Goodsorder goodsorder);  
}
```

## (6) 定义 GoodsorderServiceImpl 类

```
@Service  
public class GoodsorderServiceImpl implements GoodsorderServiceLocal {  
    1 usage  
    @Autowired  
    private GoodsorderRepository repository;  
  
    1 usage  
    @Override  
    public Goodsorder createGoodsorder(Goodsorder goodsorder) {  
        return repository.save(goodsorder);  
    }  
}
```

## (7) 定义 GoodsorderController 类

```
@RestController
public class GoodsorderController {
    1 usage
    @Autowired
    private GoodsorderServiceLocal goodsorderService;
    1 usage
    @Autowired
    private AccountService accountService;

    no usages
    @PostMapping("goodsorder/create")
    public Goodsorder createHandle(@RequestBody Goodsorder goodsorder) {
        // 计算订单总价
        Double total = goodsorder.getGoodsPrice() * goodsorder.getCount();
        // 支出账户
        Account account = accountService.debitAccount(goodsorder.getUserId(), total);

        // 账户余额不足时返回null
        if(account == null) {
            return null;
        }
        // 在账户余额充足的前提下, 添加订单记录, 返回Goodsorder
        return goodsorderService.createGoodsorder(goodsorder);
    }
}
```

## (8) 定义启动类



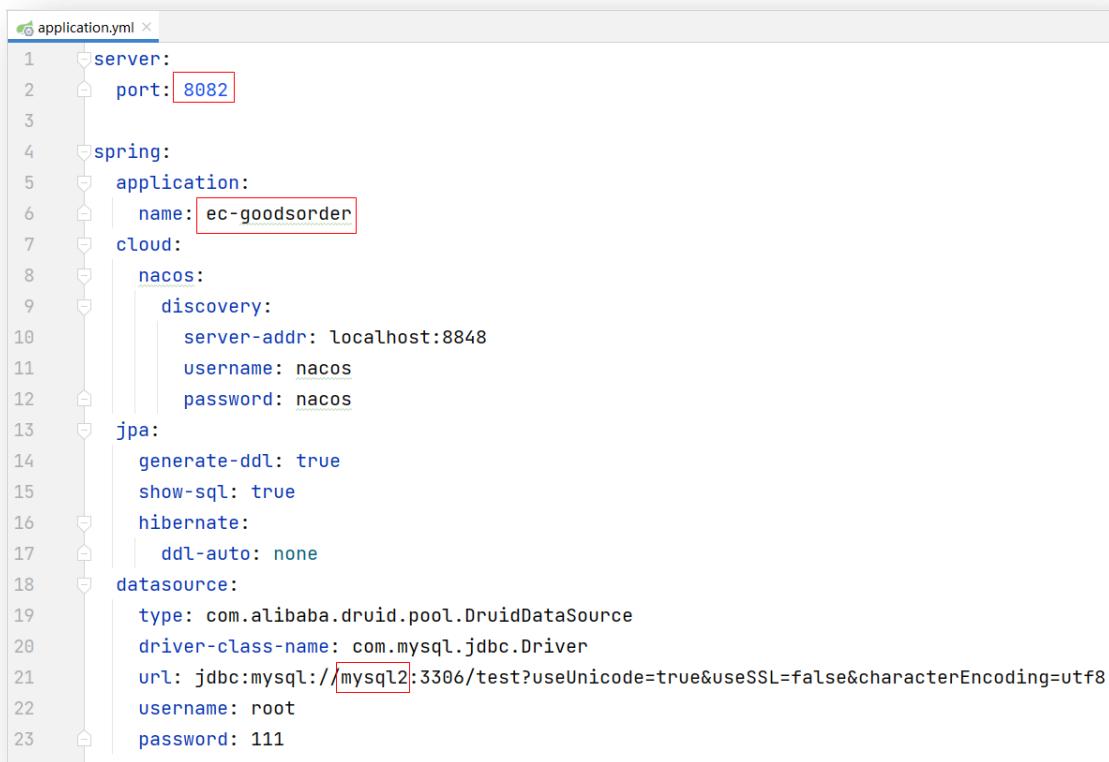
```
@EnableFeignClients
@SpringBootApplication
public class Goodsorder8082 {

    no usages

    public static void main(String[] args) {
        SpringApplication.run(Goodsorder8082.class, args);
    }

}
```

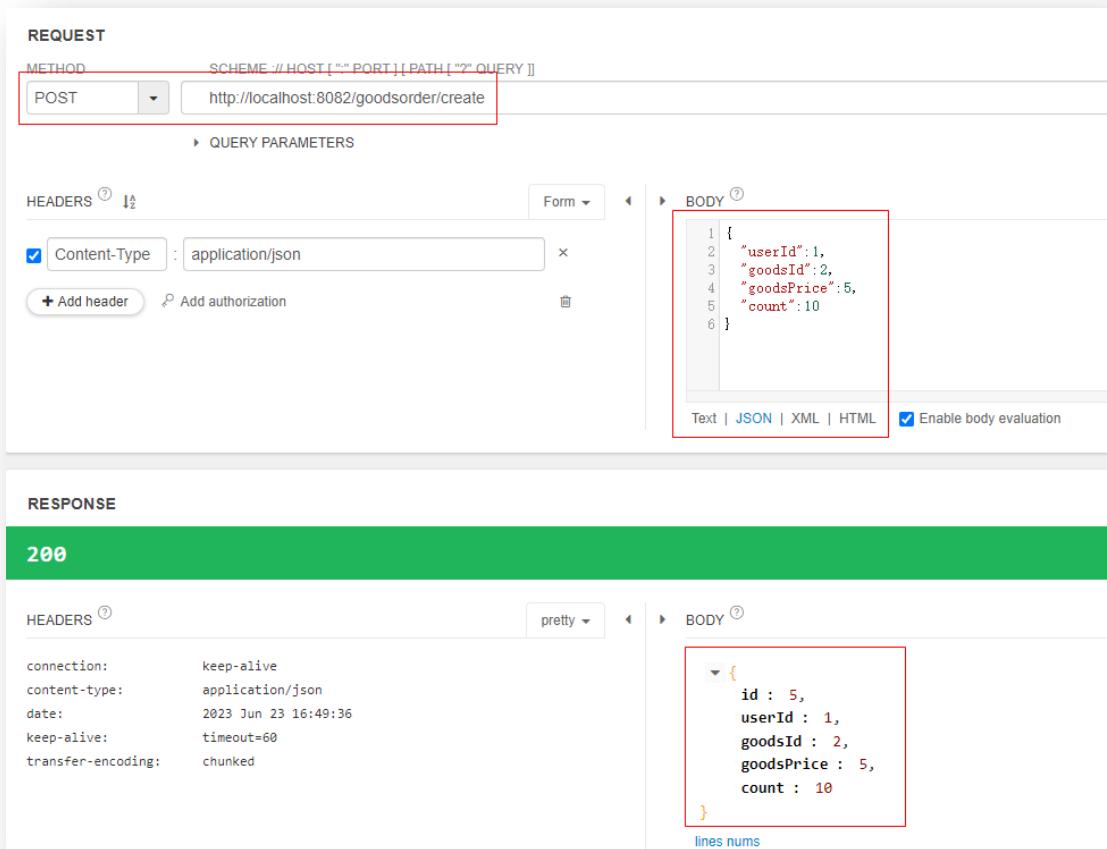
## (9) 修改配置文件



```
application.yml
1  server:
2      port: 8082
3
4  spring:
5      application:
6          name: ec-goodsorder
7
8      cloud:
9          nacos:
10             discovery:
11                 server-addr: localhost:8848
12                 username: nacos
13                 password: nacos
14
15         jpa:
16             generate-ddl: true
17             show-sql: true
18             hibernate:
19                 ddl-auto: none
20
21         datasource:
22             type: com.alibaba.druid.pool.DruidDataSource
23             driver-class-name: com.mysql.jdbc.Driver
24             url: jdbc:mysql://mysql2:3306/test?useUnicode=true&useSSL=false&characterEncoding=utf8
25             username: root
26             password: 111
```

## (10) 访问

在 Nacos 启动的前提下，需要启动 07-ec-account-8083 与 07-ec-goodsorder-8082 两个工程。然后提交 POST 请求：userId 为 1 的用户购买了 goodsId 为 2 的商品 10 件，每件单价为 5 元。查看数据库发现，goodsorder 表中的确新增了一条订单记录，且 account 表中的账户余额也的确减少了 50 元。



The screenshot shows a POST request to `http://localhost:8082/goodsorder/create`. The request body contains the following JSON data:

```
1: {  
2:   "userId": 1,  
3:   "goodsId": 2,  
4:   "goodsPrice": 5,  
5:   "count": 10  
6: }
```

The response is a 200 OK status with the following JSON data:

```
{  
  "id": 5,  
  "userId": 1,  
  "goodsId": 2,  
  "goodsPrice": 5,  
  "count": 10  
}
```

## 7.3.6 定义采购工程 07-ec-business-8080

## (1) 定义工程

复制 07-ec-goodsorder-8082 工程，并重命名为 07-ec-business-8080。将原来工程中的所有类删除，保留原有的配置文件。

## (2) 定义 BusinessController 类

```
@RestController
public class BusinessController {
    1 usage
    @Autowired
    private StockService stockService;
    1 usage
    @Autowired
    private GoodsorderService goodsorderService;
```

```
    // no usage
    @GetMapping("business/purchase")
    public String purchaseHandle(@RequestParam("userId") Integer userId,
                                 @RequestParam("goodsId") Integer goodsId,
                                 @RequestParam("goodsPrice") Double goodsPrice,
                                 @RequestParam("count") Integer count) {
        // 减库存
        Stock stock = stockService.deductStock(goodsId, count);
        if (stock == null) {
            return "库存不足，下单失败";
        }

        // 生成订单
        Goodsorder goodsorder = new Goodsorder(null, userId, goodsId, goodsPrice, count);
        // 下订单
        Goodsorder go = goodsorderService.createOrder(goodsorder);
        if (go == null) {
            return "账户余额不足，下单失败";
        }

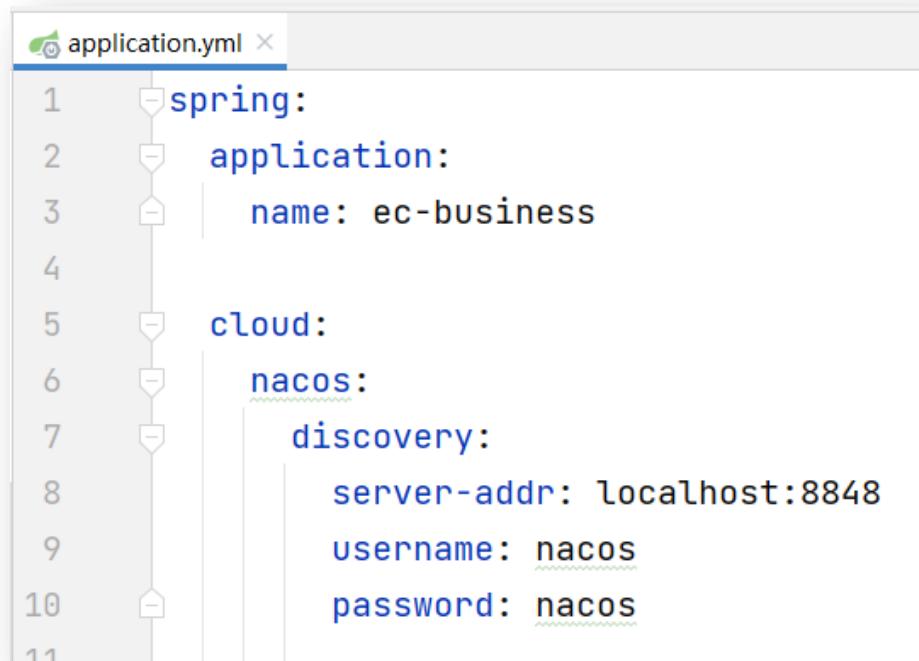
        return "下单成功";
    }
}
```

## (3) 定义启动类



```
1  @EnableFeignClients
2  @SpringBootApplication
3  public class Business8080 {
4
5      no usages
6      public static void main(String[] args) {
7          SpringApplication.run(Business8080.class, args);
8      }
9
10 }
```

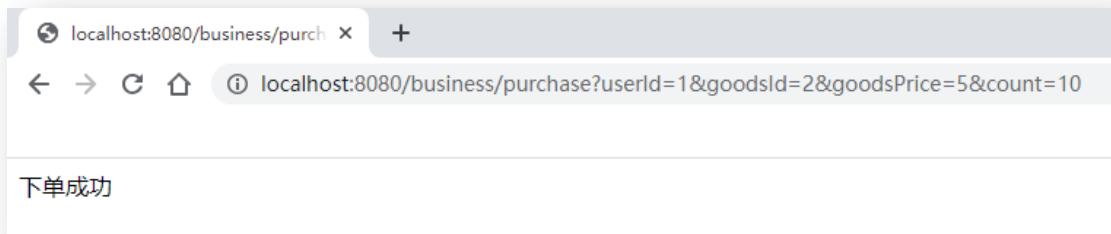
## (4) 修改配置文件



```
1  spring:
2      application:
3          name: ec-business
4
5      cloud:
6          nacos:
7              discovery:
8                  server-addr: localhost:8848
9                  username: nacos
10                 password: nacos
```

## (5) 运行访问

在 Nacos 启动的前提下，将 account、goodsorder、stock 与 business 工程都启动。通过浏览器提交 GET 请求后，即可看到 account 表中的账户余额减少了 50，stock 表中库存量减少了 5，goodsorder 表中增加了一条订单记录。

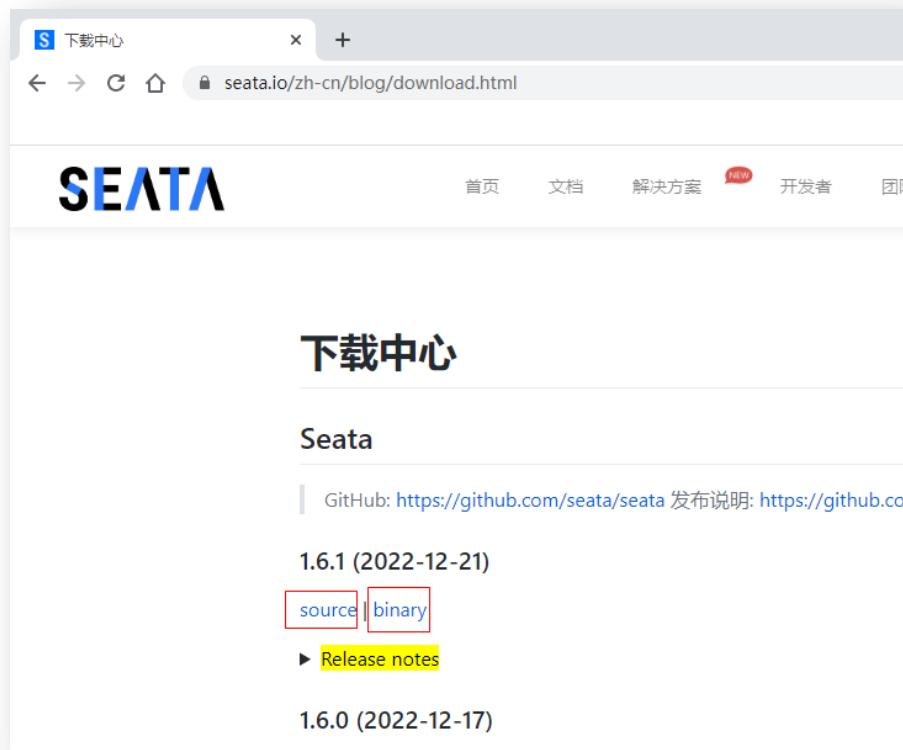


## 7.4 Seata-Server 安装与配置

无论是 AT 模式，还是 TCC 模式或 XA 模式，都需要有事务协调器 TC，即 Seata Server，所以下面先来学习 Seata Server 如何安装配置。

### 7.4.1 Seata Server 下载

Seata 的源码包与二进制包均需要下载。因为 Seata Client 需要使用 Seata 的源码包中的一个 sql 脚本文件。



## 7.4.2 Seata Server 存储模式

### (1) 分类

Seata Server 需要对全局事务与分支事务进行存储，以便对它们进行管理。其存储模式目前支持三种：file、db 与 redis。

- file 模式：会将相关数据存储在本地文件中，一般用于 Seata Server 的单机测试。
- db 模式：会将相关数据存储在数据库中，一般用于生产环境下的 Seata Server 集群部署。生产环境下使用最多的模式。
- redis 模式：会将相关数据存储在 redis 中，一般用于生产环境下的 Seata Server 集群部署。性能略高于 db 模式，如果对性能要求较高，可选择 redis 模式。

### (2) file 模式

#### A、修改 application.yml

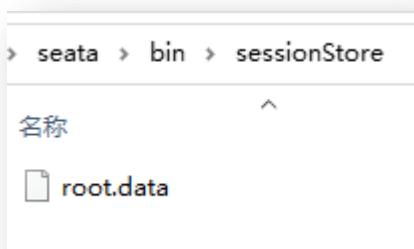
将 seata-server-1.6.1.zip 解压后，修改 seata 解压目录下的 conf 目录中的 application.yml 文件。在该文件中需要配置三类信息：Seata 的配置中心、Seata 的注册中心，及回滚日志信

息。不过，对于 file 模式，只需要修改以下位置即可。

```
38 日 seata:
39 日   config:
40     # support: nacos, consul, apollo, zk, etcd3
41     type: file
42 日   registry:
43     # support: nacos, eureka, redis, zk, consul, etcd3, sofa
44     type: file
45 日   store:
46     # support: file 、 db 、 redis
47     mode: file
48 日   session:
49     mode: file
50 日   lock:
51     mode: file
52 日   file:
53     dir: sessionStore
54     max-branch-session-size: 16384
55     max-global-session-size: 512
56     file-write-buffer-cache-size: 16384
57     session-reload-read-size: 100
58     flush-disk-mode: async
59
60 # server:
61 #   service port: 8001 #If not configured, the default is 16384
```

## B、启动 Seata-Server

在 seata/bin 目录下直接双击 seata-server.bat 批处理文件即可启动。启动后在该 bin 目录中就可以直接看到生成的 sessionStore 目录及其中的 root.data 文件。

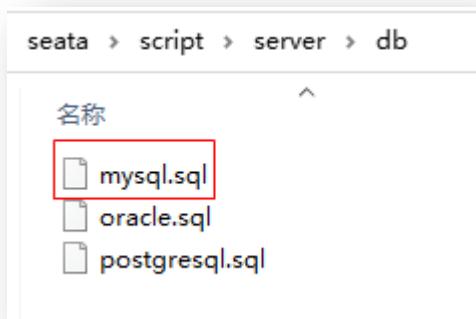


### (3) db 模式

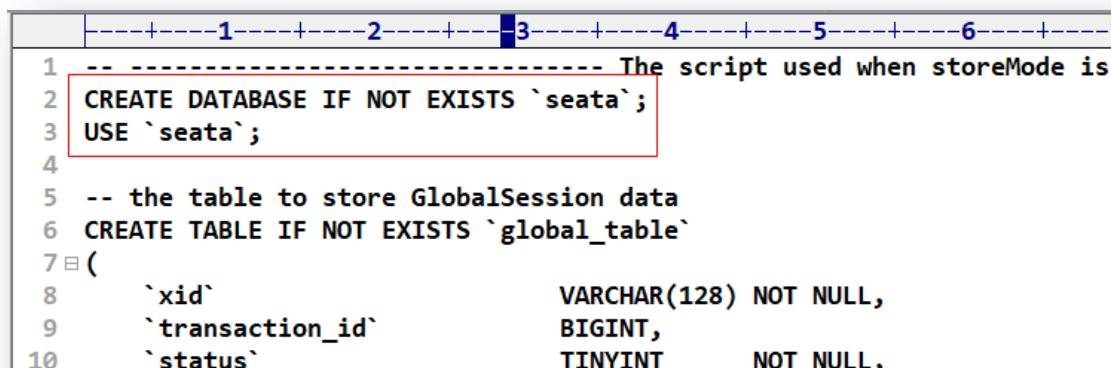
file 模式一般用于简单测试，生产环境下使用的是 db 模式。

## A、运行 mysql.sql 脚本

在 seata 包解压目录的 script/server/db 下找到 mysql.sql 文件。该脚本文件中创建了 4 张表。这 4 张表都是用于保存整个系统中分布式事务相关日志数据的。

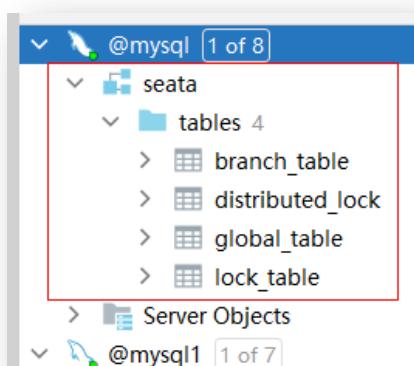


该脚本文件中仅有建表语句，没有创建数据库的语句，说明使用什么数据库名称都可以。为了方便，我们在脚本文件中直接添加了建库语句，并指定数据库名称为 seata。



```
1 ----- 1 ----- 2 ----- 3 ----- 4 ----- 5 ----- 6 -----  
1 ----- The script used when storeMode is  
2 CREATE DATABASE IF NOT EXISTS `seata`;  
3 USE `seata`;  
4  
5 -- the table to store GlobalSession data  
6 CREATE TABLE IF NOT EXISTS `global_table`  
7 (  
8     `xid`          VARCHAR(128) NOT NULL,  
9     `transaction_id` BIGINT,  
10    `status`        TINYINT      NOT NULL,
```

在 Idea 右侧 Database 中的 mysql 数据库上运行该脚本。运行完毕，从中可以查看到创建的 DB 及表。



## B、修改 application.yml

修改 seata 解压目录下的 conf 目录中的 application.yml 文件。在该文件中需要配置三类信息：Seata 的配置中心、Seata 的注册中心，及回滚日志信息。

```
38 日 seata:
39 日   config:
40     # support: nacos, consul, apollo, zk, etcd3
41     type: nacos
42 日   nacos:
43     server-addr: 127.0.0.1:8848
44     namespace:
45     group: SEATA_GROUP
46     username: nacos
47     password: nacos
48     context-path:
49     ##if use MSE Nacos with auth, mutex with userna
50     #access-key:
51     #secret-key:
52     data-id: seataServer.properties
53 日   registry:
```

```
53 日   registry:
54     # support: nacos, eureka, redis, zk, consu
55     type: nacos
56 日     nacos:
57     application: seata-server
58     server-addr: 127.0.0.1:8848
59     group: SEATA_GROUP
60     namespace:
61     #cluster: default
62     username: nacos
63     password: nacos
64     context-path:
65     ##if use MSE Nacos with auth, mutex with
66     #access-key:
67     #secret-key:
68 日   store:
```

```
68 日 store:  
69     # support: file 、 db 、 redis  
70     mode: db  
71 日 session:  
72     mode: db  
73 日 lock:  
74     mode: db  
75 日 db:  
76     datasource: druid  
77     db-type: mysql  
78     driver-class-name: com.mysql.jdbc.Driver  
79     url: jdbc:mysql://mysql:3306/seata?rewriteBatchedStatements=true  
80     user: root  
81     password: 111  
82     min-conn: 10  
83     max-conn: 100  
84     global-table: global_table  
85     branch-table: branch_table  
86     lock-table: lock_table  
87     distributed-lock-table: distributed_lock  
88     query-limit: 1000  
89     max-wait: 5000  
90
```

## C、修改 config.txt

修改 seata 解压目录的 script/config-center 下的 config.txt 文件。这里的内容都是 key-value，将来都是作为 nacos 配置中心中的数据出现的，将来打开 nacos 可以根据其 key 逐条查看到。

### a、修改存储模式

指定这里要使用的存储模式为 db。

```
67  
68 #Transaction storage configuration, only +  
69 store.mode=db  
70 store.lock.mode=db  
71 store.session.mode=db  
72 #Used for password encryption  
73 #store.publicKey=  
74
```

将 store.mode、store.lock.mode、store.session.mode 中原来的 file 值修改为 db。再将公钥行注释掉。

## b、修改 db 的连接四要素

```
82
83 #These configurations are required if the `store mode` is `db`. If `store.mode,store.lock.mo
84 store.db.datasource=druid
85 store.db.dbType=mysql
86 store.db.driverClassName=com.mysql.jdbc.Driver
87 store.db.url=jdbc:mysql://mysql2:3306/seata?useUnicode=true&rewriteBatchedStatements=true
88 store.db.user=root
89 store.db.password=111
90 store.db.minConn=5
91 store.db.maxConn=30
92 store.db.globalTable=global_table
93 store.db.branchTable=branch_table
94 store.db.distributedLockTable=distributed_lock
95 store.db.queryLimit=100
96 store.db.lockTable=lock_table
97 store.db.maxWait=5000
98
```

## c、删除其它存储模式配置

由于这里指定的存储模式是 db, 所以需要将 file 模式与 redis 模式相关的配置全部删除。

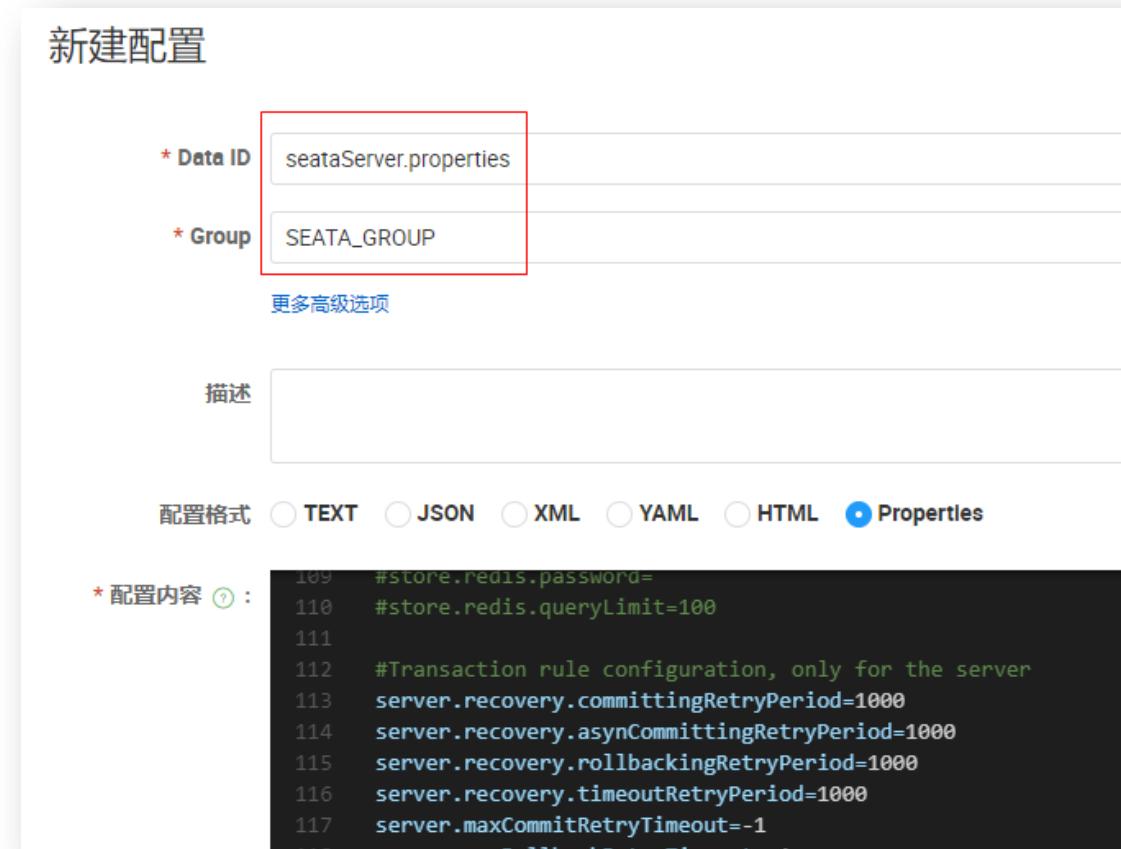
```
74
75 #If `store.mode,store.lock.mode,store.session.mode` are not equal to `file`, you can remove the configuration block.
76 store.file.dir=file_store/data
77 store.file.maxBranchSessionSize=16384
78 store.file.maxGlobalSessionSize=512
79 store.file.fileWriteBufferCacheSize=16384
80 store.file.flushDiskMode=async
81 store.file.sessionReloadReadSize=100
82
```

```
98
99 #These configurations are required if the `store mode`
100 store.redis.mode=single
101 store.redis.single.host=127.0.0.1
102 store.redis.single.port=6379
103 store.redis.sentinel.masterName=
104 store.redis.sentinel.sentinelHosts=
105 store.redis.maxConn=10
106 store.redis.minConn=1
107 store.redis.maxTotal=100
108 store.redis.database=0
109 store.redis.password=
110 store.redis.queryLimit=100
111
```

## D、定义 seataServer.properties

在 Nacos 配置中心定义 seataServer.properties 文件。该文件是在上述 application.yml 中指定的 data-id 文件。将前面修改后的 config.txt 文件内容全部复制到 seataServer.properties

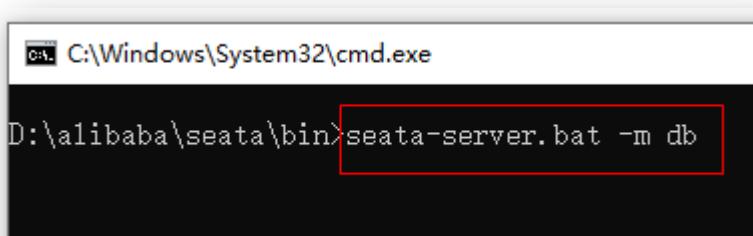
文件中。



## E、启动 Seata-server

### a、运行命令

在 seata 解压目录下的 bin 目录中有个文件 seata-server.bat，在命令行运行这个批处理文件。



```
C:\Windows\System32\cmd.exe
D:\alibaba\seata\bin>seata-server.bat -m db
```

当看到如下日志时，说明 seata server 启动成功。

```
C:\Windows\System32\cmd.exe - seata-server.bat -m db
19:28:06.686 INFO --- [           main] o.s.s.web.DefaultSecurityFilterChain : Will secure Ant [pattern='/**/*.png'] with []
19:28:06.686 WARN --- [           main] o.s.s.c.a.web.builders.WebSecurity  : You are asking Spring Security to ignore Ant [pattern='/**/*.ico']. This is not recommended -- please use permitAll via HttpSecurity#authorizeHttpRequests instead.
19:28:06.687 INFO --- [           main] o.s.s.web.DefaultSecurityFilterChain : Will secure Ant [pattern='/**/*.ico'] with []
19:28:06.687 WARN --- [           main] o.s.s.c.a.web.builders.WebSecurity  : You are asking Spring Security to ignore Ant [pattern='/console-fe/public/**']. This is not recommended -- please use permitAll via HttpSecurity#authorizeHttpRequests instead.
19:28:06.687 INFO --- [           main] o.s.s.web.DefaultSecurityFilterChain : Will secure Ant [pattern='/console-fe/public/**'] with []
19:28:06.687 WARN --- [           main] o.s.s.c.a.web.builders.WebSecurity  : You are asking Spring Security to ignore Ant [pattern='/api/v1/auth/login']. This is not recommended -- please use permitAll via HttpSecurity#authorizeHttpRequests instead.
19:28:06.687 INFO --- [           main] o.s.s.web.DefaultSecurityFilterChain : Will secure Ant [pattern='/api/v1/auth/login'] with []
19:28:06.706 INFO --- [           main] o.s.s.web.DefaultSecurityFilterChain : Will not secure any request
19:28:06.728 INFO --- [           main] o.a.coyote.http11.Http1NioProtocol : Starting ProtocolHandler ["http-nio-7091"]
19:28:06.741 INFO --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 7091 (http) with context path
19:28:06.751 INFO --- [           main] io.seata.server.ServerApplication : Started ServerApplication in 2.551 seconds (JVM running for 2.929)
19:28:07.290 INFO --- [           main] com.alibaba.druid.pool.DruidDataSource : (dataSource-1) init
19:28:07.579 INFO --- [           main] i.s.core.rpc.netty.NettyServerBootstrap : Server started, service listen port: 8091
19:28:07.592 INFO --- [           main] com.alibaba.nacos.client.naming : initializer namespace from System Property : null
19:28:07.661 INFO --- [           main] com.alibaba.nacos.client.naming : [BEAT] adding beat: BeatInfo(port=8091, ip='192.168.237.1', weight=1.0, serviceName='SEATA_GROUP@seata-server', cluster='default', metadata={}, scheduled=false, period=5000, stopped=false) to beat map
19:28:07.662 INFO --- [           main] com.alibaba.nacos.client.naming : [REGISTER-SERVICE] public registering service SEATA_GROUP@seata-server with instance: Instance(instanceId=null, ip='192.168.237.1', port=8091, weight=1.0, healthy=true, enabled=true, ephemeral=true, clusterName='default', serviceName='null', metadata={})
19:28:07.665 INFO --- [           main] io.seata.server.ServerRunner : seata server started in 913 millSeconds
```

## b、查看 nacos

在 nacos 中可以看到 seata-server 的服务时，表示 seata 启动成功了。



服务名	分组名称	集群数目
seata-server	SEATA_GROUP	1

## (4) redis 模式

### A、修改 application.yml

直接复制 db 模式的整个 seata 解压包，修改其 conf 目录中的 application.yml 文件。仅修改如下的存储模式，其它配置与 db 中的相同。

```
68 日 store:  
69     # support: file 、 db 、 redis  
70     mode: redis  
71 日 session:  
72     mode: redis  
73 日 lock:  
74     mode: redis  
75 日 redis:  
76     mode: single  
77     database: 0  
78     min-conn: 10  
79     max-conn: 100  
80     #password:  
81     max-total: 100  
82     query-limit: 1000  
83 日     single:  
84         host: 192.168.192.102  
85         port: 6379  
86 日 #sentinel:  
87     #master-name:  
88     #sentinel-hosts:  
89
```

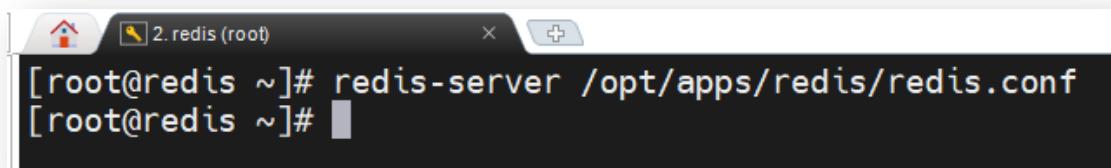
## B、修改 seataServer.properties

修改 nacos 配置中心中的 seataServer.properties 文件中有关存储模式的配置。

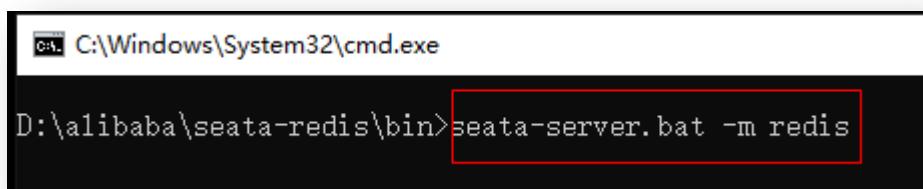
```
67  
68     #Transaction storage configuration, only t  
69     store.mode=redis  
70     store.lock.mode=redis  
71     store.session.mode=redis  
72     #Used for password encryption  
73     #store.publicKey=  
74
```

```
98
99 #These configurations are required if the `store mode` is set to `single`
100 store.redis.mode=single
101 store.redis.single.host=192.168.192.102
102 store.redis.single.port=6379
103 #store.redis.sentinel.masterName=
104 #store.redis.sentinel.sentinelHosts=
105 store.redis.maxConn=10
106 store.redis.minConn=1
107 store.redis.maxTotal=100
108 store.redis.database=0
109 #store.redis.password=
110 store.redis.queryLimit=100
111
```

### C、启动 redis



### D、启动 Seata-Server



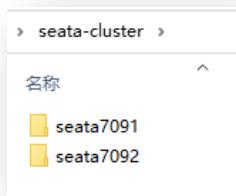
#### 7.4.3 Seata Server 集群

对于 Seata Server 集群的搭建，其存储模式只能设置为 db 或 redis。

## (1) 集群搭建

Seata Server 集群的搭建非常简单，对于同一台主机上的多个集群节点，只需复制多份前面配置好的 db 或 redis 存储模式的 Seata 解压目录，修改它们 conf/application.yml 中的内置 Tomcat 的端口号，即 server.port，保证不重复即可。

这里复制了两份 db 存储模式的 Seata 解压目录，server.port 分别指定为 7091 与 7092。即该集群包含两个节点。



## (2) 启动

分别进入这两个目录的 bin 目录下运行如下命令。命令中需要通过 -n 选项指定当前 Seata Server 节点的序号，要保证唯一。

```
C:\Windows\System32\cmd.exe
D:\alibaba\seata-cluster\seata7091\bin>seata-server.bat -m db -n 1
```

```
C:\Windows\System32\cmd.exe
D:\alibaba\seata-cluster\seata7092\bin>seata-server.bat -m db -n 2
```

启动完毕后，在 Nacos 注册中心可以看到出现了两个 seata-server 实例。



The screenshot shows the Nacos 2.2.1 interface with the 'Service List' tab selected. The search bar contains 'public | hello'. A table lists a single service: 'seata-server' under 'Service Name', 'SEATA\_GROUP' under 'Group Name', '1' under 'Cluster Count', '2' under 'Instance Count' (which is highlighted with a red border), and '2' under 'Healthy Instance Count'.

## 7.5 AT 模式下的 Seata Client

Seata Client，即安装有 Seata 依赖的应用。下面首先来学习 Seata 默认的，也是生产中使用最多的分布式事务模式 AT 模式的用法。

### 7.5.1 添加 undo\_log 表

undo\_log 表，用于保存需要回滚的业务数据。在所有业务数据库中均需要添加 undo\_log 表。在本例中，在 mysql1、mysql2 与 mysql3 的 test 数据库中均需添加 undo\_log 表。

建表语句脚本在 seata 源码解压目录的 script/client/at/db 目录下的 mysql.sql 中。

### 7.5.2 创建工程

为了保留下原始代码，我们这里复制 07-ec-stock-8081、07-ec-goodsorder-8082、07-ec-account-8083 与 07-ec-business-8080 四个工程，分别重新命名为 07-at-stock-8081、07-at-goodsorder-8082、07-at-account-8083 与 07-at-business-8080，将这四个新工程改造为 AT 模式的 Seata-Client。

### 7.5.3 共同的修改

这四个工程的改造方案完全相同。下面以 07-at-stock-8081 工程改造过程为例。

#### (1) 添加依赖

在 pom 中添加 Spring Cloud Alibaba 对于 seata 的依赖。

```
<dependency>
<groupId>io.seata</groupId>
```

```

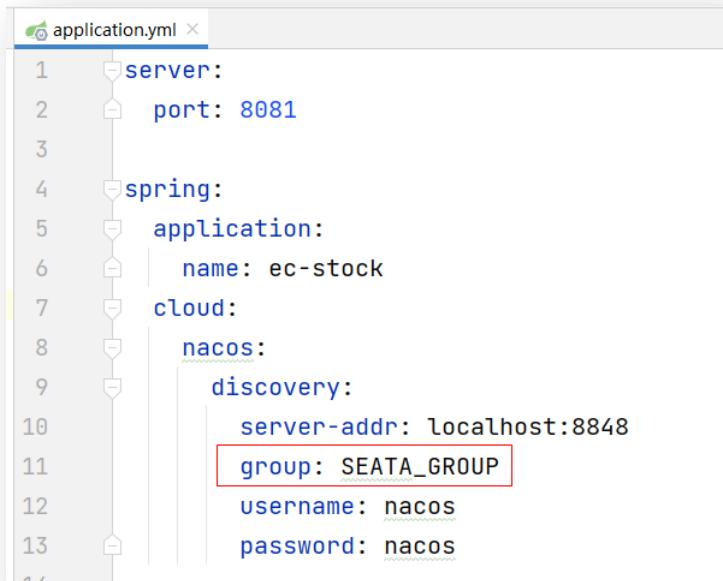
<artifactId>seata-spring-boot-starter</artifactId>
<version>1.6.1</version>
</dependency>
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.seata</groupId>
            <artifactId>seata-spring-boot-starter</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

## (2) 修改配置文件

### A、指定 group

指定当前服务要注册到 Nacos 注册中心的 SEATA\_GROUP 组中，与 Seata Server 在一个组。



### B、查看 nacos

这四个工程启动后，在 nacos 注册中心就可以看到这四个服务再包括 seata-server，它们都在 SEATA\_GROUP 分组中。



The screenshot shows the Nacos 2.2.1 service list interface. On the left, there's a sidebar with options like Configuration Management, Service Management, and Service List (which is selected). The main area is titled "服务列表" (Service List) and shows a table of services. The table has columns: 服务名 (Service Name), 分组名称 (Group Name), 集群数目 (Cluster Count), 实例数 (Instance Count), and 健康实例数 (Healthy Instance Count). Five services are listed: ec-stock, ec-account, ec-goodsorder, ec-business, and seata-server. All five services belong to the "SEATA\_GROUP" group. A red box highlights the "Group Name" column for these services.

服务名	分组名称	集群数目	实例数	健康实例数
ec-stock	SEATA_GROUP	1	1	1
ec-account	SEATA_GROUP	1	1	1
ec-goodsorder	SEATA_GROUP	1	1	1
ec-business	SEATA_GROUP	1	1	1
seata-server	SEATA_GROUP	1	1	1

#### 7.5.4 添加@GlobalTransactional 注解

这里的 07-at-business-8080 工程充当着 TM，所以需要在该工程中开启分布式事务。在其 BusinessController 类的 purchaseHandle() 方法上添加 @GlobalTransactional 注解。



The screenshot shows the Java code for the BusinessController class. The code includes autowiring for StockService and GoodsorderService, and a method named purchaseHandle(). The @GetMapping annotation points to the URL "/business/purchase". The purchaseHandle() method takes four parameters: userId, goodsId, goodsPrice, and count. It contains logic to deduct stock from the stockService. A red box highlights the @GlobalTransactional annotation above the purchaseHandle() method.

```

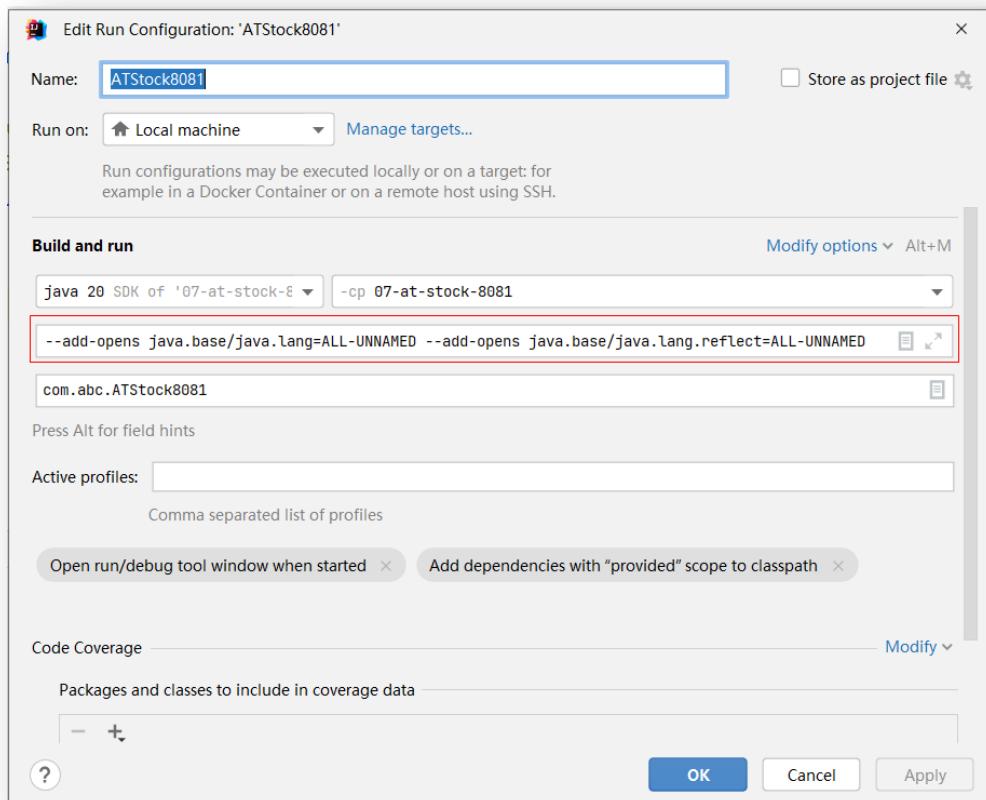
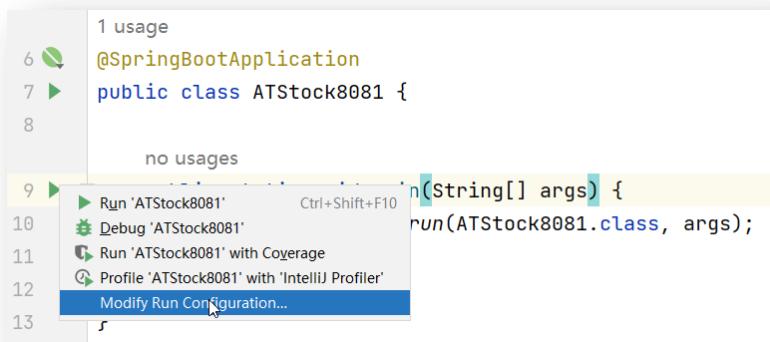

@RestController
public class BusinessController {
    1 usage
    @Autowired
    private StockService stockService;
    1 usage
    @Autowired
    private GoodsorderService goodsorderService;

    no usages
    @GlobalTransactional
    @GetMapping("/business/purchase")
    public String purchaseHandle(@RequestParam("userId") Integer userId,
                                 @RequestParam("goodsId") Integer goodsId,
                                 @RequestParam("goodsPrice") Double goodsPrice,
                                 @RequestParam("count") Integer count) {
        // 减库存
        Stock stock = stockService.deductStock(goodsId, count);
        if (stock == null) {
            return "库存不足, 下单失败";
        }
    }
}


```

## 7.5.5 启动应用

在启动 07-at-stock-8081、07-at-goodsorder-8082 与 07-at-account-8083 工程时会启动失败，主要是由于 JDK 版本不兼容导致的。在启动项中添加以下启动参数就可解决。不过，07-ec-business-8080 工程的启动是没有问题的。

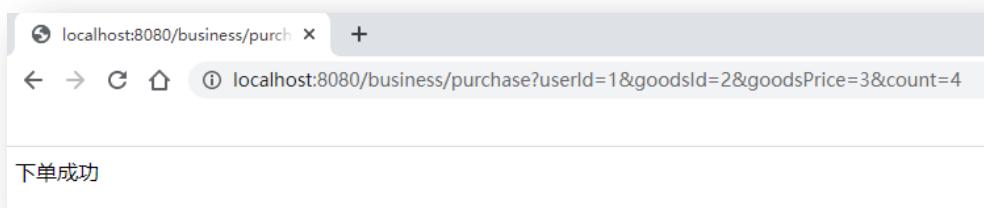


```
--add-opens java.base/java.lang=ALL-UNNAMED --add-opens  
java.base/java.lang.reflect=ALL-UNNAMED
```

## 7.5.6 回滚测试

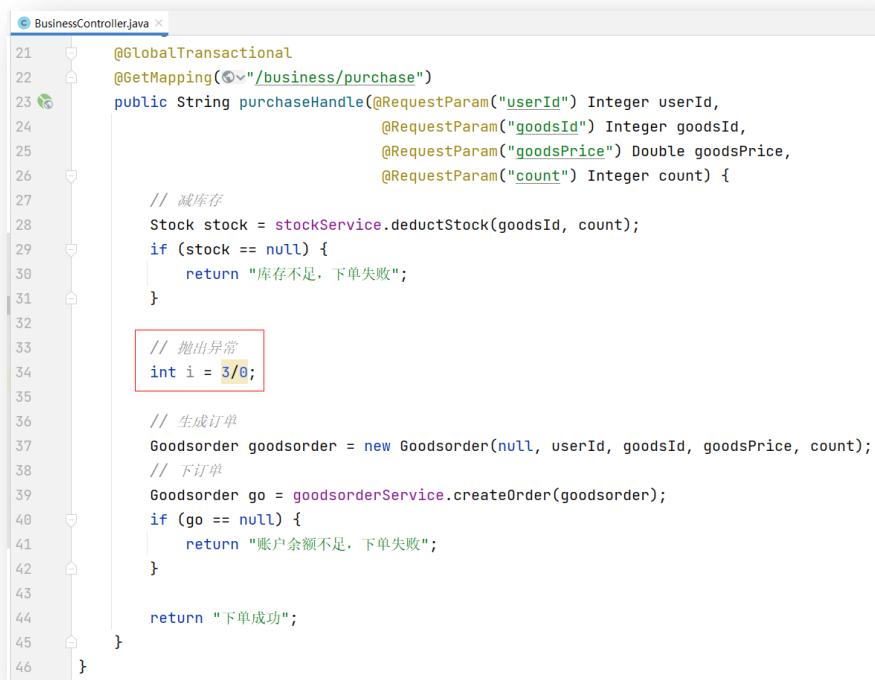
### A、正常测试

提交请求之前,首先查看 mysql1、mysql2 与 mysql3 中 test 数据库中的 stock 表、goodsorder 表与 account 表中的原始数据。然后提交如下请求后,再查看这三个表中的相关字段值,发现 stock 中的 total 减少了 4, account 表中的 balance 减少了 12, goodsorder 表中新增了一条订单记录。说明在正常情况下是没有问题的。



### B、异常测试 1

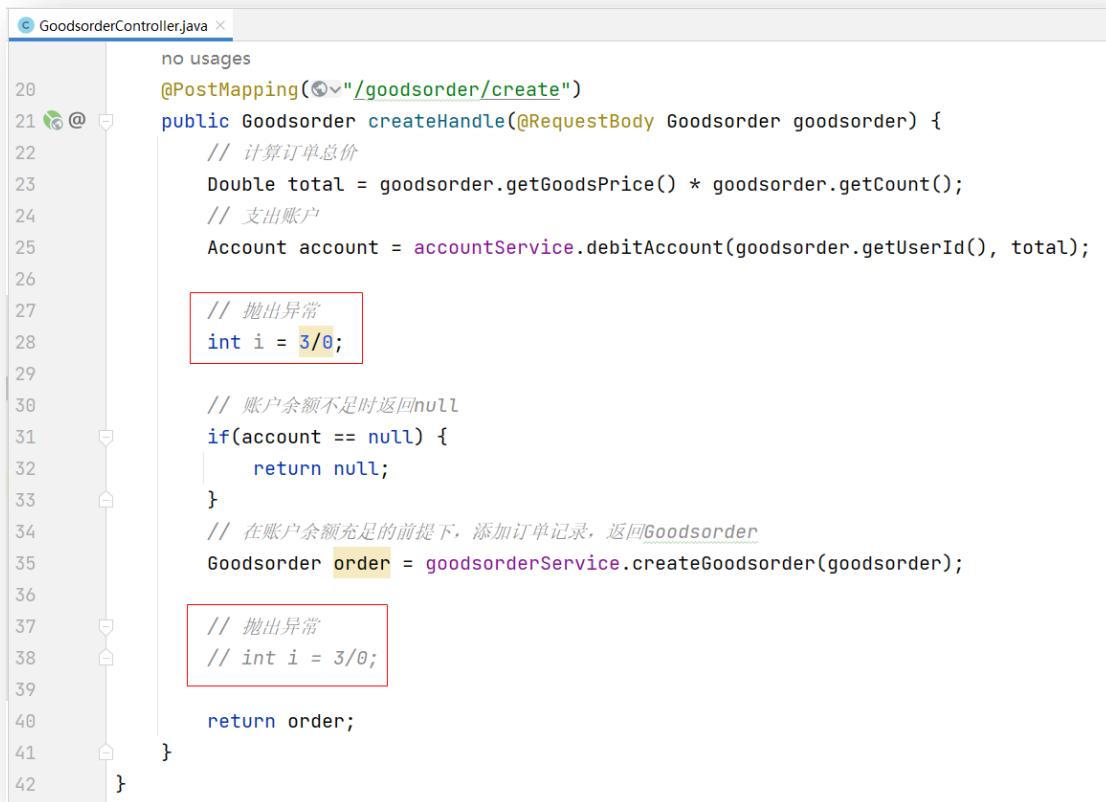
修改 BusinessController 类的 `purchaseHandle()` 方法,在减过库存后,生成订单前抛出一个异常,仍然提交如上请求。页面会报出 500 的错误,但查看 stock 表会发现,其 total 并没有减少。说明全局事务已经起作用了。



```
BusinessController.java
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6
7  package com.bnpu.controller;
8
9  import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.RequestParam;
11 import org.springframework.web.bind.annotation.RestController;
12
13 /**
14  * @author Administrator
15  */
16 @RestController
17 @RequestMapping("/business/purchase")
18 public class BusinessController {
19     @GlobalTransactional
20     @GetMapping("/business/purchase")
21     public String purchaseHandle(@RequestParam("userId") Integer userId,
22             @RequestParam("goodsId") Integer goodsId,
23             @RequestParam("goodsPrice") Double goodsPrice,
24             @RequestParam("count") Integer count) {
25         // 减库存
26         Stock stock = stockService.deductStock(goodsId, count);
27         if (stock == null) {
28             return "库存不足, 下单失败";
29         }
30
31         // 抛出异常
32         int i = 3/0;
33
34         // 生成订单
35         Goodsorder goodsorder = new Goodsorder(null, userId, goodsId, goodsPrice, count);
36         // 下订单
37         Goodsorder go = goodsorderService.createOrder(goodsorder);
38         if (go == null) {
39             return "账户余额不足, 下单失败";
40         }
41
42         return "下单成功";
43     }
44 }
45 }
```

## C、异常测试 2

删除 BusinessServiceImpl 类方法中分母为 0 的语句，修改 GoodsorderController 类的 createHandle() 方法，在支出账户余额后，订单记录插入前抛出一个异常；在支出账户余额后，订单记录插入后抛出一个异常。仍然提交如上请求。页面会报出 500 的错误，但查看三个表会发现，它们并没有变化。说明全局事务已经起作用了。



```

 20     no usages
 21     @PostMapping("/goodsorder/create")
 22     public Goodsorder createHandle(@RequestBody Goodsorder goodsorder) {
 23         // 计算订单总价
 24         Double total = goodsorder.getGoodsPrice() * goodsorder.getCount();
 25         // 支出账户
 26         Account account = accountService.debitAccount(goodsorder.getUserId(), total);
 27
 28         // 抛出异常
 29         int i = 3/0;
 30
 31         // 账户余额不足时返回null
 32         if(account == null) {
 33             return null;
 34         }
 35         // 在账户余额充足的前提下，添加订单记录，返回Goodsorder
 36         Goodsorder order = goodsorderService.createGoodsorder(goodsorder);
 37
 38         // 抛出异常
 39         // int i = 3/0;
 40
 41         return order;
 42     }

```

## 7.6 AT 模式工作过程详解

### 7.6.1 整体机制

AT 模式是 2PC 两阶段提交协议的演变：

- 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。
- 二阶段：
  - ◆ 提交异步化，非常快速地完成。
  - ◆ 回滚通过一阶段的回滚日志进行反向补偿。

## 7.6.2 工作机制

下面以一个例子来分析整个 AT 模式的工作机制。

假设有一个业务表 `product(id, name, since)`, 现在一个 AT 分支事务的业务逻辑是, 将名称为 TXC 的产品更名为 GTS, 即执行的 SQL 为

```
update product set name = 'GTS' where name = 'TXC';
```

### (1) 一阶段过程

- 解析 SQL: 得到 SQL 的类型 (UPDATE), 表 (`product`), 条件 (`where name = 'TXC'`) 等相关的信息。
- 查询要写入到 `undo_log` 表的 `rollback_info` 字段中前镜像 `beforeImage` 的值: 根据解析得到的条件信息, 生成查询语句, 定位数据。  
`select id, name, since from product where name = 'TXC';`
- 执行业务 SQL: 更新这条记录的 `name` 为 'GTS'。  
`update product set name = 'GTS' where name = 'TXC';`
- 查询要写入到 `undo_log` 表的 `rollback_info` 字段中前镜像 `afterImage` 的值: 根据前镜像的结果, 通过主键定位数据。  
`select id, name, since from product where id = 1;`
- 插入回滚日志: 把前后镜像数据以及业务 SQL 相关的信息组成一条回滚日志记录, 插入到 `UNDO_LOG` 表中。
- 提交前, 向 TC 注册分支: 申请 `product` 表中主键值为 1 的记录的全局锁。
- 本地事务提交: 申请到全局锁后, 在本地数据库中将业务数据的更新和前面步骤中生成的 `UNDO LOG` 数据一并提交。
- 将本地事务提交的结果上报给 TC。

### (2) 二阶段过程

二阶段根据接收到的 TC 发送来的不同命令, 可以分为两类过程:

#### A、回滚过程

当接收到 TC 发送给各个分支的回滚命令时, 将开启一个本地事务, 执行如下操作:

- 通过 XID 和 Branch ID 查找到相应的 UNDO LOG 记录。
- 数据校验: 拿 UNDO LOG 中的后镜像数据与当前业务数据进行比较。如果不同, 说明数据被当前全局事务之外的操作进行了修改。这种情况实际是不允许的异常情况, 需要运维进行手工处理。
- 根据 UNDO LOG 中的前镜像和业务 SQL 的相关信息生成并执行回滚的语句:  
`update product set name = 'TXC' where id = 1;`
- 提交本地事务。并把本地事务的执行结果 (即分支事务回滚的结果) 上报给 TC。

## B、提交过程

当接收到 TC 发送给各个分支的提交命令时，会执行如下操作：

- 将提交命令放入一个异步任务的队列中，并马上返回提交成功的结果给 TC。
- 异步任务将异步和批量地删除相应 UNDO LOG 记录。

### 7.6.3 写隔离

如果有多个用户同时进行购买操作，即同时有多个用户同时对库存表及订单表进行操作，那么是否会出现数据混乱的情况呢？当然不会，因为 AT 模式提供了“写隔离”。

#### (1) 前提条件

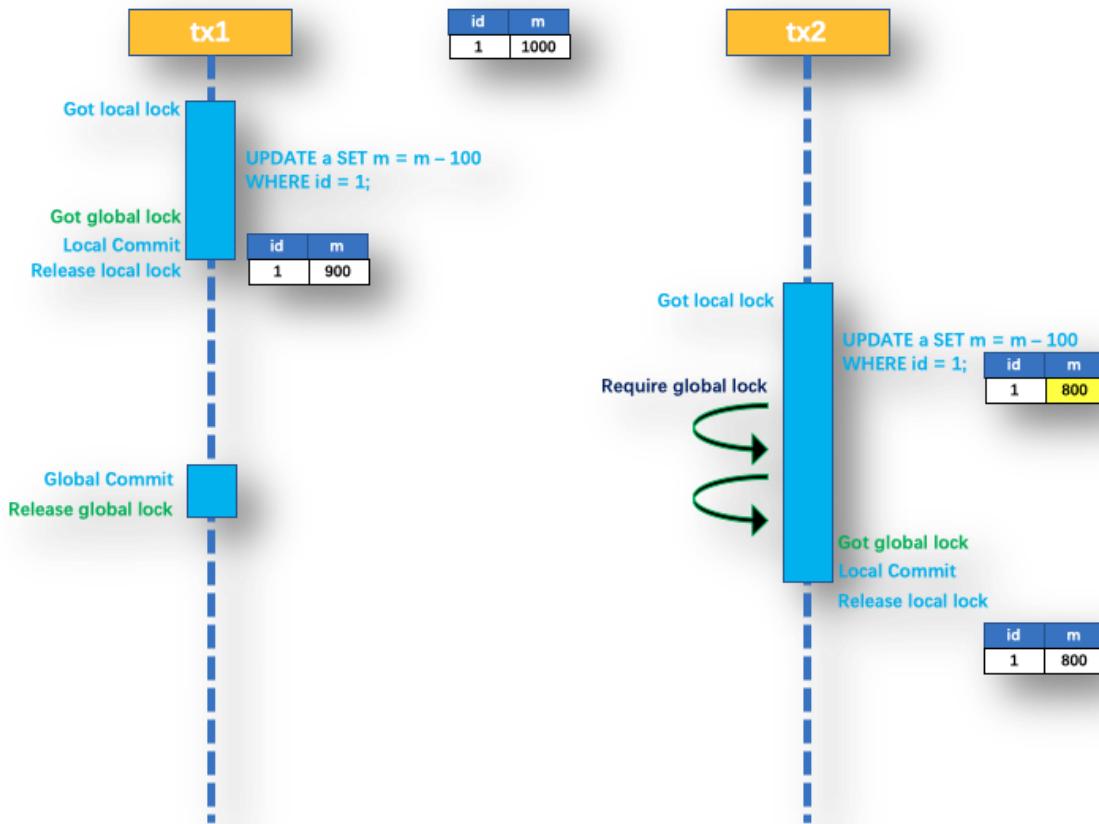
首先要清楚以下几条前提条件：

- 一阶段本地事务写操作前，需要确保先拿到本地锁。拿不到本地锁，不能执行本地数据库写操作。
- 一阶段本地事务提交前，需要确保先拿到全局锁。拿不到全局锁，不能提交本地事务。
- 拿全局锁的尝试被限制在一定范围内(10 次)，超出范围将放弃尝试，然后回滚本地事务，释放本地锁。

#### (2) 二阶段提交时的写隔离

以一个示例来说明：

两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

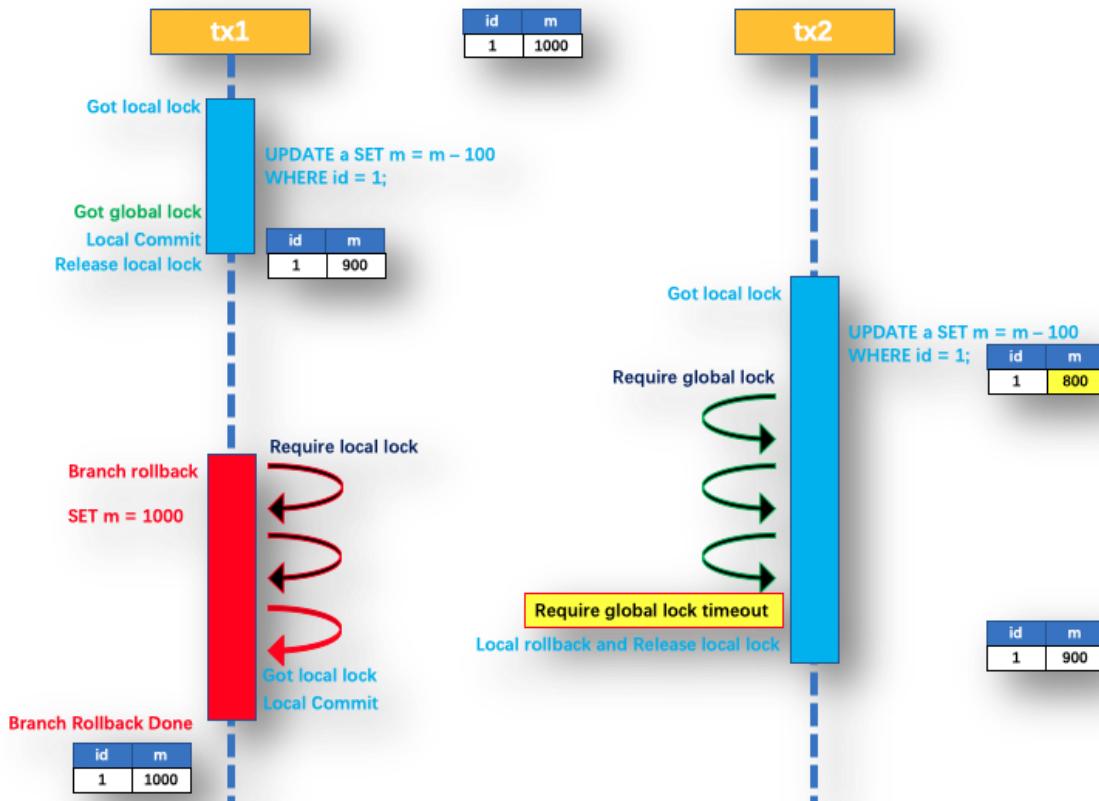


tx1 先开始，开启本地事务，拿到本地锁，更新操作  $m = 1000 - 100 = 900$ 。本地事务提交前，先拿到该记录的全局锁，本地提交释放本地锁。

tx2 后开始，开启本地事务，拿到本地锁，更新操作  $m = 900 - 100 = 800$ 。本地事务提交前，尝试拿该记录的全局锁。但在 tx1 全局提交前，该记录的全局锁被 tx1 持有，tx2 需要重试，等待 tx1 释放全局锁。

tx1 二阶段全局提交，释放 全局锁 。tx2 拿到 全局锁 提交本地事务。

### (3) 二阶段回滚时的写隔离



如果 tx1 的二阶段全局回滚，则 tx1 需要重新获取该数据的本地锁，进行反向补偿的更新操作，实现分支的回滚。

此时，如果 tx2 仍在等待该数据的全局锁，同时持有本地锁，因此 tx1 会由于没有本地锁而使得分支回滚失败。此时的系统实际发生了“死锁”：tx1 持有全局锁，但申请本地锁；tx2 持有本地锁，但申请全局锁。

tx1 分支的回滚会一直重试，直到 tx2 的对全局锁的等待超时，放弃全局锁。由于没有全局锁，所以本地事务不能提交，故 tx2 会回滚本地事务，释放本地锁。

tx1 获取到本地锁，同时其也拥有全局锁，所以 tx1 的分支回滚最终成功。由于整个过程全局锁在 tx1 结束前一直是被 tx1 持有的，所以不会发生脏写问题。

#### 7.6.4 读隔离

##### (1) 前提条件

首先要清楚以下几条前提条件：

- 普通读操作(select...)的执行需要获取到该记录的本地读锁。
- for update 的读操作(select... for update)的执行需要同时获取该记录的本地读锁与全局

读锁。

- 读锁是共享锁。一条记录上可同时添加多把读锁。
- 一条记录上若上了写锁，则将不能再上读锁。
- 一条记录上只要上了读锁，则不可能再上写锁。

## (2) Seata 读隔离

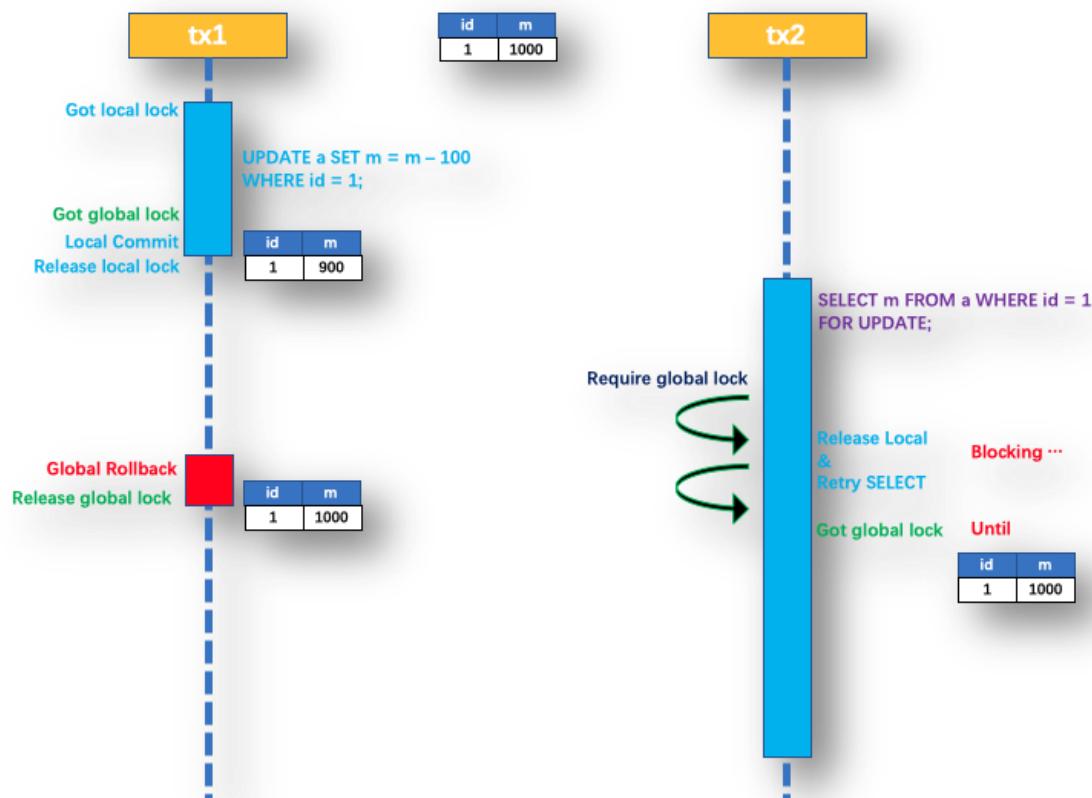
事务隔离级别(由低到高): 读未提交 -> 读已提交 -> 可重复读 -> 串行化。

Seata (AT 模式) 的默认全局隔离级别是 读未提交 (Read Uncommitted), 所以, 对于本地事务隔离级别在其之上的数据库来说, 在使用时就需要注意了。

出于总体性能上的考虑, Seata 目前并没有对所有 select 语句都进行读隔离, 仅针对 for update 的 select 语句。for update 的 select 语句的执行会申请全局读锁, 如果其它事务持有全局写锁, 则立即回滚 for update 的 select 语句的本地执行, 释放本地锁, 然后再重试执行 for update 的 select 语句。这个过程中, 查询是被 block 住的, 直到全局锁拿到, 即读取的相关数据是已提交的, 才返回。

所以, 若某数据库的隔离级别为读已提交, 此时就需要在 select 语句后添加 for update, 以告知 Seata 要对该读操作进行有效隔离。

## (3) 举例



在 tx2 执行 select 时，由于其没有申请到全局锁，所以其会一直重试，直到 tx1 执行完回滚操作，释放了全局锁，此时 tx2 才能读取到数据 1000。若没有进行读隔离，则 tx2 在执行 select 时会直接读取到 tx1 未提交的数据 900。当 tx1 将其回滚为 1000 后，tx2 读取到的这个 900 就成为了脏数据。

## 7.7 TCC 模式下的 Seata Client

### 7.7.1 删除 undo\_log 表

TCC 模式是手动实现全局回滚与提交的 AT 模式。所以，其不再需要通过 undo\_log 表来实现全局回滚。所以所有业务数据库中的 undo\_log 表均可删除了。当然，不删除也无所谓。

### 7.7.2 创建工程

为了保留下原始代码，我们这里复制 07-at-stock-8081、07-at-goodsorder-8082、07-at-account-8083 与 07-at-business-8080 四个工程，分别重新命名为 07-tcc-stock-8081、07-tcc-goodsorder-8082、07-tcc-account-8083 与 07-tcc-business-8080，并将其改造为 TCC 模式的 Seata-Client。

其实，只需对 07-tcc-stock-8081、07-tcc-goodsorder-8082 和 07-tcc-account-8083 这三个工程进行改造，且改造思路相同。

### 7.7.3 修改 07-tcc-stock-8081

#### (1) 修改 StockService

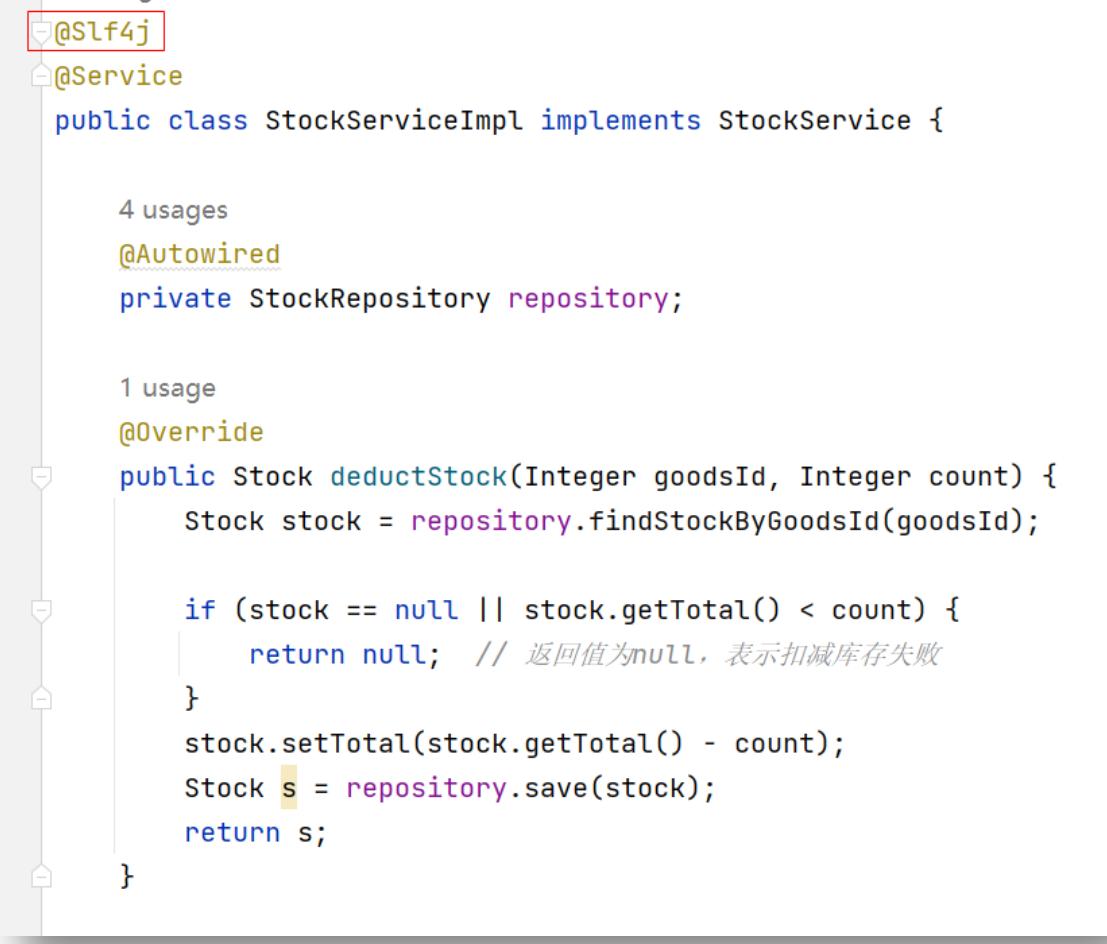
```
@LocalTCC
public interface StockService {

    1 usage 1 implementation
    @TwoPhaseBusinessAction(name = "deductStock", // tcc bean name
                            commitMethod = "tccCommit",
                            rollbackMethod = "tccRollback")
    Stock deductStock(@BusinessActionContextParameter("goodsId") Integer goodsId,
                      @BusinessActionContextParameter("count") Integer count);

    // 手动提交方法
    no usages 1 implementation
    Boolean tccCommit(BusinessActionContext context);

    // 手动回滚方法
    no usages 1 implementation
    Boolean tccRollback(BusinessActionContext context);
}
```

## (2) 修改 StockServiceImpl



```
@Slf4j
@Service
public class StockServiceImpl implements StockService {

    4 usages
    @Autowired
    private StockRepository repository;

    1 usage
    @Override
    public Stock deductStock(Integer goodsId, Integer count) {
        Stock stock = repository.findStockByGoodsId(goodsId);

        if (stock == null || stock.getTotal() < count) {
            return null; // 返回值为null，表示扣减库存失败
        }
        stock.setTotal(stock.getTotal() - count);
        Stock s = repository.save(stock);
        return s;
    }
}
```

```
@Override
public Boolean tccCommit(BusinessActionContext context) {
    // 全局提交无需执行其它操作，直接返回true即可
    log.info("库存数据变更已确认提交");
    return true;
}

no usages

@Override
public Boolean tccRollback(BusinessActionContext context) {
    log.info("库存数据变更发生回滚");
    Integer goodsId = (Integer) context.getActionContext("goodsId");
    Integer count = (Integer) context.getActionContext("count");

    Stock stock = repository.findStockByGoodsId(goodsId);
    // 将减去的库存再加回来
    stock.setTotal(stock.getTotal() + count);
    repository.save(stock);
    return true;
}
```

## 7.7.4 修改 07-tcc-account-8083

### (1) 修改 AccountService

```
@LocalTCC
public interface AccountService {

    // 支出账户
    1 usage 1 implementation
    @TwoPhaseBusinessAction(name = "debitAccount", // tcc bean name
        commitMethod = "tccCommit",
        rollbackMethod = "tccRollback")
    Account debitAccount(@BusinessActionContextParameter("userId") Integer userId,
        @BusinessActionContextParameter("amount") Double amount);

    // 手动提交方法
    no usages 1 implementation
    Boolean tccCommit(BusinessActionContext context);

    // 手动回滚方法
    no usages 1 implementation
    Boolean tccRollback(BusinessActionContext context);
}
```

## (2) 修改 AccountServiceImpl

```
└─ @Slf4j
└─ @Service
public class AccountServiceImpl implements AccountService {

    4 usages
    @Autowired
    private AccountRepository repository;

    1 usage
    @Override
    public Account debitAccount(Integer userId, Double amount) {
        Account account = repository.findAccountByUserId(userId);
        if (account == null || account.getBalance() < amount) {
            // 返回null, 表示支出账户失败
            return null;
        }
        account.setBalance(account.getBalance() - amount);
        return repository.save(account);
    }
}
```

```
    }

    @Override
    public Boolean tccCommit(BusinessActionContext context) {
        log.info("账户数据变更已确认提交");
        return true;
    }

    no usages

    @Override
    public Boolean tccRollback(BusinessActionContext context) {
        log.info("账户数据变更发生回滚");
        Integer userId = (Integer) context.getActionContext("userId");
        // 对于Double数据，其是以BigDecimal类型写context的
        BigDecimal amount = (BigDecimal) context.getActionContext("amount");

        Account account = repository.findAccountByUserId(userId);
        // // 将减去的账户余额再加回来
        account.setBalance(account.getBalance() + amount.doubleValue());
        repository.save(account);
        System.out.println("AccountServiceImpl 发生回滚");
        return true;
    }
}
```

## 7.7.5 修改 07-tcc-goodsorder-8082

### (1) 修改 GoodsorderServiceLocal

```
@LocalTCC
public interface GoodsorderServiceLocal {

    1 usage  1 implementation
    @TwoPhaseBusinessAction(name = "createGoodsorder", // tcc bean name
        commitMethod = "tccCommit",
        rollbackMethod = "tccRollback")
    Goodsorder createGoodsorder(@BusinessActionContextParameter("goodsorder") Goodsorder goodsorder);

    // 手动提交方法
    no usages  1 implementation
    Boolean tccCommit(BusinessActionContext context);

    // 手动回滚方法
    no usages  1 implementation
    Boolean tccRollback(BusinessActionContext context);
}
```

### (2) 修改 GoodsorderServiceImpl

通过成员变量的方式保存获取到的“新增订单记录的 id”。

```
@Slf4j
@Service
public class GoodsorderServiceImpl implements GoodsorderServiceLocal {
    2 usages
    @Autowired
    private GoodsorderRepository repository;
    2 usages
    private Integer goodsorderId;

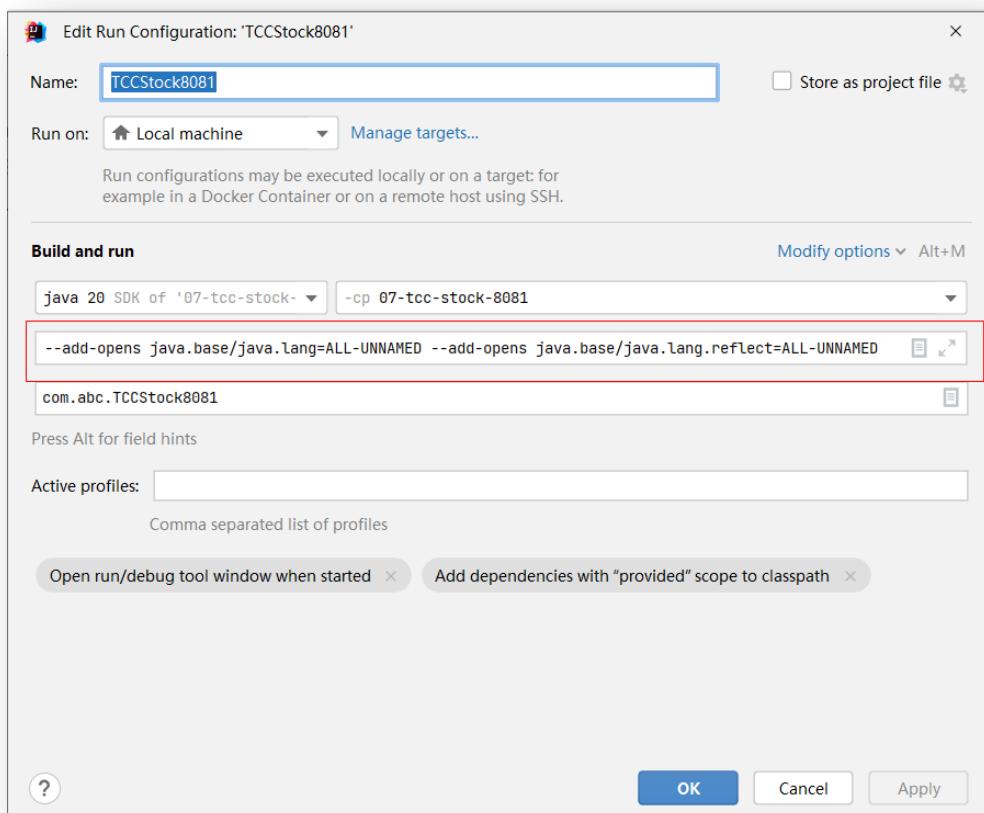
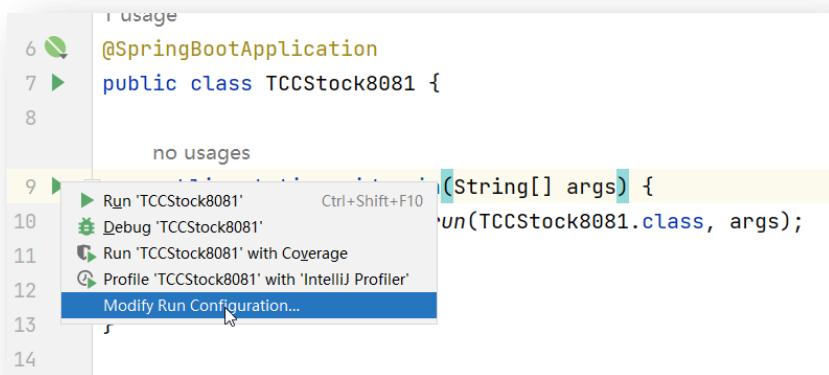
    1 usage
    @Override
    public Goodsorder createGoodsorder(Goodsorder goodsorder) {
        Goodsorder go = repository.save(goodsorder);
        // 获取新增数据行的id
        this.goodsorderId = go.getId();
        return go;
    }
}
```

```
@Override
public Boolean tccCommit(BusinessActionContext context) {
    log.info("订单数据已确认提交");
    return true;
}

no usages
@Override
public Boolean tccRollback(BusinessActionContext context) {
    log.info("订单数据发生回滚");
    repository.deleteById(this.goodsorderId);
    return true;
}
}
```

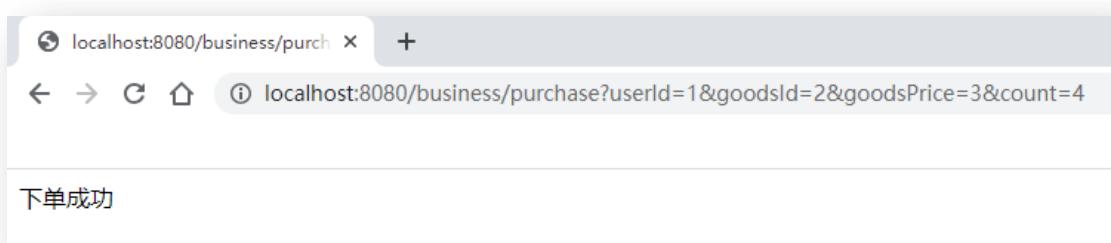
## 7.7.6 启动应用

对于 07-tcc-stock-8081、07-tcc-goodsorder-8082 和 07-tcc-account-8083 这三个工程的启动，仍需添加以下启动参数。



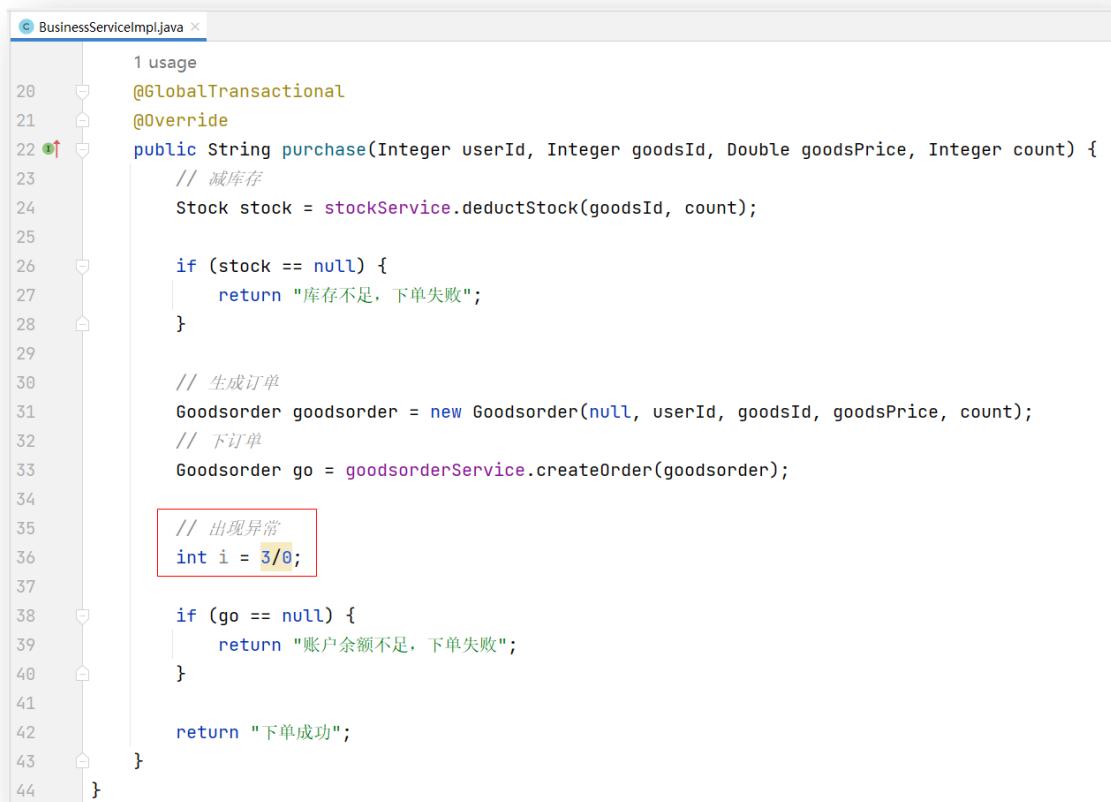
```
--add-opens java.base/java.lang=ALL-UNNAMED --add-opens  
java.base/java.lang.reflect=ALL-UNNAMED
```

### 7.7.7 测试



在浏览器提交以上请求后会发现，库存表中的库存量减少了 4，账户表中的余额减少了 12，订单表中新增了一条订单记录。

然后在 BusinessServiceImpl 类中完成订单插入后的语句后添加一个可以抛出异常的语句。



```
BusinessServiceImpl.java
1 usage
20     * @GlobalTransactional
21     * @Override
22     public String purchase(Integer userId, Integer goodsId, Double goodsPrice, Integer count) {
23         // 减库存
24         Stock stock = stockService.deductStock(goodsId, count);
25
26         if (stock == null) {
27             return "库存不足，下单失败";
28         }
29
30         // 生成订单
31         Goodsorder goodsorder = new Goodsorder(null, userId, goodsId, goodsPrice, count);
32         // 下订单
33         Goodsorder go = goodsorderService.createOrder(goodsorder);
34
35         // 出现异常
36         int i = 3/0;
37
38         if (go == null) {
39             return "账户余额不足，下单失败";
40         }
41
42         return "下单成功";
43     }
44 }
```

在浏览器再次提交以上请求，会发现这三张表中的数据没有发生变化。说明 TCC 分布式事务生效。

## 第8章 调用链跟踪 SkyWalking

随着分布式系统规模的越来越大，各微服务间的调用关系也变得越来越复杂。一般情况下，一个由客户端发起的请求在后端系统中会经过许多不同的微服务调用才能完成最终处理，而这些调用过程形成了一个复杂的分布式服务调用链路。

那么也就带来了一系列问题：怎样快速发现并定位问题？怎样判断故障影响范围？各部分调用链路性能是怎样的？对于这些问题，通过分布式服务跟踪系统可以解决。

生产环境下，Spring Cloud Alibaba 经常会使用 SkyWalking 作为调用链跟踪系统。

国内的分布式跟踪系统：Hydra(京东)、CAT(大众点评)、Watchman(新浪)、Micoscope(唯品会)及 Eagleeye(淘宝，未开源)。国际上使用较多的是 Zipkin(Twitter)及 Skywalking。

### 8.1 概述

#### 8.1.1 SkyWalking 简介

SkyWalking 是由国内开源爱好者吴晟开源并提交到 Apache 孵化器的产品，现在已是 Apache 的顶级项目。其是一个开源的 APM（Application Performance Management，应用性能管理系统）和 OAP 平台（Observability Analysis Platform，可观测性分析平台）。官网地址 <https://skywalking.apache.org/> 。

其是通过在被监测应用中插入探针，以无侵入方式自动收集所需指标，并自动推送到 OAP 系统平台。OAP 会将收集到的数据存储到指定的存储介质 Storage。UI 系统通过调用 OAP 提供的接口，可以实现对相应数据的查询。

#### 8.1.2 系统架构

SkyWalking 系统整体由四部分构成：

- **Agent**: 探针，无侵入收集，是被插入到被监测应用中的一个进程。其会将收集到的监控指标自动推送给 OAP 系统。
- **OAP**: Observability Analysis Platform，可观测性分析平台，其包含一个收集器 Collector，能够将来自于 Agent 的数据收集并存储到相应的存储介质 Storage。
- **Storage**: 数据中心。用于存储由 OAP 系统收集到的链路数据，支持 H2、MySQL、ElasticSearch 等。默认使用的是 H2（测试使用），推荐使用 ElasticSearch（生产使用）。
- **UI**: 一个独立运行的可视化 Web 平台，其通过调用 OAP 系统提供的接口，可以对存储在 Storage 中的数据进行多维度查询。

#### 8.1.3 trace 与 span

- **trace (轨迹)**：跟踪单元是从客户端所发起的请求抵达被跟踪系统的边界开始，到被跟踪系统向客户返回响应为止的过程，这个过程称为一个 trace。

- **span(跨度):** 每个 trace 中会调用若干个服务，为了记录调用了哪些服务，以及每次调用所消耗的时间等信息，在每次调用服务时，埋入一个调用记录，这样两个调用记录之间的区域称为一个 span。一个 trace 由若干个有序的 span 组成。
- **segment(片断):** 一个 trace 的一个片断，可以由多个 span 组成。

为了唯一的标识 trace、span 与 segment，跟踪系统一般会为每个 trace、span 与 segment 都指定了一个唯一标识，即 traceId、spanId 与 segmentId。

## 8.2 ES 的安装与启动

### 8.2.1 Linux 系统环境准备

#### (1) 克隆并配置主机

克隆一台干净的主机，并修改配置。

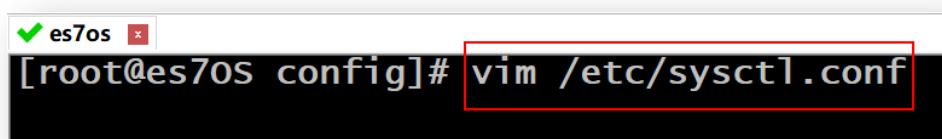
- 修改主机名: /etc/hostname
- 修改网络配置: /etc/sysconfig/network-scripts/ifcfg-ens33

#### (2) 修改虚拟内存空间大小

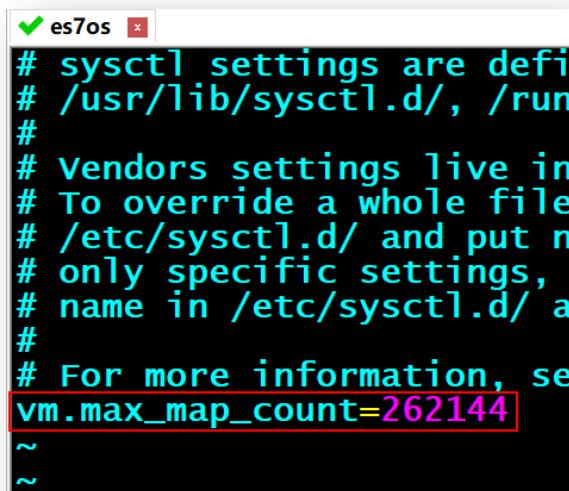
系统变量 vm.max\_map\_count 是系统中分配的虚拟内存空间大小。

#### A、永久修改 vim /etc/sysctl.conf

在/etc/sysctl.conf 文件最后插入如下一行内容。这种修改方式是永久修改。



```
[root@es7os config]# vim /etc/sysctl.conf
```



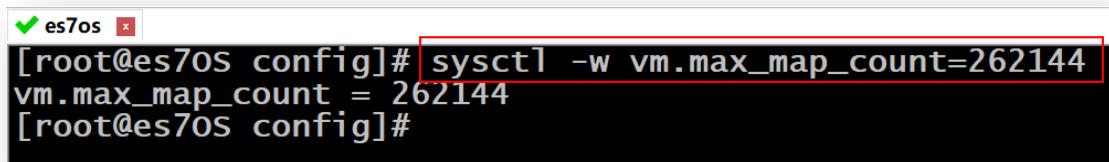
```
# sysctl settings are defined
# /usr/lib/sysctl.d/, /run/
#
# Vendors settings live in
# To override a whole file
# /etc/sysctl.d/ and put non-
# only specific settings,
# name in /etc/sysctl.d/ a
#
# For more information, see
vm.max_map_count=262144
~
```

现将其值调整为了 256K。

## B、临时修改 sysctl -w vm.max\_map\_count=262144

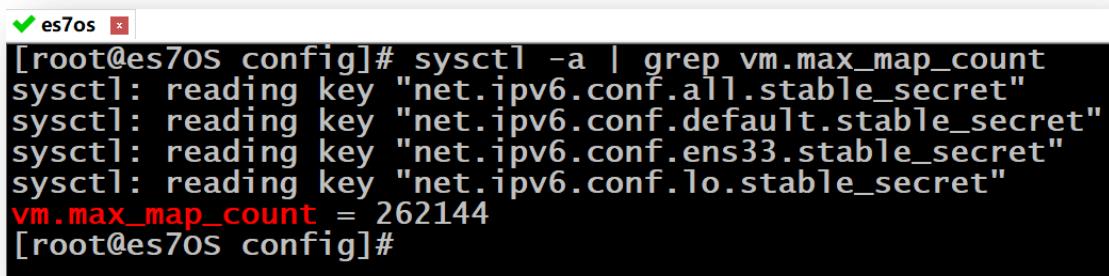
此时再查看该值，发现其并没有变化。是因为永久修改的生效需要重启系统。不过，也可以通过以下命令临时修改其值。

一般情况下我们会永久修改与临时修改均进行设置，这样的话，本次不用重启，以后重启后其值也会永久改变，一举两得。



```
[root@es7OS config]# sysctl -w vm.max_map_count=262144
vm.max_map_count = 262144
[root@es7OS config]#
```

再次查看，发现其值已经修改过了。

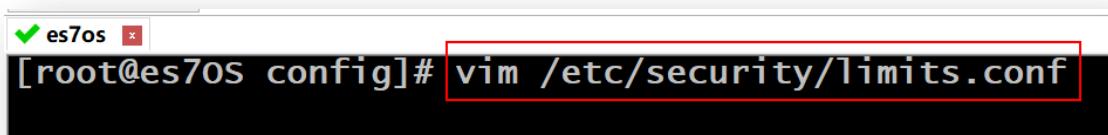


```
[root@es7OS config]# sysctl -a | grep vm.max_map_count
sysctl: reading key "net.ipv6.conf.all.stable_secret"
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.ens33.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
vm.max_map_count = 262144
[root@es7OS config]#
```

### (3) 修改最大文件描述符数量及用户最大线程数

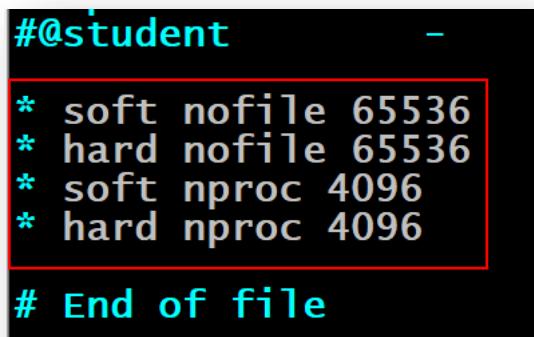
#### A、修改 vim /etc/security/limits.conf

在/etc/security/limits.conf 文件最后插入如下 4 行内容。



```
[root@es7OS config]# vim /etc/security/limits.conf
```

注：星号(\*)表示对所有用户。



```
#@student -  
* soft nofile 65536  
* hard nofile 65536  
* soft nproc 4096  
* hard nproc 4096  
  
# End of file
```

#### B、soft 与 hard

soft：柔和边界设定。可以超过该设定值，但超过后会有警告

hard：严格边界设定。不允许超过该设定的值

#### C、nofile 与 nproc

nofile：每个进程可打开的文件数的最大数量

nproc：每个用户可创建的进程数的最大数量

### (4) 创建用户与密码

ES 不能使用 root 用户进行启动，所以需要创建新的用户，并为该用户指定登录密码。

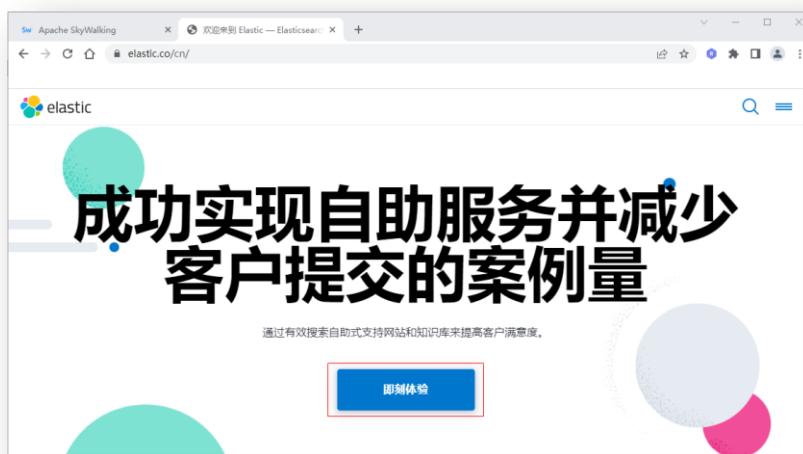
下面的两行命令创建了用户 zhangsan，指定的登录密码为 111111。

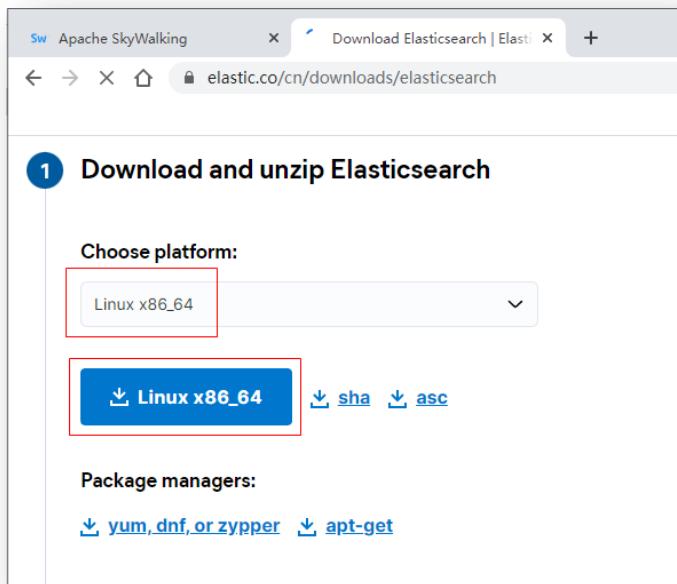
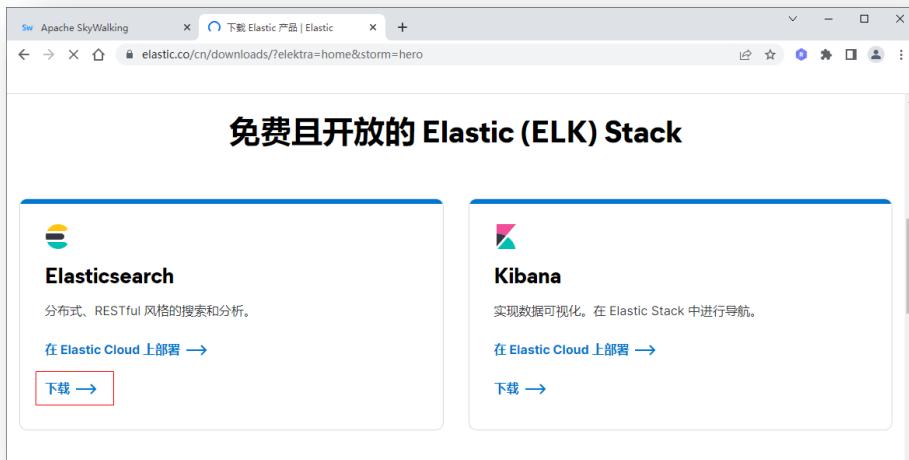
```
✓ es7os # [root@es7os bin]# useradd zhangsan  
[root@es7os bin]# echo "111111" | passwd zhangsan --stdin  
更改用户 zhangsan 的密码。  
passwd: 所有的身份验证令牌已经成功更新。  
[root@es7os bin]#
```

## 8.2.2 ES 的安装与配置

### (1) 下载 ES 安装包

从 ES 官网 <https://elastic.co/> 下载针对 Linux 系统的安装包。

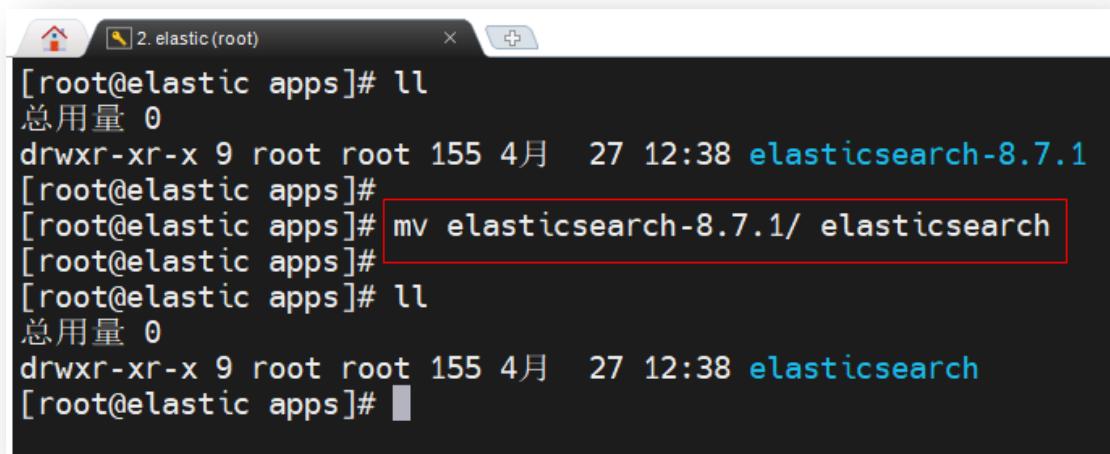




## (2) 解压

将安装包上传到/opt/tools 目录，然后解压到/opt/apps 目录下，并重命名。

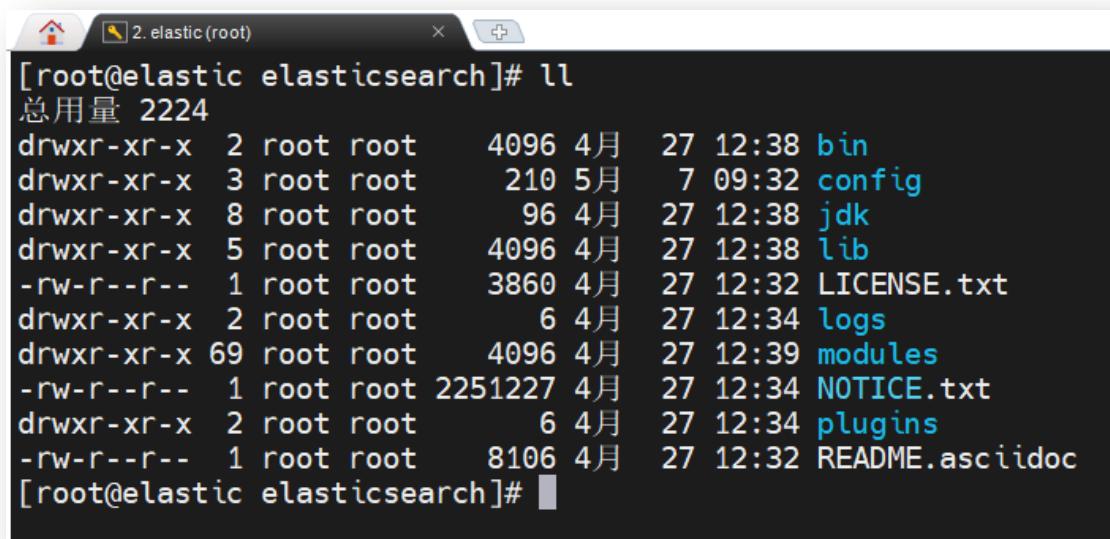
```
[# tar -zxf elasticsearch-8.7.1-linux-x86_64.tar.gz -C /opt/apps/ ]
```



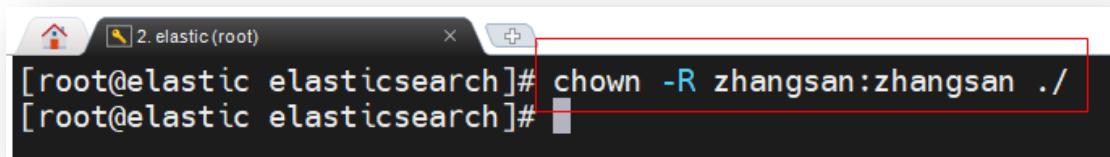
```
[root@elastic apps]# ll
总用量 0
drwxr-xr-x 9 root root 155 4月 27 12:38 elasticsearch-8.7.1
[root@elastic apps]#
[root@elastic apps]# mv elasticsearch-8.7.1/ elasticsearch
[root@elastic apps]#
[root@elastic apps]# ll
总用量 0
drwxr-xr-x 9 root root 155 4月 27 12:38 elasticsearch
[root@elastic apps]#
```

### (3) 修改所有者

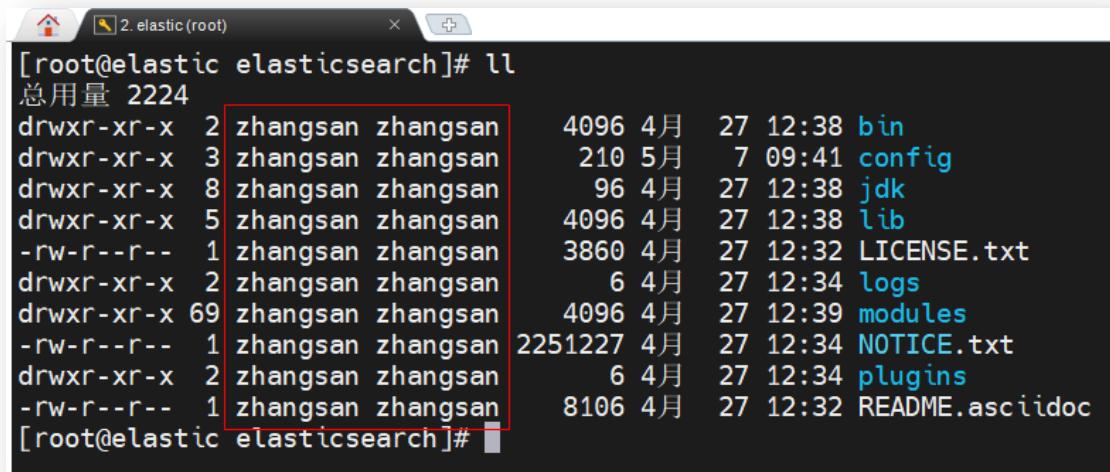
进入 es 目录，可以看到其包含的内容。其所有者全部为 root:root。但 ES 不能使用 root 用户进行启动，所以我们现在要将其修改为新建用户 zhangsan:zhangsan。



```
[root@elastic elasticsearch]# ll
总用量 2224
drwxr-xr-x 2 root root 4096 4月 27 12:38 bin
drwxr-xr-x 3 root root 210 5月 7 09:32 config
drwxr-xr-x 8 root root 96 4月 27 12:38 jdk
drwxr-xr-x 5 root root 4096 4月 27 12:38 lib
-rw-r--r-- 1 root root 3860 4月 27 12:32 LICENSE.txt
drwxr-xr-x 2 root root 6 4月 27 12:34 logs
drwxr-xr-x 69 root root 4096 4月 27 12:39 modules
-rw-r--r-- 1 root root 2251227 4月 27 12:34 NOTICE.txt
drwxr-xr-x 2 root root 6 4月 27 12:34 plugins
-rw-r--r-- 1 root root 8106 4月 27 12:32 README.asciidoc
[root@elastic elasticsearch]#
```



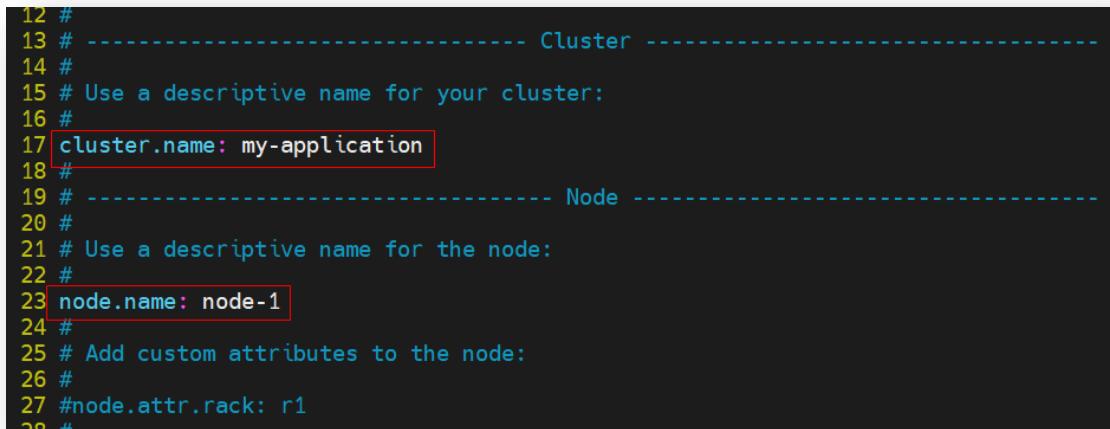
```
[root@elastic elasticsearch]# chown -R zhangsan:zhangsan ./
[root@elastic elasticsearch]#
```



```
[root@elastic elasticsearch]# ll
总用量 2224
drwxr-xr-x  2 zhangsan zhangsan  4096 4月 27 12:38 bin
drwxr-xr-x  3 zhangsan zhangsan  210  5月  7 09:41 config
drwxr-xr-x  8 zhangsan zhangsan   96  4月 27 12:38 jdk
drwxr-xr-x  5 zhangsan zhangsan  4096 4月 27 12:38 lib
-rw-r--r--  1 zhangsan zhangsan 3860  4月 27 12:32 LICENSE.txt
drwxr-xr-x  2 zhangsan zhangsan    6  4月 27 12:34 logs
drwxr-xr-x 69 zhangsan zhangsan 4096 4月 27 12:39 modules
-rw-r--r--  1 zhangsan zhangsan 2251227 4月 27 12:34 NOTICE.txt
drwxr-xr-x  2 zhangsan zhangsan    6  4月 27 12:34 plugins
-rw-r--r--  1 zhangsan zhangsan  8106 4月 27 12:32 README.asciidoc
[root@elastic elasticsearch]#
```

#### (4) 修改 elasticsearch.yml

修改 config/ elasticsearch.yml 文件中的以下几个位置。



```
12 #
13 # ----- Cluster -----
14 #
15 # Use a descriptive name for your cluster:
16 #
17 cluster.name: my-application
18 #
19 # ----- Node -----
20 #
21 # Use a descriptive name for the node:
22 #
23 node.name: node-1
24 #
25 # Add custom attributes to the node:
26 #
27 #node.attr.rack: r1
28 *
```

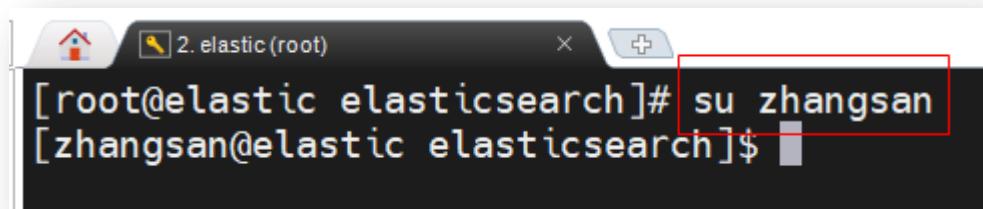
```
50 #
51 # ----- Network -----
52 #
53 # By default Elasticsearch is only accessible on localhost. Set a different
54 # address here to expose this node on the network:
55 #
56 network.host: 0.0.0.0
57 #
58 # By default Elasticsearch listens for HTTP traffic on the first free port it
59 # finds starting at 9200. Set a specific HTTP port here:
60 #
61 http.port: 9200
62 #
63 # For more information, consult the network module documentation.
64 #
65 # ----- Discovery -----
66 #
67 # Pass an initial list of hosts to perform discovery when this node is started:
68 # The default list of hosts is ["127.0.0.1", "[::1]"]
69 #
70 #discovery.seed_hosts: ["host1", "host2"]
71 #
72 # Bootstrap the cluster using an initial set of master-eligible nodes:
73 #
74 cluster.initial_master_nodes: ["node-1"]
75 #
76 # For more information, consult the discovery and cluster formation module documentation.
77 #
```

```
90 #
91 # Enable security features
92 xpack.security.enabled: false
93
94 xpack.security.enrollment.enabled: true
95
96 # Enable encryption for HTTP API client connections, such as Kibana, Logstash, and Agents
97 xpack.security.http.ssl:
98   enabled: false
99   keystore.path: certs/http.p12
100
101 # Enable encryption and mutual authentication between cluster nodes
102 xpack.security.transport.ssl:
103   enabled: true
104   verification_mode: certificate
105   keystore.path: certs/transport.p12
106   truststore.path: certs/transport.p12
107 #----- END SECURITY AUTO CONFIGURATION -----
```

### 8.2.3 ES 的启动

#### (1) 切换用户

将用户切换为 zhangsan。

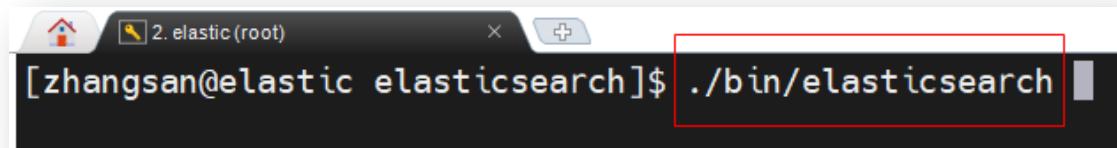


```
[root@elastic elasticsearch]# su zhangsan
[zhangsan@elastic elasticsearch]$
```

A screenshot of a terminal window titled "2. elastic (root)". The command "su zhangsan" is entered and highlighted with a red box. The terminal prompt changes to show the user "zhangsan".

## (2) 启动命令

运行 ES 解压目录的 bin 目录下的 elasticsearch 命令来启动 ES。

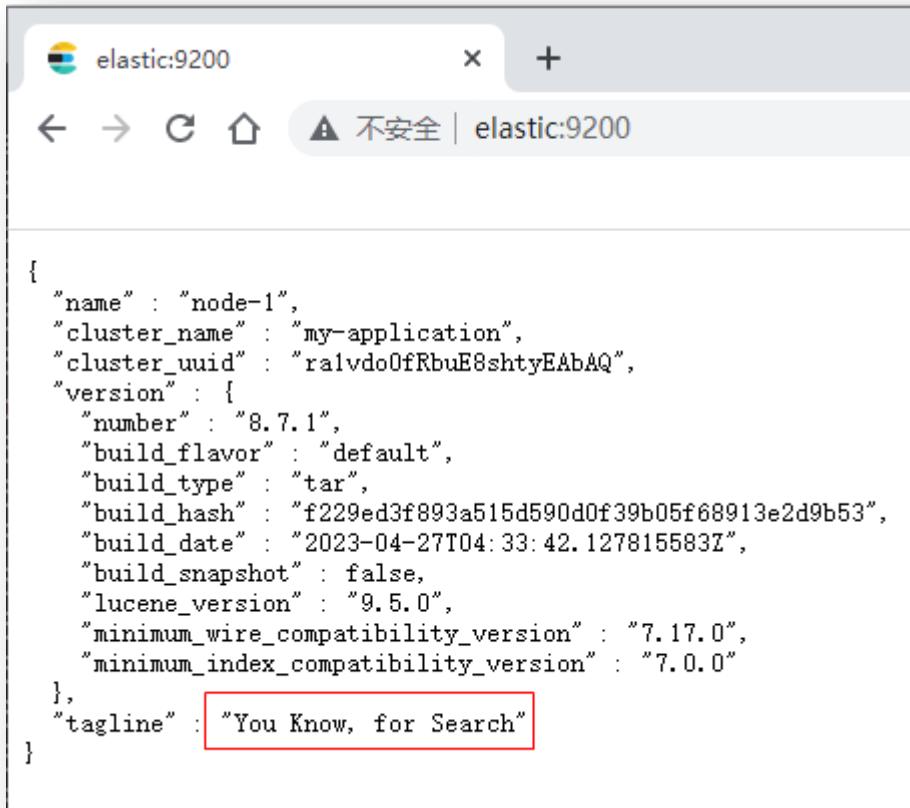


```
[zhangsan@elastic elasticsearch]$ ./bin/elasticsearch
```

A screenshot of a terminal window titled "2. elastic (root)". The command "./bin/elasticsearch" is entered and highlighted with a red box.

## (3) 成功标识

在浏览器中访问 ES 服务器时可以看到以下页面，说明启动成功了。



The screenshot shows a browser window titled "elastic:9200". The address bar indicates an "unsafe" connection to "elastic:9200". The main content area displays the following JSON configuration:

```
{  
  "name": "node-1",  
  "cluster_name": "my-application",  
  "cluster_uuid": "ra1vdo0fRbuE8shtyEAQ",  
  "version": {  
    "number": "8.7.1",  
    "build_flavor": "default",  
    "build_type": "tar",  
    "build_hash": "f229ed3f893a515d590d0f39b05f68913e2d9b53",  
    "build_date": "2023-04-27T04:33:42.127815583Z",  
    "build_snapshot": false,  
    "lucene_version": "9.5.0",  
    "minimum_wire_compatibility_version": "7.17.0",  
    "minimum_index_compatibility_version": "7.0.0"  
  },  
  "tagline": "You Know, for Search"  
}
```

## 8.2.4 ES 的关闭

直接在服务器窗口中 **Ctrl + C** 即可关闭 ES。

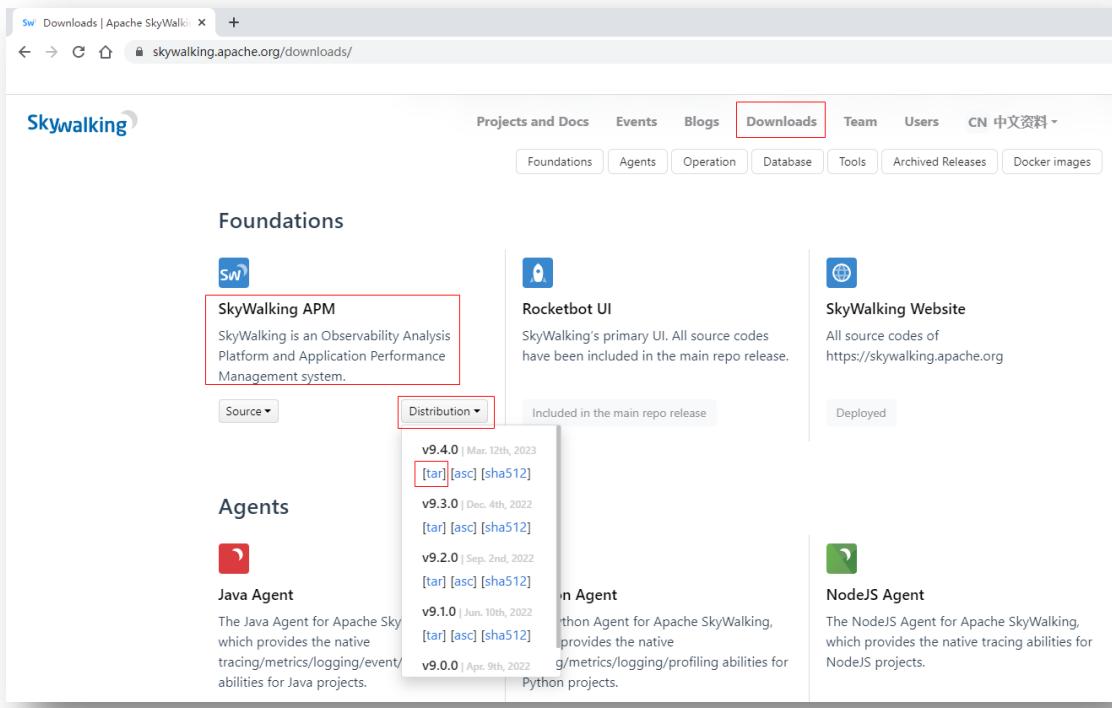
```
d325a51/96] mode [basic] - valid  
[2020-10-24T13:20:36,064][INFO ][o.e.x.s.s.SecurityStatusChangeListener] [node-1]  
[BASIC]; Security is disabled  
^C[2020-10-24T13:22:38,022][INFO ][o.e.x.m.p.NativeController] [node-1] Native cor-  
opped - no new native processes can be started  
[2020-10-24T13:22:38,024][INFO ][o.e.n.Node] [node-1] stopping ...  
[2020-10-24T13:22:38,035][INFO ][o.e.x.w.WatcherService] [node-1] stopping wat-  
tdown initiated  
[2020-10-24T13:22:38,038][INFO ][o.e.x.w.WatcherLifeCycleService] [node-1] watcher  
own  
[2020-10-24T13:22:38,270][INFO ][o.e.n.Node] [node-1] stopped  
[2020-10-24T13:22:38,271][INFO ][o.e.n.Node] [node-1] closing ...  
[2020-10-24T13:22:38,294][INFO ][o.e.n.Node] [node-1] closed  
[zhangsan@es7OS elasticsearch-7.9.2]$
```

## 8.3 SW 的安装与启动

SW 的安装与启动，即 OAP Server 的安装与启动。当前其同时也会将 UI Server 也启动。

### 8.3.1 下载 SW

在官网下载页面下载最新版本的 SkyWalking APM。



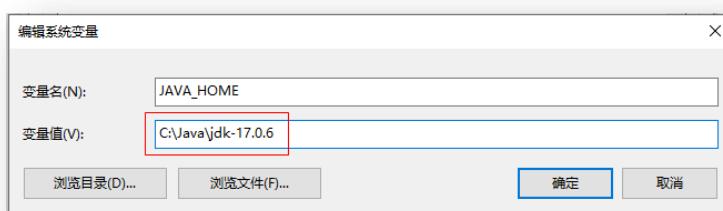
The screenshot shows the Apache SkyWalking Downloads page. The 'Downloads' tab is selected. In the 'Agents' section, the 'Java Agent' is highlighted with a red box. Below it, a dropdown menu labeled 'Distribution' is also highlighted with a red box. Under the 'Java Agent' heading, the 'v9.4.0' version is listed with download links for [tar], [asc], and [sha512].

### 8.3.2 JDK 要求

对于 SkyWalking 9.4.0 版本，其运行环境要求 JDK11 到 JDK17 都是可以的。其它 JDK 版本未经测试。

#### (1) 设置 JAVA\_HOME

将当前系统的 JAVA\_HOME 调整为 JDK17。



### 8.3.3 单机版安装与配置

#### (1) 配置 storage

直接将下载好的 SW 安装包解压，打开其 config 目录下的 application.yml，仅需要修改两处：  
指定 storage 为 elasticsearch，指定 elasticsearch 集群节点主机的 ip。

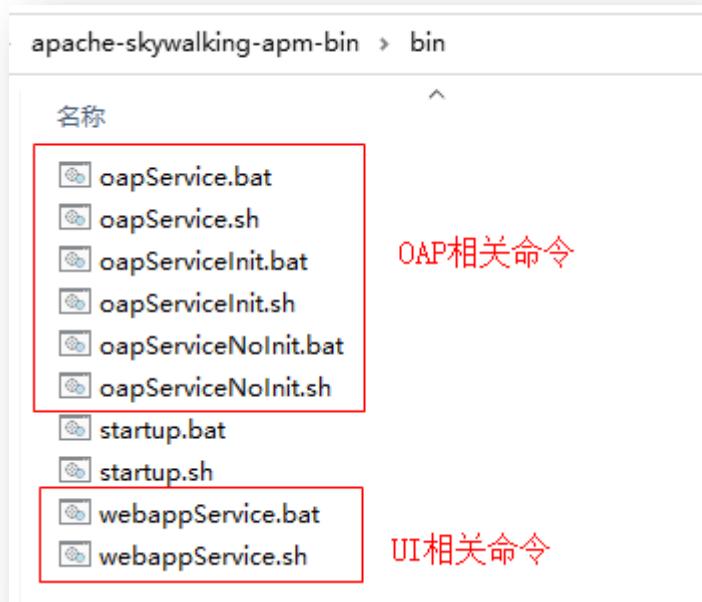
```
130      # Turn it on then automatically grouping endpoint by the given option
131      enableEndpointNameGroupingByOpenapi: ${SW_CORE_ENABLE_ENDPOINT_NAME_}
132 @storage:
133   selector: ${SW_STORAGE:elasticsearch}
134 @elasticsearch:
135   namespace: ${SW_NAMESPACE:""}
136   clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:192.168.192.113:9200}
137   protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
138   connectTimeout: ${SW_STORAGE_ES_CONNECT_TIMEOUT:3000}
139   socketTimeout: ${SW_STORAGE_ES_SOCKET_TIMEOUT:30000}
```

#### (2) 修改 UI 端口 9999

默认情况下，SW 的 UI 应用端口号为 8080，其会与后面应用的端口号冲突。该端口号的修改位置是在 SW 解压目录下的 webapp 目录中的 application.yml 文件中。

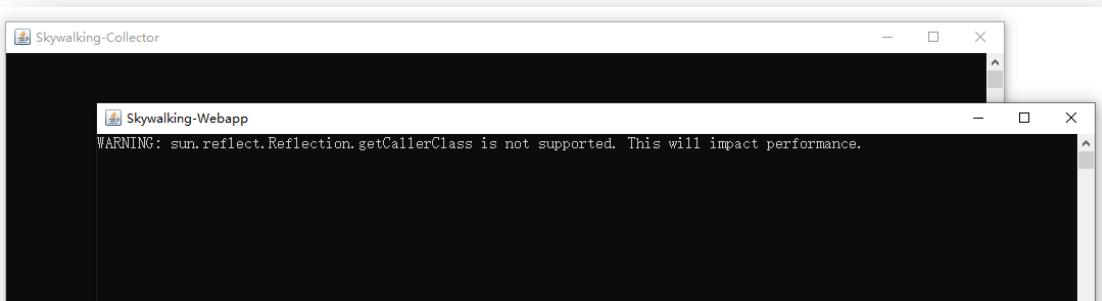
```
15
16
17 serverPort: ${SW_SERVER_PORT:-9999}
18
19 # Comma seperated list of OAP addresses.
20 oapServices: ${SW_OAP_ADDRESS:-http://localhost:12800}
21
22 zipkinServices: ${SW_ZIPKIN_ADDRESS:-http://localhost:9412}
```

### (3) 启动



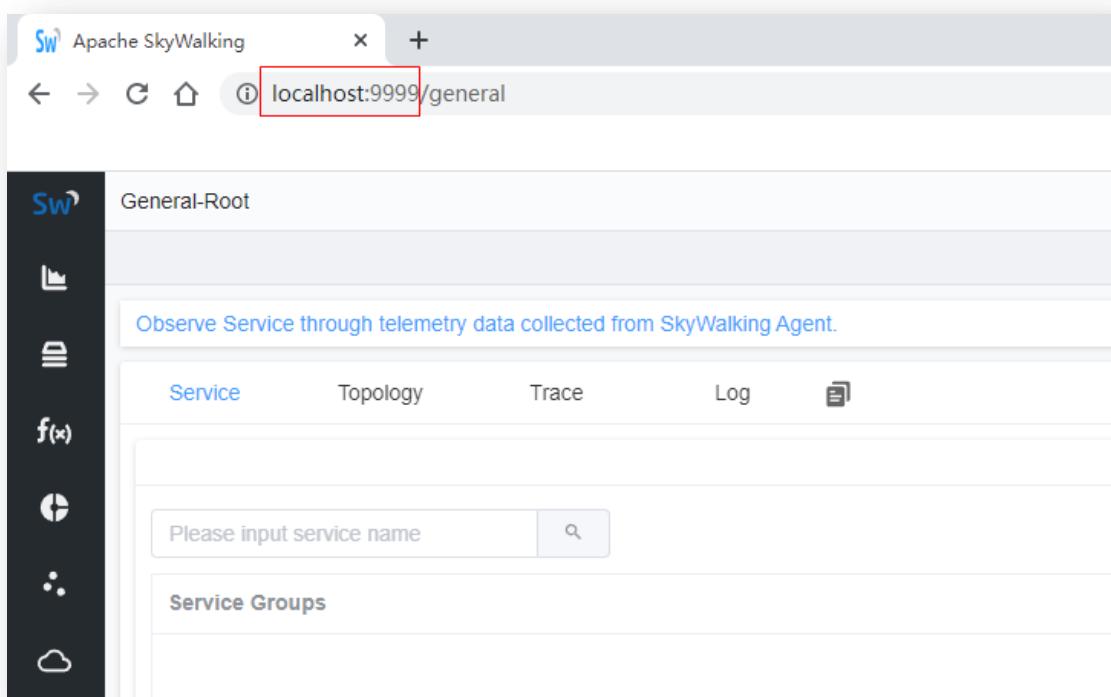
bin 目录中的命令分为三类：OAP 启动相关命令、UI 启动相关命令，与整体启动相关命令。直接运行 startup 命令，可以同时启动 OAP 与 UI。

从这里也可以看出，SW 服务端中包含 OAP Server 与 UI Server 这两部分。在 SW 目录的 bin 下直接双击 startup.bat 即可同时启动这两部分。启动后即可看到如下两个窗口，一个是 Collector，一个是 Webapp。即一个是 OAP Server，一个是 UIServer。



### (4) 访问

能够在浏览器中看到如下页面，说明 SW 启动成功。



### 8.3.4 关于 SW 集群

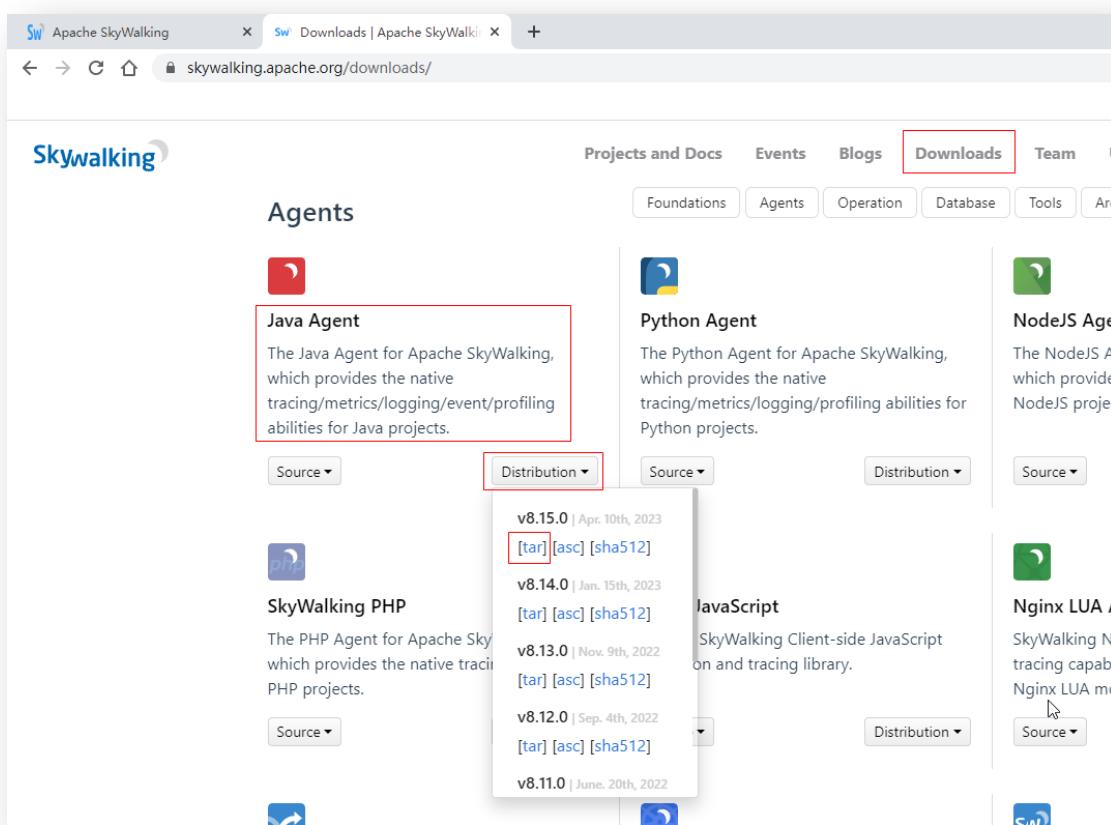
生产环境下，很多公司为了方便，SW 一般不搭建集群。因为 SW 的短暂的宕机对业务是没有影响的。SW 集群纯属于性能检测模块，非业务模块。

## 8.4 Agent 的安装配置

下面要将 Agent 安装配置到准备由 SW 跟踪的应用。我们这里创建两个应用，provider 与 consumer。

### 8.4.1 下载 Java Agent

在官网下载页面下载最新版本的 Java Agent。

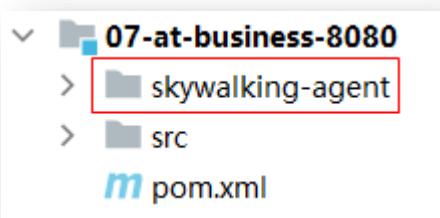


## 8.4.2 修改工程 07-at-business-8080

为了可以看到更多的服务调用层级，这里使用前面章节中的 business 工程，让 business 工程调用 stock 与 order 工程，order 工程调用 account 工程，而 stock、order 与 account 工程均调用了 mysql。

### (1) 复制 agent

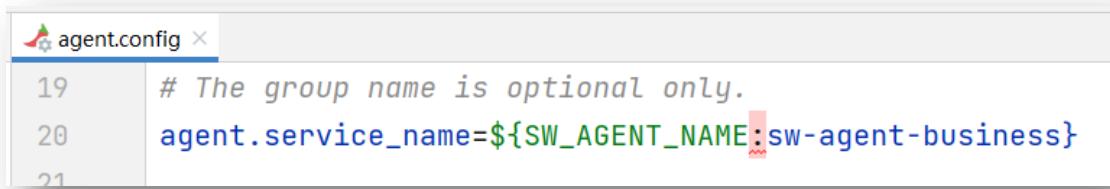
将 agent 目录全部复制到该工程中。



## (2) 修改 agent.config

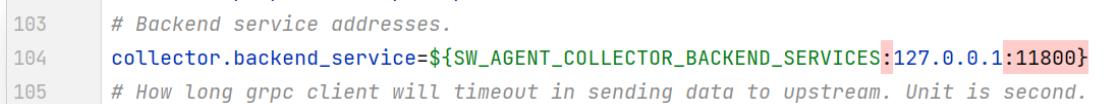
需要修改两处位置：

指定当前 agent 的名称，一般在命名时会起一个与当前应用相关的名称，以表明身份。



```
agent.config x
19 # The group name is optional only.
20 agent.service_name=${SW_AGENT_NAME:sw-agent-business}
21
```

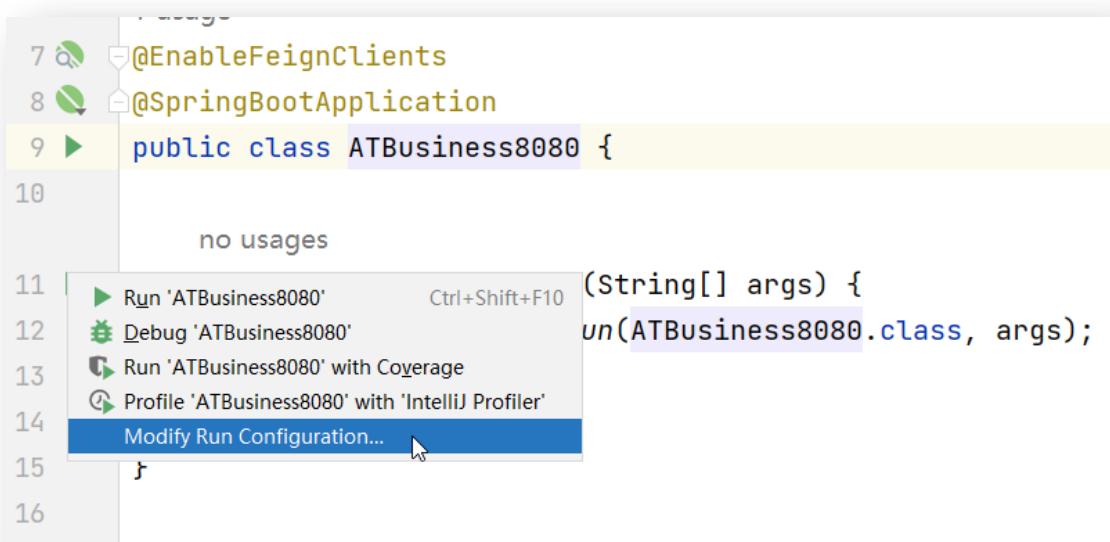
指定当前 Agent 要连接的 SkyWalking 收集器地址，即 OAP Server 地址。由于当前 SW 安装在本地，所以无需修改。若需要修改，则仅修改 ip，不能修改端口号。若要修改，则需要先修改 OAP Server 的端口。若是 OAP 集群，则使用逗号分隔。



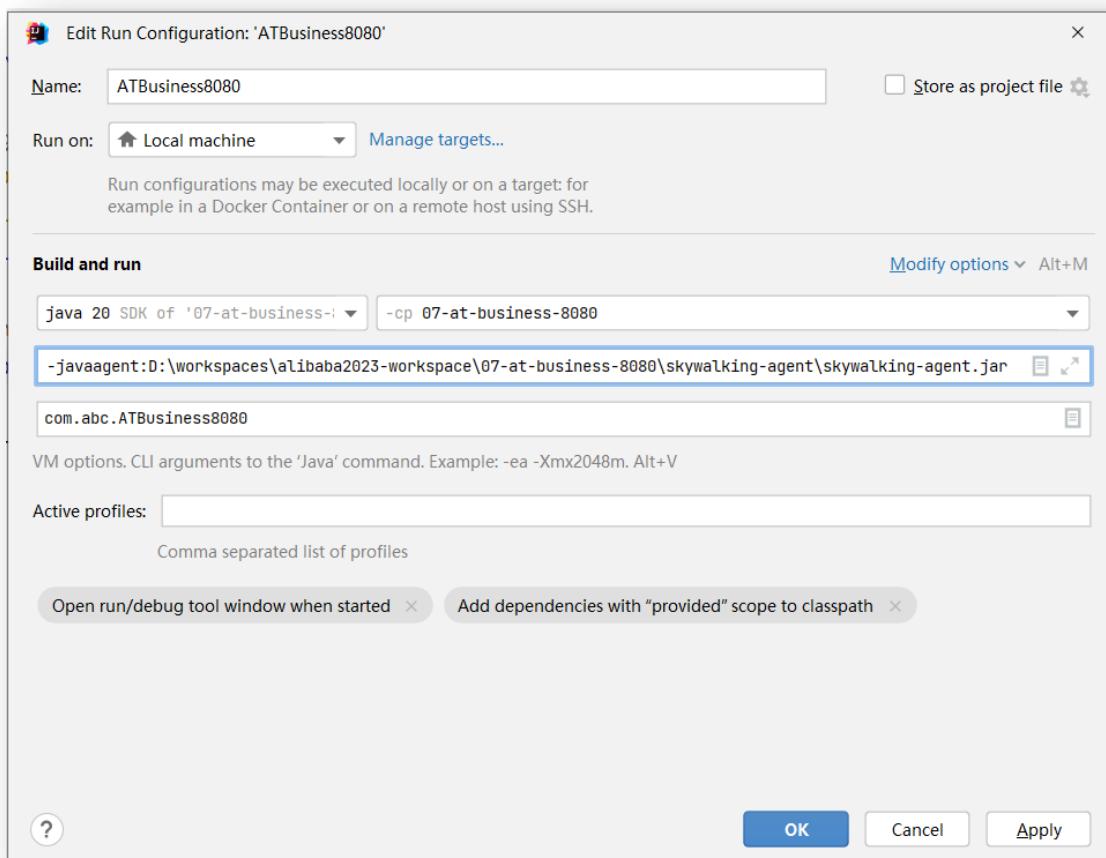
```
103 # Backend service addresses.
104 collector.backend_service=${SW_AGENT_COLLECTOR_BACKEND_SERVICES:127.0.0.1:11800}
105 # How long grpc client will timeout in sending data to upstream. Unit is second.
```

## (3) 修改启动类

在启动应用时，需要同时启动 Agent，所以需要在启动前首先配置一下该应用的 VM options。



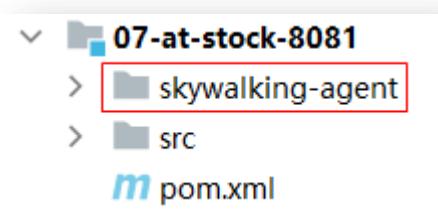
```
7 @EnableFeignClients
8 @SpringBootApplication
9 public class ATBusiness8080 {
10
11     no usages
12     ▶ Run 'ATBusiness8080'      Ctrl+Shift+F10
13     ⏹ Debug 'ATBusiness8080'
14     ⌂ Run 'ATBusiness8080' with Coverage
15     ⌂ Profile 'ATBusiness8080' with 'IntelliJ Profiler'
16 }
```



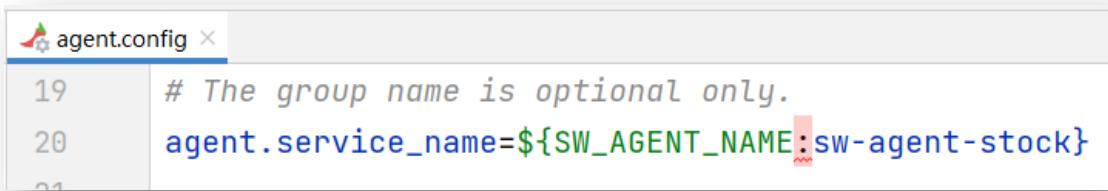
### 8.4.3 修改工程 07-at-stock-8081

#### (1) 复制 agent

将 agent 目录全部复制到该工程中。

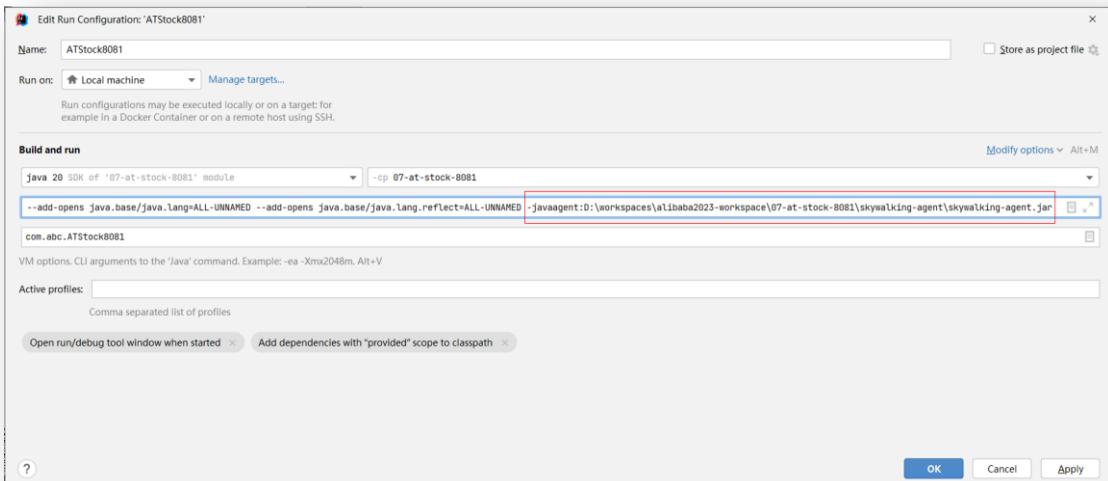


## (2) 修改 agent.config



```
agent.config x
19 # The group name is optional only.
20 agent.service_name=${SW_AGENT_NAME:sw-agent-stock}
21
```

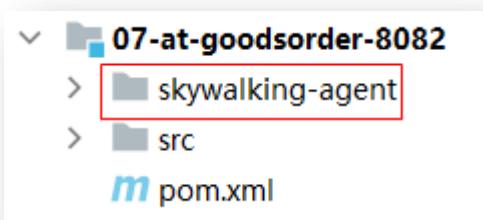
## (3) 修改启动类



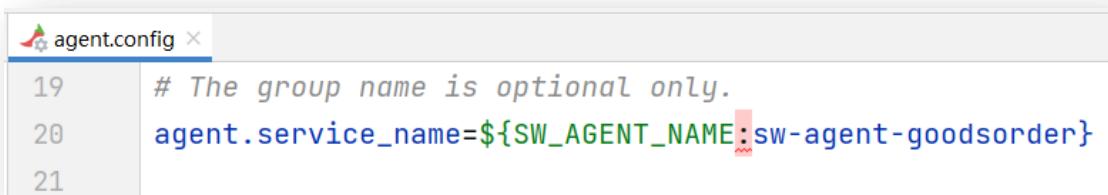
## 8.4.4 修改工程 07-at-goodsorder-8082

## (1) 复制 agent

将 agent 目录全部复制到该工程中。

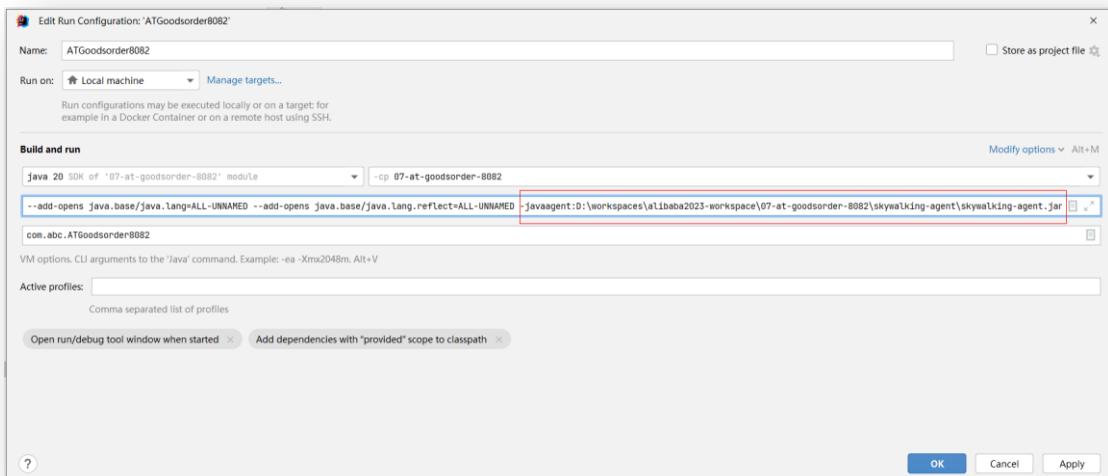


## (2) 修改 agent.config



```
19 # The group name is optional only.
20 agent.service_name=${SW_AGENT_NAME:sw-agent-goodsorder}
21
```

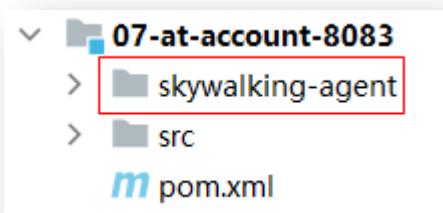
## (3) 修改启动类



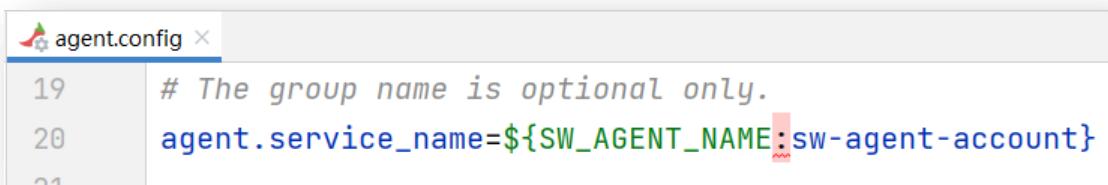
### 8.4.5 修改工程 07-at-account-8083

#### (1) 复制 agent

将 agent 目录全部复制到该工程中。

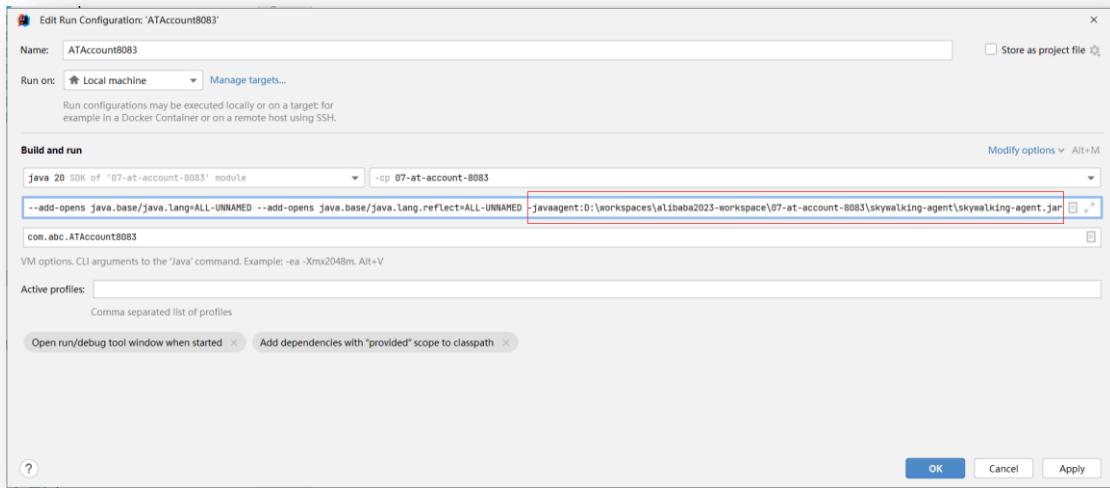


#### (2) 修改 agent.config



```
agent.config
19 # The group name is optional only.
20 agent.service_name=${SW_AGENT_NAME:sw-agent-account}
21
```

### (3) 修改启动类



## 8.4.6 启动系统

### (1) 启动服务器

- 启动 4 台 mysql 服务器：mysql、mysql1、mysql2 与 mysql3。同时，还需要在 mysql1、mysql2 与 mysql3 的 test 数据库中创建出 undo\_log 表。
- 启动 nacos
- 启动 es
- 启动 skywalking
- 启动 seata-server

### (2) 启动应用

- 启动 business 应用
- 启动 stock、goodsorder、account 应用

## (3) 查看 nacos

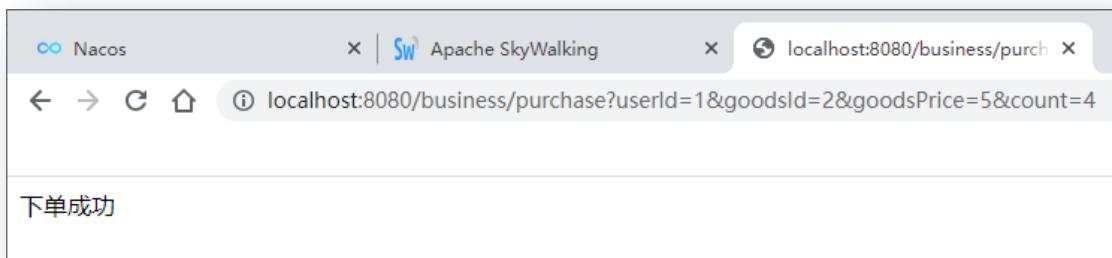


服务名	分组名称	集群数目
ec-stock	SEATA_GROUP	1
ec-account	SEATA_GROUP	1
ec-goodsorder	SEATA_GROUP	1
ec-business	SEATA_GROUP	1
seata-server	SEATA_GROUP	1

## 8.4.7 查看 UI 控制台

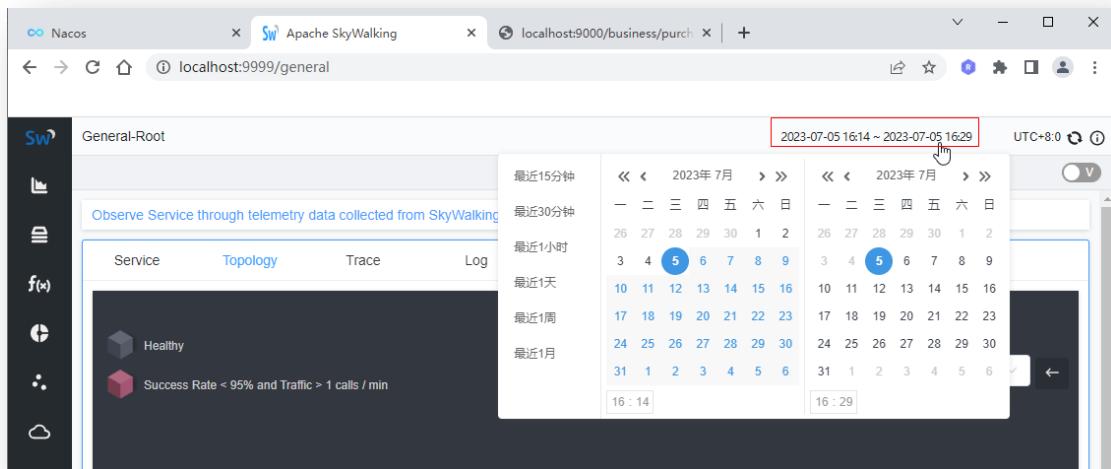
## (1) 访问应用

多次提交如下请求。



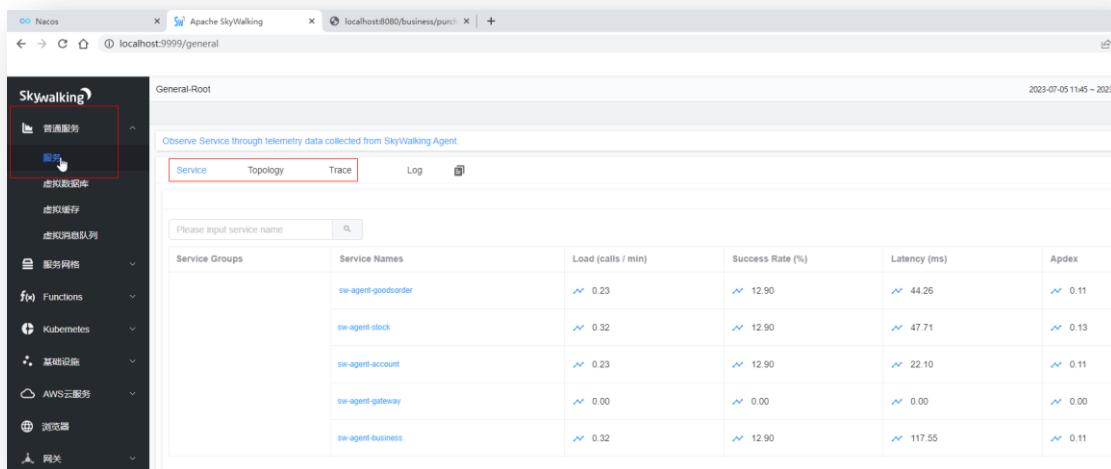
## (2) 选择时间段

在 UI 页面的左侧边栏中点击“普通服务 → 服务”，打开 General-Root 页面，点击该页面右上角的时间范围，可对要显示的调用情况时间段进行选择。

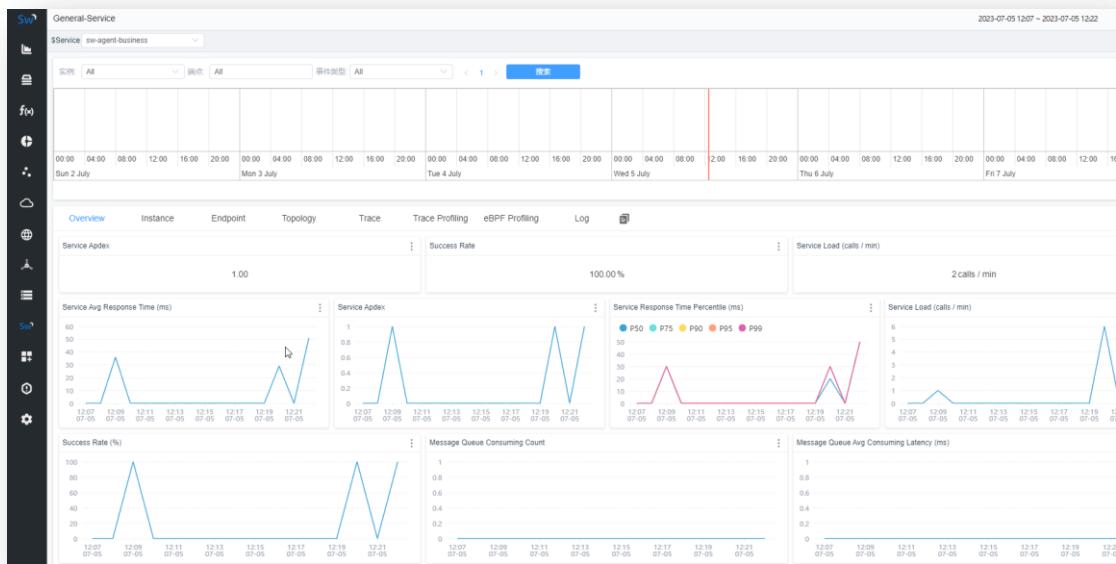


## (3) Service

在 Service 面板可看到发生调用的所有服务。

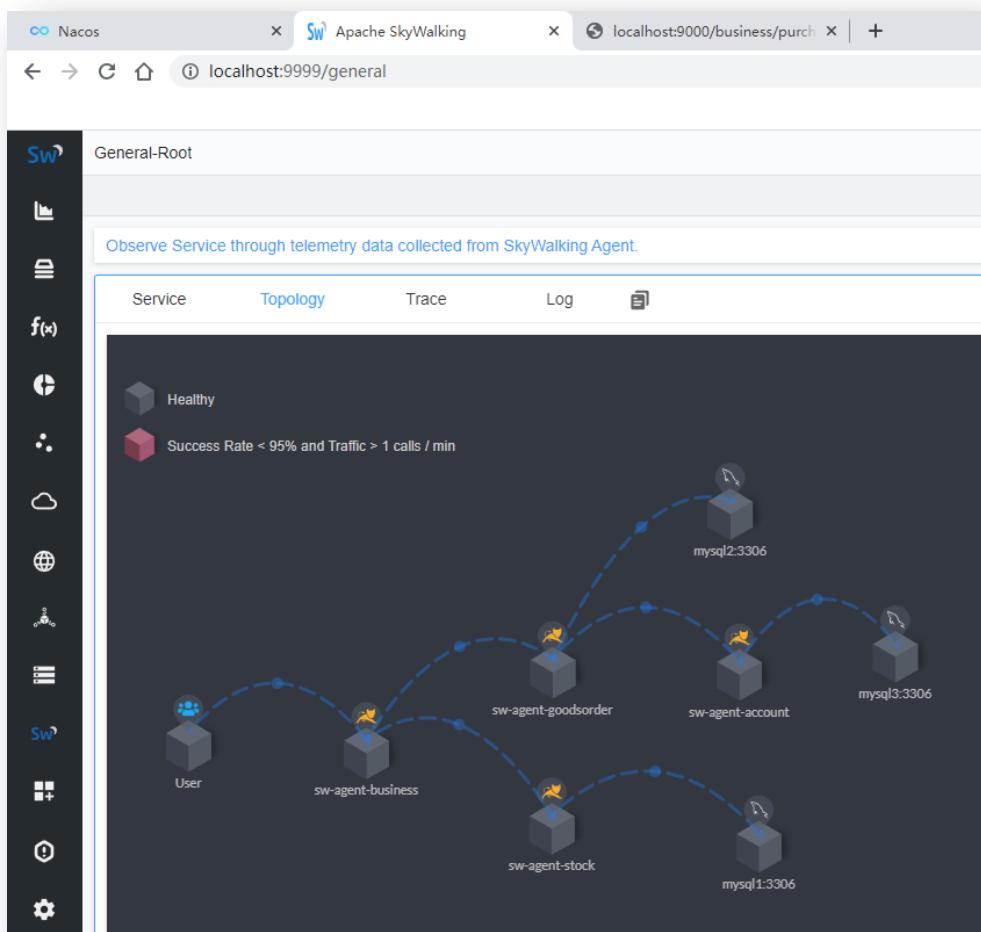


点击每个服务，都可看到更为详细的指标数据。



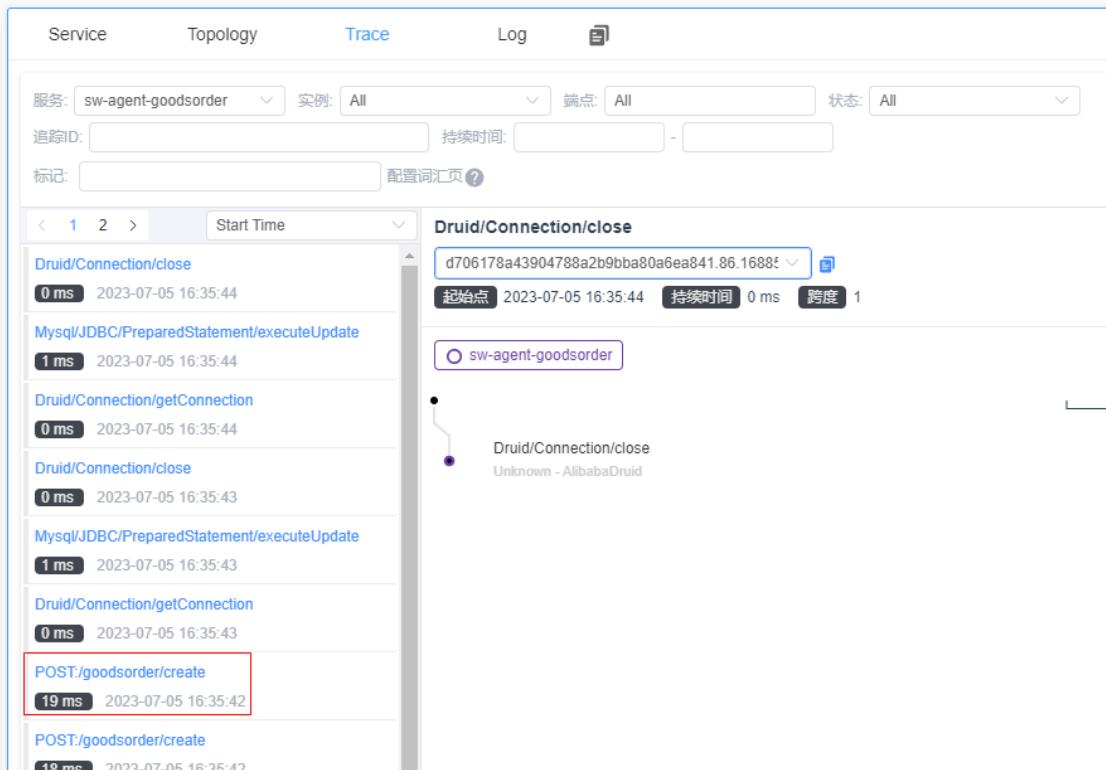
## (4) Topology

在 Topology 面板中还可看到形象的拓扑图。

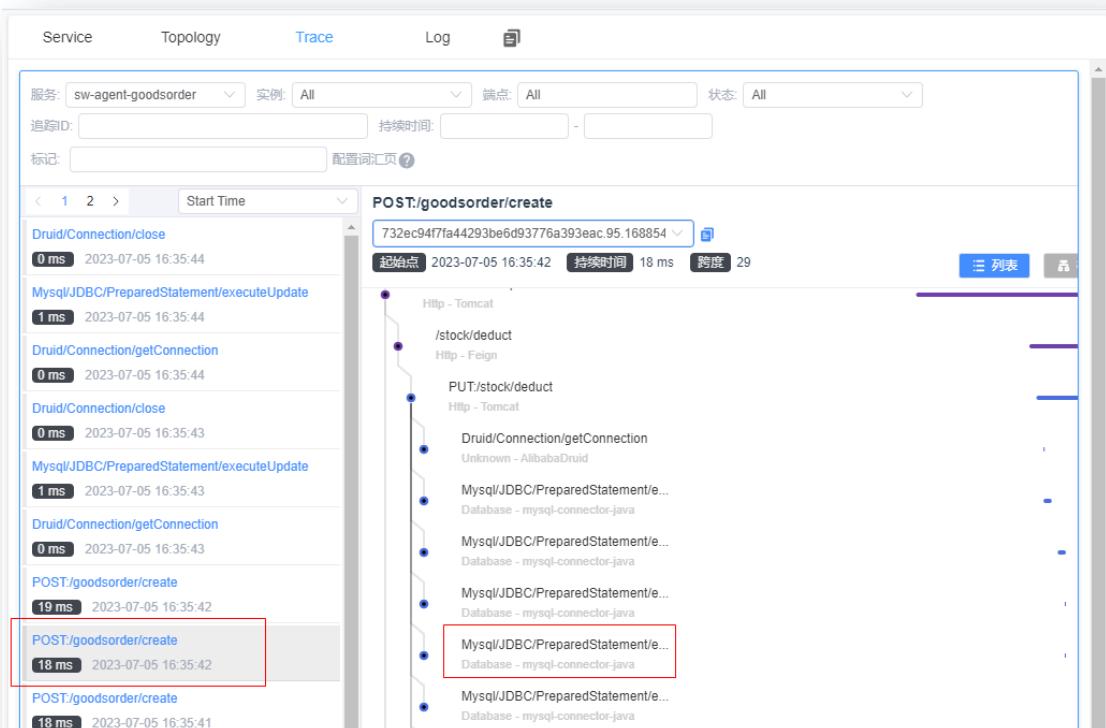


## (5) Trace

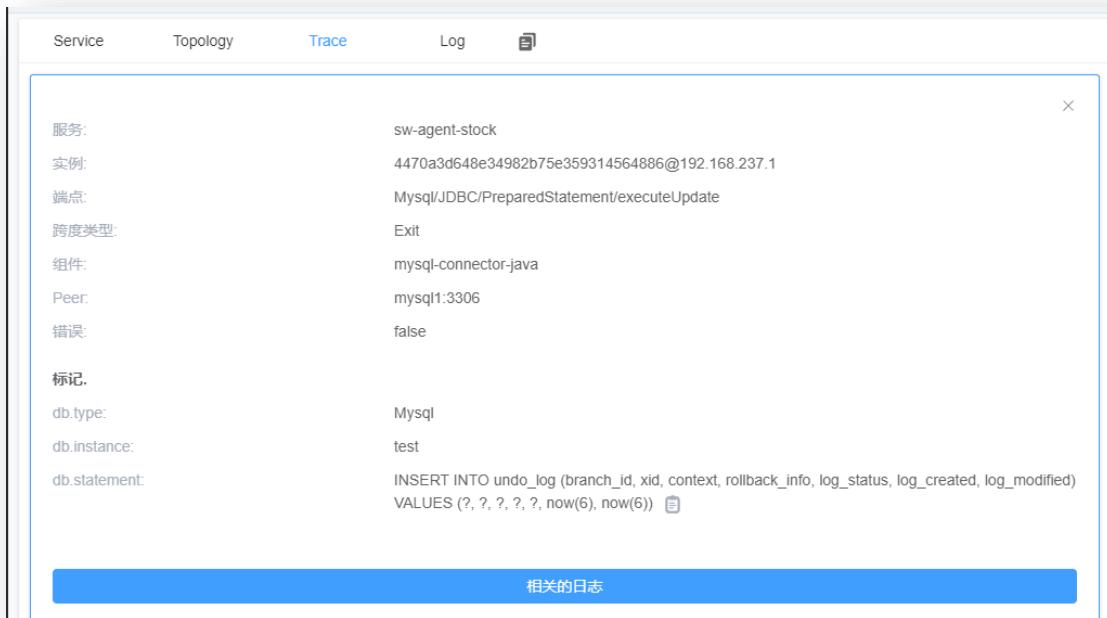
在 Trace 面板中可看到当前服务的所有调用情况。



点击一个调用，可看到更详细的调用情况。其中最上面显示的就是其 TracId。



点击任意一个调用细节，可看到更详细的说明：数据库名称、SQL语句都可看到。



Service: sw-agent-stock  
Topology: 4470a3d648e34982b75e359314564886@192.168.237.1  
Trace: MySql/JDBC/PreparedStatement/executeUpdate  
Log: Exit  
Component: mysql-connector-java  
Peer: mysql1:3306  
Error: false  
  
Marker:  
db.type: MySql  
db.instance: test  
db.statement: INSERT INTO undo\_log (branch\_id, xid, context, rollback\_info, log\_status, log\_created, log\_modified) VALUES (?, ?, ?, ?, now(6), now(6))

相关的日志

## (6) Log

在 Log 面板中可看到当前服务调用过程中生成的所有日志。



Service: sw-agent-goodsorder 实例: All 端点: All 搜索  
追踪ID: 标记: 配置词汇页  
请输入一个内容关键词或者内容不包含的关键词(key=value)之后回车  
内容关键词: 请输入一个内容关键词 内容不包含的关键词: 请输入一个内容不包含的关键词

服务	实例	端点	时间	内容类型	标记
数据为空					

## 8.5 告警

Skywalking 具有告警功能，即用户可提前设置好告警规则，一旦调用过程出现符合告警规则的调用，则会触发相应的 Webhooks 的执行，并在 UI 控制台的“告警”页面中显示相应的告警信息，以通知管理员目前出现的问题。

### 8.5.1 修改 alarm-settings.yml

修改 config/alarm-settings.yml 文件内容。首先修改以下两个告警规则。

```
10
17 # Sample alarm rules.
18 rules:
19   # Rule unique name, must be ended with `_rule`.
20   service_resp_time_rule:
21     metrics-name: service_resp_time
22     op: ">"
23     threshold: 1000
24     period: 10
25     count: 1
26     silence-period: 1
27     message: Response time of service {name} is more than 1000ms in 1 minutes of last 10 minutes.
28   service_sla_rule:
29     # Metrics value need to be long, double or int
30     metrics-name: service_sla
31     on: "<"

49   service_instance_resp_time_rule:
50     metrics-name: service_instance_resp_time
51     op: ">"
52     threshold: 1000
53     period: 10
54     count: 1
55     silence-period: 1
56     message: Response time of service instance {name} is more than 1000ms in 1 minutes of last 10 minutes
```

然后再修改 webhooks。

```
82
83 webhooks:
84   - http://127.0.0.1:8080/alarm
85   # - http://127.0.0.1/go-wechat/
86
87
```

### 8.5.2 修改 business 工程

修改 07-at-business-8080 工程的 BusinessController 类。

#### (1) 修改 purchaseHandle()

在 purchaseHandle()方法中添加 sleep()方法，以使该方法的响应时间超出 1000ms。

```
@GlobalTransactional
@GetMapping("/business/purchase")
public String purchaseHandle(@RequestParam("userId") Integer userId,
                             @RequestParam("goodsId") Integer goodsId,
                             @RequestParam("goodsPrice") Double goodsPrice,
                             @RequestParam("count") Integer count) {

    try {
        TimeUnit.MILLISECONDS.sleep(1500);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    // 减库存
    Stock stock = stockService.deductStock(goodsId, count);
    if (stock == null) {
```

## (2) 添加 alarmHandle()方法

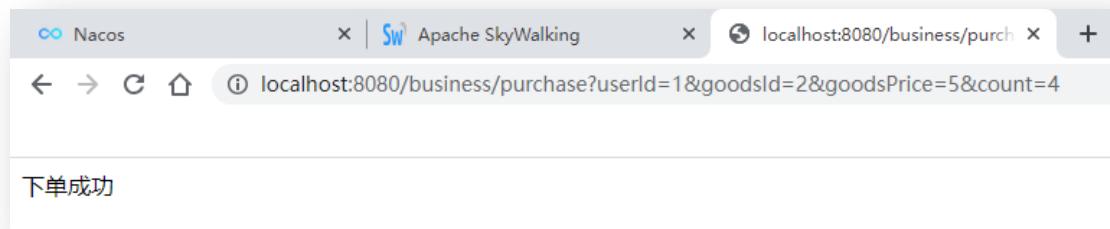
在 BusinessController 类中添加 alarmHandle()方法。该方法是告警的 webhooks 方法。该方法原本应该定义在专门的一个工程中，为了简单，就定义在了这里。

当发生符合告警规则的调用时，skywalking 会向 webhooks 提交一个 POST 请求，并携带着告警相关的信息。

```
@PostMapping("/alarm")
public void alarmHandle(@RequestBody Object alarmMsg) {
    // 向管理发送告警信息：短信、钉钉、微信等消息
    System.out.println("alarm message:" + alarmMsg);
}
```

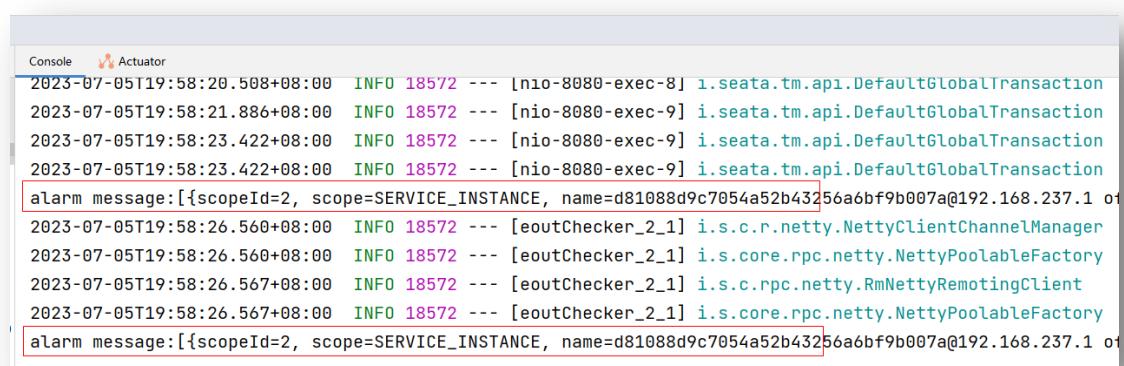
### 8.5.3 启动并访问

由于 alarm-settings.yml 文件及 BusinessController 类发生了改变，所以需要重新启动 Skywalking 服务器及 07-at-business-8080 工程。然后多次提交如下请求。



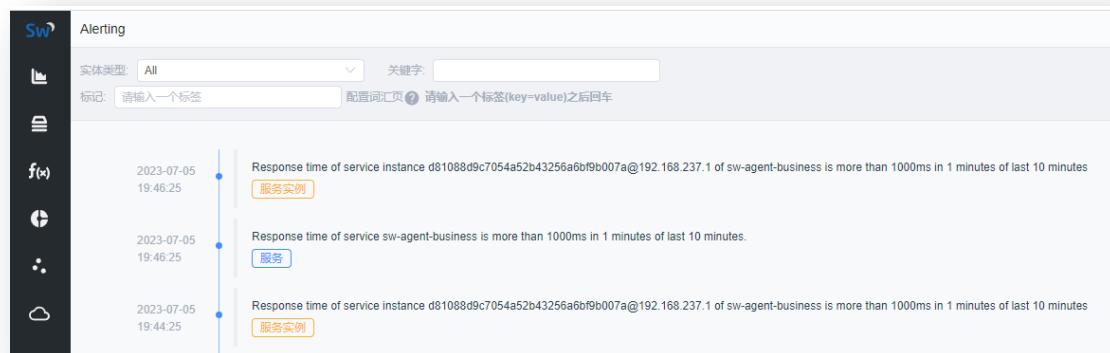
#### 8.5.4 查看告警信息

查看 Idea 控制台，可以看到触发的 webhooks 调用。



```
Console Actuator
2023-07-05T19:58:20.508+08:00 INFO 18572 --- [nio-8080-exec-8] i.seata.tm.api.DefaultGlobalTransaction
2023-07-05T19:58:21.886+08:00 INFO 18572 --- [nio-8080-exec-9] i.seata.tm.api.DefaultGlobalTransaction
2023-07-05T19:58:23.422+08:00 INFO 18572 --- [nio-8080-exec-9] i.seata.tm.api.DefaultGlobalTransaction
2023-07-05T19:58:23.422+08:00 INFO 18572 --- [nio-8080-exec-9] i.seata.tm.api.DefaultGlobalTransaction
alarm message:[{scopeId=2, scope=SERVICE_INSTANCE, name=d81088d9c7054a52b43256a6bf9b007a@192.168.237.1 of
2023-07-05T19:58:26.560+08:00 INFO 18572 --- [eoutChecker_2_1] i.s.c.r.netty.NettyClientChannelManager
2023-07-05T19:58:26.560+08:00 INFO 18572 --- [eoutChecker_2_1] i.s.core.rpc.netty.NettyPoolableFactory
2023-07-05T19:58:26.567+08:00 INFO 18572 --- [eoutChecker_2_1] i.s.c.rpc.netty.RmNettyRemotingClient
2023-07-05T19:58:26.567+08:00 INFO 18572 --- [eoutChecker_2_1] i.s.core.rpc.netty.NettyPoolableFactory
alarm message:[{scopeId=2, scope=SERVICE_INSTANCE, name=d81088d9c7054a52b43256a6bf9b007a@192.168.237.1 of
```

查看 UI 控制台的告警页面，可以看到相应的告警信息。



时间	告警描述
2023-07-05 19:46:25	Response time of service instance d81088d9c7054a52b43256a6bf9b007a@192.168.237.1 of sw-agent-business is more than 1000ms in 1 minutes of last 10 minutes <span style="background-color: orange; border: 1px solid orange; padding: 2px;">服务实例</span>
2023-07-05 19:46:25	Response time of service sw-agent-business is more than 1000ms in 1 minutes of last 10 minutes. <span style="background-color: blue; border: 1px solid blue; padding: 2px;">服务</span>
2023-07-05 19:44:25	Response time of service instance d81088d9c7054a52b43256a6bf9b007a@192.168.237.1 of sw-agent-business is more than 1000ms in 1 minutes of last 10 minutes <span style="background-color: orange; border: 1px solid orange; padding: 2px;">服务实例</span>

不过需要注意的是，无论是 Idea 控制台还是 UI 控制台告警页面，其告警信息的显示并不是实时的，即并不是立即可以看到的，需要等待几分钟时间。

## 8.6 Tracing API

Skywalking 除了为客户端提供了探针 Agent 外，还提供了相应的 API，用于在代码中对调用跟踪进行处理。当然，该方式会对代码形成侵入，会污染代码。

### 8.6.1 TraceContext

这里以 TraceContext 为例来简单学习 Tracing API 的使用。TraceContext，跟踪上下文，通过其可以获取到 traceId，也可在其中完成属性的写与读。

下面仍在上面例子的基础上修改代码来说明问题。

#### (1) 导入依赖

在需要使用 Tracing API 的工程中导入依赖。这里在 07-at-business-8080 与 07-at-account-8083 工程中导入依赖。

```
<dependency>
    <groupId>org.apache.skywalking</groupId>
    <artifactId>apm-toolkit-trace</artifactId>
    <version>8.16.0</version>
</dependency>
```

#### (2) 修改 BusinessController 类

在 07-at-business-8080 工程的 BusinessController 类的 purchaseHandle()方法中添加如下代码：获取 traceId 与当前系统时间 businessTime，然后放入到 TraceContext 上下文中。

```
@GlobalTransactional
@GetMapping("/business/purchase")
public String purchaseHandle(@RequestParam("userId") Integer userId,
                             @RequestParam("goodsId") Integer goodsId,
                             @RequestParam("goodsPrice") Double goodsPrice,
                             @RequestParam("count") Integer count) {

    // try {
    //     TimeUnit.MILLISECONDS.sleep(1500);
    // } catch (InterruptedException e) {
    //     throw new RuntimeException(e);
    // }

    String traceId = TraceContext.traceId();
    System.out.println("traceId == " + traceId);

    long businessTime = System.currentTimeMillis();
    System.out.println("businessTime == " + businessTime);
    TraceContext.putCorrelation("businessTime", String.valueOf(businessTime));
}
```

### (3) 修改 AccountController 类

在 07-at-account-8083 工程的 AccountController 类的 debitHandle()方法中添加如下代码：获取 traceId，并从 TraceContext 上下文中获取前面放入的 businessTime。

```
@PutMapping("/account/debit")
public Account debitHandle(@RequestParam("userId") Integer userId,
                           @RequestParam("amount") Double amount) {

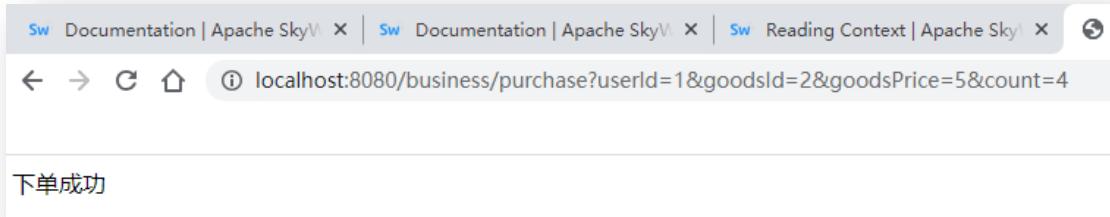
    String traceId = TraceContext.traceId();
    System.out.println("traceId == " + traceId);

    Optional<String> businessTime = TraceContext.getCorrelation("businessTime");
    System.out.println("businessTime == " + businessTime.get());

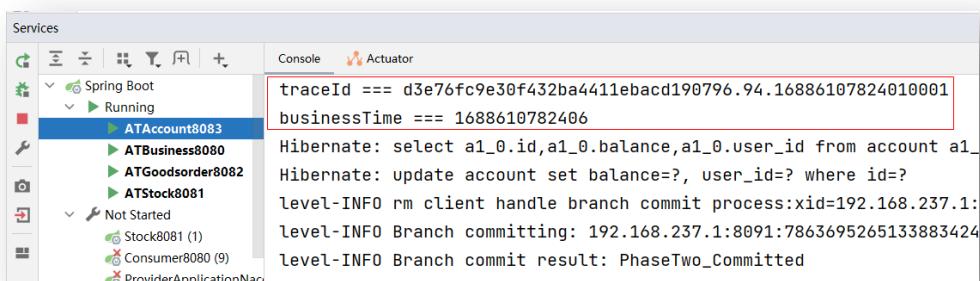
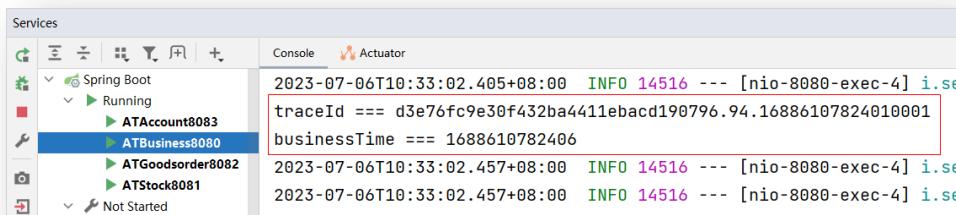
    // 支出账户
    return service.debitAccount(userId, amount);
}
```

## (4) 运行访问

重启 07-at-business-8080 与 07-at-account-8083 工程，然后再次提交如下请求。



此时查看 Idea 中 07-at-business-8080 与 07-at-account-8083 工程控制台，可以看出它们输出的 `traceId` 与 `businessTime` 是相同的。说明它们同在一个 Trace 中，并且 `TraceContext` 可以在同一个 Trace 中传递数据。

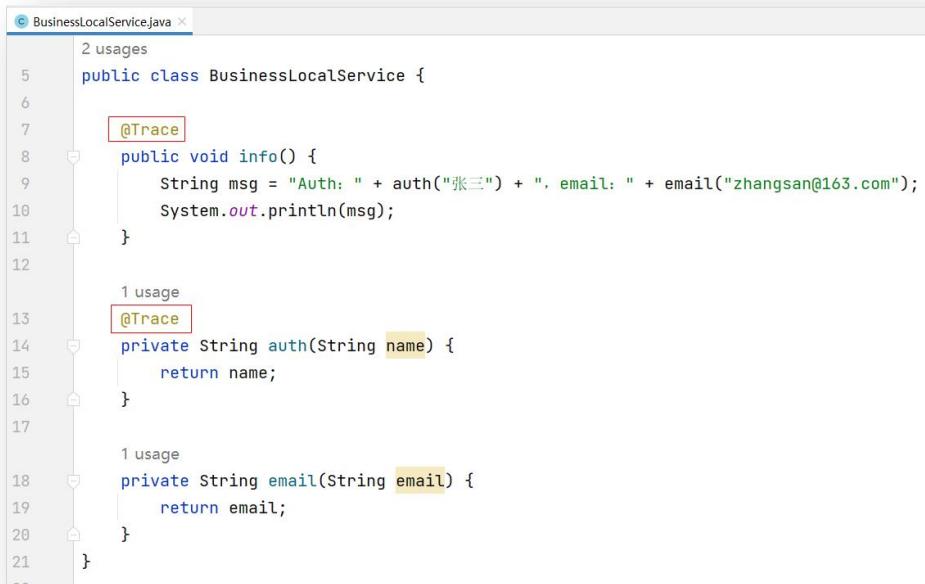


### 8.6.2 @Trace

默认情况下 Skywalking 仅会对服务间的远程调用进行跟踪，对于本地调用是不进行跟踪的，即本地调用过程在 Skywalking 的 UI 控制台是看不到的。通过在本地方法上添加`@Trace`，可将该方法纳入到 Skywalking 的跟踪范围，从 UI 控制台可看到其调用过程。不过需要注意的是，该注解不能用于静态方法。

## (1) 定义 BusinessLocalService 类

在 07-at-business-8080 工程中的 com.abc.servcie 包下定义类 BusinessLocalService。



```
BusinessLocalService.java
2 usages
public class BusinessLocalService {
    @Trace
    public void info() {
        String msg = "Auth: " + auth("张三") + ", email: " + email("zhangsan@163.com");
        System.out.println(msg);
    }

    1 usage
    @Trace
    private String auth(String name) {
        return name;
    }

    1 usage
    private String email(String email) {
        return email;
    }
}
```

## (2) 修改 BusinessController 类



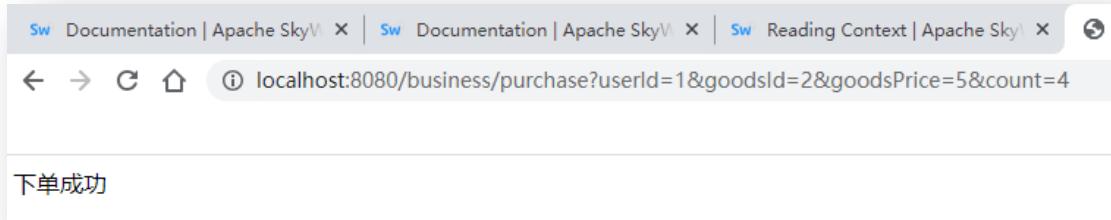
```
@GlobalTransactional
@GetMapping("/business/purchase")
public String purchaseHandle(@RequestParam("userId") Integer userId,
                             @RequestParam("goodsId") Integer goodsId,
                             @RequestParam("goodsPrice") Double goodsPrice,
                             @RequestParam("count") Integer count) {

    new BusinessLocalService().info();

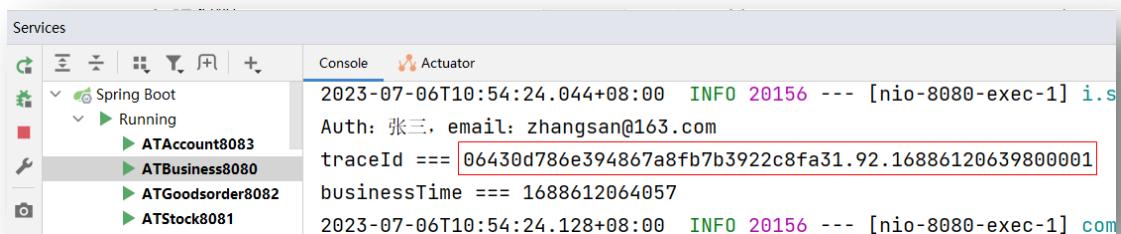
    // try {
    //     TimeUnit.MILLISECONDS.sleep(1500);
    // } catch (InterruptedException e) {
    //     throw new RuntimeException(e);
    // }
}
```

### (3) 运行访问

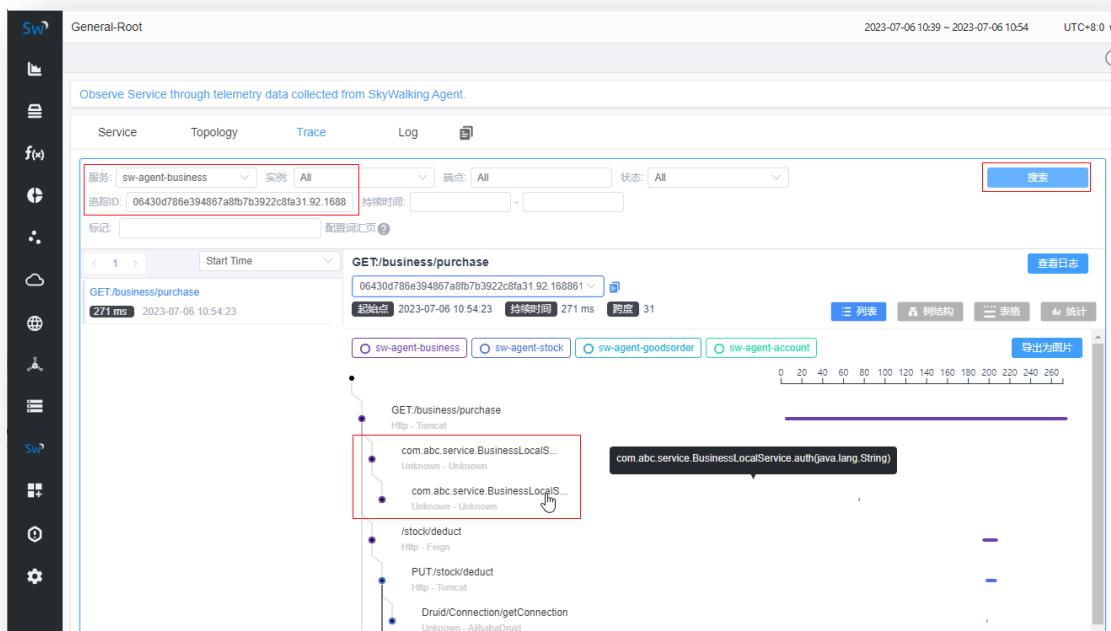
重启 07-at-business-8080 工程，然后再次提交如下请求。



然后从 Idea 控制台中复制出 traceId。



在 Skywalking 的 UI 控制台的 Trace 中选择 sw-agent-business 服务，并将复制来的 traceId 粘贴到“追踪 ID”中进行搜索，此时在下面可以看到对于 BusinessLocalService 类中 info()方法与 auth()方法的调用，但看不到 email()方法的，因为 email()方法上没有@Trace 注解。



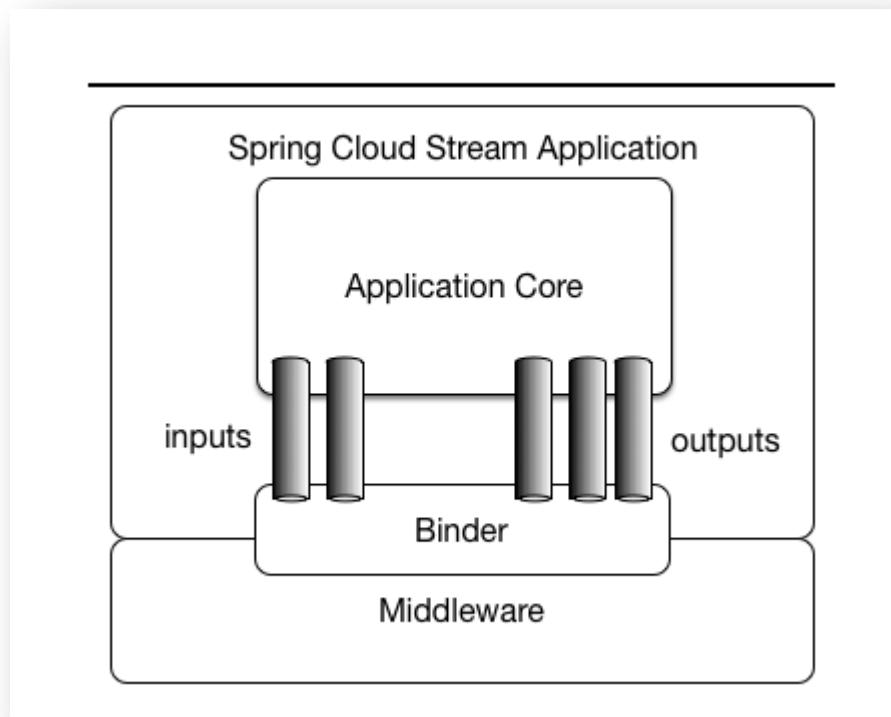
## 第9章 消息系统整合 Spring Cloud Stream

### 9.1 简介

Spring Cloud Stream 是一个用来为微服务应用构建消息驱动能力的框架。通过使用 Spring Cloud Stream，可以有效简化开发人员对消息中间件的使用复杂度，降低代码与消息中间件间的耦合度，屏蔽消息中间件之间的差异性，让开发人员可以有更多的精力关注于核心业务逻辑的处理。

### 9.2 应用程序模型

#### 9.2.1 模型图



## 9.2.2 概念

### (1) Destination Binders

Destination Binders: Components responsible to provide integration with the external messaging systems.

目的地绑定器：负责提供与外部消息系统集成的组件。

### (2) Bindings

Bindings: Bridge between the external messaging systems and application provided Producers and Consumers of messages (created by the Destination Binders).

固定器：介于外部消息系统与应用程序间的桥梁，这个应用程序提供了生产者和消费者的消

### (3) Input Bindings

输入管道，消费者通过 Input Bindings 连接 Binder，而 Binder 与 MQ 连接，即消费者通

过 Input Bindings 从 MQ 读取数据。

### (4) Output Bindings

输出管道，生产者通过 Output Bindings 连接 Binder，而 Binder 与 MQ 连接，即生产者

通过 Output Bindings 向 MQ 写入数据。

### (5) Message

Message: The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

消息：生产者和消费者使用的规范数据结构，用于与 Binders 通信（从而通过外部消息系统与其他应用程序通信）。

## 9.3 RocketMQ 的安装与启动

### 9.3.1 准备工作

#### (1) 系统要求

系统要求是 64 位 Linux/Unix/macOS，JDK 要求是 64 位 1.8+。

#### (2) 克隆并配置主机

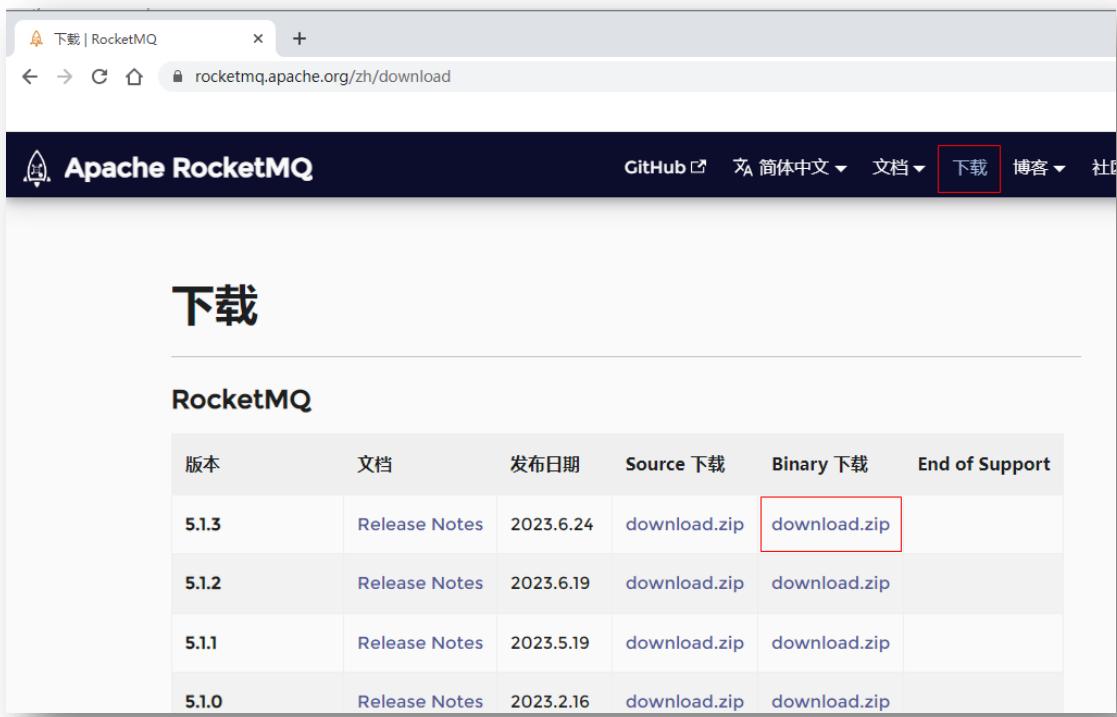
克隆一台干净的主机，并修改配置。

- 修改主机名: /etc/hostname
- 修改网络配置: /etc/sysconfig/network-scripts/ifcfg-ens33

### 9.3.2 安装配置 RocketMQ

#### (1) 下载 RocketMQ 安装包

从官网下载 RocketMQ 安装包，并上传到 Linux 的/opt/tools 目录。

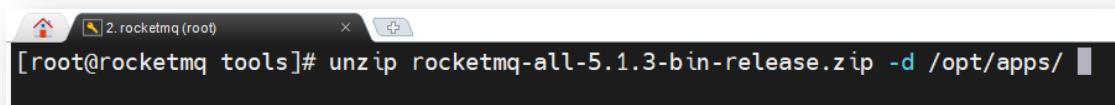


The screenshot shows the Apache RocketMQ download page. At the top, there's a navigation bar with links for 'GitHub', '简体中文' (Simplified Chinese), '文档' (Documentation), '下载' (Download) which is highlighted with a red box, '博客' (Blog), and '社区' (Community). Below the navigation is a large 'Apache RocketMQ' logo. The main content area has a title 'RocketMQ' and a subtitle '下载'. A table lists the available versions:

版本	文档	发布日期	Source 下载	Binary 下载	End of Support
5.1.3	<a href="#">Release Notes</a>	2023.6.24	<a href="#">download.zip</a>	<a href="#">download.zip</a>	
5.1.2	<a href="#">Release Notes</a>	2023.6.19	<a href="#">download.zip</a>	<a href="#">download.zip</a>	
5.1.1	<a href="#">Release Notes</a>	2023.5.19	<a href="#">download.zip</a>	<a href="#">download.zip</a>	
5.1.0	<a href="#">Release Notes</a>	2023.2.16	<a href="#">download.zip</a>	<a href="#">download.zip</a>	

## (2) 解压-unzip

解压到/opt/apps 目录下，并通过 mv 命令将其重命名为 rocketmq。



## (3) 修改 runserver.sh 初始内存

使用 vim 命令打开 bin/runserver.sh 文件。我们发现其初始值太大了，不是 4g 就是 2g。

```
82
83 choose_gc_options()
84 {
85     # Example of JAVA_MAJOR_VERSION value : '1', '9', '10', '11', ...
86     # '1' means releases before Java 9
87     JAVA_MAJOR_VERSION=$( "$JAVA" -version 2>&1 | awk -F '"' '/version/ {print $2}' )
88     if [ -z "$JAVA_MAJOR_VERSION" ] || [ "$JAVA_MAJOR_VERSION" -lt "9" ] ; then
89         JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -Xmn2g -XX:MetaspaceSize=128m -XX:
90         kEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRa
91         JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -XX:
92         JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:G1Reser
93     else
94         JAVA_OPT="${JAVA_OPT} -server -Xms4g -Xmx4g -XX:MetaspaceSize=128m -XX:MaxMet
95     JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePerce
96         JAVA_OPT="${JAVA_OPT} -Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,
97     fi
98 }
```

现将这些值修改为如下：

```
82
83 choose_gc_options()
84 {
85     # Example of JAVA_MAJOR_VERSION value : '1', '9', '10', '11', ...
86     # '1' means releases before Java 9
87     JAVA_MAJOR_VERSION=$( "$JAVA" -version 2>&1 | awk -F '"' '/version/ {print $2}' )
88     if [ -z "$JAVA_MAJOR_VERSION" ] || [ "$JAVA_MAJOR_VERSION" -lt "9" ] ; then
89         JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=
90         JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollecti
91         kEnabled -XX:SoftRefLRUPolicyMSPerMB=0 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRa
92         JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -XX:
93     else
94         JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -XX:MetaspaceSize=128m -XX:MaxMet
95         JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePerce
96         JAVA_OPT="${JAVA_OPT} -Xlog:gc*:file=${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log:time,
97     fi
98 }
```

#### (4) 修改 runbroker.sh

使用 vim 命令打开 bin/runbroker.sh 文件。我们发现其初始值更大了，动不动就是 8g。

```
100
101 choose_gc_log_directory
102
103 JAVA_OPT="${JAVA_OPT} -server -Xms8g -Xmx8g"
104 choose_gc_options
105 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
106 JAVA_OPT="${JAVA_OPT} -XX:+AlwaysPreTouch"
```

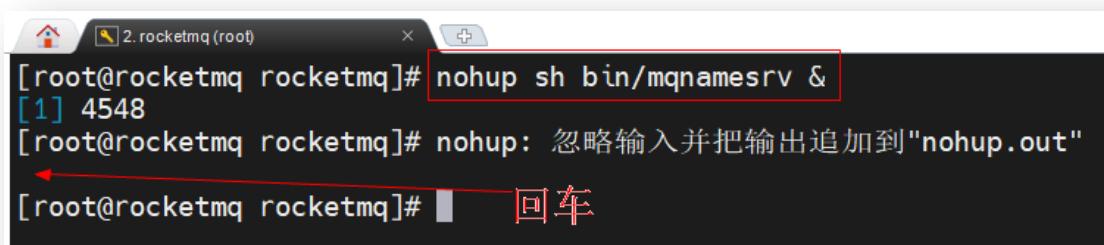
现将这些值修改为 256m:

```
100
101 choose_gc_log_directory
102
103 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m"
104 choose_gc_options
105 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow"
106 JAVA_OPT="${JAVA_OPT} -XX:+AlwaysPreTouch"
```

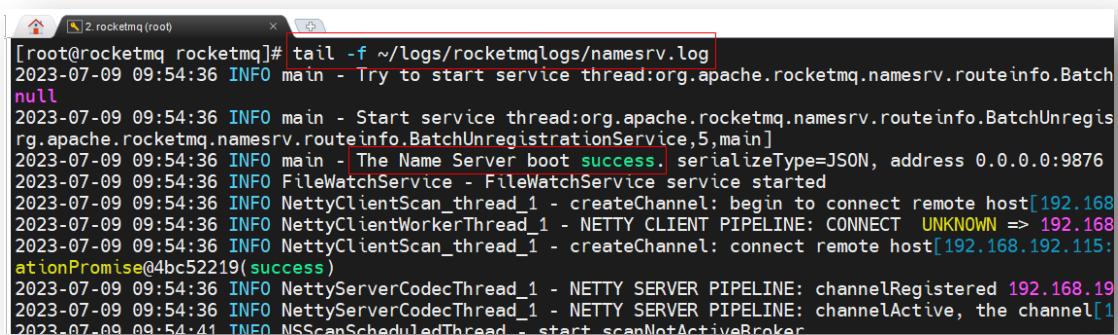
### 9.3.3 启动

#### (1) 启动 NameServer

对于 RocketMQ 的启动，首先需要启动 Name Sever，然后再启动 Broker。Broker 会注册到 Name Server。直接复制官网命令运行即可。



```
[root@rocketmq rocketmq]# nohup sh bin/mqnamesrv &
[1] 4548
[root@rocketmq rocketmq]# nohup: 忽略输入并把输出追加到"nohup.out"
[root@rocketmq rocketmq]# 回车
```



```
[root@rocketmq rocketmq]# tail -f ~/logs/rocketmqlogs/namesrv.log
2023-07-09 09:54:36 INFO main - Try to start service thread:org.apache.rocketmq.namesrv.routeinfo.Batch
null
2023-07-09 09:54:36 INFO main - Start service thread:org.apache.rocketmq.namesrv.routeinfo.BatchUnregis
rg.apache.rocketmq.namesrv.routeinfo.BatchUnregistrationService,5,main]
2023-07-09 09:54:36 INFO main - The Name Server boot success, serializeType=JSON, address 0.0.0.0:9876
2023-07-09 09:54:36 INFO FileWatchService - FileWatchService service started
2023-07-09 09:54:36 INFO NettyClientScan_thread_1 - createChannel: begin to connect remote host[192.168
2023-07-09 09:54:36 INFO NettyClientWorkerThread_1 - NETTY CLIENT PIPELINE: CONNECT UNKNOWN => 192.168
2023-07-09 09:54:36 INFO NettyClientScan_thread_1 - createChannel: connect remote host[192.168.192.115:
actionPromise@4bc52219(success)
2023-07-09 09:54:36 INFO NettyServerCodecThread_1 - NETTY SERVER PIPELINE: channelRegistered 192.168.19
2023-07-09 09:54:36 INFO NettyServerCodecThread_1 - NETTY SERVER PIPELINE: channelActive, the channel[1
2023-07-09 09:54:41 INFO NSScanScheduledThread - start scanNotActiveBroker
```

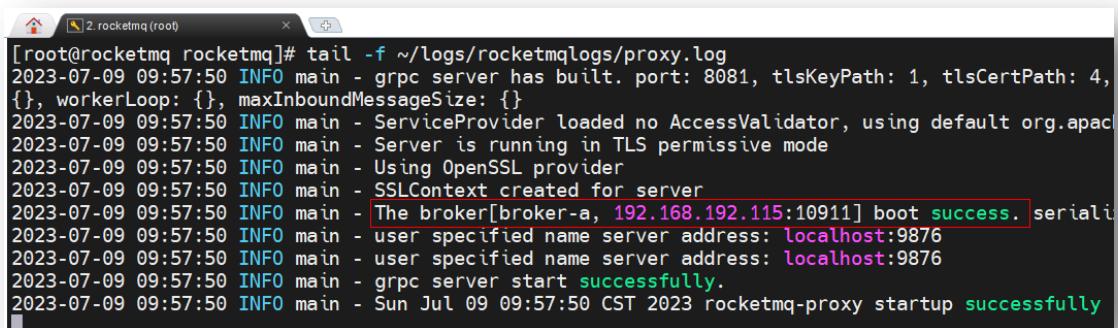
当在日志中看到 The Name Server boot success 时，说明 Name Server 启动成功。

## (2) 启动 broker

直接使用官网命令。



```
[root@rocketmq rocketmq]# nohup sh bin/mqbroker -n localhost:9876 --enable-proxy &
[2] 4983
[root@rocketmq rocketmq]# nohup: 忽略输入并把输出追加到"nohup.out"
[root@rocketmq rocketmq]# 回车
```



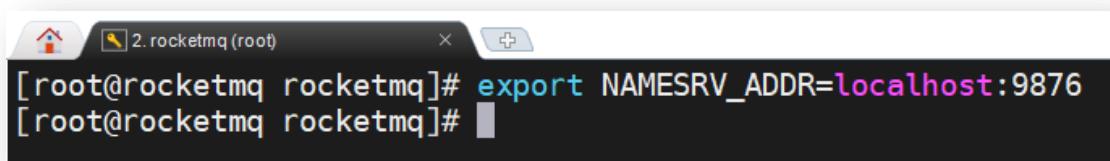
```
[root@rocketmq rocketmq]# tail -f ~/logs/rocketmqlogs/proxy.log
2023-07-09 09:57:50 INFO main - grpc server has built. port: 8081, tlsKeyPath: 1, tlsCertPath: 4,
{}, workerLoop: {}, maxInboundMessageSize: {}
2023-07-09 09:57:50 INFO main - ServiceProvider loaded no AccessValidator, using default org.apac
2023-07-09 09:57:50 INFO main - Server is running in TLS permissive mode
2023-07-09 09:57:50 INFO main - Using OpenSSL provider
2023-07-09 09:57:50 INFO main - SSLContext created for server
2023-07-09 09:57:50 INFO main - The broker[broker-a, 192.168.192.115:10911] boot success, seriali
2023-07-09 09:57:50 INFO main - user specified name server address: localhost:9876
2023-07-09 09:57:50 INFO main - user specified name server address: localhost:9876
2023-07-09 09:57:50 INFO main - grpc server start successfully.
2023-07-09 09:57:50 INFO main - Sun Jul 09 09:57:50 CST 2023 rocketmq-proxy startup successfully
```

当在日志中看到类似 The broker[broker-a, 192.168.192.115:10911] boot success 时，说明该 Broker 启动成功。

### 9.3.4 发送/接收消息测试

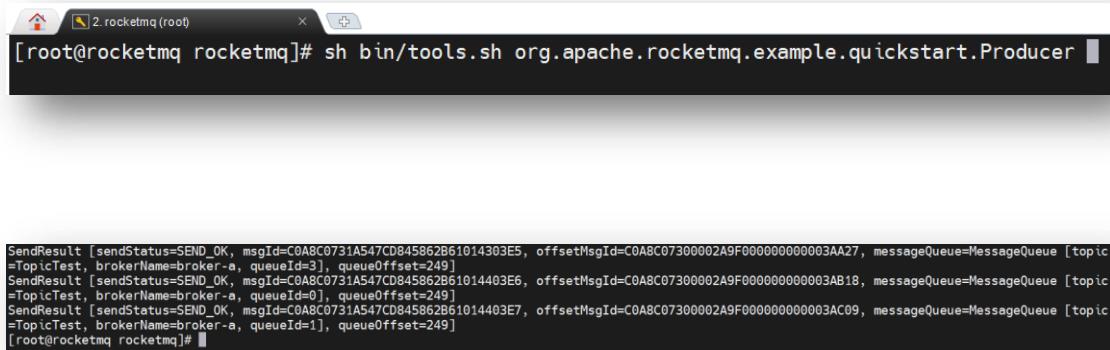
#### (1) 定义系统变量

无论是 Producer 还是 Consumer，其都需要指定 NameServer 的地址，然后从中获取到路由表才能运行。而 bin/tools.sh 命令是 Producer 与 Consumer 快速启动命令，都是从系统变量 NAMESRVADDR 中获取 NameServer 地址的，所以需要首先为 NAMESRVADDR 变量赋值。



```
[root@rocketmq rocketmq]# export NAMESRV_ADDR=localhost:9876
[root@rocketmq rocketmq]#
```

#### (2) 发送消息



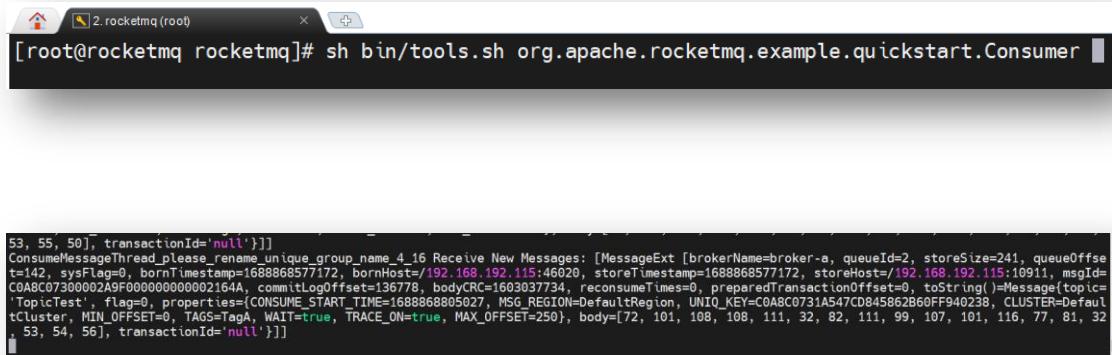
```
[root@rocketmq rocketmq]# sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
```

```
SendResult [sendStatus=SEND_OK, msgId=C0A8C0731A547CD845862B61014303E5, offsetMsgId=C0A8C07300002A9F00000000003AA27, messageQueue=MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=3], queueOffset=249]
SendResult [sendStatus=SEND_OK, msgId=C0A8C0731A547CD845862B61014403E6, offsetMsgId=C0A8C07300002A9F00000000003AB18, messageQueue=MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=0], queueOffset=249]
SendResult [sendStatus=SEND_OK, msgId=C0A8C0731A547CD845862B61014403E7, offsetMsgId=C0A8C07300002A9F00000000003AC09, messageQueue=MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=1], queueOffset=249]
[root@rocketmq rocketmq]#
```

该命令时会发送大量的消息给 broker。其会自动停止发送。消息的 topic 为 TopicTest。

## (3) 接收消息



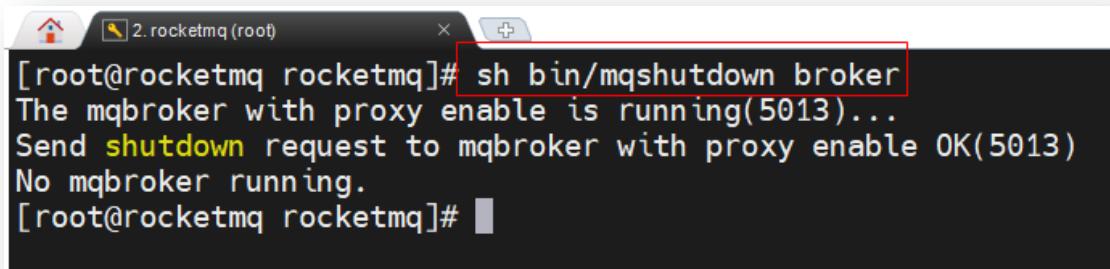
```
[root@rocketmq rocketmq]# sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
[53, 55, 50], transactionId='null'}]
ConsumeMessageThread_please_rename_unique_group_name_4_16 Receive New Messages: [MessageExt [brokerName=broker-a, queueId=2, storeSize=241, queueOffset=142, sysFlag=0, bornTimestamp=1688868577172, bornHost=/192.168.192.115:46020, storeTimestamp=1688868577172, storeHost=/192.168.192.115:10911, msgId=C0ABC07300002A9F000000000002164A, commitLogOffset=136778, bodyCRC=1603037734, reconsumeTimes=0, preparedTransactionOffset=0, toString()=Message{topic='TopicTest', flag=0, properties={CONSUME_START_TIME=1688868805027, MSG_REGION=DefaultRegion, UNIQ_KEY=C0ABC0731A547CD845862B60FF94023B, CLUSTER=DefaultCluster, MIN_OFFSET=0, TAGS=TagA, WAIT=true, TRACE_ON=true, MAX_OFFSET=250}, body=[72, 101, 108, 108, 111, 32, 82, 111, 99, 107, 101, 116, 77, 81, 32, 53, 54, 56], transactionId='null'}]]
```

当执行了以上命令后会看到大量的消息被消费，直到所有消息被消费完毕，然后等待后续消息的到来。可使用 **Ctrl + C** 终止执行。

## 9.3.5 关闭 Server

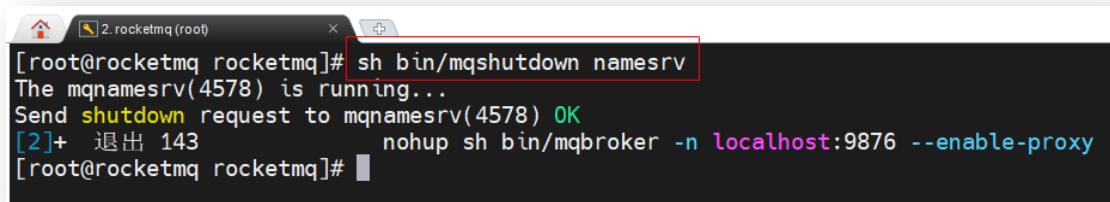
无论是关闭 name server 还是 broker，都是使用 `bin/mqshutdown` 命令。

## (1) 关闭 broker



```
[root@rocketmq rocketmq]# sh bin/mqshutdown broker
The mqbroker with proxy enable is running(5013)...
Send shutdown request to mqbroker with proxy enable OK(5013)
No mqbroker running.
[root@rocketmq rocketmq]#
```

## (2) 关闭 Name Server



```
[root@rocketmq rocketmq]# sh bin/mqshutdown namesrv
The mqnamesrv(4578) is running...
Send shutdown request to mqnamesrv(4578) OK
[2]+ 退出 143      nohup sh bin/mqbroker -n localhost:9876 --enable-proxy
[root@rocketmq rocketmq]#
```

## 9.4 Stream RocketMQ 微服务

### 9.4.1 生产者应用 09-stream-rocketmq-producer-8081

#### (1) 创建工程

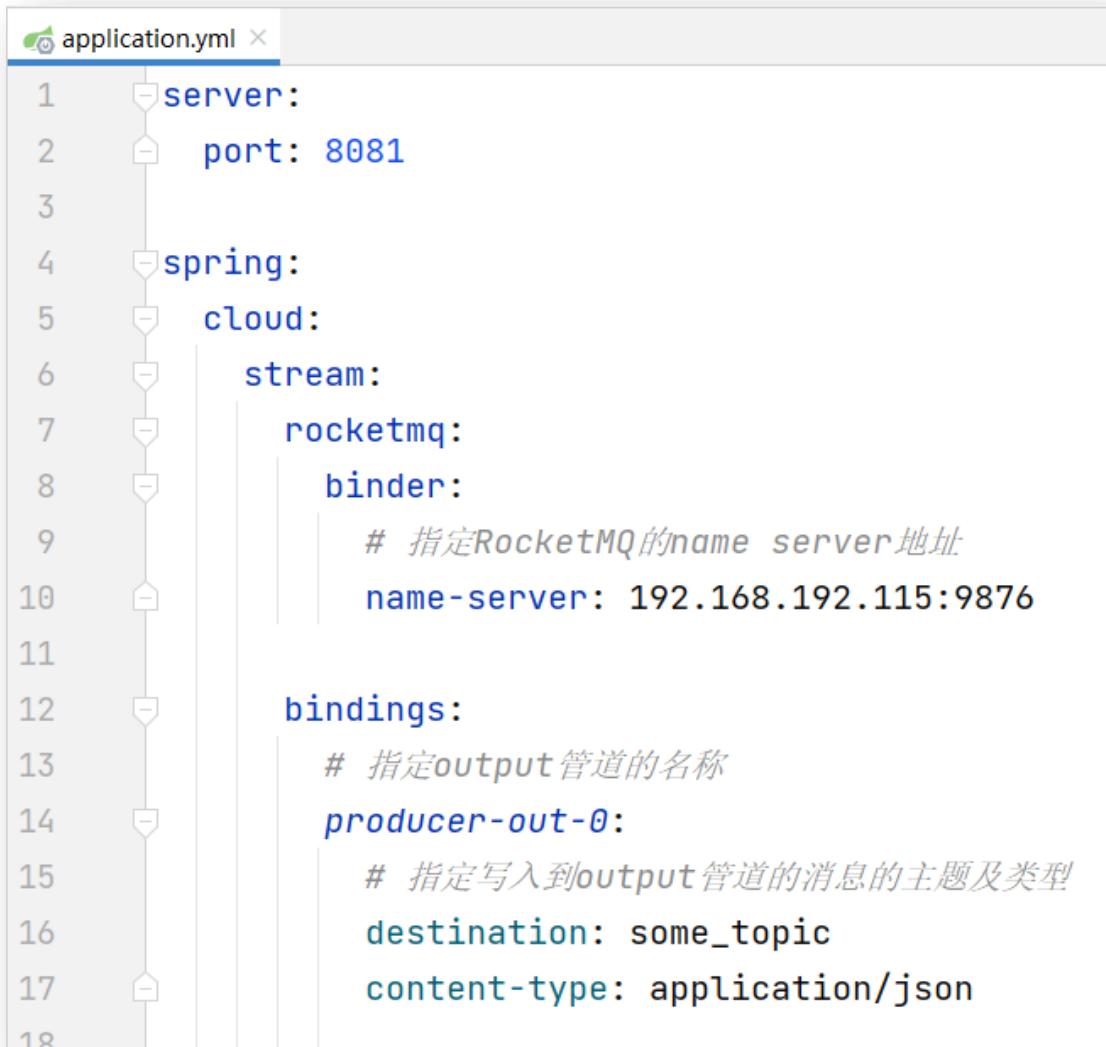
任意复制前面的一个提供者或消费者工程，将其中的除启动类之外的其它代码全部删除，并命名为 09-stream-rocketmq-producer-8081。

#### (2) 导入依赖

仅需再添加一个 Spring Cloud Stream Kafka 相关的依赖即可。

```
<!--stream-rocketmq 依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rocketmq</artifactId>
</dependency>
```

## (3) 修改配置文件



```
application.yml
1 server:
2   port: 8081
3
4 spring:
5   cloud:
6     stream:
7       rocketmq:
8         binder:
9           # 指定RocketMQ的name server地址
10          name-server: 192.168.192.115:9876
11
12         bindings:
13           # 指定output管道的名称
14             producer-out-0:
15               # 指定写入到output管道的消息的主题及类型
16               destination: some_topic
17               content-type: application/json
18
```

## (4) 定义 SomeController

```
@RestController
public class SomeController {

    1 usage
    @Autowired
    private StreamBridge bridge;

    no usages
    @GetMapping("/message/send")
    public String sendMsg(String msg) {
        // 构建MessageHeaders
        Map<String, Object> headers = new HashMap<>();
        headers.put(MessageConst.PROPERTY_TAGS, "some_tag");
        MessageHeaders msgHeaders = new MessageHeaders(headers);

        // 创建Message
        Message<String> message = MessageBuilder.createMessage(msg, msgHeaders);
        // 将消息写入到管道
        bridge.send("producer-out-0", message);

        return JSON.toJSONString(message);
    }
}
```

## (5) 创建启动类

启动类无需做任何处理。

```
@SpringBootApplication
public class StreamProducer8081 {

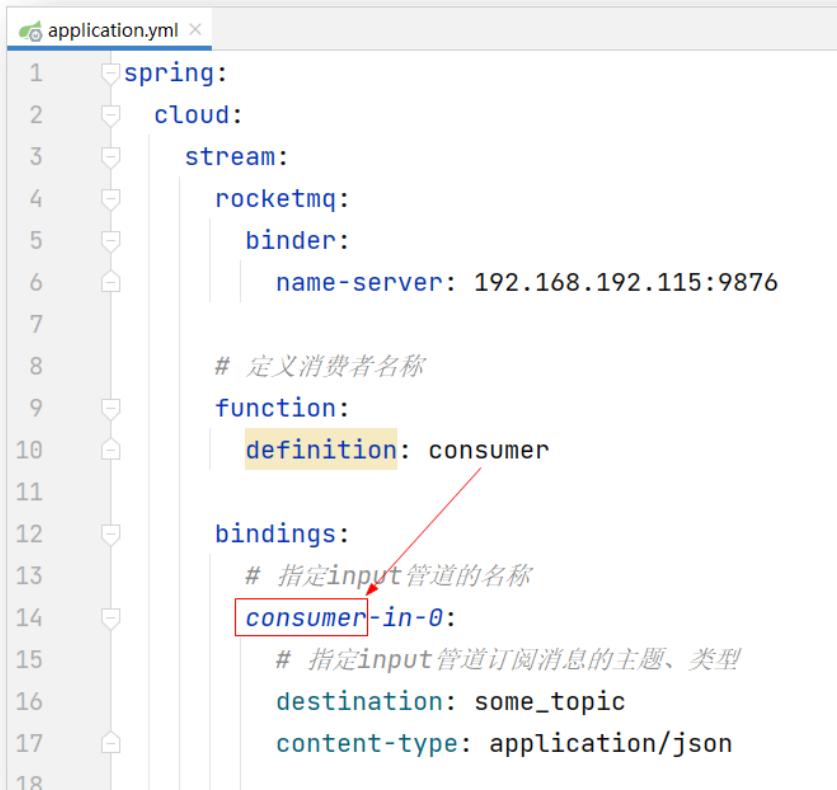
    no usages
    public static void main(String[] args) {
        SpringApplication.run(StreamProducer8081.class, args);
    }
}
```

## 9.4.2 消费者应用 09-stream-rocketmq-consumer-8080

### (1) 创建工程

复制 09-stream-rocketmq-producer-8081 应用，在其基础上进行修改。  
删除其生产者类与控制器类。

### (2) 修改配置文件



```
application.yml
1 spring:
2   cloud:
3     stream:
4       rocketmq:
5         binder:
6           name-server: 192.168.192.115:9876
7
8         # 定义消费者名称
9         function:
10          definition: consumer
11
12         bindings:
13           # 指定input管道的名称
14           consumer-in-0:
15             # 指定input管道订阅消息的主题、类型
16             destination: some_topic
17             content-type: application/json
18
```

### (3) 创建启动类

```
@SpringBootApplication
public class StreamConsumer8080 {

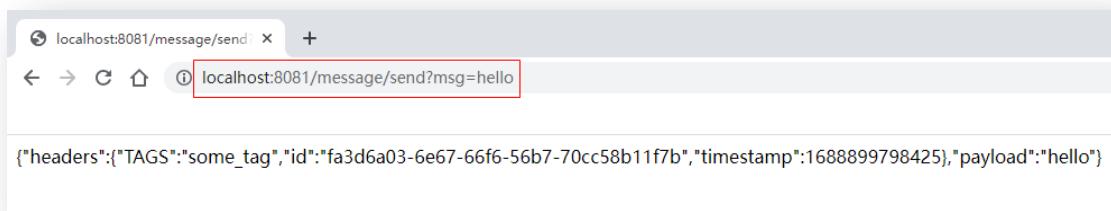
    no usages
    public static void main(String[] args) {
        SpringApplication.run(StreamConsumer8080.class, args);
    }

    no usages
    @Bean
    public Consumer<Message<String>> consumer() {
        return msg -> {
            long time = System.currentTimeMillis();
            System.out.println(time + "接收到消息: " + msg.getPayload());
        };
    }

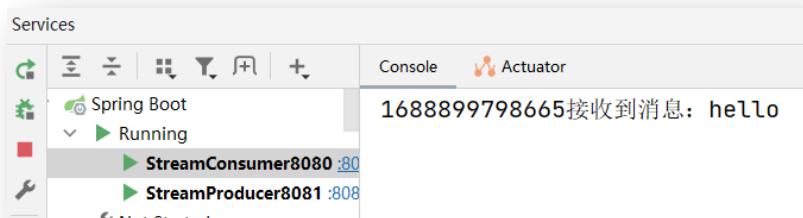
}
```

#### 9.4.3 测试

在启动 RocketMQ 后，再分别启动前面定义的 producer 与 consumer 应用。然后在浏览器对 producer 进行访问。



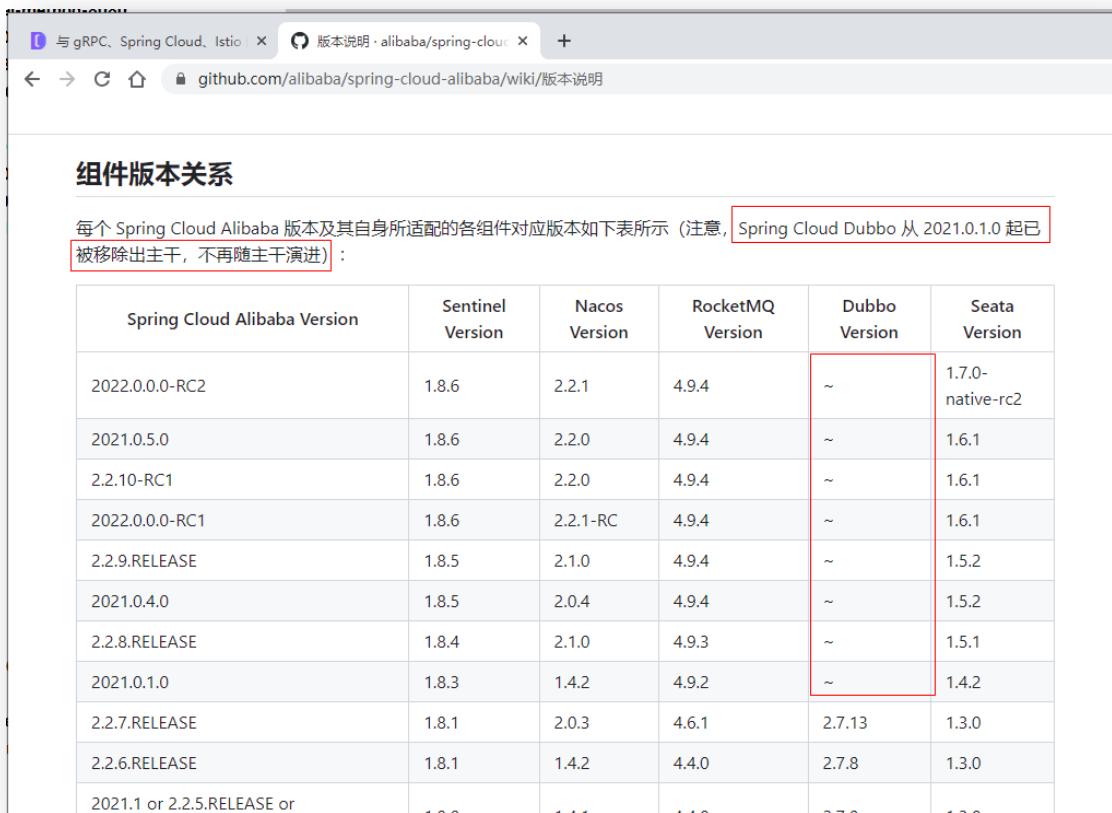
此时查看 consumer 控制台，可以看到其接收到的消息。



## 第10章 Dubbo Spring Cloud

Dubbo 从 3.0 开始已经不再与 Spring Cloud Alibaba 进行集成了。原因主要是与其发展方向定位有关。

### 10.1 Dubbo 定位变迁



The screenshot shows a browser window displaying the Spring Cloud Alibaba version compatibility table. The table lists various Spring Cloud Alibaba versions along with their compatible components like Sentinel, Nacos, RocketMQ, Dubbo, and Seata.

Spring Cloud Alibaba Version	Sentinel Version	Nacos Version	RocketMQ Version	Dubbo Version	Seata Version
2022.0.0.0-RC2	1.8.6	2.2.1	4.9.4	~	1.7.0-native-rc2
2021.0.5.0	1.8.6	2.2.0	4.9.4	~	1.6.1
2.2.10-RC1	1.8.6	2.2.0	4.9.4	~	1.6.1
2022.0.0.0-RC1	1.8.6	2.2.1-RC	4.9.4	~	1.6.1
2.2.9.RELEASE	1.8.5	2.1.0	4.9.4	~	1.5.2
2021.0.4.0	1.8.5	2.0.4	4.9.4	~	1.5.2
2.2.8.RELEASE	1.8.4	2.1.0	4.9.3	~	1.5.1
2021.0.1.0	1.8.3	1.4.2	4.9.2	~	1.4.2
2.2.7.RELEASE	1.8.1	2.0.3	4.6.1	2.7.13	1.3.0
2.2.6.RELEASE	1.8.1	1.4.2	4.4.0	2.7.8	1.3.0
2021.1 or 2.2.5.RELEASE or	1.8.0	1.4.1	4.4.0	2.7.9	1.3.0

最初的 Dubbo 定位是微服务框架，与 Spring Cloud 定位相同。但没有撼动了 Spring Cloud 的江湖地位，搞得自己停止维护了。

后来 Spring Cloud Alibaba 大火起来，Dubbo 搭乘了 Spring Cloud Alibaba 的快车，解决其通信效率低下的问题，Dubbo 又火爆了起来。此时 Dubbo 的定位是 RPC 通信框架。

随着 Dubbo 的逐步完善，Dubbo 又定位到了微服务框架。该观点在 Dubbo 官网中的文章“与 gRPC、Spring Cloud、Istio 的关系”中有详细说明。特别是 Dubbo 与 gRPC 的关系描述上更加说明了此观点。

## Dubbo 与 gRPC

Dubbo 与 gRPC 最大的差异在于两者的定位上：

- **gRPC 定位为一款 RPC 框架**, Google 推出它的核心目标是定义云原生时代的 rpc 通信规范与标准实现；
- **Dubbo 定位是一款微服务开发框架**, 它侧重解决微服务实践从服务定义、开发、通信到治理的问题，因此 Dubbo 同时提供了 RPC 通信、与应用开发框架的适配、服务治理等能力。

## 10.2 Dubbo 与 Spring Cloud 的重大区别

在 Dubbo 官网中的文章“与 gRPC、Spring Cloud、Istio 的关系”中有详细说明了 Dubbo 的 Spring Cloud 的区别，但有一个区别是它们的重大区别，也正因为此区别，才使得 Dubbo 有底气不再作为“小弟”去集成 Spring Cloud Alibaba，而是选择了与 Spring Cloud Alibaba“平起平坐”。

这个重大区别就是，Spring Cloud 只适用于小规模的微服务应用，而对于超大规模需求会出现各种各样的问题，但 Dubbo 可以完美处理。原因就是，Dubbo 是由阿里内部代码抽象出来的，是经历过“高并发大场面”的。

Spring Cloud 的问题有：

- 只提供抽象模式的定义不提供官方稳定实现，开发者只能寻求类似 Netflix、Alibaba、Azure 等不同厂商的实现套件，而每个厂商支持的完善度、稳定性、活跃度各异
- 有微服务全家桶却不是能拿来就用的全家桶，demo 上手容易，但落地推广与长期使用的成本非常高
- 欠缺服务治理能力，尤其是流量管控方面如负载均衡、流量路由方面能力都比较弱
- 编程模型与通信协议绑定 HTTP，在性能、与其他 RPC 体系互通上存在障碍
- 总体架构与实现只适用于小规模微服务集群实践，当集群规模增长后就会遇到地址推送效率、内存占用等各种瓶颈的问题，但此时迁移至其他体系却很难实现
- 很多微服务实践场景的问题需要用户独自解决，比如优雅停机、启动预热、服务测试，再比如双注册、双订阅、延迟注册、服务按分组隔离、集群容错等

而以上这些点，都是 Dubbo 的优势所在：

- 完全支持 Spring & Spring Boot 开发模式，同时在服务发现、动态配置等基础模式上提供与 Spring Cloud 对等的能力。
- 是企业级微服务实践方案的整体输出，Dubbo 考虑到了企业微服务实践中会遇到的各种问题如优雅上下线、多注册中心、流量管理等，因此其在生产环境的长期维护成本更低
- 在通信协议和编码上选择更灵活，包括 rpc 通信层协议如 HTTP、HTTP/2(Triple、gRPC)、TCP 二进制协议、rest 等，序列化编码协议 Protobuf、JSON、Hessian2 等，支持单端口多协议。
- Dubbo 从设计上突出服务治理能力，如权重动态调整、标签路由、条件路由等，支持 Proxyless 等多种模式接入 Service Mesh 体系
- 高性能的 RPC 协议编码与实现，
- Dubbo 是在超大规模微服务集群实践场景下开发的框架，可以做到百万实例规模的集群水平扩容，应对集群增长带来的各种问题
- Dubbo 提供 Java 外的多语言实现，使得构建多语言异构的微服务体系成为可能

其用于“超大规模”需求还可以从官网首页上看出来：其现在已经用于淘宝，取代了

HSF 框架。

