SpringCloud Alibaba 课程

1、今日知识点

- SpringCloud Alibaba Nacos
- SpringCloud Alibaba Sentinel

2、Nacos 入门

2.1、Alibaba 各组件版本说明

具体参考地址:https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E

2.2、Nacos 介绍

Nacos 是阿里巴巴的产品，现在是 SpringCloud Alibaba 中的一个组件。其官网：https://nacos.io/

相对于 Spring Cloud Eureka 来说，在国内受欢迎程度较高,功能也更为强大。Nacos 就是 **注册中心+配置中心** 的组合,

等价于: Nacos = Eureka+Config+Bus。Nacos 能与 Spring, Spring Boot,Spring Cloud 集成,并能代替 Spring Cloud Eureka, Spring Cloud Config, Spring Cloud Bus。

通过NacosServer和spring-cloud-starter-alibaba-nacos-config 实现配置的动态变更
通过NacosServer和spring-cloud-starter-alibaba-nacos-discovery实现服务注册与发现

nacos 在阿里内部有超过 10 万的实例运行，已经过了类似双十一等各种大型流量的考验。

面试题：微服务间远程交互的过程?

1.先去注册中心查询服务的服务器地址
2.调用方给对方发送http请求

2.3、Nacos 主要提供以下四大功能

服务发现和服务健康监测
动态配置服务
动态DNS服务
服务及其元数据管理: 管理平台的角度，nacos也有一个ui页面，可以看到注册的服务及其实例信息（元数据信息）等）
动态的服务权重调整、动态服务优雅下线

2.4、Nacos 下载及安装

下载地址：https://github.com/alibaba/nacos/releases。

Nacos 底层由于是 java 语言编写的，所以安装 Nacos 之前要先安装 JDK，在开发阶段，安装 Nacos 单机版即可。

2.4.1、Windows 安装

第一步：解压 nacos-server-1.4.3.zip 并查看其目录结构

目录说明：

- bin：启动脚本
- conf：配置文件
- target:编译后的文件

第二步：端口和项目名配置

Nacos 的默认端口是 8848，如要修改 nacos 默认的端口号，可以进入 nacos 的 conf 目录，修改 application.properties 文件：

```
server.port=8848
```

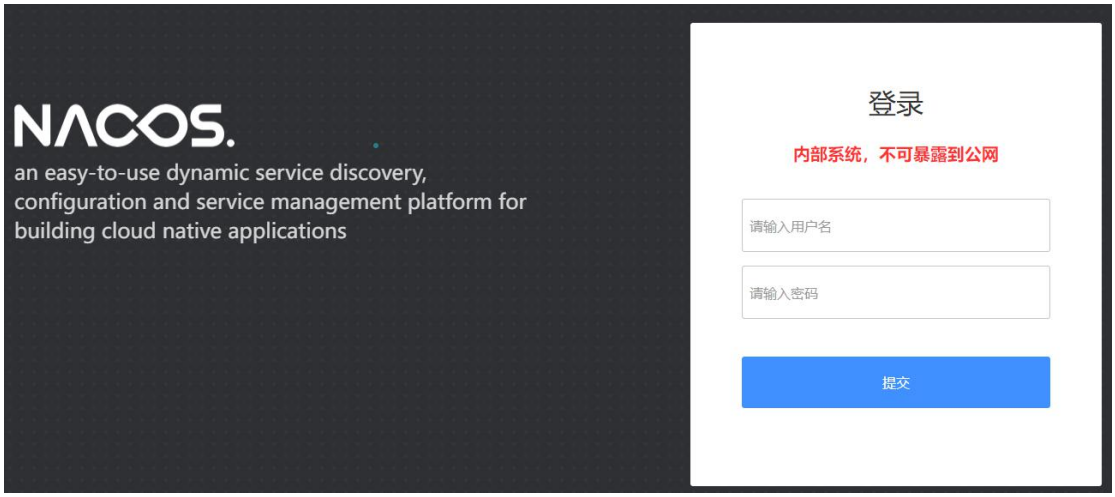
第三步：启动

Nacos 默认以集群方式启动，如果想启动单机版，windows 命令为：startup.cmd -m standalone

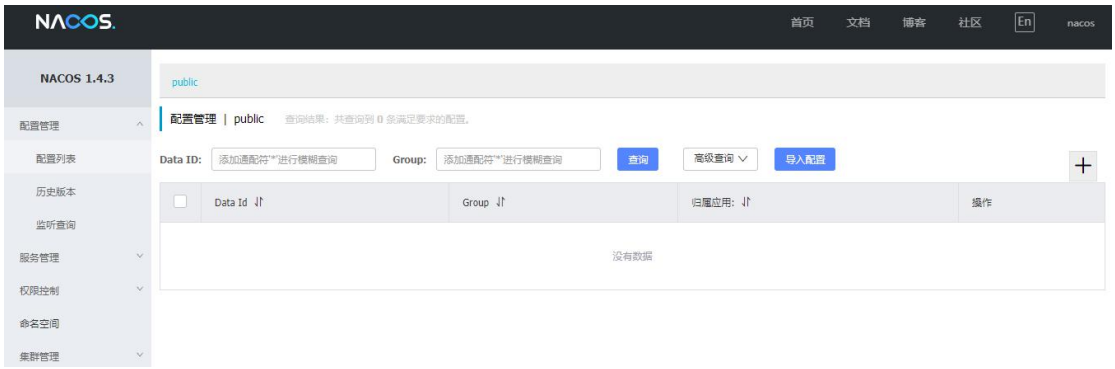
启动后，效果为：

第四步：访问

在浏览器输入地址：http://127.0.0.1:8848/nacos 即可.



默认的账号和密码都是 nacos，进入后：



2.4.2、Linux 安装

第一步：安装 8 及以上版本的 JDK，并配置环境变量

第二步：解压到指定的目录：tar -xvf nacos-server-1.4.3.tar.gz -C /opt/install

第三步：修改 conf 目录中的 application.properties 文件中的端口号

第四步：进入 nacos 的 bin 目录中，执行命令：**sh startup.sh -m standalone**

2.5、Nacos 依赖说明

版本号控制：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.2.7.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

作为注册中心依赖：

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

作为配置中心依赖：

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

3、Nacos 作为注册中心

3.1、Nacos 作为注册中心

3.1.1、与其他注册中心比较

Nacos 与其他注册中心特性对比

	Nacos	Eureka	Consul	CoreDNS	ZooKeeper
一致性协议	CP+AP	AP	CP	/	CP
健康检查	TCP/HTTP/MySQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	/	Client Beat
负载均衡	权重/DSL/metadata/CMDB	Ribbon	Fabio	RR	/
雪崩保护	支持	支持	不支持	不支持	不支持
自动注销实例	支持	支持	不支持	不支持	支持
访问协议	HTTP/DNS/UDP	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心	支持	不支持	支持	不支持	不支持
SpringCloud 集成	支持	支持	支持	不支持	不支持
Dubbo 集成	支持	不支持	不支持	不支持	支持
K8s 集成	支持	不支持	支持	支持	不支持

3.1.2、具体使用参考

https://github.com/alibaba/spring-cloud-alibaba/wiki/Nacos-discovery

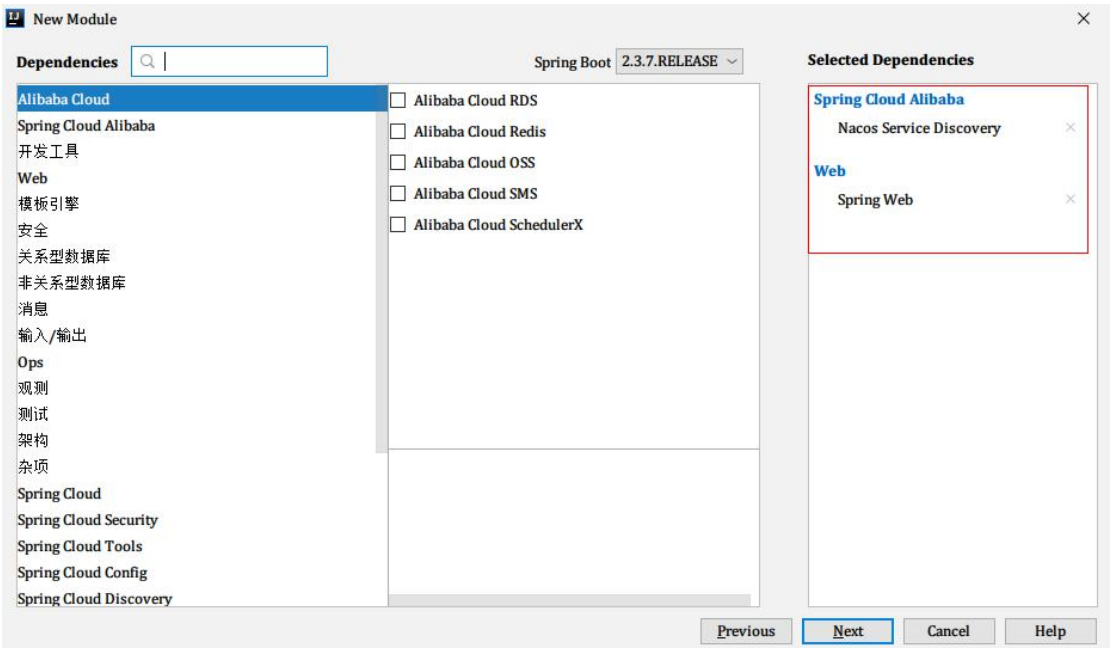
3.2、基于 nacos 创建服务提供者

案例需求：电影查询系统

用户服务:可以查询用户信息、查询用户信息+电影信息
电影服务:可以查询电影信息

3.2.1、创建新模块并引入 web 和 Nacos Service Discovery 依赖

创建模块: atguigu-nacos-provider-movie6600



3.2.2、配置 yml 文件

```yaml
server:
  port: 6600
spring:
  application:
    name: atguigu-nacos-provider-movie
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #nacos服务端地址
```

3.2.3、主程序添加注解：@EnableDiscoveryClient

3.2.4、编写业务逻辑层

第一步：编写 bean 对象

```
public class Movie {
  private Integer id;
  private String movieName;
  //...此处省略了getter、setter、toString方法
}
```

第二步：编写 dao 层

```
@Repository
public class MovieDao {
    public Movie getNewMovie(){
        Movie movie = new Movie();
        movie.setId(1);
        movie.setMovieName("战狼");
        return movie;
    }
}
```

第三步：编写 service 层

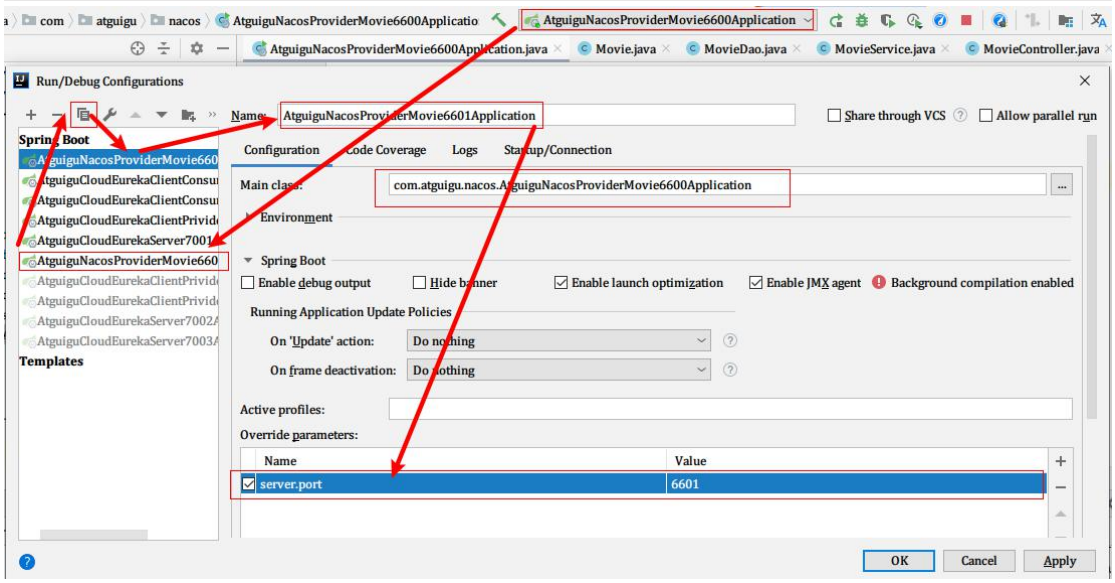
```java
@Service
public class MovieService {
    @Autowired
   private MovieDao movieDao;

    public Movie getNewMovie(){
    return movieDao.getNewMovie();
    }
}
```

第四步：编写控制层

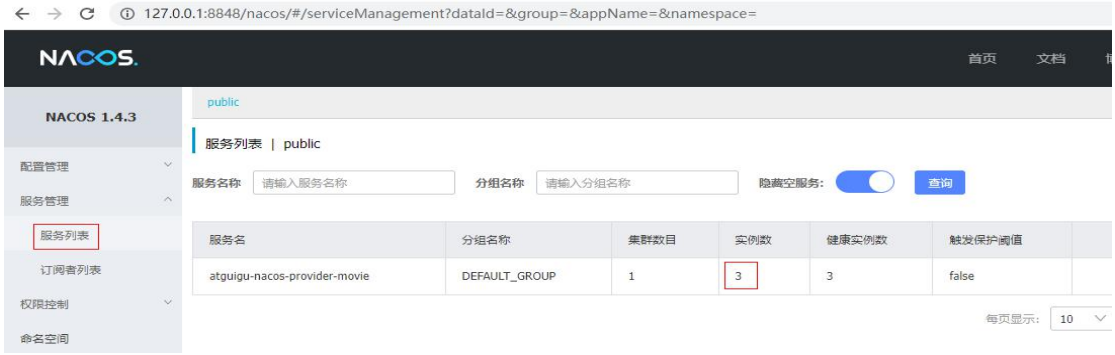
```java
@RestController
@RequestMapping("/movie")
public class MovieController {
    @Autowired
    private MovieService movieService;

    // 获取最新电影
    @GetMapping("/latest")
    public Movie getNewMovie(){
        return movieService.getNewMovie();
    }
}
```

第五步：访问电影服务: http://localhost:6600/movie/latest

3.2.5、多实例启动，查看 Nacos 控制台



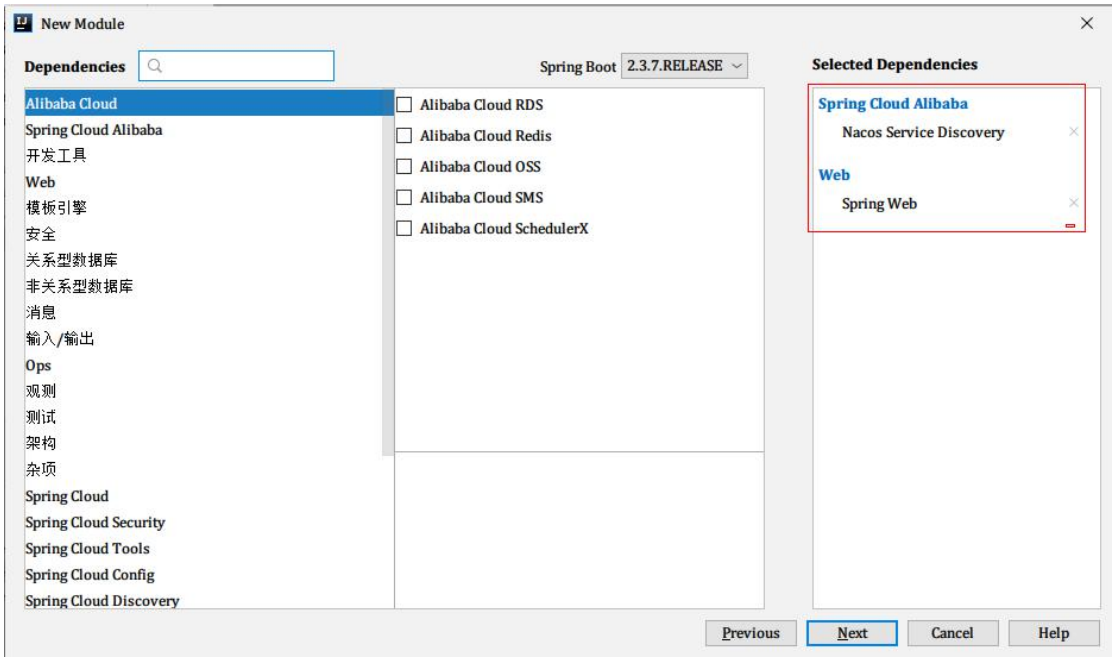
浏览器访问：http://localhost:8488/nacos



3.3、基于 Nacos 创建服务消费方(用户服务)

3.3.1、创建模块并引入 web、nacos discovery 依赖

创建模块: atguigu-nacos-consumer-user6700



3.3.2、配置 yml 文件

```yaml
server:
  port: 6700
spring:
  application:
    name: atguigu-nacos-consumer-user
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
```

3.3.3、在主启动类添加@EnableDiscoveryClient 注解

3.3.4、编写业务逻辑层

第一步:编写实体类 user

```
public class User {
    private Integer id;
    private String userName;//此处省略属性的getter、setter、toString方法
}
```

第二步:编写 UserDao

```
@Repository
public class UserDao {

    public User getUser(Integer id){
        User user = new User();
        user.setId(id);
        user.setUserName("张三");
        return user;
    }
}
```

第三步:编写 UserService

```
@Service
public class UserService {

  @Autowired
  private   UserDao userDao;

    public User getUserById(Integer id){
        User user = userDao.getUser(id);
        return user;
    }

    public Map<String, Object> getUserAndMovie (Integer id){
        Map<String, Object> result = new HashMap<>();
        //1、查询用户信息
        User user = getUserById(id);
        //2、查到最新电影票
        result.put("user", user);
        result.put("movie", null);//暂时为null
        return result;
    }
}
```

第四步:编写 UserController

```
@RestController
public class UserController {
  @Autowired
  private   UserService userService;

  @GetMapping("/user")
  public User getUser(@RequestParam("id")Integer id){
    User user = userService.getUserById(id);
    return user;
  }
    @GetMapping("/getMovie")
  public Map<String, Object> buyMovie(@RequestParam("id")Integer userid){
    Map<String, Object> map = userService.getUserAndMovie(userid);
    return map;
  }
}
```

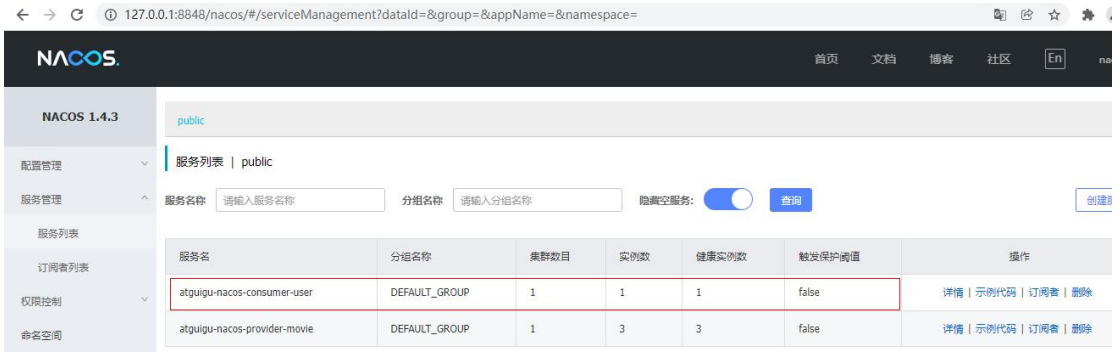
第五步:测试注册中心、用户服务

访问注册中心，查看用户服务已经注册到注册中心： http://localhost:8848/nacos

访问用户服务: http://localhost:6700/user?id=1

访问用户服务中的电影服务。 http://localhost:6700/getMovie?id=1

3.3.5、查看 Nacos 控制台



3.4、整合 OpenFeign 进行远程调用

修改 **atguigu-nacos-consumer-user6700** 模块

3.4.1、添加 Feign 的依赖

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

3.4.2、在主程序类上添加@EnableFeignClients 注解

3.4.3、编写要调用的远程的 FeignClient 接口

```
@FeignClient(value = "atguigu-nacos-provider-movie")//绑定Feign客户端要访问的服务。
public interface MovieFeignClient {
    // 获取最新电影
    @GetMapping("/movie/latest")
    public Movie getNewMovie();
}
```

3.4.4、在用户服务中使用 FeignClient 完成远程调用业务

```
@Service
public class UserService {

    @Autowired
    private   UserDao userDao;
    @Autowired
    private MovieFeignClient movieFeignClient;

    public Map<String, Object> getUserAndMovie (Integer id){
        Map<String, Object> result = new HashMap<>();
        User user = getUserById(id);
        result.put("user", user);
        result.put("movie", movieFeignClient.getNewMovie());//暂时为null
        return result;
    }
    public User getUserById(Integer id){
        User user = userDao.getUser(id);
        return user;
    }
}
```

3.4.5、访问测试

重启用户服务并访问：http://localhost:6700/getMovie?id=1 完成远程调用。

注意:**nacos 是默认是支持负载均衡的**。在 nacos 的 jar 包依赖了 ribbon

3.5、加入 Hystrix 熔断保护

1.停掉电影服务

2.用户服务在远程调用电影服务的时候就会出错,如下所示:

解决方案：给**用户服务**加入 Hystrix 断路保护机制

3.5.1、引入 Hystrix 依赖

```
<!-- 引入hystrix进行服务熔断 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

3.5.2、在主启动类上加注解:@EnableCircuitBreaker

3.5.3、开启 Feign 对 Hystrix 支持

```
feign:
  hystrix:
    enabled: true
```

3.5.4、Feign 已经集成了 Hystrix，使用起来非常简单

@FeignClient(name=" atguigu-nacos-provider-movie",fallback="异常处理类")

```java
@FeignClient(value = "atguigu-nacos-provider-movie",fallback = MovieFeignServiceExceptionHandler.class)//绑定Feign客户端要访问的服务。
public interface MovieFeignClient {
    // 获取最新电影
    @GetMapping("/movie/latest")
    public Movie getNewMovie();
}
```

3.5.5、fallback="异常处理类"指定实现该接口的异常处理类，并放在容器中

```java
@Service
public class MovieFeignServiceExceptionHandler implements MovieFeignClient {
    @Override
    public Movie getNewMovie() {
        Movie movie = new Movie();
        movie.setId(-100);
        movie.setMovieName("访问失败，请刷新重试");
        return movie;
    }
}
```

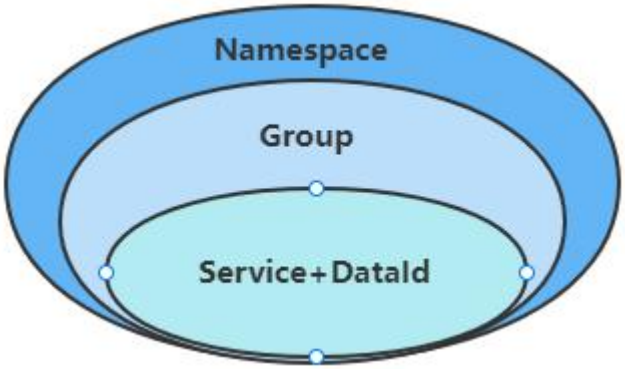
3.5.6、重启用户服务，测试熔断是否生效

访问：http://localhost:6700/getMovie?id=1

3.6、Nacos 服务与配置的数据模型

3.6.1、数据模型介绍

Namespace 命名空间、Group 分组、集群这些都是为了进行归类管理，把服务和配置文件进行归类，归类之后就可以实现一定的效果， 比如，对于服务来说，不同命名空间中的服务不能够互相访问调用。



Namespace：命名空间，对不同的环境进行隔离，比如隔离开发环境、测试环境和生产环境

Group：分组，将若干个服务或者若干个配置集归为一组，通常习惯一个系统归为一个组

Service：某一个服务，比如简历微服务

DataId：配置集或者可以认为是一个配置文件

```
Namespace + Group + Service 如同 Maven 中的GAV坐标，GAV坐标是为了锁定Jar，二这里是为了锁定服务
Namespace + Group + DataId 如同 Maven 中的GAV坐标，GAV坐标是为了锁定Jar，二这里是为了锁定配置文件
```

3.6.2、最佳实践

Nacos 抽象出了 Namespace、Group、Service、DataId 等概念，具体代表什么取决于怎么用（非常灵活），推荐用法如下

概念	概述
Namespace	代表不同的环境，如开发dev、测试test、生产环境prod
Group	代表某项目，比如谷粒商城项目
Service	代表某个项目中具体的xxx服务
DataId	某个项目中具体的xxx配置文件

3.6.3、服务的分级存储模型

一个**服务**可以有多个**实例**，例如我们的**用户服务**，可以有：

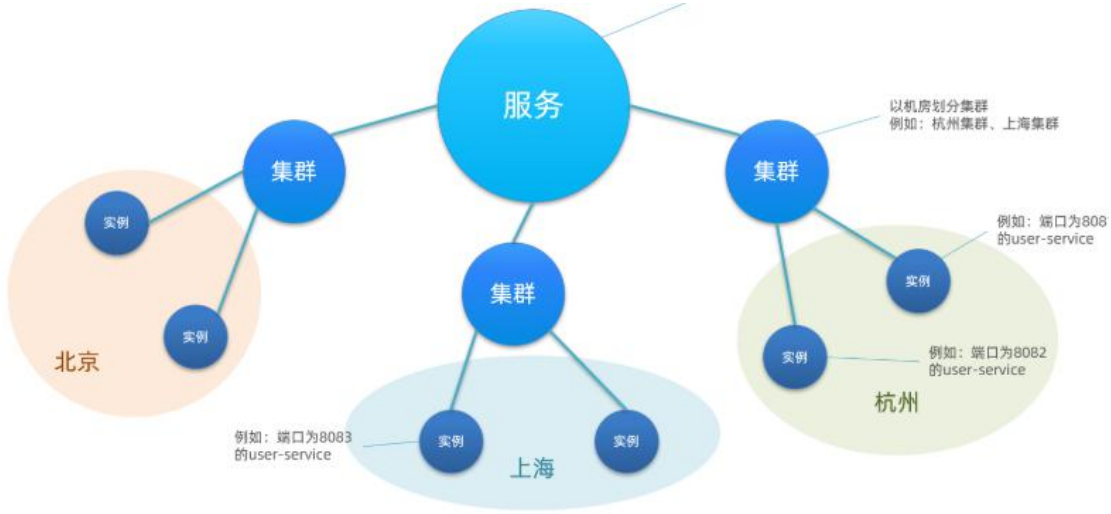
```
- 127.0.0.1:8081
- 127.0.0.1:8082
- 127.0.0.1:8083
```

假如这些实例分布于全国各地的不同机房，例如：

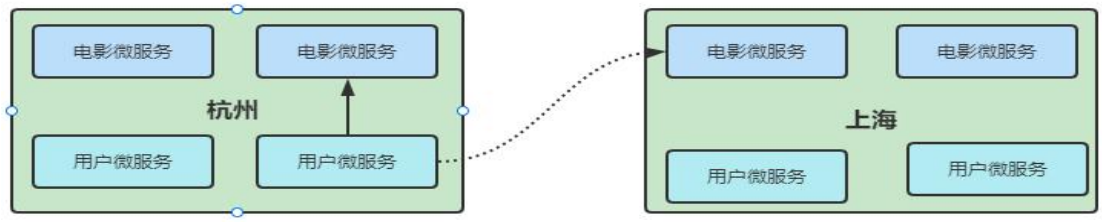
```
- 127.0.0.1:8081，在上海机房
- 127.0.0.1:8082，在上海机房
- 127.0.0.1:8083，在杭州机房
```

Nacos 就将同一机房内的实例 划分为一个**集群**。

也就是说一个服务可以包含多个集群，如杭州、上海，每个集群下可以有多个实例，形成分级模型，如图：



微服务互相访问时，应该尽可能访问**同集群实例**，因为本地访问速度更快。当本集群内不可用时，才访问其它集群。如：



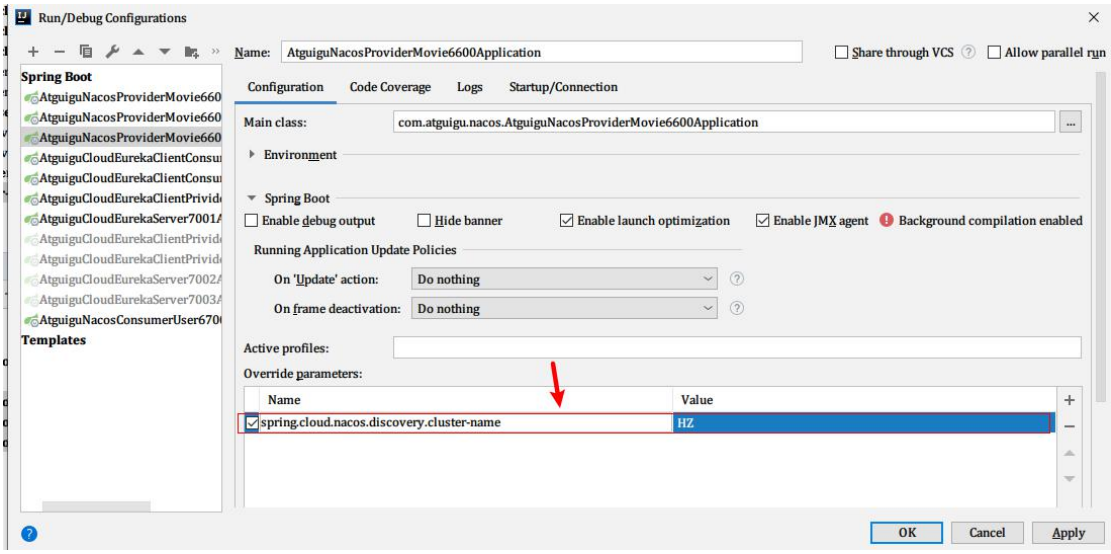
杭州机房内的**用户微服务**应该优先访问同机房的**电影微服务**。

3.6.4、服务集群配置

第一步:修改**电影微服务**的 application.yml 文件，添加集群配置:

```yaml
spring:
  cloud:
    nacos:
      discovery:
        cluster-name: HZ
```

或者如下位置:



将端口号为 6600、6601 的电影微服务放在**杭州**集群、6602 的电影微服务放在**上海集群**

第二步：在电影微服务 controller 层添加:

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

// 获取最新电影
@GetMapping("/latest")
public Movie getNewMovie(){
    System.out.println(port);
    return movieService.getNewMovie();
}
```

第三步：重启三个电影微服务 实例后，我们可以在 nacos 控制台看到下面结果:

点击**详情**：

第四步：修改用户微服务

第五步：访问：http://localhost:6700/getMovie?id=1

会发现，**默认的`ZoneAvoidanceRule`并不能实现根据同集群优先来实现负载均衡，因此 Nacos 中提供了一个`NacosRule`的实现，可以优先从同集群中挑选实例。**

3.6.5、修改默认的负载均衡规则

修改**用户微服务**的 application.yml 文件，修改负载均衡规则：

```
atguigu-nacos-provider-movie: #被调用方在nacos注册中心的服务名
  ribbon:
    NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule # 负载均衡规则
```

3.6.6、权重配置

实际部署中会出现这样的场景：服务器设备性能有差异，部分实例所在机器性能较好，另一些较差，我们希望性能好的机器承担更多的用户请求。但默认情况下 NacosRule 是同集群内随机挑选，不会考虑机器的性能问题。因此，Nacos 提供了权重配置来控制访问频率，权重越大则访问频率越高。在 nacos 控制台，找到 user-service 的实例列表，点击编辑，即可修改权重：

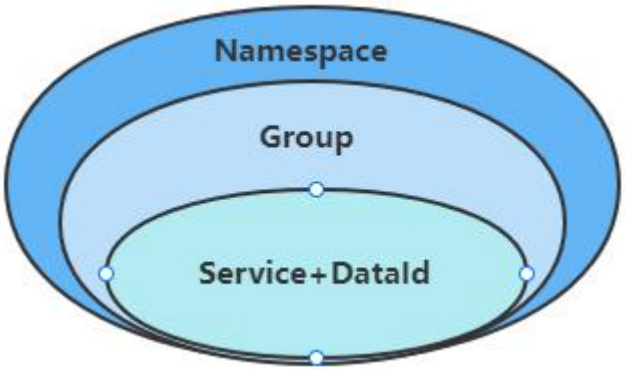
在弹出的编辑窗口，修改权重：

注意：**如果权重修改为 0，则该实例永远不会被访问**

3.6.7、环境隔离

Nacos 提供了 namespace 来实现环境隔离功能。

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



第一步：创建 namespace

然后，填写表单：

就能在页面看到一个新的 namespace：

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		0	详情 删除 编辑
dev	07eaf477-b423-4862-8e84-f3cb72b631fa	0	详情 删除 编辑

第二步：修改微服务的 namespace

例如，修改 **用户微服务** 的 application.yml 文件：

```yaml
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
        cluster-name: HZ
        namespace: 07eaf477-b423-4862-8e84-f3cb72b631fa
```

第三步：重启 **用户微服务** 后，访问控制台，会发现**用户微服务**跑到 **dev 开发环境**下面了。

此时访问**用户微服务**，因为 namespace 不同，**会导致找不到电影微服务**：

访问地址：http://localhost:6700/getMovie?id=1

控制台会报错:

```
WARN 17584 --- [rovider-movie-1] c.alibaba.cloud.nacos.ribbon.NacosRule    : no instance in service atguigu-nacos-provider-movie
WARN 17584 --- [rovider-movie-1] c.alibaba.cloud.nacos.ribbon.NacosRule    : no instance in service atguigu-nacos-provider-movie
WARN 17584 --- [rovider-movie-2] c.alibaba.cloud.nacos.ribbon.NacosRule    : no instance in service atguigu-nacos-provider-movie
WARN 17584 --- [rovider-movie-2] c.alibaba.cloud.nacos.ribbon.NacosRule    : no instance in service atguigu-nacos-provider-movie
WARN 17584 --- [rovider-movie-3] c.alibaba.cloud.nacos.ribbon.NacosRule    : no instance in service atguigu-nacos-provider-movie
```

3.7、Nacos 作为注册中心与 Eureka 区别

Nacos 的服务实例分为两种 l 类型：

- 临时实例：如果实例宕机超过一定时间，会从服务列表剔除，默认的类型。
- 非临时实例：如果实例宕机，不会从服务列表剔除，也可以叫永久实例。

配置一个服务实例为永久实例：

```
spring:
  cloud:
    nacos:
      discovery:
        ephemeral: false # 设置为非临时实例
```

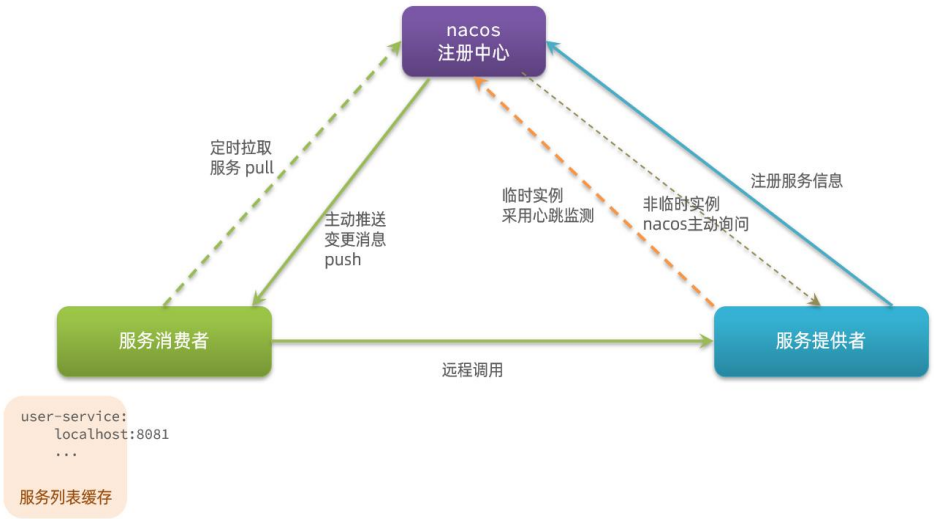
Nacos 和 Eureka 整体结构类似，服务注册、服务拉取、心跳等待，但是也存在一些差异：

Nacos 与 eureka 的共同点:

```
1.都支持服务注册和服务拉取
2.都支持服务提供者心跳方式做健康检测
```

Nacos 与 Eureka 的区别:

```
1.Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式
2.临时实例心跳不正常会被剔除，非临时实例则不会被剔除
3.Nacos支持服务列表变更的消息推送模式，服务列表更新更及时
4.Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式
```

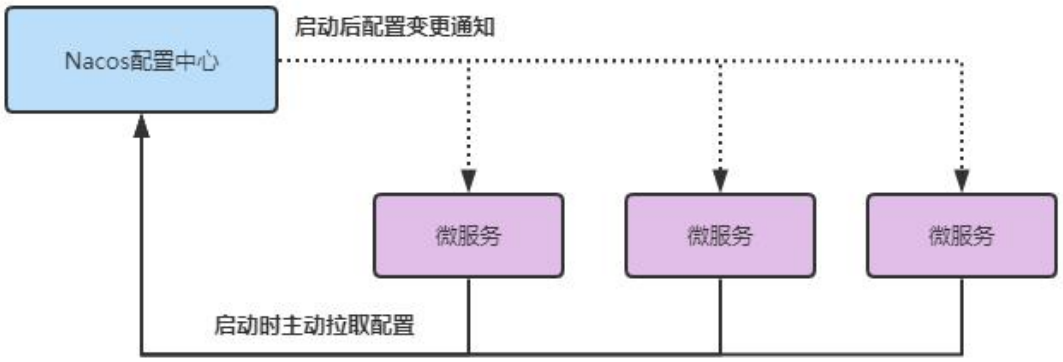


4、Nacos 作为配置中心

Nacos 除了可以做注册中心，同样可以做配置管理来使用。**nacos 作为配置中心可以做到系统配置的集中管理（编辑、存储、分发）、动态更新不重启、回滚配置（变更管理、历史版本管理、变更审计）等所有与配置相关的活动。**有 Nacos 之后，分布式配置就简单很多 Github 不需要了（配置信息直接配置在 Nacos server 中），Bus 也不需要了(依然可以完成配置文件的**热更新，及时通知微服务**)。**如果微服务架构中没有使用统一配置中心时，所存在的问题**：

```
配置文件分散在各个项目里，微服务实例的数量达到上百个的时候，实在是不方便维护
配置内容安全与权限
更新配置后，项目需要重启
```

热更新:



案例：改造服务消费方中的动态配置项，由配置中心统一管理。

4.1、统一配置管理

4.1.1、添加配置文件

4.1.2、dataId 格式说明

1.prefix 默认为所属工程配置spring.application.name 的值（nacos-provider-movie*）,也可以通过配置项spring.cloud.nacos.config.prefix来配置。
2.spring.profile.active 即为当前环境对应的 profile，详情可以参考 Spring Boot文档。 注意：当 spring.profile.active 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 ${prefix}.${file-extension}
3.file-exetension 为配置内容的数据格式，可以通过配置项spring.cloud.nacos.config.file-extension 来配置。目前只支持 properties 和 yaml 类型。

总结：配置所属工程的 spring.application.name 的值 + "." + properties/yml

4.2、从配置中心读取配置

4.2.1、在 atguigu-nacos-consumer-user6700 模块引入依赖

```xml
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
   <version>0.2.1.RELEASE</version>
</dependency>
```

4.2.2、在 bootstrap.properties 中配置 nacos

```
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
spring.cloud.nacos.config.file-extension=yml    #配置文件默认的后缀是properties文件，如果是yml，必须指定
```

4.2.3、添加@RefreshScope 实现自动更新

```java
@RefreshScope
@RestController
public class UserController {

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

   @GetMapping("/config")
   public String getConfiguraiton(){
      return port;
   }
}
```

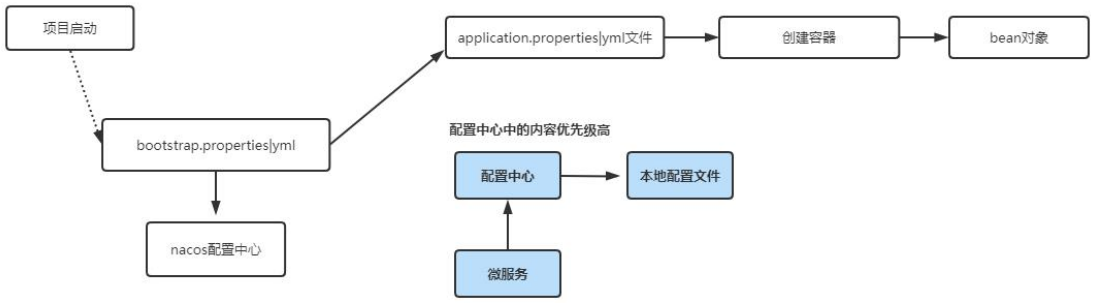
4.2.4、测试

重启该生产者模块,访问:http://localhost:6800/config,特别注意会发现当在本地配置了启动端口号，在 nacos 配置中心也配置启动端口号之后，以 nacos 配置中心的为准。

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

4.2.5、配置文件的优先级及配置信息合并

从上述项目启动的端口号中不难看出，bootstrap.properties 的优先级是优于 application.yml 文件的。并且微服务要拉取 nacos 中管理的配置，并且与本地的 application.yml 配置合并，才能完成项目启动。也就是说：bootstrap.properties[yaml]文件，会在 application.yml 之前被读取，流程如下：



所以：加载顺序及配置内容优先级：bootstrap.properties|yml 要优于 application.properties|yml.

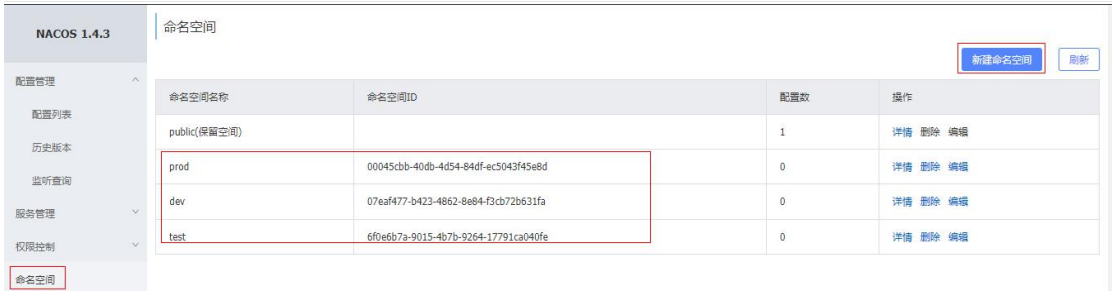
需要注意：bootstrap.**yml** 文件，内容格式如下调整：

```yaml
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml     #这里决定的是当前bootstrap.yml文件去nacos配置中心读取的配置文件格式
```

4.3、名称空间切换环境

在实际开发中，通常有多套不同的环境（默认只有 public），那么这个时候可以根据指定的环境来创建不同的 namespace，例如，开发、测试和生产三个不同的环境，那么使用一套 nacos 集群可以分别建以下三个不同的 namespace。以此来实现**多环境的隔离**。

4.3.1、创建名称空间

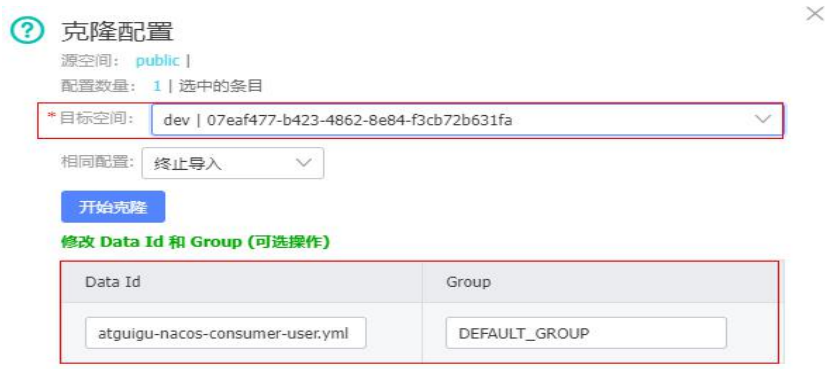


4.3.2、切换到配置列表

可以发现有四个名称空间：public（默认）以及我们自己添加的 3 个名称空间（prod、dev、test），可以点击查看每个名称空间下的配置文件，当然现在只有 public 下有一个配置。默认情况下，项目会到 public 下找 服务名.properties|yml 文件。接下来，在 dev 名称空间中也添加一个 atguigu-nacos-consumer-user.yml-dev.yml 配置。这时有两种方式：

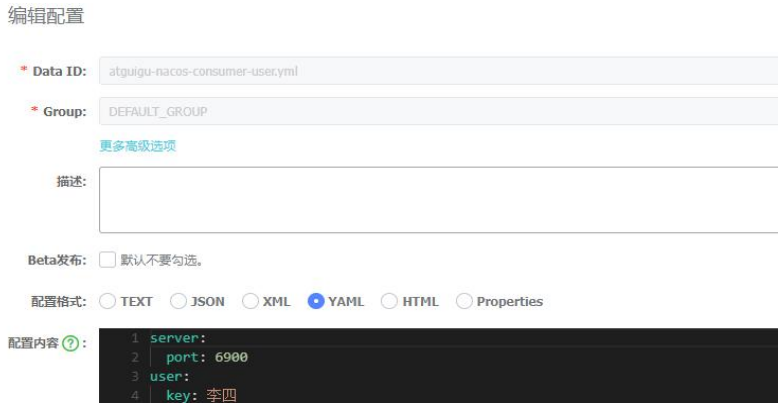
1 切换到dev名称空间，添加一个新的配置文件。缺点:每个环境都要重复配置类似的项目
2 直接通过clone方式添加配置，并修改即可。**推荐**

选择目标名称空间：



对这个文件编辑：

点击编辑：修改配置内容，以作区分



4.3.3、在 atguigu-nacos-consumer-user6700 中切换命名空间

修改 bootstrap.properties 添加如下配置：

```
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml
        namespace: 07eaf477-b423-4862-8e84-f3cb72b631fa
```

namespace 的值为：

4.3.4、重启服务提供方服务测试：

在浏览器访问：http://localhost:6900/config

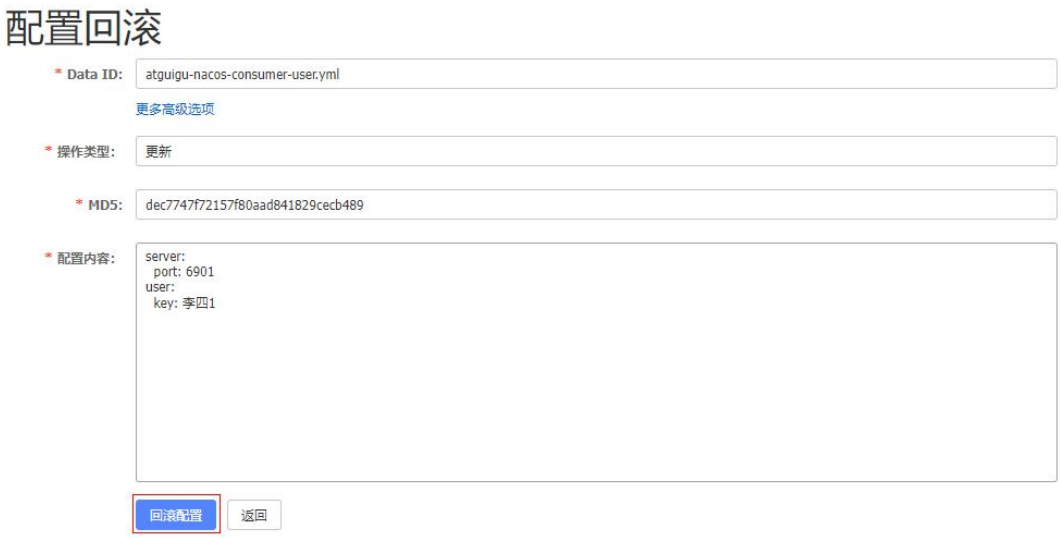
4.4、回滚配置

回滚配置只需要两步：

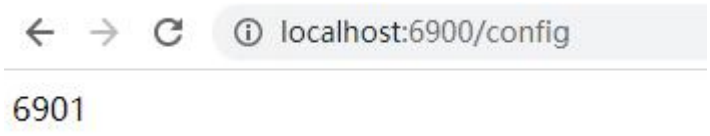
4.4.1、查看历史版本

4.4.2、回滚到某个历史版本

Data ID	Group	操作人	最后更新时间	操作
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:38:48	详情 | 回滚
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:38:36	详情 | 回滚
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:38:25	详情 | 回滚
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:33:23	详情 | 回滚
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:31:16	详情 | 回滚
atguigu-nacos-consumer-user.yml	DEFAULT_GROUP	nacos	2022/2/17 下午2:20:04	详情 | 回滚



访问地址：http://localhost:6900/config



4.4.3、加载多配置文件

偶尔情况下需要加载多个配置文件。假如现在 dev 名称空间下有三个配置文件：

atguigu-nacos-consumer-user6700 默认加载，怎么加载另外两个配置文件？在 bootstrap.yml 文件中添加如下配置：

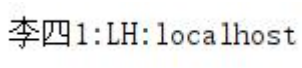
```yaml
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml
        namespace: 07eaf477-b423-4862-8e84-f3cb72b631fa
        ext-config: [{dataId: jdbc.yml,refresh: true},{dataId: redis.yml,refresh: true}]
```

修改 UserController 读取 redis.yml 和 jdbc.yml 配置文件中的参数：

```java
@RefreshScope
@RestController
public class UserController {

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

    @Value("${jdbc.username}")
    private String jdbcName;
    @Value("${redis.host}")
    private String redisHost;

    @GetMapping("/config")
    public String getConfiguraiton(){
        System.out.println(username+":"+jdbcName+":"+redisHost);
        return port;
    }
}
```

测试效果：http://localhost:6901/config

控制台输出如下：



问题：

```
1.修改一下配置中心中 redis.yml 中的配置，不重启服务。能否加载配置信息
2.删掉 refresh=true，再修改redis.yml 中的配置试试
```

4.5、配置分组

在实际开发中，除了不同的环境外。不同的微服务或者业务功能，可能有不同的 redis 及 mysql 数据库(集群中的相同微服务都连接自己的数据库或者 redis)。区分不同的环境我们使用名称空间（namespace），区分不同的微服务或功能，使用分组（group）。当然，你也可以反过来使用，名称空间和分组只是为了更好的区分配置，提供的两个维度而已。

新增一个 redis.yml，所属分组为 provider,需要添加如下配置：

```yaml
spring:
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml
        namespace: 07eaf477-b423-4862-8e84-f3cb72b631fa
        ext-config: [{dataId: jdbc.yml,refresh: true},{dataId: redis.yml,refresh: true,group: provider}]
```

访问：http://localhost:6901/config ,查看控制台输出

最佳实践：命名空间区分环境，分组区分业务。

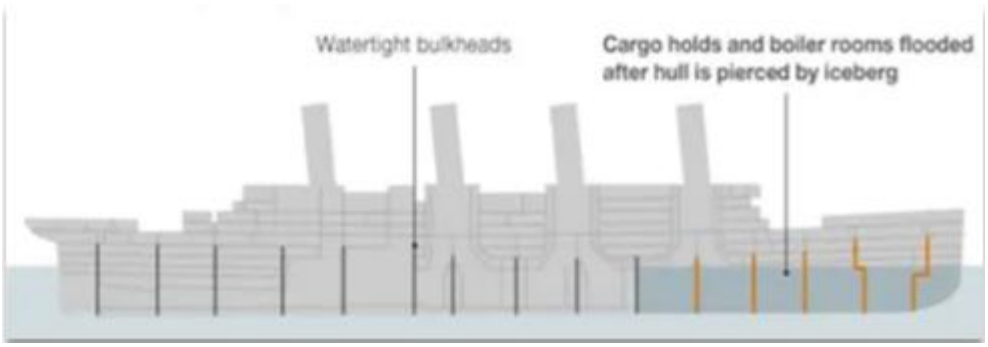
5、微服务保护和熔断降级技术：Sentinel

5.1、微服务调用存在问题

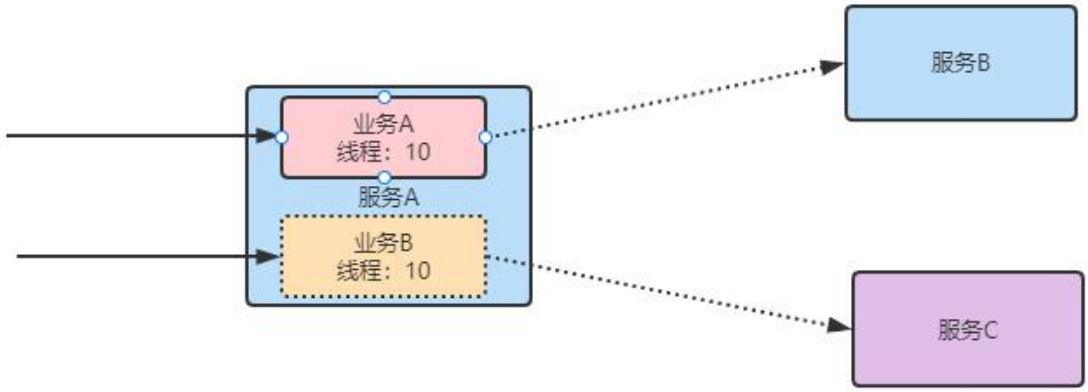
由于一个服务不可用，有可能会导致一连串的微服务跟着不可用**[服务器支持的线程和并发数有限，请求一直阻塞，会导致服务器资源耗尽，从而导致所有其它服务都不可用]**，形成级联失败，最终会导致服务雪崩问题。针对服务雪崩，有如下几种解决方案：

方案 1：超时处理：设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待

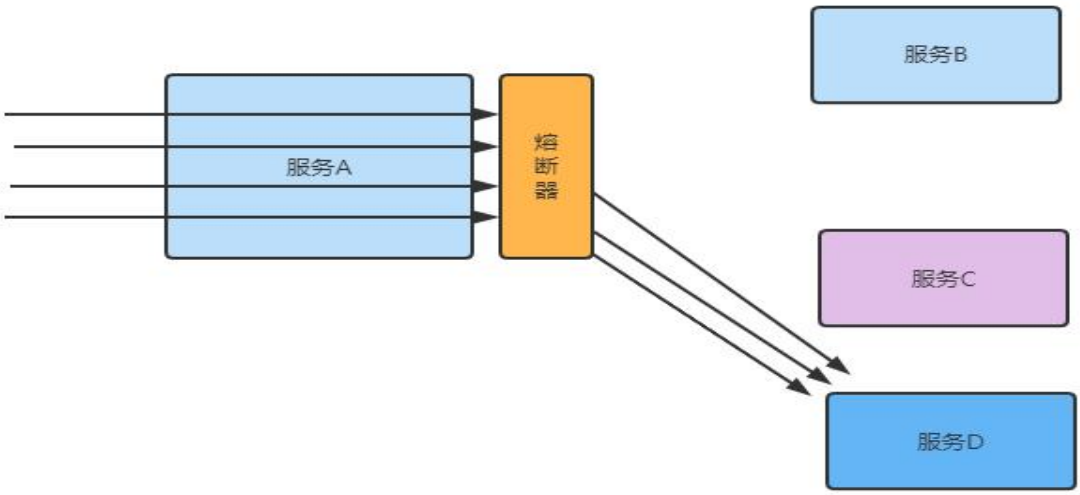
方案 2：仓壁模式，仓壁模式来源于船舱的设计：



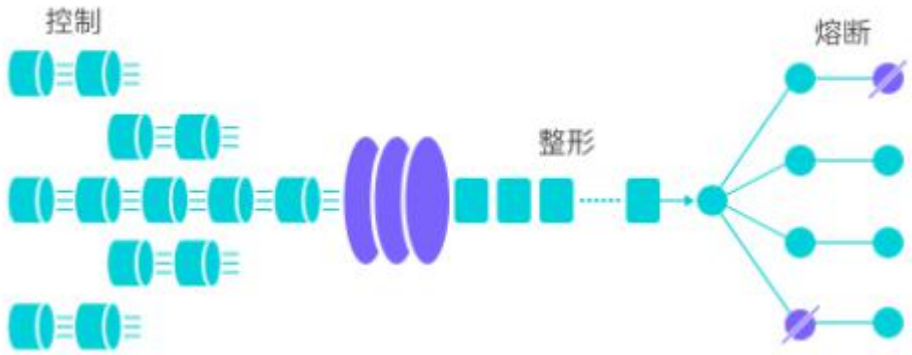
船舱都会被隔板分离为多个独立空间，当船体破损时，只会导致部分空间进入，将故障控制在一定范围内，避免整个船体都被淹没。于此类似，我们可以限定每个业务能使用的线程数，避免耗尽整个 tomcat 的资源，因此也叫线程隔离。



方案 3：断路器模式：由**断路器**统计业务执行的异常比例，如果超出阈值则会**熔断**该业务，拦截访问该业务的一切请求。断路器会统计访问某个服务的请求数量，异常比，当发现访问服务 D 的请求异常比例过高时，认为服务 D 有导致雪崩的风险，会拦截访问服务 D 的一切请求，形成熔断：



方案 4：**限流,流量控制**：限制业务访问的 QPS，避免服务因流量的突增而故障。



小结:

1.什么是雪崩问题？
微服务之间相互调用，因为调用链中的一个服务故障，引起整个链路都无法访问的情况。
2.可以认为：
限流是对服务的保护，避免因瞬间高并发流量而导致服务故障，进而避免雪崩。是一种**预防**措施。
超时处理、线程隔离、降级熔断是在部分服务故障时，将故障控制在一定范围，避免雪崩。是一种**补救**措施

5.2、常见服务保护技术对比

早期比较流行的是 Hystrix 框架，但目前国内实用最广泛的还是阿里巴巴的 Sentinel 框架，这里我们做下对比：

	Sentinel	Hystrix
隔离策略	信号量隔离	线程池隔离/信号量隔离
熔断降级策略	基于慢调用比例或异常比例	基于失败比率
实时指标实现	滑动窗口	滑动窗口（基于 RxJava）
规则配置	支持多种数据源	支持多种数据源
扩展性	多个扩展点	插件的形式
基于注解的支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持
流量整形	支持慢启动、匀速排队模式	不支持
系统自适应保护	支持	不支持
控制台	开箱即用，可配置规则、查看秒级监控、机器发现等	不完善
常见框架的适配	Servlet、Spring Cloud、Dubbo、gRPC 等	Servlet、Spring Cloud Netflix

5.3、Sentinel 入门

5.3.1、初识 sentinel

Sentinel 是阿里巴巴开源的一款微服务流量控制组件。官网地址：https://sentinelguard.io/zh-cn/index.html

Sentinel 具有以下特征:

丰富的应用场景：Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。

完备的实时监控：Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。

广泛的开源生态：Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。

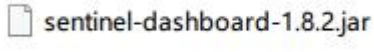
完善的 SPI 扩展点：Sentinel 提供简单易用、完善的 SPI 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

有兴趣的同学可参考：https://github.com/alibaba/Sentinel/wiki/%E7%86%94%E6%96%AD%E9%99%8D%E7%BA%A7

5.3.2、下载与安装

➢下载：

sentinel 官方提供了 UI 控制台，方便我们对系统做限流设置，下载地址为：https://github.com/alibaba/Sentinel/releases,也可使用课件中：



➢运行：将 jar 包放到任意非中文目录，执行命令：

```
java -jar sentinel-dashboard-1.8.2.jar
```

如果要修改 Sentinel 的默认端口、账户、密码，可以通过下列配置：

配置项	默认值	说明
server.port	8080	服务端口
sentinel.dashboard.auth.username	sentinel	默认用户名
sentinel.dashboard.auth.password	sentinel	默认密码

例如，修改端口：

```
java -Dserver.port=8090 -jar sentinel-dashboard-1.8.2.jar
```

➢访问，访问 http://localhost:8080 页面，就可以看到 sentinel 的控制台了：

需要输入账号和密码，默认都是：sentinel，登录后，发现一片空白，什么都没有：

这是因为我们还没有与微服务整合。

5.4、微服务整合 Sentinel

我们在 **用户微服务** 中整合 sentinel，并连接 sentinel 的控制台，步骤如下：

5.4.1、引入 sentinel 依赖

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

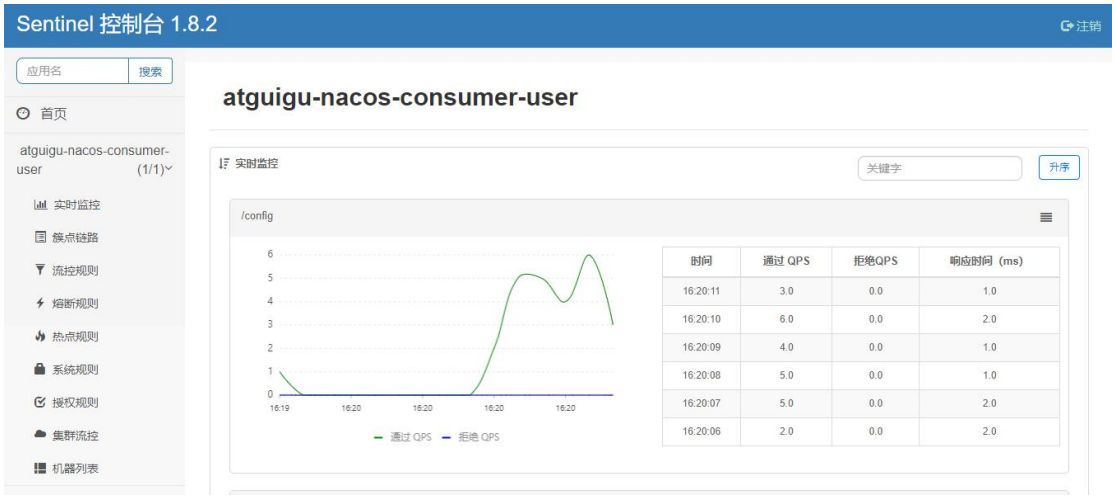
5.4.2、配置控制台

修改 application.yaml 文件，添加下面内容：

```
spring:
  cloud:
    nacos:
      sentinel:
        transport:
          dashboard: localhost:8080
```

5.4.3、访问用户微服务的任意接口

打开浏览器，访问 http://localhost:6700/config，这样才能触发 sentinel 的监控。然后再访问 sentinel 的控制台查看效果：



即可查看到实时监控信息。**sentinel 默认采用的是懒加载方式。**

QPS（Query Per Second）：每秒请求数，就是说服务器在一秒的时间内处理了多少个请求。

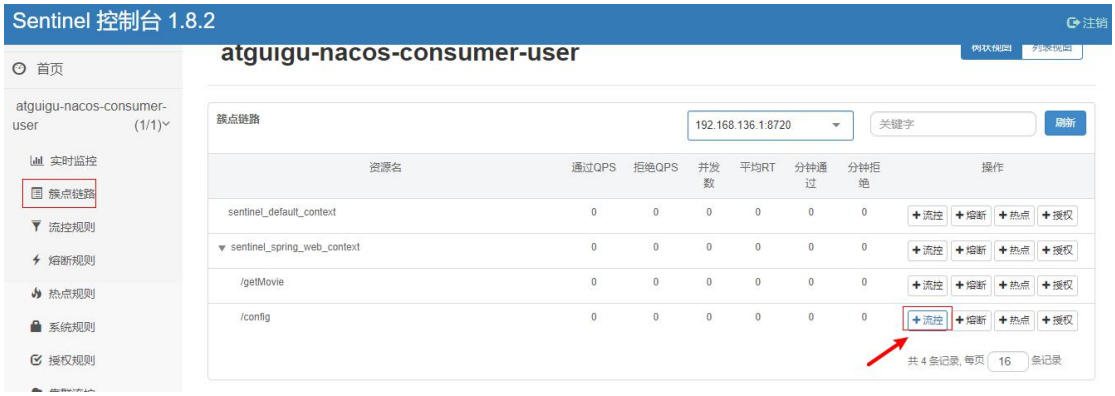
5.5、流量控制

雪崩问题虽然有四种方案，但是限流是避免服务因突发的流量而发生故障，是对微服务雪崩问题的预防。我们先学习这种模式。

5.5.1、入门介绍

流控、熔断等都是针对簇点链路中的资源来设置的，因此我们可以点击对应资源后面的按钮来设置规则：

- 流控：流量控制
- 降级：降级熔断
- 热点：热点参数限流，是限流的一种
- 授权：请求的权限控制



流量规则的重要属性：

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，QPS 模式（1）或并发线程数模式（0）	QPS 模式
limitApp	流控针对的调用来源	`default`，代表不区分调用来源
strategy	调用关系限流策略：直接、链路、关联	根据资源本身（直接）
controlBehavior	流控效果（直接拒绝/WarmUp/匀速+排队等待），不支持按调用关系限流	直接拒绝
clusterMode	是否集群限流	否

同一个资源可以同时有多个限流规则，检查规则时会依次检查。

理解上面规则的定义之后，我们可以通过调用 `FlowRuleManager.loadRules()` 方法来用硬编码的方式定义流量控制规则，比如：

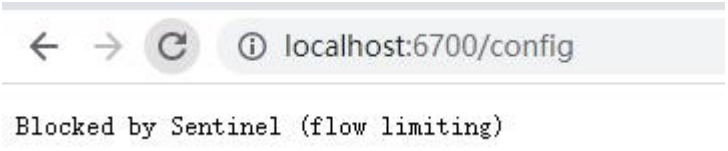
```
private void initFlowQpsRule() {
    List<FlowRule> rules = new ArrayList<>();
    FlowRule rule = new FlowRule(resourceName);
    // set limit qps to 20
    rule.setCount(20);
    rule.setGrade(RuleConstant.FLOW_GRADE_QPS);
    rule.setLimitApp("default");
    rules.add(rule);
    FlowRuleManager.loadRules(rules);
}
```

5.5.2、阈值类型

测试 1：测试 QPS

给 config 路径设置 默认流控效果(直接快速失败)

表示 1 秒钟内查询超过次数 2，就直接-快速失败，报默认错误.浏览器快速访问地址: http://localhost:6700/config



Blocked by Sentinel (flow limiting)

测试 2：测试线程数流控

表示，最大并发数是 3，超过 3 将直接流控.这里我们借助于 Jemeter 测试:。

5.5.3、流控模式

直接：统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式

关联：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流

链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流

测试关联模式：添加两个接口，如下所示:

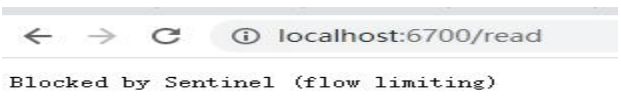
```java
@GetMapping("/write")
public String write(){
    return port;
}
@GetMapping("/read")
public String read(){
    return port;
}
```

设置关联流控规则：

查询和修改操作会争抢数据库锁，产生竞争。当/write 资源访问量触发阈值时，就会对/read 资源限流，避免影响/write 资源。

在 Jemeter 中设置：

在浏览器访问：http://localhost:6700/read ，结果如下所示：




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

满足下面条件可以使用关联模式：

两个有竞争关系的资源

一个优先级较高，一个优先级较低

测试链路模式：

链路模式：只针对从指定链路访问到本资源的请求做统计，判断是否超过阈值。

例如有两条请求链路：

/test1 /common

/test2 /common

如果只希望统计从/test2 进入到/common 的请求，则可以这样配置：

Sentinel 默认只标记 Controller 中的方法为资源，如果要标记其它方法，需要利用@SentinelResource 注解，示

```
@SentinelResource
public void common() {
    System.out.println("请求来了.....");
}
```

Sentinel 默认会将 Controller 方法做 context 整合，导致链路模式的流控失效，需要修改 application.yml，添加

```
spring:
  cloud:
    sentinel:
      web-context-unify: false #如果不关闭，所有controller层的方法对service层调用都认为是同一个根链路
```

流控模式小结：

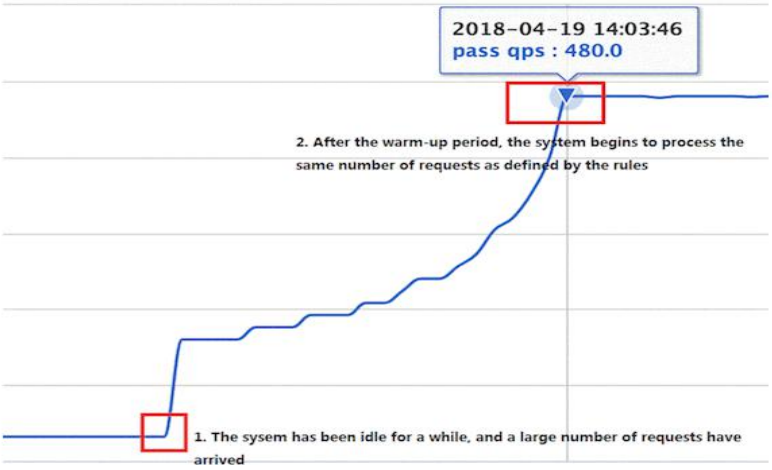
```
直接：对当前资源限流
关联：高优先级资源触发阈值，对低优先级资源限流。
链路：阈值统计时，只统计从指定资源进入当前资源的请求，是对请求来源的限流
```

5.5.4、流控效果

```
1.快速失败：直接失败，抛出异常
2.Warm Up：根据冷加载因子（默认3）的值，从阈值/冷加载因子，经过预热时长，才达到设置的QPS阈值
3.排队等待：匀速排队，让请求匀速通过，阈值类型必须设置为QPS，否则无效
```

测试 1:**Warm up(预热)流控模式**

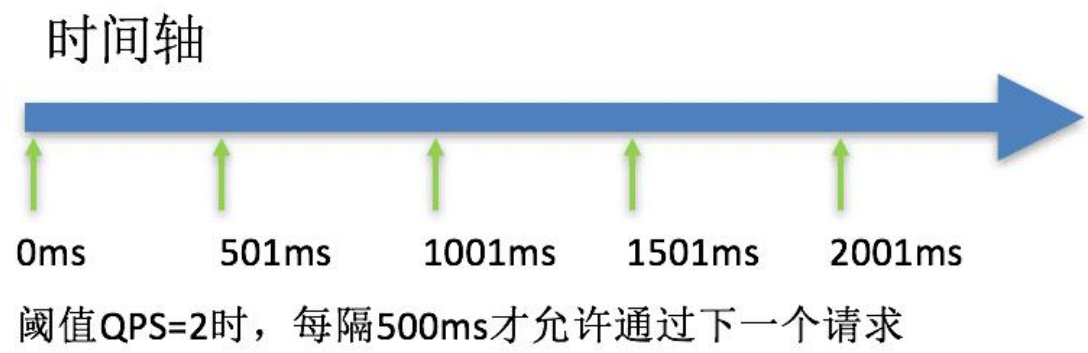
当系统运行时,有并发请求来访问系统时，为了避免系统崩溃，可以设置一个阈值，让项目可以处理请求的数量逐渐增加到设定的阈值。请求阈值初始值是 threshold / coldFactor，持续指定时长后，逐渐提高到 threshold 值。而 coldFactor 的默认值是 3。通常冷启动的过程系统允许通过的 QPS 曲线如下图所示：



测试设置：

测试 2:匀速排队（RuleConstant.CONTROL_BEHAVIOR_RATE_LIMITER）方式会严格控制请求通过的间隔时间，也即是让请求以均匀的速度通过，对应的是漏桶算法。详细文档可以参考 流量控制 - 匀速器模式

该方式的作用如下图所示：



这种方式主要用于处理间隔性突发的流量，例如消息队列。想象一下这样的场景，在某一秒有大量的请求到来，而接下来的几秒则处于空闲状态，我们希望系统能够在接下来的空闲期间逐渐处理这些请求，而不是在第一秒直接拒绝多余的请求。

注意：匀速排队模式暂时不支持 QPS > 1000 的场景。

添加 controller 方法：

```
@GetMapping("/test")
public String test1(){
    System.out.println(System.currentTimeMillis());
    return port;
}
```

控制台打印如下：

```
1645093263021
1645093264021
1645093265021
```

流控效果总结:

快速失败：QPS超过阈值时，拒绝新的请求
warm up： QPS超过阈值时，拒绝新的请求；QPS阈值是逐渐提升的，可以避免冷启动时高并发导致服务宕机。
排队等待：请求会进入队列，按照阈值允许的时间间隔依次执行请求；如果请求预期等待时长大于超时时间，直接拒绝

5.6、热点 key 限流

热点参数规则（ParamFlowRule）类似于流量控制规则（FlowRule）：

属性	说明	默认值
resource	资源名，必填	
count	限流阈值，必填	
grade	限流模式	QPS 模式
durationInSec	统计窗口时间长度（单位为秒），1.6.0 版本开始支持	1s
controlBehavior	流控效果（支持快速失败和匀速排队模式），1.6.0 版本开始支持	快速失败
maxQueueingTimeMs	最大排队等待时长（仅在匀速排队模式生效），1.6.0 版本开始支持	0ms
paramIdx	热点参数的索引，必填，对应 `SphU.entry(xxx, args)` 中的参数索引位置	
paramFlowItemList	参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 `count` 阈值的限制。**仅支持基本类型和字符串类型**	
clusterMode	是否是集群参数流控规则	`false`
clusterConfig	集群流控相关配置	

我们可以通过 ParamFlowRuleManager 的 loadRules 方法更新热点参数规则，下面是一个示例：

```
ParamFlowRule rule = new ParamFlowRule(resourceName)
    .setParamIdx(0)
    .setCount(5);
// 针对 int 类型的参数 PARAM_B，单独设置限流 QPS 阈值为 10，而不是全局的阈值 5.
ParamFlowItem item = new ParamFlowItem().setObject(String.valueOf(PARAM_B))
    .setClassType(int.class.getName())
    .setCount(10);
rule.setParamFlowItemList(Collections.singletonList(item));

ParamFlowRuleManager.loadRules(Collections.singletonList(rule));
```

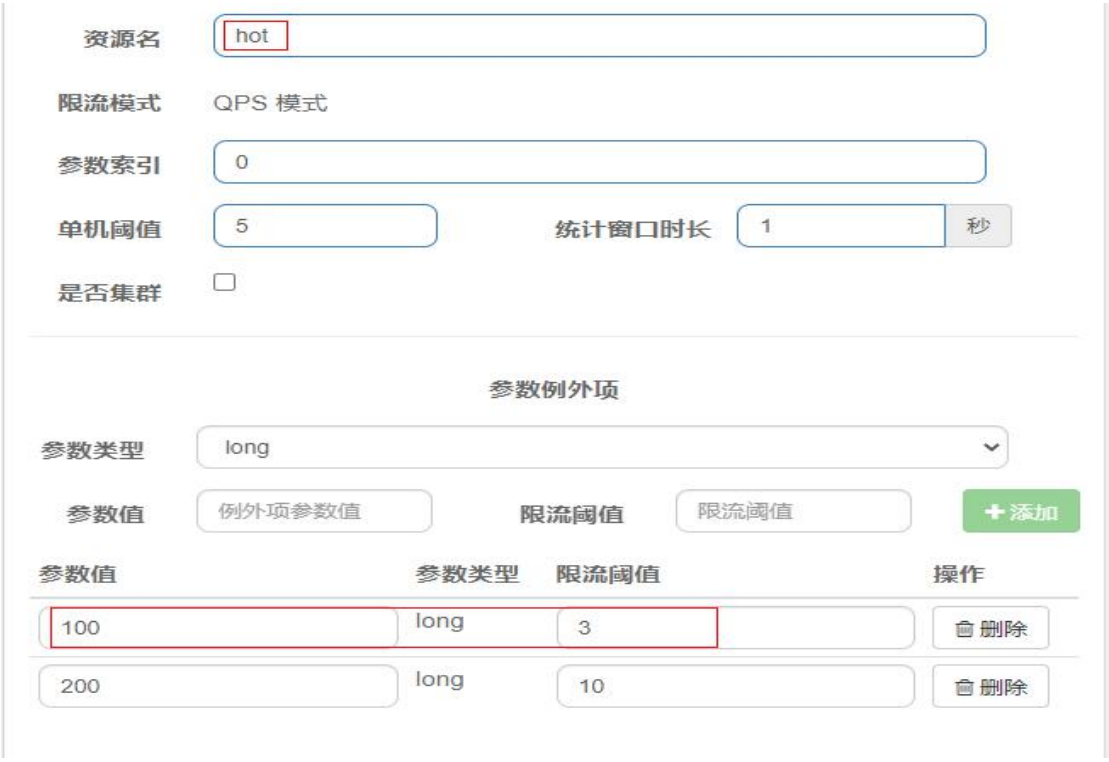
总之，流控是对某个资源的所有请求，判断是否超过 QPS 阈值。而热点参数限流是分别统计**参数值相同**的请求，判断是否超过 QPS 阈值。

上述所示：针对 id 值为 1 的，QPS 为 3，针对 id 值为 2 的，QPS 为 1

添加接口：

```
@SentinelResource(value = "hot")
@GetMapping("/resource/{hid}")
public String testHotResource(@PathVariable("hid") Long hid){
  System.out.println(hid);
  return port;
}
```

配置如下：



代表的含义是：对 hot 这个资源的 **0** 号参数（第一个参数）做统计，**每 1 秒相同参数值的请求数不能超过 5**

在热点参数限流的高级选项中，可以对部分参数设置例外配置：

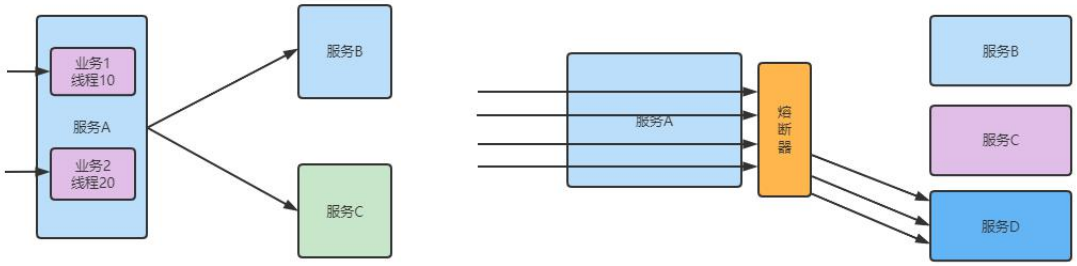
结合上一个配置，这里的含义是对0号的long类型参数限流，每1秒相同参数的QPS不能超过5，有两个例外：
如果参数值是100，则每1秒允许的QPS为3
如果参数值是200，则每1秒允许的QPS为10

需要注意：热点参数限流对默认的 SpringMVC 资源无效，所以需要使用 SentinelResource 注解。

5.7、线程隔离与熔断降级介绍

虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因为其它原因而故障。而要将这些故障控制在一定范围，避免雪崩，就要靠**线程隔离（舱壁模式）**和**熔断降级**手段了。

不管是线程隔离还是熔断降级，都是对**客户端（调用方）**的保护。



5.8、线程隔离

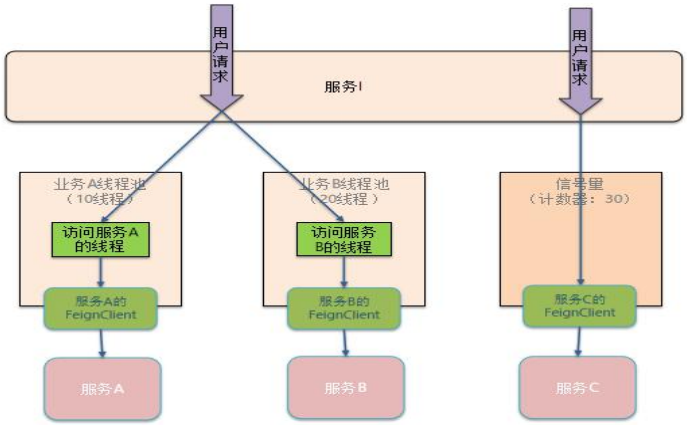
5.8.1、线程隔离方式介绍

线程隔离有两种方式实现：

1.线程池隔离

2.信号量隔离（Sentinel 默认采用）

5.8.2、两者区别



线程池和信号量的区别？		
	线程池隔离	信号量隔离
线程	请求线程和调用provider线程不是同一条线程	请求线程和调用provider线程是同一条线程
开销	排队、调度、上下文开销等	无线程切换，开销低
异步	支持	不支持
并发支持	支持（最大线程池大小）	支持（最大信号量上限）
传递Header	无法传递http Header	可以传递http Header
支持超时	能支持超时	不支持超时

线程池隔离：调用线程和 hystrixCommand 线程不是同一个线程，并发请求数受到线程池（不是容器 tomcat 的线程池，而是 hystrixCommand 所属于线程组的线程池）中的线程数限制。

信号量隔离：调用线程和 hystrixCommand 线程是同一个线程，默认最大并发请求数是 10

注意：当然上述线程池隔离和信号量隔离都是以 Hystrix 为例讲的,sentinel 只支持信号量隔离。

5.8.3、线程池隔离与信号量隔离各自的应用场景

1.什么情况下，用线程池隔离？

请求并发量大，并且耗时长（请求耗时长一般是计算量大，或读数据库）：采用线程隔离策略，这样的话，可以保证大量的容器(tomcat)线程可用，不会由于服务原因，一直处于阻塞或等待状态，快速失败返回。

2.什么情况下，用信号量隔离？

请求并发量大，并且耗时短（请求耗时短可能是计算量小，或读缓存）：采用信号量隔离策略，因为这类服务的返回通常会非常的快，不会占用容器线程太长时间，而且也减少了线程切换的一些开销，提高了缓存服务的效率。

5.8.4、Hystrix 与 sentinel 各自配置

Hystrix 的隔离策略有两种：分别是线程隔离和信号量隔离。

THREAD（线程隔离）：使用该方式HystrixCommand将会在单独的线程上执行，并发请求受线程池中线程数量的限制。
SEMAPHORE（信号量隔离）：使用该方式，HystrixCommand将会在调用线程上执行，开销相对较小，并发请求受信号量的个数的限制。

Hystrix 支持信号量隔离与线程池隔离，**默认使用你的是线程池隔离**。如下所示：

```
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: 默认是Thread, 可选Thread | Semaphore
```

sentinel 仅仅支持**信号量**隔离：

线程数：是该资源能使用用的 tomcat 线程数的最大值。也就是通过限制线程数量，实现**舱壁模式**。

5.8.5、线程隔离小结

线程隔离的两种手段是？信号量隔离、线程池隔离

信号量隔离的特点是？基于计数器模式，简单，开销小

线程池隔离的特点是？基于线程池模式，有额外开销，但隔离控制更强

5.9、熔断降级

5.9.1、sentinel 与 feign 整合

第一步：修改 **用户微服务** 的 application.yml 文件，开启 Feign 的 Sentinel 功能

```
feign:
 sentinel:
   enabled: true # 开启Feign的Sentinel功能
```

第二步：给 FeignClient 编写失败后的降级逻辑

方式一：FallbackClass，无法对远程调用的异常做处理

方式二：FallbackFactory，可以对远程调用的异常做处理，我们选择这种

具体步骤：

步骤一：在 atguigu-nacos-consumer-user6700 项目中定义类，实现 FallbackFactory：

```
public class MovieFeignClientFallbackFactory implements FallbackFactory<MovieFeignClient> {
    @Override
    public MovieFeignClient create(Throwable throwable) {
        return new MovieFeignClient() {
            @Override
            public Movie getNewMovie() {
                System.out.println(throwable.getMessage());
                return new Movie(-200,"网络拥挤，请稍后重试.....");
            }
        };
    }
}
```

步骤二：在 atguigu-nacos-consumer-user6700 项目中将 MovieFeignClientFallbackFactory 定义为一个 bean 对象

```
@Configuration
public class FallbackConfig {

    @Bean
    public MovieFeignClientFallbackFactory getMovieFeignClientFallbackFactory(){
        return new MovieFeignClientFallbackFactory();
    }
}
```

步骤三：在 atguigu-nacos-consumer-user6700 项目中的 MovieFeignClient 接口中使用 MovieFeignClientFallbackFactory

```
@FeignClient(value = "atguigu-nacos-provider-movie",fallbackFactory = MovieFeignClientFallbackFactory.class)
//绑定Feign客户端要访问的服务。
public interface MovieFeignClient {
    // 获取最新电影
    @GetMapping("/movie/latest")
    public Movie getNewMovie();
}
```

先正常访问用户微服务：能够获取到用户信息和远程调用的电影信息。

将电影微服务停掉，会发现再次请求用户微服务，只能获取到用户信息，不能获取到电影信息，走了降级方法。如下：

```
{
  - movie: {
        id: -200,
        movieName: "网络拥挤，请稍后重试....."
    },
  - user: {
        id: 2,
        userName: "张三"
    }
}
```

Sentinel 支持的雪崩解决方案：

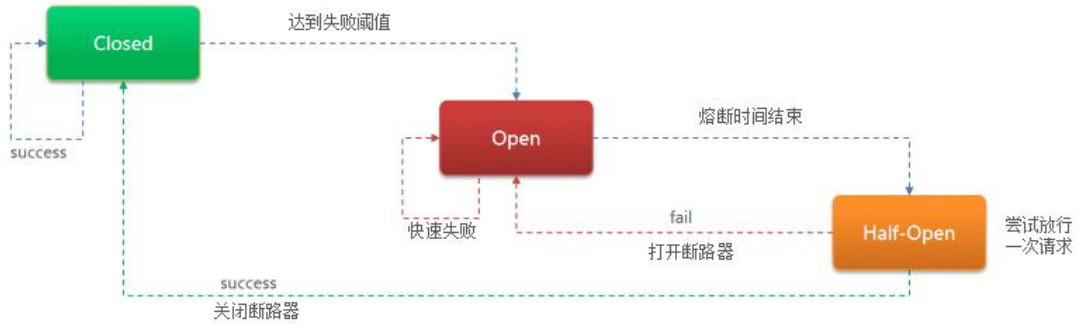
```
线程隔离（仓壁模式）
降级熔断
```

Feign 整合 Sentinel 的步骤：

```
在application.yml中配置：feign.sentienl.enable=true
给FeignClient编写FallbackFactory并注册为Bean
将FallbackFactory配置到FeignClient
```

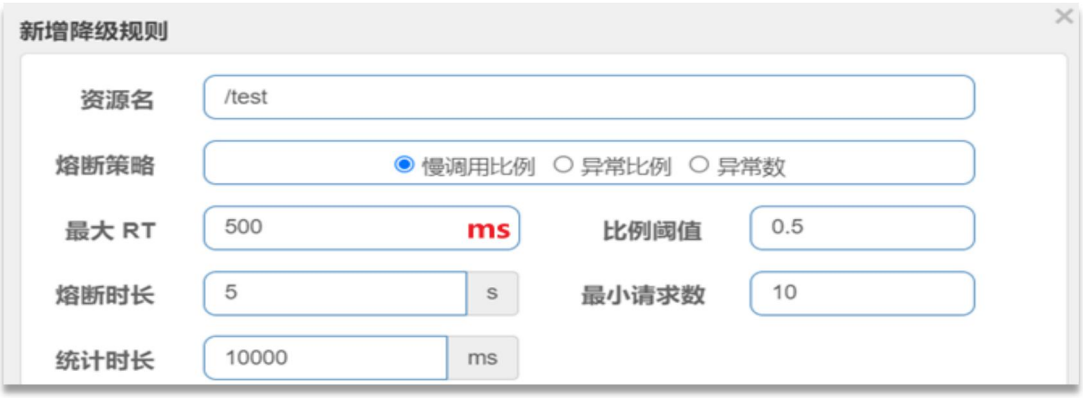
5.9.2、熔断器的三种状态

熔断降级是解决雪崩问题的重要手段。其思路是由**断路器**统计服务调用的**异常比例、慢请求比例**，如果**超出阈值**则会**熔断**该服务。即拦截访问该服务的一切请求；而当服务恢复时，断路器会放行访问该服务的请求。



断路器熔断策略有三种：慢调用、异常比例、异常数

慢调用：业务的响应时长（RT）大于指定时长的请求认定为慢调用请求。在指定时间内，如果请求数量超过设定的最小数量，慢调用比例大于设定的阈值，则触发熔断。例如:



解读：RT 超过 500ms 的调用是慢调用，统计最近 10000ms 内的请求，如果请求量超过 10 次，并且慢调用比例不低于 0.5，则触发熔断，熔断时长为 5 秒。然后进入 half-open 状态，放行一次请求做测试。

在电影微服务中加入如下代码测试：

```
// 获取最新电影
@GetMapping("/latest")
public Movie getNewMovie(@RequestParam("id") Integer id) throws InterruptedException {
    if(id == 1){
        Thread.sleep( millis: 3000);
    }
    return movieService.getNewMovie();
}
```

异常比例或异常数：统计指定时间内的调用，如果调用次数超过指定请求数，并且出现异常的比例达到设定的比例阈值（或超过指定异常数），则触发熔断。例如：

解读：统计最近 1000ms 内的请求，如果请求量超过 10 次，并且异常比例不低于 0.5，则触发熔断，熔断时长为 5 秒。然后进入 half-open 状态，放行一次请求做测试。

在电影微服务中加入如下代码进行测试：

```
// 获取最新电影
@GetMapping("/latest")
public Movie getNewMovie(@RequestParam("id") Integer id) throws InterruptedException {
    if(id == 1){
        Thread.sleep( millis: 3000);
    }else if(id==2){
        throw new RuntimeException("测试异常比例与异常数。");
    }
    return movieService.getNewMovie();
}
```

5.9.3、熔断降级小结

Sentinel 熔断降级的策略有哪些?

慢调用比例：超过指定时长的调用为慢调用，统计单位时长内慢调用的比例，超过阈值则熔断

异常比例：统计单位时长内异常调用的比例，超过阈值则熔断

异常数：统计单位时长内异常调用的次数，超过阈值则熔断

5.10、授权规则及自定义异常处理

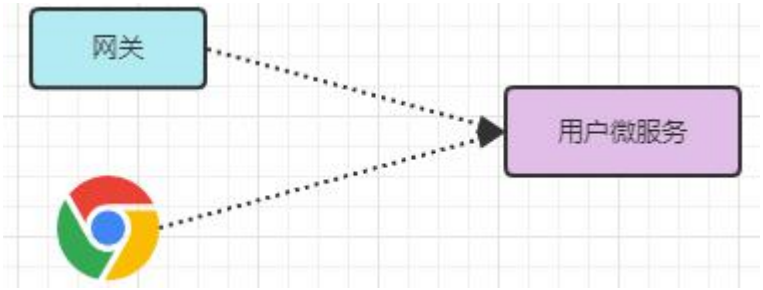
5.10.1、授权规则

授权规则可以对**调用方的来源**做控制，有白名单和黑名单两种方式。

白名单：来源（origin）在白名单内的调用者允许访问

黑名单：来源（origin）在黑名单内的调用者不允许访问

例如，我们限定只允许从网关来的请求访问 用户微服务，那么流控应用中就填写网关的名称



Sentinel 是通过 RequestOriginParser 这个接口的 parseOrigin 来获取请求的来源的。

```
public interface RequestOriginParser {
    #从请求request对象中获取origin，获取方式自定义
    String parseOrigin(HttpServletRequest var1);
}
```

例如，我们尝试从 request 中获取一个名为 origin 的请求头，作为 origin 的值：

```
@Component
public class HeaderOriginParser implements RequestOriginParser {

    @Override
    public String parseOrigin(HttpServletRequest request) {
        String origin = request.getHeader("origin");
        if(StringUtils.isEmpty(origin)){return "blank";}
         return origin;
   }
}
```

我们还需要在 gateway 服务中，利用网关的过滤器添加名为 gateway 的 origin 头：

```
spring:
  cloud:
    gateway:
      default-filters:
        - AddRequestHeader=origin,gateway # 添加名为origin的请求头，值为gateway
```

给用户微服务的/movie 接口设置授权规则:

5.10.2、自定义异常结果

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 BlockExceptionHandler 接口:

```
public interface BlockExceptionHandler {
    #处理请求被限流、降级、授权拦截时抛出的异常：BlockException
    void handle(HttpServletRequest var1, HttpServletResponse var2, BlockException var3) throws Exception;
}
```

而 BlockException 包含很多个子类，分别对应不同的场景:

异常	说明
FlowException	限流异常
ParamFlowException	热点参数限流的异常
DegradeException	降级异常
AuthorityException	授权规则异常
SystemBlockException	系统规则异常

我们在 **用户微服务** 中定义类，实现 BlockExceptionHandler 接口:

具体参考课件中 SentinelExceptionHandler 文件:

```java
@Component
public class SentinelExceptionHandler implements BlockExceptionHandler {
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response, BlockException e) throws Exception {
        String msg = "未知异常";
        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;
        }

        response.setContentType("application/json;charset=utf-8");
        response.setStatus(status);
        Map<String,Object> map = new HashMap<String,Object>();
        map.put("msg", msg);
        map.put("status",status);
        response.getWriter().println(JSON.toJSONString(map));
    }
}
```

小结:

获取请求来源的接口是什么？ RequestOriginParser
处理BlockException的接口是什么？BlockExceptionHandler

5.11、系统保护规则（SystemRule）

Sentinel 系统自适应限流从**整体维度**对应用入口流量进行控制，结合应用的 Load、CPU 使用率、总体平均 RT、入口 QPS 和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。

系统规则支持以下的模式：

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 load1 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）。系统容量由系统的 `maxQps * minRt` 估算得出。设定参考值一般是 `CPU cores * 2.5`。
- **CPU usage**（1.5.0+ 版本）：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。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。

我们这里以系统规则中的入口 QPS 为例，给大家测试(系统规则，对整个系统的所有资源起作用，而不再是只对某个资源起作用。)

5.12、规则持久化

一旦我们重启应用，Sentinel 规则将消失，生产环境需要将配置规则进行持久化。Nacos 规则持久化的三种方式：

原始模式：控制台配置的规则直接推送到 Sentinel 客户端，也就是我们的应用。然后保存在内存中，服务重启则丢失

pull 模式：控制台将配置的规则推送到 Sentinel 客户端，而客户端会将配置规则保存在本地文件或数据库中。以后会**定时**去本地文件或数据库中查询，更新本地规则。

push 模式：控制台将配置规则推送到远程配置中心，例如 Nacos。Sentinel 客户端监听 Nacos，获取配置变更的推送消息，完成本地配置更新。

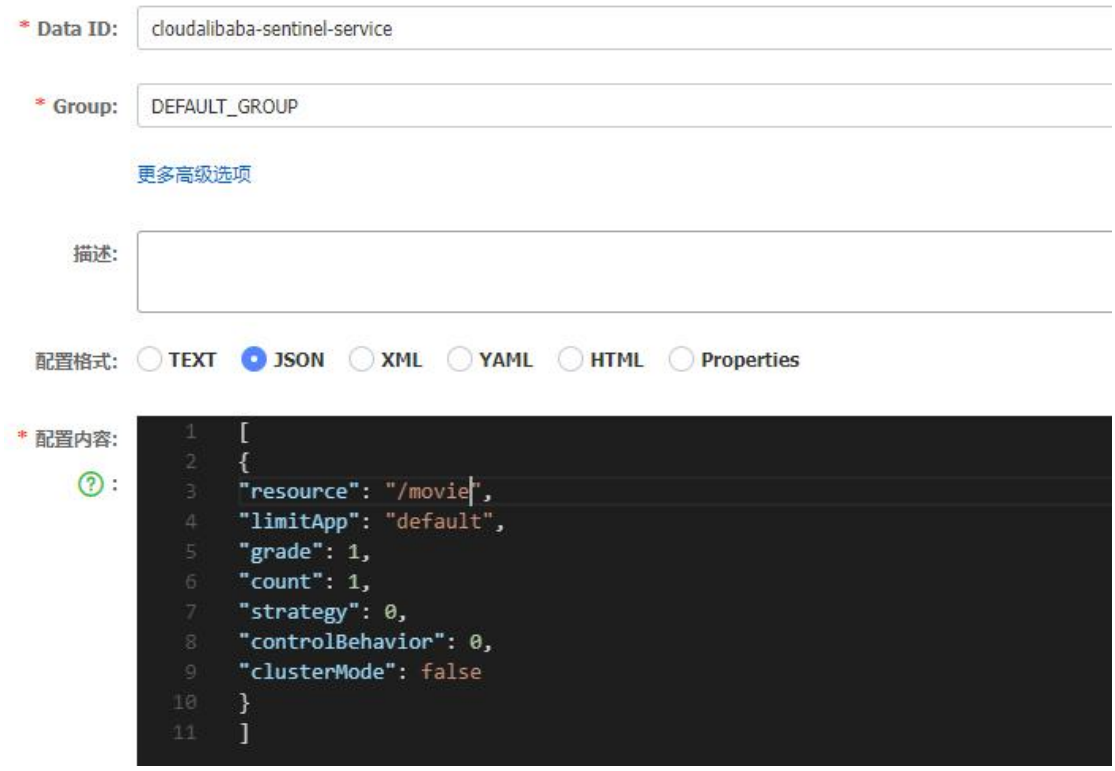
5.12.1、添加依赖信息

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

5.12.2、修改 yml 文件

```
spring:
  cloud:
    sentinel:
      datasource:
        ds1:
          nacos:
            username: nacos   #高版本的nacos一定要指定用户名和密码，否则连接不上，报错
            password: nacos
            server-addr: localhost:8848
            dataId: cloudalibaba-sentinel-service
            groupId: DEFAULT_GROUP
            data-type: json
            rule-type: flow # 还可以是：degrade、authority、param-flow,system
```

5.12.3、添加 Nacos 业务规则配置




```
[
{
"resource": "/movie",
"limitApp": "default",
"grade": 1,
"count": 1,
"strategy": 0,
"controlBehavior": 0,
"clusterMode": false
}
]
```