1.Spring Cloud

1.什么是微服务

单体架构：所有业务模块全部打包到一个工程

单体应用各个业务模块耦合度太高，一个出问题，所有都无法使用，如何解决?微服务应运而生

微服务：将一个单体应用拆分成若干个小型的服务，协同完成系统功能的一种架构模式

Spring Cloud

实现微服务架构的框架，Spring全家桶的一部分，基于SpringBoot

SpringBoot快速构建工程的框架，SpringCloud快速构建微服务工程的框架

服务治理、配置中心、消息总线、负载均衡、熔断器、数据监控

Spring Cloud Alibaba

Spring Boot-->Spring Cloud--> Spring Cloud Alibaba

Spring Cloud Alibaba 2.2.1

Spring Cloud Hoxton.SR3

Spring Boot 2.3.0

搭建SpringCloud

1、创建 Spring Boot 工程，选择常用的 Lombok，Spring Cloud Alibaba 还没有完全集成到 Spring Boot Initialzr 中，我们需要手动添加。

Spring Boot ---》Spring Cloud ---》Spring Cloud Alibaba

Spring Boot 版本修改为 2.3.0，因为高版本有 bug。

pom.xml 中添加。

```xml
<dependencyManagement>
    <dependencies>
        <!-- Spring Cloud Hoxton -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- Spring Cloud Alibaba -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.2.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2.Nacos 服务注册

服务注册，这里我们使用 Nacos 的服务注册，下载对应版本的 Nacos：https://github.com/alibaba/nacos/releases

解压，启动服务。

启动 nacos-server

双击 bin 中的 startup.cmd 文件

访问 http://localhost:8848/nacos/

使用默认的 nacos/nacos 进行登录

Nacos 搭建成功，接下来注册服务。

在父工程路径下创建子工程，让子工程继承父工程的环境依赖

在子工程的中换为父工程的

```
<parent>
    <groupId>com.ishang</groupId>
    <artifactId>myspringcloud_001</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</parent>
```

在子工程pom.xml 中添加 nacos 发现组件。

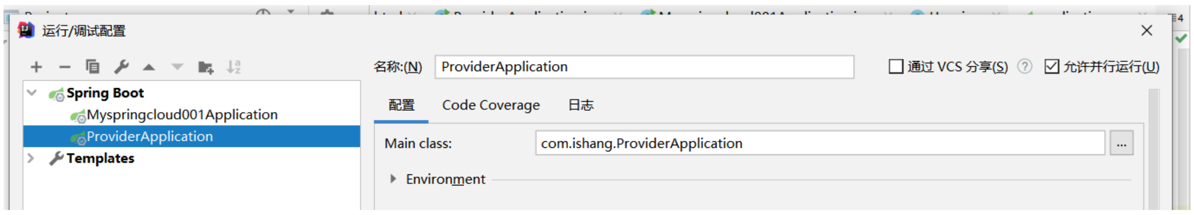
```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

在子工程application.yml 中配置

```
spring:
  cloud:
    nacos:
      discovery:
        # 指定nacos server地址
        server-addr: localhost:8848
  application:
    name: provider
```

需要引入 spring-web 依赖才能实现 Nacos 注册

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.3.0.RELEASE</version>
</dependency>
```


运行/调试配置

- Spring Boot
  - Myspringcloud001Application
  - ProviderApplication
- Templates

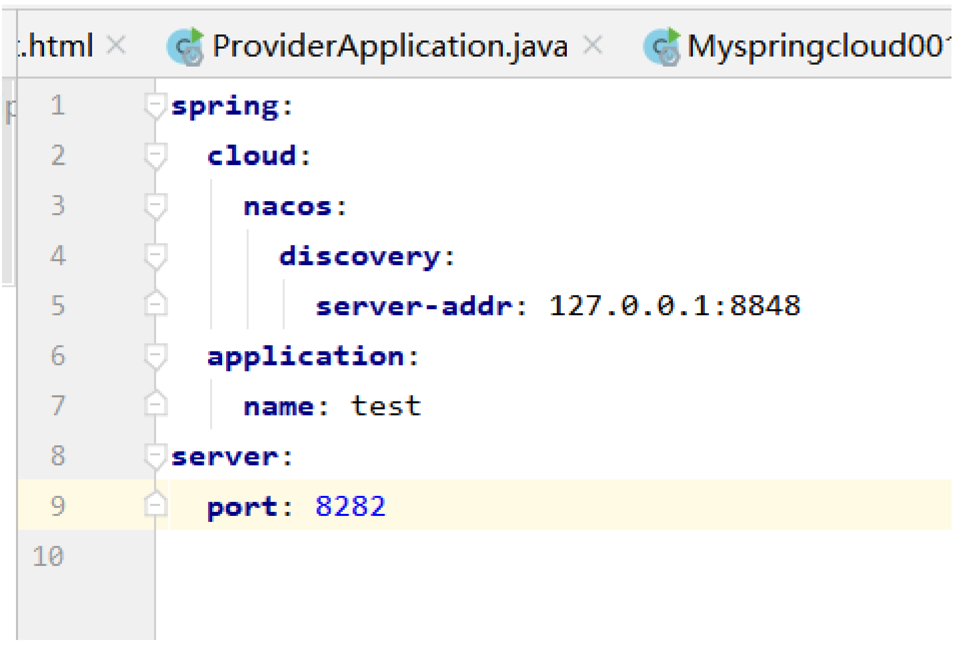
名称:(N) ProviderApplication

☐ 通过 VCS 分享(S) ☑ 允许并行运行(U)

配置 | Code Coverage | 日志

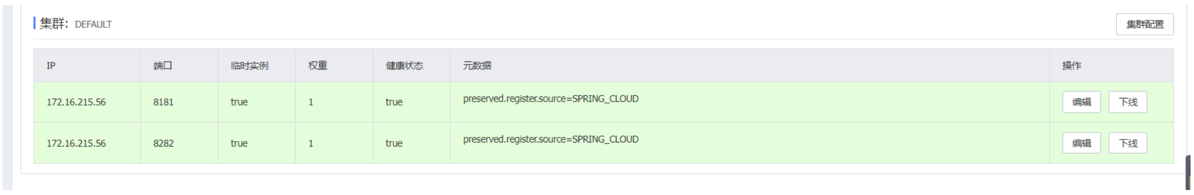
Main class: com.ishang.ProviderApplication

Environment



```yaml
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
  application:
    name: test
server:
  port: 8282
```

修改端口号即可运行多个实例

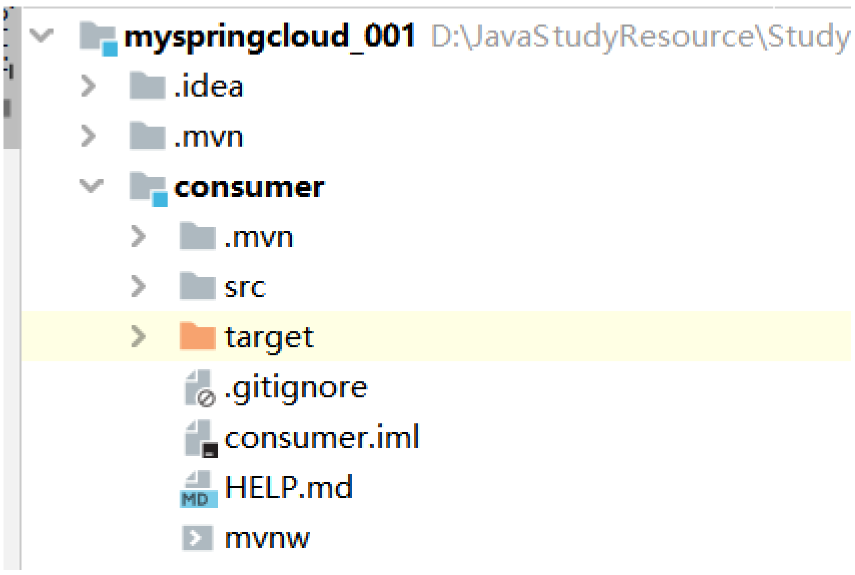

集群: DEFAULT

IP	端口	临时实例	权重	健康状态	元数据	操作
172.16.215.56	8181	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线
172.16.215.56	8282	true	1	true	preserved.register.source=SPRING_CLOUD	编辑 下线

2.1Nacos服务发现和调用

同样的方式先将服务注册进nacos

1.创建Module-----consumer



```
myspringcloud_001 D:\JavaStudyResource\Study
  .idea
  .mvn
  consumer
    .mvn
    src
    target
    .gitignore
    consumer.iml
    HELP.md
    mvnw
```

2.导入nacos依赖

```xml
<!--nacos组件，服务注册与发现-->
<dependency>
   <groupId>com.alibaba.cloud</groupId>
   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

将改为父工程

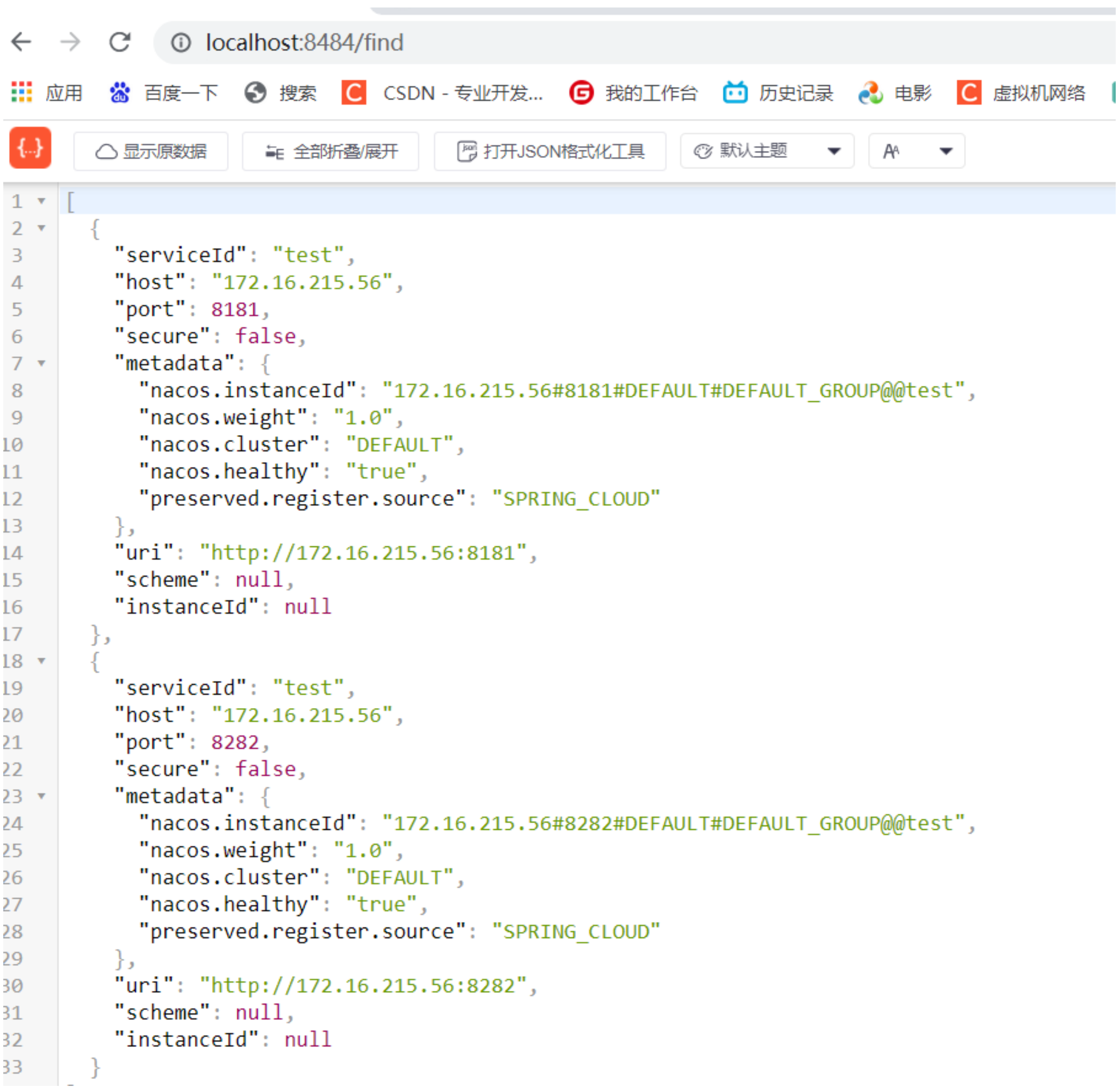
```xml
<parent>
   <groupId>com.ishang</groupId>
   <artifactId>myspringcloud_001</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>
```

3.配置application.yml

```yaml
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  application:
    name: consumer
```

4.从consumer服务中调用provider服务

```java
import java.util.List;

@RestController
public class FindProviderController {

    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping("/find")
    public List<ServiceInstance> find(){
        List<ServiceInstance> test = this.discoveryClient.getInstances("test");
        return  test;
    }
}
```



```
[
  {
    "serviceId": "test",
    "host": "172.16.215.56",
    "port": 8181,
    "secure": false,
    "metadata": {
      "nacos.instanceId": "172.16.215.56#8181#DEFAULT#DEFAULT_GROUP@@test",
      "nacos.weight": "1.0",
      "nacos.cluster": "DEFAULT",
      "nacos.healthy": "true",
      "preserved.register.source": "SPRING_CLOUD"
    },
    "uri": "http://172.16.215.56:8181",
    "scheme": null,
    "instanceId": null
  },
  {
    "serviceId": "test",
    "host": "172.16.215.56",
    "port": 8282,
    "secure": false,
    "metadata": {
      "nacos.instanceId": "172.16.215.56#8282#DEFAULT#DEFAULT_GROUP@@test",
      "nacos.weight": "1.0",
      "nacos.cluster": "DEFAULT",
      "nacos.healthy": "true",
      "preserved.register.source": "SPRING_CLOUD"
    },
    "uri": "http://172.16.215.56:8282",
    "scheme": null,
    "instanceId": null
  }
```

2.2Nacos服务治理

包括

- 服务注册 如上图步骤
- 服务发现

服务发现利用DiscoveryClient

通过 discoveryClient 发现注册到 nacos 中的 provider 服务。

再通过Restemplate进行调用

服务提供者（被调用）：provider

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class IndexController {
    @Value("${server.port}")
    private  String port;

    @RequestMapping("/find")
    public  String index(){
        return "该端口号是"+this.port;
    }

}
```

服务提供者：consumer

1.创建配置类RestemplateConfiguration

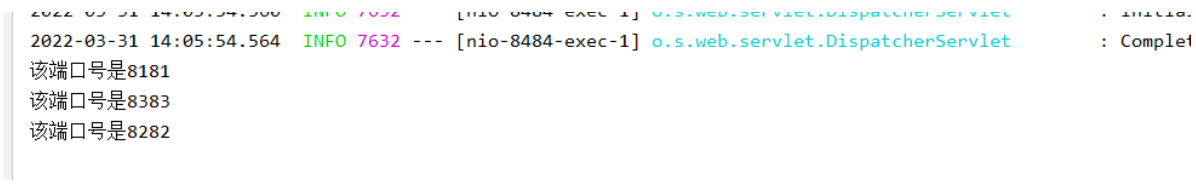
将RestTemplate注入到IOC容器中

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class ResttemplateConfiguration {

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

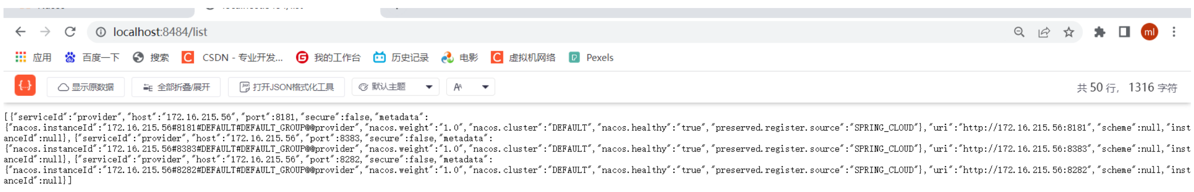
2.将ioc容器中的Restemplate注入，调用者在nacos中找到提供者的url，利用resttemplate实现调用

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.List;

@RestController
public class FindProviderController {

    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    @RequestMapping("/list")
    public List<ServiceInstance> list() {
        List<ServiceInstance> list =
this.discoveryClient.getInstances("provider");
```

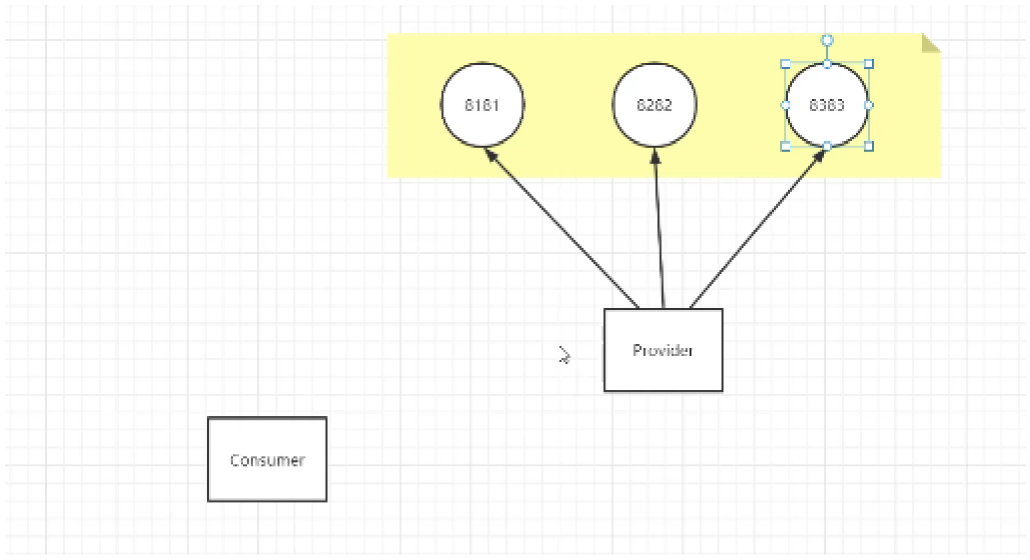
```
        for (ServiceInstance serviceInstance : list) {
            String host = serviceInstance.getHost();
            int port = serviceInstance.getPort();
            String url = "http://" + host + ":" + port+ "/find";
//              将调用的路径传进去，返回什么样的数据 xxx.class
            String result = this.restTemplate.getForObject(url, String.class);
//              在控制台输出被调用服务接口的内容
            System.out.println(result);
        }

        return list;
    }
}
```



```
2022-03-31 14:05:54.564  INFO 7632 --- [nio-8484-exec-1] o.s.web.servlet.DispatcherServlet        : Complet
该端口号是8181
该端口号是8383
该端口号是8282
```



```
[{"serviceId":"provider","host":"172.16.215.56","port":8181,"secure":false,"metadata":
{"nacos.instanceId":"172.16.215.56#8181#DEFAULT#DEFAULT_GROUP@@provider","nacos.weight":"1.0","nacos.cluster":"DEFAULT","nacos.healthy":"true","preserved.register.source":"SPRING_CLOUD"},"uri":"http://172.16.215.56:8181","scheme":null,"inst
anceId":null},{"serviceId":"provider","host":"172.16.215.56","port":8383,"secure":false,"metadata":
{"nacos.instanceId":"172.16.215.56#8383#DEFAULT#DEFAULT_GROUP@@provider","nacos.weight":"1.0","nacos.cluster":"DEFAULT","nacos.healthy":"true","preserved.register.source":"SPRING_CLOUD"},"uri":"http://172.16.215.56:8383","scheme":null,"inst
anceId":null},{"serviceId":"provider","host":"172.16.215.56","port":8282,"secure":false,"metadata":
{"nacos.instanceId":"172.16.215.56#8282#DEFAULT#DEFAULT_GROUP@@provider","nacos.weight":"1.0","nacos.cluster":"DEFAULT","nacos.healthy":"true","preserved.register.source":"SPRING_CLOUD"},"uri":"http://172.16.215.56:8282","scheme":null,"inst
anceId":null}]
```



3.Ribbon负载均衡

3.1轮询策略

Ribbon 负载均衡算法默认是轮询，交替访问。

1.在Resttemplate中添加@LoadBalanced

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
```

```java
public class ResttemplateConfiguration {

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

2.不需要再获取实例，直接将服务者的名字传入，自动解析

```java
@RestController
public class FindProviderController {
    @Autowired
    private DiscoveryClient discoveryClient;
    @Autowired
    private RestTemplate restTemplate;
    @RequestMapping("/list")
    public List<Account> list() {
        return this.restTemplate.getForObject("http://provider/finAll",
List.class);
    }

    @RequestMapping("/index")
    public String index(){
        return
this.restTemplate.getForObject("http://provider/find",String.class);
    }
}
```

服务提供者

```java
@RestController
public class IndexController {

    @Value("${server.port}")
    private  String port;

    @Autowired
    private AccountMapper accountMapper;
    @RequestMapping("/find")
    public  String index(){
        return "该端口号是"+this.port;
    }

    @RequestMapping("/finAll")
    public List<Account> findAll(){
        System.out.println("当前调用的是"+this.port);
        return this.accountMapper.selectList(null);
    }

}
```

3.2随机策略

在服务调用者的application.yml中添加服务提供者的

修改为随机，在 consumer 的 application.yml 添加配置随机规则即可。注意是谁去使用ribbon谁application.yml去加。这里是consumer调用provider，所以在consumer里面加

```
#将服务提供者的名字传进来，设置随机调用
provider:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

3.3基于Nacos权重的负载均衡

1、创建 NacosWeightedRule 类定义基于权重的负载均衡

```
package com.ishang.configuration;


import com.alibaba.cloud.nacos.NacosDiscoveryProperties;
import com.alibaba.cloud.nacos.ribbon.NacosServer;
import com.alibaba.nacos.api.exception.NacosException;
import com.alibaba.nacos.api.naming.NamingService;
import com.alibaba.nacos.api.naming.pojo.Instance;
import com.netflix.client.config.IClientConfig;
import com.netflix.loadbalancer.AbstractLoadBalancerRule;
import com.netflix.loadbalancer.BaseLoadBalancer;
import com.netflix.loadbalancer.ILoadBalancer;
import com.netflix.loadbalancer.Server;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;

@Slf4j
public class NacosWeightedRule extends AbstractLoadBalancerRule {

    @Autowired
    private NacosDiscoveryProperties nacosDiscoveryProperties;

    @Override
    public void initWithNiwsConfig(IClientConfig iClientConfig) {
        //读取配置文件
    }

    @Override
    public Server choose(Object o) {
        ILoadBalancer loadBalancer = this.getLoadBalancer();
        BaseLoadBalancer baseLoadBalancer = (BaseLoadBalancer) loadBalancer;
        //获取要请求的微服务名称
        String name = baseLoadBalancer.getName();
        //获取服务发现的相关API
        NamingService namingService =
nacosDiscoveryProperties.namingServiceInstance();
        try {
            Instance instance = namingService.selectOneHealthyInstance(name);
            log.info("选择的实例是port={},instance=
{}",instance.getPort(),instance);
            return new NacosServer(instance);
        } catch (NacosException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

2.在application.yml中配置

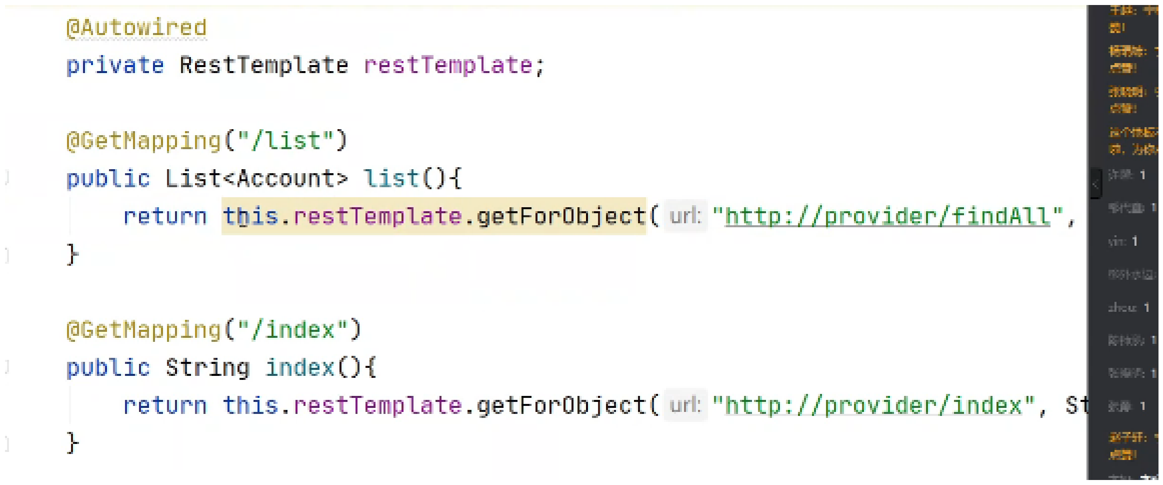
```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  application:
    name: consumer

server:
  port: 8484

#将服务提供者的名字传进来
provider:
  ribbon:
    NFLoadBalancerRuleClassName: com.ishang.configuration.NacosWeightedRule
```

基于权重的策略可以在某台服务器并发量过高的时候，可以争对某个服务调整权重，减少对这台服务的请求量，保护系统，防止服务器挂机。

合理分配服务器对于请求的处理，充分利用资源

4.Feign 声明式接口调用

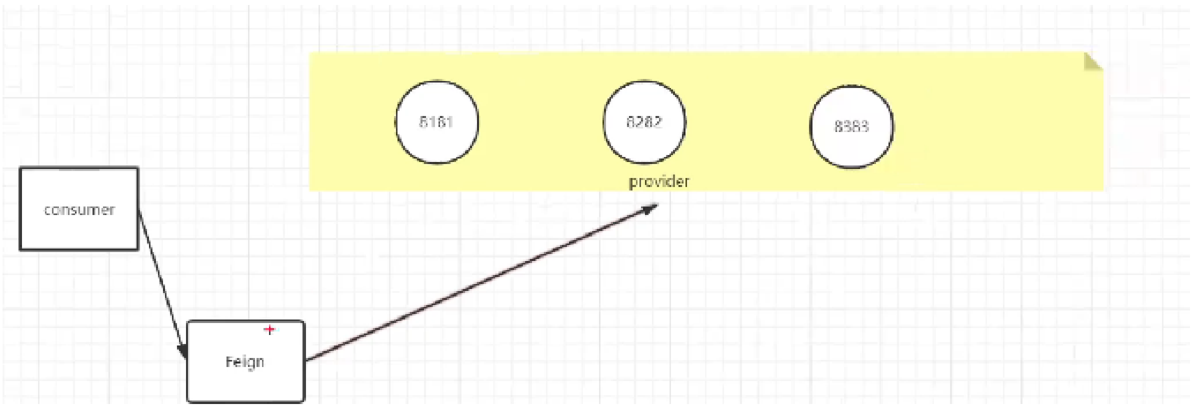
使用RestTemplate调用远程接口，将调用方法写在Controller层实际上代码可读性性差，维护性也比较差。（因为在开发的时候，controller主要是调用service层的接口方法，面向接口编程，而不是把实现写在controller层，应该是封装在接口里面，直接调用接口，才符合面向接口编程的规范）



```
@Autowired
private RestTemplate restTemplate;

@GetMapping("/list")
public List<Account> list(){
    return this.restTemplate.getForObject( url: "http://provider/findAll",
}

@GetMapping("/index")
public String index(){
    return this.restTemplate.getForObject( url: "http://provider/index", St
}
```

因此可以通过Feign将RestTemplate封装在接口里面，直接调用Feign接口即可

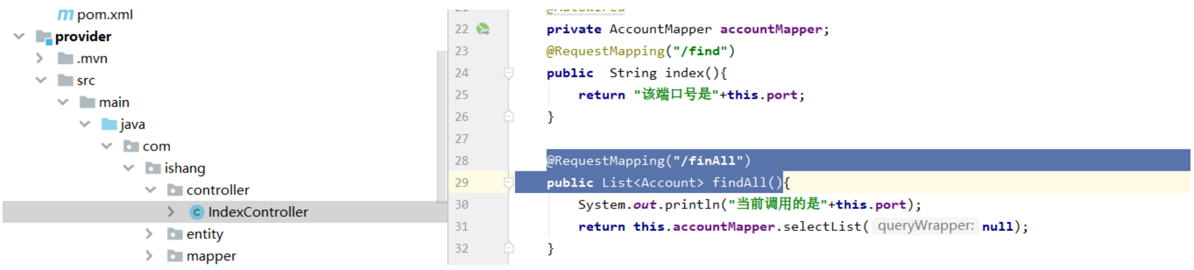
Feign是基于Ribbon负载均衡实现的一种声明式接口调用机制，同时他更加灵活，使用起来也更加简单，取代Ribbon+RestTemplate的方式来实现基于接口调用的负载均衡；比Ribbon使用起来更加简便，只需要创建接口并添加相关注解配置，即可实现服务消费的负载均衡



1.pom.xml导入依赖

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

2.创建接口Feign，将provider服务中的方法进行映射



```java
    private AccountMapper accountMapper;
    @RequestMapping("/find")
    public  String index(){
        return "该端口号是"+this.port;
    }

    @RequestMapping("/finAll")
    public List<Account> findAll(){
        System.out.println("当前调用的是"+this.port);
        return this.accountMapper.selectList( queryWrapper: null);
    }
```

```java
package com.ishang.feign;

import com.ishang.entity.Account;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;
import java.util.List;

@FeignClient("provider")
public interface Feign {

    @RequestMapping("/find")
    public  String index();

    @RequestMapping("/finAll")
    public List<Account> findAll();
}
```

3.在consumerApplication中添加@EnableFeignClients

```java
package com.ishang;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableFeignClients
```

```
public class ConsumerApplication {

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

}
```

4.在controller进行注入调用

```
package com.ishang.controller;


import com.ishang.entity.Account;
import com.ishang.feign.Feign;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

@RestController
public class FindProviderController {

    @Autowired
    private Feign feign;

    @RequestMapping("/list")
    public List<Account> list() {
        return  this.feign.findAll();
    }

    @RequestMapping("/index")
    public String index(){
        return this.feign.index();
    }
}
```

5.Sentinel 服务限流降级

 雪崩效应：

高并发系统中，因为一个小问题而引发的系统崩溃

比如B调用A，A服务挂机后，B调A的线程一直存在得不到处理，随着堆积越来越多，B服务的内存就会崩溃，随即调用C服务的一系列服务也会崩溃，造成雪崩效应

解决方案：

1.设置线程超时，当某个请求在特定的时间内无法调用，则直接释放线程（比如一直不买，就赶出去）

2.设置限流,某个微服务到达访问上限之后，其他请求就暂时无法访问该服务（比如只允许50个人进店，50个人没走，后面的人就没办法进店）

3.熔断器，这是目前微服务中比较主流的解决方案，如Hystrix，相当于家里的保险丝，如果电流过大，为了保护家电设备，字段稍短保险丝，断点，阻断电流

1.pom.xml导入依赖

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2.application配置 （在被调用者中使用 provider ）

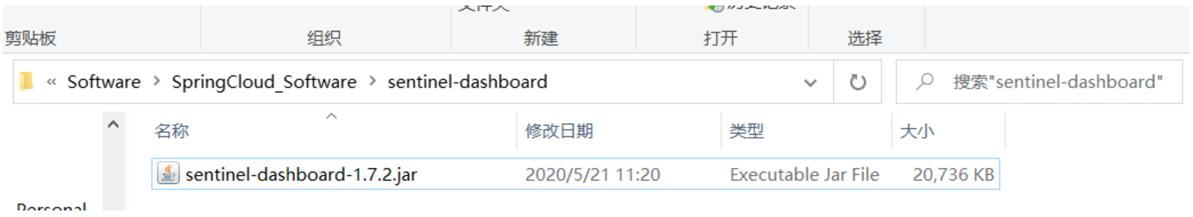
```
management:
  endpoints:
    web:
      exposure:
        include: '*'
spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080
```

3、下载 Sentinel 控制台

https://github.com/alibaba/Sentinel/releases

4、解压，启动 sentinel-dashboard-1.7.2.jar，在所在包cmd后输入

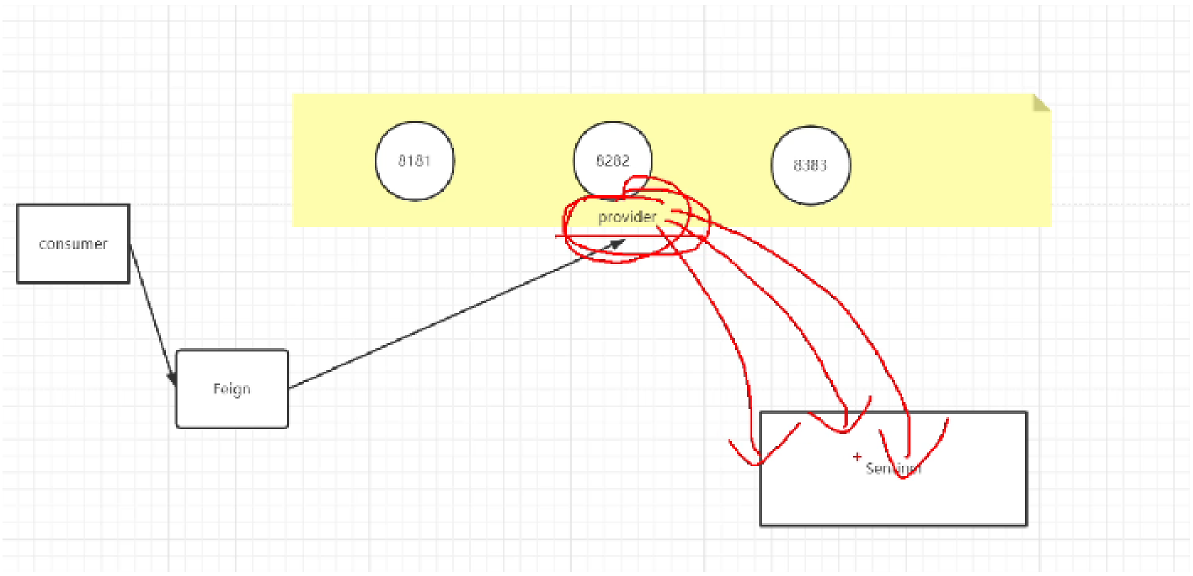
命令：java -jar sentinel-dashboard-1.7.2.jar



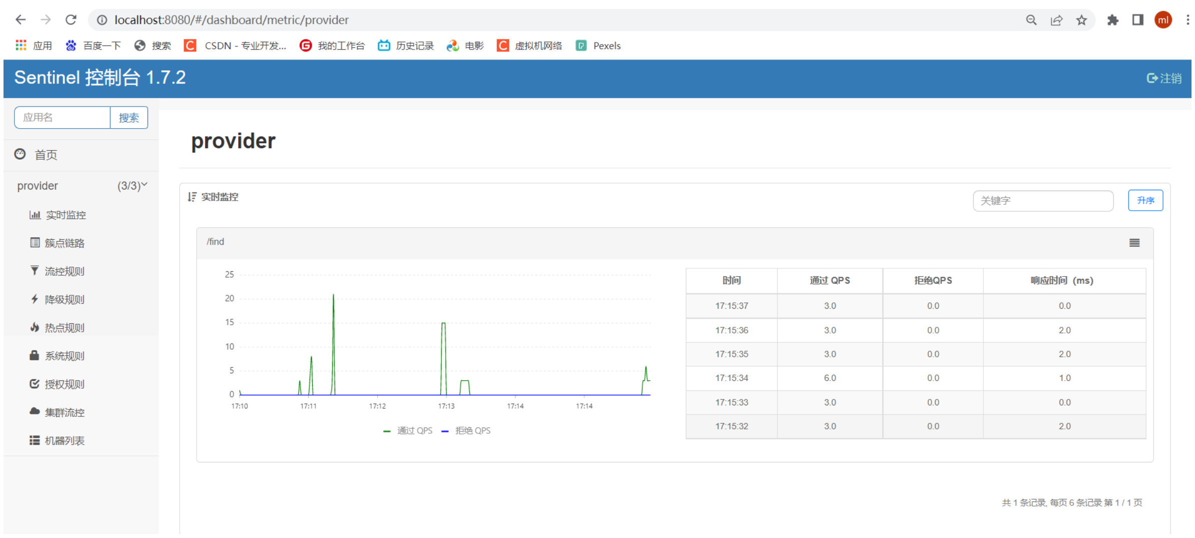
sentinel-dashboard-1.7.2.jar 2020/5/21 11:20 Executable Jar File 20,736 KB

浏览器访问 localhost:8080

用户名/密码都是sentinel



访问provider，实时监控进行捕获，QPS每秒钟请求数，每隔10s更新一次



provider

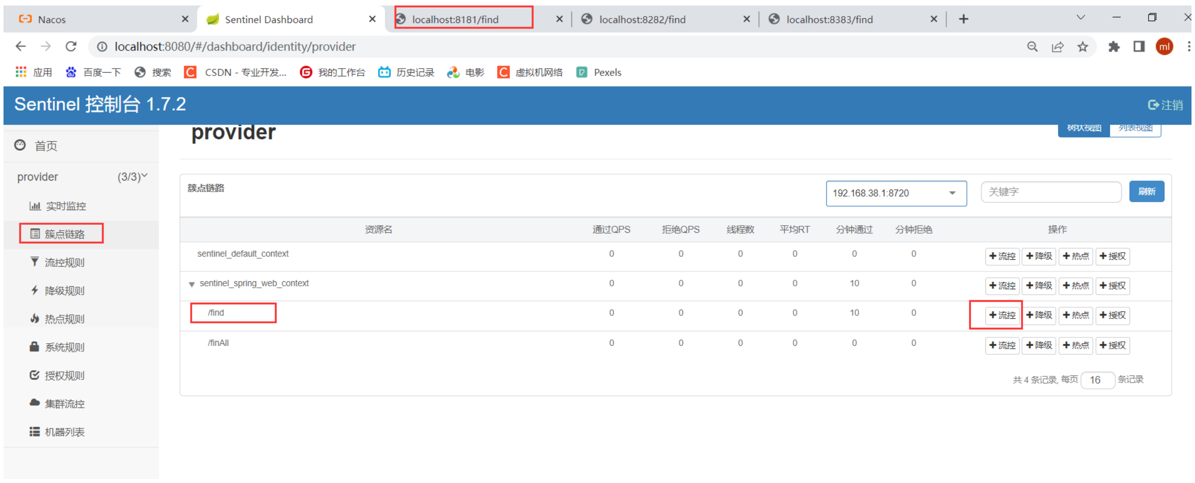
时间	通过 QPS	拒绝QPS	响应时间（ms)
17:15:37	3.0	0.0	0.0
17:15:36	3.0	0.0	2.0
17:15:35	3.0	0.0	2.0
17:15:34	6.0	0.0	1.0
17:15:33	3.0	0.0	0.0
17:15:32	3.0	0.0	2.0

共 1 条记录, 每页 6 条记录 第 1 / 1 页

5.1流控规则

直接限流

选择请求的簇点链路，对资源进行限流设置

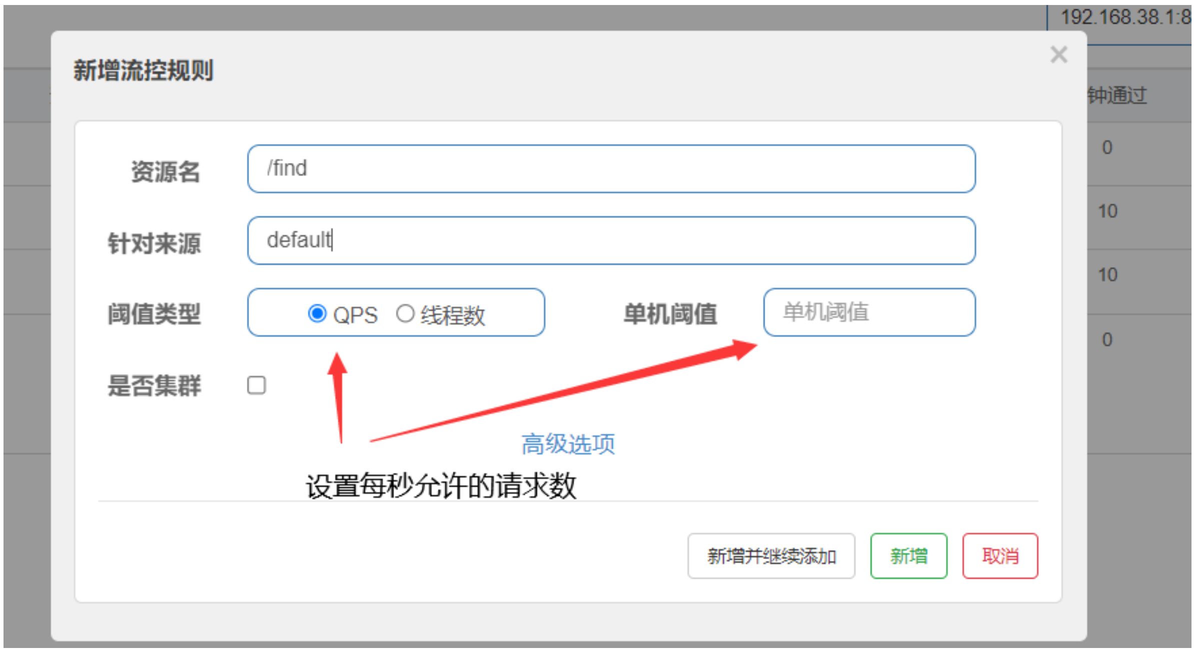


provider

簇点链路

资源名	通过QPS	拒绝QPS	线程数	平均RT	分钟通过	分钟拒绝	操作
sentinel_default_context	0	0	0	0	0	0	+流控 +降级 +热点 +授权
▼ sentinel_spring_web_context	0	0	0	0	10	0	+流控 +降级 +热点 +授权
/find	0	0	0	0	10	0	+流控 +降级 +热点 +授权
/finAll	0	0	0	0	0	0	+流控 +降级 +热点 +授权

共 4 条记录, 每页 16 条记录



新增流控规则

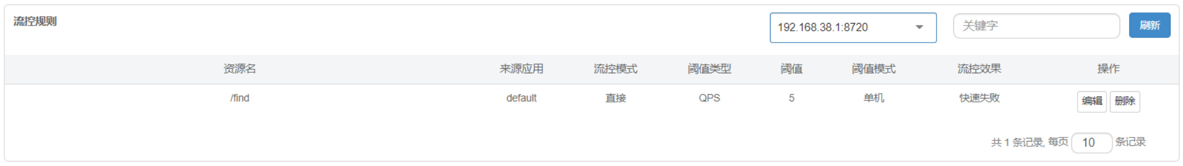
资源名: /find

针对来源: default

阈值类型: [✓ QPS] 线程数

单机阈值: 单机阈值

是否集群: ☐


流控规则

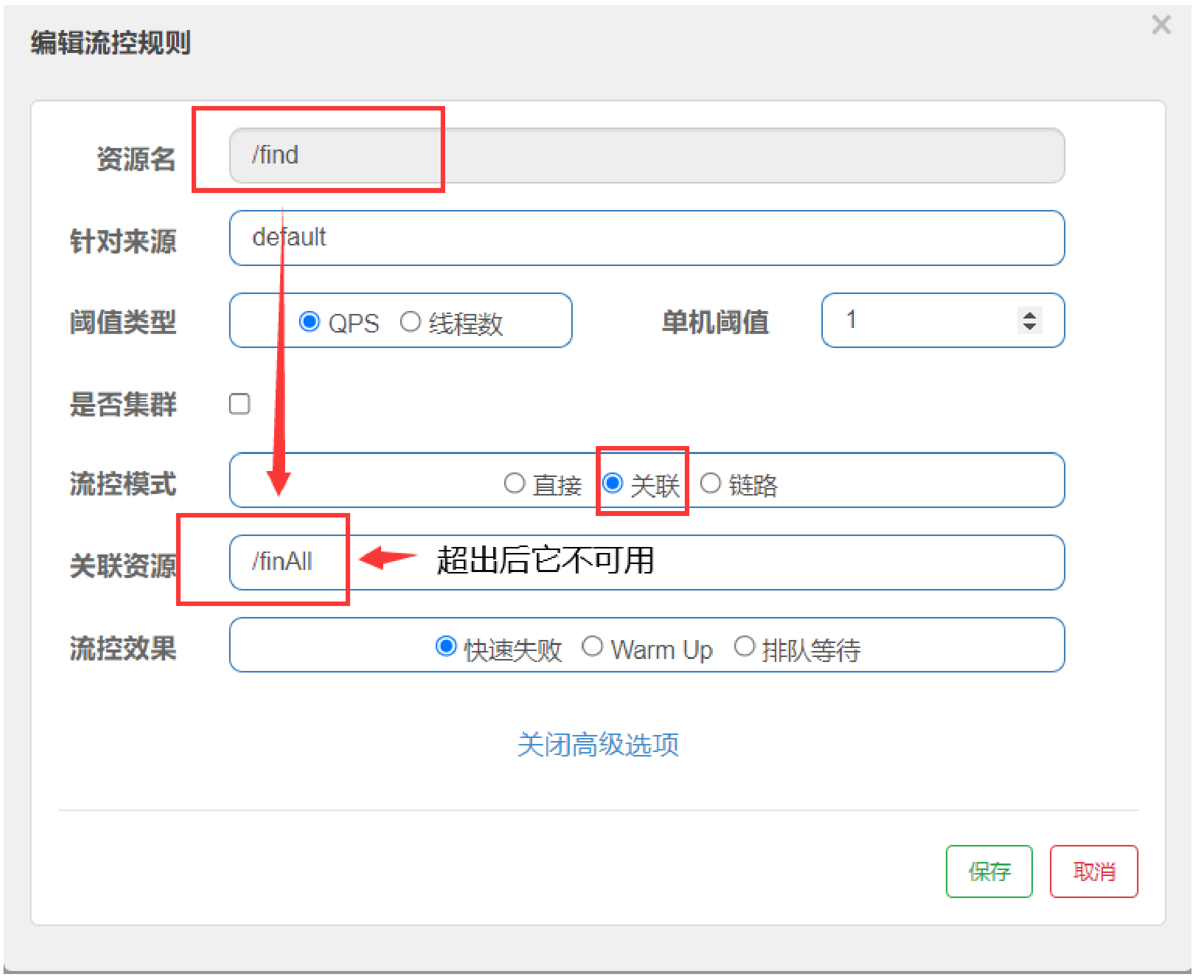
资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
/find	default	直接	QPS	5	单机	快速失败	编辑 删除

共 1 条记录, 每页 10 条记录

关联限流

关联限流指给某资源设置QPS后，若请求数超出设置的QPS，则会导致另外一个与它关联的资源不可访问。

```
@RestController
public class IndexController {

    @Value("${server.port}")
    private  String port;

    @Autowired
    private AccountMapper accountMapper;
    @RequestMapping("/find")
    public  String index(){
        return "该端口号是"+this.port;
    }

    @RequestMapping("/finAll")
    public List<Account> findAll(){
        System.out.println("当前调用的是"+this.port);
        return this.accountMapper.selectList(null);
    }

}
```



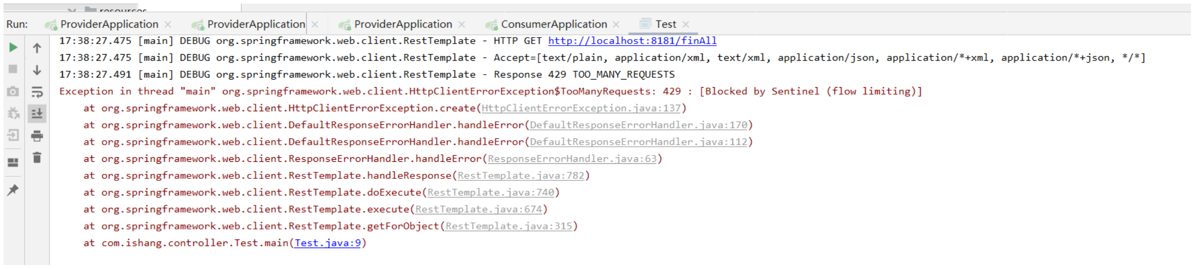
编辑流控规则

资源名	/find
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	1
是否集群	☐
流控模式	直接 [✓ 关联] 链路
关联资源	/finAll
流控效果	[✓ 快速失败] Warm Up 排队等待

测试方法:

```
import org.springframework.web.client.RestTemplate;

public class Test {
    public static void main(String[] args) throws InterruptedException {
        RestTemplate restTemplate = new RestTemplate();
        for (int i = 0; i < 1000; i++) {

 restTemplate.getForObject("http://localhost:8181/find",String.class);
            //0.5秒访问一次, QPS为2
            Thread.sleep(500);
        }
    }
}
```

访问/finAll时，会报错



```
17:38:27.475 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/finAll
17:38:27.475 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
17:38:27.491 [main] DEBUG org.springframework.web.client.RestTemplate - Response 429 TOO_MANY_REQUESTS
Exception in thread "main" org.springframework.web.client.HttpClientErrorException$TooManyRequests: 429 : [Blocked by Sentinel (flow limiting)]
	at org.springframework.web.client.HttpClientErrorException.create(HttpClientErrorException.java:137)
	at org.springframework.web.client.DefaultResponseErrorHandler.handleError(DefaultResponseErrorHandler.java:170)
	at org.springframework.web.client.DefaultResponseErrorHandler.handleError(DefaultResponseErrorHandler.java:112)
	at org.springframework.web.client.ResponseErrorHandler.handleError(ResponseErrorHandler.java:63)
	at org.springframework.web.client.RestTemplate.handleResponse(RestTemplate.java:782)
	at org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:740)
	at org.springframework.web.client.RestTemplate.execute(RestTemplate.java:674)
	at org.springframework.web.client.RestTemplate.getForObject(RestTemplate.java:315)
	at com.ishang.controller.Test.main(Test.java:9)
```

表示关联限流生效

链路限流

是一种更细粒度的限流，对某个资源（Service）进行限流，则请求资源的接口无法访问，这里有个坑，高版本（1.6.3+）的Sentinel Web filter默认收敛所有URL的入口context，因此链路限流不生效

因此我们需要手动关闭收敛，1.7.0版本开始（对应SCA的2.1.1.RELEASE),官方在CommonFilter引入参数,用于控制是否收敛context,将其配置为false即可根据不同的URL进行链路限流

解决如下：

1.pom.xml添加依赖

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-core</artifactId>
    <version>1.7.1</version>
</dependency>

<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-web-servlet</artifactId>
    <version>1.7.1</version>
</dependency>
```

2.application.yml

```
spring:
    cloud:
        sentinel:
          filter:
            enabled: false
```

3.创建配置类

```
package com.configuration;

import com.alibaba.csp.sentinel.adapter.servlet.CommonFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FilterConfiguration {

    @Bean
    public FilterRegistrationBean registrationBean(){
        FilterRegistrationBean registrationBean = new FilterRegistrationBean();
        registrationBean.setFilter(new CommonFilter());
        registrationBean.addUrlPatterns("/*");

 registrationBean.addInitParameter(CommonFilter.WEB_CONTEXT_UNIFY,"false");
        registrationBean.setName("sentinelFilter");
        return registrationBean;
    }
}
```

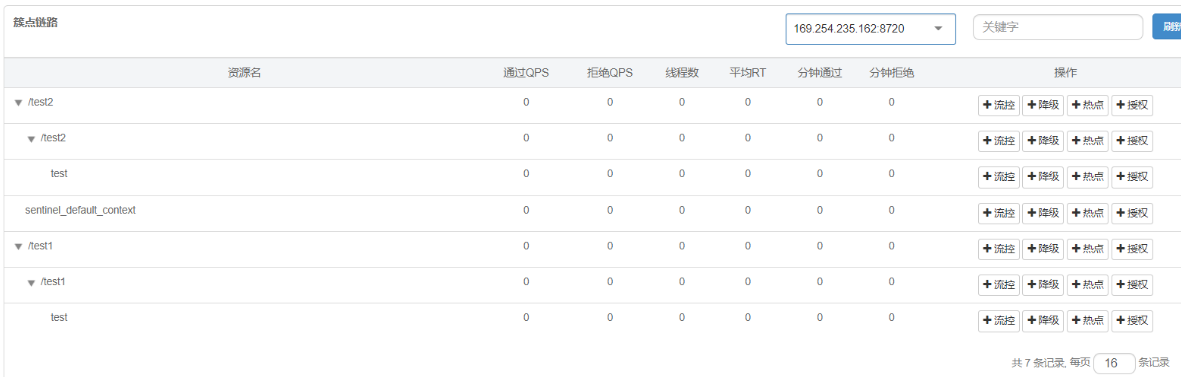
4.service

```java
import com.alibaba.csp.sentinel.annotation.SentinelResource;
import org.springframework.stereotype.Service;

@Service
public class HelloService {

    //添加该注解，表示对该链路进行限流
    @SentinelResource("hello")
    public void hello(){
        System.out.println("hello");
    }
}
```

5.controller

```java
@Autowired
private HelloService helloService;

@GetMapping("/test1")
public String test1(){
    this.helloService.test();
    return "test1";
}

@GetMapping("/test2")
public String test2(){
    this.helloService.test();
    return "test2";
}
```

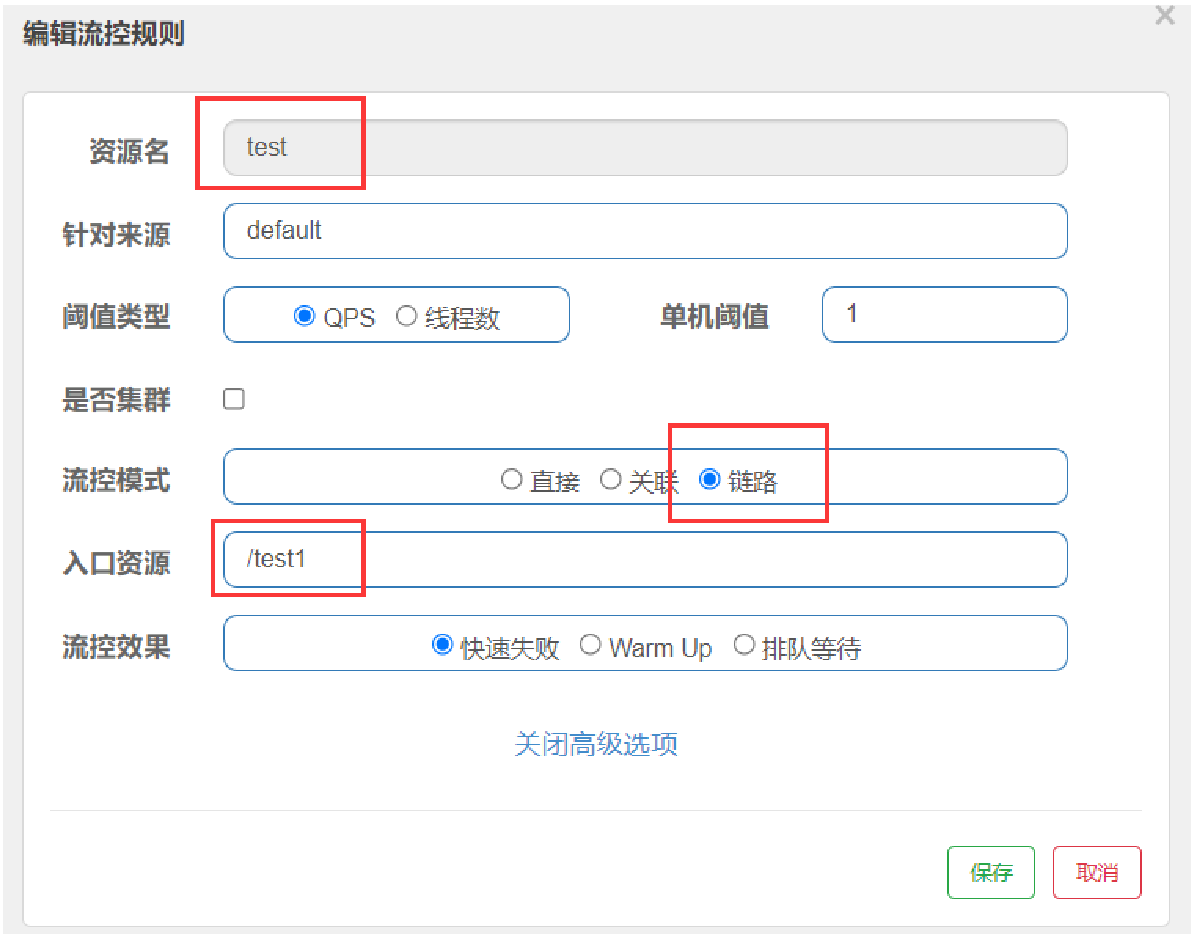
6.设置链路限流，将service中的test()链路进行限流



簇点链路　169.254.235.162:8720　关键字　刷新

资源名	通过QPS	拒绝QPS	线程数	平均RT	分钟通过	分钟拒绝	操作
▼ /test2	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
▼ /test2	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
test	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
sentinel_default_context	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
▼ /test1	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
▼ /test1	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权
test	0	0	0	0	0	0	✚流控 ✚降级 ✚热点 ✚授权

共 7 条记录, 每页 16 条记录


编辑流控规则

资源名	test
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	1
是否集群	☐
流控模式	直接 关联 [✓ 链路]
入口资源	/test1
流控效果	[✓ 快速失败] Warm Up 排队等待

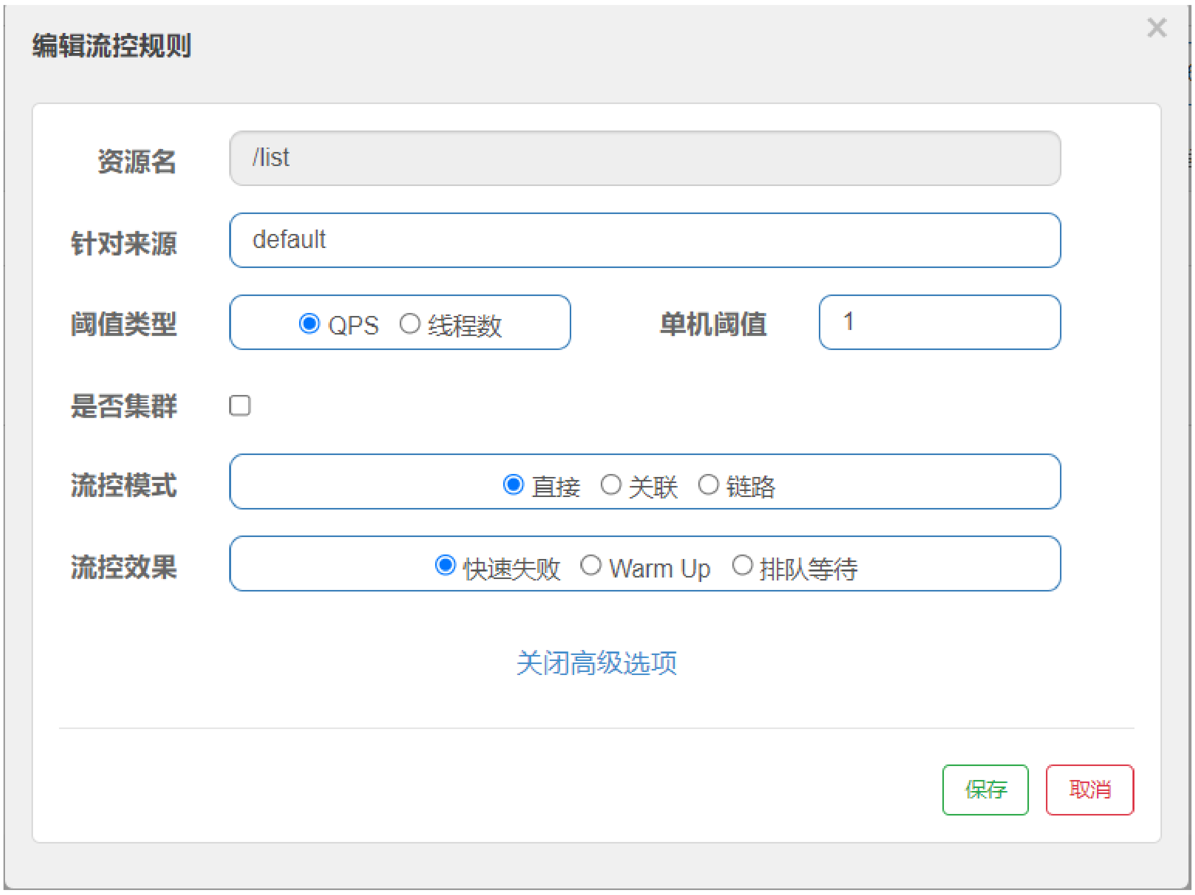
入口 /test1调用test()方法，会进行链路限流。

入口/test2调用test()方法，不会进行链路限流

5.2 流控效果

快速失败

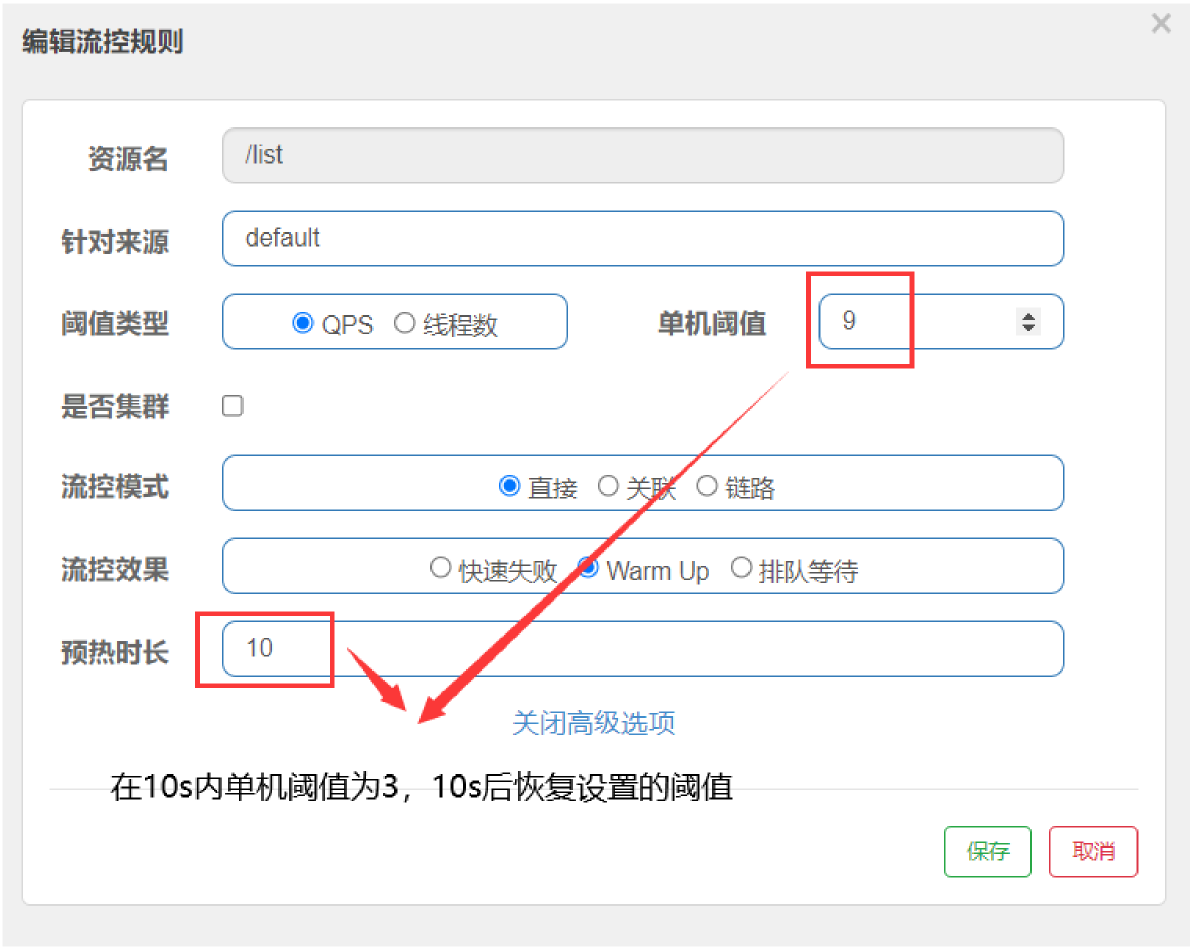
直接抛出异常


编辑流控规则

资源名	/list
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	1
是否集群	☐
流控模式	[✓ 直接] 关联 链路
流控效果	[✓ 快速失败] Warm Up 排队等待

Warm UP

给系统一个预热的时间，预热时间段内单机阈值较低，预热时间过后单机阈值增加，预热时间内当前的单机阈值是设置的阈值的三分之一，预热时间过后单机阈值恢复设置的值

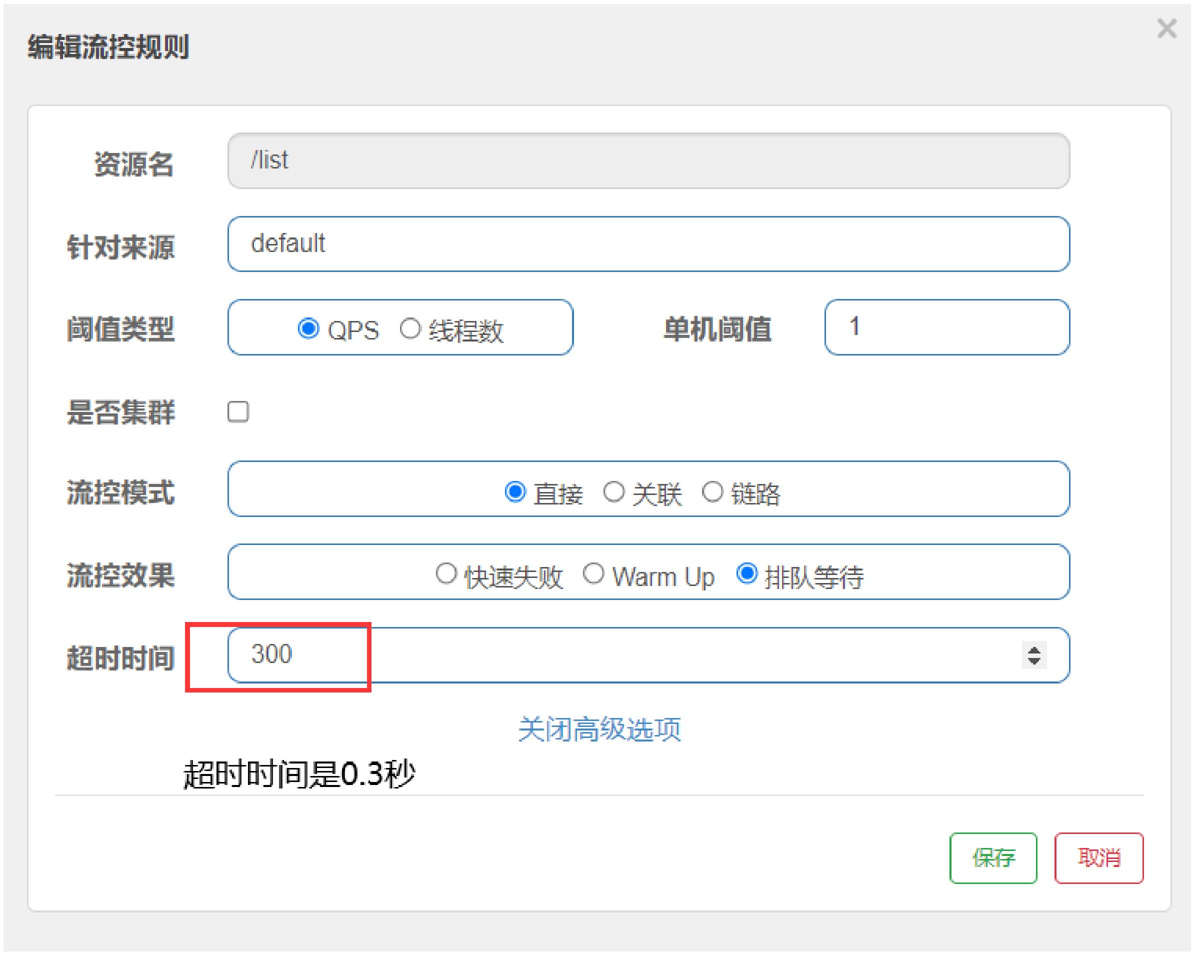

编辑流控规则

资源名	/list
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	9
是否集群	☐
流控模式	[✓ 直接] 关联 链路
流控效果	快速失败 [✓ Warm Up] 排队等待
预热时长	10

在10s内单机阈值为3，10s后恢复设置的阈值

排队等待

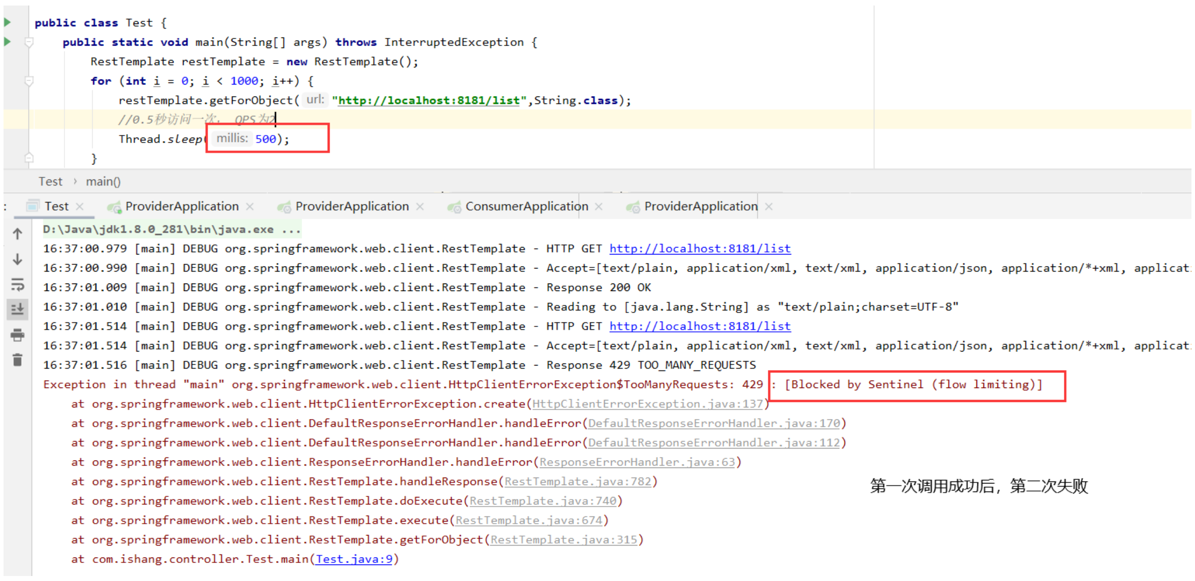
当请求调用失败后，不会立即抛出异常，等待下一次调用，时间范围是超时时间，在时间范围内如果请求成功则不抛出异常，如果请求不成功则抛出异常。（在规定时间内再给一次机会，如果没成功在抛出异常）



编辑流控规则

资源名	/list
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	1
是否集群	☐
流控模式	[✓ 直接] 关联 链路
流控效果	快速失败 Warm Up [✓ 排队等待]
超时时间	300

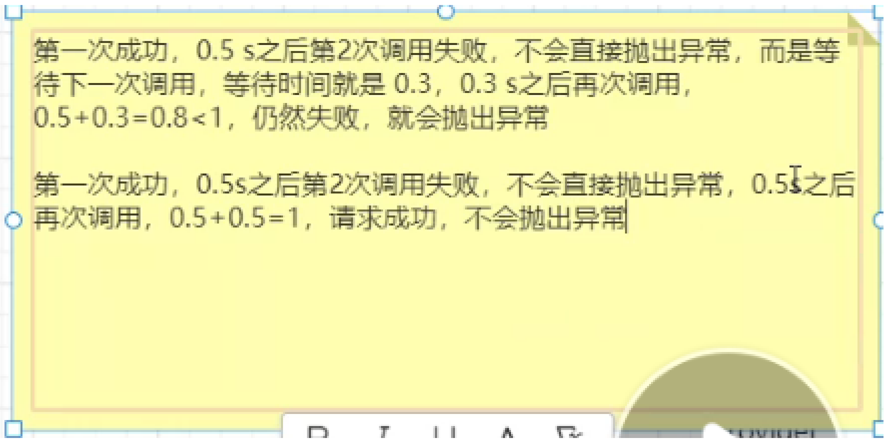
超时时间是0.3秒



```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        RestTemplate restTemplate = new RestTemplate();
        for (int i = 0; i < 1000; i++) {
            restTemplate.getForObject( url: "http://localhost:8181/list",String.class);
            //0.5秒访问一次， QPS为2
            Thread.sleep( millis: 500);
        }
```

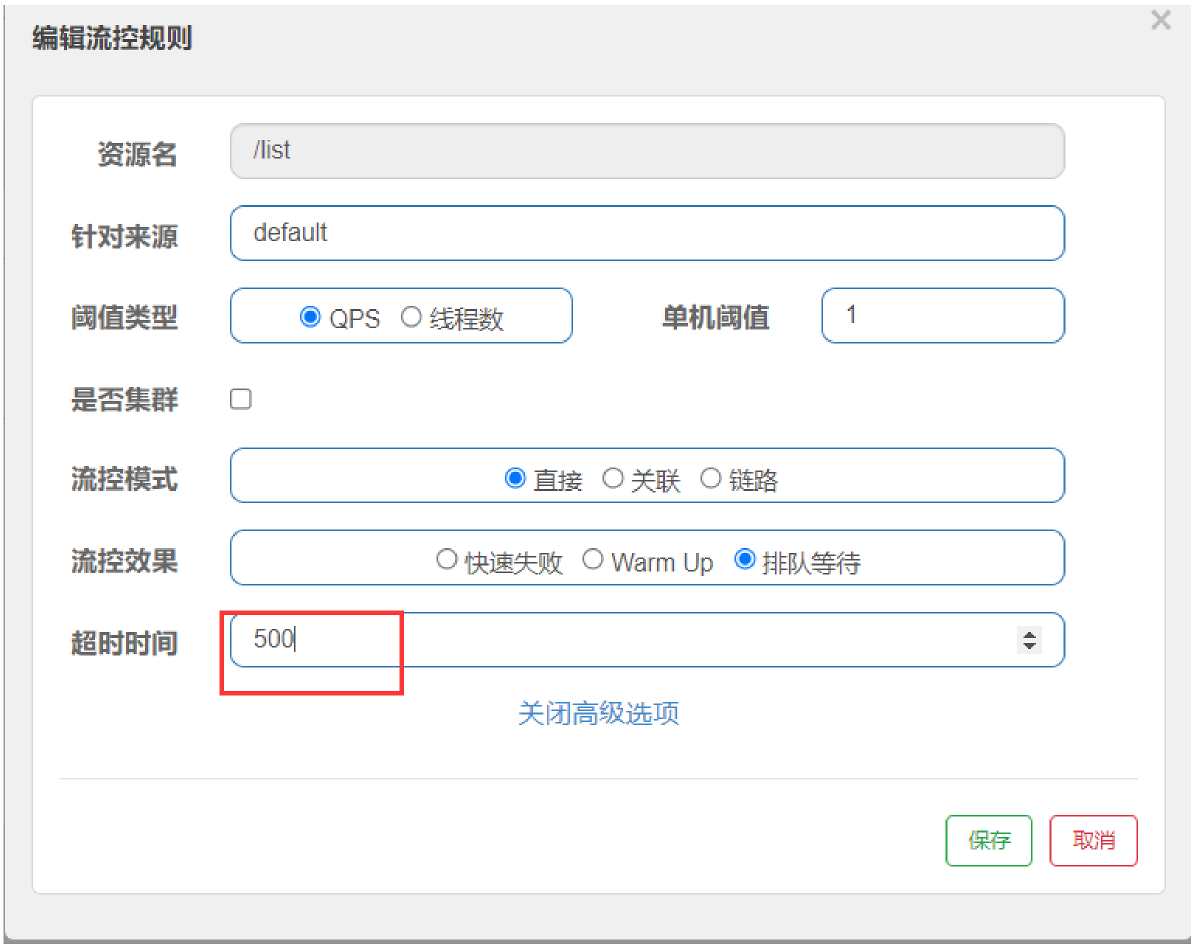
```
D:\Java\jdk1.8.0_281\bin\java.exe ...
16:37:00.979 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:37:00.990 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, applicat
16:37:01.009 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
16:37:01.010 [main] DEBUG org.springframework.web.client.RestTemplate - Reading to [java.lang.String] as "text/plain;charset=UTF-8"
16:37:01.514 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:37:01.514 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, applicat
16:37:01.516 [main] DEBUG org.springframework.web.client.RestTemplate - Response 429 TOO_MANY_REQUESTS
Exception in thread "main" org.springframework.web.client.HttpClientErrorException$TooManyRequests: 429 : [Blocked by Sentinel (flow limiting)]
    at org.springframework.web.client.HttpClientErrorException.create(HttpClientErrorException.java:137)
    at org.springframework.web.client.DefaultResponseErrorHandler.handleError(DefaultResponseErrorHandler.java:170)
    at org.springframework.web.client.DefaultResponseErrorHandler.handleError(DefaultResponseErrorHandler.java:112)
    at org.springframework.web.client.ResponseErrorHandler.handleError(ResponseErrorHandler.java:63)
    at org.springframework.web.client.RestTemplate.handleResponse(RestTemplate.java:782)
    at org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:740)
    at org.springframework.web.client.RestTemplate.execute(RestTemplate.java:674)
    at org.springframework.web.client.RestTemplate.getForObject(RestTemplate.java:315)
    at com.ishang.controller.Test.main(Test.java:9)
```

第一次调用成功后，第二次失败



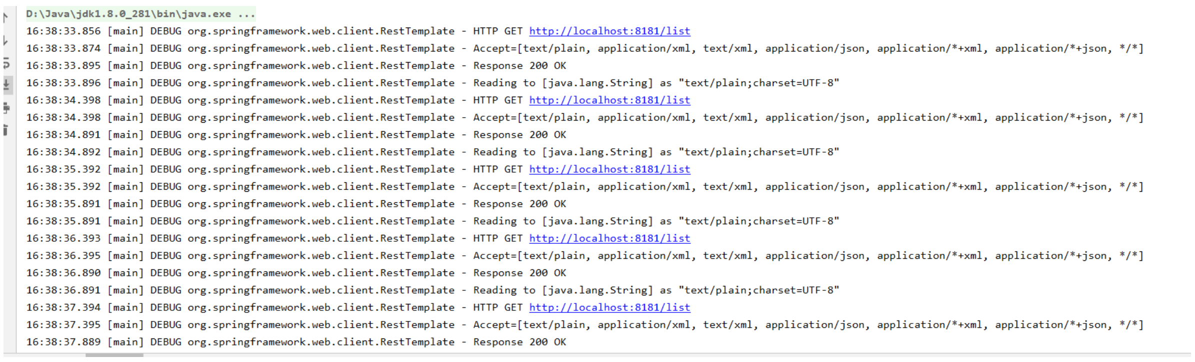
第一次成功，0.5 s之后第2次调用失败，不会直接抛出异常，而是等待下一次调用，等待时间就是 0.3，0.3 s之后再次调用，0.5+0.3=0.8<1，仍然失败，就会抛出异常

第一次成功，0.5s之后第2次调用失败，不会直接抛出异常，0.5s之后再次调用，0.5+0.5=1，请求成功，不会抛出异常



编辑流控规则

资源名	/list
针对来源	default
阈值类型	[✓ QPS] 线程数
单机阈值	1
是否集群	☐
流控模式	[✓ 直接] 关联 链路
流控效果	快速失败 Warm Up [✓ 排队等待]
超时时间	500



```
D:\Java\jdk1.8.0_281\bin\java.exe ...
16:38:33.856 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:38:33.874 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
16:38:33.895 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
16:38:33.896 [main] DEBUG org.springframework.web.client.RestTemplate - Reading to [java.lang.String] as "text/plain;charset=UTF-8"
16:38:34.398 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:38:34.398 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
16:38:34.891 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
16:38:34.892 [main] DEBUG org.springframework.web.client.RestTemplate - Reading to [java.lang.String] as "text/plain;charset=UTF-8"
16:38:35.392 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:38:35.392 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
16:38:35.891 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
16:38:35.891 [main] DEBUG org.springframework.web.client.RestTemplate - Reading to [java.lang.String] as "text/plain;charset=UTF-8"
16:38:36.393 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:38:36.395 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
16:38:36.890 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
16:38:36.891 [main] DEBUG org.springframework.web.client.RestTemplate - Reading to [java.lang.String] as "text/plain;charset=UTF-8"
16:38:37.394 [main] DEBUG org.springframework.web.client.RestTemplate - HTTP GET http://localhost:8181/list
16:38:37.395 [main] DEBUG org.springframework.web.client.RestTemplate - Accept=[text/plain, application/xml, text/xml, application/json, application/*+xml, application/*+json, */*]
16:38:37.889 [main] DEBUG org.springframework.web.client.RestTemplate - Response 200 OK
```

改为500后成功

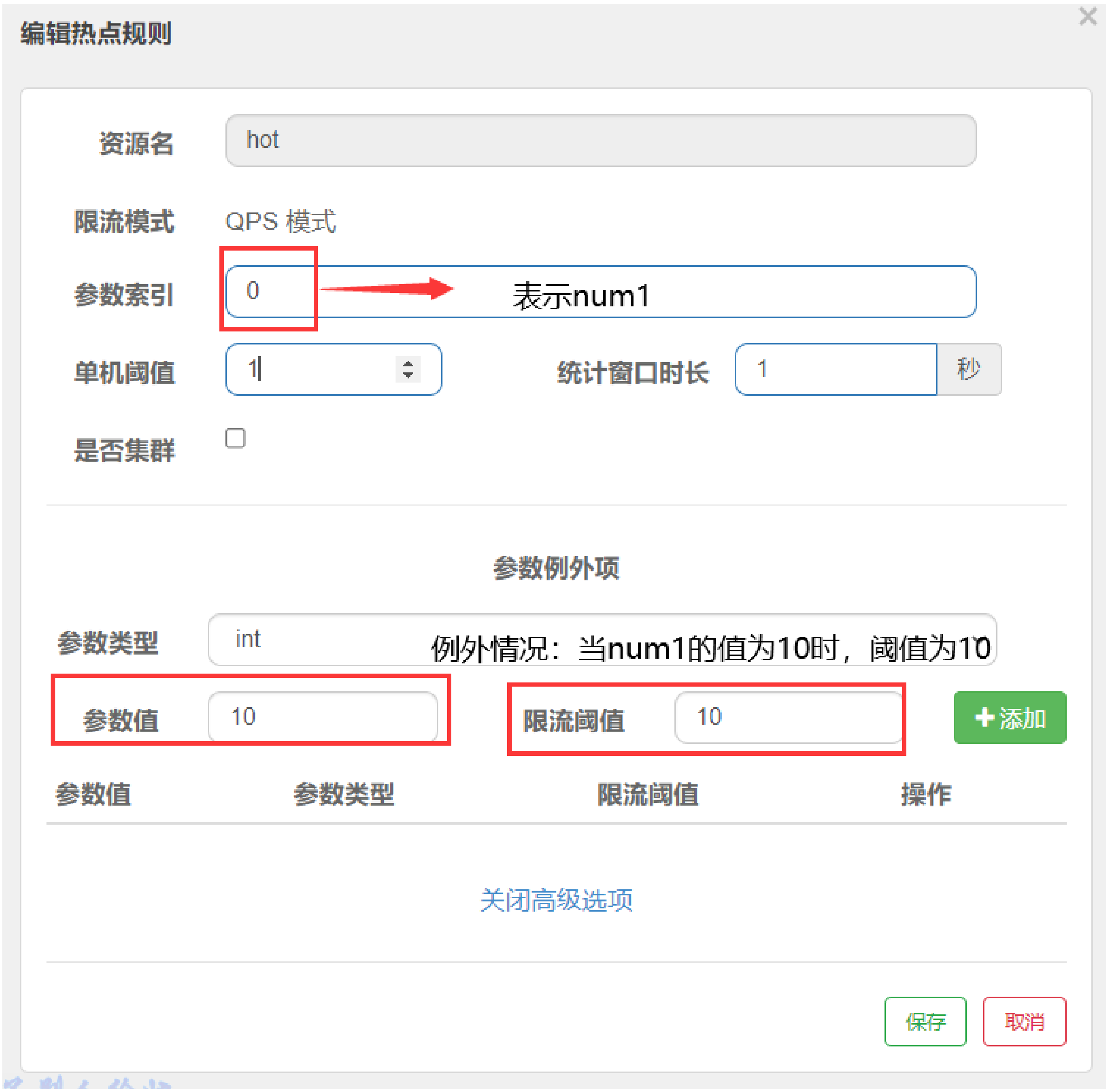
5.3 热点规则

热点规则是流控规则的更细粒度操作，可以具体到对某个热点参数的限流，设置限流之后，如果带着限流参数的请求量超过阈值，则进行限流，时间为统计窗口的时长

高级设置是指可以给限流参数设置例外的值，同时设置对应的阈值，当参数的值为例外的值时，阈值采用对应的阈值，其他情况都是默认阈值

必须要添加@SentinelResource,即对资源进行流控

```
@RequestMapping("/hot")
@SentinelResource("hot")
public String hot(
        @RequestParam(value = "num1",required = false) Integer num1,
        @RequestParam(value = "num2",required = false) Integer num2
        ){
    return num1+"-"+num2;

}
```



编辑热点规则

资源名	hot
限流模式	QPS 模式
参数索引	0
单机阈值	1
统计窗口时长	1 秒
是否集群	☐

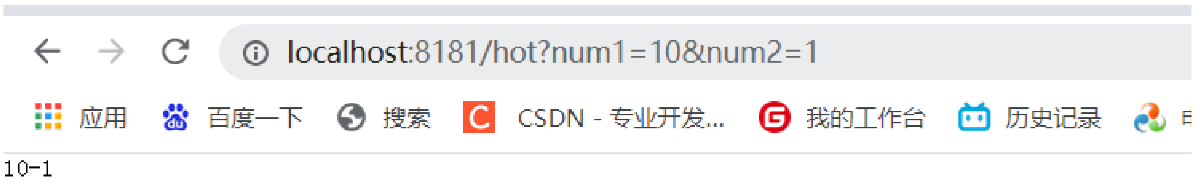
表示num1

参数例外项

参数类型	int
参数值	10
限流阈值	10

例外情况：当num1的值为10时，阈值为10

参数值	参数类型	限流阈值	操作



```
10-1
```

5.4 授权规则

给指定的资源设置流控应用（追加参数），可以对流控应用进行访问权限的设置，具体就是添加白名单和黑名单

如果设置白名单，那么只有出现在白名单上的流控应用才能访问，其他应用不能访问。

如果设置黑名单，那么出现在黑名单上的流控应用不能访问，其他正常访问

如何给请求指定流控应用：通过实现RequestOriginParser接口来完成，

代码如下所示：

1.创建RequestOriginParserDefinition类

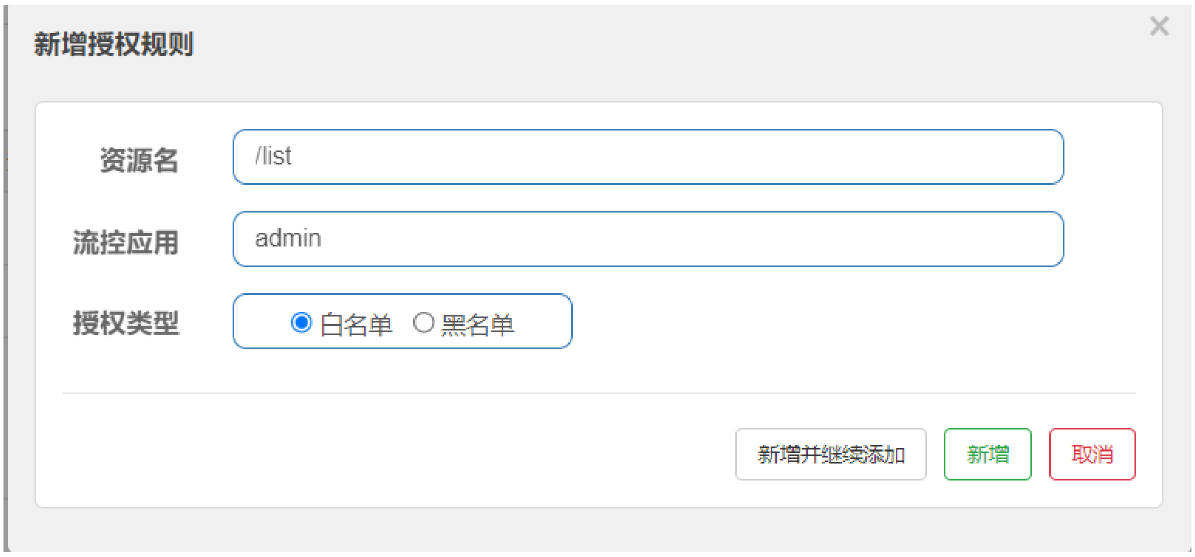
```
package com.ishang.configuration;

import com.alibaba.csp.sentinel.adapter.servlet.callback.RequestOriginParser;
import org.springframework.util.StringUtils;

import javax.servlet.http.HttpServletRequest;

public class RequestOriginParserDefinition implements RequestOriginParser {
    @Override
    public String parseOrigin(HttpServletRequest httpServletRequest) {
        //设置的追加参数名
        String name = httpServletRequest.getParameter("name");
        if(StringUtils.isEmpty(name)){
            throw new RuntimeException("name is null");
        }
        return name;
    }
}
```

2.创建配置类

```
package com.ishang.configuration;

import com.alibaba.csp.sentinel.adapter.servlet.callback.WebCallbackManager;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;

@Configuration
public class SentinelConfiguration {

    @PostConstruct
    public void init(){
        WebCallbackManager.setRequestOriginParser(new
RequestOriginParserDefinition());
```

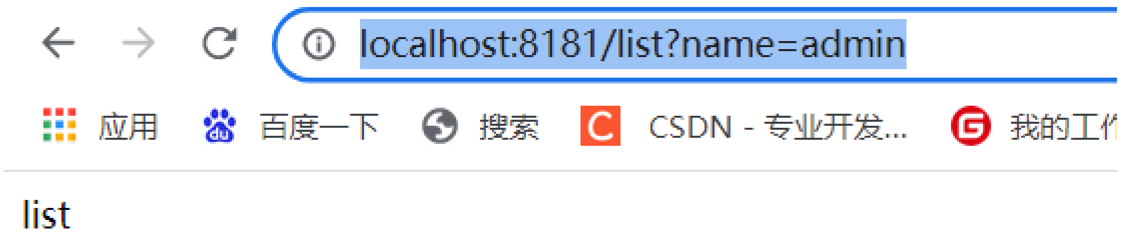
```
    }
}
```

3.设置白名单/黑名单

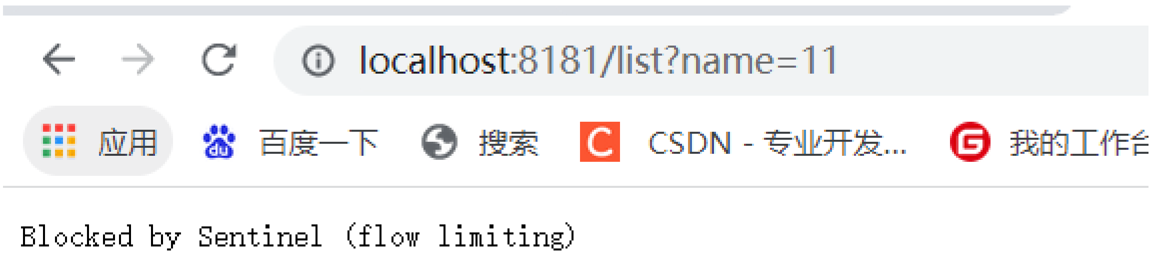

新增授权规则

资源名	/list
流控应用	admin
授权类型	[✓ 白名单] 黑名单

添加白名单时，只有白名单上面的参数才能访问

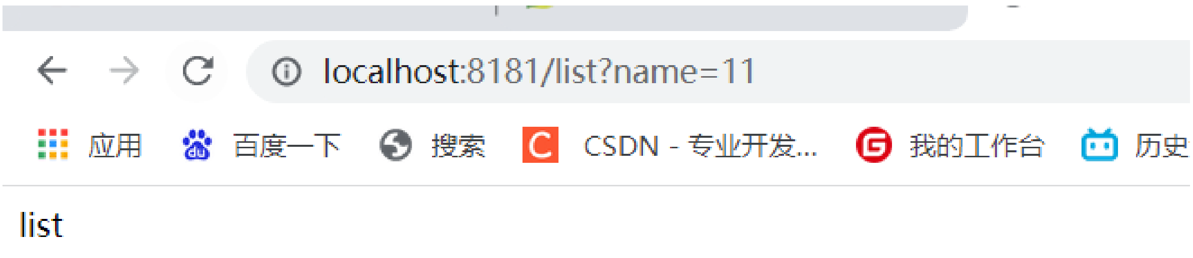

localhost:8181/list?name=admin

list

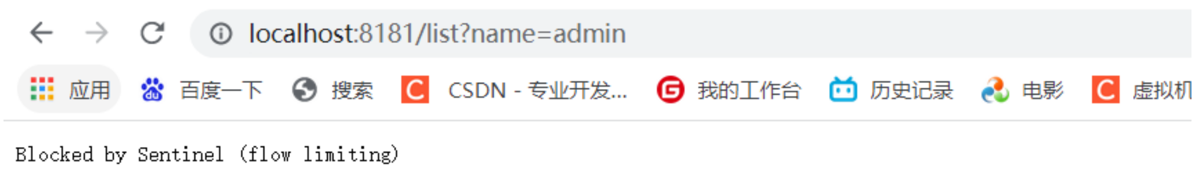

localhost:8181/list?name=11

```
Blocked by Sentinel (flow limiting)
```

添加黑名单，即除了黑名单上的，都可以访问


localhost:8181/list?name=11

list

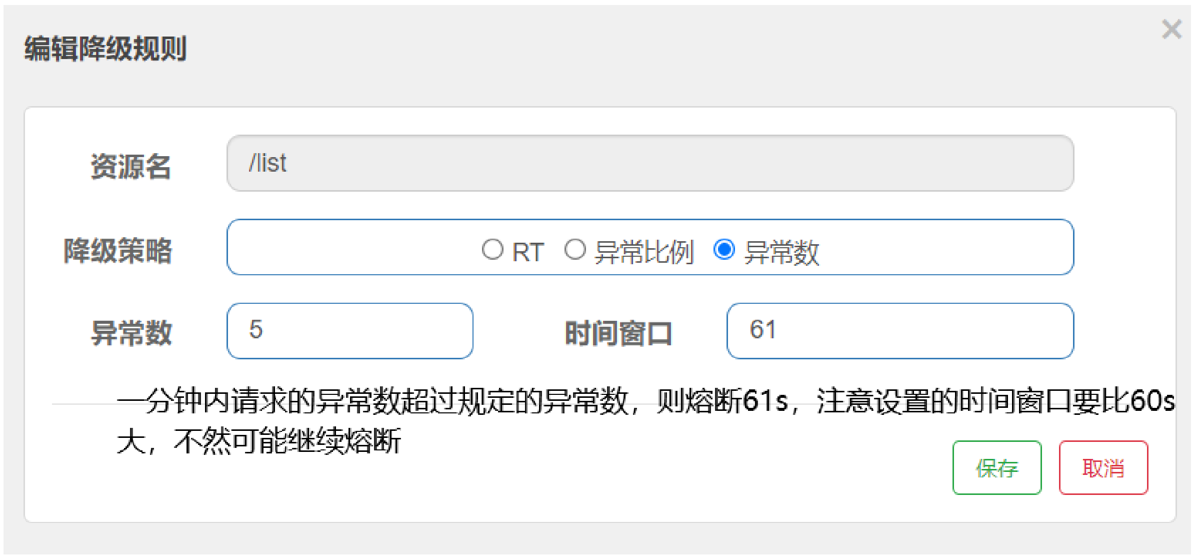


```
Blocked by Sentinel (flow limiting)
```

5.5 降级规则

异常数

1分钟内的异常数超过阈值就进行降级处理，时间窗口的值要大于60s，否则刚结束熔断又进入下一次熔断

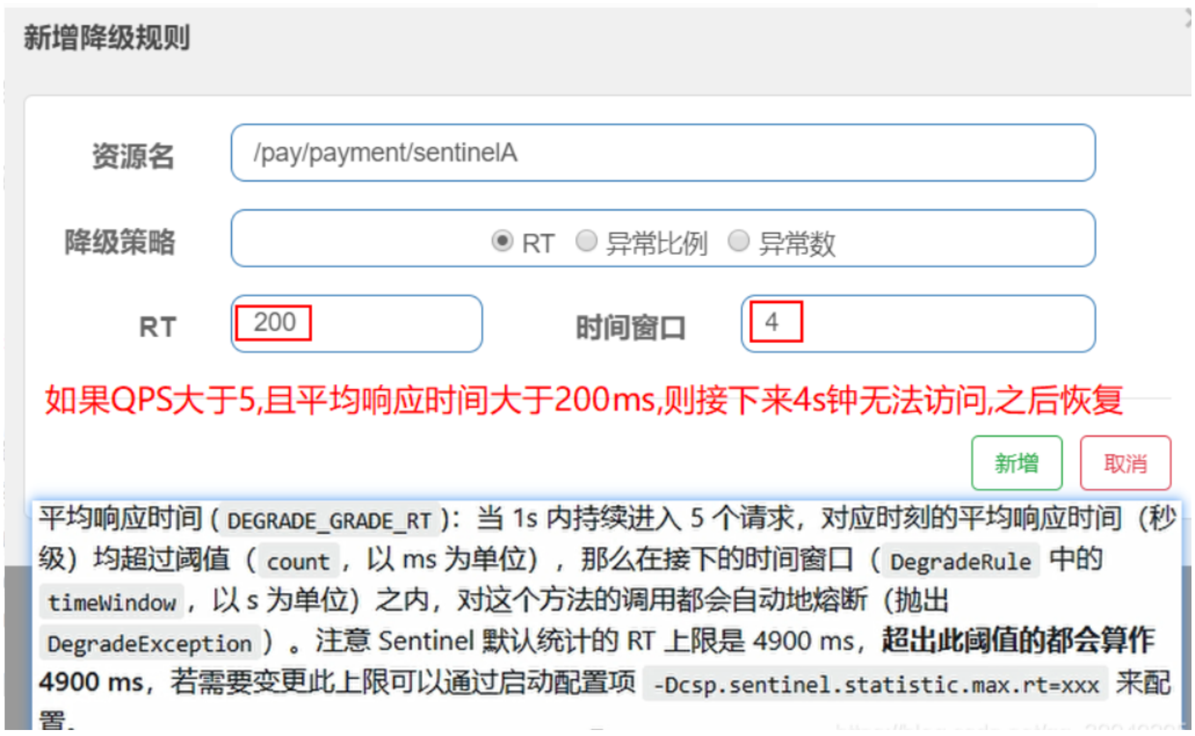


编辑降级规则

资源名	/list		
降级策略	RT 异常比例 [✓ 异常数]		
异常数	5	时间窗口	61

一分钟内请求的异常数超过规定的异常数，则熔断61s，注意设置的时间窗口要比60s大，不然可能继续熔断

RT



新增降级规则

资源名	/pay/payment/sentinelA		
降级策略	[✓ RT] 异常比例 异常数		
RT	200	时间窗口	4

如果QPS大于5,且平均响应时间大于200ms,则接下来4s钟无法访问,之后恢复

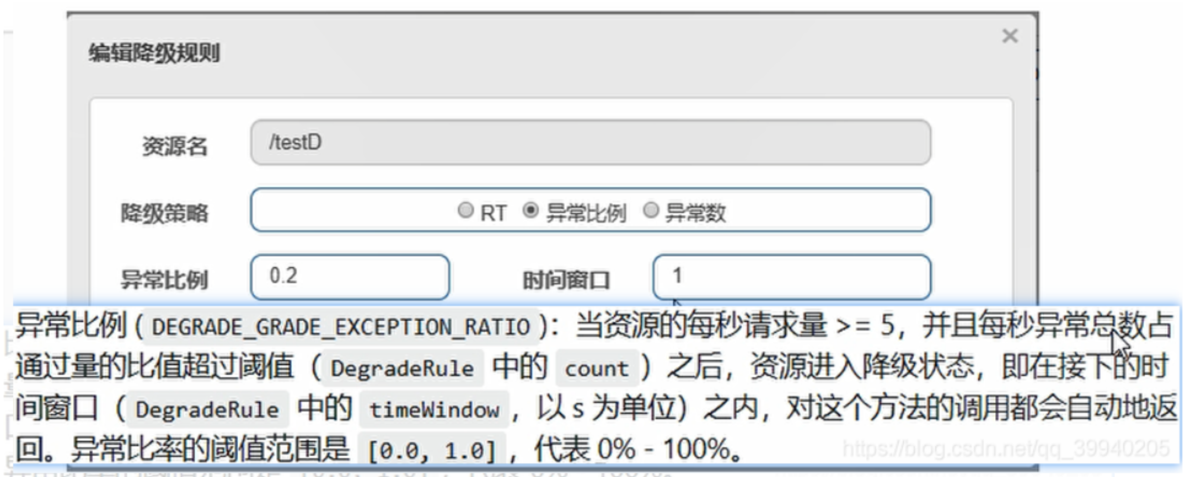
平均响应时间 (`DEGRADE_GRADE_RT`)：当 1s 内持续进入 5 个请求，对应时刻的平均响应时间（秒级）均超过阈值（ `count` ，以 ms 为单位），那么在接下的时间窗口（ `DegradeRule` 中的 `timeWindow` ，以 s 为单位）之内，对这个方法的调用都会自动地熔断（抛出 `DegradeException` ）。注意 Sentinel 默认统计的 RT 上限是 4900 ms，**超出此阈值的都会算作 4900 ms**，若需要变更此上限可以通过启动配置项 `-Dcsp.sentinel.statistic.max.rt=xxx` 来配置。

单个请求的响应时间超过阈值，则进入准降级状态，接下来 1 S 内连续 5 个请求响应时间均超过阈值，就进行降级，持续时间为时间窗口的值。

当第一次请求响应时间超过 1 毫秒，就准降级，1 S 之内连续发送 5 个请求，响应时间均超过 1 毫秒则进入降级状态，降级状态持续时间为 8 s，8 s 之后恢复正常，进入下一轮循环。

异常比例

每秒异常数量占通过量的比例大于阈值，就进行降级处理，持续时间为时间窗口的值。



异常比例 (DEGRADE_GRADE_EXCEPTION_RATIO): 当资源的每秒请求量 >= 5，并且每秒异常总数占通过量的比值超过阈值 (DegradeRule 中的 count) 之后，资源进入降级状态，即在接下的时间窗口 (DegradeRule 中的 timeWindow ，以 s 为单位) 之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 [0.0, 1.0] ，代表 0% - 100%。

5.6自定义规则异常返回

默认情况下，发生限流、降级、授权拦截时，都会抛出异常到调用方。如果要自定义异常时的返回结果，需要实现UrlBlockHandler接口：

```
package com.ishang.handler;

import com.alibaba.csp.sentinel.adapter.servlet.callback.UrlBlockHandler;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.alibaba.csp.sentinel.slots.block.authority.AuthorityException;
import com.alibaba.csp.sentinel.slots.block.degrade.DegradeException;
import com.alibaba.csp.sentinel.slots.block.flow.FlowException;
import com.alibaba.csp.sentinel.slots.block.flow.param.ParamFlowException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

//创建异常处理类
public class ExceptionHandler implements UrlBlockHandler {
    @Override
    public void blocked(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, BlockException e) throws IOException {

        httpServletResponse.setContentType("text/html;charset=utf-8");
        String msg=null;
        int status =429;
        if(e instanceof FlowException){
            msg ="请求被限流了";
        }else if (e instanceof ParamFlowException){
            msg="请求被热点参数限流";
        }else if (e instanceof DegradeException){
            msg ="请求被降级了";
        }else if(e instanceof AuthorityException){
            msg = "没有权限访问";
            status =401;
        }
        httpServletResponse.setStatus(status);
        httpServletResponse.getWriter().println("{\"msg\": " + msg + ",
\"status\": " + status + "}");
```

```
    }
}
```

2.创建SentinelConfiguration配置

```java
package com.ishang.configuration;

import com.alibaba.csp.sentinel.adapter.servlet.callback.WebCallbackManager;
import com.ishang.handler.ExceptionHandler;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;

@Configuration
public class SentinelConfiguration {

    @PostConstruct
    public void init(){
//          WebCallbackManager.setRequestOriginParser(new
RequestOriginParserDefinition());
        WebCallbackManager.setUrlBlockHandler(new ExceptionHandler());
    }
}
```

当请求被限流降级时会给客户端做出相应的响应。

5.6Feign 整合 Sentinel

创建另外一个微服务，通过 Feign 调用 provider，pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

application.yml

```yaml
server:
  port: 8380
feign:
  sentinel:
    enabled: true
spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8080
  application:
    name: feign
```

Feign

```java
package com.southwind.feign;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

@FeignClient("provider")
public interface ProviderFeign {

    @GetMapping("/index")
    public String index();
}
```

Controller

```java
package com.southwind.controller;

import com.southwind.feign.ProviderFeign;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FeignController {

    @Autowired
    private ProviderFeign providerFeign;

    @GetMapping("/index")
    public String index(){
        return this.providerFeign.index();
    }
}
```

如果要自定义异常，创建 Fallback 类，实现接口。

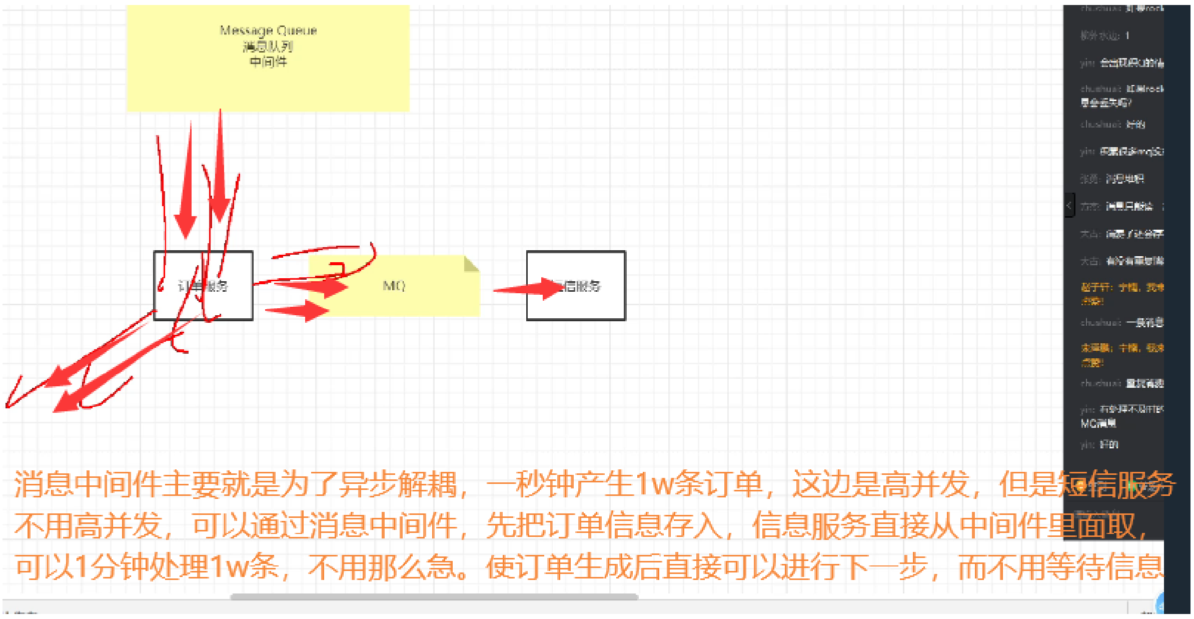
```java
package com.southwind.fallback;

import com.southwind.feign.ProviderFeign;
import org.springframework.stereotype.Component;

@Component
public class ProviderFeignFallback implements ProviderFeign {
    @Override
    public String index() {
        return "index-fallback";
    }
}
```

FeignClient 指定 fallback

```java
package com.southwind.feign;

import com.southwind.fallback.ProviderFeignFallback;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(value = "provider",fallback = ProviderFeignFallback.class)
public interface ProviderFeign {

    @GetMapping("/index")
    public String index();
}
```

6.Rocket MQ

异步解耦



消息中间件主要就是为了异步解耦，一秒钟产生1w条订单，这边是高并发，但是短信服务不用高并发，可以通过消息中间件，先把订单信息存入，信息服务直接从中间件里面取，可以1分钟处理1w条，不用那么急。使订单生成后直接可以进行下一步，而不用等待信息

Active MQ,RibbateMQ,RocketMQ,Kafaka

RocketMQ是阿里巴巴的MQ中间件

http://rocketmq.apache.org/release_notes/release-notes-4.7.1/

6.1安装RocketMQ

1、

```
cd /usr/local
mkdir rokerMQ
```

2.传入 rokerMQ文件夹

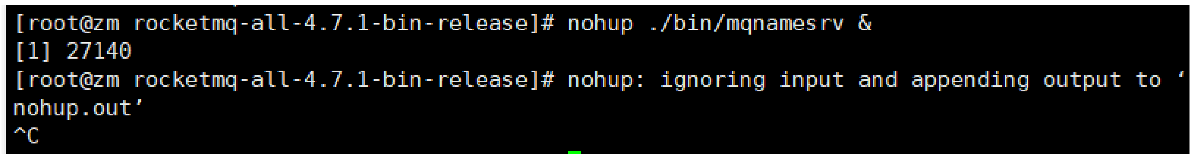
3.进入rokerMQ文件夹后，解压缩

```
unzip rocketmq-all-4.7.1-bin-release.zip
```

4.进入解压后的文件夹，启动 NameServer

```
cd rocketmq-all-4.7.1-bin-release
```

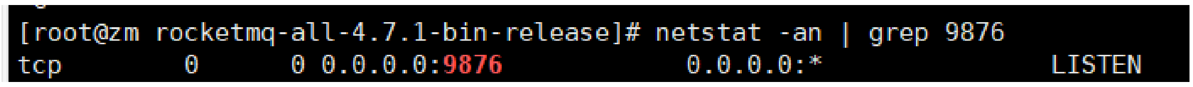
```
nohup ./bin/mqnamesrv &
```



```
[root@zm rocketmq-all-4.7.1-bin-release]# nohup ./bin/mqnamesrv &
[1] 27140
[root@zm rocketmq-all-4.7.1-bin-release]# nohup: ignoring input and appending output to ‘
nohup.out’
^C
```

5.检查是否启动成功

```
netstat -an | grep 9876
```



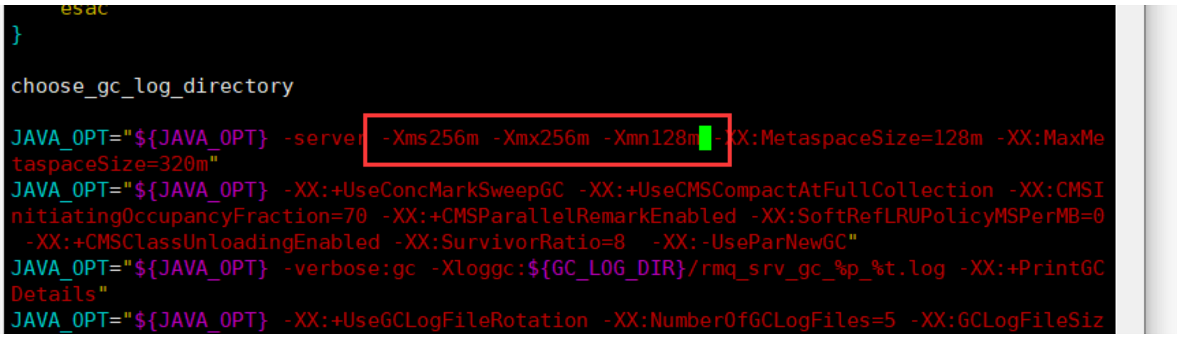
```
[root@zm rocketmq-all-4.7.1-bin-release]# netstat -an | grep 9876
tcp        0      0 0.0.0.0:9876            0.0.0.0:*               LISTEN
```

6.启动Broker

启动之前需要编辑配置文件，修改JVM内存设置，默认给的内存4GB,超过我们的JVM了。

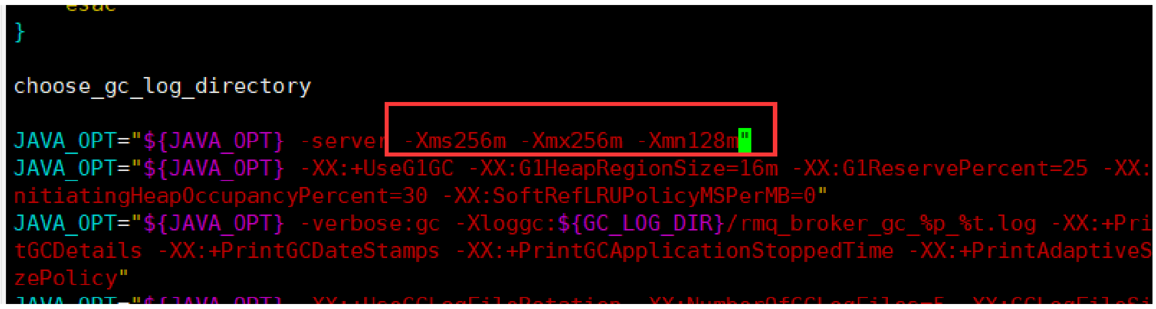
```
cd bin
vim runserver.sh
```

修改为一下数值



```
    esac
}

choose_gc_log_directory

JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMe
taspaceSize=320m"
JAVA_OPT="${JAVA_OPT} -XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection -XX:CMSI
nitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=0
 -XX:+CMSClassUnloadingEnabled -XX:SurvivorRatio=8  -XX:-UseParNewGC"
JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_srv_gc_%p_%t.log -XX:+PrintGC
Details"
JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSiz
```

```
vim runbroker.sh
```

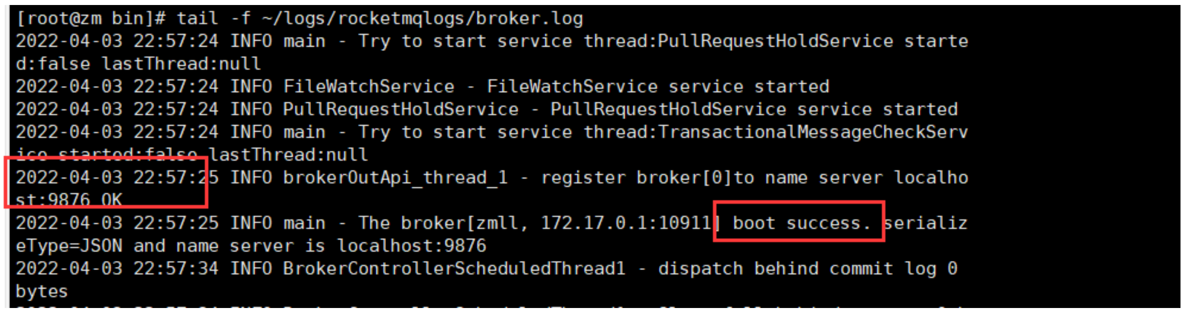


```
    esac
}

choose_gc_log_directory

JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m"
JAVA_OPT="${JAVA_OPT} -XX:+UseG1GC -XX:G1HeapRegionSize=16m -XX:G1ReservePercent=25 -XX:I
nitiatingHeapOccupancyPercent=30 -XX:SoftRefLRUPolicyMSPerMB=0"
JAVA_OPT="${JAVA_OPT} -verbose:gc -Xloggc:${GC_LOG_DIR}/rmq_broker_gc_%p_%t.log -XX:+Pri
tGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintAdaptiveS
zePolicy"
JAVA_OPT="${JAVA_OPT} -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=5 -XX:GCLogFileSi
```

7.启动 Broker

```
nohup ./mqbroker -n localhost:9876 &
```

通过查看日志，检查是否成功启动



```
[root@zm bin]# tail -f ~/logs/rocketmqlogs/broker.log
2022-04-03 22:57:24 INFO main - Try to start service thread:PullRequestHoldService starte
d:false lastThread:null
2022-04-03 22:57:24 INFO FileWatchService - FileWatchService service started
2022-04-03 22:57:24 INFO PullRequestHoldService - PullRequestHoldService service started
2022-04-03 22:57:24 INFO main - Try to start service thread:TransactionalMessageCheckServ
ice started:false lastThread:null
2022-04-03 22:57:25 INFO brokerOutApi_thread_1 - register broker[0]to name server localho
st:9876 OK
2022-04-03 22:57:25 INFO main - The broker[zmll, 172.17.0.1:10911] boot success. serializ
eType=JSON and name server is localhost:9876
2022-04-03 22:57:34 INFO BrokerControllerScheduledThread1 - dispatch behind commit log 0
bytes
```

启动成功

8、测试 RocketMQ

消息发送

```
cd bin
export NAMESRV_ADDR=localhost:9876
./tools.sh org.apache.rocketmq.example.quickstart.Producer
```

消息接收

```
cd bin
export NAMESRV_ADDR=localhost:9876
./tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

9、关闭 RocketMQ

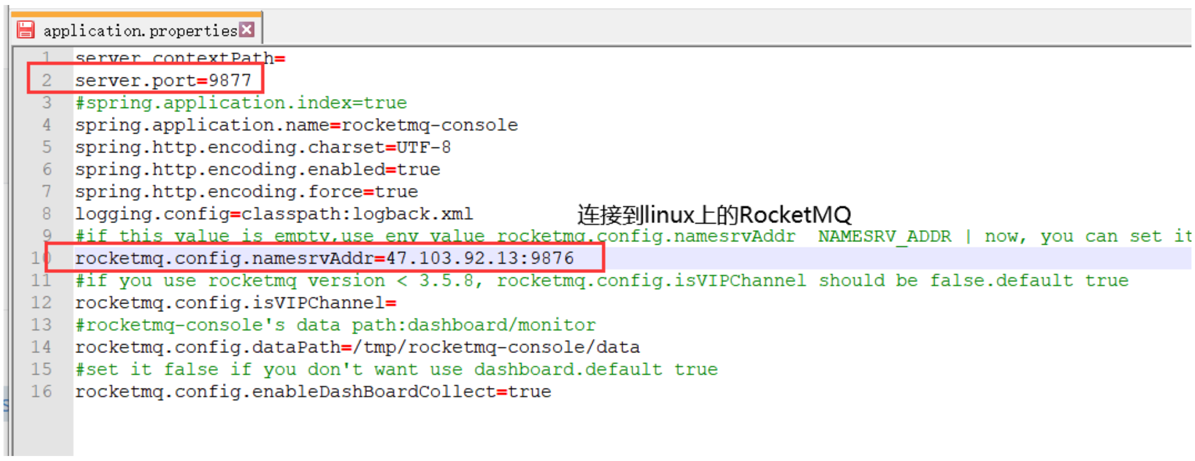
```
cd bin
./mqshutdown broker
./mqshutdown namesrv
```

6.2 安装RocketMQ控制台

1、下载

https://github.com/apache/rocketmq-externals/releases

2.解压缩，修改application.properties



```
server.contextPath=
server.port=9877
#spring.application.index=true
spring.application.name=rocketmq-console
spring.http.encoding.charset=UTF-8
spring.http.encoding.enabled=true
spring.http.encoding.force=true
logging.config=classpath:logback.xml
#if this value is empty,use env value rocketmq.config.namesrvAddr  NAMESRV_ADDR | now, you can set it
rocketmq.config.namesrvAddr=47.103.92.13:9876
#if you use rocketmq version < 3.5.8, rocketmq.config.isVIPChannel should be false.default true
rocketmq.config.isVIPChannel=
#rocketmq-console's data path:dashboard/monitor
rocketmq.config.dataPath=/tmp/rocketmq-console/data
#set it false if you want use dashboard.default true
rocketmq.config.enableDashBoardCollect=true
```

3.进入控制台根路径，进行打包

```
mvn clean package -Dmaven.test.skip=true
```

4、进入 target 启动 jar

```
java -jar rocketmq-console-ng-1.0.0.jar
```

第三步和第四步也有可以直接从idea运行

原因，JDK 9 以上版本缺失 jar，需要手动导入，打开项目 pom.xml 添加

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
</dependency>
```

重新构建 maven

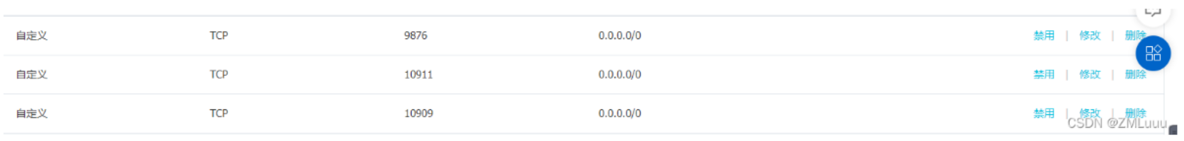
```
mvn clean install
```

5.开放端口

这是因为我们的 RocketMQ 安装在 Linux 中，控制台在 windows，Linux 需要开放端口才能访问，开放 10909 和 9876， 10911端口

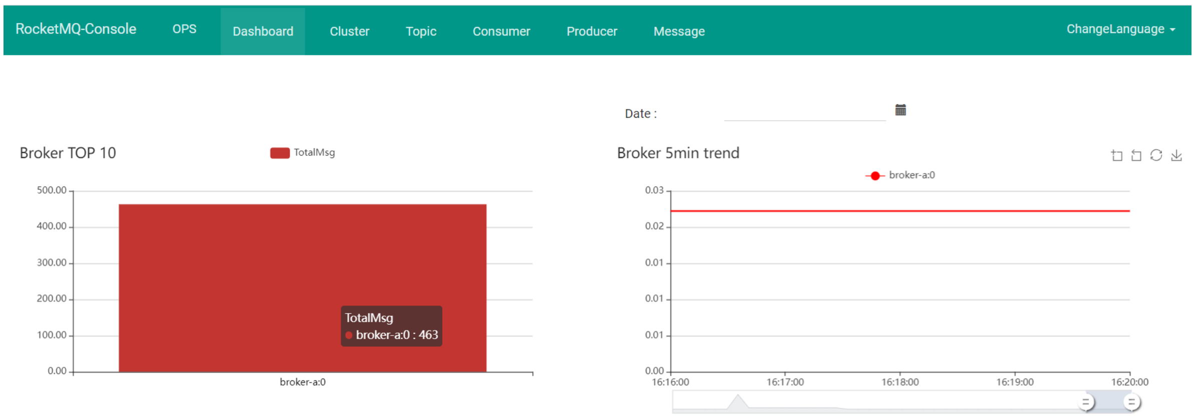
```
firewall-cmd --zone=public --add-port=10909/tcp --permanent
firewall-cmd --zone=public --add-port=10911/tcp --permanent
firewall-cmd --zone=public --add-port=9876/tcp --permanent
systemctl restart firewalld.service
firewall-cmd --reload
```

注意：这一步我用的轻量级应用服务器，所以需在阿里云的防火墙中添加端口



自定义	TCP	9876	0.0.0.0/0	禁用 | 修改 | 删除
自定义	TCP	10911	0.0.0.0/0	禁用 | 修改 | 删除
自定义	TCP	10909	0.0.0.0/0	禁用 | 修改 | 删除

6.重新启动控制台



6.3 java实现消息发送

1.provider的pom.xml中引入依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>
```

2.生产消息

```
package com.ishang.controller;

import org.apache.rocketmq.client.exception.MQBrokerException;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.remoting.exception.RemotingException;
import org.springframework.web.client.RestTemplate;

public class Test {
    public static void main(String[] args) throws InterruptedException,
MQClientException, RemotingException, MQBrokerException {

//          创建消息生产者,分组
        DefaultMQProducer producer = new DefaultMQProducer("myproducer-group");
//          设置NameServer,指向linux上的rocketmq
```

```
        producer.setNamesrvAddr("47.103.92.13:9876");
//        启动生产者
        producer.start();
//        构建消息对象                          消息标题          标签          内容
        Message message = new Message("myTopic","myTag",("Test MQ").getBytes());
//        发送消息，超过超时消息这条消息将不再发送
        SendResult result = producer.send(message,10000);
        System.out.println(result);
//        关闭生产者
        producer.shutdown();

    }
}
```

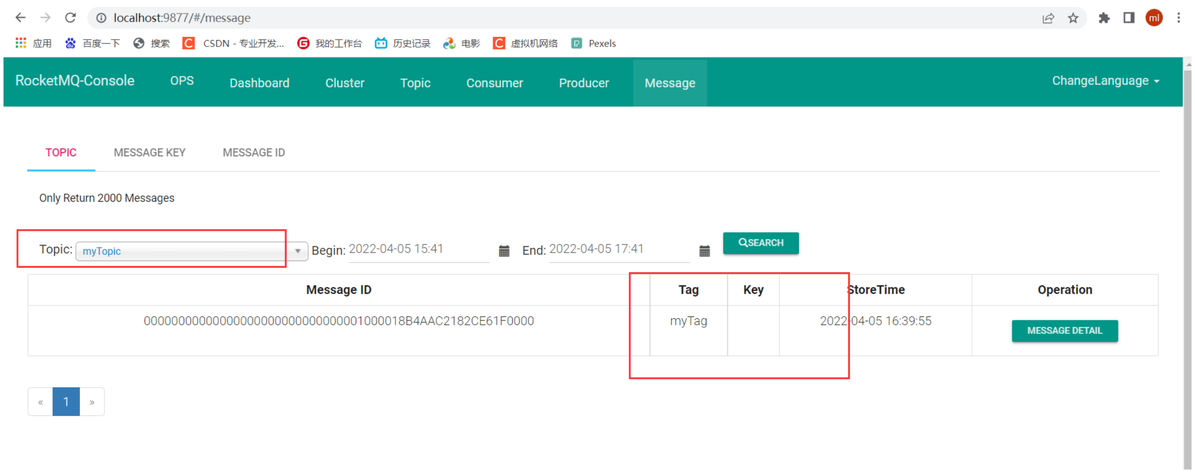
3、直接运行，如果报错 sendDefaultImpl call timeout，可以手动关闭 Linux 防火墙

```
# 关闭防火墙
systemctl stop firewalld
# 查看防火墙状态
firewall-cmd --state
```

或者开放 10911 端口

```
firewall-cmd --zone=public --add-port=10911/tcp --permanent
systemctl restart firewalld.service
firewall-cmd --reload
```

打开 RocketMQ 控制台，可查看消息。



Only Return 2000 Messages

Topic: myTopic Begin: 2022-04-05 15:41 End: 2022-04-05 17:41

Message ID	Tag	Key	StoreTime	Operation
000000000000000000000000000000001000018B4AAC2182CE61F0000	myTag		2022-04-05 16:39:55	MESSAGE DETAIL

6.4 java实现消息消费

1.导入依赖

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>
```

```
package com.ishang.controller;
```

```java
import lombok.extern.slf4j.Slf4j;
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.common.message.MessageExt;

import java.util.List;

import static jdk.nashorn.internal.runtime.regexp.joni.Config.log;
@Slf4j
public class ConsumerTest {

    public static void main(String[] args) throws MQClientException {

//          创建消息消费者
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("myconsumer-
group");
//          设置NameServer
        consumer.setNamesrvAddr("47.103.92.13:9876");
//          指定订阅的主题和标签
        consumer.subscribe("myTopic","*");
//          回调函数，一旦读到myTopic的消息就读取
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
list, ConsumeConcurrentlyContext consumeConcurrentlyContext) {
                log.info("Message=>{}",list);
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
        //启动消费者
        consumer.start();
    }
}
```

6.5SpringBoot整合RocketMQ

provider:

1.导入依赖

```xml
<!--        rocketmq-->
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-spring-boot-starter</artifactId>
            <version>2.1.0</version>
        </dependency>
<!--        rocketmq client-->
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.7.0</version>
        </dependency>
```

2.在application.yml中进行配置

```
rocketmq:
  name-server: 47.103.92.13:9876
  producer:
    group: myprovider
```

3.实体类order

```
package com.ishang.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

@AllArgsConstructor
@Data
@NoArgsConstructor
public class Order {
    private Integer id;
    private String name;
    private String address;
    private Date createDate;
}
```

4.Controller

```
@Autowired
private RocketMQTemplate rocketMQTemplate;

@RequestMapping("/create")
public Order create(){
    Order order = new Order(
            1,
            "张三",
            "攀枝花",
            new Date()
    );

    this.rocketMQTemplate.convertAndSend("myOrder",order);
    return order;
}
```

consumer

1.引入依赖

```
<!--        rocketmq-->
      <dependency>
         <groupId>org.apache.rocketmq</groupId>
         <artifactId>rocketmq-spring-boot-starter</artifactId>
         <version>2.1.0</version>
      </dependency>
      <!--        rocketmq client-->
      <dependency>
         <groupId>org.apache.rocketmq</groupId>
         <artifactId>rocketmq-client</artifactId>
         <version>4.7.0</version>
      </dependency>
```

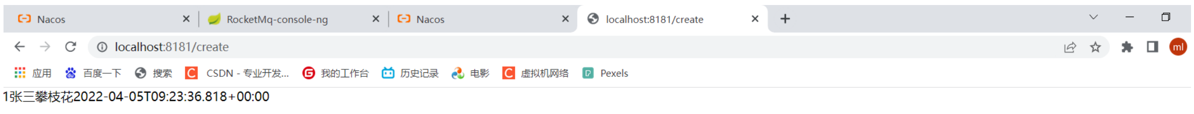
2.同步provider的实体类

```
package com.ishang.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;

@AllArgsConstructor
@Data
@NoArgsConstructor
public class Order {
    private Integer id;
    private String name;
    private String address;
    private Date createDate;
}
```

3.service（因为读取自动读取，不需要手动访问controller）

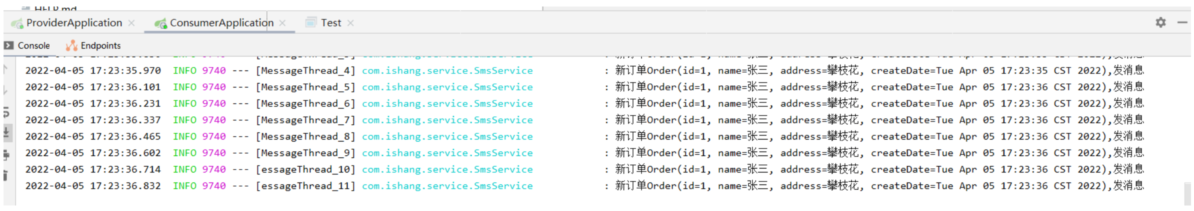
```
package com.ishang.service;

import com.ishang.entity.Order;
import lombok.extern.slf4j.Slf4j;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Service;


@Slf4j
@Service
@RocketMQMessageListener(consumerGroup = "myConsumer",topic = "myOrder")
public class SmsService implements RocketMQListener<Order> {
    @Override
    public void onMessage(Order order) {
        log.info("新订单{},发消息",order);
    }
}
```

访问/create

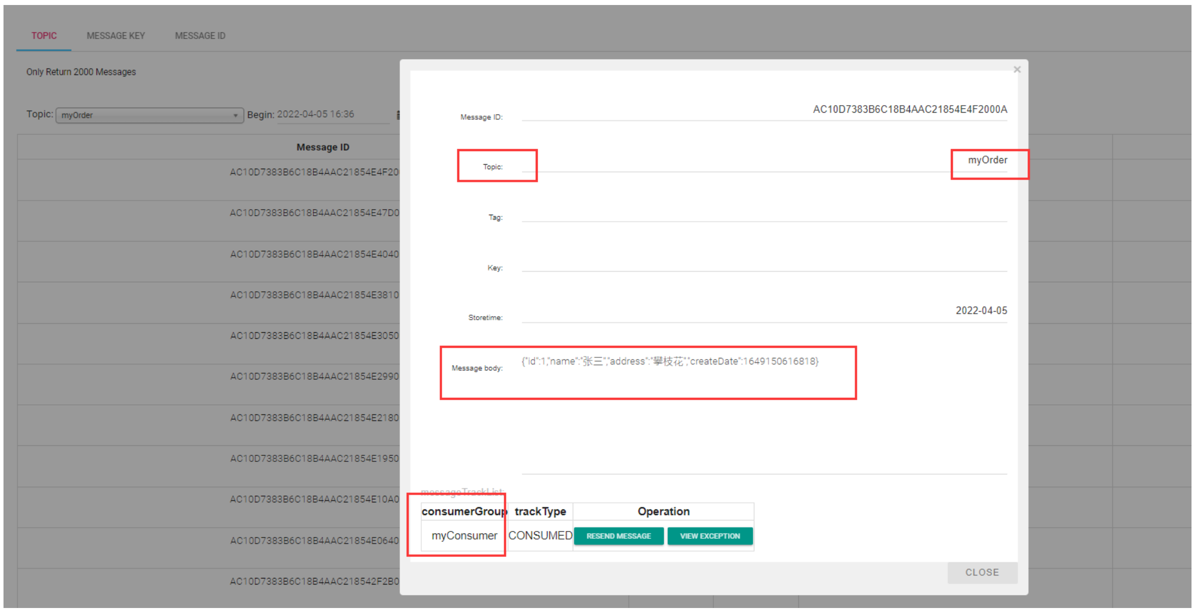


1张三攀枝花2022-04-05T09:23:36.818+00:00

读取出来



```
2022-04-05 17:23:35.970  INFO 9740 --- [MessageThread_4] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:35 CST 2022),发消息
2022-04-05 17:23:36.101  INFO 9740 --- [MessageThread_5] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.231  INFO 9740 --- [MessageThread_6] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.337  INFO 9740 --- [MessageThread_7] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.465  INFO 9740 --- [MessageThread_8] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.602  INFO 9740 --- [MessageThread_9] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.714  INFO 9740 --- [essageThread_10] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
2022-04-05 17:23:36.832  INFO 9740 --- [essageThread_11] com.ishang.service.SmsService            : 新订单Order(id=1, name=张三, address=攀枝花, createDate=Tue Apr 05 17:23:36 CST 2022),发消息
```



Topic: myOrder Begin: 2022-04-05 16:36

Message ID
AC10D7383B6C18B4AAC21854E4F20
AC10D7383B6C18B4AAC21854E47D0
AC10D7383B6C18B4AAC21854E4040
AC10D7383B6C18B4AAC21854E3810
AC10D7383B6C18B4AAC21854E3050
AC10D7383B6C18B4AAC21854E2990
AC10D7383B6C18B4AAC21854E2180
AC10D7383B6C18B4AAC21854E1950
AC10D7383B6C18B4AAC21854E10A0
AC10D7383B6C18B4AAC21854E0640
AC10D7383B6C18B4AAC218542F2B0

Message ID: AC10D7383B6C18B4AAC21854E4F2000A

Topic: myOrder

Tag:

Key:

Storetime: 2022-04-05

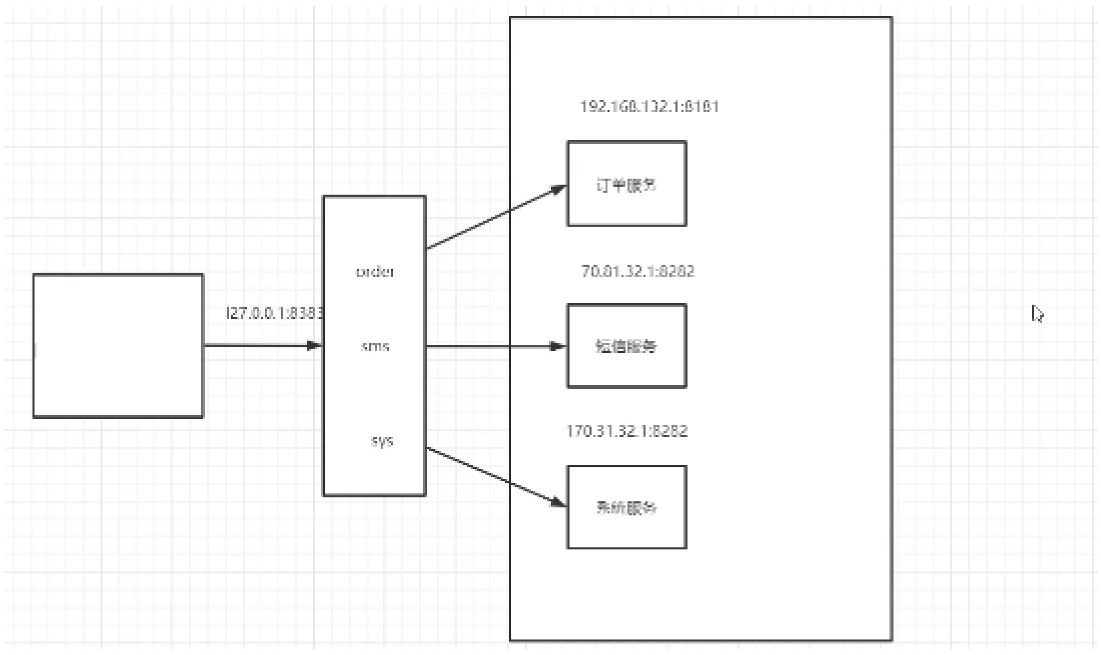
Message body: {"id":1,"name":"张三","address":"攀枝花","createDate":1649150616818}

messageTrackList:

consumerGroup	trackType	Operation
myConsumer	CONSUMED	RESEND MESSAGE VIEW EXCEPTION

7.服务网关

给不同的服务提供统一的入口，通过统一的网关端口号+不同服务名即可，不用记每个服务的ip+端口号

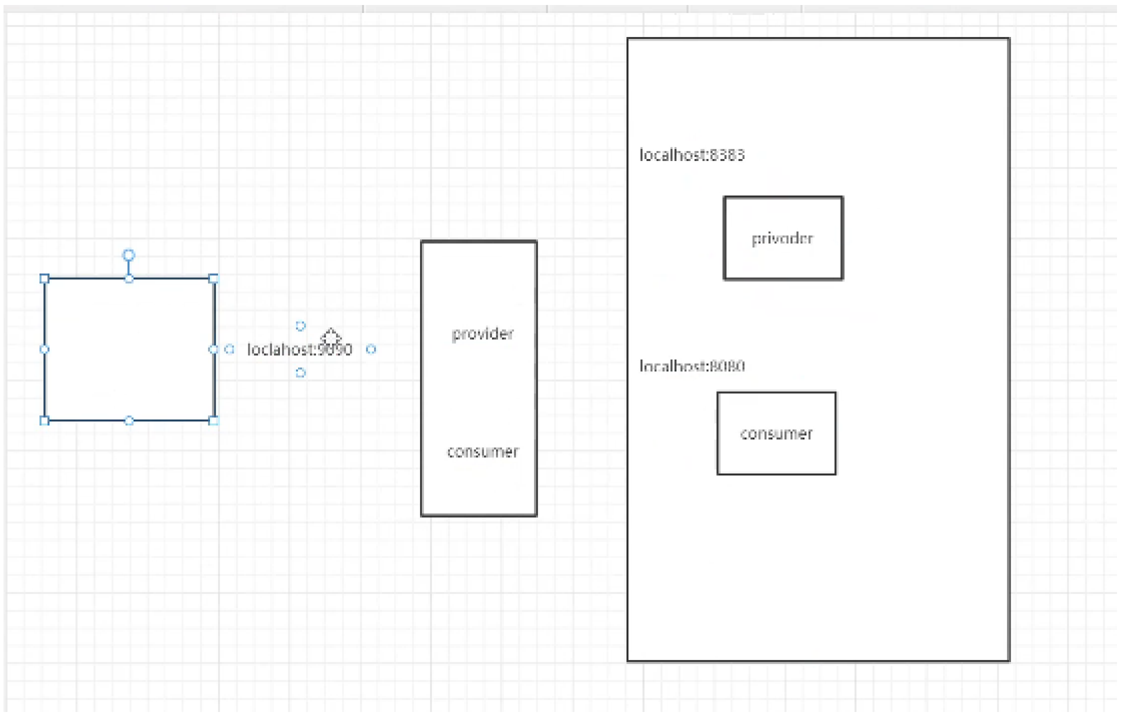


7.1什么是Spring Cloud Gateway

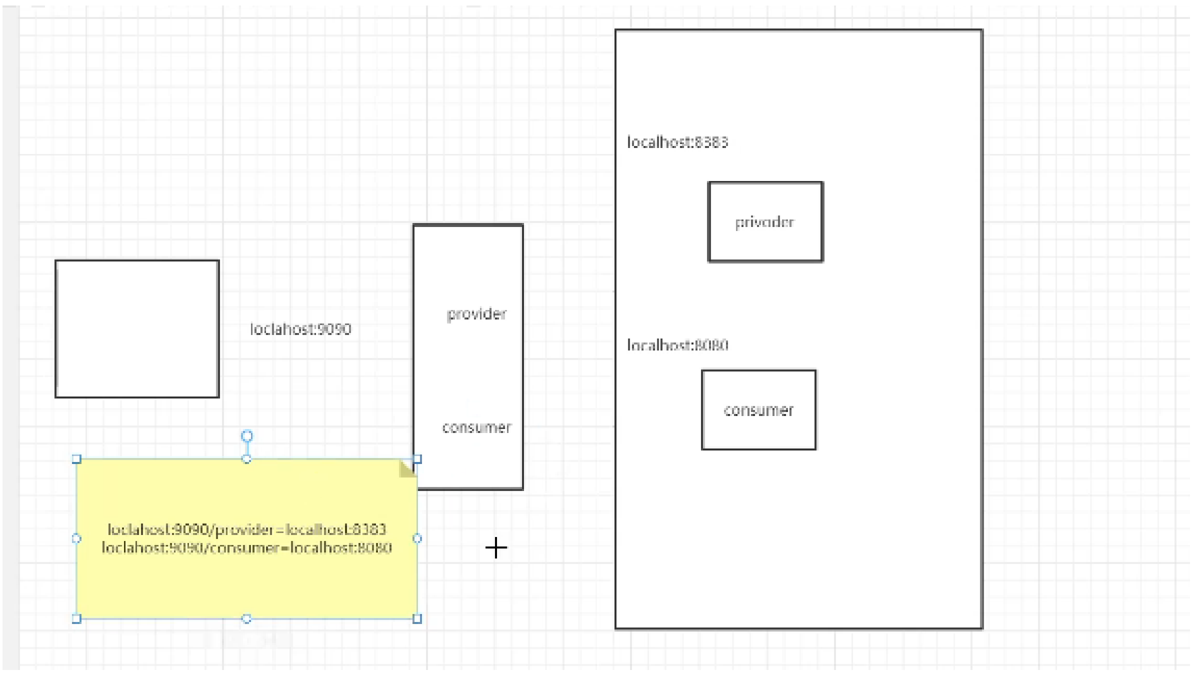
微服务第二代网关，第一代是Netiflix的组件Zuul，Gateway的性能更加强大。

SpringCloud Gateway是spring cloud官方提供的新一代网关，spring cloud 阿里巴巴没有提供网关组件，直接用的是Spring Cloud Gateway

spring cloud gateway是基于Netty、WebFlux开发的，与Servlet不兼容，使用时不能引入Spring MVC依赖，不能发布war包，只能发布jar包



快速上手


localhost:8383
privader
loclahost:9090
provider
consumer
localhost:8080
consumer
loclahost:9090/provider=localhost:8383
loclahost:9090/consumer=localhost:8080

1.创建gateway模块

2.创建依赖pom.xml

```
<!--        gateway-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
            <version>2.2.3.RELEASE</version>
        </dependency>
```

注意：由于geteway模块需要引入父模块的依赖，如果父模块中有spring-boot-start-web需要将父模块中的这个依赖分别放入子模块的pom.xml中，否则网关不生效

2.配置application.yml

```
server:
  port: 8010
spring:
  application:
    name: gateway
#    开启网关
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
#          路由
      routes:
        - id: provider_route #自己任意取
          uri: http://localhost:8181 #真实路径
          predicates:
            - Path=/provider/** #映射名
          filters:
            - StripPrefix=1
        - id: consumer_route #自己任意取
          uri: http://localhost:8484 #真实路径
          predicates:
            - Path=/consumer/** #映射名
          filters:
            - StripPrefix=1
```

这是访问就可用：http://localhost:8010/provider/find访问就接口

7.2 nacos整合gateway

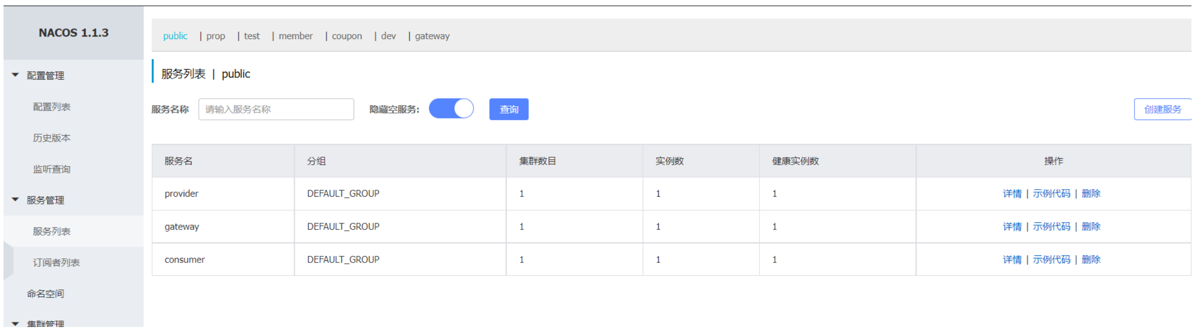
上面这种做法其实没有用到 nacos ， 现在我们让 gateway 直接去 nacos 中发现服务，配置更加简单了。

1.在gateway模块的pom.xml导入依赖

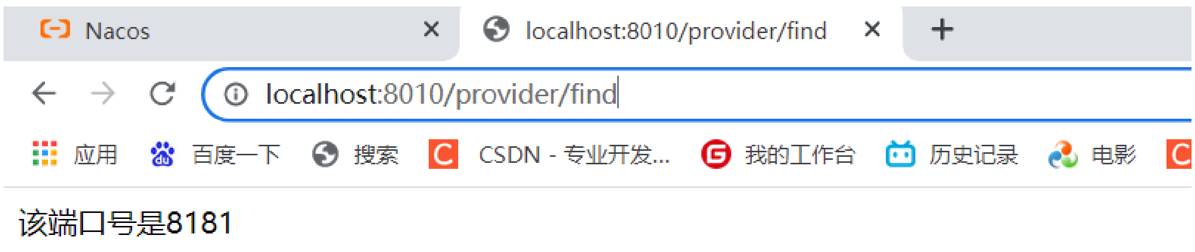
```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
```

2.application.yml

```yaml
server:
  port: 8010
spring:
  application:
    name: gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
```



服务名	分组	集群数目	实例数	健康实例数	操作
provider	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除
gateway	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除
consumer	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除



该端口号是8181

8.配置中心

Nacos 提供了配置中心的服务，即可以将微服务的配置文件直接在 Nacos 中进行配置管理。使配置文件可以实现动态更新，环境隔离

1.新建config工程,引入父依赖

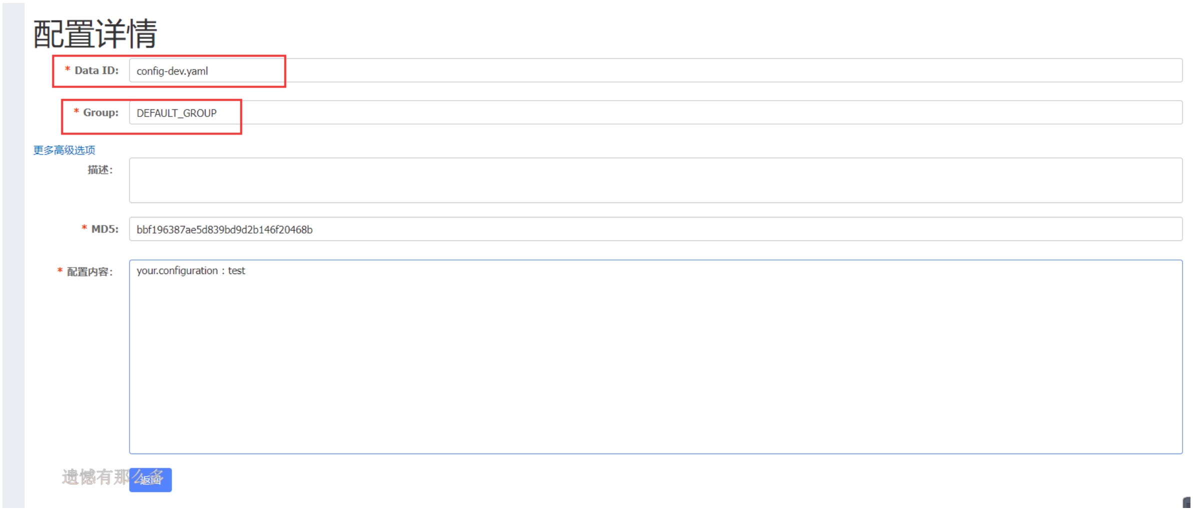
1.pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

2.创建bootstrap.yml

```yaml
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yaml   //后缀名
  application:
    name: config
  profiles:
    active: dev
        //根据 config-dev.yaml查找nacos中的配置信息
```

3.在nacos中添加配置，Data ID 的写法必须和 bootstrap.yml 保持一致


配置详情

* Data ID: config-dev.yaml

* Group: DEFAULT_GROUP

更多高级选项

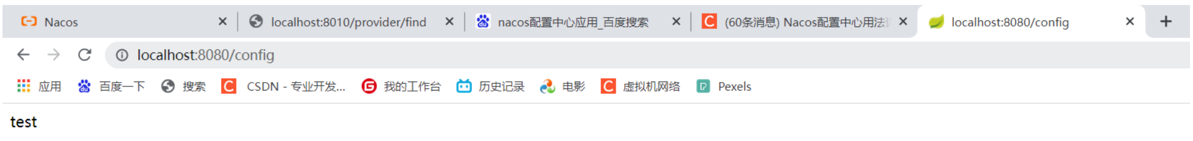
描述:

* MD5: bbf196387ae5d839bd9d2b146f20468b

* 配置内容: your.configuration : test

3.创建controller测试

```java
package com.ishang.controller;


import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ConfigController {

    @Value("${your.configuration}")
    private String configuration;
    @GetMapping("/config")
    public String config(){
        return this.configuration;
    }
}
```



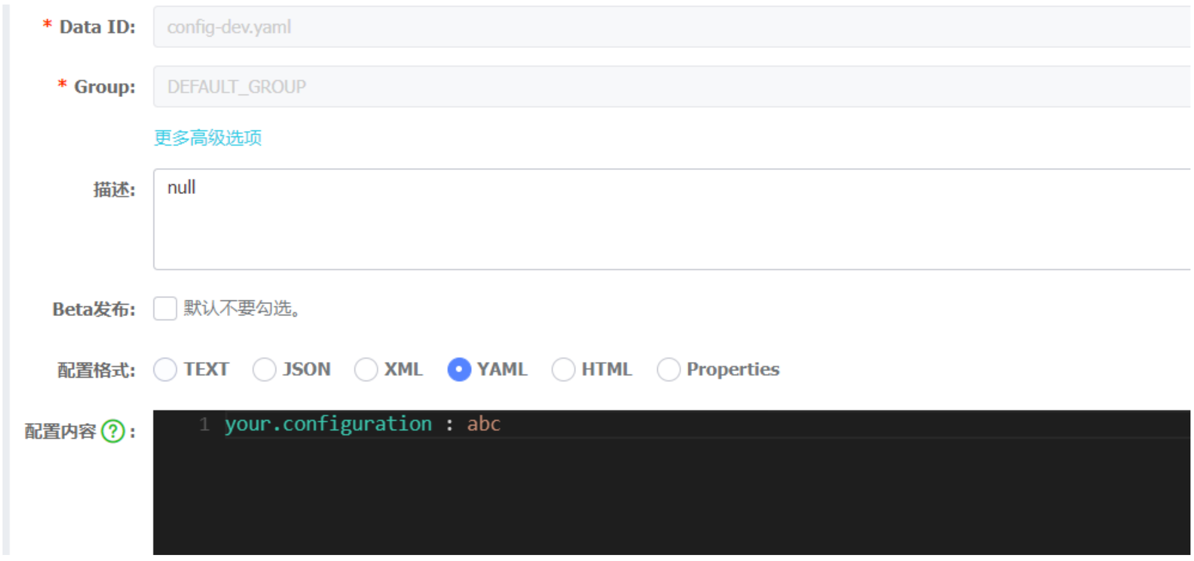
```
test
```

8.1动态刷新

在conreoller添加@RefreshScop实现动态刷新

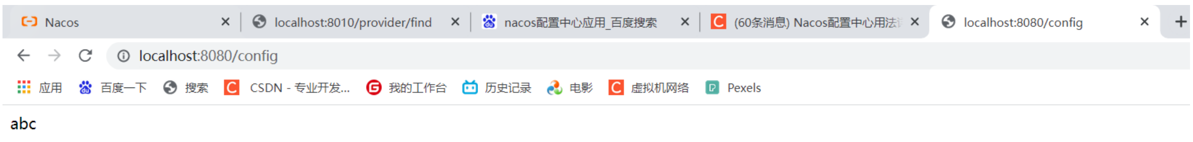
```java
package com.ishang.controller;


import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RefreshScope
public class ConfigController {

    @Value("${your.configuration}")
    private String configuration;
    @GetMapping("/config")
    public String config(){
        return this.configuration;
    }
}
```

在nacos中修改配置即可自动刷新，不用再重启服务



* Data ID:	config-dev.yaml
* Group:	DEFAULT_GROUP
	更多高级选项
描述:	null
Beta发布:	☐ 默认不要勾选。
配置格式:	TEXT JSON XML [✓ YAML] HTML Properties
配置内容:	

```yaml
your.configuration : abc
```



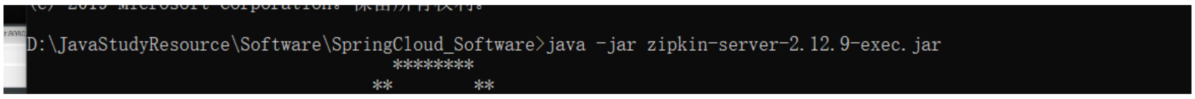
```
abc
```

9.Zipkin服务追踪

zipkin将请求记录下来，当前请求了哪些服务，成功还是失败，请求了服务的哪些接口全部记录下来，方便我们去进行排查。如果遇到问题，尤其是服务调用比较复杂，多个服务互相调用时，出现问题，解决需要知道这个请求经给哪些服务，zipkin记录下来，帮助我们排查问题。

1.下载zipkin应用

2.启动



```
D:\JavaStudyResource\Software\SpringCloud_Software>java -jar zipkin-server-2.12.9-exec.jar
                                  ********
                                **         **
```

创建zipkin项目,导入父依赖

1.pom.xml

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

2.application.yml

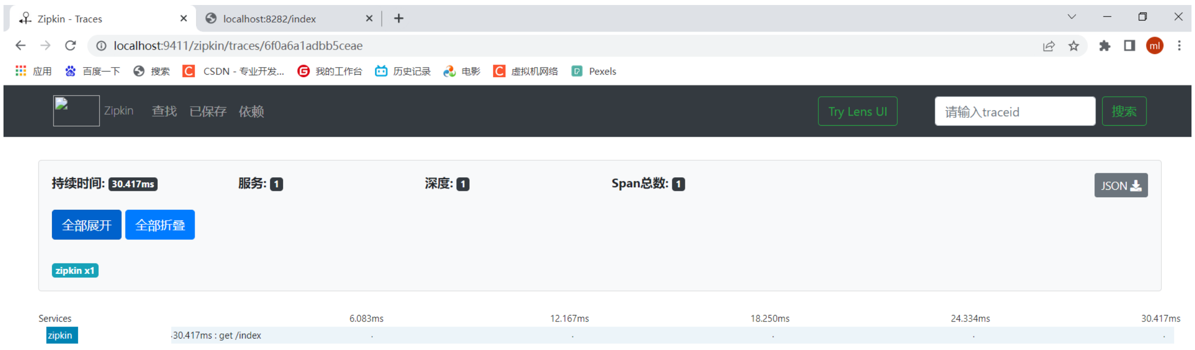
```yaml
server:
  port: 8282
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      # 抽样率，默认 0.1，90%数据都会丢弃
      probability: 1.0
  application:
    name: zipkin
```

3.controller

```java
package com.ishang.controller;


import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class indexcontroller {

    private String name="你好";

    @RequestMapping("/index")
    public String named(){
        return this.name;
    }
}
```

4.启动应用并访问

zipkin即可监控到


Zipkin - Traces
localhost:8282/index
localhost:9411/zipkin/traces/6f0a6a1adbb5ceae
应用 百度一下 搜索 CSDN - 专业开发... 我的工作台 历史记录 电影 虚拟机网络 Pexels
Zipkin 查找 已保存 依赖
Try Lens UI
请输入traceid
搜索
持续时间: 30.417ms 服务: 1 深度: 1 Span总数: 1
JSON
全部展开 全部折叠
zipkin x1
Services 6.083ms 12.167ms 18.250ms 24.334ms 30.417ms
zipkin
30.417ms : get /index

10.分布式事务

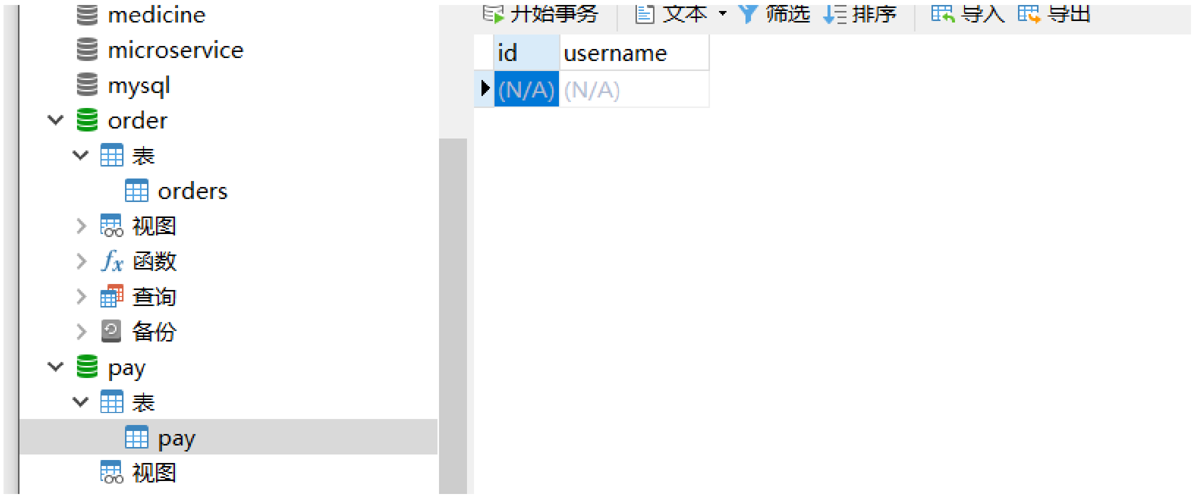
在分布式项目中，两个数据库实现的是同一个业务，事务统一进行回滚

10.1模拟分布式事务异常

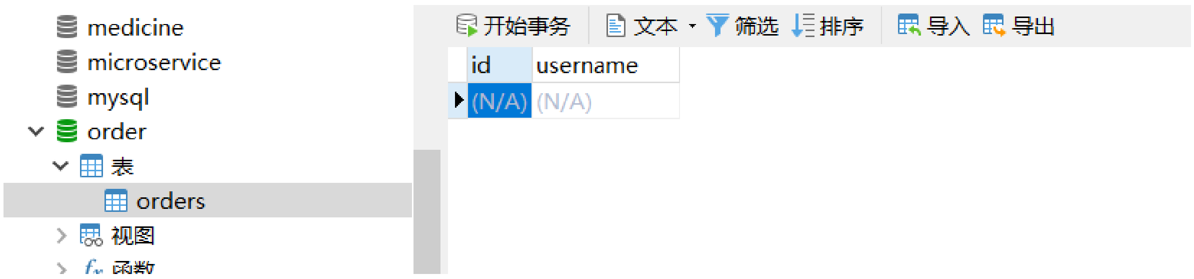
1.创建pay和order模块，导入父依赖，两个模块中加入依赖（这里模拟用jdbc，也可用mybatis...）

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

2.创建两个数据库pay、order



id	username
(N/A)	(N/A)



id	username
(N/A)	(N/A)

3.配置数据源

```
server:
  port: 8686
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/pay?useUnicode=true&characterEncoding=UTF-
8&serverTimezone=Asia/Shanghai
    username: root
    password: 123456
  application:
    name: pay
```

```
server:
  port: 8585
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/order?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shjavaanghai
    username: root
    password: 123456
  application:
    name: order
```

4.目的是当订单表中存入客户信息时，支付表中也存入

创建service

pay:

```
package com.ishang.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.RequestMapping;

@Service
public class payservice {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void save(){
        this.jdbcTemplate.update("insert  into pay(username) values ('张三')");
    }
}
```

order

```
package com.ishang.service;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Service;

@Service
public class orderservice {

    @Autowired
    private JdbcTemplate jdbcTemplate;
    public void save(){
        this.jdbcTemplate.update("insert into orders(username) values('张
三'java)");
    }
}
```

5.创建controller，利用RestTemplate，连接两个服务，在启动类中添加

```
package com.ishang;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class OrderApplication {
    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();

    }
}
```

controller:

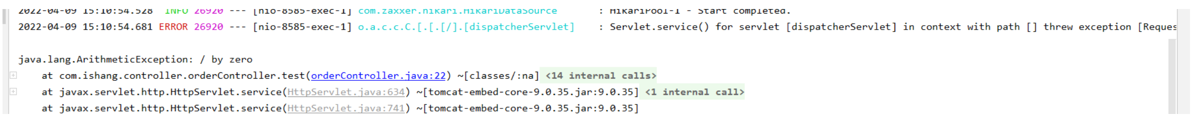
paycontroller:

```
package com.ishang.controller;
import com.ishang.service.payservice;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class paycontroller {

    @Autowired
    private payservice service;

    @RequestMapping("/test")
    public String test(){
        this.service.save();
        return "success";
    }
}
```

ordercontroller

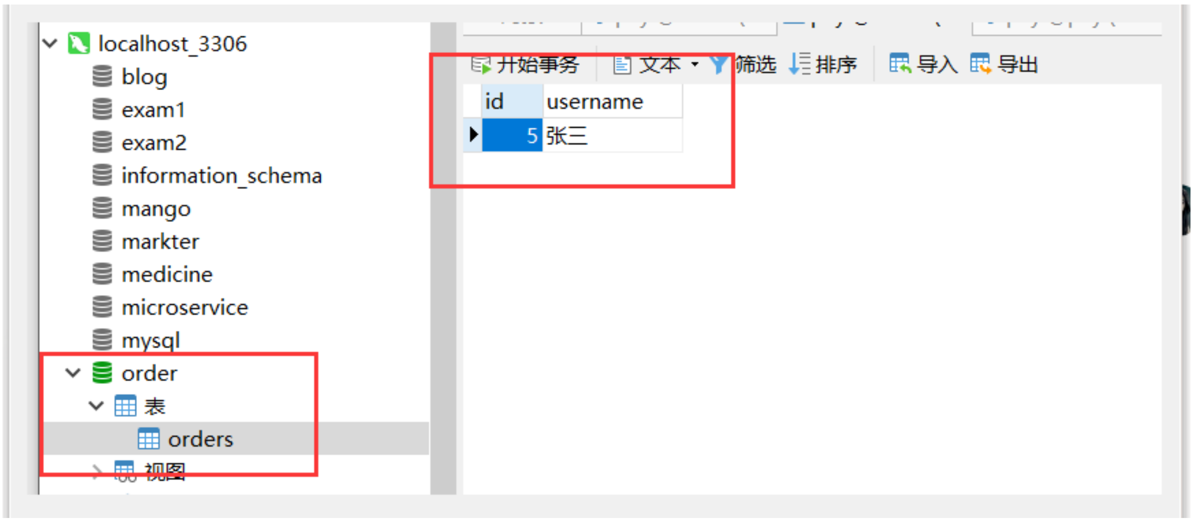
```
package com.ishang.controller;


import com.ishang.service.orderservice;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class orderController {
        @Autowired
        private orderservice orderservice;

        @Autowired
        private RestTemplate restTemplate;

        @RequestMapping("/test")
        public String test(){
            this.orderservice.save();
            int i=10/0;

 this.restTemplate.getForObject("http://localhost:8686/test",String.class);
            return "success";
        }
}
```

此时调用接口，会报错

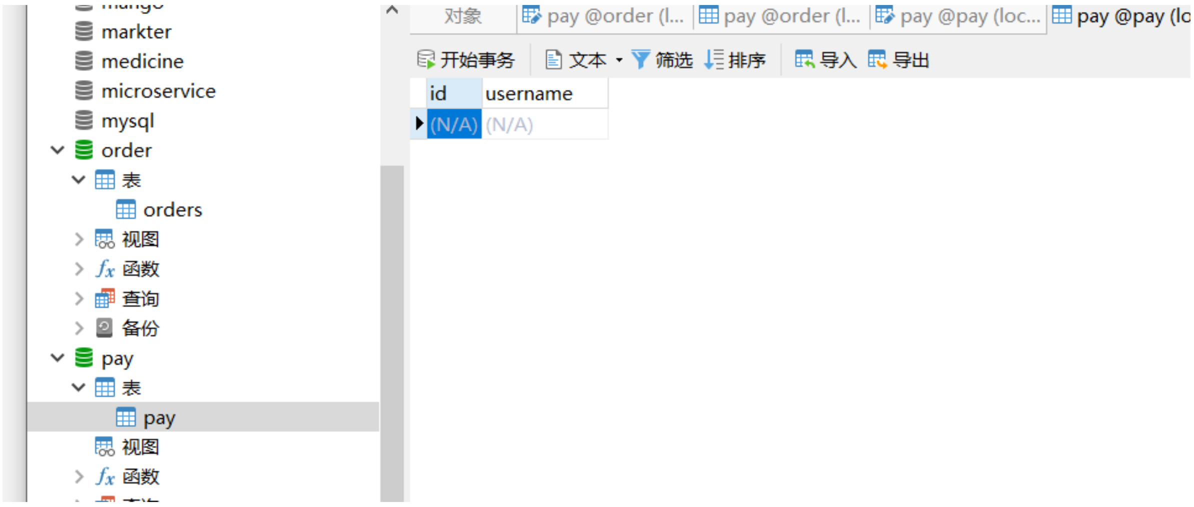


```
2022-04-09 15:10:54.528  INFO 26920 --- [nio-8585-exec-1] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2022-04-09 15:10:54.681 ERROR 26920 --- [nio-8585-exec-1] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Reques

java.lang.ArithmeticException: / by zero
    at com.ishang.controller.orderController.test(orderController.java:22) ~[classes/:na] <14 internal calls>
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:634) ~[tomcat-embed-core-9.0.35.jar:9.0.35] <1 internal call>
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:741) ~[tomcat-embed-core-9.0.35.jar:9.0.35]
```

导致pay表信息存不进去



id	username
5	张三



id	username
(N/A)	(N/A)

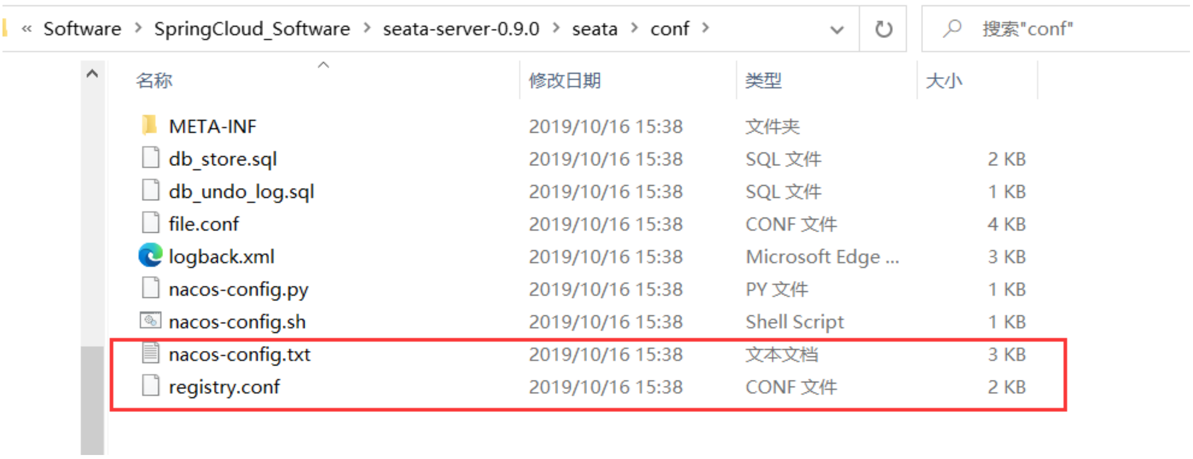
分布式异常模拟结束，Order 存储完成之后，出现异常，会导致 Pay 无法存储，但是 Order 数据库不会进行回滚。

10.2 Seata解决

1、下载 Seate 0.9.0

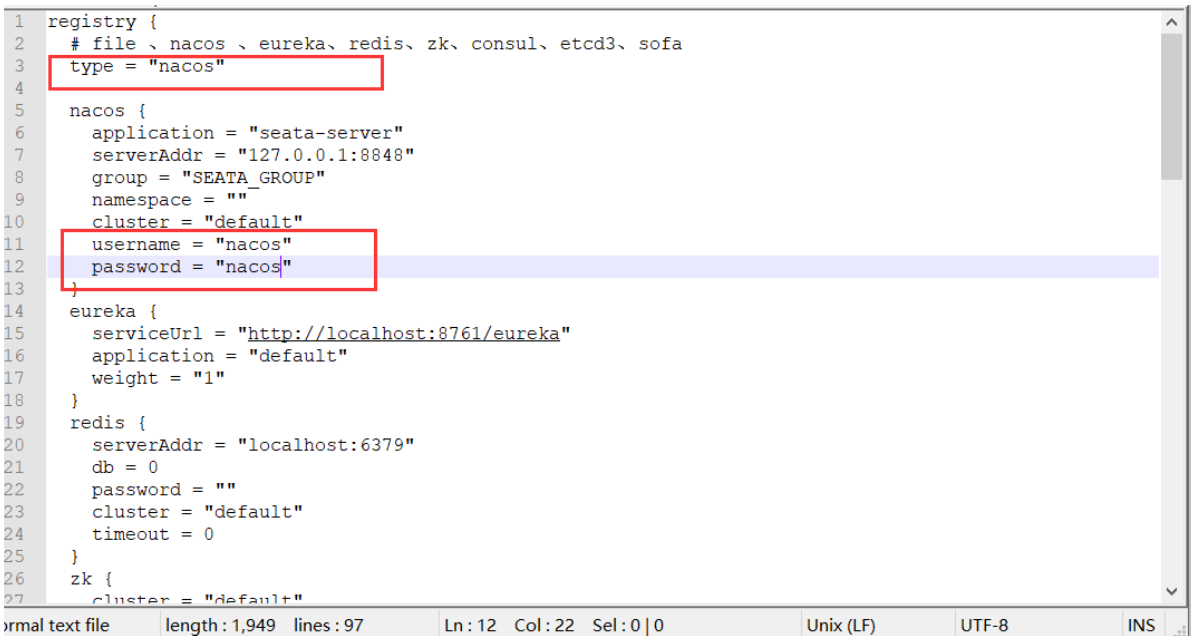
https://github.com/seata/seata/releases

2.解压，修改两个文件

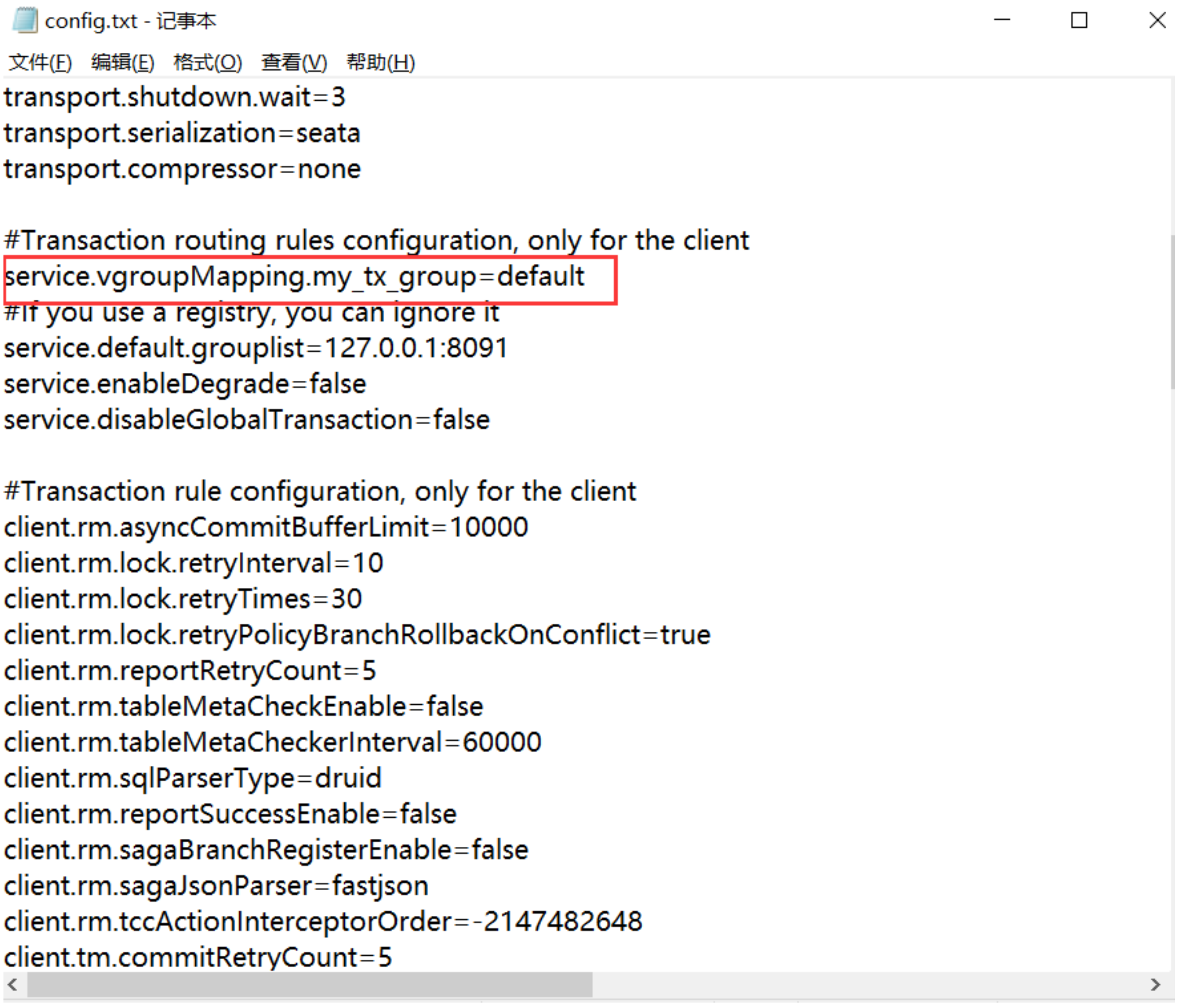

« Software › SpringCloud_Software › seata-server-0.9.0 › seata › conf ›
搜索"conf"

名称	修改日期	类型	大小
META-INF	2019/10/16 15:38	文件夹	
db_store.sql	2019/10/16 15:38	SQL 文件	2 KB
db_undo_log.sql	2019/10/16 15:38	SQL 文件	1 KB
file.conf	2019/10/16 15:38	CONF 文件	4 KB
logback.xml	2019/10/16 15:38	Microsoft Edge ...	3 KB
nacos-config.py	2019/10/16 15:38	PY 文件	1 KB
nacos-config.sh	2019/10/16 15:38	Shell Script	1 KB
nacos-config.txt	2019/10/16 15:38	文本文档	3 KB
registry.conf	2019/10/16 15:38	CONF 文件	2 KB

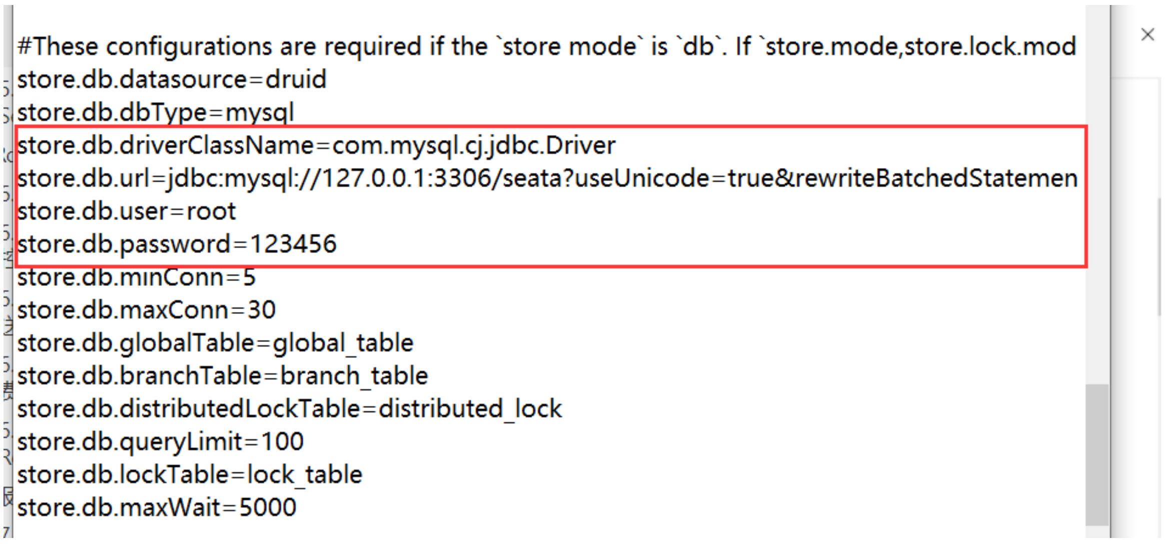
regisry.conf：修改一下配置


```
registry {
  # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
  type = "nacos"

  nacos {
    application = "seata-server"
    serverAddr = "127.0.0.1:8848"
    group = "SEATA_GROUP"
    namespace = ""
    cluster = "default"
    username = "nacos"
    password = "nacos"
  }
  eureka {
    serviceUrl = "http://localhost:8761/eureka"
    application = "default"
    weight = "1"
  }
  redis {
    serverAddr = "localhost:6379"
    db = 0
    password = ""
    cluster = "default"
    timeout = 0
  }
  zk {
    cluster = "default"
```
rmal text file   length : 1,949   lines : 97   Ln : 12   Col : 22   Sel : 0 | 0   Unix (LF)   UTF-8   INS

修改config.txt

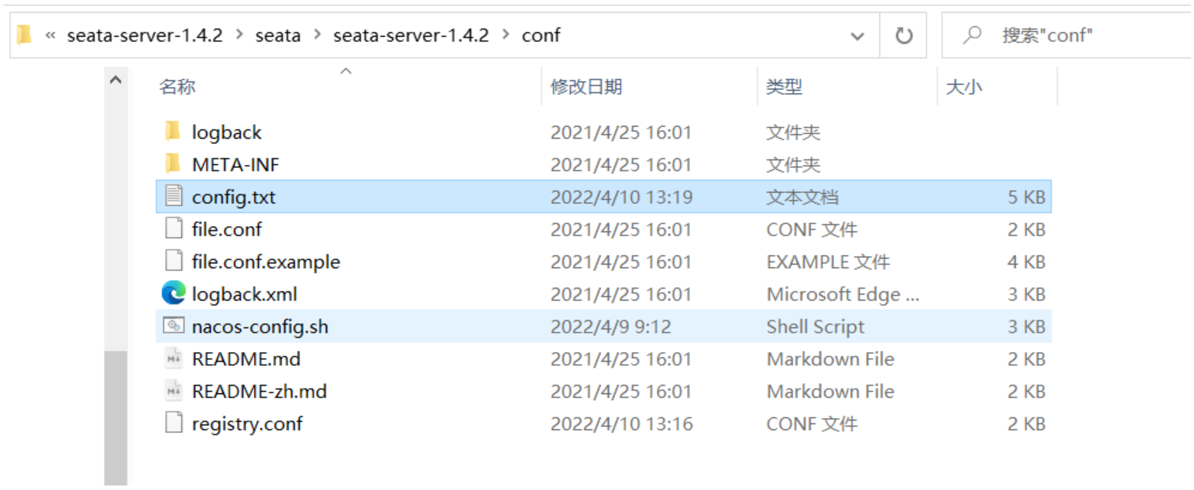

config.txt - 记事本

```
transport.shutdown.wait=3
transport.serialization=seata
transport.compressor=none

#Transaction routing rules configuration, only for the client
service.vgroupMapping.my_tx_group=default
#If you use a registry, you can ignore it
service.default.grouplist=127.0.0.1:8091
service.enableDegrade=false
service.disableGlobalTransaction=false

#Transaction rule configuration, only for the client
client.rm.asyncCommitBufferLimit=10000
client.rm.lock.retryInterval=10
client.rm.lock.retryTimes=30
client.rm.lock.retryPolicyBranchRollbackOnConflict=true
client.rm.reportRetryCount=5
client.rm.tableMetaCheckEnable=false
client.rm.tableMetaCheckerInterval=60000
client.rm.sqlParserType=druid
client.rm.reportSuccessEnable=false
client.rm.sagaBranchRegisterEnable=false
client.rm.sagaJsonParser=fastjson
client.rm.tccActionInterceptorOrder=-2147482648
client.tm.commitRetryCount=5
```



```
#These configurations are required if the `store mode` is `db`. If `store.mode,store.lock.mod
store.db.datasource=druid
store.db.dbType=mysql
store.db.driverClassName=com.mysql.cj.jdbc.Driver
store.db.url=jdbc:mysql://127.0.0.1:3306/seata?useUnicode=true&rewriteBatchedStatemen
store.db.user=root
store.db.password=123456
store.db.minConn=5
store.db.maxConn=30
store.db.globalTable=global_table
store.db.branchTable=branch_table
store.db.distributedLockTable=distributed_lock
store.db.queryLimit=100
store.db.lockTable=lock_table
store.db.maxWait=5000
```

3.将config.txt和nacos-config.sh放在一下位置



« seata-server-1.4.2 › seata › seata-server-1.4.2 › conf   搜索"conf"

名称	修改日期	类型	大小
logback	2021/4/25 16:01	文件夹	
META-INF	2021/4/25 16:01	文件夹	
config.txt	2022/4/10 13:19	文本文档	5 KB
file.conf	2021/4/25 16:01	CONF 文件	2 KB
file.conf.example	2021/4/25 16:01	EXAMPLE 文件	4 KB
logback.xml	2021/4/25 16:01	Microsoft Edge ...	3 KB
nacos-config.sh	2022/4/9 9:12	Shell Script	3 KB
README.md	2021/4/25 16:01	Markdown File	2 KB
README-zh.md	2021/4/25 16:01	Markdown File	2 KB
registry.conf	2022/4/10 13:16	CONF 文件	2 KB

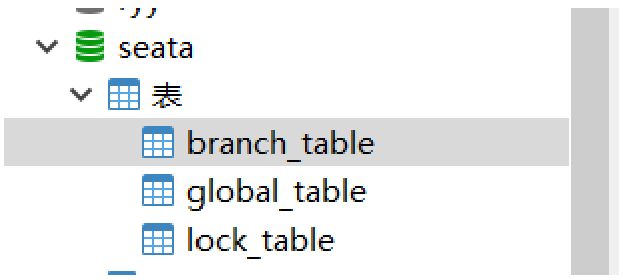
config.txt 和nacos-config.sh下载地址：

https://github.com/seata/seata/tree/develop/script/config-center
https://github.com/seata/seata/tree/develop/script/config-center/nacos

4.创建seata数据库，运行一下脚本

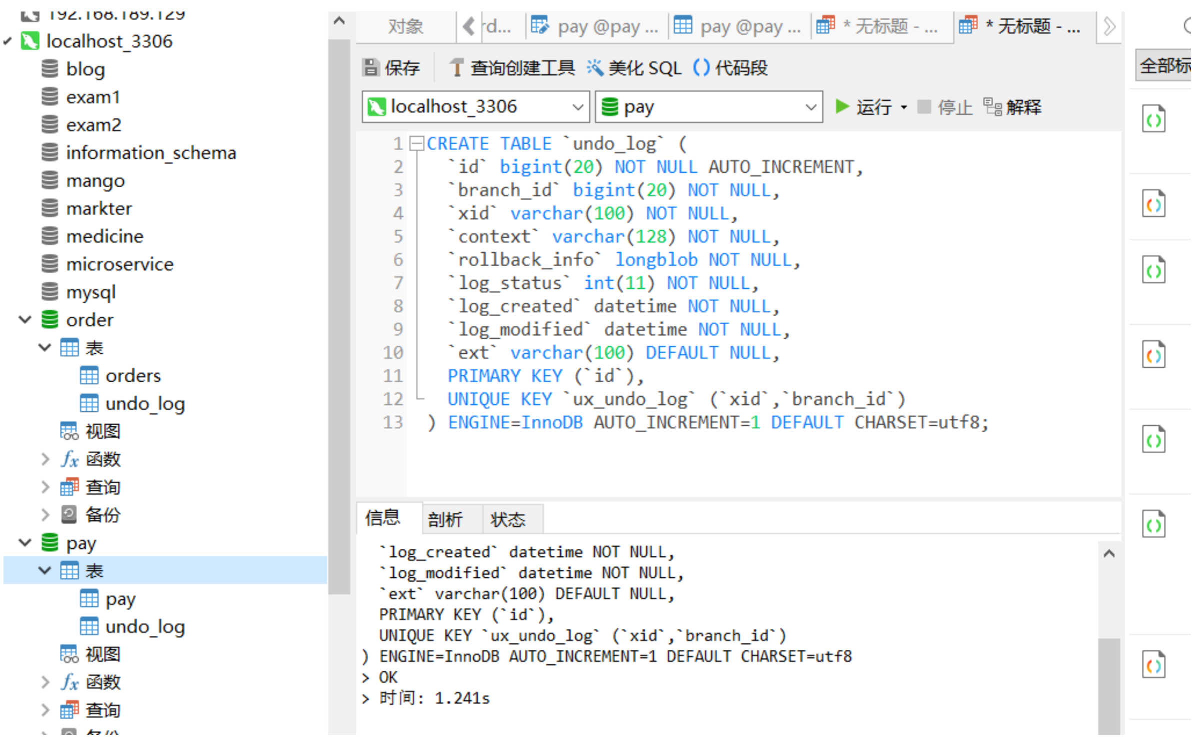
```
-- -------------------------------- The script used when storeMode is 'db' ------
--------------------------
-- the table to store GlobalSession data
CREATE TABLE IF NOT EXISTS `global_table`
(
    `xid`                       VARCHAR(128) NOT NULL,
    `transaction_id`            BIGINT,
    `status`                    TINYINT      NOT NULL,
    `application_id`            VARCHAR(32),
    `transaction_service_group` VARCHAR(32),
    `transaction_name`          VARCHAR(128),
    `timeout`                   INT,
    `begin_time`                BIGINT,
    `application_data`          VARCHAR(2000),
    `gmt_create`                DATETIME,
    `gmt_modified`              DATETIME,
    PRIMARY KEY (`xid`),
    KEY `idx_status_gmt_modified` (`status` , `gmt_modified`),
    KEY `idx_transaction_id` (`transaction_id`)
) ENGINE = InnoDB
```

```sql
  DEFAULT CHARSET = utf8mb4;

-- the table to store BranchSession data
CREATE TABLE IF NOT EXISTS `branch_table`
(
    `branch_id`         BIGINT       NOT NULL,
    `xid`               VARCHAR(128) NOT NULL,
    `transaction_id`    BIGINT,
    `resource_group_id` VARCHAR(32),
    `resource_id`       VARCHAR(256),
    `branch_type`       VARCHAR(8),
    `status`            TINYINT,
    `client_id`         VARCHAR(64),
    `application_data`  VARCHAR(2000),
    `gmt_create`        DATETIME(6),
    `gmt_modified`      DATETIME(6),
    PRIMARY KEY (`branch_id`),
    KEY `idx_xid` (`xid`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

-- the table to store lock data
CREATE TABLE IF NOT EXISTS `lock_table`
(
    `row_key`        VARCHAR(128) NOT NULL,
    `xid`            VARCHAR(128),
    `transaction_id` BIGINT,
    `branch_id`      BIGINT       NOT NULL,
    `resource_id`    VARCHAR(256),
    `table_name`     VARCHAR(32),
    `pk`             VARCHAR(36),
    `status`         TINYINT      NOT NULL DEFAULT '0' COMMENT '0:locked
,1:rollbacking',
    `gmt_create`     DATETIME,
    `gmt_modified`   DATETIME,
    PRIMARY KEY (`row_key`),
    KEY `idx_status` (`status`),
    KEY `idx_branch_id` (`branch_id`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

CREATE TABLE IF NOT EXISTS `distributed_lock`
(
    `lock_key`       CHAR(20) NOT NULL,
    `lock_value`     VARCHAR(20) NOT NULL,
    `expire`         BIGINT,
    primary key (`lock_key`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4;

INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('HandleAllSession', ' ', 0);
```



5.在pay和order服务中添加undo_log表

```sql
CREATE TABLE undo_log (
id bigint(20) NOT NULL AUTO_INCREMENT,
branch_id bigint(20) NOT NULL,
xid varchar(100) NOT NULL,
context varchar(128) NOT NULL,
rollback_info longblob NOT NULL,
log_status int(11) NOT NULL,
log_created datetime NOT NULL,
log_modified datetime NOT NULL,
ext varchar(100) DEFAULT NULL,
PRIMARY KEY (id),
UNIQUE KEY ux_undo_log (xid,branch_id)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

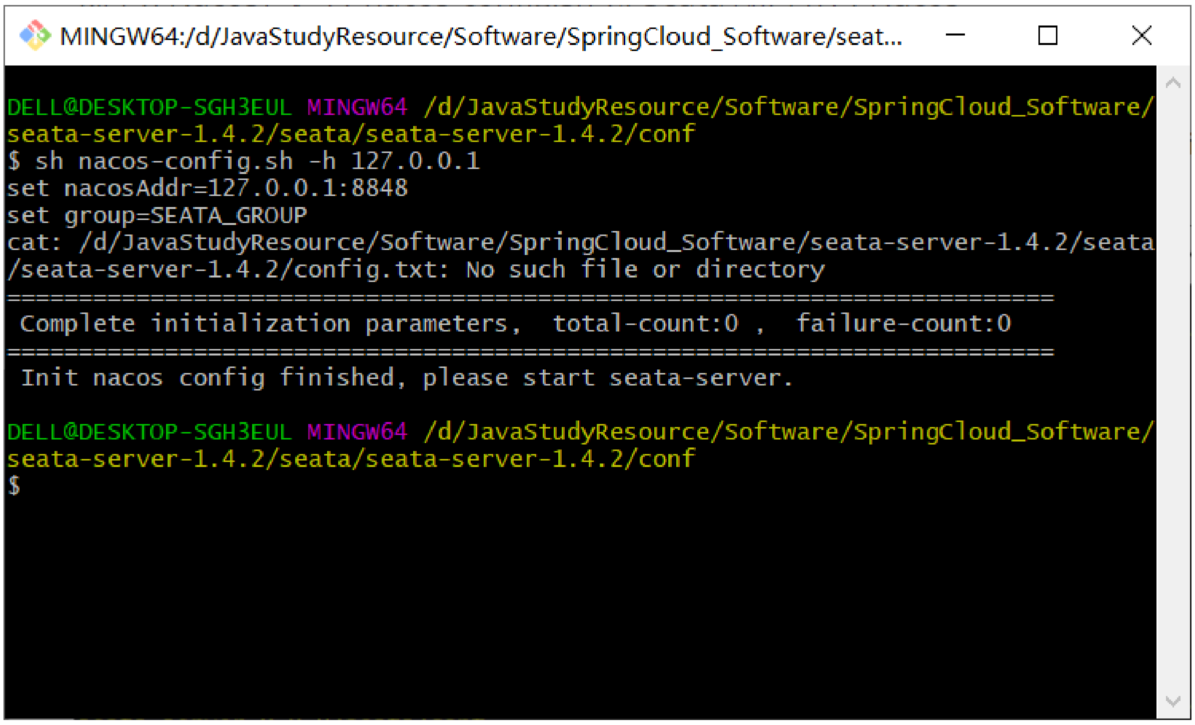


```sql
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `context` varchar(128) NOT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  `ext` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

```
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  `ext` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8
> OK
> 时间: 1.241s
```

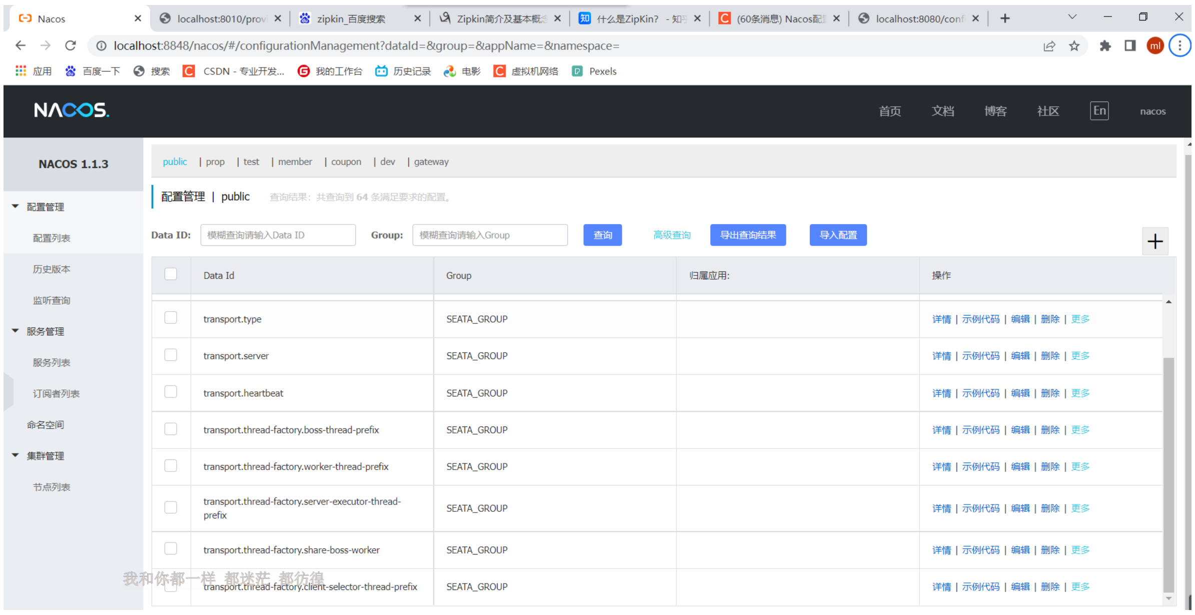
6.启动 Nacos，运行 nacos-config.sh 将 Seata 配置导入 Nacos

进入 conf，右键 Git Bash Here



```
MINGW64:/d/JavaStudyResource/Software/SpringCloud_Software/seat...
DELL@DESKTOP-SGH3EUL MINGW64 /d/JavaStudyResource/Software/SpringCloud_Software/
seata-server-1.4.2/seata/seata-server-1.4.2/conf
$ sh nacos-config.sh -h 127.0.0.1
set nacosAddr=127.0.0.1:8848
set group=SEATA_GROUP
cat: /d/JavaStudyResource/Software/SpringCloud_Software/seata-server-1.4.2/seata
/seata-server-1.4.2/config.txt: No such file or directory
=========================================================================
 Complete initialization parameters,  total-count:0 ,  failure-count:0
=========================================================================
 Init nacos config finished, please start seata-server.

DELL@DESKTOP-SGH3EUL MINGW64 /d/JavaStudyResource/Software/SpringCloud_Software/
seata-server-1.4.2/seata/seata-server-1.4.2/conf
$
```

执行成功，刷新 Nacos，配置加入



localhost:8848/nacos/#/configurationManagement?dataId=&group=&appName=&namespace=

NACOS 1.1.3

public | prop | test | member | coupon | dev | gateway

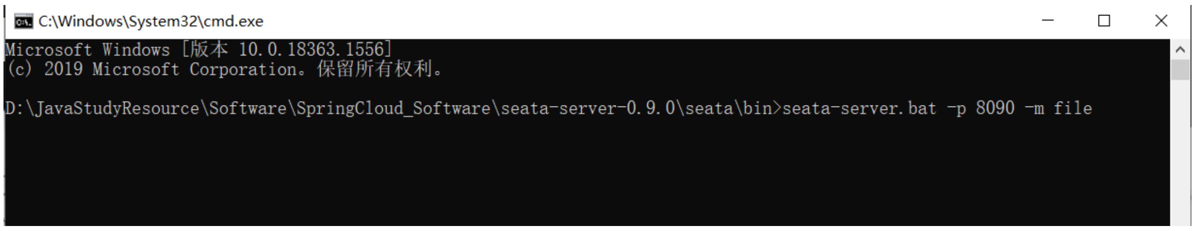
配置管理 | public 查询结果：共查询到 64 条满足要求的配置。

Data Id	Group	归属应用:	操作
transport.type	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.server	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.heartbeat	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.thread-factory.boss-thread-prefix	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.thread-factory.worker-thread-prefix	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.thread-factory.server-executor-thread-prefix	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.thread-factory.share-boss-worker	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多
transport.thread-factory.client-selector-thread-prefix	SEATA_GROUP		详情 | 示例代码 | 编辑 | 删除 | 更多

7、启动 Seata Server， **JDK 8 以上环境无法启动**

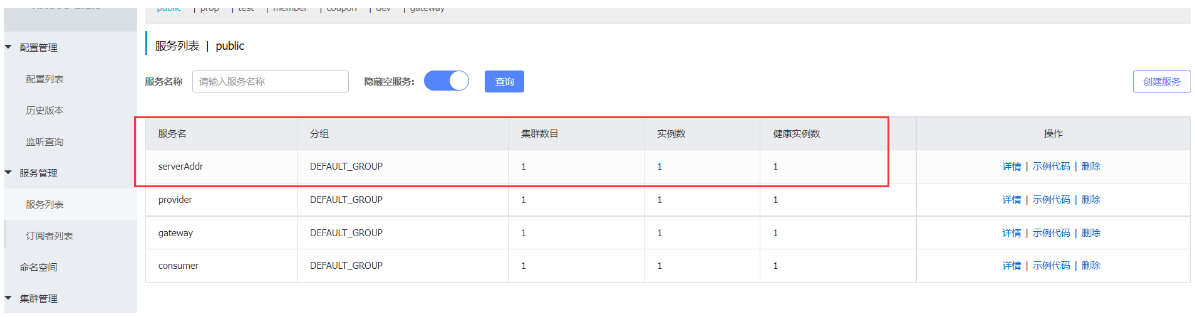
进入seata 的bin路径java

```
cd bin
seata-server.bat -p 8090 -m file
```



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.18363.1556]
(c) 2019 Microsoft Corporation。保留所有权利。

D:\JavaStudyResource\Software\SpringCloud_Software\seata-server-0.9.0\seata\bin>seata-server.bat -p 8090 -m file
```

启动成功，Nacos 注册成功。



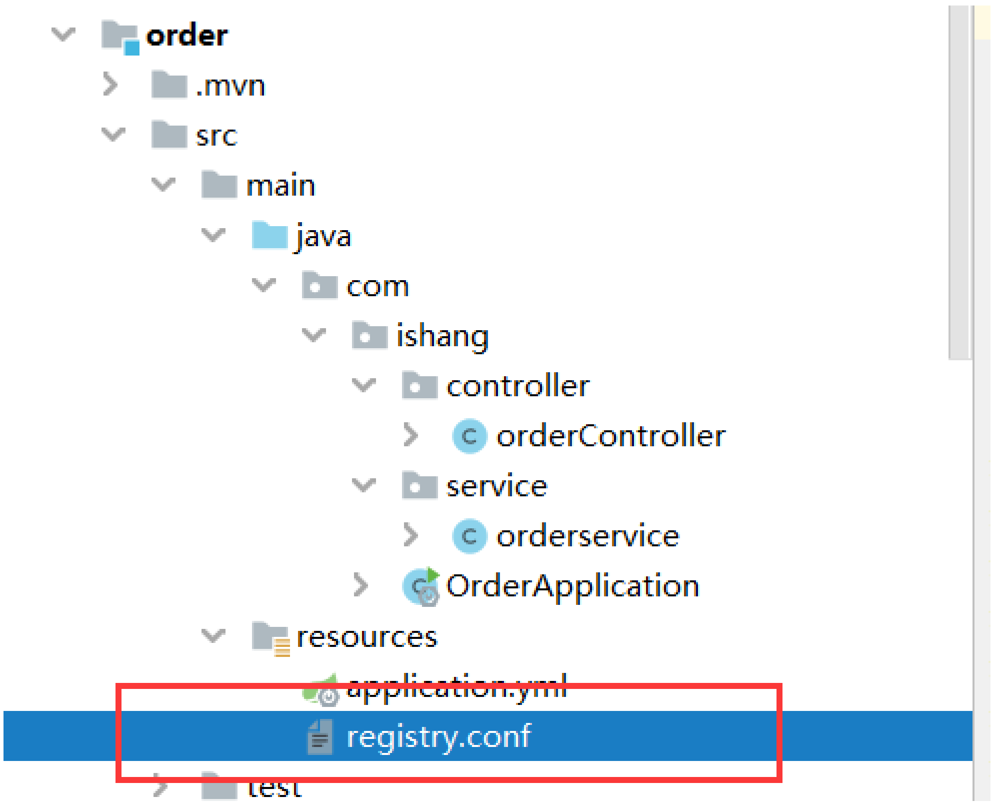
服务名	分组	集群数目	实例数	健康实例数	操作
serverAddr	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除
provider	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除
gateway	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除
consumer	DEFAULT_GROUP	1	1	1	详情 | 示例代码 | 删除

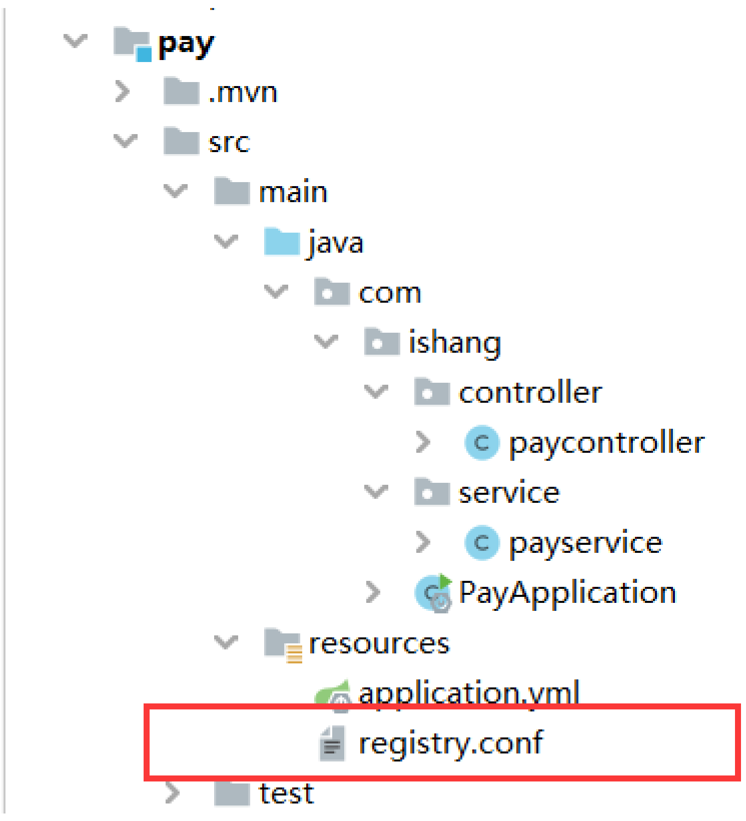
8.整合到微服务

8.1两个工程的 pom.xml 添加 Seata 组件和 Nacos Config 组件。

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

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

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

8.2、将 registry.conf 复制到两个工程的 resources 下。





8.3、给两个工程添加 application.yml 读取 Nacos 配置。

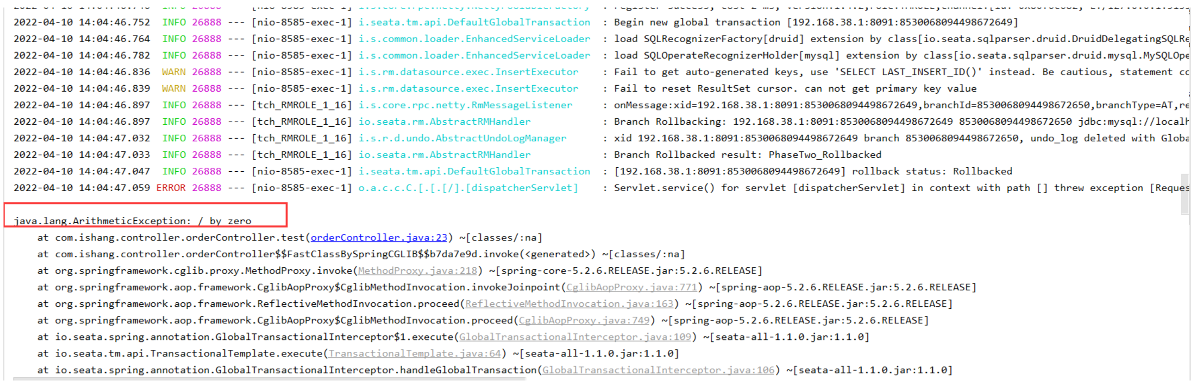
```yaml
server:
  port: 8686
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/pay?useUnicode=true&characterEncoding=UTF-
8&serverTimezone=Asia/Shanghai
    username: root
    password: 123456
  application:
    name: pay
  cloud:
    alibaba:
      seata:
        tx-service-group: my_test_tx_group
```

```yaml
server:
  port: 8585
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/order?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
    username: root
    password: 123456
  application:
    name: order
  cloud:
    alibaba:
      seata:
        tx-service-group: my_test_tx_group
```

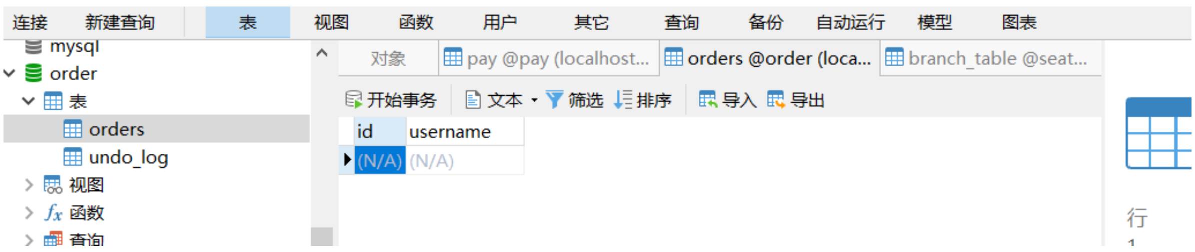
8.4、在 Order 调用 Pay 处添加注解 @GlobalTransactional

```java
package com.ishang.controller;


import com.ishang.service.orderservice;
import io.seata.spring.annotation.GlobalTransactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class orderController {
        @Autowired
        private orderservice orderservice;

        @Autowired
        private RestTemplate restTemplate;

        @RequestMapping("/test")
        @GlobalTransactional
        public String test(){
            this.orderservice.save();
            int i=10/0;

 this.restTemplate.getForObject("http://localhost:8686/test",String.class);
            return "success";
        }
}
```

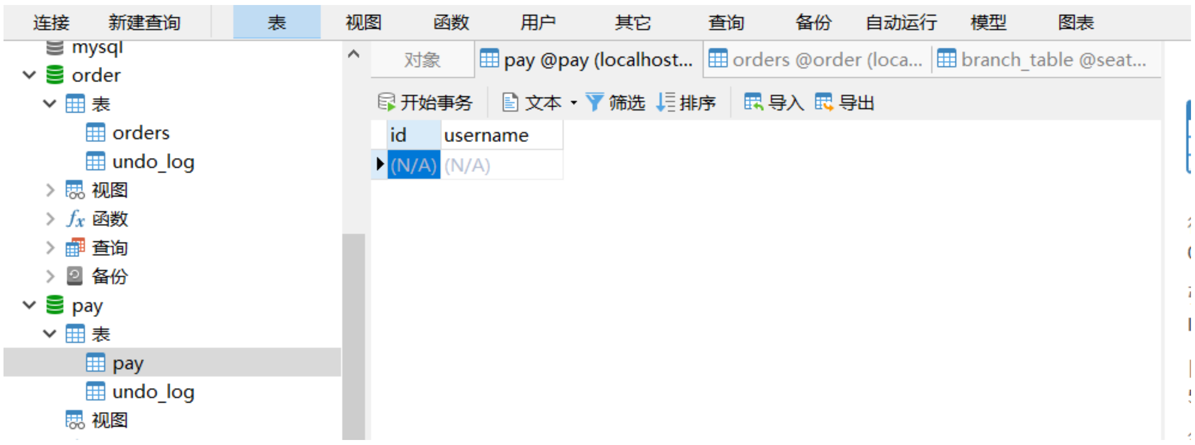
8.5运行



```
2022-04-10 14:04:46.752  INFO 26888 --- [nio-8585-exec-1] i.seata.tm.api.DefaultGlobalTransaction  : Begin new global transaction [192.168.38.1:8091:8530068094498672649]
2022-04-10 14:04:46.764  INFO 26888 --- [nio-8585-exec-1] i.s.common.loader.EnhancedServiceLoader  : load SQLRecognizerFactory[druid] extension by class[io.seata.sqlparser.druid.DruidDelegatingSQLRe
2022-04-10 14:04:46.782  INFO 26888 --- [nio-8585-exec-1] i.s.common.loader.EnhancedServiceLoader  : load SQLOperateRecognizerHolder[mysql] extension by class[io.seata.sqlparser.druid.mysql.MySQLOpe
2022-04-10 14:04:46.836  WARN 26888 --- [nio-8585-exec-1] i.s.rm.datasource.exec.InsertExecutor    : Fail to get auto-generated keys, use 'SELECT LAST_INSERT_ID()' instead. Be cautious, statement cc
2022-04-10 14:04:46.839  WARN 26888 --- [nio-8585-exec-1] i.s.rm.datasource.exec.InsertExecutor    : Fail to reset ResultSet cursor. can not get primary key value
2022-04-10 14:04:46.897  INFO 26888 --- [tch_RMROLE_1_16] i.s.core.rpc.netty.RmMessageListener     : onMessage:xid=192.168.38.1:8091:8530068094498672649,branchId=8530068094498672650,branchType=AT,re
2022-04-10 14:04:46.897  INFO 26888 --- [tch_RMROLE_1_16] io.seata.rm.AbstractRMHandler            : Branch Rollbacking: 192.168.38.1:8091:8530068094498672649 8530068094498672650 jdbc:mysql://localh
2022-04-10 14:04:47.032  INFO 26888 --- [tch_RMROLE_1_16] i.s.r.d.undo.AbstractUndoLogManager      : xid 192.168.38.1:8091:8530068094498672649 branch 8530068094498672650, undo_log deleted with Globa
2022-04-10 14:04:47.033  INFO 26888 --- [tch_RMROLE_1_16] io.seata.rm.AbstractRMHandler            : Branch Rollbacked result: PhaseTwo_Rollbacked
2022-04-10 14:04:47.047  INFO 26888 --- [nio-8585-exec-1] i.seata.tm.api.DefaultGlobalTransaction  : [192.168.38.1:8091:8530068094498672649] rollback status: Rollbacked
2022-04-10 14:04:47.059 ERROR 26888 --- [nio-8585-exec-1] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Reques

java.lang.ArithmeticException: / by zero
	at com.ishang.controller.orderController.test(orderController.java:23) ~[classes/:na]
	at com.ishang.controller.orderController$$FastClassBySpringCGLIB$$b7da7e9d.invoke(<generated>) ~[classes/:na]
	at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218) ~[spring-core-5.2.6.RELEASE.jar:5.2.6.RELEASE]
	at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:771) ~[spring-aop-5.2.6.RELEASE.jar:5.2.6.RELEASE]
	at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:163) ~[spring-aop-5.2.6.RELEASE.jar:5.2.6.RELEASE]
	at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:749) ~[spring-aop-5.2.6.RELEASE.jar:5.2.6.RELEASE]
	at io.seata.spring.annotation.GlobalTransactionalInterceptor$1.execute(GlobalTransactionalInterceptor.java:109) ~[seata-all-1.1.0.jar:1.1.0]
	at io.seata.tm.api.TransactionalTemplate.execute(TransactionalTemplate.java:64) ~[seata-all-1.1.0.jar:1.1.0]
	at io.seata.spring.annotation.GlobalTransactionalInterceptor.handleGlobalTransaction(GlobalTransactionalInterceptor.java:106) ~[seata-all-1.1.0.jar:1.1.0]
```



id	username
(N/A)	(N/A)



id	username
(N/A)	(N/A)

数据表进行回滚