# 微服务之SpringCloud Alibaba实践

# 微服务架构分析及发展

## 背景分析

讲微服务之前，我们先分析以下单体应用。所谓单体应用一般是基于idea/eclipse，maven等建一个工程，然后基于SpringBoot，spring，mybatis框架进行整合，接下来再写一堆dao、mapper、service、controller，再加上一些的配置文件，有可能还会引入redis、elasticsearch、mq等其它项目的依赖，开发好之后再将项目打包成一个jar包/war包。然后再将包扔到类似tomcat这样的web服务中，最后部署到公司提供给你的linux服务器上。 接下来，你针对服务提供的访问端口（例如8080端口）发起http请求，请求会由tomcat直接转交给你的spring web组件，进行一层一层的代码调用。对于这样的设计一般适合企业的内部应用，访问量不大，体积比较小，5人以内的团队即可开发和维护。但对于一些大型互联网项目，假如需要10人以上的开发和维护团队，仅频繁的创建代码分支，编写业务功能，然后合并分支，就会出现很多代码冲突。每次解决这些大量的代码冲突，可能就会耗费好几天的时间。总之对于大型的单体应用可能存在的如下几个问题：

（1）很多人维护一个单块应用，频繁的进行代码合并，频繁的解决代码冲突，解决冲突的时间和成本很高的，导致开发效率低下。

（2）每次上线都要跟最新代码进行合并，重新进行全量功能的回归测试，很多代码都可能有变动，必须全量回归测试，耗费时间很多，开发效率低下。

（3）多人频繁上线，你等我，我等你，互相协调困难，而且可能会出现别人多次先上线，你多次重复的合并代码，解决冲突，全量回归测试，做很多次重复的事情，导致效率低下

（4）测试服务器都很少，就一台测试服务器，你开发好了代码，你连测试都不能测，必须等待别人的分支先测试完毕上线，你才能去合并代码，解决冲突，等到一个测试环境可以用，回归测试。

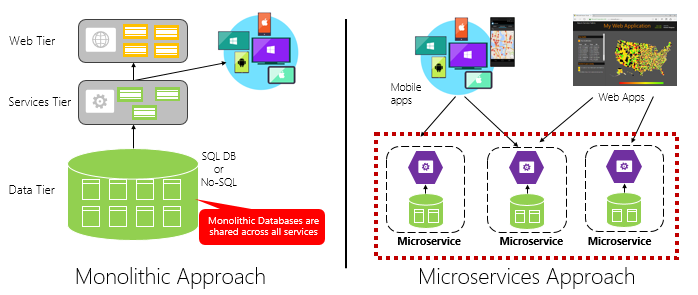
（5）10个人以内维护一个单块应用，基本这些成本还不算太大，但是一旦10人以上维护一个单块应用，上述的成本会极大，导致系统每个需求的测试和上线，都非常缓慢，要耗费大量时间做全量回归测试，上线日期还得互相配合互相等待互相协调，一个疏忽，就可能导致没侧测试完全的代码上线出线上事故。

（6）如果你想要升级技术架构，不能随意升级，因为可能影响别人，你得让所有人都学习这个新技术架构才行；如果你想要升级已有技术的版本，不能随意升级，因为可能新版本对你的代码没影响，但是对别人的代码有影响。

基于如上问题，超出需要10人开发和维护的项目一般建议的比较合理的做法就是进行微服务架构设计，把大系统拆分为很多小系统，几个人负责一个服务这样每个服务独立的开发、测试和上线，代码冲突少了，每次上线就回归测试自己的一个服务即可，测试速度快了，上线是独立的，只要向后兼容接口就行了，不需要跟别人等待和协调，技术架构和技术版本的升级，几个人ok就行，成本降低，更加灵活了。

## 什么是微服务

微服务架构（MSA）的基础是将单个应用程序开发为一组小型独立服务，这些独立服务在自己的进程中运行，独立开发和部署。如图所示：



打开Git客户端工具，配置用户和密码，用于识别提交代码的用户。

这些服务使用轻量级 API 通过明确定义的接口进行通信。这些服务是围绕业务功能构建的，每项服务执行一项功能。由于它们是独立运行的，因此可以针对各项服务进行更新、部署和扩展，以满足对应用程序特定功能的需求。

生活中的微服务，如图所示：



总之，微服务是分布式系统中的一种流行的架构模型，它并不是银弹，所以，也不要寄希望于微服务构架能够解决所有的问题。微服务架构主要解决的是如何快速地开发和部署我们的服务，这对于一个能够适应快速开发和成长的公司是非常必要的。同时，微服务设计中有很多很不错的想法和理念，通过学习微服务架构我们可以更快的迈向卓越。

## SpringCloud 解决方案

### 概述

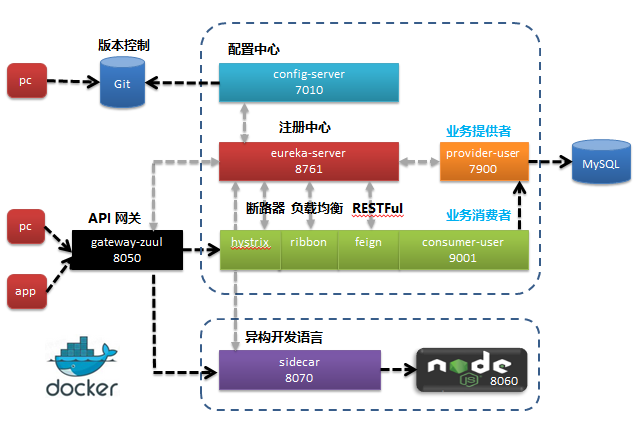
对于微服务架构，几乎所有的大公司走的都是自研路线。例如，国内最初的典型代表就是基于阿里的Dubbo进行实现，后来国外的netflix公司将他们内部的微服务技术架构进行了开源，在互联网行业产生了很大的影响力，然后接着就被整合到了spring社区，变成了spring cloud项目。

### 核心组件分析

Spring Cloud这个项目里面整合的基本上都是netflix公司的微服务相关组件，例如借助eureka做注册中心、feign和ribbon结合实现远程过程调用和负载均衡、zuul做网关、hystrix做熔断，用zipkin和sleuth做链路监控，stream做消息中间件集成，contract做契约测试支持，然后跟spring security配合的做安全认证，跟k8s配合的进行容器化操作等。Spring cloud将这些组件打包集成在一起，进行微服务架构实现，最初在国外比较有市场，几年前在国内也火了，大量公司都开始拥抱spring cloud，尤其是中小型公司。

### 解决方案架构设计

基于Spring Cloud实现的微服务，解决方案设计架构如图所示：



## Spring Cloud Alibaba 解决方案

### 概述

Spring Cloud Alibaba 是Spring Cloud的一个子项目，致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。

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

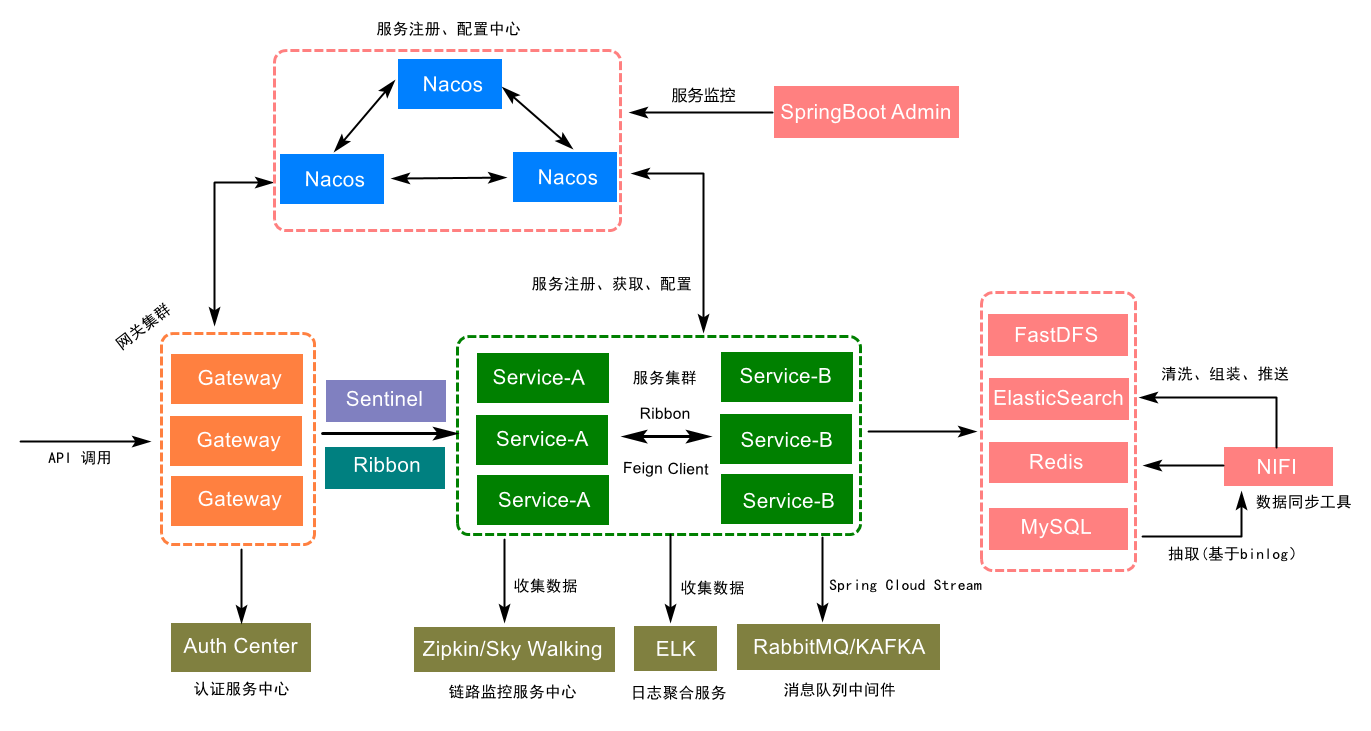
### 核心组件分析

Spring Cloud Alibaba 默认提供了如下核心功能：

* 服务限流降级：默认支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Spring Cloud Gateway, Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
* 服务注册与发现：基于Spring Cloud 服务注册与发现标准，借助Nacos进行实现，默认还集成了 Ribbon 的支持。
* 分布式配置管理：基于Nacos支持分布式系统中的外部化配置，配置更改时自动刷新。
* 消息驱动能力：基于Spring Cloud Stream 为微服务应用构建消息驱动能力。
* 分布式事务：使用 @GlobalTransactional 注解， 高效并且对业务零侵入地解决分布式事务问题。。
* 分布式任务调度：提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker（schedulerx-client）上执行。

### 解决方案架构设计

基于Spring Cloud Alibaba实现的微服务，解决方案设计架构如图所示：



## 小节总结(Summary)

总之，微服务是一个架构设计方式，此架构中的每个服务都是针对一组功能而设计的，并专注于解决特定的问题。如果开发人员逐渐将更多代码增加到一项服务中并且这项服务变得复杂，那么可以将其拆分成多项更小的服务。接下来进行独立的开发、测试、部署、运行、维护。进而更好，更灵活的处理客户端的请求并提高系统的可靠性，可扩展性。

# 服务注册中心Nacos应用实践

## 背景分析

在微服务中，首先需要面对的问题就是如何查找服务，如何在不同的服务之间进行通信？如何更好更方便的管理应用中的每一个服务，如何建立各个服务之间联系的纽带，由此注册中心诞生（例如淘宝网卖家提供服务，买家调用服务）。

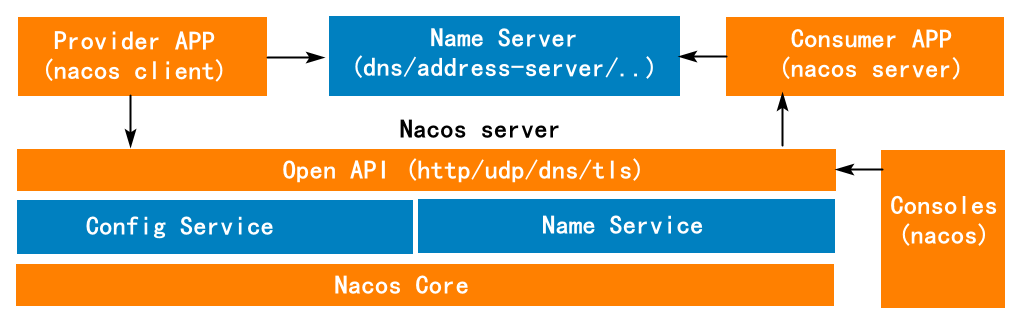
市面上常用注册中心有Zookeeper,Eureka,Nacos,Consul,那他们分别都有什么特点，我们如何进行选型呢？

从社区活跃度上分析，consul和新晋的nacos的社区活跃度是比较高的，nacos可以同时支持AP和CP；consul则结合了zookeeper和nacos的诸多优点，还天然

支持多数据中心，而zookeeper则又可以唯一感知到服务状态的实时变化；Eureka的自我保护机制使得它即使只剩下一台服务，也不影响正常运行，具有高可用性，那么选择的时候，究竟应该如何选择呢？需要结合业务场景来进行选择。比如说，对于金融类的业务场景，对于一致性要求更高，那么就会排除掉Eureka,然后根据易用性、性价比等其他方面再进行后续的选择；对于高可用比较注重的项目，如电商类项目，则可以选择Eurek或者Nacos，但再比较其他方面，Nacos不仅可以做注册中心，还可以作为架构中的配置中心，并且社区活跃度比较高，功能也日渐在完善，使用的人越来越多，因此综合来讲，就选择了Nacos。

## Nacos简介

Nacos（DynamicNaming and Configuration Service）是一个应用于服务注册与发现、配置管理的平台。它孵化于阿里巴巴，成长于十年双十一的洪峰考验，沉淀了简单易用、稳定可靠、性能卓越的核心竞争力。其基本架构如图所示：



其中：

* Provider APP为服务提供方，是指提供可复用和可调用服务的应用方。
* Consumer App 为服务消费方，是指会发起对某个服务调用的应用方。
* Name Server 实现namespace到clusterid路由，用户与nacos物理环境映射。
* Open API 暴露标准Rest风格HTTP接口，简单易用，方便多语言集成。
* Console 易用控制台，做服务管理、配置管理等操作。
* Config Service 负责系统的所有配置信息管理。
* Name Service提供分布式系统中所有对象的映射管理服务。

Nacos致力于帮助您发现、配置和管理微服务。Nacos提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。Nacos帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos是构建以“服务”为中心的现代应用架构的服务基础设施。

我们在后面微服务架构设计中，将使用Nacos作为微服务架构中的注册中心以及配置中心（spring cloud config）来使用。Nacos官网地址如下：

<https://nacos.io/zh-cn/docs/quick-start.html>

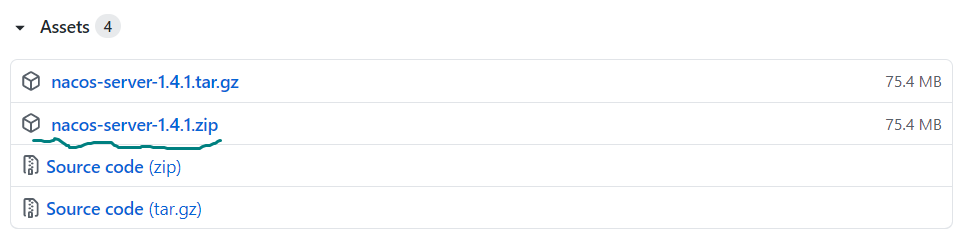
## 构建Nacos服务

### 下载与安装

第一步：Nacos下载,可在浏览器直接输入如下地址：

<https://github.com/alibaba/nacos/releases>

第二步：选择对应版本，直接下载，如图所示：

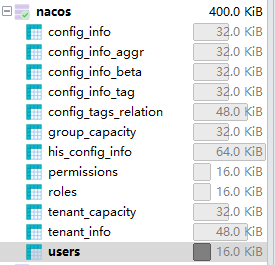


第三步：解压Nacos（最好不要解压到中文目录下），其目录结构如下：



### 初始化配置

第一步：找到/conf/nacos-mysql.sql文件里的sql脚本，然后登陆mysql，然后基于脚本文件中的描述创建数据库，并执行脚本文件，脚本执行成功会创建一些表，如图所示：



第二步：打开/conf/application.properties里打开默认配置，并基于你当前环境配置要连接的数据库，连接数据库时使用的用户名和密码：

### If use MySQL as datasource:

spring.datasource.platform=mysql

### Count of DB:

db.num=1

### Connect URL of DB:

db.url.0=jdbc:mysql://127.0.0.1:3306/nacos\_config?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useUnicode=true&useSSL=false&serverTimezone=UTC

db.user=root

db.password=root

### 服务启动与访问

第一步:启动Nacos服务。

Linux/Unix/Mac启动命令(standalone代表着单机模式运行，非集群模式):

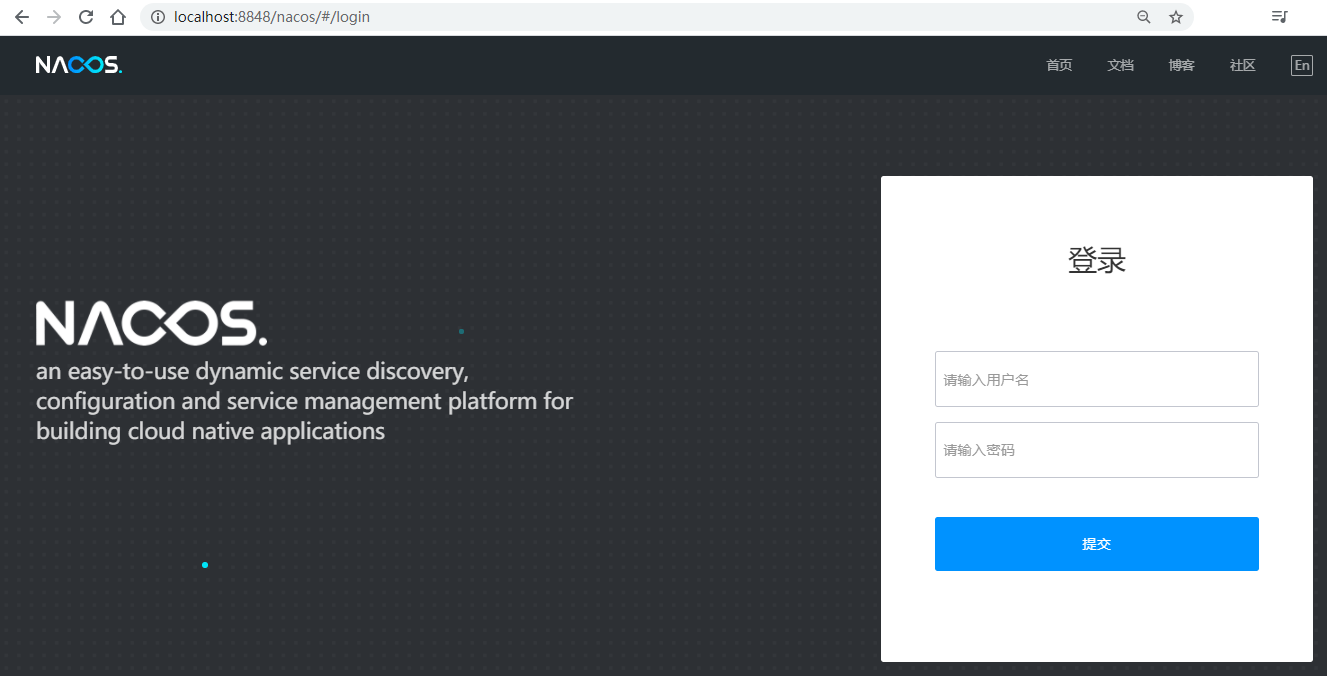
sh startup.sh -m standalone

Windows启动命令(standalone代表着单机模式运行，非集群模式):

startup.cmd -m standalone

第二步:访问Nacos服务。

打开浏览器，输入<http://localhost:8848/nacos>地址，出现如下登陆页面：

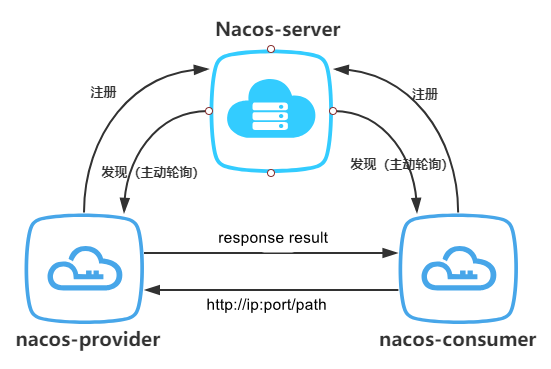


其中，默认账号密码为nacos/nacos.

## 服务注册与调用入门

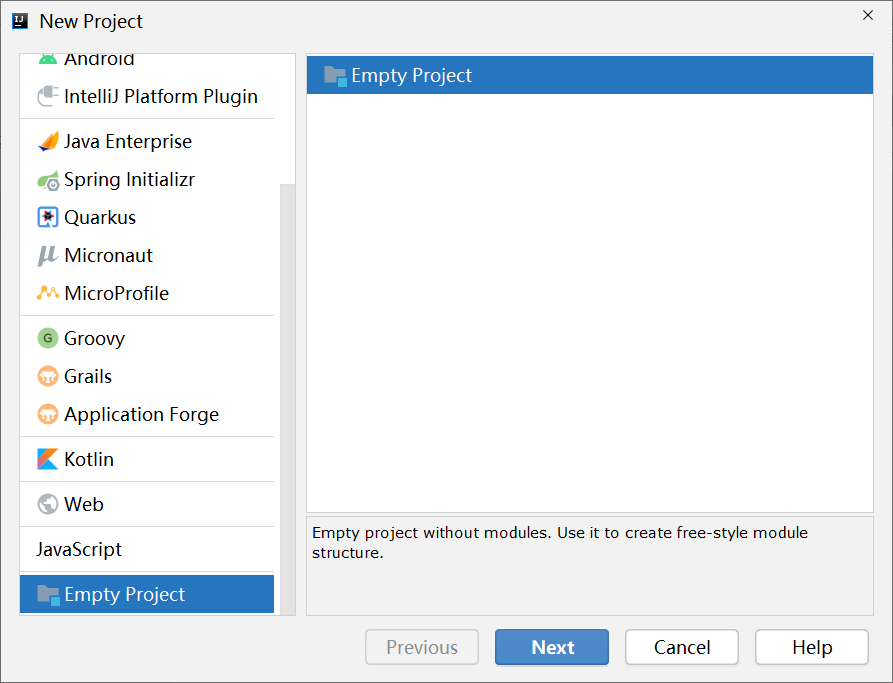
### 业务描述

创建两个项目Module分别为服务提供者和服务消费者，两者都要注册到NacosServer中，然后服务提供者可以为服务消费者提供远端调用服务，如图所示：

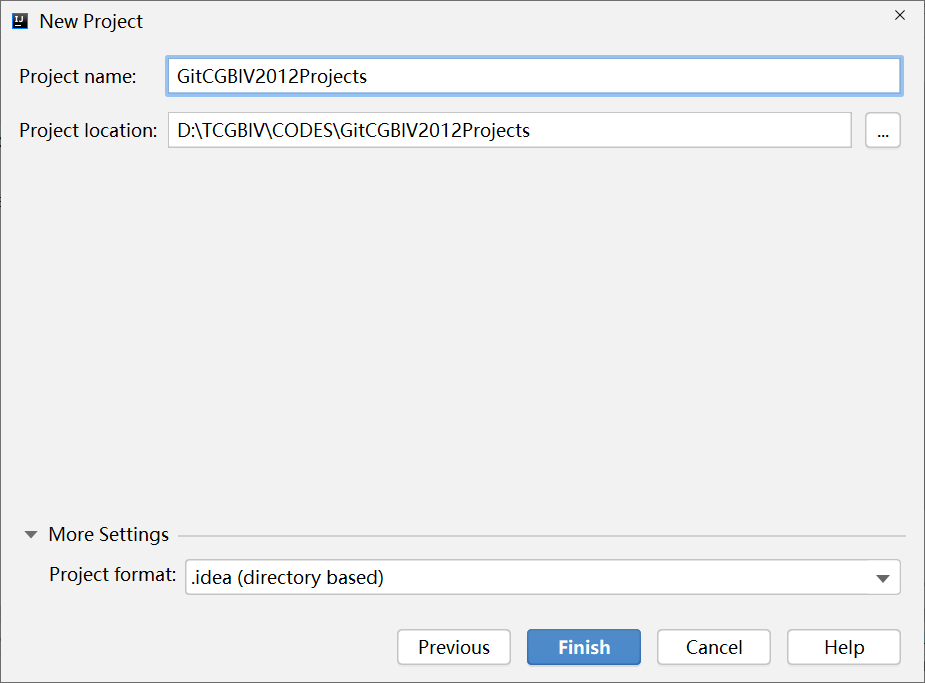


### 创建空项目

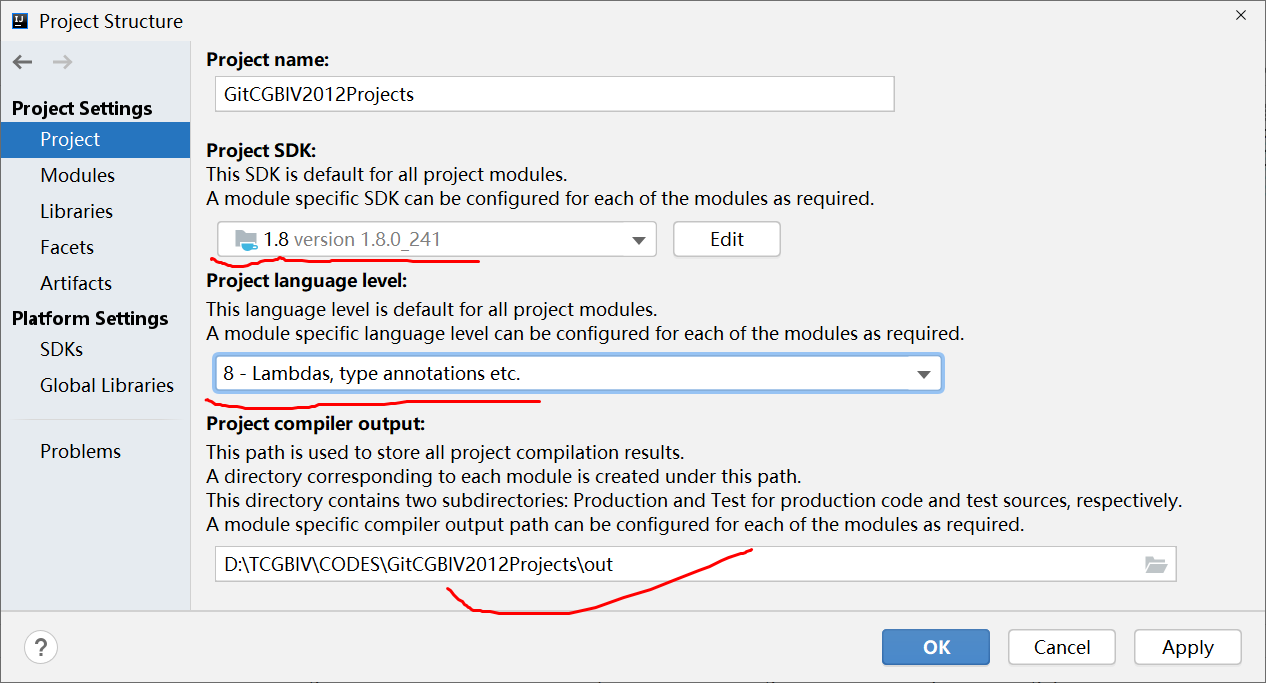
第一步：创建空项目(Empty Project)，如图所示：



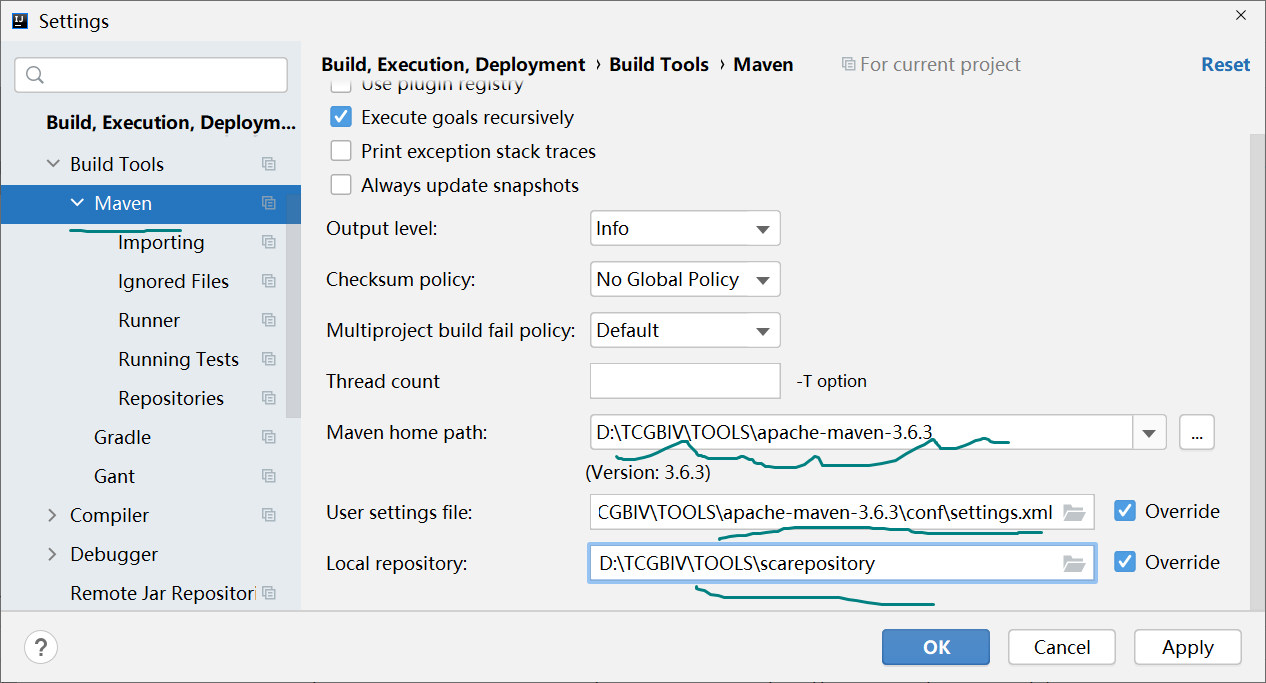
第二步：填写项目信息，如图所示：



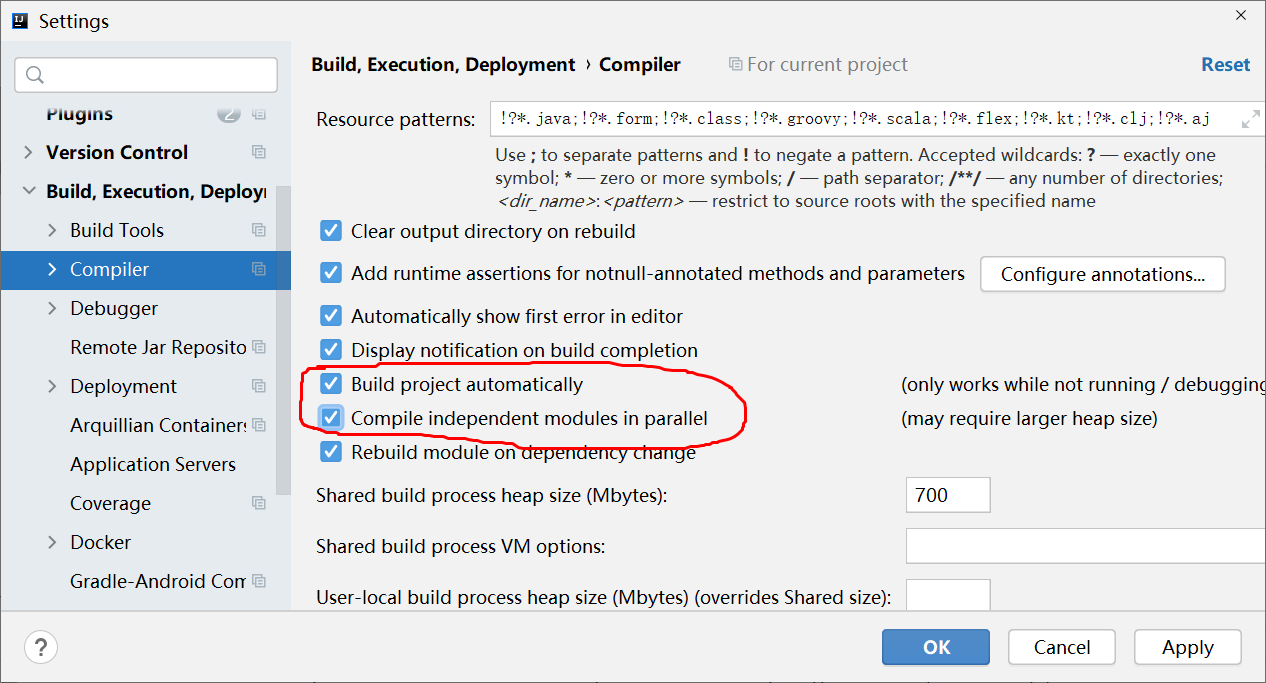
第三步：配置项目的JDK环境，如图所示：



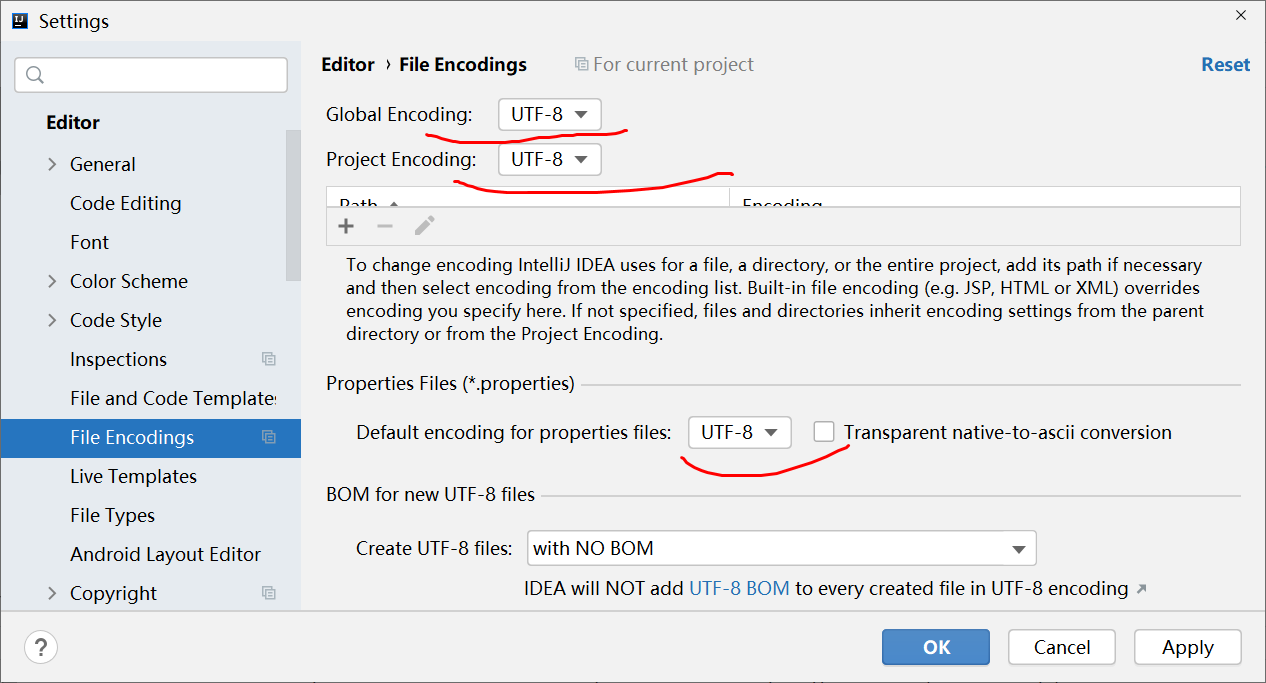
第四步：配置项目所需要的maven环境，如图所示：



第五步：配置项目自动编译，如图所示：

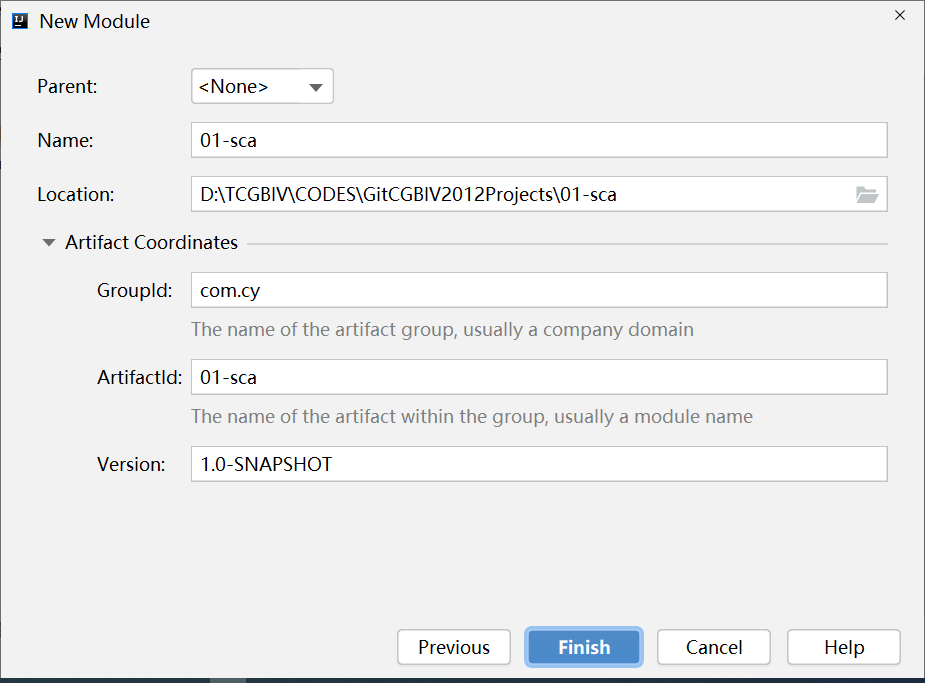


第六步：配置项目编码，如图所示：



### 创建Maven父工程

创建maven module工程01-sca，如图所示：



通过此工程管理其核心依赖，其pom文件内容如下：

*<?*xml version="1.0" encoding="UTF-8"*?>*<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.2.RELEASE</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.cy</groupId>  
 <artifactId>00-sca </artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>pom</packaging>  
 <properties>  
 <java.version>1.8</java.version>  
 <project.build.sourceEncoding>UTF-8

</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8

</project.reporting.outputEncoding>

<spring.cloud-version>Hoxton.SR8</spring.cloud-version>  
 <spring.cloud.alibaba-version>

2.2.3.RELEASE

</spring.cloud.alibaba-version>  
 </properties>

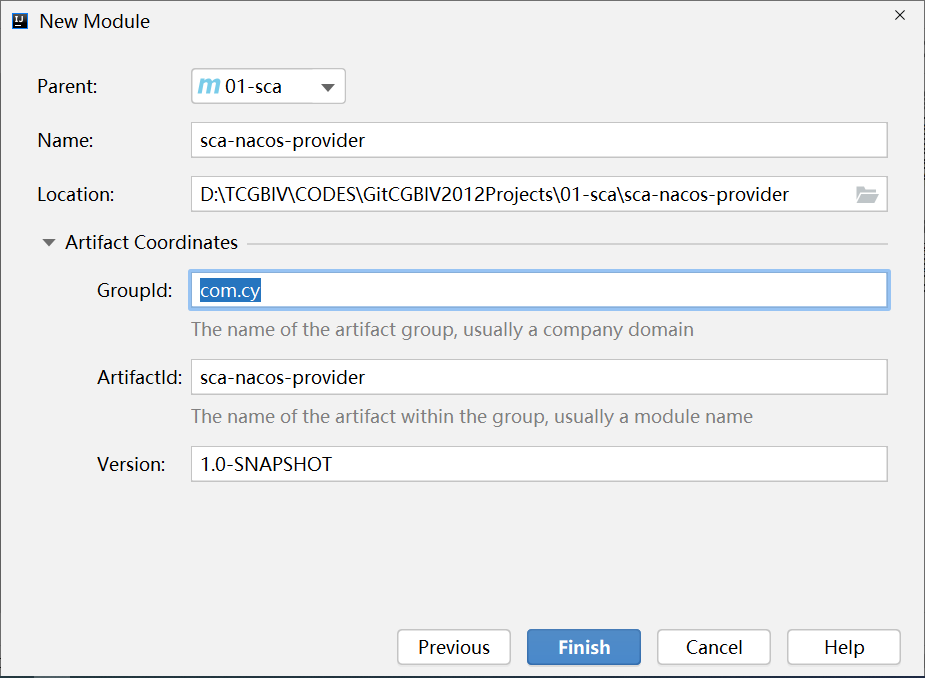
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring.cloud-version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.alibaba.cloud</groupId>  
 <artifactId>spring-cloud-alibaba-dependencies</artifactId>  
 <version>${spring.cloud.alibaba-version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
  
</project>

其中的版本可参考如下网址：

<https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E>

### 生产者服务创建及注册

第一步：创建服务提供者继承parent工程，module名为sca-nacos-provider。



第二步：添加项目依赖，关键代码如下：

<parent>  
 <groupId>com.cy</groupId>  
 <artifactId>01-sca </artifactId>  
 <version>1.0-SNAPSHOT</version>  
</parent>

<dependencies>  
 <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>  
   
 </dependency>  
</dependencies>

第三步：修改配置文件application.yml，实现服务注册，关键代码如下：

server:  
 port: 8081  
spring:  
 application:  
 name: nacos-provider  
 cloud:  
 nacos:  
 server-addr: localhost:8848

注意：服务名不要使用下划线(“\_”),应使用横杠(“-”),这是规则。

第四步：创建启动类，并定义处理请求的控制层对象和方法，关键代码如下：

package com.cy;  
  
@SpringBootApplication  
public class NacosProviderApplication {

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

@Value("${server.port}")

private String server;

@RestController  
 public class ProviderController {  
 @GetMapping(value = "/provider/echo/{string}")  
 public String doEcho(@PathVariable String string) {  
 return server+"say:Hello Nacos Discovery " + string;  
 }  
 }

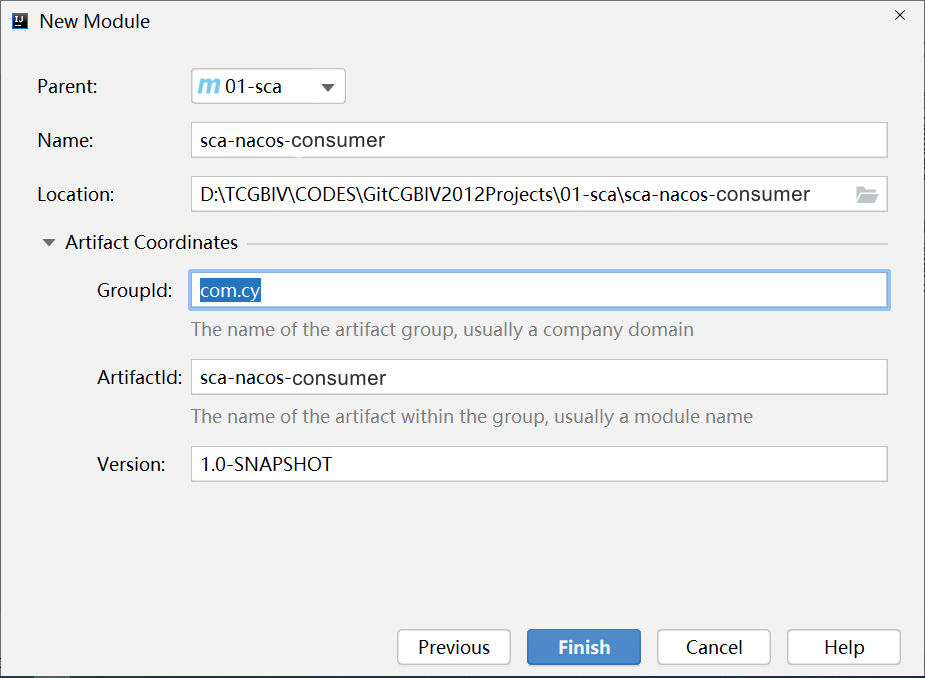
}

第五步：启动启动类，然后刷先nacos服务，检测是否服务注册成功，如图所示：



### 消费者服务发现及调用

第一步：创建服务消费者继承parent工程，module名为sca-nacos-consumer。



第二步：添加项目依赖，关键代码如下：

<parent>  
 <groupId>com.cy</groupId>  
 <artifactId>01-sca</artifactId>  
 <version>1.0-SNAPSHOT</version>  
</parent>

<dependencies>  
 <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>  
 </dependency>  
</dependencies>

第三步：修改配置文件application.yml，关键代码如下：

server:  
 port: 8090  
spring:  
 application:  
 name: nacos-consumer  
 cloud:  
 nacos:  
 discovery:  
 server-addr: localhost:8848 *#从哪里去查找服务*

第四步：创建启动类并实现服务消费，关键代码如下：

package com.cy;  
@SpringBootApplication  
public class NacosConsumerApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(NacosConsumerApplication.class,args);  
 }  
 @Bean  
 public RestTemplate restTemplate(){  
 return new RestTemplate();  
 }  
 @RestController  
 public class ConsumerController{  
  
 @Value("${spring.application.name}")  
 private String appName;  
 @Autowired  
 private LoadBalancerClient loadBalancerClient;  
 @Autowired  
 private RestTemplate restTemplate;  
 @GetMapping("/consumer/doRestEcho")  
 public String doRestEcho(){  
 ServiceInstance serviceInstance =

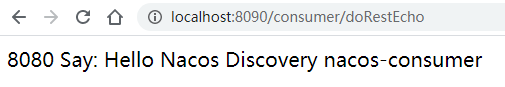
loadBalancerClient.choose("nacos-provider");  
 String url = String.*format*("http://%s:%s/provider/echo/%s",

serviceInstance.getHost(),

serviceInstance.getPort(),

appName);  
 System.*out*.println("request url:"+url);  
 return restTemplate.getForObject(url, String.class);  
 }  
 }  
}

第五步：启动消费者服务，并在浏览器输入<http://localhost:8090/doRestEcho>地址进行访问，假如访问成功会出现，如图所示效果：



### 小节面试分析

* 我们的服务实例是如何注册到Nacos服务的？(基于HTTP请求)
* 在Nacos中服务提供者是如何向注册中心(Registry)续约的？(5秒心跳)
* 对于Nacos服务来讲它是如何判定服务实例的状态？(检测心跳包，15,30)
* 服务启动时如何找到服务启动注册配置类?(NacosNamingService)
* 服务消费方是如何调用服务提供方的服务的？(RestTemplate+

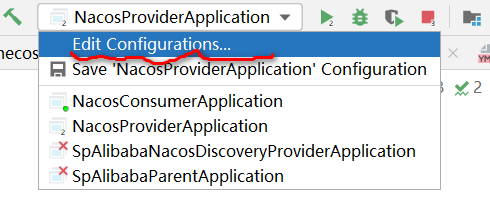
LoadBalancerClient)

* 当nacos注册中心关闭了，服务消费方还能调用到服务吗？(可以，本地缓存)
* 服务消费方是如何拿到nacos注册中心的服务提供者实例的？(先pull，假如没有pull到，检查本地缓存->nacos推送)
* Nacos注册中心基于什么协议进行服务提供者的实例信息推送？(UDP协议)
* Nacos 是如何实现其高可用的?(重试，心跳，缓存、集群)

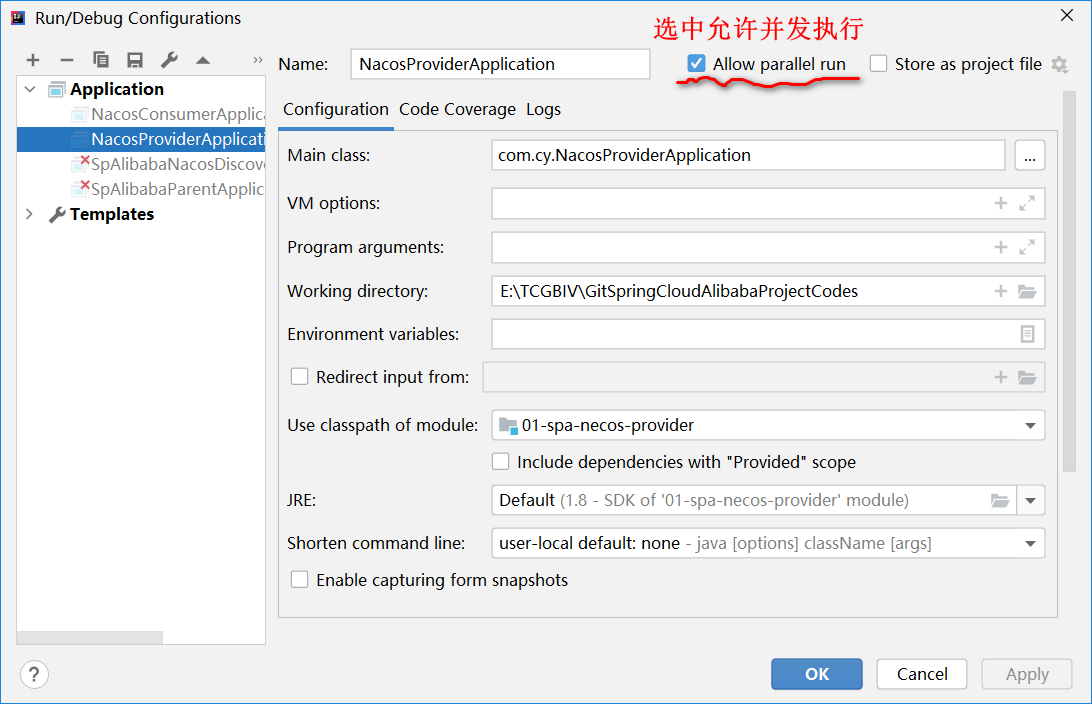
## 服务负载均衡设计及实现

### 入门案例

第一步：打开服务启动配置，如图所示：



第二步：修改并发运行选项，如图所示：



第三步：修改nacos-provider的配置文件端口，分别以8080，8081端口方式进行启动。

server:  
 port: 8081  
spring:  
 application:  
 name: nacos-provider  
 cloud:  
 nacos:  
 server-addr: localhost:8848

启动成功以后，访问nacos的服务列表，检测服务是否成功注册，如图所示：



第四步：启动nacos-consumer项目模块，打开浏览器，输入如下网址进行反复服务访问：

<http://localhost:8090/consumer/doRestEcho>

然后会发现nacos-provider的两个服务都可以处理nacos-consumer的请求。

这里多个实例并发提供服务的方式为负载均衡，这里的负载均衡实现默认是因为Nacos集成了Ribbon来实现的，Ribbon配合RestTemplate，可以非常容易的实现服务之间的访问。Ribbon是Spring Cloud核心组件之一，它提供的最重要的功能就是客户端的负载均衡(客户端可以采用一定算法,例如轮询访问，访问服务端实例信息)，这个功能可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡方式的服务调用。

### @LoadBalanced

当使用RestTemplate进行远程服务调用时，假如需要负载均衡可以在RestTemplate对象构建时，使用@LoadBalanced对构建RestTemplate的方法进行修饰，例如：

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

@Autowired  
private RestTemplate loadBalancedRestTemplate;

接下来，可以在对应的服务端调用方的方法内，基于RestTemplate借助服务名进行服务调用， 例如：

String url="http://nacos-provider/provider/echo/"+applicationName; return loadBalancedRestTemplate.getForObject(url, String.class);

RestTemplate在发送请求的时候会被LoadBalancerInterceptor拦截，它的作用就是用于RestTemplate的负载均衡，LoadBalancerInterceptor将负载均衡的核心逻辑交给了loadBalancer，核心代码如下所示

public ClientHttpResponse intercept(final HttpRequest request,

final byte[] body,

final ClientHttpRequestExecution execution)

throws IOException {

final URI originalUri = request.getURI();

String serviceName = originalUri.getHost();

return this.loadBalancer.execute(serviceName,

requestFactory.createRequest(request, body, execution));

}

@LoadBalanced注解是属于Spring，而不是Ribbon的，Spring在初始化容器的时候，如果检测到Bean被@LoadBalanced注解，Spring会为其设置LoadBalancerInterceptor的拦截器。

### Ribbon负载均衡策略

基于Ribbon方式的负载均衡，Netflix默认提供了七种负载均衡策略，对于SpringCloud Alibaba解决方案中又提供了NacosRule策略，默认的负载均衡策略是轮训策略。如图所示：



当系统提供的负载均衡策略不能满足我们需求时，我们还可以基于IRule接口自己定义策略.

### 小节面试分析

* Ribbon 是什么？(Netflix公司提供的一个负载均衡客户端)
* Ribbon 可以解决什么问题? (基于负载均衡策略进行服务调用)
* Ribbon 内置的负载策略都有哪些?(8种)
* @LoadBalanced的作用是什么？(告诉Spring框架，再使用RestTempalte进行服务调用时，要启动负载均衡策略。)
* 我们可以自己定义负载均衡策略吗？(可以，基于IRule接口进行策略定义)

## RestTemplate 应用分析

### 概述

RestTemplate 是从 Spring3.0 开始支持的一个 HTTP 请求工具，它提供了常见的REST请求方案的模版，例如 GET 请求、POST 请求、PUT 请求、DELETE 请求等。RestTemplate 继承自 InterceptingHttpAccessor 并且实现了 RestOperations 接口，其中 RestOperations 接口定义了基本的 RESTful 操作，这些操作在 RestTemplate 中都得到了具体实现。

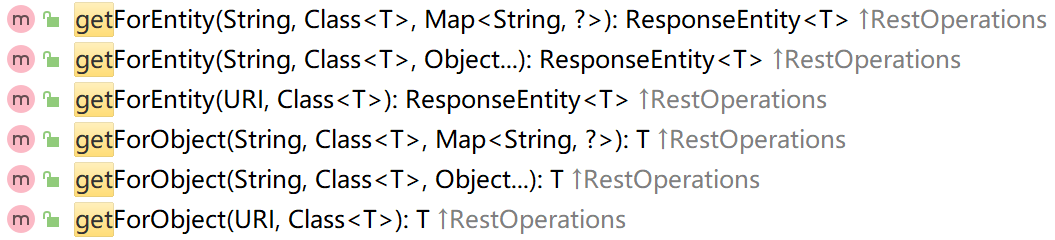
在微服务接口设计时，大部分接口都满足 RESTful 风格，使用 RestTemplate 则可以非常方便地发送 RESTful 风格的请求。

### 相关方法

RestTemplate 可以看到有很多方法，我们可以提取出主要的几种方法是：

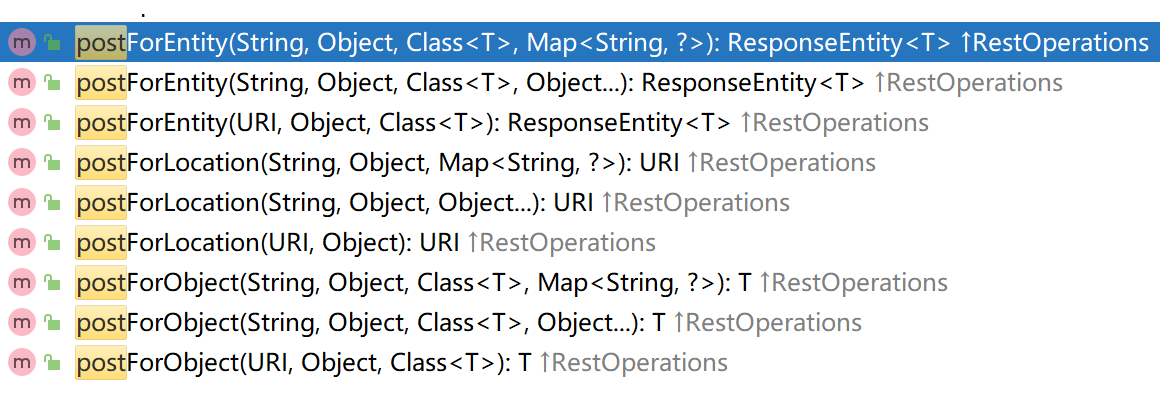
* GET
* POST
* PUT
* DELETE
* HEAD
* OPTIONS
* EXCHANGE
* EXECUTE
* Get方法应用

常见Get方法如图所示（在idea中可以ctrl+f12进行查看）：



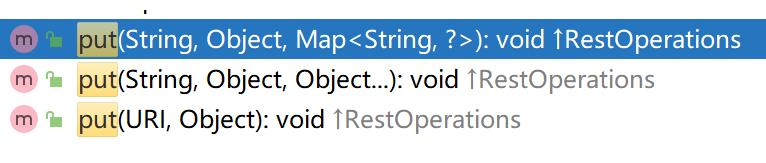
* Post方法应用

常见post方法如图所示：



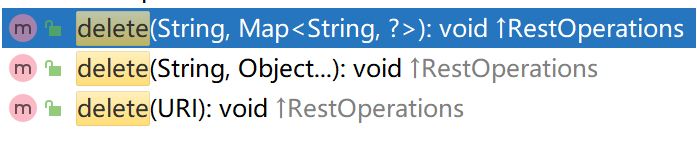
* Put方法应用

常见put方法如图所示：



* Delete方法应用

常见Delete方法如图所示：



### 应用案例分析

通过案例分析RestTemplate类中相关方法的应用。

第一步：服务提供方法类及方法的定义，代码如下：

package com.cy;  
import java.util.Map;  
  
*/\*\*通过此案例演示rest请求的处理\*/*@RestController  
@RequestMapping("/provider/template/")  
public class RestProviderController {  
 @Value("${server.port}")  
 private String server;  
  
 @DeleteMapping("{id}")  
 public void doDelete(@PathVariable Integer id){  
 System.*out*.println(id+" is delete by "+server);  
 *// throw new RuntimeException("delete error");* }  
  
 @PostMapping  
 public Map<String,Object> doPost(

@RequestBody Map<String,Object> map){  
 System.*out*.println("consumer post data: "+map);  
 map.put("status", 1);  
 map.put("server.port", server);  
 return map;  
 }  
  
 @PutMapping  
 public void doPut(@RequestBody Map<String,Object> map){  
 System.*out*.println("consumer put data: "+map);  
 }  
}

第二步：服务消费方类及方法定义，关键代码如下：

package com.cy;  
  
import java.util.Map;  
  
@RestController  
@RequestMapping("/consumer/template/")  
public class RestConsumerController {  
 @Autowired  
 private RestTemplate loadBalancedRestTemplate;  
  
 @DeleteMapping("{id}")  
 public String doDelete(@PathVariable Integer id){  
 String url=String.*format*("http://%s/provider/template/%s",  
 "nacos-provider",id);  
 loadBalancedRestTemplate.delete(url);  
 return "delete ok";  
 }

@PostMapping  
 public Map<String,Object> doPost(

@RequestBody Map<String,Object> map){  
 *//定义服务提供方的url* String url=String.*format*("http://%s/provider/template/",

"nacos-provider");  
 return loadBalancedRestTemplate.postForObject(url,  
 map,*//这里map表示要提交到服务提供方的数据* Map.class*//这个类型通常对应响应结果类型* );  
 }  
 @PutMapping  
 public String doPut(@RequestBody Map<String,Object> map){  
 *//定义服务提供方的url* String url=String.*format*("http://%s/provider/template/",

"nacos-provider");  
 loadBalancedRestTemplate.put(url, map);*//这里map表示要提交到服务提供方的数据* return "put ok";  
 }  
}

第三步：启动服务和postman进行访问测试。

### 小节面试分析

* RestTemplate 有什么用？(远程服务调用，跨进程调用)
* RestTemplate 有负载均衡功能吗？(没有)
* RestTemplate 常用方法有哪些？(put,get,delete,post,…)
* RestTemplate 底层应用协议?(Http协议)

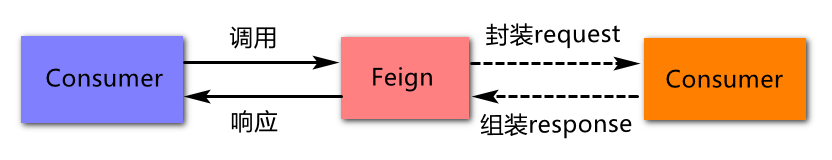
## 基于Feign的远程服务调用

### 背景分析

服务消费方基于rest的请求服务提供方的服务时，一种直接的方式就是自己拼接url，拼接参数然后实现服务调用，但每次服务调用都需要这样拼接，打码量复杂且不易维护，此时Feign诞生。

### Feign是什么

Feign 是一种声明式Web服务客户端，底层封装了对Rest技术的应用,通过Feign可以简化服务消费方对远程服务提供方法的调用实现。如图所示：



Feign 最早是由 Netflix 公司进行维护的，后来 Netflix 不再对其进行维护，最终 Feign 由社区进行维护，更名为 OpenFeign。

### Feign应用实践

第一步：在服务消费方，添加项目依赖(SpringCloud团队基于OpenFeign研发了starter)，代码如下：

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

第二步：在启动类上添加@EnableFeignClients注解，代码如下：

@EnableFeignClients  
@SpringBootApplication  
public class NacosConsumerApplication {…}

第三步：定义Http请求API，基于此API借助OpenFeign访问远端服务，代码如下：

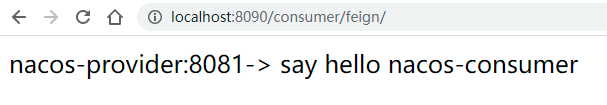
@FeignClient(name="nacos-provider")//nacos-providers为服务提供者名称  
@RestController  
interface ConsumerHttpApi{  
 @GetMapping("/provider/echo/{string}")  
 public String echoMessage(@PathVariable("string") String string);  
}

其中，@FeignClient描述的接口底层会为其创建实现类。

第四步：在FeignConsumerController中添加feign访问，代码如下：

@RestController  
@RequestMapping("/consumer/feign/")  
public class FeignConsumerController {  
  
 @Value("${spring.application.name}")  
 private String appName;  
  
 @Autowired  
 private ConsumerHttpApi consumerHttpApi;  
 */\*\*基于feign方式的服务调用\*/* @GetMapping  
 public String doFeignEcho(){  
 return consumerHttpApi.echoMessage(appName);  
 }  
}

第五步：启动消费者服务，在浏览器中直接通过feign客户端进行访问，如图所示（反复刷新检测其响应结果）：



### Feign 调用过程分析

Feign应用过程分析（底层逻辑先了解）：

1. 通过 @EnableFeignCleints 注解启动 Feign Starter 组件。
2. Feign Starter 在项目启动过程中注册全局配置，扫描包下所有的 @FeignClient 接口类，并进行注册 IOC 容器。
3. @FeignClient 接口类被注入时，通过 FactoryBean#getObject 返回动态代理对象。
4. 接口被调用时被动态代理类逻辑拦截，将 @FeignClient 请求信息通过编码器生成 Request对象
5. 请求对象经Ribbon进行负载均衡，挑选出一个健康的 Server 实例。
6. 通过 Client 携带 Request 调用远端服务返回请求响应。
7. 通过解码器生成 Response 返回客户端，将信息流解析成为接口返回数据。

### Feign 配置增强分析

feign.client.config.default.read-timeout=1  
feign.client.config.default.connect-timeout=1  
feign.client.config.nacos-provider.read-timeout=1

### 小节面试分析

* 为什么使用feign?(基于Feign可以更加友好的实现服务调用，简化服务消费方对服务提供方方法的调用)。
* @FeignClient注解的作用是什么？
* Feign方式调用底层负载均衡是如何实现的？

## 小节总结(Summary)

### 重难点分析

* Nacos是什么，提供了什么特性？
* Nacos在windows环境下安装及初步配置？
* Nacos服务注册与消费的基本过程？
* Nacos服务负载均衡逻辑及设计实现？

### FAQ分析

* 注册中心的核心数据是什么？（服务的名字和它对应的网络地址）
* 注册中心中心核心数据的存取为什么会采用读写锁?(安全和性能)
* 注册中心的一致性选型方案?(非对等leader模型，对等模型)
* Nacos健康检查的方式？(临时实例使用客户端上报，持久实例使用服务端反向探测)
* Nacos是如何保证高可用的? 客户端重试, 一致性协议 distro, 本地缓存文件 Failover 机制, 心跳同步服务, 集群部署模式。

### Bug分析

* ……

# 限流熔断Sentinel解决方案

## 背景分析

在我们日常生活中，经常会在淘宝、天猫、京东、拼多多等平台上参与商品的秒杀、抢购以及一些优惠活动，也会在节假日使用12306 手机APP抢火车票、高铁票，甚至有时候还要帮助同事、朋友为他们家小孩拉投票、刷票，这些场景都无一例外的会引起服务器流量的暴涨，导致网页无法显示、APP反应慢、功能无法正常运转，甚至会引起整个网站的崩溃。

我们如何在这些业务流量变化无常的情况下，保证各种业务安全运营，系统在任何情况下都不会崩溃呢？我们可以在系统负载过高时，采用限流、降级和熔断，三种措施来保护系统，由此一些流量控制中间件诞生。例如Sentinel。

## Sentinel简介

Sentinel (分布式系统的流量防卫兵) 是阿里开源的一套用于服务容错的综合性解决方案。它以流量为切入点, 从流量控制、熔断降级、系统负载保护等多个维度来保护服务的稳定性。

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。

Sentinel 提供了实时的监控功能，通过控制台可以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。

Sentinel 提供了开箱即用特性，可以快速实现与它开源框架整合, 例如与 Spring

Cloud、Dubbo、gRPC 的整合。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核心分为两个部分:

* 核心库（Java 客户端）：能够运行于所有 Java 运行时环境，同时对Dubbo /Spring Cloud 等框架也有较好的支持。
* 控制台（Dashboard）：基于 Spring Boot 开发，打包后可以直接运行。

## 安装Sentinel服务

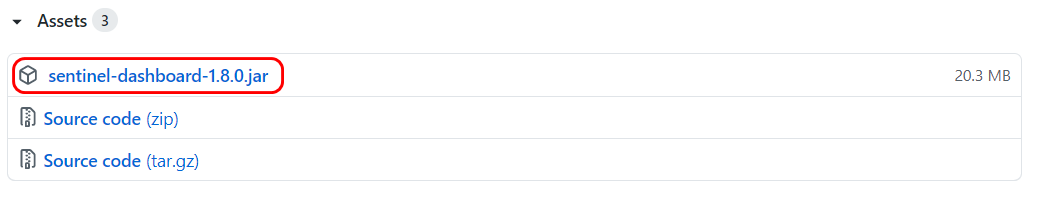
### 下载并启动服务

Sentinel 提供一个轻量级的控制台, 它提供机器发现、单机资源实时监控以及规则管理等功能，其控制台安装步骤如下：

第一步：打开sentinel下载网址

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

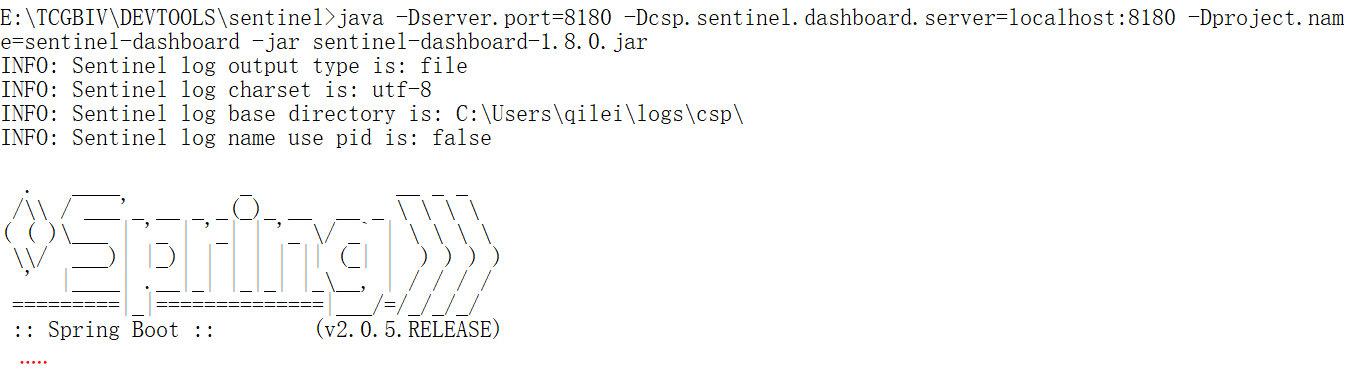
第二步：下载Jar包（可以存储到一个sentinel目录），如图所示：



第三步：在sentinel对应目录，打开命令行(cmd),启动运行sentinel

java -Dserver.port=8180 -Dcsp.sentinel.dashboard.server=localhost:8180 -Dproject.name=sentinel-dashboard -jar sentinel-dashboard-1.8.0.jar

检测启动过程，如图所示：

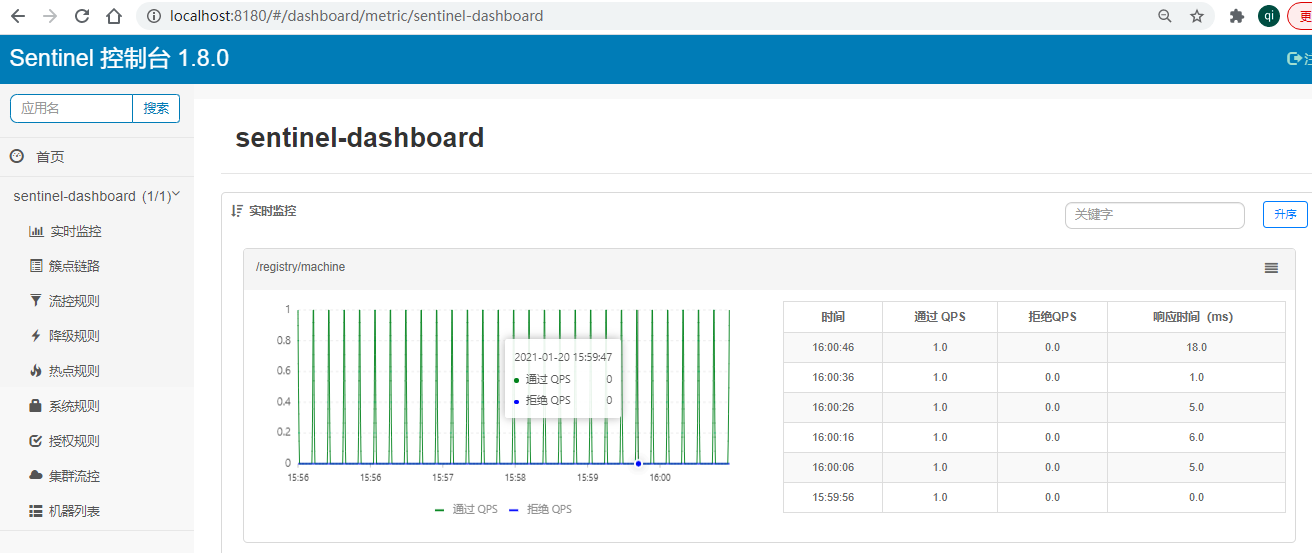


### 访问Sentinal服务

第一步：假如Sentinal启动ok，通过浏览器进行访问测试，如图所示：



第二步：登陆sentinel,默认用户和密码都是sentiel,登陆成功以后的界面如图所示：



## Sentinel限流入门

### 概述

我们系统中的数据库连接池，线程池，nginx的瞬时并发，MQ消息等在使用时都会跟定一个限定的值，这本身就是一种限流的设计。限流的目的防止恶意请求流量、恶意攻击，或者防止流量超过系统峰值。

### Sentinel集成

第一步：Sentinel 应用于服务消费方(Consumer)，在消费方添加依赖如下：

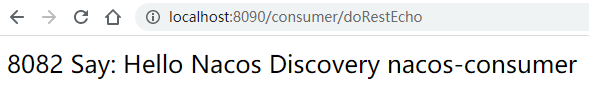
<dependency>  
 <groupId>com.alibaba.cloud</groupId>  
 <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>  
</dependency>

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

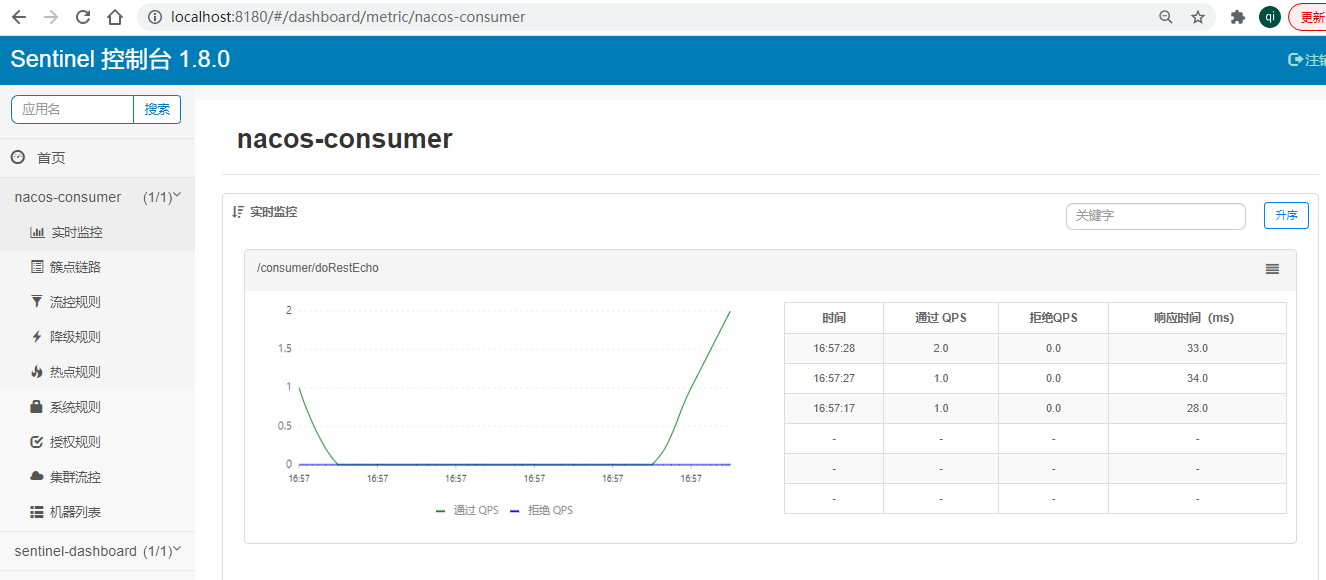
第二步：打开服务消费方配置文件，添加sentinel配置，代码如下：

spring:  
 cloud:  
 sentinel:  
 transport:  
 port: 8099 #跟sentinel控制台交流的端口,随意指定一个未使用的端口即可  
 dashboard: localhost:8180 # 指定sentinel控制台地址。

第三步：启动服务提供者，服务消费者，然后在浏览器访问消费者url，如图所示：



第四步：刷新sentinel 控制台，检测服务列表，如图所示：

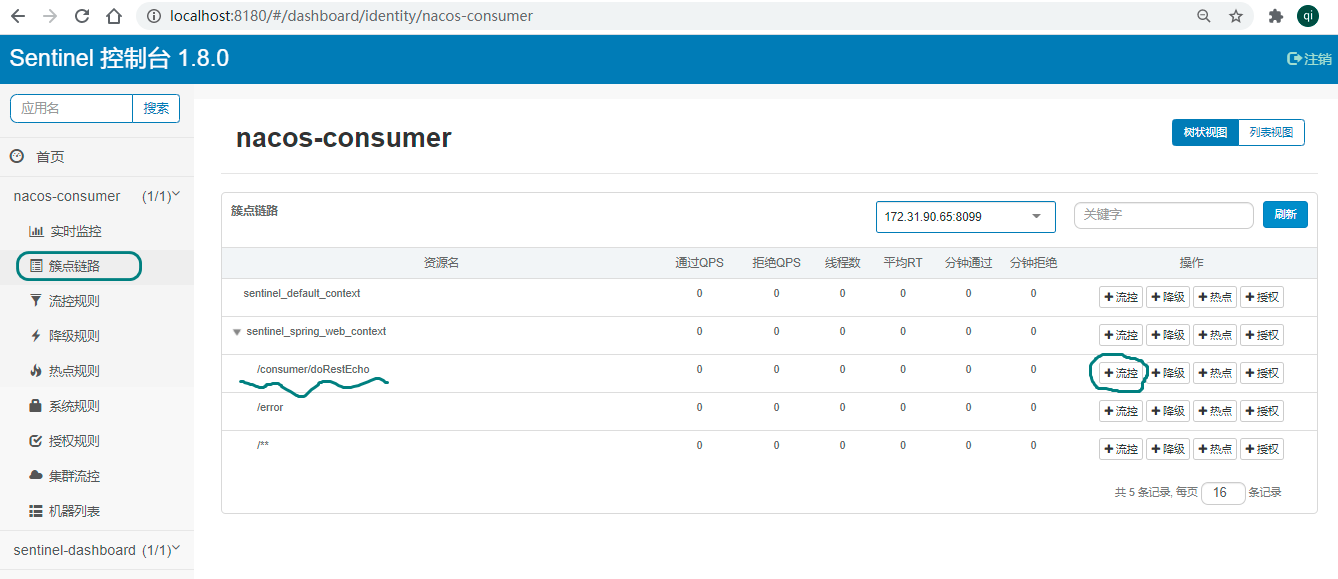


Sentinel的控制台其实就是一个SpringBoot编写的程序，我们需要将我们的服务注册到控制台上，即在微服务中指定控制台的地址，并且还要在消费端开启一个与sentinel控制台传递数据端的端口，控制台可以通过此端口调用微服务中的监控程序来获取各种信息。

### Sentinel限流入门

我们设置一下指定接口的流控，QPS单机阀值为1，代表每秒请求不能超出1，要不然就做流控处理，处理方式直接调用失败。

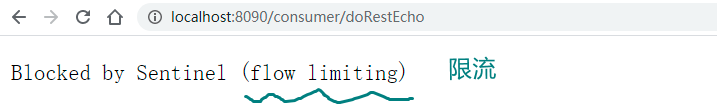
第一步：选择要限流的链路，如图所示：



第二步：设置限流策略，如图所示：



第三步：反复刷新访问消费端端服务，检测是否有限流信息输出，如图所示：



## Sentinel流控规则分析

### 阈值类型分析

* **QPS(Queries Per Second)**：当调用这个api的时，QPS达到单机阈值时，就会限流。
* 线程数：当调用这个api的时，线程数达到单机阈值时，就会限流。

### 设置限流模式

Sentinel的流控模式代表的流控的方式，默认【直接】，还有关联，链路。

* **直接模式：**

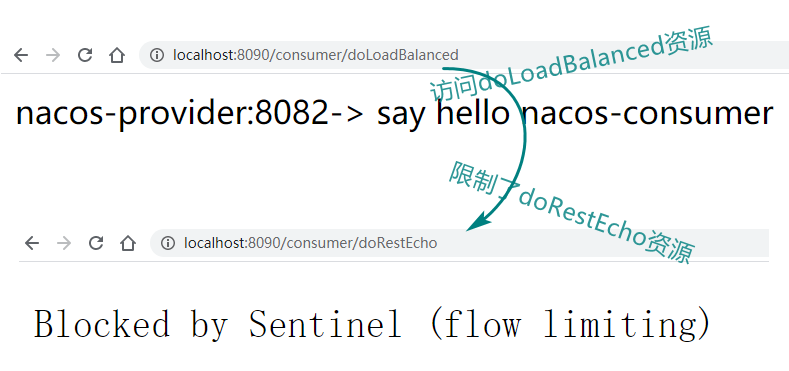
Sentinel默认的流控处理就是【直接->快速失败】。



* **关联模式：**

当关联的资源达到阈值，就限流自己。例如设置了关联资源为/ur2时，假如关联资源/url2的qps阀值超过1时，就限流/url1接口（是不是感觉很霸道，关联资源达到阀值，是本资源接口被限流了）。这种关联模式有什么应用场景呢？我们举个例子，订单服务中会有2个重要的接口，一个是读取订单信息接口，一个是写入订单信息接口。在高并发业务场景中，两个接口都会占用资源，如果读取接口访问过大，就会影响写入接口的性能。业务中如果我们希望写入订单比较重要，要优先考虑写入订单接口。那就可以利用关联模式；在关联资源上面设置写入接口，资源名设置读取接口就行了；这样就起到了优先写入，一旦写入请求多，就限制读的请求。例如：





* **链路模式**

只记录链路入口的流量。也就是当多个服务对指定资源调用时，假如流量超出了指定阈值，则进行限流。被调用的方法用@SentinelResource进行注解，然后分别用不同业务方法对此业务进行调用，假如A业务设置了链路模式的限流在B业务不受影响。例如现在设计一个业务对象，代码如下：

@Service  
public class ConsumerService{  
 @SentinelResource("doGetResource")  
 public String doGetResource(){  
 return "doGetResource";  
 }  
}

接下来我们分别在/consumer/doRestEcho，/consumer/doLoadBalanced对应的方法中对ConsumerService中的doGetResource方法进行调用。其路由规则配置如下：





说明，流控模式为链路模式时，假如是sentinel 1.7.2以后版本，Sentinel Web过滤器默认收敛所有URL的入口上下文，因此互连限流不生效，需要在application.yml添加如下语句来关闭URL PATH聚合：

sentinel:web-context-unify: false

### 设计限流效果

* **快速失败**

流量达到指定阀值，直接返回报异常。（类似路前方坍塌，后面设定路标，让后面的车辆返回）

* **WarmUp (预热)**

WarmUp也叫预热，根据codeFactor（默认3）的值，（阀值/codeFactor）为初始阈值，经过预热时长，才到达设置的QPS的阈值，假如单机阈值为100，系统初始化的阈为 100/3 ，即阈值为33，然后过了10秒，阈值才恢复到100。这个预热的应用场景，如：秒杀系统在开启的瞬间，会有很多流量上来，很有可能把系统打死，预热方式就是把为了保护系统，可慢慢的把流量放进来，慢慢的把阈值增长到设置的阈值。例如：



* **排队等待**

从字面上面就能够猜到，匀速排队，让请求以均匀的速度通过，阈值类型必须设成QPS，否则无效。比如有时候系统在某一个时刻会出现大流量，之后流量就恢复稳定，可以采用这种排队模式，大流量来时可以让流量请求先排队，等恢复了在慢慢进行处理，例如：



### 小节面试分析

* Sentinel是什么？(阿里推出一个流量控制平台，防卫兵)
* 类似Sentinel的产品你知道有什么？(hystrix)
* 你了解哪些限流算法？(计数器、令牌桶、漏斗算法，滑动窗口算法，…)
* Sentinel 默认的限流算法是什么？(滑动窗口算法)
* 你了解sentinel中的阈值应用类型吗?（两种-QPS,线程数）
* Sentinel 限流规则中默认有哪些限流模式?(直连，关联，链路)
* Sentinel的限流效果有哪些？(快速失败，预热，排队)
* Sentinel 为什么可以对我们的业务进行限流，原理是什么？

我们在访问web应用时，在web引用内部会有一个拦截器，这个拦截器会对请求的url进行拦截，拦截到请求以后，读取sentinel 控制台推送到web应用的流控规则，基于流控规则对流量进行限流操作。

* Sentinel如何针对热点参数进行限流？(作业)

## Sentinel降级入门

### 概述

除了流量控制以外，对调用链路中不稳定的资源进行熔断降级也是保障高可用的重要措施之一。由于调用关系的复杂性，如果调用链路中的某个资源不稳定，最终会导致请求发生堆积。

Sentinel 熔断降级会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断（默认行为是抛出 DegradeException）。

### Sentinel降级入门

第一步：服务启动后，选择要降级的链路，如图所示：

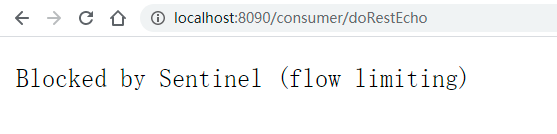


第二步：选择要降级的链路，如图所示：

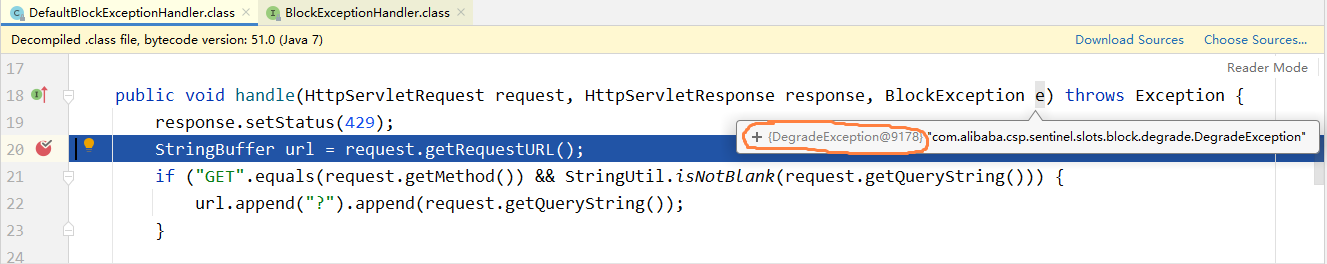


这里表示熔断策略为慢调用比例，表示链路请求数超过3时，假如平均响应时间假如超过200毫秒的有50%，则对请求进行熔断，熔断时长为10秒钟，10秒以后恢复正常。

第三步：对指定链路进行刷新，多次访问测试，假如出现了降级熔断，会出现如下结果：



我们也可以进行断点调试，在DefaultBlockExceptionHandler中的handle方法内部加断点，分析异常类型，假如异常类型为DegradeException则为降级熔断。



### 小节面试分析

* 何为降级熔断？（让外部应用停止对服务的访问）
* 为什么要进行熔断呢？(一般服务不稳定，例如平均响应速度越来越慢或经常出现异常，这样可能会导致调用链堆积，最终系统崩溃)
* Sentinel中限流，降级的异常父类是谁？(BlockException)
* Sentinel 出现降级熔断时，系统底层抛出的异常是谁？(DegradeException)
* Sentinel中异常处理接口是谁？（BlockExceptionHandler）
* Sentinel中异常处理接口下默认的实现类为? (DefaultBlockExceptionHandler)
* 假如Sentinel中默认的异常处理规则不满足我们的需求怎么办?(自己定义)
* 我们如何自己定义Sentinel中异常处理呢？(

直接或间接实现BlockExceptionHandler接口，并将对象交给spring管理)

@Component

public class ServiceBlockExceptionHandler

implements BlockExceptionHandler {

@Override

public void handle(HttpServletRequest request,

HttpServletResponse response,

BlockException e) throws Exception {

//response.setStatus(601);

//设置响应数据的编码

response.setCharacterEncoding("utf-8");

//告诉客户端要响应的数据类型以及客户端以什么编码呈现数据

response.setContentType("text/html;charset=utf-8");

PrintWriter pw=response.getWriter();

Map<String,Object> map=new HashMap<>();

if(e instanceof DegradeException){//降级、熔断

map.put("status",601);

map.put("message", "服务被熔断了!");

}else if(e instanceof FlowException){

map.put("status",602);

map.put("message", "服务被限流了!");

}else{

map.put("status",603);

map.put("message",

"Blocked by Sentinel (flow limiting)");

}

//将map对象转换为json格式字符串

String jsonStr=new ObjectMapper().writeValueAsString(map);

pw.println(jsonStr);

pw.flush();

}

}

## Sentinel降级策略分析

Sentinel熔断降级支持慢调用比例、异常比例、异常数三种策略。

### 慢调用比例

慢调用指耗时大于阈值RT（Response Time）的请求称为慢调用，阈值RT由用户设置。其属性具体含义说明如下：



慢调用逻辑中的状态分析如下：

* **熔断（OPEN）：**请求数大于最小请求数并且慢调用的比率大于比例阈值则发生熔断，熔断时长为用户自定义设置。
* **探测（HALFOPEN）：**当熔断过了定义的熔断时长，状态由熔断（OPEN）变为探测（HALFOPEN）。
* **关闭（CLOSED）：**如果接下来的一个请求小于最大RT，说明慢调用已经恢复，结束熔断，状态由探测（HALF\_OPEN）变更为关闭（CLOSED），如果接下来的一个请求大于最大RT，说明慢调用未恢复，继续熔断，熔断时长保持一致

注意：Sentinel默认统计的RT上限是4900ms，超出此阈值的都会算作4900ms，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

### 异常比例

当资源的每秒请求数大于等于最小请求数，并且异常总数占通过量的比例超过比例阈值时，资源进入降级状态。其属性说明如下：



异常比例中的状态分析如下：

* **熔断（OPEN）：**当请求数大于最小请求并且异常比例大于设置的阈值时触发熔断，熔断时长由用户设置。
* **探测（HALFOPEN）：**当超过熔断时长时，由熔断（OPEN）转为探测（HALFOPEN）
* **关闭（CLOSED）：**如果接下来的一个请求未发生错误，说明应用恢复，结束熔断，状态由探测（HALF\_OPEN）变更为关闭（CLOSED）**。**如果接下来的一个请求继续发生错误，说明应用未恢复，继续熔断，熔断时长保持一致。

### 异常数量

当资源近1分钟的异常数目超过阈值（异常数）之后会进行服务降级。注意，由于统计时间窗口是分钟级别的，若熔断时长小于60s，则结束熔断状态后仍可能再次进入熔断状态。其属性说明如下：



基于异常数的状态分析如下：

* **熔断（OPEN）：**当请求数大于最小请求并且异常数量大于设置的阈值时触发熔断，熔断时长由用户设置。
* **探测（HALFOPEN）：**当超过熔断时长时，由熔断（OPEN）转为探测（HALFOPEN）
* **关闭（CLOSED）：**如果接下来的一个请求未发生错误，说明应用恢复，结束熔断，状态由探测（HALF\_OPEN）变更为关闭（CLOSED）如果接下来的一个请求继续发生错误，说明应用未恢复，继续熔断，熔断时长保持一致。

### 小节面试分析

* Sentinel 降级熔断策略有哪些？（慢调用，异常比例，异常数）
* Sentinel 熔断处理逻辑中的有哪些状态？(Open，HalfOpen,Closed)
* Sentinel 对服务调用进行熔断以后处于什么状态？(熔断打开状态-Open)
* Sentinel 设置的熔断时长到期以后，Sentinel的熔断会处于什么状态？(探测-HalfOpen,假如再次访问时依旧响应时间比较长或依旧有异常，则继续熔断)
* Sentinel 中的熔断逻辑恢复正常调用以后，会出现什么状态?(熔断关闭-closed)

## Sentinel热点规则分析

### 概述

何为热点？热点即经常访问的数据。很多时候我们希望统计某个热点数据中访问频次最高的 Top N 数据，并对其访问进行限制。比如：

* 商品 ID 为参数，统计一段时间内最常购买的商品 ID 并进行限制。
* 用户 ID 为参数，针对一段时间内频繁访问的用户 ID 进行限制。

热点参数限流会统计传入参数中的热点数据，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。其中，Sentinel会利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。

### 快速入门

第一步：定义热点业务代码，如图所示：

@GetMapping("/consumer/findById")  
@SentinelResource("doFindById")  
public String doFindById(@RequestParam("id") Integer id){  
 return "hot id is "+id;  
}

第二步：服务启动后，选择要限流的热点链路，如图所示：

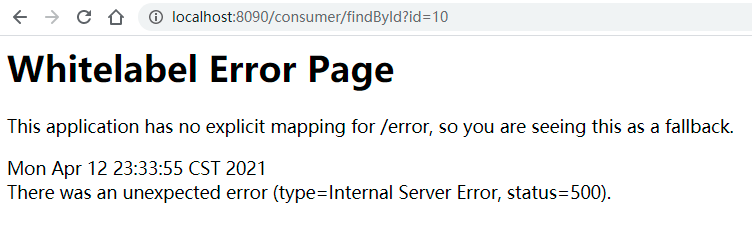


第三步：设置要限流的热点，如图所示：



热点规则的限流模式只有QPS模式（这才叫热点）。参数索引为@SentinelResource注解的方法参数下标，0代表第一个参数，1代表第二个参数。单机阈值以及统计窗口时长表示在此窗口时间超过阈值就限流。

第四步：多次访问热点参数方法，前端会出现如下界面，如图所示：



然后，在后台出现如下异常表示限流成功。

com.alibaba.csp.sentinel.slots.block.flow.param.ParamFlowException: 2

其中，热点参数其实说白了就是特殊的流控，流控设置是针对整个请求的；但是热点参数他可以设置到具体哪个参数，甚至参数针对的值，这样更灵活的进行流控管理。

一般应用在某些特殊资源的特殊处理，如：某些商品流量大，其他商品流量很正常，就可以利用热点参数限流的方案。

### 特定参数设计

配置参数例外项，如图所示：



这里表示参数值为5时阈值为100，其它参数值阈值为1，例如当我们访问<http://ip:port/consumer/doRestEcho?id=5>时的限流阈值为100。

### 小节面试分析

* 如何理解热点数据？(访问频度比较高的数据)
* 热点数据的限流规则是怎样的?(主要是针对参数进行限流设计)
* 热点数据中的特殊参数如何理解？(热点限流中的某个参数值的阈值设计)
* 对于热点数据的访问出现限流以后底层异常是什么？(ParamFlowException)

## Sentinel系统规则

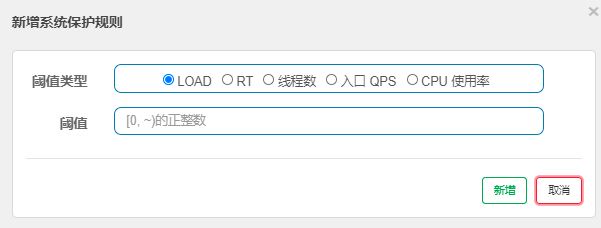
### 概述

系统在生产环境运行过程中，我们经常需要监控服务器的状态，看服务器CPU、内存、IO等的使用率；主要目的就是保证服务器正常的运行，不能被某些应用搞崩溃了；而且在保证稳定的前提下，保持系统的最大吞吐量。

长期以来，系统自适应保护的思路是根据硬指标，即系统的负载 (load1) 来做系统过载保护。当系统负载高于某个阈值，就禁止或者减少流量的进入；当 load 开始好转，则恢复流量的进入。

### 快速入门

Sentinel的系统保护规则是从应用级别的入口流量进行控制，从单台机器的总体 Load、RT、入口 QPS 、线程数和CPU使用率五个维度监控应用数据，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。如图所示：



其中，

* Load（仅对 Linux/Unix-like 机器生效）：当系统 load1 超过阈值，且系统当前的并发线程数超过系统容量时才会触发系统保护。系统容量由系统的 maxQps \* minRt 计算得出。设定参考值一般是 CPU cores \* 2.5。
* CPU使用率：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0）。
* 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。
* 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。

系统保护规则是应用整体维度的，而不是资源维度的，并且仅对入口流量生效。入口流量指的是进入应用的流量（EntryType.IN），比如 Web 服务。

### 小节面试分析

* 如何理解sentinel中的系统规则？(是对所有链路的控制规则,是一种系统保护策略)
* Sentinel的常用系统规则有哪些？(RT,QPS,CPU,线程,Load-linux,unix)
* Sentinel系统保护规则被触发以后底层会抛出什么异常？（SystemBlockException）

## Sentinel授权规则

### 概述

很多时候，我们需要根据调用方来限制资源是否通过，这时候可以使用 Sentinel 的黑白名单控制的功能。黑白名单根据资源的请求来源（origin）限制资源是否通过，若配置白名单则只有请求来源位于白名单内时才可通过；若配置黑名单则请求来源位于黑名单时不通过，其余的请求通过。

### 快速入门



黑白名单规则（AuthorityRule）非常简单，主要有以下配置项：

* resource：资源名，即限流规则的作用对象
* limitApp：对应的黑名单/白名单，不同 origin 用 , 分隔，如 appA,appB
* strategy：限制模式，AUTHORITY\_WHITE 为白名单模式，AUTHORITY\_BLACK 为黑名单模式，默认为白名单模式

我们在定义应用时，可以实现RequestOriginParser接口，在接口方法中解析请求参数来判定放行还是限流；

@Component  
public class DefaultRequestOriginParser implements RequestOriginParser {  
 @Override  
 public String parseOrigin(HttpServletRequest request) {  
 String origin = request.getParameter("origin");  
 return origin;  
 }  
}

当我们配置的流控应用值为echo时，假如规则为白名单，则

<http://ip:port/path?origin=echo>的请求才可以通过。

### 小节面试分析

* 如何理解Sentinel中的授权规则？(对指定资源的访问给出的一种简易的授权策略)
* Sentinel的授权规则是如何设计的？(白名单和黑名单)
* 如何理解Sentinel中的白名单？（允许访问的资源名单）
* 如何理解Sentinel中的黑名单？（不允许访问的资源名单）、
* Sentinel如何识别白名单和黑名单？(在拦截器中通过调用RequestOriginParser对象的方法检测具体的规则)
* 授权规则中RequestOriginParser类的做用是什么？（对流控应用值进行解析，检查服务访问时传入的值是否与RequestOriginParser的parseOrigin方法返回值是否相同。）

## Sentinel规则持久化

### 概述

我们知道，Sentinel Dashboard中配置完规则之后，假如重启应用规则就会丢失，所以实际生产环境中需要配置规则的持久化实现，Sentinel提供多种不同的数据源来持久化规则配置，包括file，redis、nacos、zk(没有界面)。

我们可以通过控制台设置规则后将规则推送到统一的规则中心，客户端监听规则中心实时获取变更。DataSource 扩展常见的实现方式有两种，分别为拉模式和推模式：

* **拉模式：**客户端主动向某个规则管理中心定期轮询拉取规则，这个规则中心可以是 RDBMS、文件，甚至是 VCS 等。这样做的方式是简单，缺点是无法及时获取变更；
* **推模式：**规则中心统一推送，客户端通过注册监听器的方式时刻监听变化，比如使用 Nacos、Zookeeper 等配置中心。这种方式有更好的实时性和一致性保证。

### 快速入门

将sentinel流控的配置信息通过nacos进行持久化，具体步骤如下：

第一步：在sentinel客户端(一个服务)添加依赖，代码如下：

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

第二步：修改服务的配置文件application.yml或application.properties

sentinel:

datasource:

ds:

nacos:

### nacos连接地址

server-addr: localhost:8848

## nacos连接的分组

group-id: DEFAULT\_GROUP

###路由存储规则

rule-type: flow

### 读取配置文件的 data-id

data-id: nacos-consumer

### 读取配置文件类型为json

data-type: json

第三步：在nacos配置中心中配置流控规则，例如：

[

{

"resource": "/consumer/doRestEcho",

"controlBehavior": 0,

"count": 1.0,

"grade": 1,

"limitApp": "default",

"strategy": 0

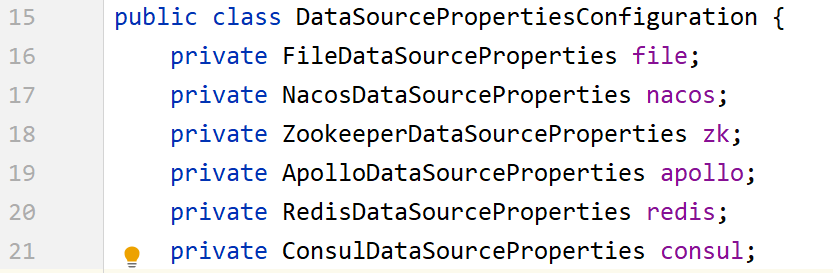
}  
]

其中：

* resource：资源名，即限流规则的作用对象
* limitApp：流控针对的调用来源，若为 default 则不区分调用来源
* grade：限流阈值类型（QPS 或并发线程数）；0代表根据并发数量来限流，1代表根据QPS来进行流量控制
* count：限流阈值
* strategy：调用关系限流策略
* controlBehavior：流量控制效果（快速失败、Warm Up、匀速排队）
* clusterMode：是否为集群模式

### 小节面试分析

* Sentinel 限流熔断、系统保护等规则默认是存储在哪里的？（内存）
* Sentinel 限流熔断等规则是否支持持久化?(支持)
* Sentinel 限流熔断规则为什么要持久化？(防止系统在重启以后规则数据调用)
* Sentinel模式支持的持久化方式有哪些？(file,nacos,zk,redis, Apollo, consul)



* Sentinel持久化的流控信息，在微服务客户端是如何获取的？(服务端推，客户端拉)
* Sentinel持久化的流控信息，在生产环境一般推荐采用什么模式？(推模式，实时性比较好，缺陷是服务端压力会比较大，还有可能会产生消息堆积)
* Sentinel 基于nacos的持久化的实现过程是怎样的？(依赖，数据源配置，nacos新建配置)
* Nacos中存储sentinel流控信息的格式是什么？(JSON)

## 小节总结(Summary)

总之，Sentinel可为秒杀、抢购、抢票、拉票等高并发应用，提供API接口层面的流量限制，让突然暴涨而来的流量用户访问受到统一的管控，使用合理的流量放行规则使得用户都能正常得到服务。

# 配置中心之Nacos应用实践

## 背景分析

我们知道，除了代码之外，软件还有一些配置信息，比如数据库的用户名和密码，还有一些我们不想写死在代码里的东西，例如像线程池大小、队列长度等运行参数，以及日志级别、算法策略等， 还有一些是软件运行环境的参数，如Java 的内存大小，应用启动的参数，包括操作系统的一些 参数配置…… 所有这些东西，我们都叫做软件配置。以前，我们把软件配置写在一个配置文件中，就像 Windows 下的 ini 文件，或是 Linux 下的 conf 文件。然而，在分布式系统下，这样的方式就变得非常不好管理，并容易出错。假如生产环境下，项目现在正在运行，此时修改了配置文件，我们需要让这些配置生效，通常的做法是不是要重启服务。但重启是不是会带来系统服务短时间的暂停，从而影响用户体验呢，还有可能会带来经济上的很大损失（例如双11重启下服务）。基于这样的背景，配置中心诞生了。

## 配置中心简介

### 概述

配置中心最基础的功能就是存储一个键值对，用户发布一个配置（configKey），然后客户端获取这个配置项（configValue）；进阶的功能就是当某个配置项发生变更时，不停机就可以动态刷新服务内部的配置项，例如，在生产环境上我们可能把我们的日志级别调整为 error 级别，但是，在系统出问题我们希望对它 debug 的时候，我们需要动态的调整系统的行为的能力，把日志级别调整为 debug 级别。

### 应用场景

当你一个电商系统，设计大促预案一定会考虑，同时涌进来超过一亿人并发访问的时候，假如系统是扛不住的，你会怎么办，在这个过程中我们一般会采用限流，降级。系统的限流和降级本质上来讲就是从日常的运行态切换到大促态的一个行为的动态调整，这个本身天然就是配置起到作用的一个相应的场景。随着大促时间点的临近，这些分布式系统中，每个子系统会有大大小小的预案，这些预案其实都是以配置的形式去存在的。配置中心会在这个时间轴上，定时地安排执行预案，每个系统哪些功能降级，在什么时候降级，什么时候开放那些专门为大促存在的一些功能，在大促之前的时间点，整个应用发布会封盘，被禁止掉了，已经不允许做任何线上发布了，那在这个时候要切换系统的行为，那就是靠配置中心去做这个事情。

现在的大型应用，为了高可用，业务可能部署在 2 个机房，现在一般同城双机房是标配。数据存储，比如像 mysql，在设计上我们为了高可用，可能会配备一主几备，主库是可写的，备库可能是只读的。在生产上可能有几台机器坏了或者甚至一个机房坏了，出问题、故障了。DBA 会根据整个业务系统在机房的部署拓扑，来找到这个坏掉的机房里的所有的主库，来做一个主备库的切换，把备库切成可写。在这个过程中，配置中心的作用呢，就是跟 DBA 的高可用切换工具保持联动，DBA 工具负责数据库切换，产生数据源配置变更，所有应用基于配置中心监听各自的数据源配置变更，当产生主备库切换，配置中心会将数据源配置变更推送到应用，整个过程对应用是透明的，无感知的。应用是不用知道底下机房出问题了，主备库出现切换了。这就是配置中心的容灾设计中的应用。

还有，现在一些大型的互联网系统，CDN 后面会有一个统一接入层，包括 PC 和移动端的流量可能都是从这个统一接入层进入到生产的机房的。在这个过程中，统一接入层负责根据前端用户的属性，去分配用户的流量到后端不同的单元。当一个单元挂了之后，这些流量需要无缝地切到其它的业务单元里面去，这个单元它糅合了机房以及业务域的一些划分，在这个过程中，配置中心要起到一个关键的作用是要在统一接入的机器上，要让它们对于全局的流量规则达成一个分布式共识，这个实际上是分布式一致性的一个要求。

### 问题与挑战

在单体应用时代，或者说在集中式应用开发时代，我们应用可能就是打成一个包，那在这个包里我们可能提供一些配置文件，当我们的系统上到生产环境之后，如果我们需要修改系统的行为，我们只需要登录到机器，修改一下配置文件，然后 reload 一下，实际上不是什么大的负担。

在微服务架构中，我们有没有可能登陆所有机器一台一台地改配置文件呢？尤其像现在的分布式系统规模越来越大。不同的微服务是由不同的团队，不同的组织去负责开发和维护的，微服务架构给大家承诺的是，每个微服务可以采用不同的技术栈，在这种情况下，我们作为运维人员，作为 Ops，甚至不知道配置文件在哪里，因为配置文件名，配置文件放置的目录，可能是五花八门，所以没有办法去做基于配置文件的管理。大规模的分布式系统可能部署在不同的机房，有各种部署，那当一个配置改变了之后，这个配置什么时候生效的，它有没有生效，有多少机器上的配置变更了，但运行失败了，这些状态，通过配置文件是没办法明确地把这些状态暴露出来的，简单的来说，应用暴露了哪些配置，你依赖的那些三方的服务，其它团队开发的服务，到底暴露了哪些配置参数，可能这个简单的问题通过配置文件管理的时候，我们都没有办法回答。再不用说分布式系统中某一些子系统我们想一致性的改变它们的行为的话，还有比如配置如何容灾，配置文件如果丢失了，我要回滚到某一个历史版本，这个事情怎么做？这些都是采用微服务分布式架构给配置管理带来的挑战。

### 配置中心选型

在面向分布式的微服务系统中，如何通过更高效的配置管理方式，实现微服务系统架构持续“无痛”的演进，并动态调整和控制系统的运行时态，配置中心的选型和设计起着举足轻重的作用。

市场上主流配置中心有Apollo(携程开源)，nacos(阿里开源)，Spring Cloud Config(Spring Cloud 全家桶成员）。我们在对这些配置中心进行选型时重点要从产品功能、使用体验、实施过程和性能等方面进行综合考量。

### 小节面试分析

* 什么是配置中心？（存储项目配置信息的一个服务）
* 为什么要使用配置中心？(集中管理配置信息，动态发布配置信息)
* 市场上有哪些主流的配置中心？(Apollo,nacos,…..)

## Nacos配置快速入门

### 创建项目

创建maven项目module,名字为sca-nacos-config，其pom.xml文件内容如下：

*<?*xml version="1.0" encoding="UTF-8"*?>*<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.cy</groupId>  
 <artifactId>sca-nacos-config</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <parent>  
 <groupId>com.cy</groupId>  
 <artifactId>01-sca</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <dependencies>  
 <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>  
 </dependencies>  
</project>

### 创建配置文件

在resource目录下创建bootstrap.yml配置文件（启动优先级最高），代码如下：

spring:  
 application:  
 name: nacos-config  
 cloud:  
 nacos:  
 config:  
 server-addr: 127.0.0.1:8848  
 group: DEFAULT\_GROUP *# Group, default is DEFAULT\_GROUP* file-extension: yml *# Configure the data format of the content, default to properties*

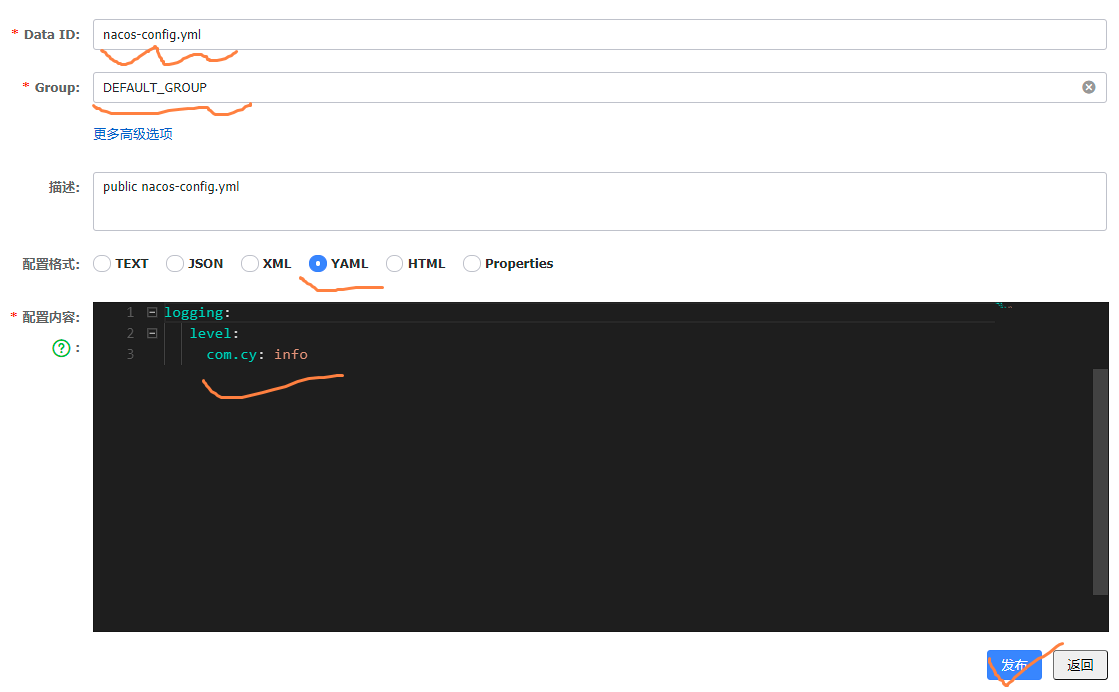
### 启动测试分析

创建启动类，对环境进行启动测试，代码如下：

package com.cy;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class NacosConfigApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(NacosConfigApplication.class,args);  
 }  
}

### Nacos基本配置

打开nacos配置中心，新建配置，如图所示：



其中Data IDs的值要与bootstrap.yml中定义的spring.application.name的值相同(服务名-假如有多个服务一般会创建多个配置实例，不同服务对应不同的配置实例)。

### 创建Controller处理器

创建配置中心Controller，也可以将Controller添加到启动类内部，如图所示：

package com.cy;  
@SpringBootApplication  
public class NacosConfigApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(NacosConfigApplication.class);  
 }

@RefreshScope //支持配置动态刷新  
 @RestController  
 @RequestMapping("/config/")  
 public class NacosConfigController {

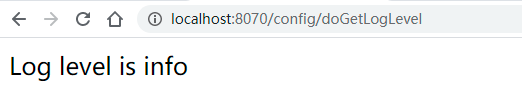
@Value("${logging.level.com.cy:info}")

private String logLevel;  
  
 @RequestMapping("/doGetLogLevel")  
 public String doGetLogLevel() {  
 return "Log level is "+logLevel;

}  
 }  
}

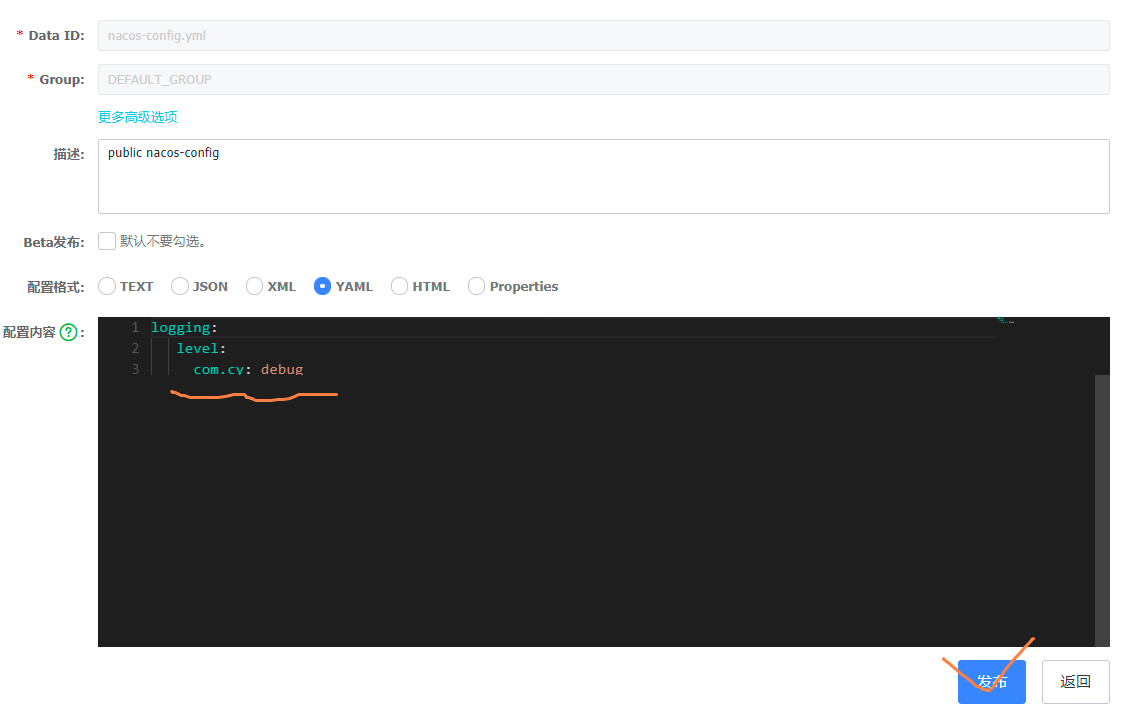
其中,@RefreshScope的作用是，在配置中心的相关配置发生变化以后，能够及时看到更新

Controller编写好以后，启动配置中心服务，然后进行访问测试。，打开浏览器直接在地址栏输入<http://localhost:8070/config/doGetLogLevel>，检测输出结果是否为我们配置中配置的信息，如图所示。

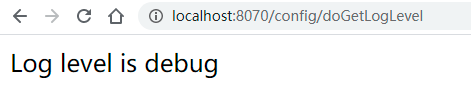


### Nacos配置动态更新实现

修改Nacos的日志级别配置并重新重新发布，如图所示：



刷新浏览器url，检测其配置输出。



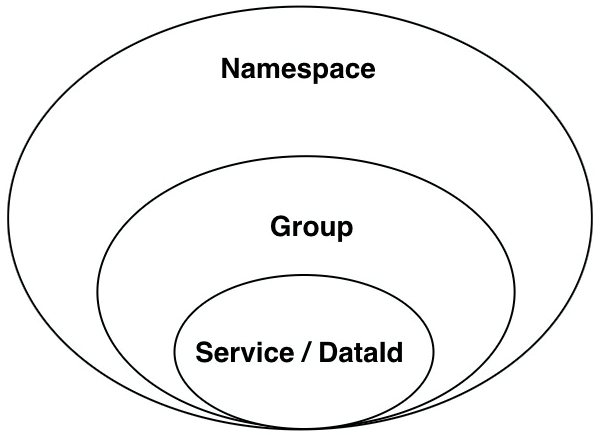
### 小节面试分析

* 配置中心一般都会配置什么内容？(可能会经常变化的配置信息，例如连接池，日志、线程池、限流熔断规则)
* 什么信息一般不会写到配置中心?(服务端口，服务名，服务的注册地址，配置中心)
* 项目中为什么要定义bootstrap.yml文件？(定义优先访问的配置信息)
* 微服务应用中我们的客户端如何获取配置中心的信息?(可以基于客户端轮询的方式)
* 微服务应用中客户端如何感知配置中心数据变化？(@RefreshScope)

## Nacos配置管理模型

### 概述

Nacos 配置管理模型由三部分构成，如图所示：



其中：

* Namespace：命名空间，对不同的环境进⾏隔离，⽐如隔离开发环境和⽣产环境。
* Group：分组，将若⼲个服务或者若⼲个配置集归为⼀组。
* Service/DataId：某⼀个服务或配置集，一般对应一个配置文件。

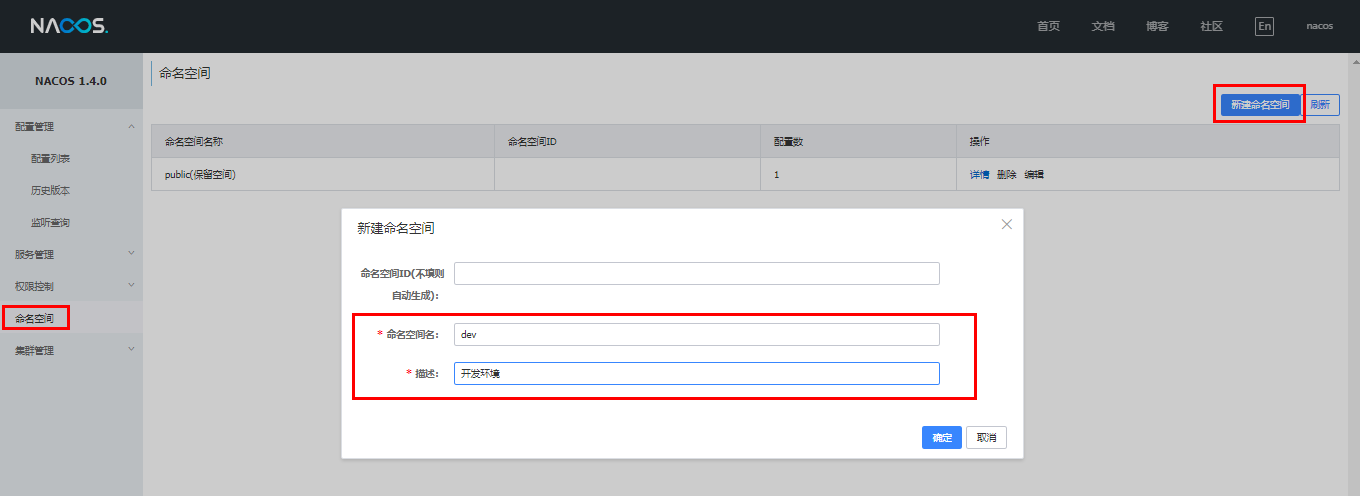
### 命名空间设计

Nacos中的命名空间一般用于配置隔离，这种命名空间的定义一般会按照环境（开发，生产等环境）进行设计和实现.我们默认创建的配置都存储到了public命名空间，如图所示：

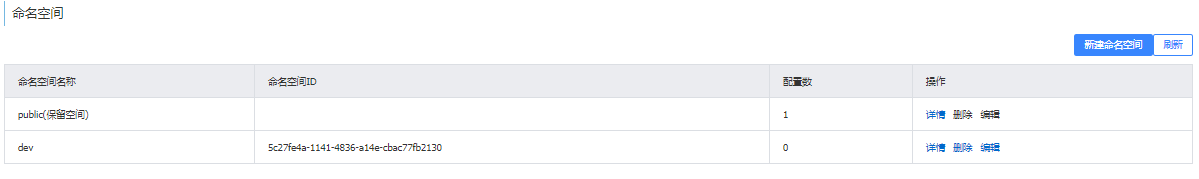


创建新的开发环境并定义其配置，然后从开发环境的配置中读取配置信息，该如何实现呢？

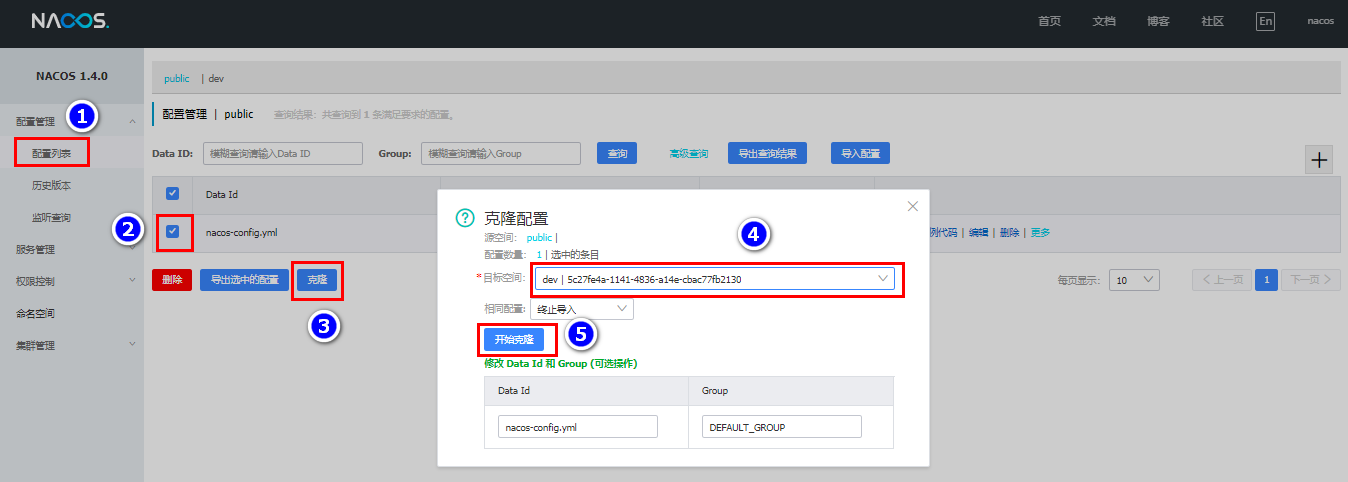
第一步：创建新命名空间，如图所示：



命名空间成功创建以后，会在如下列表进行呈现。



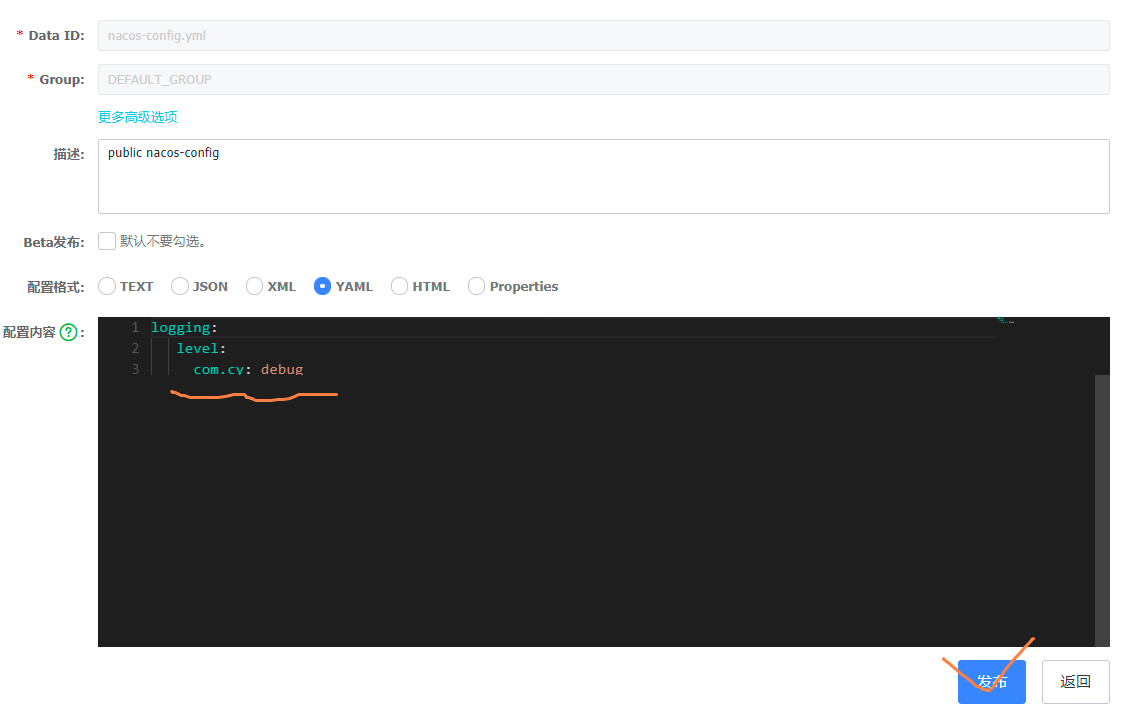
在指定命名空间下添加配置，也可以直接取配置列表中克隆，例如：



克隆成功以后，我们会发现在指定的命名空间中有了我们克隆的配置，如图所示：



此时我们修改dev命名空间中Data Id的nacos-config配置，如图所示：



修改项目module中的配置文件bootstrap.yml,添加如下配置，关键代码如下：

spring:

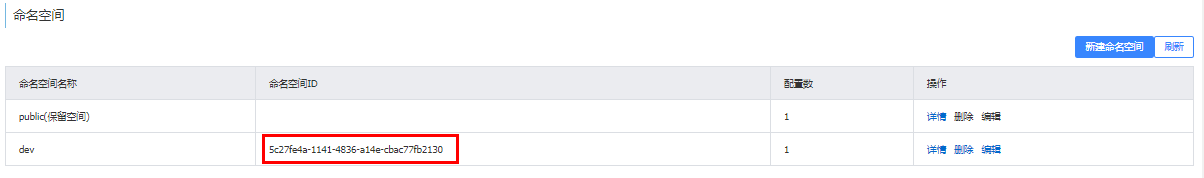
cloud:

nacos:

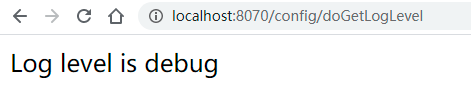
config:

namespace: 5c27fe4a-1141-4836-a14e-cbac77fb2130

其中，namespace后面的字符串为命名空间的id，可直接从命名空间列表中进行拷贝，如图所示：



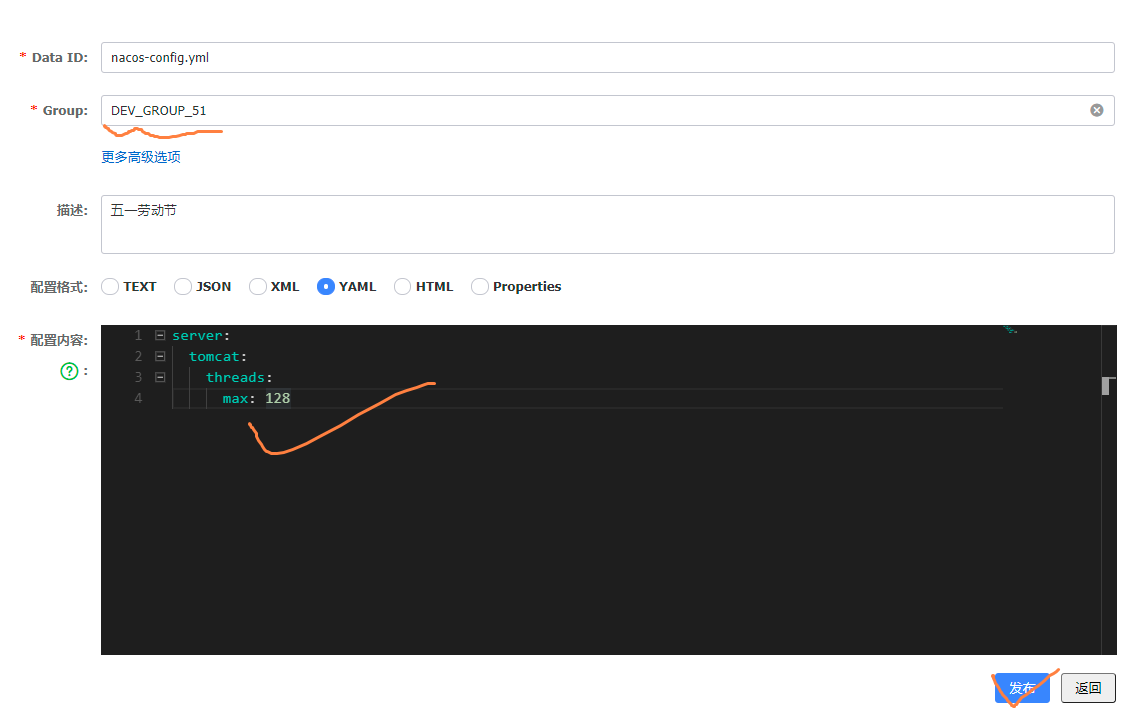
重启服务，继续刷新<http://localhost:8070/config/doGetLogLevel>地址。检测输出，看看输出的内容是什么，是否为dev命名空间下配置的内容，如图所示：



我们还可以创建生产环境，依次类推进行设计和实现即可。

### 分组设计及实现

当我们在指定命名空间下，按环境或服务做好了配置以后，有时还需要基于服务做分组配置，例如，一个服务在不同时间节点(节假日，活动等)切换不同的配置，可以在新建配置时指定分组名称，如图所示：



配置发布以后，修改boostrap.yml配置类，在其内部指定我们刚刚创建的分组，代码如下：

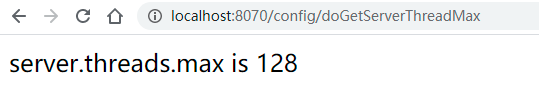
server:  
 port: 8070  
spring:  
 application:  
 name: nacos-config  
 cloud:  
 nacos:  
 config:  
 server-addr: 127.0.0.1:8848  
 group: DEV\_GROUP\_51 *# Group, default is DEFAULT\_GROUP* file-extension: yml *# Configure the data format of the content, default to properties* namespace: 5c27fe4a-1141-4836-a14e-cbac77fb2130

在NacosConfigController类中添加属性和方法用于获取和输出DEV\_GROUP\_51配置中设置的线程数，代码如下：

@Value("${server.tomcat.threads.max:200}")  
private Integer serverThreadMax;

@RequestMapping("/doGetServerThreadMax")  
public String doGetserverThreadMax(){  
 return "server.threads.max is "+serverThreadMax;  
}

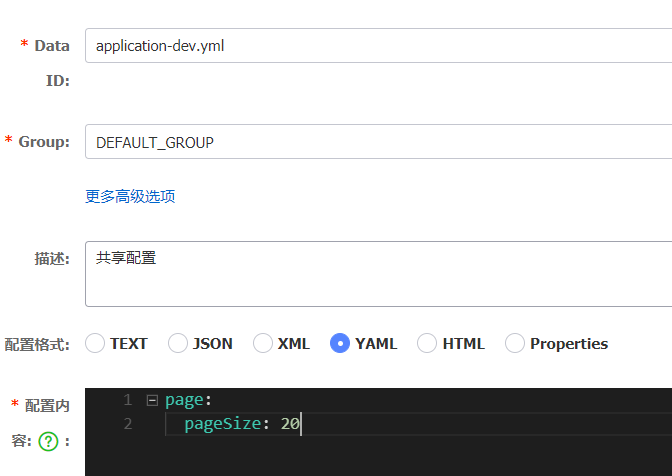
然后重启服务，进行测试，检测内容输出，如图所示:



### 共享配置设计及读取

当同一个namespace的多个配置文件中都有相同配置时，可以对这些配置进行提取，然后存储到nacos配置中心的一个或多个指定配置文件，哪个微服务需要，就在服务的配置中设置读取即可。例如：

第一步：在nacos中创建一个共享配置文件，例如：



第二步：在指定的微服务配置文件(bootstrap.yml)中设置对共享配置文件的读取，例如：

见红色区域内容。

spring:  
 application:  
 name: nacos-config  
 cloud:  
 nacos:  
 config:  
 server-addr: localhost:8848  
 *# 命名空间* namespace: 83ed55a5-1dd9-4b84-a5fe-a734e4a6ec6d  
 *# 分组名  
 # group: DEFAULT\_GROUP  
 # 配置中心文件扩展名* file-extension: yml  
 *# 共享配置* shared-configs[0]:  
 data-id: application-dev.yml  
 group: DEFAULT\_GROUP  
 refresh: true *#默认false*

第三步：在指定的业务类中读取和应用共享配置即可，例如：

@Value("${page.pageSize:10}")  
private Integer pageSize;

### 小节面试分析

* Nacos配置中的管理模型是怎样的？（namespace,group,service/data-id）
* Nacos配置中心关闭，微服务还能都到配置信息吗?(可以，都本地缓存)
* Nacos的客户端是采用什么机制从配置中心获取数据的？(

拉取+namespce+group+dataId)

* Nacos 的服务端为什么不采取配置推送(push)方式，而是客户端pull方式去获取配置信息？(服务端的压力可能会比较大，除非服务端的性能要求非常高)
* Nacos客户端(微服务)是否可以读取共享配置?(可以)

## 小节总结(Summary)

### 重难点分析

* 配置中心的选型。(市场活跃度、稳定性、性能)
* Nacos配置基本实现。(新建，修改、删除配置以后，在Nacos客户端应用配置)
* Nacos配置管理模型应用。(namespace,group,service/dataId)

### FAQ分析

* 为什么需要配置中心？
* 市面上有哪些主流配置中心？
* 配置中心选型时要重点考虑哪些因素？
* Nacos配置中心的配置如何动态刷新？
* Nacos配置管理模型是怎样的？

### Bug分析

* 。。。。

# 网关Gateway 应用分析及实现

## 简介

我们知道，一个大型系统在设计时，经常会被拆分为很多个微服务。那么作为客户端要如何去调用 这么多的微服务呢？如果没有网关的存在，我们只能在客户端记录每个微服务的地址，然后分别去调用。这样的架构，会存在着诸多的问题，例如，客户端请求不同的微服务可能会增加客户端代码或配置的复杂性。还有就是每个服务，在调用时都需要独立认证。

并且存在跨域请求，也在一定程度上提高了代码的复杂度。

基于微服务架构中的设计及实现上的问题，为了在项目中简化前端的调用逻辑，同时也简化内部服务之间互相调用的复杂度，提出了网关的概念，网关本质上要提供一个各种服务访问的入口，并提供服务接收并转发所有内外部的客户端调用，还有就是权限认证，限流控制等等。

Spring Cloud Gateway是Spring公司基于Spring 5.0，Spring Boot 2.0 和 等技术开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API入口，负责服务请求路由、组合及协议转换，并且基于 Filter 链的方式提供了权限认证，监控、限流等功能。

Spring Cloud Gateway优缺点分析：

* 优点：

1. 性能强劲：是第一代网关Zuul的1.6倍。
2. 功能强大：内置了很多实用的功能，例如转发、监控、限流等
3. 设计优雅，容易扩展。

* 缺点：

1. 其实现依赖Netty与WebFlux，不是传统的Servlet编程模型，学习成本高。
2. 需要Spring Boot 2.0及以上的版本，才支持

## 快速入门

### 业务描述

通过网关作为服务访问入口，对系统中的服务进行访问。

### 入门业务实现

第一步：创建sca-gateway模块，其pom.xml文件如下：

*<?*xml version="1.0" encoding="UTF-8"*?>*<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>

<parent>  
 <artifactId>01-sca</artifactId>  
 <groupId>com.cy</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <groupId>org.cy</groupId>  
 <artifactId>sca-gateway</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
 </dependency>  
 </dependencies>  
</project>

第二步：添加相关配置，代码如下：

server:

port: 9000

spring:  
 application:  
 name: sca-gateway  
 cloud:  
 gateway:  
 routes:  
 - id: route01  
 uri: http://localhost:8081/  
 predicates: *###匹配规则* - Path=/nacos/provider/echo/\*\*

filters:

- StripPrefix=1 *#转发之前去掉path中第一层路径，例如nacos*

其中：路由(Route) 是 gateway 中最基本的组件之一，表示一个具体的路由信息载体。主要定义了下面的几个信息:

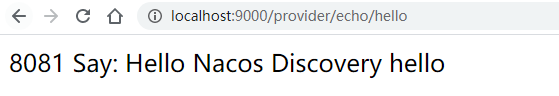
1) id，路由标识符，区别于其他 Route。

2) uri，路由指向的目的地 uri，即客户端请求最终被转发到的微服务。

3) predicate，断言(谓词)的作用是进行条件判断，只有断言都返回真，才会执行路由。

4) filter，过滤器用于修改请求和响应信息。

第三步：启动项目进行访问测试，如图所示：



### 小节面试分析？

* 什么是网关？服务访问的一个入口，类似生活中的“海关“
* 为什么使用网关？(服务安全，统一服务入口管理，负载均衡，限流，鉴权)
* Spring Cloud Gateway 应用的初始构建过程(添加依赖，配置)
* Gateway 服务的启动底层是通过谁去实现的？(Netty)
* Gateway 服务做请求转发时一定要在注册中心进行注册吗？

## 负载均衡设计

### 为什么负载均衡？

网关才是服务访问的入口，所有服务都会在网关层面进行底层映射，所以在访问服务时，要基于服务serivce id（服务名）去查找对应的服务，让请求从网关层进行均衡转发，以平衡服务实例的处理能力。

### Gateway中负载均衡实现？

第一步：项目中添加服务发现依赖，代码如下：

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

第二步：修改其配置文件，代码如下

server:  
 port: 9000  
spring:  
 application:  
 name: sca-gateway  
 cloud:  
 nacos:

discovery:  
 server-addr: localhost:8848  
 gateway:  
 discovery:  
 locator:  
 enabled: true *#开启通过服务注册中心的serviceId创建路由* routes:  
 - id: route01  
 *##uri: http://localhost:8081/* uri: lb://nacos-provider *# lb为服务前缀，不能随意写*  
 predicates: *###匹配规则* - Path=/nacos/provider/echo/\*\*

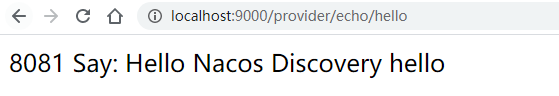
filters:

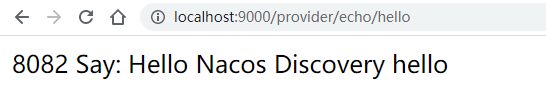
- StripPrefix=1 *#转发之前去掉path中第一层路径，例如nacos*

其中，lb指的是从nacos中按照名称获取微服务,并遵循负载均衡策略。同时建议开发阶段打开gateway日志，代码如下：

logging:  
 level:  
 org.springframework.cloud.gateway: debug

第三步：启动服务，进行访问测试，并反复刷新分析，如图所示：





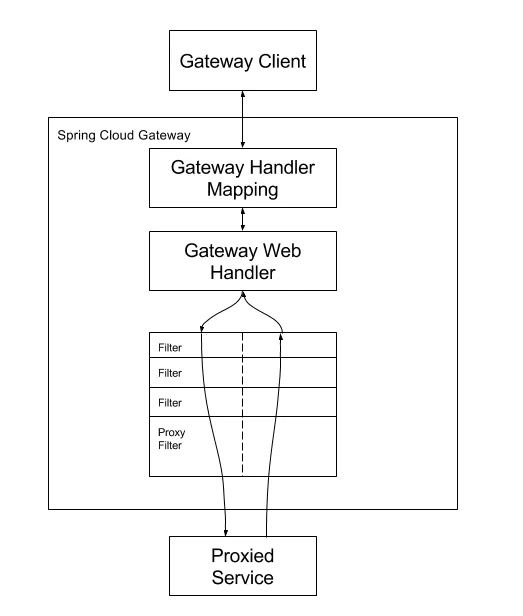
### 小节面试分析？

* 网关层面是如何实现负载均衡的？(通过服务名去查找具体的服务实例)
* 网关层面是如何通过服务名查找服务实例的？(Ribbon)
* 你了解Ribbon中的哪些负载均衡算法?(轮询，权重，hash,…..)

## 执行流程分析

### 流程解析

根据官方的说明，其Gateway具体工作流程，如图所示：



客户端向Spring Cloud Gateway发出请求。 如果Gateway Handler Mapping 通过断言的集合确定请求与路由匹配，则将其发送到Gateway Web Handler。 Gateway Web Handler 通过确定的路由中所配置的过滤器集合链式调用过滤器（也就是所谓的责任链模式）。 Filter由虚线分隔的原因是， Filter可以在发送代理请求之前和之后运行逻辑。处理的逻辑是 在处理请求时 排在前面的过滤器先执行，而处理返回相应的时候，排在后面的过滤器先执行。

### 小节面试分析？

* 网关进行请求转发的流程是怎样，有哪些关键对象？
* 网关层面服务的映射方式怎样的？(断言-path,服务名/服务实例)
* 网关层如何记录服务的映射？(通过map，并要考虑锁的应用)

## 断言增强分析

### Predicate 简介

Predicate(断言)又称谓词，用于条件判断，只有断言结果都为真，才会真正的执行路由。断言其本质就是定义路由转发的条件。

### Predicate 内置工厂

SpringCloud Gateway包括需要内置的断言工厂，所有这些断言都与http请求的不同属性相匹配，具体如下：

基于Datetime类型的断言工厂 此类型的断言根据时间做判断，主要有三个：

1. AfterRoutePredicateFactory：判断请求日期是否晚于指定日期
2. BeforeRoutePredicateFactory：判断请求日期是否早于指定日期
3. BetweenRoutePredicateFactory：判断请求日期是否在指定时间段内

-After=2020-12-31T23:59:59.789+08:00[Asia/Shanghai]

当且仅当请求时的时间After配置的时间时，才转发该请求，若请求时的时间不是After配置的时间时，则会返回404 not found。同时，当predicates配置项只配置了一个Predicate且没有配置Path时，Path的默认值为/\*\*。所以该段配置会使访问 GATEWAY\_URL/\*\* 时转发到 user-center微服务的/\*\*。时间值可通过ZonedDateTime.now()获取。

基于远程地址的断言工厂 RemoteAddrRoutePredicateFactory：接收一个IP地址段，判断请求主机地址是否在指定地址段中，例如：

-RemoteAddr=192.168.1.1/24

基于Cookie的断言工厂，CookieRoutePredicateFactory：接收两个参数，cookie 名字和一个正则表达式。 判断请求 cookie是否具有给定名称且值与正则表达式匹配。例如：

-Cookie=chocolate, ch

基于header的断言工厂，HeaderRoutePredicateFactory：接收两个参数，标题名称和正则表达式。 判断请求Header是否具有给定名称且值与正则表达式匹配。例如：

-Header=X-Request-Id, \d+

基于Host的断言工厂，HostRoutePredicateFactory：接收一个参数，主机名模式。判断请求的Host是否满足匹配规则，例如：

-Host=\*\*.testhost.org

基于Method请求方法的断言工厂，MethodRoutePredicateFactory接收一个参数，判断请求类型是否跟指定的类型匹配。例如：

-Method=GET

基于Path请求路径的断言工厂PathRoutePredicateFactory，接收一个参数，判断请求的URI部分是否满足路径规则，例如：

-Path=/foo/{segment}

基于Query请求参数的断言工厂，QueryRoutePredicateFactory ：接收两个参数，请求param和正则表达式， 判断请求参数是否具 有给定名称且值与正则表达式匹配。例如：

-Query=baz, ba.

基于路由权重的断言工厂，WeightRoutePredicateFactory：接收一个[组名,权重], 然后对于同一个组内的路由按照权重转发，例如：

routes:

-id: weight\_route1

-uri: host1 predicates:

-Path=/ehco/\*\*

-Weight=group2, 2

-id: weight\_route2

-uri: host2 predicates:

-Path=/ehco/\*\*

-Weight= group2, 8

### Predicate 应用案例实践

内置的路由断言工厂应用案例，例如：

server:  
 port: 9000  
spring:  
 application:  
 name: sca-gateway  
 cloud:  
 nacos:  
 server-addr: localhost:8848  
 gateway:  
 discovery:  
 locator:  
 enabled: true *#开启通过服务中心的serviceId 创建路由的功能* routes:  
 - id: bd-id  
 *##uri: http://localhost:8081/* uri: lb://nacos-provider  
 predicates: *###匹配规则* - Path=/nacos/provider/echo/\*\*

- Before=2021-01-30T00:00:00.000+08:00

- Method=GET  
 filters:  
 - StripPrefix=1 *# 转发之前去掉1层路径*

说明：当条件不满足时，则无法进行路由转发，会出现404异常。

### Predicate 自定义分析及实现

业务描述：通过断言设置分页页码page的取值。

业务实现：

第一步：定义断言

package com.cy.predicates;  
  
import org.springframework.cloud.gateway.handler.predicate.AbstractRoutePredicateFactory;  
import org.springframework.stereotype.Component;  
import org.springframework.web.server.ServerWebExchange;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Predicate;  
  
@Component  
class PageRoutePredicateFactory extends AbstractRoutePredicateFactory<PageRoutePredicateFactory.Config> {  
  
 public PageRoutePredicateFactory() {  
 super(PageRoutePredicateFactory.Config.class);  
 }  
 @Override  
 public List<String> shortcutFieldOrder() {  
*//这里的顺序要跟配置文件中的参数顺序一致* return Arrays.*asList*("minPage", "maxPage");  
 }  
  
 @Override  
 public Predicate<ServerWebExchange> apply(PageRoutePredicateFactory.Config config) {  
 return new Predicate<ServerWebExchange>() {  
 @Override  
 public boolean test(ServerWebExchange serverWebExchange) {  
 String page=serverWebExchange.getRequest().getQueryParams().getFirst("Page");  
 if (page!=null&&"".equals(page.trim())) {  
 int page= Integer.*parseInt*(page);  
 return page > config.getMinPage() && age < config.getMaxPage();  
 }  
 return true;  
  
 }  
 };  
 }  
static class Config {//必须是静态内部类  
 private int minPage;  
 private int maxPage;  
  
 public int getMinPage() {  
 return minPage;  
 }  
  
 public void setMinPage(int minPage) {  
 this.minPage = minPage;  
 }  
  
 public int getMaxPage() {  
 return maxPage;  
 }  
  
 public void setMaxPage(int maxPage) {  
 this.maxPage = maxPage;  
 }  
 }  
}

第二步：修改配置文件，添加Page断言设置

server:  
 port: 9000  
spring:  
 application:  
 name: jt-api-gateway  
 cloud:  
 nacos:  
 server-addr: localhost:8848  
 gateway:  
 discovery:  
 locator:  
 enabled: true *#开启通过服务中心的serviceId 创建路由的功能* routes:  
 - id: bd-id  
 *##uri: http://localhost:8081/* uri: lb://nacos-provider  
 predicates: *###匹配规则* - Path=/nacos/provider/echo/\*\*  
 - Before=2021-01-20T00:00:00.000+08:00  
 - Method=GET  
 - Page=10,20  
 filters:  
 - StripPrefix=1 *# 转发之前去掉1层路径*

第四步，重启Gateway服务，在浏览器输入url及参数，检测输出结果:

<http://localhost:9000/nacos/provider/echo/hello?Page=15>

### 小节面试分析

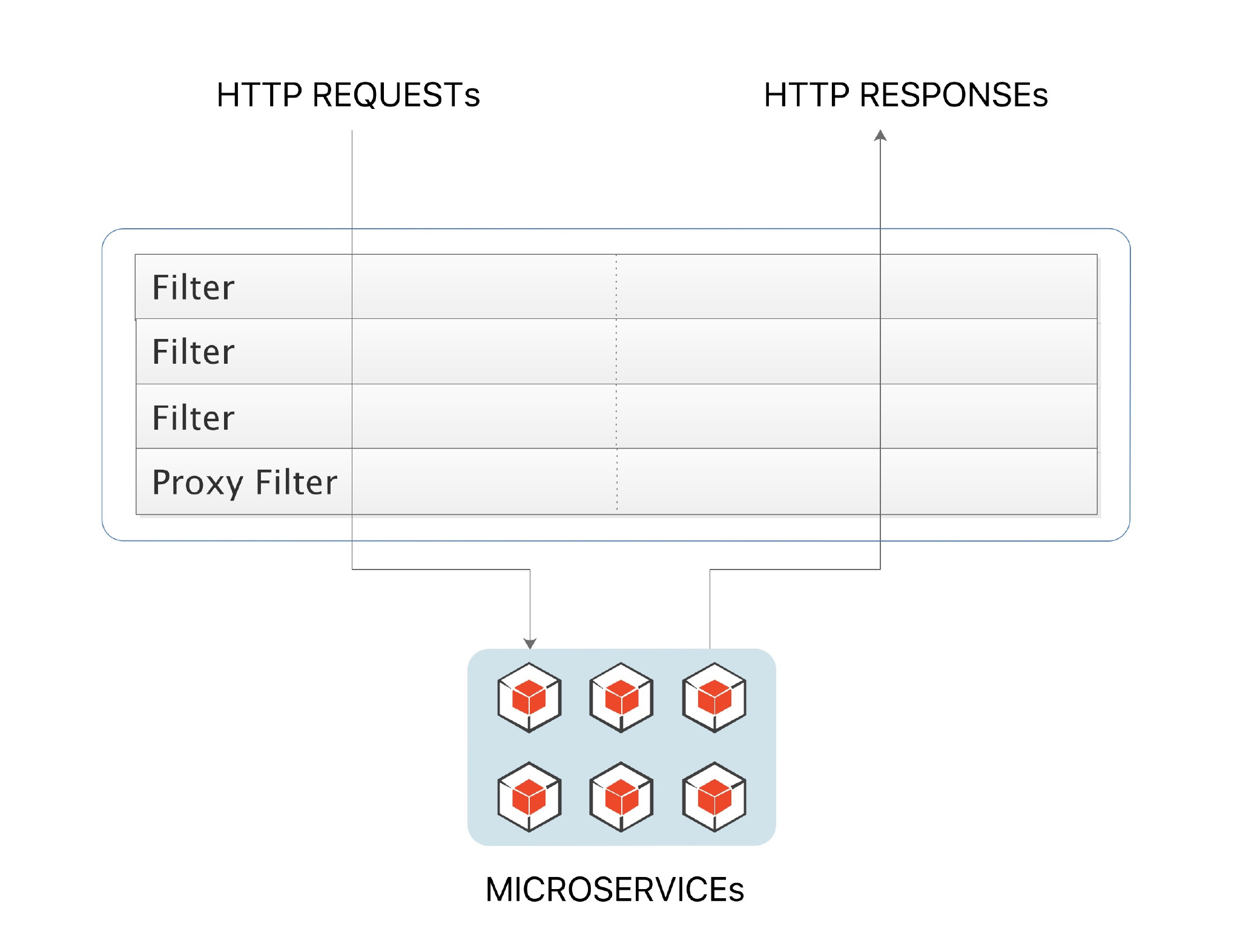
* 何为谓词?(网关中封装了判断逻辑的一个对象)
* 谓词逻辑的设计是怎样的？(谓词判断逻辑返回值为true则进行请求转发)
* 你了解哪些谓词逻辑？(path,请求参数，ip，请求方式，cookie,请求头,….)
* 我们可以自己定义谓词工厂对象吗？(可以的)

## 过滤器增强分析

过滤器(Filter)就是在请求传递过程中，对请求和响应做一个处理。

### Filter生命周期

在Gateway中, Filter的生命周期只有两个：“pre” 和 “post”。如图所示：



其中：

1）PRE： 此过滤器在请求被路由之前调用，可以基于此过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。

2）POST：此过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。

### 局部过滤器分析

Gateway 的Filter从作用范围可分为两种：GatewayFilter与GlobalFilter。其中：

1. GatewayFilter：应用到单个路由或者一个分组的路由上。
2. GlobalFilter：应用到所有的路由上。

在SpringCloud Gateway中内置了很多不同类型的网关路由过滤器。具体如下：



**案例分析：**

1) 基于AddRequestHeaderGatewayFilterFactory，为原始请求添加Header。例如，为原始请求添加名为 X-Request-Foo ，值为 Bar 的请求头：

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: add\_request\_header\_route  
 uri: https://example.org  
 filters:  
 - AddRequestHeader=X-Request-Foo, Bar

2） 基于AddRequestParameterGatewayFilterFactory，为原始请求添加请求参数及值，例如，为原始请求添加名为foo，值为bar的参数，即：foo=bar。

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: add\_request\_parameter\_route  
 uri: https://example.org  
 filters:  
 - AddRequestParameter=foo, bar

3）基于PrefixPathGatewayFilterFactory，为原始的请求路径添加一个前缀路径，例如，该配置使访问${GATEWAY\_URL}/hello 会转发到uri/mypath/hello。

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: prefixpath\_route  
 uri: https://example.org  
 filters:  
 - PrefixPath=/mypath

4）基于RequestRateLimiterGatewayFilterFactory，实现限流操作，算法为令牌桶算法,配置如下：

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: requestratelimiter\_route  
 uri: https://example.org  
 filters:  
 - name: RequestRateLimiter  
 args:  
 redis-rate-limiter.replenishRate: 10  
 redis-rate-limiter.burstCapacity: 20

5)基于RequestSizeGatewayFilterFactory，设置允许接收最大请求包的大小，配置示例：

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: request\_size\_route  
 uri: http://localhost:8080/upload  
 predicates:  
 - Path=/upload  
 filters:  
 - name: RequestSize  
 args:  
 *# 单位为字节* maxSize: 5000000

如果请求包大小超过设置的值，则会返回 413 Payload Too Large以及一个errorMessage

### 全局过滤器以及自定义

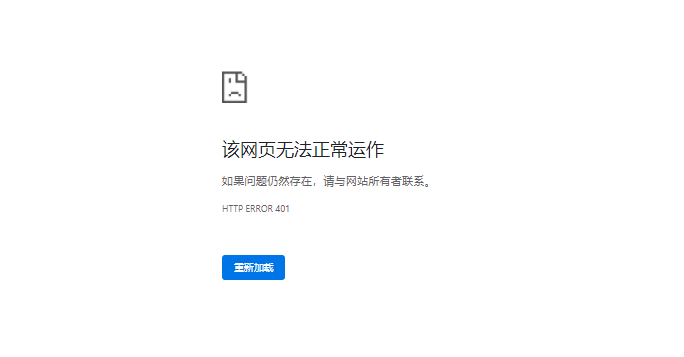
全局过滤器(GlobalFilter)作用于所有路由, 无需配置。在系统初始化时加载，并作用在每个路由上。通过全局过滤器可以实现对权限的统一校验，安全性验证等功能。常用全局过滤器如图所示：



内置的过滤器已经可以完成大部分的功能，但是对于企业开发的一些业务功能处理，还是需要我们 自己编写过滤器来实现的，那么我们一起通过代码的形式自定义一个过滤器，去完成统一的权限校验。 例如，当客户端第一次请求服务时，服务端对用户进行信息认证（登录） 认证通过，将用户信息进行加密形成token，返回给客户端，作为登录凭证 以后每次请求，客户端都携带认证的token 服务端对token进行解密，判断是否有效。

package com.cy.filters;  
  
import org.springframework.cloud.gateway.filter.GatewayFilterChain;  
import org.springframework.cloud.gateway.filter.GlobalFilter;  
import org.springframework.core.Ordered;  
import org.springframework.http.HttpStatus;  
import org.springframework.stereotype.Component;  
import org.springframework.web.server.ServerWebExchange;  
import reactor.core.publisher.Mono;  
@Component  
public class AuthGlobalFilter implements GlobalFilter, Ordered {  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 String token =exchange.getRequest()  
 .getQueryParams().getFirst("token");  
 if (!"admin".equals(token)) {  
 System.*out*.println("认证失败");  
 exchange.getResponse().setStatusCode(HttpStatus.*UNAUTHORIZED*);  
 return exchange.getResponse().setComplete();  
 }  
 *//调用chain.filter继续向下游执行* return chain.filter(exchange);  
  
 }  
  
 @Override  
 public int getOrder() {  
 return 0;  
 }  
}

启动Gateway服务，假如在访问的url中不带“token=admin”这个参数，可能会出现异常，如图所示：



### 小节面试分析

* 网关过滤器的作用是什么？(对请求和响应数据做一个预处理)
* 网关过滤器的类型有哪些？(局部过滤器，全局过滤器)
* 如何理解局部过滤器？(针对具体链路的应用的过滤器，需要进行配置)
* 你了解哪些局部过滤器？
* 如何理解全局过滤器？(作用于所有请求链路)
* 如何自己定义全局过滤器?(直接或间接实现GlobalFilter接口)
* 假如现在让你进行平台的网关自研设计，你可以吗？(可以)
* 网关过滤器的作用是什么？(对请求和响应数据做一个预处理)
* 网关过滤器的类型有哪些？(局部过滤器，全局过滤器)
* 如何理解局部过滤器？(针对具体链路的应用的过滤器，需要进行配置)
* 你了解哪些局部过滤器？
* 如何理解全局过滤器？(作用于所有请求链路)
* 如何自己定义全局过滤器?(直接或间接实现GlobalFilter接口)
* 假如现在让你进行平台的网关自研设计，你可以吗？(可以)

## 限流设计及实现

### 限流简述

网关是所有请求的公共入口，所以可以在网关进行限流，而且限流的方式也很多，我们采用Sentinel组件来实现网关的限流。Sentinel支持对SpringCloud Gateway、Zuul等主流网关进行限流。参考网址如下：

<https://github.com/alibaba/spring-cloud-alibaba/wiki/Sentinel>

### 限流快速入门

第一步：添加依赖

在原有spring-cloud-starter-gateway依赖的基础上再添加如下两个依赖，例如：

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

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

第二步：添加sentinel及路由规则(假如已有则无需设置)

routes:  
 - id: route01  
 uri: lb://nacos-provider  
 predicates: *###匹配规则* - Path=/provider/echo/\*\*

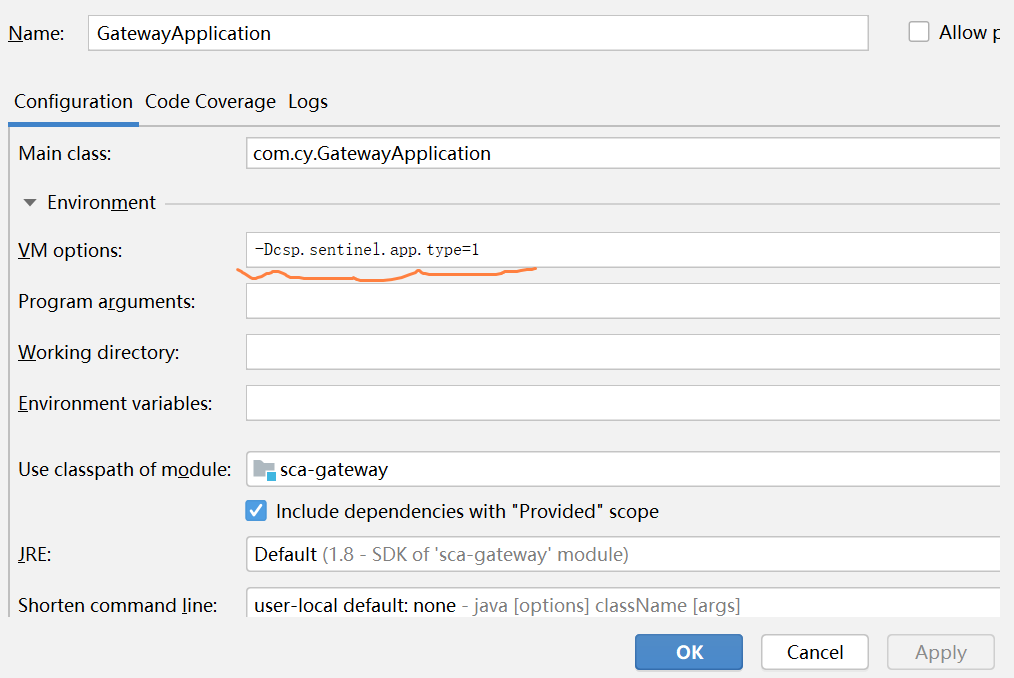
sentinel:  
 transport:  
 dashboard: localhost:8180 *#Sentinel 控制台地址* port: 8719 *#客户端监控API的端口* eager: true *#取消Sentinel控制台懒加载,即项目启动即连接*

第三步：启动网关项目，检测sentinel控制台的网关菜单。

启动时，添加sentinel的jvm参数，通过此菜单可以让网关服务在sentinel控制台显示不一样的菜单（假如没有发现变化关闭网关项目，关闭sentinel，然后重启sentinel,重启网关项目），代码如下。

-Dcsp.sentinel.app.type=1

假如是在idea中，可以参考下面的图进行配置



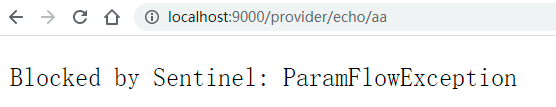
Sentinel 控制台启动以后，界面如图所示：



第四步：在sentinel面板中设置限流策略，如图所示：



第五步：通过url进行访问检测是否实现了限流操作

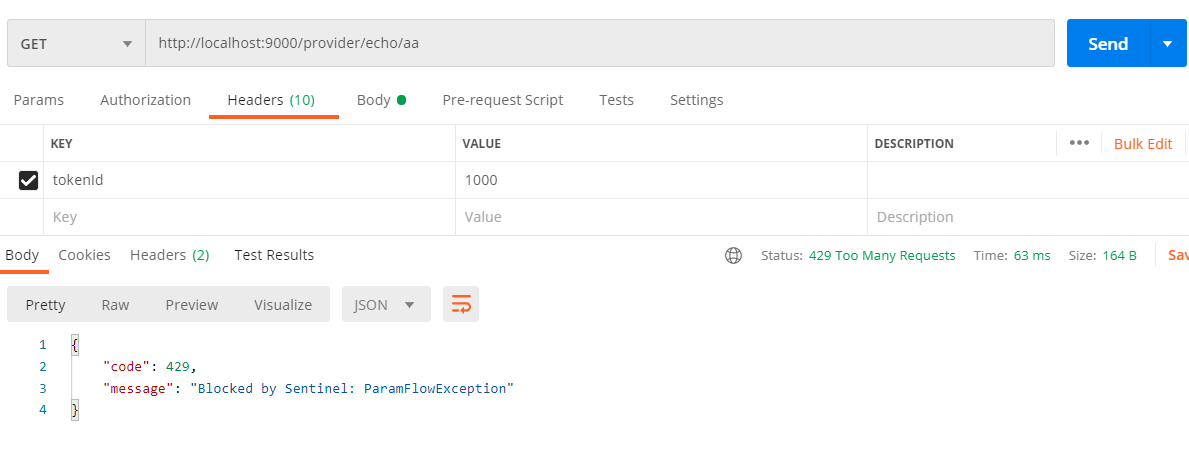


### 基于请求属性限流

定义指定routeId的基于属性的限流策略如图所示：



通过postman进行测试分析



### 自定义API维度限流

自定义API分组，是一种更细粒度的限流规则定义，它允许我们利用sentinel提供的API，将请求路径进行分组，然后在组上设置限流规则即可。

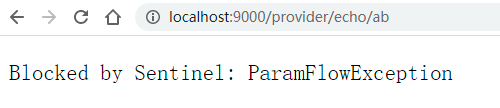
第一步：新建API分组，如图所示：



第二步：新建分组流控规则，如图所示：



第三步：进行访问测试，如图所示



### 定制流控网关返回值

定义配置类，设计流控返回值，代码如下：

@Configuration  
public class GatewayConfig {  
  
 public GatewayConfig(){  
 GatewayCallbackManager.*setBlockHandler*(

new BlockRequestHandler() {  
 @Override  
 public Mono<ServerResponse> handleRequest(ServerWebExchange serverWebExchange, Throwable throwable) {  
 Map<String,Object> map=new HashMap<>();  
 map.put("state",429);  
 map.put("message","two many request");  
 String jsonStr=JSON.*toJSONString*(map);  
 return ServerResponse.*ok*().body(Mono.*just*(jsonStr),String.class);  
 }  
 });  
 }  
}

其中，Mono 是一个发出(emit)0-1个元素的Publisher<T>。

### 小节面试分析？

* 网关层面结合sentinel实现限流，其限流的类型有几种？(两种-route id,api)
* 网关层面可以自定义限流后的异常处理结果吗？(可以)
* 你知道Sentinel底层限流的算法有哪些？(滑动窗口，令牌桶，漏斗，。。。)

# 链路监控zipkin设计及实现

## 背景分析

随着业务越来越复杂，系统也随之进行各种拆分，特别是随着微服务架构和容器技术的兴起，看似简单的一个应用，后台可能有几十个甚至几百个服务在支撑；一个前端的请求可能需要多次的服务调用最后才能完成；当请求变慢或者不可用时，我们无法得知是哪个后台服务引起的，这时就需要解决如何快速定位服务故障点，Zipkin分布式跟踪系统就能很好的解决这样的问题。

## ZipKin是什么？

 Zipkin是一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 Google Dapper的论文设计而来，由 Twitter 公司开发贡献。其主要功能是聚集来自各个异构系统的实时监控数据。分布式跟踪系统还有其他比较成熟的实现，例如：Naver的Pinpoint、Apache的HTrace、阿里的鹰眼Tracing、京东的Hydra、新浪的Watchman，美团点评的CAT，skywalking等。

## ZipKin快速入门实现？

### 业务描述

通过ZipKin实现微服务各个链路之间的调用关系及耗时等。

### 启动zipKin服务

第一步：从官网下载zipkin,默认下载依赖是一个jar文件,网址如下：

<https://github.com/openzipkin/zipkin>

第二步：启动下载的Jar文件，例如：

Java -jar zipkin-server-2.23.2-exec.jar

### ZipKin链路监控应用实践

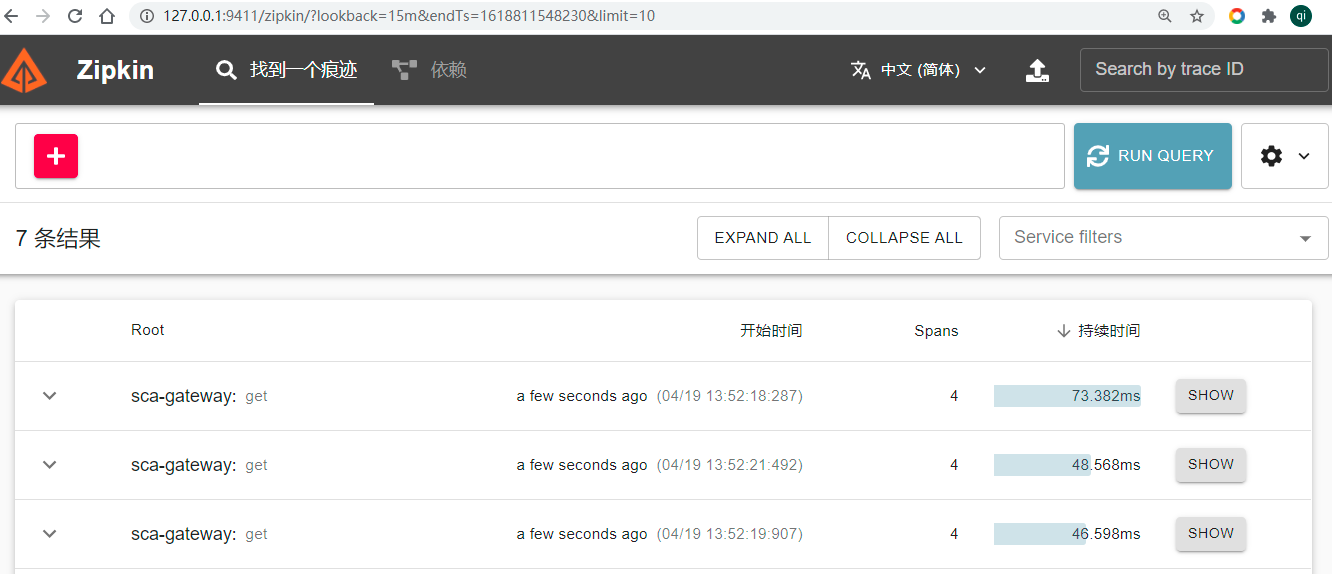
第一步：父工程添加依赖

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

第二步：启动微服务项目进行链路监控（多次访问）

Zipkin底层会基于sleuth进行链路监控，但sleuth默认只对10%的链路请求进行监控，所以访问时多刷新几次。

第三步：访问zipkin检测链路信息，如图所示：



### 相关属于应用说明

* **traceId**

一次请求全局只有一个traceId。用来在海量的请求中找到同一链路的几次请求。比如servlet服务器接收到用户请求，调用微服务，然后将结果返回给用户，整条链路只有一个traceId。开始于用户请求，结束于用户收到结果。

* **spanId**

一个链路中每次请求都会有一个spanId。例如一次rpc，一次sql都会有一个单独的spanId从属于traceId。

* **cs**

Clent Sent 客户端发起请求的时间，比如微服务调用端开始执行远程调用之前。

* **cr**

Client Receive 客户端收到处理完请求的时间。

* **ss**

Server Receive 服务端处理完逻辑的时间。

* **sr**

Server Receive 服务端收到调用端请求的时间。

例如：

sr - cs = 请求在网络上的耗时。

ss - sr = 服务端处理请求的耗时。

cr - ss = 回应在网络上的耗时。

cr - cs = 一次调用的整体耗时。

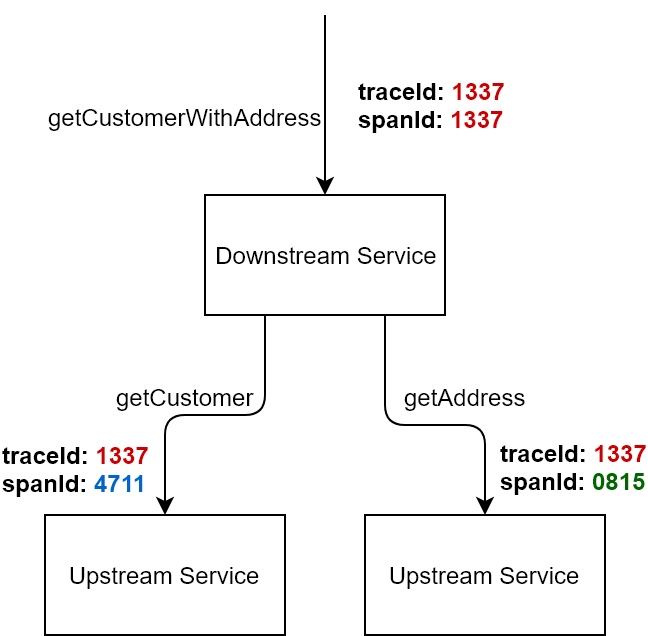
### ZipKin 工作过程分析

当用户发起一次调用时，Zipkin 的客户端会在入口处为整条调用链路生成一个全局唯一的 trace id，并为这条链路中的每一次分布式调用生成一个 span id。span 与 span 之间可以有父子嵌套关系，代表分布式调用中的上下游关系。span 和 span 之间可以是兄弟关系，代表当前调用下的两次子调用。一个 trace 由一组 span 组成，可以看成是由 trace 为根节点，span 为若干个子节点的一棵树。

 Zipkin 会将 trace 相关的信息在调用链路上传递，并在每个调用边界结束时异步的把当前调用的耗时信息上报给 Zipkin Server。Zipkin Server 在收到 trace 信息后，将其存储起来。随后 Zipkin 的 Web UI 会通过 API 访问的方式从存储中将 trace 信息提取出来分析并展示。

### 小节面试分析

* 为什么要进行链路监控?(及时发现问题、找到问题并解决问题)
* 你了解市场那些链路监控产品?(zipkin/skywalking)
* Zipkin进行链路监控的过程是怎样的？
* Zipkin 进行链路监控时有什么缺陷吗？(性能问题，监控数据的设计需要更详细一些)



# 总结(Summary)

## 重难点分析

## FAQ分析

## Bug分析