spring cloud alibaba

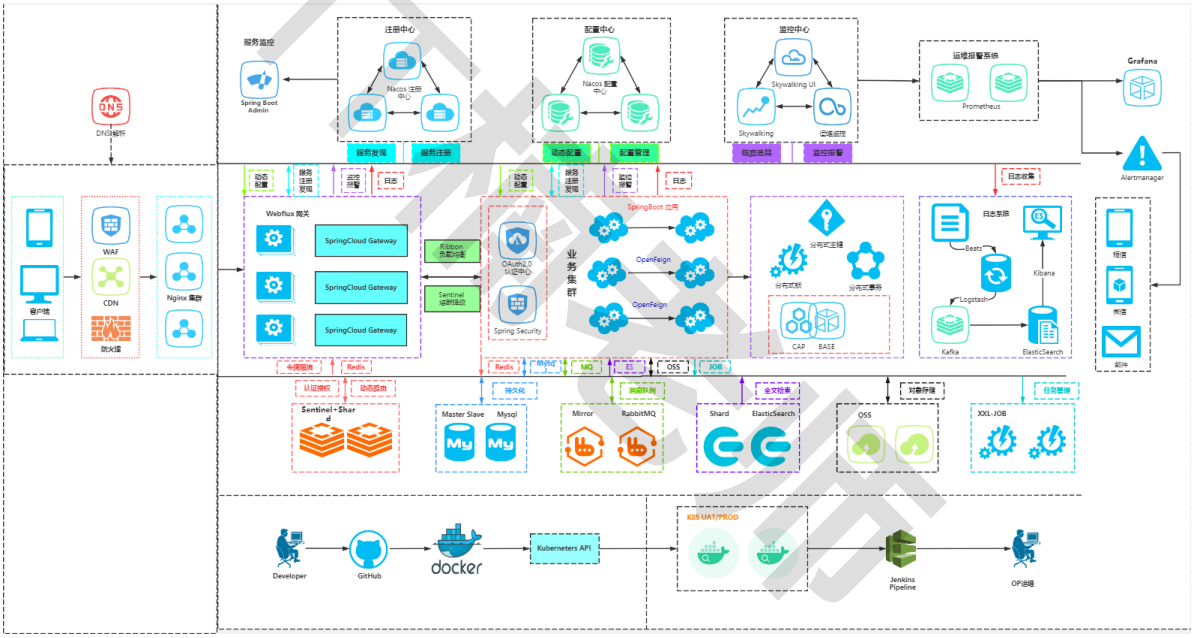
第一章 初识 Spring Cloud

一、微服务架构

- "微服务”一词源于 Martin Fowler的名为 Microservices的博文,可以在他的官方博客上找到http://martinfowler.com/articles/microservices.html
- 微服务是系统架构上的一种设计风格,它的主旨是将一个原本独立的系统拆分成多个小型服务,这些小型服务都在各自独立的进程中运行,服务之间一般通过 HTTP 的 RESTfuL API 进行通信协作。
- 由于有了轻量级的通信协作基础,所以这些微服务可以使用不同的语言来编写。

https://blog.csdn.net/zpoison/article/details/80729052

看一个真真牛逼的项目架构



二、走进 Spring Cloud

- Spring Cloud 是**一系列框架**的有序集合，他不是一个简单的框架，他是一套解决方案。
- Spring Cloud 并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来。
- 通过 **Spring Boot** 风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。
- 它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、 断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。
- Spring Cloud项目官方网址：https://spring.io/projects/spring-cloud
- Spring Cloud 版本命名方式采用了**伦敦地铁站的名称**，同时根据字母表的顺序来对应版本时间顺序，比如：最早的Release版本：Angel，第二个Release版本：Brixton，然后是Camden、Dalston、Edgware，Finchley，Greenwich，Hoxton。

目前最新的是Hoxton版本。

Spring Cloud Version	Spring Cloud Alibaba Version	Spring boot Version
Spring Cloud Edgware	1.5.1.RELEASE	1.5.X.RELEASE
Spring Cloud Finchley	2.0.2.RELEASE	2.0.X.RELEASE
Spring Cloud Greenwich	2.1.2.RELEASE	2.1.X.RELEASE
Spring Cloud Hoxton.RELEASE	2.2.0.RELEASE	2.2.X.RELEASE
Spring Cloud Hoxton.SR3	2.2.1.RELEASE	2.2.5.RELEASE

• 微服务就是将项目的各个模块拆分为可独立运行、部署、测试的架构设计风格。

• Spring 公司将其他公司中微服务架构常用的组件整合起来，并使用 SpringBoot 简化其开发、配置。称为 Spring Cloud

• Spring Cloud 与 Dubbo都是实现微服务有效的工具。Dubbo 性能更好，而 Spring Cloud 功能更全面。

第二章 服务通信的问题

所谓微服务就是要将一个臃肿的项目拆分成多个细分业务的服务，要解决的就是三高问题。

1. 高性能：响应要快，
2. 高并发：人多了不蹦
3. 高可用：崩了一些服务器还能用

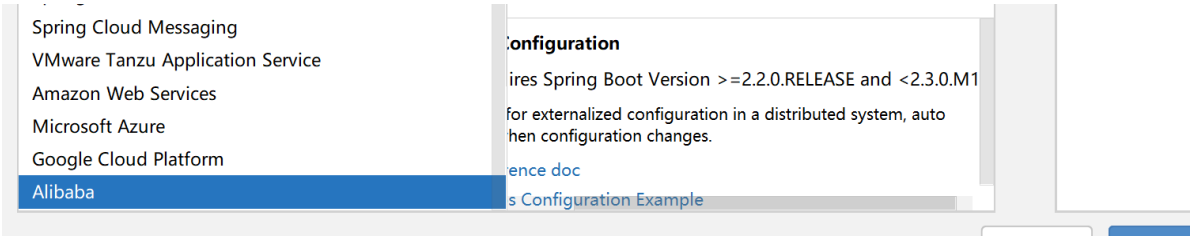
首先我们解决第一个问题、多个服务之间怎么通信呀

服务之间通信，两个后台通信，就是我们的网络通信，可以是socket写，当然我们有更简单的http

这类的框架很多，httpclient，okhttp等都行

一、RestTemplate 远程调用

- Spring提供的一种简单便捷的模板类，用于在 java 代码里访问 restful 服务。
- 其功能与 HttpClient 类似，但是 RestTemplate 实现更优雅，使用更方便。



手动创建一个父工程

1、父工程 spring-cloud-parent

pom.xml

```xml
<?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>cn.itnanls</groupId>
    <artifactId>study-spring-cloud</artifactId>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>provider</module>
    </modules>
    <packaging>pom</packaging>

    <!--统一管理jar包版本-->
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <spring-boot.version>2.2.2</spring-boot.version>
        <alibaba.cloud.version>2.1.0</alibaba.cloud.version>
    </properties>


    <!--子模块继承之后,提供作用:锁定版本+子modlue不用写groupId和version-->
    <dependencyManagement>
        <dependencies>
            <!--spring  boot  2.2.2-->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.2.0.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>

            <!-- spring cloud  Hoxton.SR3-->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Hoxton.SR3</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>

            <!--spring cloud alibaba  2.1.0 RELEASE-->
            <dependency>
                <groupId>com.alibaba.cloud</groupId>
                <artifactId>spring-cloud-alibaba-dependencies</artifactId>
                <version>2.2.5.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
```

```
    </dependencyManagement>

    <build>
        <finalName>${project.artifactId}</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <fork>true</fork> <!-- 如果没有该配置，devtools不会生效 -->
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

注入一个RestTemplate

RestTemplateConfig

```
@Configuration
public class RestTemplateConfig {

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

服务一

```
/**
 * @author itnanls
 * @date 2021/4/12
 */
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(){
        return "hello, B!" ;
    }
}
```

服务二

```
/**
 * @author itnanls
 * @date 2021/4/12
 */
@RestController
public class HelloTwo {
```

```
7
8       @Resource
9       private RestTemplate restTemplate;
10
11      @GetMapping("fromOne")
12      public String fromA(){
13          // 发送http请求，调用远程服务
14          ResponseEntity<String> stringResponseEntity =
15                  restTemplate.getForEntity("/hello", String.class);
16          return stringResponseEntity.getBody();
17      }
18  }
```

调用成功。

当然其中有很多的方法，大家可以尝试一下

但是在微服务里还是觉得麻烦

> 问题拓展

将来服务越来越多，订单服务一台机子也撑不下，同时我们总是依靠ip去寻找服务也不太合适，如果每一个服务都有一个名字就好了。

大家还记得dns吗?

当大家记不住ip地址，就搞一个域名嘛，访问的时候根据域名找一下ip不就行了？，只要域名不变，以后ip随便变，都没有影响

所以在微服务里出现第一个词语，叫注册中心

这类组件有很多 eureka，zookeeper等等

这玩意就提供了一个服务中心，给我们上线的服务注册。

二、Nacos服务管理

- Nacos（Dynamic Naming and Configuration Service） 是阿里巴巴2018年7月开源的项目。
- 它专注于服务发现和配置管理领域 致力于帮助您发现、配置和管理微服务。Nacos 支持几乎所有主流类型的“服务”的发现、配置和管理。
- 一句话概括就是Nacos = Spring Cloud注册中心 + Spring Cloud配置中心。
- 官网：https://nacos.io/
- 下载地址： https://github.com/alibaba/nacos/releases

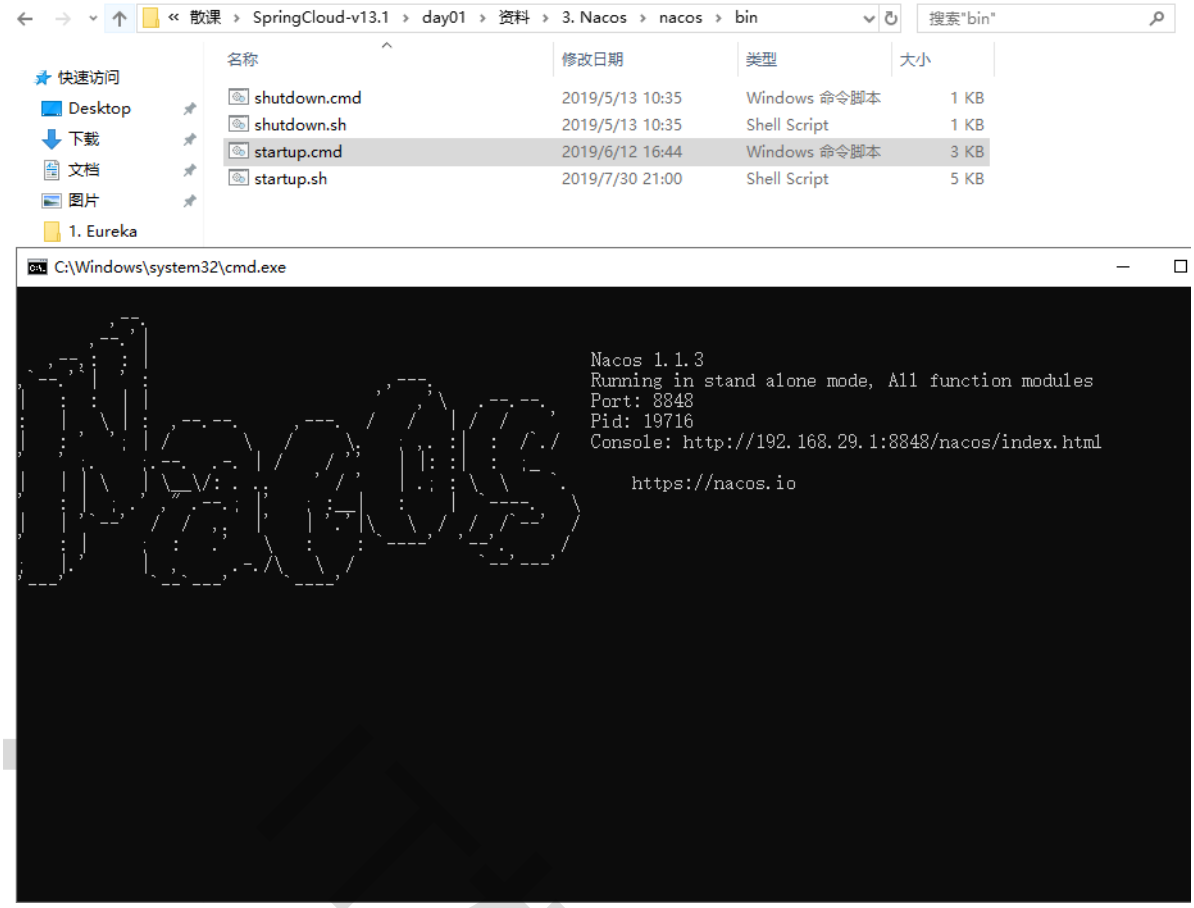
####

1、环境搭建

看资料中的代码即可

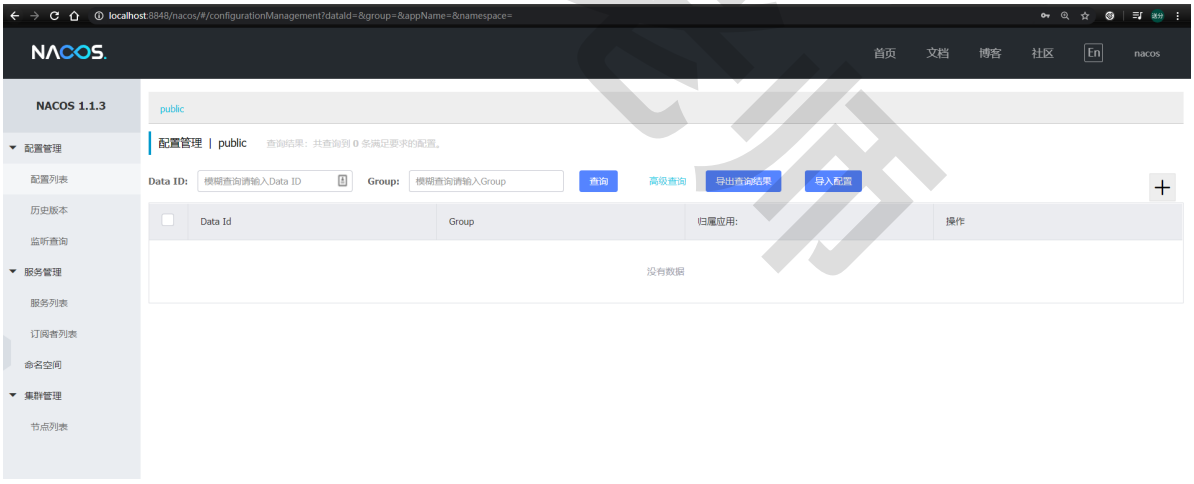
1、环境准备。

下载nacos，双击startup.cmd。



2、访问控制台

http://localhost:8848/nacos/#/login 用户名 密码为nacos



2、理解服务注册

服务注册：把自己注册上去，你开了个卖家具的，登记在58同城

服务发现：让别人发现你，你要卖家具去58同城找到电话去买

> nacos是一个独立的工程，他其实就是暴露了一些接口

（1）服务注册

```
curl -X POST 'http://127.0.0.1:8848/nacos/v1/ns/instance?
serviceName=nacos.naming.serviceName&ip=20.18.7.10&port=8080'

registerInstance(String serviceName,String ip,int port)
```

(2) 服务发现

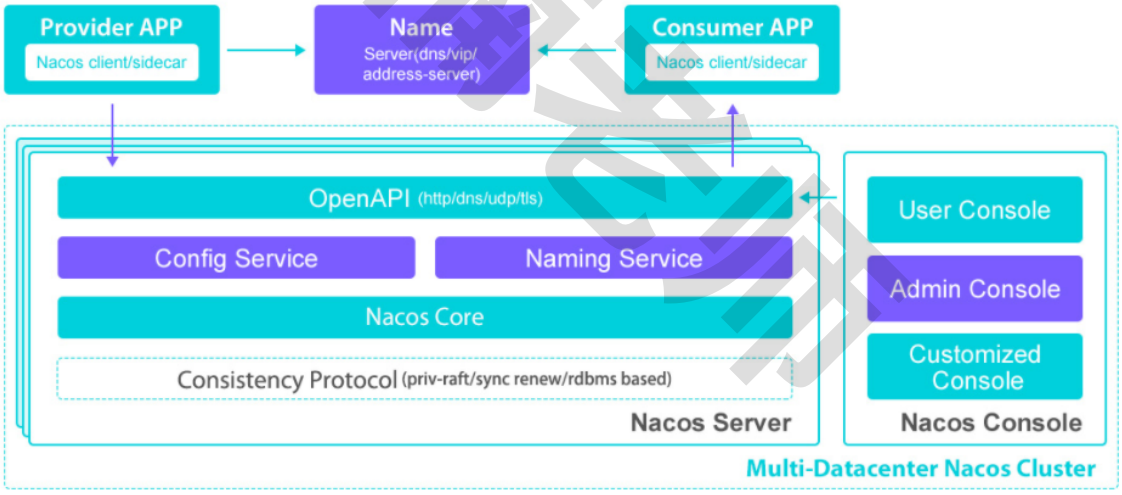
```
curl -X GET 'http://127.0.0.1:8848/nacos/v1/ns/instance/list?
serviceName=nacos.naming.serviceName'

getAllInstances(String serviceName)
```

(3) 发布配置

```
curl -X POST "http://127.0.0.1:8848/nacos/v1/cs/configs?
dataId=nacos.cfg.dataId&group=test&content=HelloWorld"
```

(4) 获取配置

```
curl -X GET "http://127.0.0.1:8848/nacos/v1/cs/configs?
dataId=nacos.cfg.dataId&group=test"
```



3、编写服务提供者

pom

```
<?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>
        <artifactId>study-spring-cloud</artifactId>
        <groupId>cn.itnanls</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
```

```
    <artifactId>provider</artifactId>

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

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

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```


application.properties

```
server.port=8081
spring.application.name=nacos-provider
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
management.endpoints.web.exposure.include=*
```

启动类

```
@SpringBootApplication
@EnableDiscoveryClient
public class NacosProviderDemoApplication {

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

    @RestController
    public class EchoController {
        @GetMapping(value = "/echo/{string}")
        public String echo(@PathVariable String string) {
            return "Hello Nacos Discovery " + string;
        }
```

```
    }
}
```

4、编写消费者

pom

```
<?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>
        <artifactId>study-spring-cloud</artifactId>
        <groupId>cn.itnanls</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>consumer</artifactId>

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

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

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

启动类

```
@SpringBootApplication
@EnableDiscoveryClient
```

```
public class NacosConsumerApp {

    @RestController
    public class NacosController{

        @Autowired
        private LoadBalancerClient loadBalancerClient;
        @Autowired
        private RestTemplate restTemplate;

        @Value("${spring.application.name}")
        private String appName;

        @GetMapping("/echo/app-name")
        public String echoAppName(){
            //Access through the combination of LoadBalanceClient and
RestTemplate
            ServiceInstance serviceInstance =
loadBalancerClient.choose("nacos-provider");
            String path =
String.format("http://%s:%s/echo/%s",serviceInstance.getHost(),serviceInstan
ce.getPort(),appName);
            System.out.println("request path:" +path);
            return restTemplate.getForObject(path,String.class);
        }

    }

    //Instantiate RestTemplate Instance
    @Bean
    public RestTemplate restTemplate(){

        return new RestTemplate();
    }

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

结果

loadBalancerClient

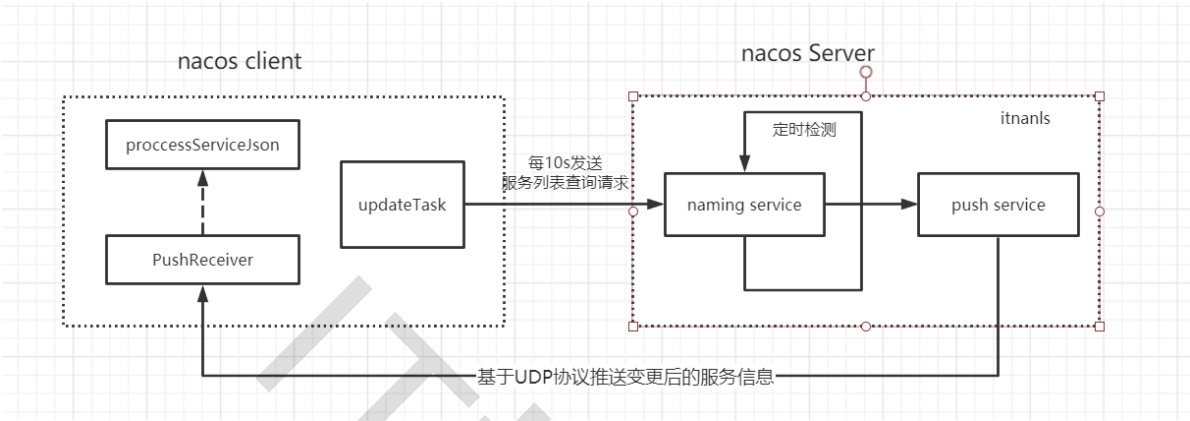
之前我们说过，一个服务一台电脑也扛不住呀，还有问题，单个服务如果崩了，是不是会出问题，单点故障是绝对不允许出现的，那么我们每个服务就需要部署多个，这要根据你的实际情况来确定。根据服务名称按照一定的策略选取合适的服务叫做负载均衡，这个是我说的。

loadBalancerClient就是一个负载均衡客户端，其内部是使用ribbon这个工具。

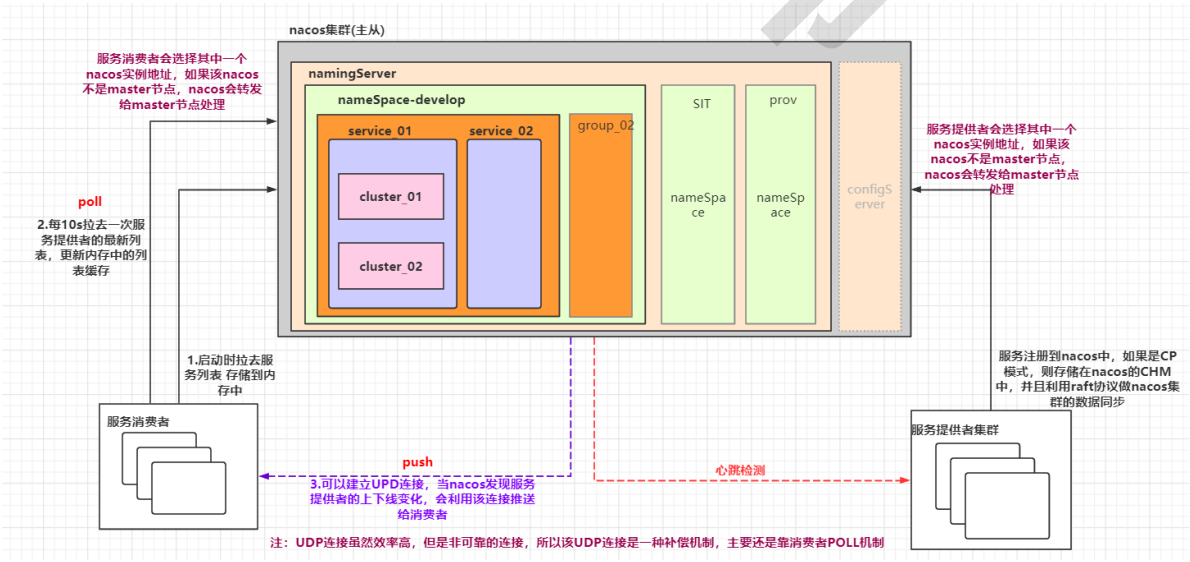
5、原理探究

健康监测

服务地址动态感知



1. 客户端发起事件订阅后，在hostreactor中有一个updateTask线程，每十秒发送一次pull请求，获得服务端最新的地址列表
2. 对于服务端，他和服务者之间的实例维护了心跳检测，一旦服务者发生异常，则会发送一个push消息给nacos客户端，也就是服务消费者
3. 服务消费者收到请求之后，使用hostReactor中提供的proccessServicceJson解析消息，并更新到本地服务列表



6、nacos 高可用集群搭建

nacos 高可用集群搭建使用了raft协议的数据一致性算法，包含一个leader和多个follower，同样采用的还有redis sentinel的leader选举。

（1）修改集群文件

在nacos的解压目录nacos/的conf目录下，有配置文件cluster.conf，请每行配置成ip:port。（请配置3个或3个以上节点）

首先我们进入conf目录下，默认只有一个cluster.conf.example文件，我们需要自行复制一份，修改名称为cluster.conf

```
cp cluster.conf.example cluster.conf
```

然后使用vi编辑器 打开cluster.config，按a/i/o 键可进入插入模式，输入以下内容

```
#ip:port
192.168.120.200:8848
192.168.120.201:8848
192.168.120.202:8848
```

然后按ESC键返回到命令模式，再按shif+:进入末行模式，输入wq敲回车（保存并退出）。

（2）配置mysql数据库

集群环境我们要使用mysql数据库，单体可以使用数据库默认是Derby这种文件类的数据库。

修改配置：

```
db.num=1
db.url.0=jdbc:mysql://192.168.120.201:3306/nacos?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useUnicode=true&useSSL=false&serverTimezone=UTC
db.user.0=root
db.password.0=root
```

新建一个名为nacos的数据库，执行conf文件下自带的sql脚本文件。

（3）克隆三台机子，ip按照要求设置。

成功搭建：

> 集群下客户端怎么连?

第一种：可以在单机的基础上加多个IP和端口中间用逗号隔开；

spring.cloud.nacos.discovery.server-addr=192.168.172.128:8801,192.168.172.128:8802,192.168.172.128:8803

第二种：配合Nginx代理我们的Nacos集群，配置里就直接写Nginx的IP和端口即可；

Nacos集群节点有三种角色状态：leader、follower、candidate；

当leader宕机，会从剩下的follower中投票选举出一个新的leader，选举算法是基于*Raft**算法**实现；*

*经测试，发现有一点与三个角色不符，部署3个nacos节点，其中宕机2台，只剩下一个节点，此节点将变为*candidate角色，但是此时该nacos集群仍然可以注册服务，订阅服务，（按照正确的理论应该是：如果nacos集群中没有leader角色的节点就不能注册服务，因为leader角色处理事务性请求），这比较匪夷所思，有待研究；

第三章 理论

> 思考一个问题：

一、nacos的一致性算法raft

> 看这个博客就可以了：

https://blog.csdn.net/shuchuntang2729/article/details/108393781

二、cap和base理论

1、概念

Consistency (一致性)：

“all nodes see the same data at the same time”,即更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致，这就是分布式的一致性。一致性的问题在并发系统中不可避免，对于客户端来说，一致性指的是并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。

Availability (可用性):

可用性指“Reads and writes always succeed”，即服务一直可用，而且是正常响应时间。好的可用性主要是指系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。

Partition Tolerance (分区容错性):

即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，对于用户而言并没有什么体验上的影响。

2、只能三选其二

CA without P：如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但放弃P的同时也就意味着放弃了系统的扩展性，也就是分布式节点受限，没办法部署子节点，这是违背分布式系统设计的初衷的。

CP without A：如果不要求A（可用），相当于每个请求都需要在服务器之间保持强一致，而P（分区）会导致同步时间无限延长(也就是等待数据同步完才能正常访问服务)，一旦发生网络故障或者消息丢失等情况，就要牺牲用户的体验，等待所有数据全部一致了之后再让用户访问系统。

AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。典型的应用就如某米的抢购手机场景，可能前几秒你浏览商品的时候页面提示是有库存的，当你选择完商品准备下单的时候，系统提示你下单失败，商品已售完。这其实就是先在 A（可用性）方面保证系统可以正常的服务，然后在数据的一致性方面做了些牺牲，虽然多少会影响一些用户体验，但也不至于造成用户购物流程的严重阻塞。

3、Base理论

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。接下来看一下BASE中的三要素：

1、基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性—-注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

2、软状态

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

3、最终一致性

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事物ACID特性是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

4、常见产品

1、Ereka

ereka是SpringCloud系列用来做服务注册和发现的组件，作为服务发现的一个实现，在设计的时候就更考虑了可用性，保证了AP。

2、Zookeeper

Zookeeper在实现上牺牲了可用性，保证了一致性（单调一致性）和分区容错性，也即：CP。所以这也是SpringCloud抛弃了zookeeper而选择Ereka的原因。

3、nacos

既可以支持cp模型也可以支持ap模型，问题服务中心对可用性要求高还是一致性要求高呢？

CP VS AP

ZK定位于分布式协调服务，在其管辖下的所有服务之间保持同步、一致(Zab算法，CP)，若作Service发现服务，其本身没有正确处理网络分割的问题<当多个zk之间网络出现问题-造成出现多个leader-脑裂>，即在同一个网络分区的节点数达不到zk选取leader的数目，它们就会从zk中断开，同时也不能提供Service发现服务。

5、总结

对于分布式系统的项目，使用中没有强制要求一定是CAP中要达到某几种，具体根据各自业务场景所需来制定相应的策略而选择适合的产品服务等。例如：支付订单场景中，由于分布式本身就在数据一致性上面很难保证，从A服务到B服务的订单数据有可能由于服务宕机或其他原因而造成数据不一致性。因此此类场景会酌情考虑：AP，不强制保证数据一致性，但保证数据最终一致性。

##

第四章 负载均衡

> 问题，啥是个负载均衡？

假如你是古代一个大贪官，有十个老婆，不能天天只让一个老婆侍寝，比较和谐的生活就要雨露均沾，这就叫均衡。每个老婆轮流侍寝是一种均衡，分出等级，比如一个月和夫人15天，剩下的给小妾也是一种均衡。

什么是负载，十个老婆负载就有点大了，二十个是不是超负荷了。

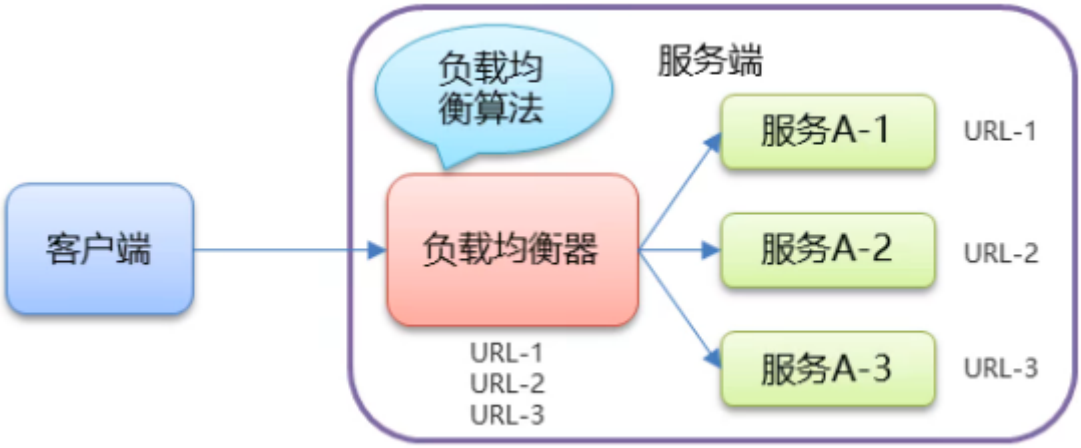
ribben是Netflix公司推出的http和TCP的客户端负载均衡工具。他能简化远程调用代码，还内置很多负载均衡算法。

一、负载均衡分类

1、服务端负载均衡

负载均衡算法在服务端

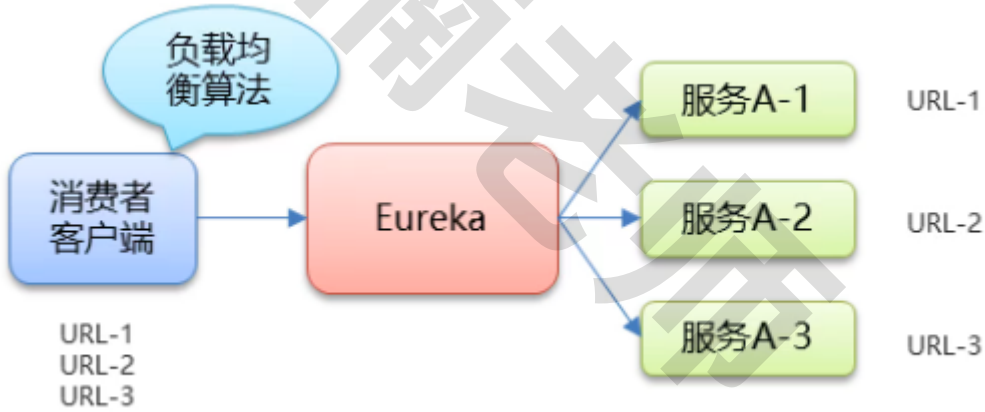
服务端维护服务列表



2、客户端负载均衡

负载均衡算法在客户端

客户端维护服务列表



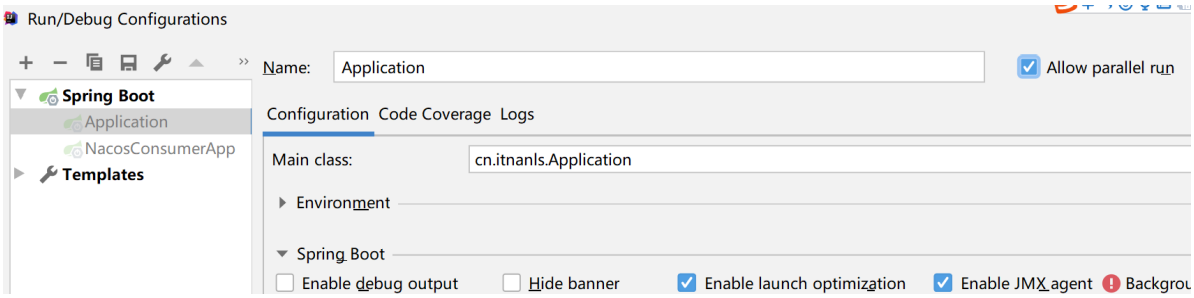
二、集成restTemplate

（1）声明restTemplate时。加上@LoadBalanced注解

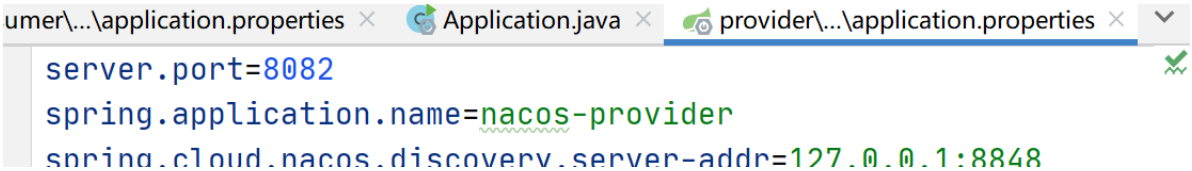
```
@Bean
@LoadBalanced
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

2、启动连个provider

设置启动项，允许并行运行



（3）修改配置文件中的端口。



（4）修改提供者，增加端口

```java
/**
 * @author itnanls
 * @date 2021/4/15
 */
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

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

    @RestController
    public class EchoController {
        @GetMapping(value = "/echo/{string}")
        public String echo(@PathVariable String string) {
            return "Hello Nacos Discovery " + string + ". port is "+ port;
        }
    }

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

}
```

启动多个

修改消费者，将ip修改为服务的名字

```
@GetMapping("/echo/app-name")
public String echoAppName(HttpServletRequest request){
    //Access through the combination of LoadBalanceClient and RestTemplate
    String path = String.format("http://nacos-provider/echo/%s",appName);
    return restTemplate.getForObject(path,String.class);
}
```

3多次访问consumer，发现每次访问的服务不同

三、ribbon 负载均衡策略

内置负载均衡规则类	规则描述
RoundRobinRule	简单轮询服务列表来选择服务器。它是Ribbon默认的负载均衡规则。
AvailabilityFilteringRule	对以下两种服务器进行忽略：（1）在默认情况下，这台服务器如果3次连接失败，这台服务器就会被设置为“短路”状态。短路状态将持续30秒，如果再次连接失败，短路的持续时间就会几何级地增加。注意：可以通过修改配置loadbalancer..connectionFailureCountThreshold来修改连接失败多少次之后被设置为短路状态。默认是3次。（2）并发数过高的服务器。如果一个服务器的并发连接数过高，配置了AvailabilityFilteringRule规则的客户端也会将其忽略。并发连接数的上线，可以由客户端的..ActiveConnectionsLimit属性进行配置。
WeightedResponseTimeRule	为每一个服务器赋予一个权重值。服务器响应时间越长，这个服务器的权重就越小。这个规则会随机选择服务器，这个权重值会影响服务器的选择。
ZoneAvoidanceRule	以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。
BestAvailableRule	忽略哪些短路的服务器，并选择并发数较低的服务器。
RandomRule	随机选择一个可用的服务器。
Retry	重试机制的选择逻辑

拓展资料：https://www.cnblogs.com/cxxjohnson/p/9027919.html

> 设置ribbon 负载均衡策略

1、使用代码配置

1MyRule 返回想要的规则即可

```
package cn.itnanls.consumer.config;

import com.netflix.loadbalancer.IRule;
import com.netflix.loadbalancer.RandomRule;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * creste by itheima.itcast
 */
@Configuration
public class MyRule {
    @Bean
    public IRule rule(){
        return new RandomRule();
    }
}

```

2启动类

```
@RibbonClient(name ="eureka-provider",configuration = MyRule.class)
```

2、使用配置文件配置

consumer工程

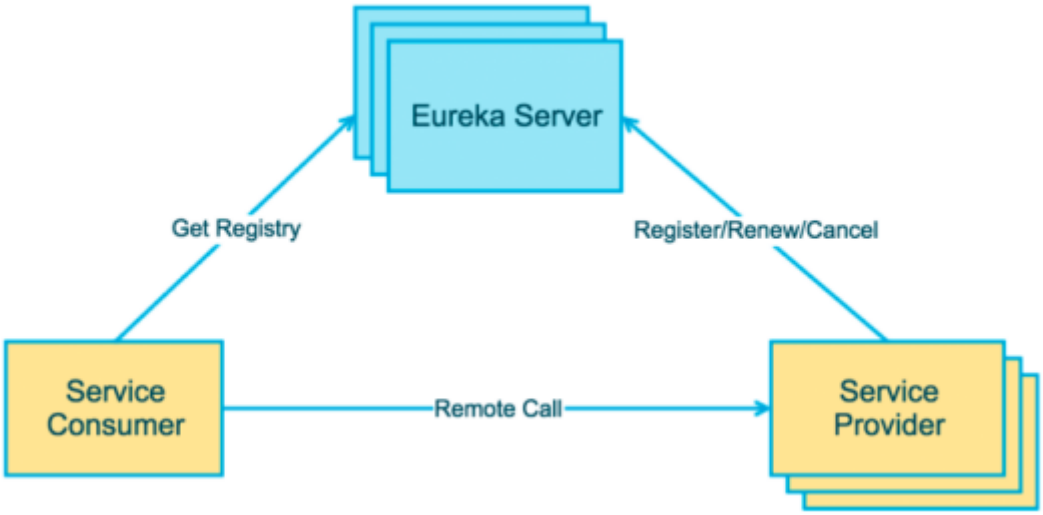
application.yml

```
eureka-provider: #远程服务名
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule #策略
```

第五章 服务调用组件

一、rest调用，http协议

- Feign 是一个声明式的 REST 客户端，它用了**基于接口的注解方式**，很方便实现客户端配置。
- Feign 最初由 Netflix 公司提供，但不支持SpringMVC注解，后由 SpringCloud 对其封装，支持了SpringMVC注解，让使用者更易于接受。



1、实战

1. 在消费端引入 open-feign 依赖

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

2. 编写Feign调用接口。**复制粘贴被调方的conreoller方法,加上类路径。**

```java
/**
 * @author itnanls
 * @date 2021/4/16
 */
@FeignClient(value = "nacos-provider")
public interface NacosProvider {

    /**
     * 调用远程接口
     * @return String
     */
    @GetMapping("/echo/hello")
    public String echo();
}
```

3. 在启动类 添加 @EnableFeignClients 注解，开启Feign功能
4. 测试调用

```java
@RestController
public class NacosController{

    @Autowired
    private NacosProvider nacosProvider;

    @GetMapping("/echo/app-name")
    public String echoAppName(HttpServletRequest request){
        return nacosProvider.echo();
    }
}
```

二、dubbo

dubbo是一款优秀的rpc远程调用框架。

dubbo经常使用zookeeper作为注册中心，但是咱学些springcloud就使用nacos作为注册中心就可以了。

1、实战

> 创建统一api【dubbo-api】

```java
/**
 * @author itnanls
 * @date 2021/4/16
 */
public interface IHelloDubbo {

    /**
     * 公共的接口
     * @return String
     */
    String helloDubbo();

}
```

> 创建dubbo-provider

```xml
<?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>
        <artifactId>study-spring-cloud</artifactId>
        <groupId>cn.itnanls</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
```

```
    <modelVersion>4.0.0</modelVersion>

    <artifactId>dubbo-provider</artifactId>

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

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

        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-dubbo</artifactId>
        </dependency>
        <dependency>
            <artifactId>dubbo-api</artifactId>
            <groupId>cn.itnanls</groupId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

配置

```
server.port=8085
spring.application.name=dubbo-provider
dubbo.scan.base-packages=cn.itnanls.service.rpc
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
dubbo.registry.address=spring-cloud://localhost
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
```

实现这个接口，并注入

```java

/**
 * @author itnanls
 * @date 2021/4/16
 */
@DubboService
public class HelloDubbo implements IHelloDubbo {
    @Override
    public String helloDubbo() {
        return "hello dubbo";
    }
}

```

provider启动类

```java
/**
 * @author itnanls
 * @date 2021/4/16
 */
@SpringBootApplication
@EnableDiscoveryClient
public class DubboProviderApplication {

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

}

```

dubbo-consumer

```xml
<?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>
        <artifactId>study-spring-cloud</artifactId>
        <groupId>cn.itnanls</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>dubbo-consumer</artifactId>

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

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

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

        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-dubbo</artifactId>
        </dependency>
        <dependency>
            <artifactId>dubbo-api</artifactId>
            <groupId>cn.itnanls</groupId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

配置

```
server.port=8086
spring.application.name=dubbo-consumer
dubbo.protocol.name=dubbo
dubbo.protocol.port=20881
dubbo.registry.address=spring-cloud://localhost
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
```

启动类

```
/**
 * @author itnanls
 * @date 2021/4/16
 */
@SpringBootApplication
@EnableDiscoveryClient
public class DubboConsumerApplication {

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

```
12
13      @RestController
14      public class HelloController{
15
16          @DubboReference
17          private IHelloDubbo iHelloDubbo;
18
19          @GetMapping("/hello")
20          public String hello(){
21              return iHelloDubbo.helloDubbo();
22          }
23      }
24  }
```

> 调用

```
1   @RestController
2   public class HelloController{
3
4       @DubboReference
5       private IHelloDubbo iHelloDubbo;
6
7       @GetMapping("/hello")
8       public String hello(){
9           return iHelloDubbo.helloDubbo();
10      }
11  }
```


第六章 统一配置

nacos除了提供了服务注册发现的能力，还提供了统一配置的能力

1. 集中管理配置文件
2. 不同环境不同配置，动态化的配置更新
3. 配置信息改变时，不需要重启即可更新配置信息到服务

1、Nacos服务配置数据模型

这里面涉及到三个概念

> 命名空间

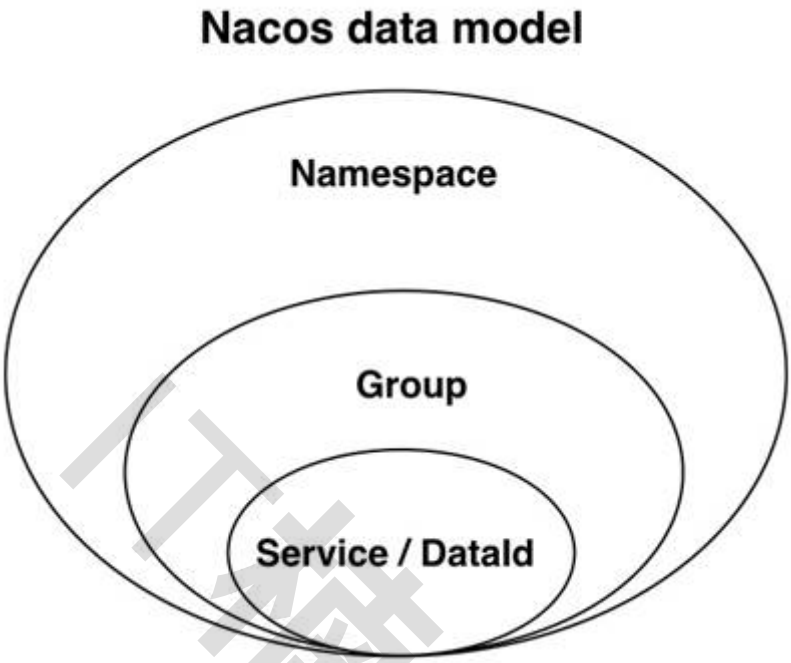
用于配置隔离，不同命名空间下，可以存在相同的Group或Data ID配置，Namespace的常用场景之一是不同环境的配置进行区分隔离，例如开发环境、测试环境和生产环境的资源（如配置、服务）隔离等；

> Group

Nacos中的一组配置集合，是组织配置的维度之一，通过一个有意义的字符串（如 Buy 或 Trade）对一组配置集合进行分组，从而区分Data ID相同的配置集合，当在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP，配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database_url配置和MQ_topic配置；

Data Id

Nacos中的某个配置集合的ID，配置集合ID是组织划分配置的维度之一，Data ID通常用于组织划分系统的配置集合，一个系统或者应用可以包含多个配置集合，每个配置集都可以被一个有意义的名称标识；



配置文件的命名规范

命名空间 + Group分组 + 自定义Data Id（没有默认值）；

默认命名空间：public

默认Group：DEFAULT_GROUP

自定义Data Id：

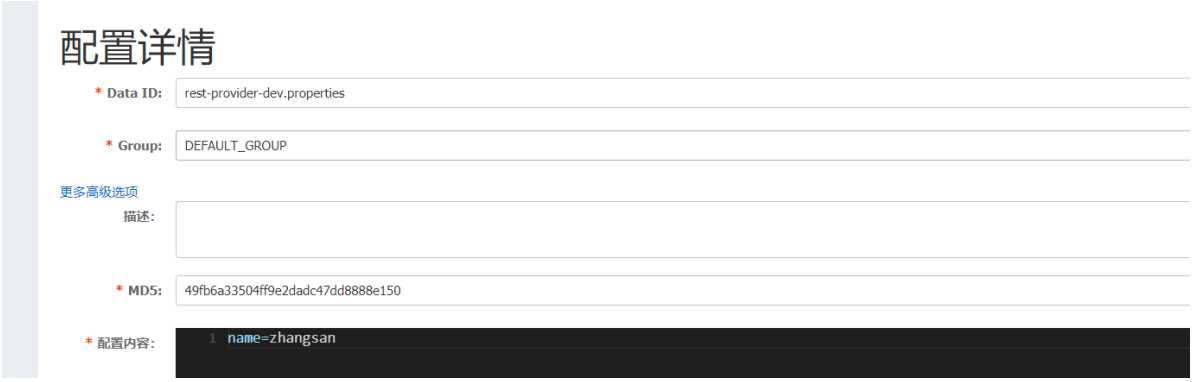
服务名-环境.后缀

${spring.application.name}-${profile}.${file-extension:properties}

比如：nacos-config-dev.yaml

2、实战

（1）启动好Nacos之后，在Nacos添加如下的配置：



（2）添加依赖

spring.cloud.nacos.config.server-addr 配置的方式为 域名:port，例如 Nacos的域名为 nacos.power.com，监听的端口为80，

则 spring.cloud.nacos.config.server-addr=nacos.power.com:80,注意80 端口不能省略；

如果要在项目中使用Nacos来实现应用的外部化配置，需要添加如下依赖：

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

（3）在 `bootstrap.properties` 中配置 Nacos server 的地址和应用名

```
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
spring.application.name=rest-provider
```

说明：之所以需要配置 spring.application.name ，是因为它是构成 Nacos 配置管理 dataId字段的一部分。

在 Nacos Spring Cloud 中，dataId 的完整格式如下：

```
${prefix}-${spring.profiles.active}.${file-extension}
```

（4）通过 Spring Cloud 原生注解 `@RefreshScope` 实现配置自动更新：

```
@RestController
@RequestMapping("/hello")
@RefreshScope
public class ConfigController {

    @Value("${user.name}")
    private boolean username;

    @RequestMapping("/get")
    public boolean get() {
        return username;
    }
}
```

我们还可以统统开放api添加配置

比如：

```
curl -X POST "http://127.0.0.1:8848/nacos/v1/cs/configs?
dataId=example.properties&group=DEFAULT_GROUP&content=useLocalCache=true"
```

3、Nacos Config配置中心动态刷新

Nacos Config Starter 默认为所有获取数据成功的 Nacos 的配置项添加了监听功能，在监听到服务端配置发生变化时会实时触发 org.springframework.cloud.context.refresh.ContextRefresher的refresh 方法；

可以通过配置 spring.cloud.nacos.config.refresh.enabled=false 来关闭动态刷新；

配置中心和客户端(我们自己的业务工程)实现配置的动态感知一般无外乎两种办法，

一种是客户端主动发起pull

一种是配置中心发起push。

nacos作为配置中心和客户端是通过客户端主动发起的pull模式，但是这不是单纯的pull，是一个httppost的长轮询，过期时间默认是30S。

什么意思呢，客户端发起一个http请求发给配置中心，配置中心接到了请求那一刻如果正好配置有变化就立即返回变化后的配置。但是如果在这一刻没有变化配置中心不会给客户端返回数据，等到29.5S后配置中心会启动一个延时定时任务，回去检查配置是否有更新。但是我们都清楚如果这29.5S期间，配置修改了难道客户端要过了这个期间才会知道配置修改了么，显然这个是不合理的。nacos采用了发布订阅模式，配置修改了服务端会发布一个事件，这个事件还会有一个监听的地方，这样就达到了动态感知配置的修改。

第七章 服务保护

Alibaba Sentinel 是面向云原生微服务的流量控制,熔断降级组件,监控保护你的微服务

> 先来看看Sentinel的前身 Hystrix ,总结下来有几点

1. 需要我们程序员手工搭建监控平台
2. 没有一套web界面可以给我们进行更加细粒度配置流控,速率控制,服务熔断,服务降级...

> 而Sentinel的优势

1. 单独一个组件,可以独立出来
2. 直接界面化的细粒度统一配置

一、Sentinel是什么

> 分布式系统的流量防卫兵

- **丰富的应用场景**: Sentinel承接了阿里巴巴近十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
- **完美的实时监控**:
  Sentinel同事提供实时的监控功能,您可以在控制台看到接入应用的单台机器秒级数据,甚至500台一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他框架/库的整合模块,例如与SpringCloud,Dubbo,gRPC的整合,您只需要引入响应的依赖并进行简单的配置即可快速接入Sentinel.
- **完美的SPI扩展点**:
  Sentinel提供简单易用的,完美的SPI扩展接口,可以通过实现扩展接口来快速定制逻辑,例如定制规则管理,适配动态数据源等.

> Sentinel 分为两个部分

- 核心库(java客户端) 不依赖于任何框架/库,能够运行所有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Srping Boot开发,打包后可以直接运行,不需要额外的Tomcat等容器

二、实战

1、引入依赖

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

2、配置sentinel控制台

application.yml

```
spring.cloud.sentinel.transport.port=8719
spring.cloud.sentinel.transport.dashboard=localhost:8080
```

指定的端口号' spring.cloud.sentinel.transport。port'将在应用程序的相应服务器上启动一个HTTP服务器，这个服务器将与Sentinel仪表板交互。例如，如果在Sentinel仪表板中添加了速率限制规则，那么规则数据将被推到HTTP服务器并由HTTP服务器接收，而HTTP服务器将该规则注册到Sentinel。

3、丰富的配置

1、响应时间(RT)：响应时间是指系统对请求作出响应的时间。

2、吞吐量(Throughput)：吞吐量是指系统在单位时间内处理请求的数量。对于无并发的应用系统而言，吞吐量与响应时间成严格的反比关系，实际上此时吞吐量就是响应时间的倒数。

3、并发用户数：并发用户数是指系统可以同时承载的正常使用系统功能的用户的数量。与吞吐量相比，并发用户数是一个更直观但也更笼统的性能指标。实际上，并发用户数是一个非常不准确的指标，因为用户不同的使用模式会导致不同用户在单位时间发出不同数量的请求。

4、QPS每秒查询率(Query Per Second)

每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准，在因特网上，作为域名系统服务器的机器的性能经常用每秒查询率来衡量。对应fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。 （看来是类似于TPS，只是应用于特定场景的吞吐量）

5、限流

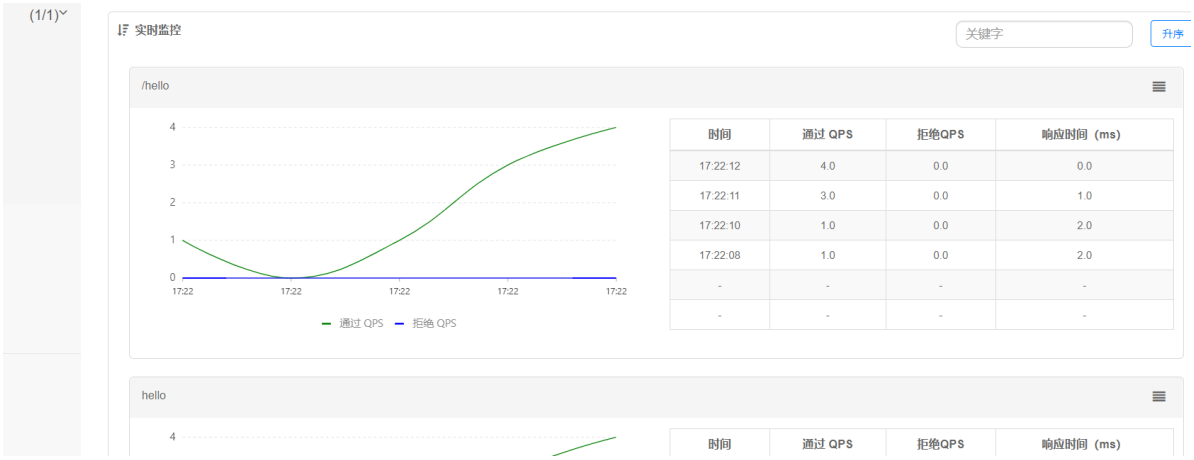
6、熔断

7、降级

（1）限流的配置

> 需要访问一下，才能出来

> 实时查看QPS



簇点链路:查看Http请求

流控规则 :控制qps,线程等等

查过峰值,会抛出异常(可自行配置服务降级回调方法)

Blocked by Sentinel (flow limiting)

- 阈值关联 : 当关联资源/testB的qps阈值超过1时,就限流/testA的Rest访问地址
- 预热 : 每个系统平时处于低水位的情况下,突然来高并发的流量,肯定是承受不住的,要经过预热后才会使系统达到稳定可以接受的状态
- 队列等待(常用)

匀速排队,让请求以均匀的速度通过,阈值类型必须需设成QPS,否则无效

如图所示,每秒请求1次,超过的话就排队等待,等待时间0.5秒

这种方式主要用于处理间隔性突发流量,例如消息队列,在某一秒有大量的请求道来,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,而不是一秒直接拒绝多余的请求

Sentinel 支持通过 `@SentinelResource` 注解定义资源并配置 `blockHandler` 和 `fallback` 函数来进行限流之后的处理。示例:

```
// 原本的业务方法.
@SentinelResource(blockHandler = "blockHandlerForGetUser")
public User getUserById(String id) {
    throw new RuntimeException("getUserById command failed");
}

// blockHandler 函数，原方法调用被限流/降级/系统保护的时候调用
public User blockHandlerForGetUser(String id, BlockException ex) {
    return new User("admin");
}

```

注意 `blockHandler` 函数会在原方法被限流/降级/系统保护的时候调用，而 `fallback` 函数会针对所有类型的异常。请注意 `blockHandler` 和 `fallback` 函数的形式要求https://github.com/alibaba/Sentinel/wiki/注解支持)。

(2) 服务降级配置

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**,避免影响到其他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(默认抛出DegradeException)

- RT

每秒平均响应时间,超过阈值,且时间窗口内通过的请求>=5,两个条件同时满足触发降级,RT最大4900

- 异常比例(/s)

QPS>=5 且异常比例超过与知识,触发降级,时间窗口结束后,关闭降级

- 异常数(/m)

异常数超过阈值是,触发降级,时间窗口结束后,关闭降级

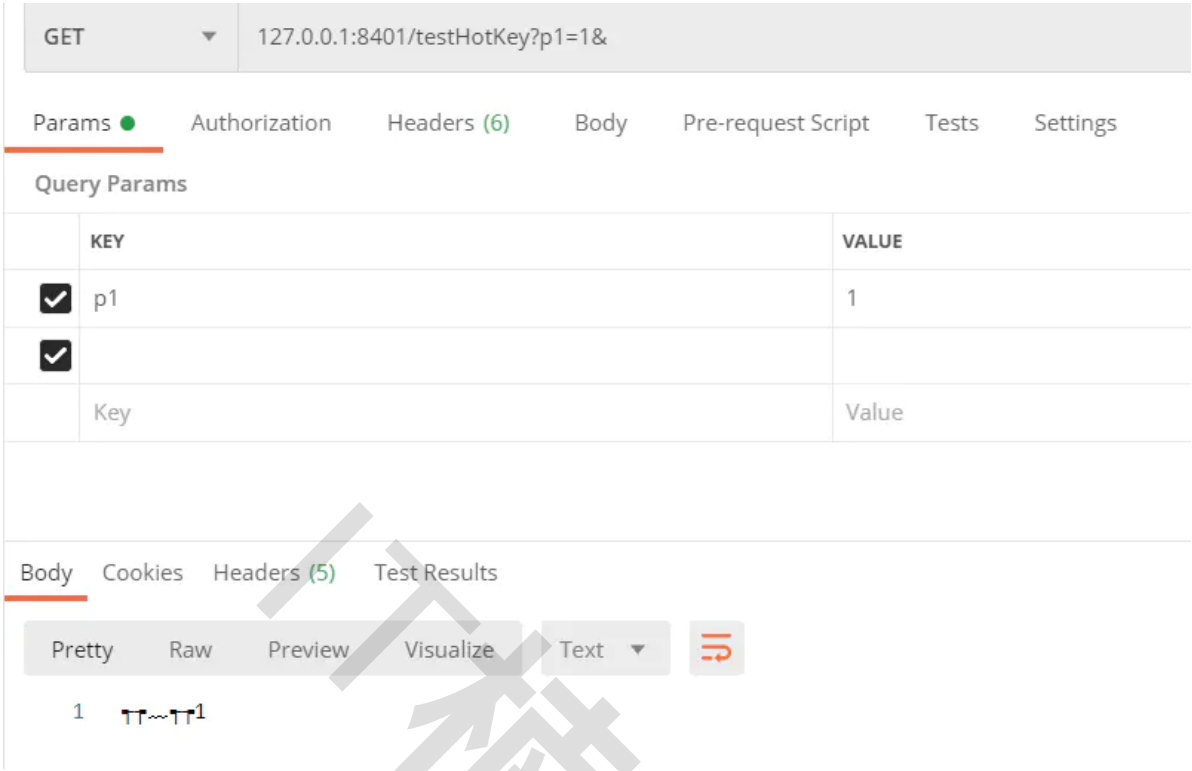
(3) 热点配置

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 K数据,并对其访问进行限制,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.

- 资源名即对应@sentenelResource对应的value
- 参数索引对应方法参数位置,从第几个参数开始检查
- 阈值,窗口就不用解释了

> @sentenelResource (前身HystrixCommond)

```java
@GetMapping("/testHotKey")
@SentinelResource(value = "testHotKey",blockHandler = "delHotKey")
public String testHotKey(@RequestParam(value = "p1",required = false) String p1,@RequestParam(value = "p2",required = false) String p2) {
    return "---testHotKey";
}


public String delHotKey(String p1, String p2, BlockException blockException) {
    return "TT~TT"+p1;
}
```



每个SentenelResource都应该有个对应的兜底方法,来处理出错后的返回,不配blockhandler的话会报page Error

高级玩法

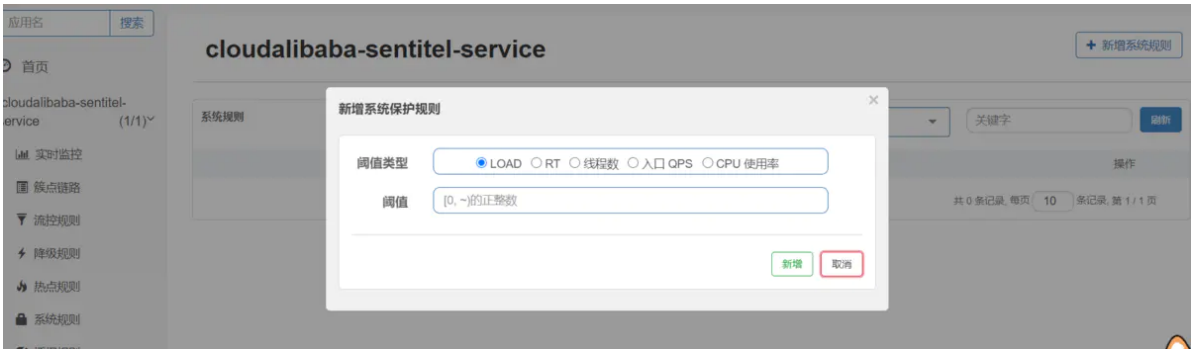
普通我们配置热点参数时候,匹配规则就会返回block,但我们想有特定的参数后,例行放开阈值

当参数等于5时,可以单独配置阈值(必须是基本类型)

注意

@sentinelResource处理的是控制台配置的违规情况,有blockHandler方法进行兜底,但java运行时异常不归他管,这是两个东西(下面有fallback方法处理)

（4）系统规则



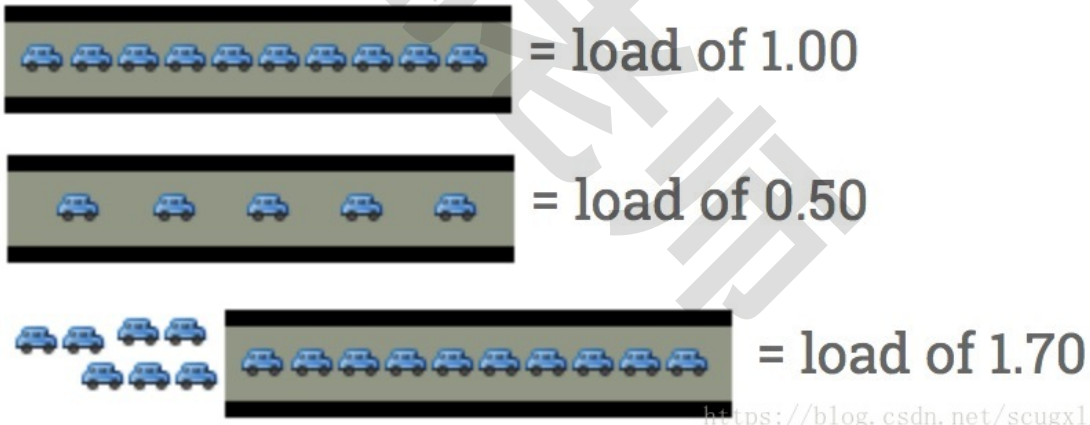
系统规则自然对应所有入口的rest请求做处理,整体管控,设置有风险,需谨慎

你可能对于 Linux 的负载均值（load averages）已有了充分的了解。负载均值在 uptime 或者 top 命令中可以看到，它们可能会显示成这个样子：

load average: 1.84, 1.34, 0.68

很多人会这样理解负载均值：三个数分别代表不同时间段的系统平均负载（一分钟、五 分钟、以及十五分钟），它们的数字当然是越小越好。数字越高，说明服务器的负载越大，这也可能是服务器出现某种问题。

- **0.00 表示桥上上没有任何流量**. 事实上0.0 到1.00 之间的值都表示这条公路上没有任何阻塞,大家都可以及时通过。
- **1.00 表示这座桥刚好达到它的最大容量。** 目前,工作都很正常，但是如果车更多一点，事情就开始变慢了。
- **超过1.00表示已经有排队等待了** 怎么看? 2.00 表示该有2条车道, 一条车道在桥上, 另外一条车道在排队等待. 3.00表示该有3条车道, 一条车道在桥上, 另外两条车道在排队等待. 诸如此类的.



（5）全局自定义限流处理逻辑

在我们上面配置SentinelResource后,有没有发现每个类里面每个方法都要配置一个blockhandle,又麻烦又代码膨胀,那我们有没有一个自定义的全局处理方案呢?

1. 创建CustomerBlockHandler类用于自定义限流处理逻辑
2. 定义公共返回值
3. 创建一个或多个对应blockhandler方法

```
public class CustomerBlockHandler {


    public static CommonResult<String> BlocHandler1(BlockException blockException) {
        return new CommonResult(500,"自定义block1");
    }


    public static CommonResult<String> BlocHandler2(BlockException blockException) {
        return new CommonResult(500,"自定义block2");
    }

}
```

配置在我们需要保障的方法内

```
@GetMapping("/testB")
@SentinelResource(value = "testB",blockHandlerClass = CustomerBlockHandler.class,blockHandler = "BlocHandler1")
public CommonResult<String> testB() {
    return new CommonResult(200,"testB");
}
```

其实还有一种保障方法,可以用代码级别来控制,类似try catch,但我不推荐,写起来又长又麻烦,感兴趣的可以去官网查找写法

(6) 服务熔断Fallback

前面说的SentinelResouce,只会管理我们的服务限流级别的规则采取措施,若java代码级别的RuntimeException是不归他管的,该怎么报错还怎么报错,这里就要提一个fallback参数

> 新建全局异常处理类

```
public class CustomerFallBackHandler {

    public static CommonResult resultException(Throwable throwable) {
        return new CommonResult(500,"运行异常回调处理");
    }
}
```

> 配置要保障的方法

```
@GetMapping("/testA")
@SentinelResource(value = "testA", fallbackClass = {CustomerFallBackHandler.class}, fallback = "resultException")
public CommonResult testA() {
    int a = 1 / 0;
    return new CommonResult(200, "testA");
}
```

可以看到,原理其实跟block差不多,但细节是如果两个都配置的话,并且两种异常都触发的情况下,谁先触发就返回谁的回调,正常情况下限流会大于运行异常的

> Sentinel与openFeign整合

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

This is a simple usage of `FeignClient`:

```
@FeignClient(name = "service-provider", fallback =
EchoServiceFallback.class, configuration = FeignConfiguration.class)
public interface EchoService {
    @GetMapping(value = "/echo/{str}")
    String echo(@PathVariable("str") String str);
}

class FeignConfiguration {
    @Bean
    public EchoServiceFallback echoServiceFallback() {
        return new EchoServiceFallback();
    }
}

class EchoServiceFallback implements EchoService {
    @Override
    public String echo(@PathVariable("str") String str) {
        return "echo fallback";
    }
}
```


（7）代码定义规则

其实每一个规则都能够使用代码来确定，举个例子

通过代码定义流量控制规则

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);
}
```

（8）规则持久化

问题: 在我们配置好规则后,每次重启微服务,我们先配置好的规则就都消失了

方案: 将限流配置规则持久化进Nacos保存,只要刷新某个rest地址,sentinel控制台的流控规则就能看到,只要Nacos里面的配置不删除,针对于微服务的流控规则持续有效

解决

1. 添加maven坐标,将规则持久化进nacos

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

1. 修改yml,将nacos配置文件对应

```
spring.cloud.sentinel.datasource.ds1.nacos.data-id=provider-sentinel
spring.cloud.sentinel.datasource.ds1.nacos.group-id=DEFAULT_GROUP
spring.cloud.sentinel.datasource.ds1.nacos.data-type=json
spring.cloud.sentinel.datasource.ds1.nacos.rule-type=flow
```

1. 配置nacos文件

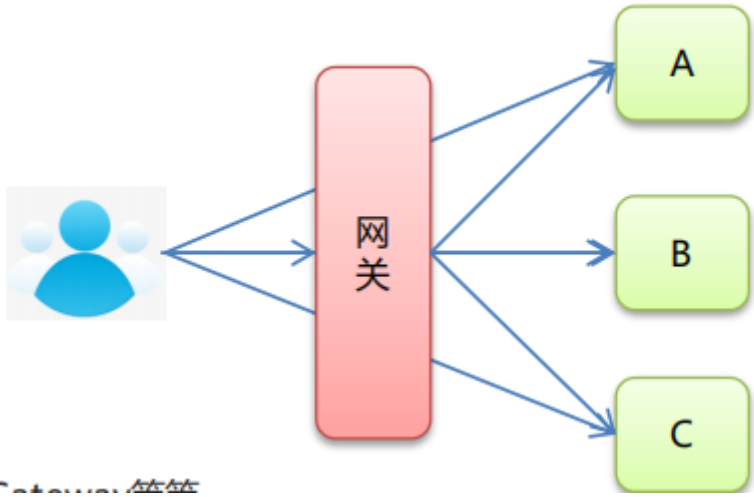
```
[
    {
        "resource":"testB",
        "limitAPP":"default",
        "grade":1,
        "count":1,
        "strategy":0,
        "controlBehavior":0,
        "clusterMode":false
    }

]
```

- resource: 资源名称
- limitApp: 来源应用
- grade: 阈值类型, 0表示线程数,1表示QPS
- count: 单机阈值
- strategy: 流控模式,0表示直接,1表示关联,2表示链路
- controlBehavior: 流控效果,0表示快速失败,1表示WarmUp,2表示排队等待
- clusterMode: 是否集群

依次对应后重启我们的微服务,就会看到我们的持久化配置,只要Nacos存在,就会一直生效

第八章 服务网关

一、概述



• 网关旨在为微服务架构提供一种简单而有效的统一的API路由管理方式。

• 在微服务架构中，不同的微服务可以有不同的网络地址，各个微服务之间通过互相调用完成用户请求，客户端可能通过调用N个微服务的接口完成一个用户请求。

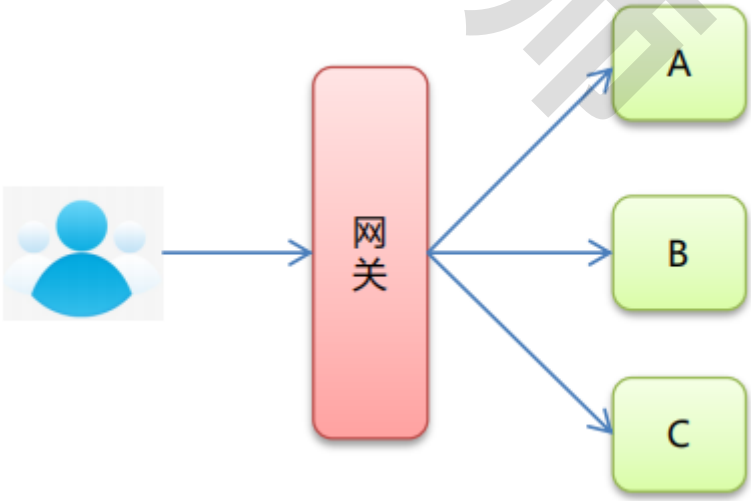
存在的问题：

```
客户端多次请求不同的微服务，增加客户端的复杂性
认证复杂，每个服务都要进行认证
http请求不同服务次数增加，性能不高
```

• 网关就是系统的入口，封装了应用程序的内部结构，为客户端提供统一服务，一些与业务本身功能无关的公共逻辑可以在这里实现，诸如认证、鉴权、监控、缓存、负载均衡、流量管控、路由转发等

• 在目前的网关解决方案里，有Nginx+ Lua、Netflix Zuul 、Spring Cloud Gateway等等

二、快速入门



1. 搭建网关模块 api-gateway-server
2. 引入依赖：starter-gateway

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

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

3. 编写启动类

ApiGatewayApp

```java
package cn.itnanls.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class ApiGatewayApp {

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

}
```

4. 编写配置文件

```yaml
sserver:
  port: 8808
spring:
  application:
    name: api-gateway-server
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
    gateway:
      # 路由配置：转发规则
      routes: #集合。
        # id: 唯一标识。默认是一个UUID
        # uri: 转发路径
        # predicates: 条件,用于请求网关路径的匹配规则
        # filters: 配置局部过滤器的
        - id: example
          uri: http://localhost:8001/
          predicates:
            - Path=/goods/**
```

5. 启动测试

发现起不来，发现咱们的cloud版本集成有问题：

```xml
 <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-dependencies</artifactId>
     <version>2.2.2.RELEASE</version>
     <type>pom</type>
     <scope>import</scope>
</dependency>

<!-- spring cloud  Hoxton.SR3-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>Hoxton.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

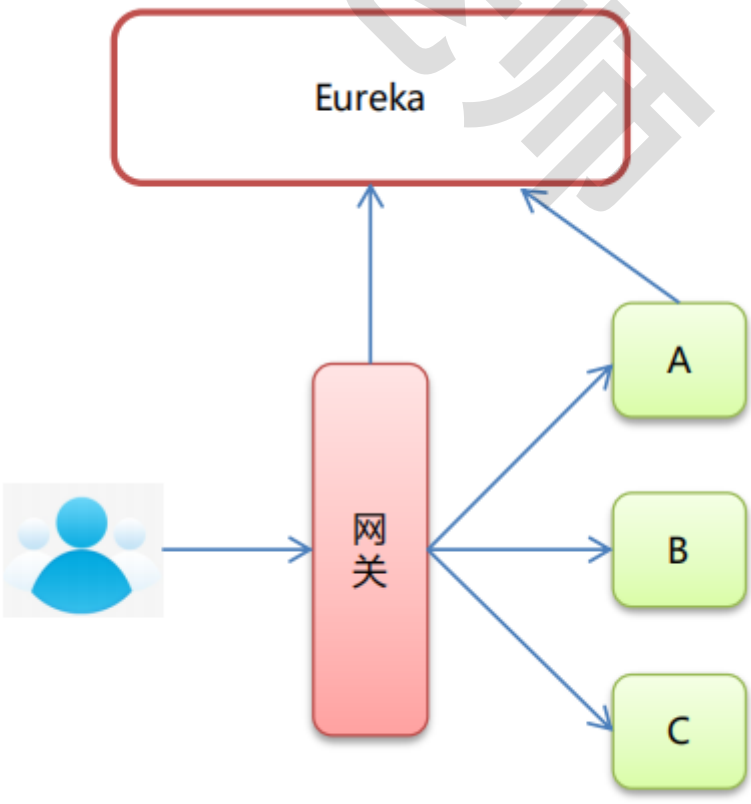
重新启动即可

静态路由

```
uri: http://localhost:8001/
```

动态路由

我们总不能把服务都在配置文件写死呀，我们有注册中心记录着所有的服务。



修改uri属性：uri: lb://服务名称

```
uri: lb://GATEWAY-PROVIDER
```

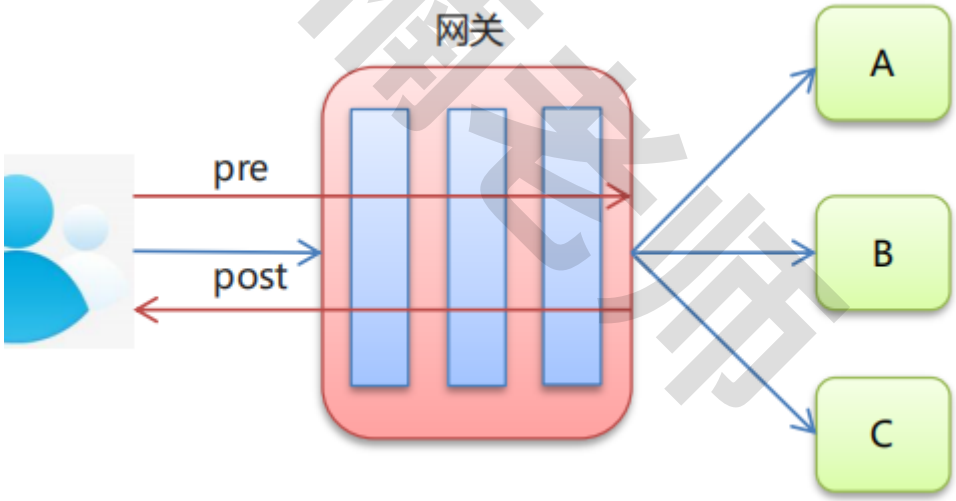
> 微服务名称配置

```
spring:
  cloud:
    # 网关配置
    gateway:
    # 微服务名称配置
      discovery:
        locator:
          enabled: true # 设置为true 请求路径前可以添加微服务名称
          lower-case-service-id: true # 允许为小写
```

三、过滤器

内置过滤器 自定义过滤器

局部过滤器 全局过滤器



- Gateway 支持过滤器功能，对请求或响应进行拦截，完成一些通用操作。
- Gateway 提供两种过滤器方式："pre"和"post"

```
pre 过滤器，在转发之前执行，可以做参数校验、权限校验、流量监控、日志输出、协议转换等。
post 过滤器，在响应之前执行，可以做响应内容、响应头的修改，日志的输出，流量监控等。
```

- Gateway 还提供了两种类型过滤器

```
GatewayFilter：局部过滤器，针对单个路由
GlobalFilter ：全局过滤器，针对所有路由
```

内置过滤器 局部过滤器：

```
filters:
  - AddResponseHeader=foo, bar
```

内置过滤器 全局过滤器： route同级

```
default-filters:
  - AddResponseHeader=xiqinling,lizhiyong
```

拓展：

1、内置的过滤器工厂

这里简单将Spring Cloud Gateway内置的所有过滤器工厂整理成了一张表格。如下：

过滤器工厂	作用	参数
AddRequestHeader	为原始请求添加Header	Header的名称及值
AddRequestParameter	为原始请求添加请求参数	参数名称及值
AddResponseHeader	为原始响应添加Header	Header的名称及值
DedupeResponseHeader	剔除响应头中重复的值	需要去重的Header名称及去重策略
Hystrix	为路由引入Hystrix的断路器保护	`HystrixCommand`的名称
FallbackHeaders	为fallbackUri的请求头中添加具体的异常信息	Header的名称
PrefixPath	为原始请求路径添加前缀	前缀路径
PreserveHostHeader	为请求添加一个preserveHostHeader=true的属性，路由过滤器会检查该属性以决定是否要发送原始的Host	无
RequestRateLimiter	用于对请求限流，限流算法为令牌桶	keyResolver、rateLimiter、statusCode、denyEmptyKey、emptyKeyStatus
RedirectTo	将原始请求重定向到指定的URL	http状态码及重定向的url
RemoveHopByHopHeadersFilter	为原始请求删除IETF组织规定的一系列Header	默认就会启用，可以通过配置指定仅删除哪些Header
RemoveRequestHeader	为原始请求删除某个Header	Header名称
RemoveResponseHeader	为原始响应删除某个Header	Header名称
RewritePath	重写原始的请求路径	原始路径正则表达式以及重写后路径的正则表达式
RewriteResponseHeader	重写原始响应中的某个Header	Header名称，值的正则表达式，重写后的值
SaveSession	在转发请求之前，强制执行`WebSession::save`操作	无
secureHeaders	为原始响应添加一系列起安全作用的响应头	无，支持修改这些安全响应头的值
SetPath	修改原始的请求路径	修改后的路径
SetResponseHeader	修改原始响应中某个Header的值	Header名称，修改后的值

过滤器工厂	作用	参数
SetStatus	修改原始响应的状态码	HTTP 状态码，可以是数字，也可以是字符串
StripPrefix	用于截断原始请求的路径	使用数字表示要截断的路径的数量
Retry	针对不同的响应进行重试	retries、statuses、methods、series
RequestSize	设置允许接收最大请求包的大小。如果请求包大小超过设置的值，则返回 `413 Payload Too Large`	请求包大小，单位为字节，默认值为5M
ModifyRequestBody	在转发请求之前修改原始请求体内容	修改后的请求体内容
ModifyResponseBody	修改原始响应体的内容	修改后的响应体内容
Default	为所有路由添加过滤器	过滤器工厂名称及值

Tips：每个过滤器工厂都对应一个实现类，并且这些类的名称必须以`GatewayFilterFactory`结尾，这是Spring Cloud Gateway的一个约定，例如`AddRequestHeader`对应的实现类为`AddRequestHeaderGatewayFilterFactory`。

1、AddRequestHeader GatewayFilter Factory

为原始请求添加Header，配置示例：

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        filters:
        - AddRequestHeader=X-Request-Foo, Bar
```

为原始请求添加名为`X-Request-Foo`，值为`Bar`的请求头

2、AddRequestParameter GatewayFilter Factory

为原始请求添加请求参数及值，配置示例：

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_parameter_route
        uri: https://example.org
        filters:
        - AddRequestParameter=foo, bar
```

为原始请求添加名为foo，值为bar的参数，即：`foo=bar`

3、AddResponseHeader GatewayFilter Factory

为原始响应添加Header，配置示例：

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        filters:
        - AddResponseHeader=X-Response-Foo, Bar
```

为原始响应添加名为 `X-Request-Foo` ，值为 `Bar` 的响应头

4、DedupeResponseHeader GatewayFilter Factory

DedupeResponseHeader可以根据配置的Header名称及去重策略剔除响应头中重复的值，这是Spring Cloud Greenwich SR2提供的新特性，低于这个版本无法使用。

我们在Gateway以及微服务上都设置了CORS（解决跨域）Header的话，如果不做任何配置，那么请求 -> 网关 -> 微服务，获得的CORS Header的值，就将会是这样的：

```
Access-Control-Allow-Credentials: true, true
Access-Control-Allow-Origin: https://musk.mars, https://musk.mars
```

可以看到这两个Header的值都重复了，若想把这两个Header的值去重的话，就需要使用到DedupeResponseHeader，配置示例：

```
spring:
  cloud:
    gateway:
      routes:
      - id: dedupe_response_header_route
        uri: https://example.org
        filters:
        # 若需要去重的Header有多个，使用空格分隔
        - DedupeResponseHeader=Access-Control-Allow-Credentials Access-
Control-Allow-Origin
```

去重策略：

- RETAIN_FIRST：默认值，保留第一个值
- RETAIN_LAST：保留最后一个值
- RETAIN_UNIQUE：保留所有唯一值，以它们第一次出现的顺序保留

若想对该过滤器工厂有个比较全面的了解的话，建议阅读该过滤器工厂的源码，因为源码里有详细的注释及示例，比官方文档写得还好：
`org.springframework.cloud.gateway.filter.factory.DedupeResponseHeaderGatewayFilterFactory`

- https://cloud.spring.io/spring-cloud-static/Greenwich.SR2/single/spring-cloud.html#fallback-headers)

5、PrefixPath GatewayFilter Factory

为原始的请求路径添加一个前缀路径，配置示例：

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - PrefixPath=/mypath
```

该配置使访问`${GATEWAY_URL}/hello`会转发到`https://example.org/mypath/hello`

8、PreserveHostHeader GatewayFilter Factory

为请求添加一个preserveHostHeader=true的属性，路由过滤器会检查该属性以决定是否要发送原始的Host Header。配置示例：

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: preserve_host_route
        uri: https://example.org
        filters:
        - PreserveHostHeader
```

如果不设置，那么名为`Host`的Header将由Http Client控制

6、RequestRateLimiter GatewayFilter Factory

用于对请求进行限流，限流算法为令牌桶。配置示例：

```yaml
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
```

由于另一篇文章中已经介绍过如何使用该过滤器工厂实现网关限流，所以这里就不再赘述了：

- Spring Cloud Gateway - 扩展

或者参考官方文档：

- RequestRateLimiter GatewayFilter Factory

7、RedirectTo GatewayFilter Factory

将原始请求重定向到指定的Url，配置示例：

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: redirect_route
        uri: https://example.org
        filters:
        - RedirectTo=302, https://acme.org
```

该配置使访问 `${GATEWAY_URL}/hello` 会被重定向到 `https://acme.org/hello` ，并且携带一个 `Location:http://acme.org` 的Header，而返回客户端的HTTP状态码为302

注意事项：

- HTTP状态码应为3xx，例如301
- URL必须是合法的URL，该URL会作为`Location` Header的值

2、Default Filters

Default Filters用于为所有路由添加过滤器工厂，也就是说通过Default Filter所配置的过滤器工厂会作用到所有的路由上。配置示例：

```yaml
spring:
  cloud:
    gateway:
      default-filters:
      - AddResponseHeader=X-Response-Default-Foo, Default-Bar
      - PrefixPath=/httpbin
```

官方文档：https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.1.0.RELEASE/single/spring-cloud-gateway.html#_gatewayfilter_factories

3、局部过滤器

GatewayFilter 局部过滤器，是针对单个路由的过滤器。

在Spring Cloud Gateway 组件中提供了大量内置的局部过滤器，对请求和响应做过滤操作。

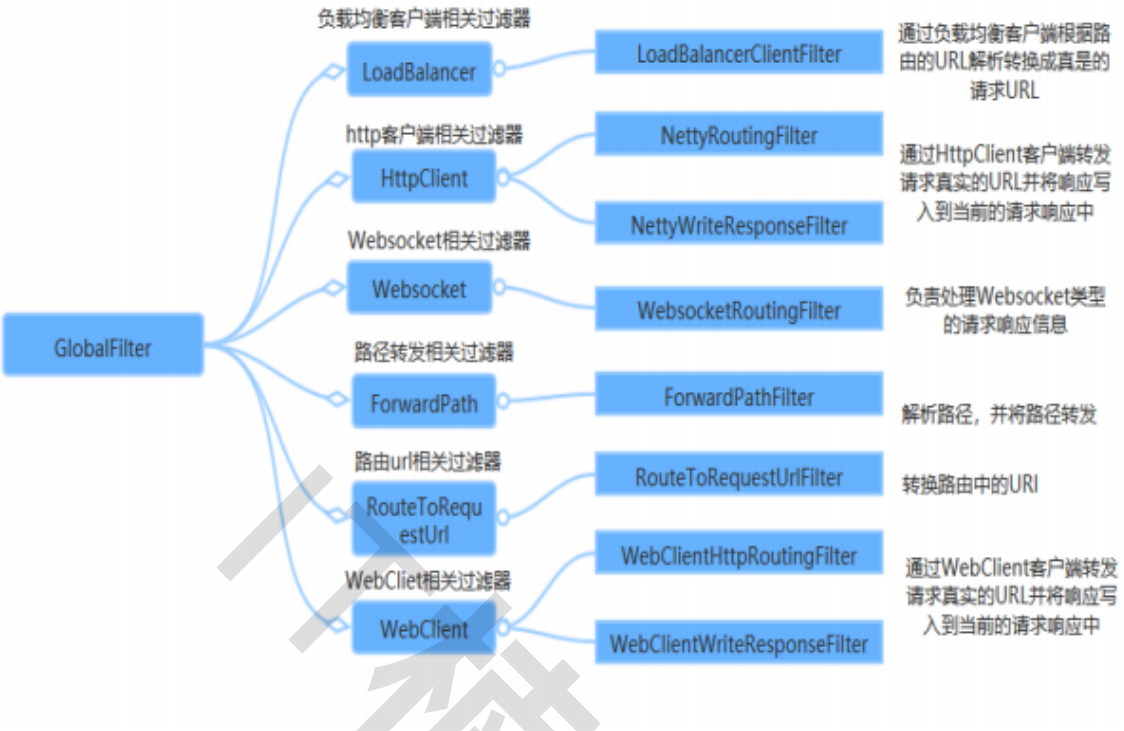
遵循约定大于配置的思想，只需要在配置文件配置局部过滤器名称，并为其指定对应的值，就可以让其生效

```yaml
      - id: gateway-provider
        # 静态路由
        # uri: http://localhost:8001/
        # 动态路由
        uri: lb://GATEWAY-PROVIDER
        predicates:
        - Path=/goods/**
        filters:
        - AddRequestParameter=username,zhangsan
```

4、全局过滤器

自定义 全局过滤器

自定义 局部过滤器



GlobalFilter 全局过滤器，不需要在配置文件中配置，系统初始化时加载，并作用在每个路由上。

Spring Cloud Gateway 核心的功能也是通过内置的全局过滤器来完成。

自定义全局过滤器步骤：

```
1. 定义类实现 GlobalFilter 和 Ordered接口
2. 复写方法
3. 完成逻辑处理
```

5、自定义过滤器

```
package cn.itnanls.gateway.filter;

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

@Component
public class MyFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
```

```
14
15          System.out.println("自定义全局过滤器执行了~~~");
16          return chain.filter(exchange);//放行
17      }
18
19      /**
20       * 过滤器排序
21       * @return 数值越小 越先执行
22       */
23      @Override
24      public int getOrder() {
25          return 0;
26      }
27  }
```

作业：

1使用log4l self4j logbak 分别在项目一中把 info error打出来

2**隔离** 线程池 信号量 默认参数 怎么改

四、集成ribbon负载均衡

```
1  实现原理是在全局LoadBalancerClientFilter中进行拦截，然后再该过滤器中依赖
   LoadBalancerClient loadBalancer，而此负载均衡接口的具体实现是
   RibbonLoadBalancerClient，即spring cloud gateway已经整合好了ribbon，已经可以实现负
   载均衡，我们不需要做任何工作，网关对后端微服务的转发就已经具有负载均衡功能；
```

五、集成Sentinel

```
1  网关集成Sentinel是为了流控熔断降级，具体集成整合步骤如下：
```

1、添加依赖；

```
1  <!-- sentinel-spring-cloud-gateway-adapter -->
2  <dependency>
3      <groupId>com.alibaba.csp</groupId>
4      <artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>
5      <version>1.7.2</version>
6  </dependency>
7
```

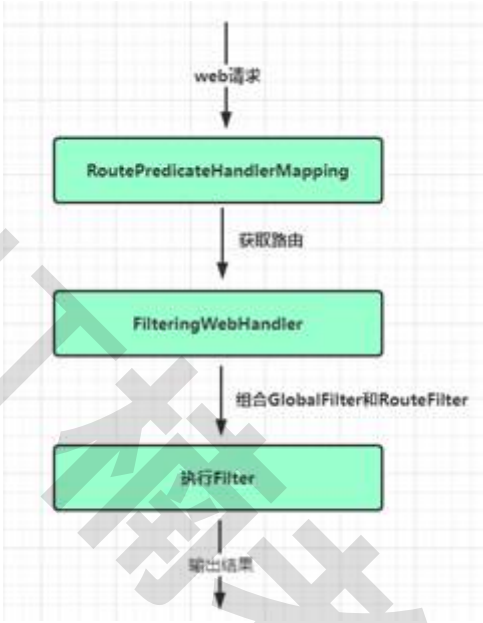
2、添加sentinel控制台配置；

```
1  #sentinel dashboard管理后台
2  spring:
3    cloud:
4      sentinel:
5        eager: true
6        transport:
7          dashboard: 192.168.172.128:8080
```

3、写代码，在spring容器中配置一个Sentinel的全局过滤器；

```
@Bean
@Order(-1)
public GlobalFilter sentinelGatewayFilter() {
    return new SentinelGatewayFilter();
}

```

六、内部流程源码分析



（1）根据自动装配spring-cloud-gateway-core.jar的spring.factories；

```
（2）GatewayClassPathWarningAutoConfiguration检查前端控制器；
（3）网关自动配置GatewayAutoConfiguration；
（4）RoutePredicateHandlerMapping.getHandlerInternal(...)获取Route；
（5）执行FilteringWebHandler

```

第九章 分布式链路跟踪

一、skywalking概述

1. 分布式链路跟踪是分布式系统的应用程序性能监视工具，专为微服务、云原生架构和基于容器（Docker、K8s）架构而设计；
2. 也就是说Skywalking是用于微服务的“跟踪"；
3. 对于一个大型的几十个、几百个微服务构成的微服务架构系统，通常会遇到下面一些问题，比如：
4. 如何串联整个调用链路，快速定位问题？
5. 如何理清各个微服务之间的依赖关系？

6. 如何进行各个微服务接口的性能分析?
7. 如何跟踪整个业务流程的调用处理顺序?
8. Skywalking提供分布式追踪、服务网格遥测分析、度量聚合和可视化一体化解决方案;
9. Skywalking是国人采用Java开发的，现在已经是apache下的一个等级项目;

Skywalking主要功能特性

1、多种监控手段，可以通过语言探针和service mesh获得监控的数据;

2、支持多种语言自动探针，包括 Java，.NET Core 和 Node.JS;

3、轻量高效，无需大数据平台和大量的服务器资源;

4、模块化，UI、存储、集群管理都有多种机制可选;

5、支持告警;

6、优秀的可视化解决方案;

官网：http://skywalking.apache.org/

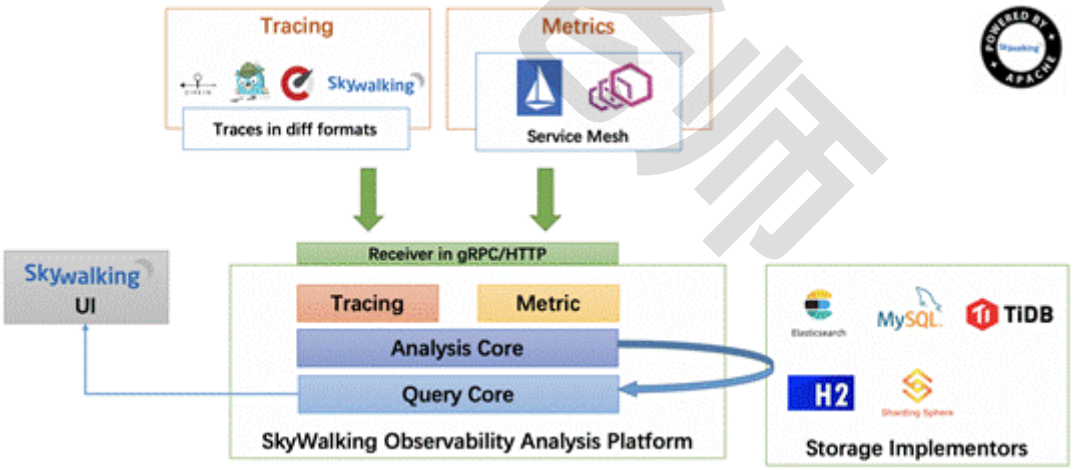
下载：http://skywalking.apache.org/downloads/

Github：https://github.com/apache/skywalking

使用公司：（国内非常多）

https://github.com/apache/skywalking/blob/master/docs/powered-by.md

Skywalking整体架构



整个架构分成四部分：

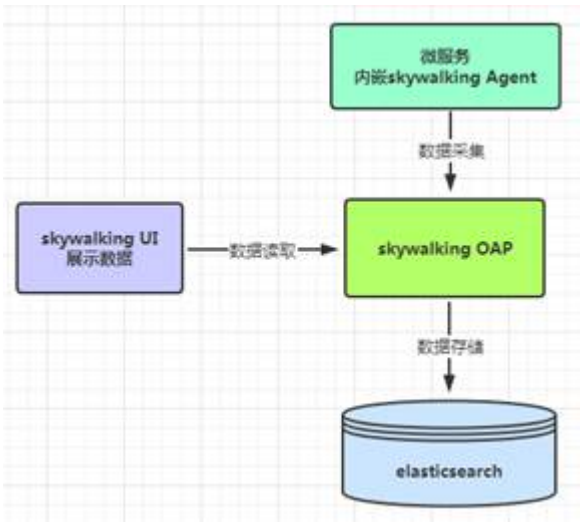
1、上部分Agent：负责从应用中，收集链路信息，发送给 SkyWalking OAP 服务器;

2、下部分 SkyWalking OAP：负责接收Agent发送的Tracing数据信息，然后进行分析(Analysis Core)，存储到外部存储器(Storage)，最终提供查询(Query)功能;

3、右部分Storage：Tracing数据存储，目前支持ES、MySQL、Sharding Sphere、TiDB、H2多种存储器，目前采用较多的是ES，主要考虑是SkyWalking开发团队自己的生产环境采用ES为主;

4、左部分SkyWalking UI：负责提供控台，查看链路等等;

二、SkyWalking 环境搭建部署



1、下载软件包

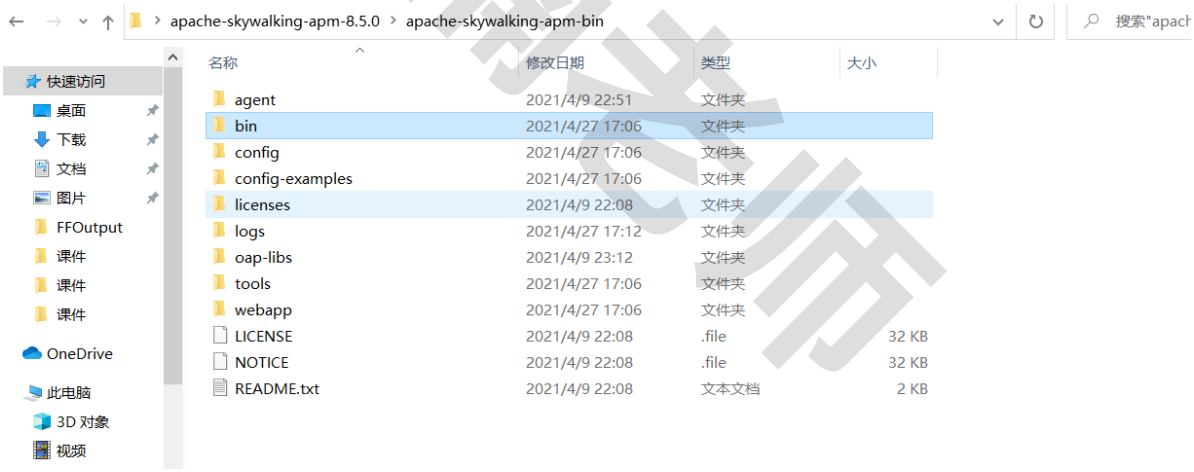
对于 SkyWalking 的软件包，有两种方式获取：

手动编译或者官方包

一般情况下，我们建议使用官方包，手动编译也可以；

从这里下载：http://skywalking.apache.org/downloads/

解压后即完成了安装，不需要做其他操作；



目录说明：

- agent #SkyWalking Agent
- bin #执行脚本
- config #SkyWalking OAP Server 配置文件
- webapp #SkyWalking UI

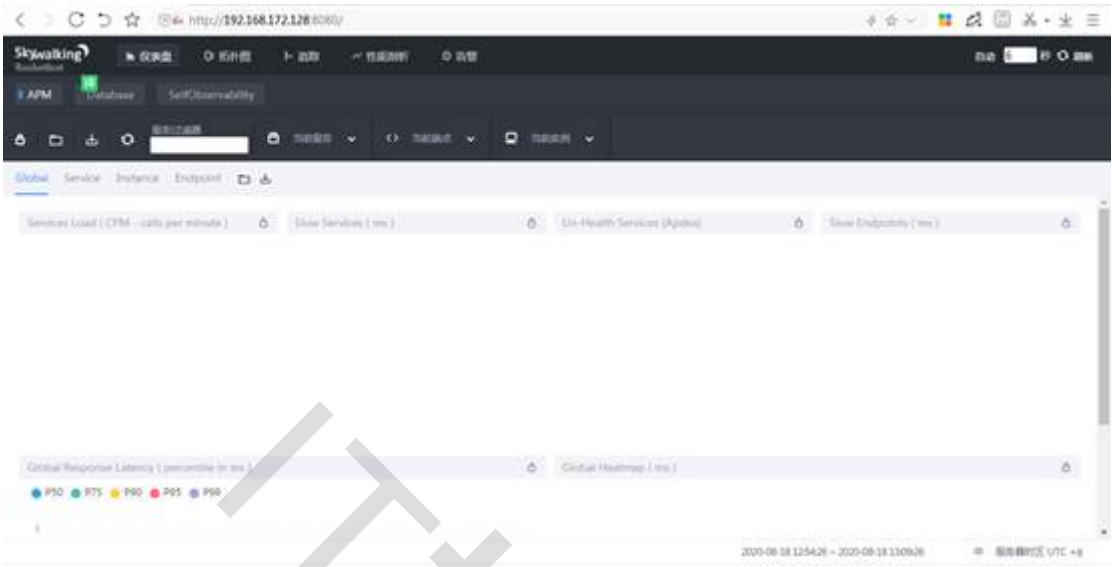
2、启动OAP 服务

1. 切换到bin目录：./startup.sh
2. 启动后会启动两个服务，一个是skywalking-oap-server，一个是skywalking-web-ui；
3. 查看安装目录下的 ./logs 下的日志文件，检查两个服务的日志文件是否启动成功；
4. skywalking-oap-server服务启动后会占用：11800 和 12800 两个端口；
5. skywalking-web-ui服务会占用 8080 端口；

6. 如果想要修改SkyWalking UI服务的参数，可以编辑webapp/webapp.yml 配置文件，比如：
7. server.port：SkyWalking UI服务端口，默认是8080；
8. collector.ribbon.listOfServers：SkyWalking OAP服务地址数组，SkyWalking UI界面的数据是通过请求SkyWalking OAP服务来获得；

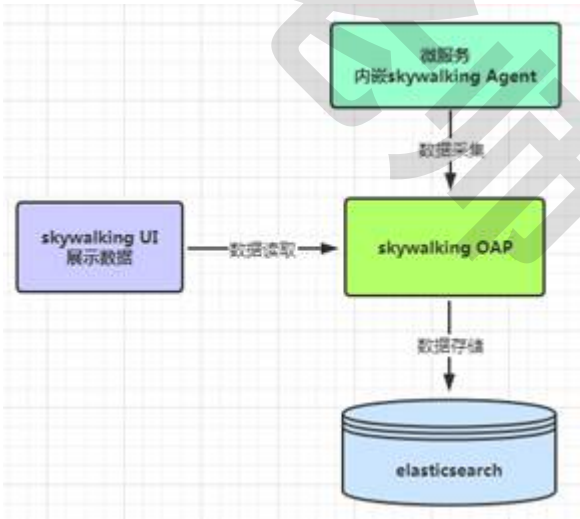
3、访问UI界面

http://192.168.172.128:8080/



页面的右下角可以中英文切换，可以切换选择要展示的时间区间的跟踪数据；

三、SkyWalking Agent跟踪微服务



案例一：linux启动

准备一个springboot程序，打成可执行jar包，写一个shell脚本，在启动项目的Shell脚本上，通过 -javaagent 参数进行配置SkyWalking Agent来跟踪微服务；

```
#!/bin/sh

# SkyWalking Agent配置
```

```
export SW_AGENT_NAME=11-springboot #Agent名字,一般使用`spring.application.name`
export SW_AGENT_COLLECTOR_BACKEND_SERVICES=127.0.0.1:11800 #配置 Collector 地址。
export SW_AGENT_SPAN_LIMIT=2000 #配置链路的最大Span数量，默认为 300。
export JAVA_AGENT=-javaagent:/usr/local/apache-skywalking-apm-bin/agent/skywalking-agent.jar
java $JAVA_AGENT -jar 11-springboot-1.0.0.jar #jar启动
```

在启动程序前加一个-javaagent 参数即可完成对程序的跟踪；

案例二：tomcat中使用

在tomcat中部署war包配置SkyWalking Agent来跟踪微服务；

修改/usr/local/apache-tomcat-9.0.31/bin/catalina.sh 文件，在顶部第一行加上：

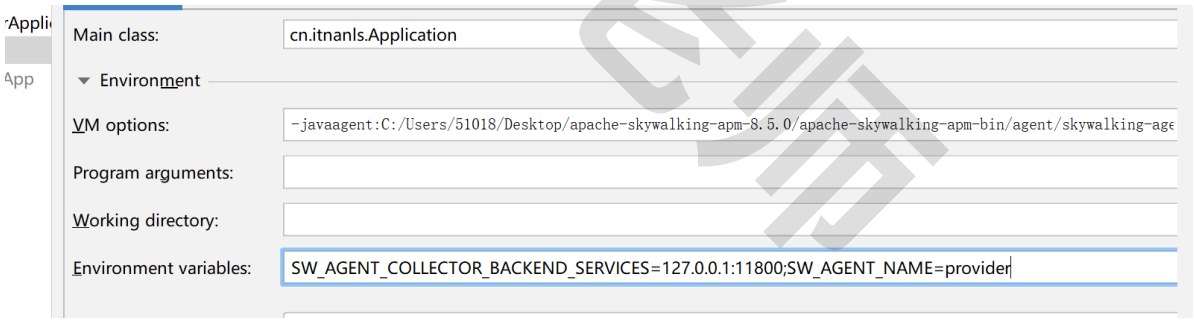
```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:/usr/local/apache-skywalking-apm-bin/agent/skywalking-agent.jar";
export CATALINA_OPTS;
```

如果tomcat端口与skywalking ui端口冲突的话，修改一下tomcat端口；

测试，访问一下项目，然后进入 SkyWalking UI 界面查看跟踪情况，由于上传数据是异步的，访问完项目后，可能需要等几秒才能看到跟踪数据；

案例三：idea中使用

在运行的程序配置jvm参数和环境变量参数，如下图所示：




```
-javaagent:C:/Users/51018/Desktop/apache-skywalking-apm-8.5.0/apache-skywalking-apm-bin/agent/skywalking-agent.jar
SW_AGENT_COLLECTOR_BACKEND_SERVICES=127.0.0.1:11800;SW_AGENT_NAME=provider
```

> SkyWalking中三个概念

服务(Service)：表示对请求提供相同行为的一系列或一组工作负载，在使用Agent时，可以定义服务的名字，我们可以看到 Spring Boot 应用服务为 "provider"，就是我们在环境变量 SW_AGENT_NAME 中所定义的；

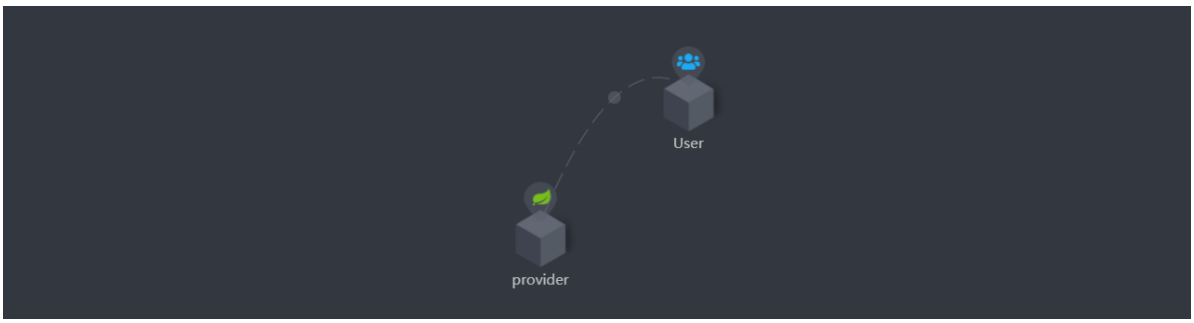
服务实例(Service Instance)： 上述的一组工作负载中的每一个工作负载称为一个实例， 一个服务实例实际就是操作系统上的一个真实进程；

这里我们可以看到 Spring Boot 应用的服务为 {agent_name}-pid:{pid}@{hostname}，由 Agent 自动生成；

端点(Endpoint)： 对于特定服务所接收的请求路径, 如HTTP的URI路径和gRPC服务的类名 + 方法签名；

我们可以看到 Spring Boot 应用的一个端点，为API接口 /index；

跑通一条链路，再看ui



四、ui页面功能

仪表盘：查看被监控服务的运行状态；

拓扑图：以拓扑图的方式展现服务之间的关系，并以此为入口查看相关信息；

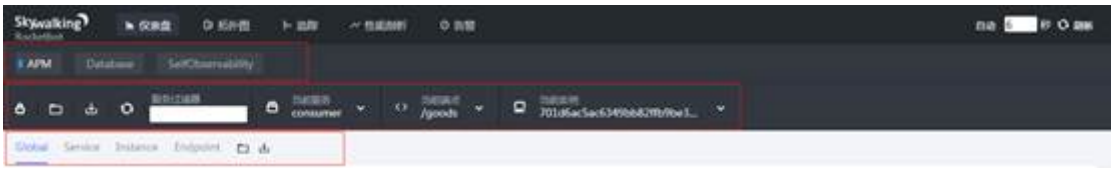
追踪：以接口列表的方式展现，追踪接口内部调用过程；

性能剖析：对端点进行采样分析，并可查看堆栈信息；

告警：触发告警的告警列表，包括服务失败率，请求超时等；

自动刷新：刷新当前页面数据内容；

1、控制栏

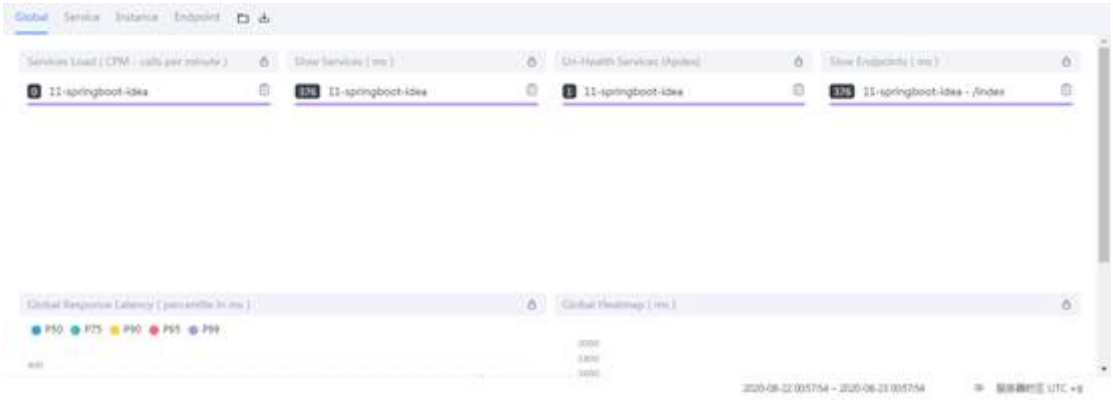


第一栏：不同内容主题的监控面板，应用性能管理/数据库/容器等；

第二栏：操作，包括 编辑/导出当前数据/倒入展示数据/不同服务端点筛选展示；

第三栏：不同纬度展示，全局/服务/实例/端点；

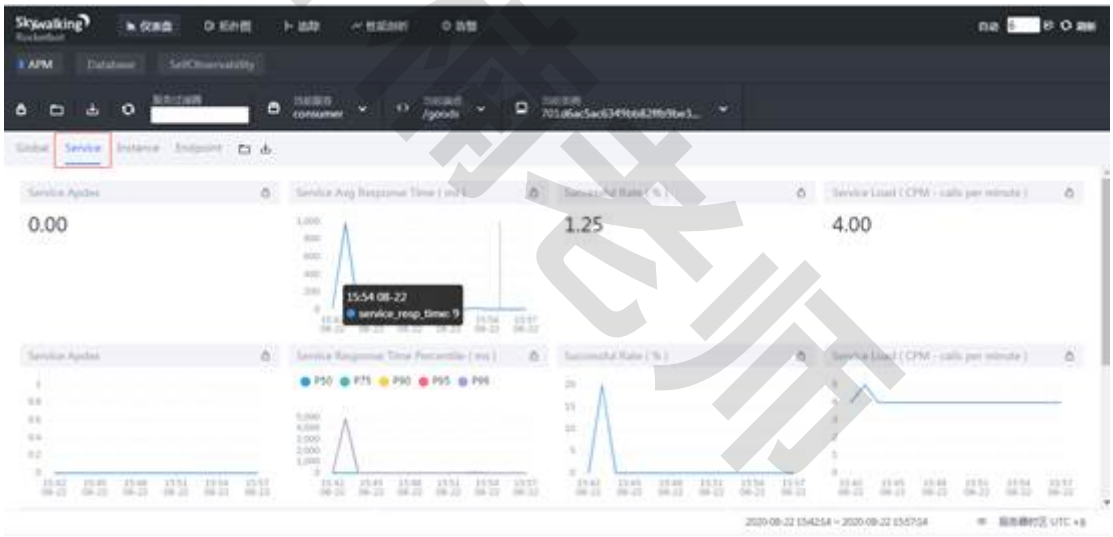
2、展示栏



Global全局维度****

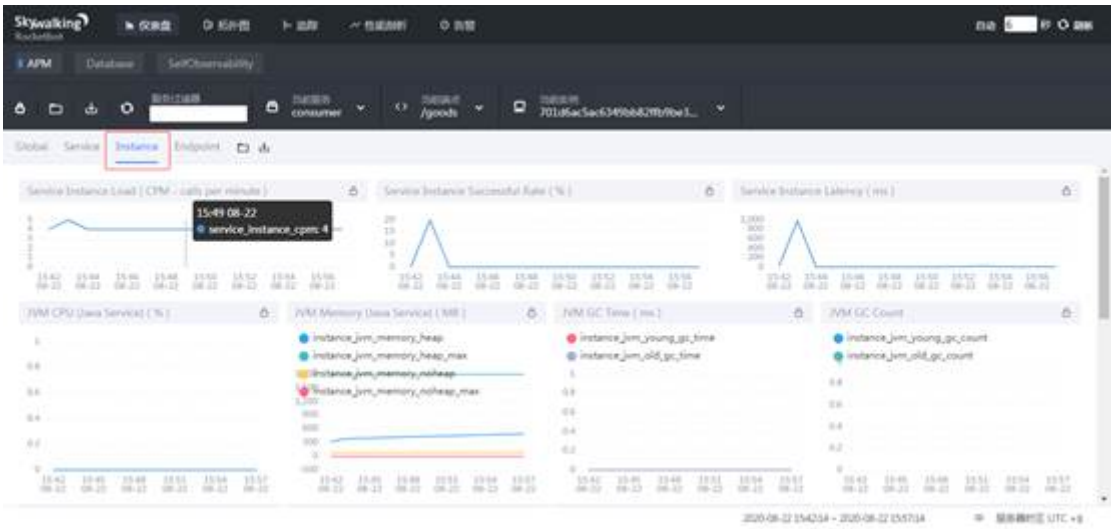
- 第一栏：Global、Service、Instance、Endpoint不同展示面板；
- Services load：服务每分钟请求数；
- Slow Services：慢响应服务，单位ms；
- Un-Health services(Apdex): Apdex性能指标，1为满分；
- Slow Endpoint：慢响应端点，单位ms；
- Global Response Latency：百分比响应延时，不同百分比的延时时间，单位ms；
- Global Heatmap：服务响应时间热力分布图，根据时间段内不同响应时间的数量显示颜色深度；
- 底部栏：展示数据的时间区间，点击可以调整；

3、Service服务维度



- Service Apdex（数字）:当前服务的评分；
- Service Apdex（折线图）：不同时间的Apdex评分；
- Service Avg Response Times：平均响应延时，单位ms；
- Global Response Time Percentile：百分比响应延时；
- Successful Rate（数字）：请求成功率；
- Successful Rate（折线图）：不同时间的请求成功率；
- Servce Load（数字）：每分钟请求数；
- Servce Load（折线图）：不同时间的每分钟请求数；
- Servce Instances Load：每个服务实例的每分钟请求数；
- Show Service Instance：每个服务实例的最大延时；
- Service Instance Successful Rate：每个服务实例的请求成功率；

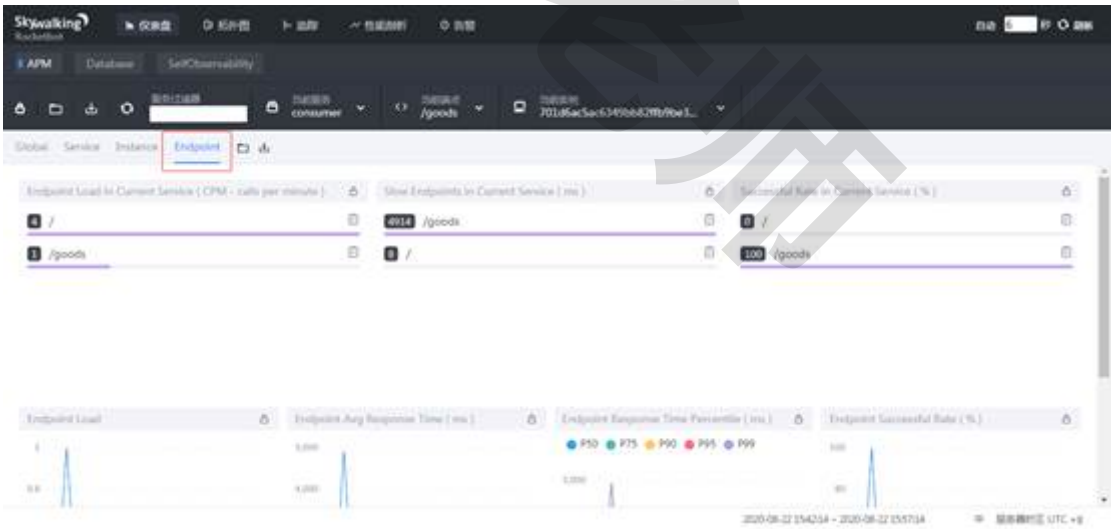
4、Instance实例维度



- Service Instance Load：当前实例的每分钟请求数；
- Service Instance Successful Rate：当前实例的请求成功率；
- Service Instance Latency：当前实例的响应延时；
- JVM CPU：jvm占用CPU的百分比；
- JVM Memory：JVM内存占用大小，单位m；
- JVM GC Time：JVM垃圾回收时间，包含YGC和OGC；
- JVM GC Count：JVM垃圾回收次数，包含YGC和OGC；
- JVM Thread Count：JVM线程数；

还有几个是.NET的，类似于JVM虚拟机，暂时不做说明；

5、Endpoint端点（API）维度



- Endpoint Load in Current Service：每个端点的每分钟请求数；
- Slow Endpoints in Current Service：每个端点的最慢请求时间，单位ms；
- Successful Rate in Current Service：每个端点的请求成功率；
- Endpoint Load：当前端点每个时间段的请求数据；
- Endpoint Avg Response Time：当前端点每个时间段的请求行响应时间；
- Endpoint Response Time Percentile：当前端点每个时间段的响应时间占比；
- Endpoint Successful Rate：当前端点每个时间段的请求成功率；

6、拓扑图

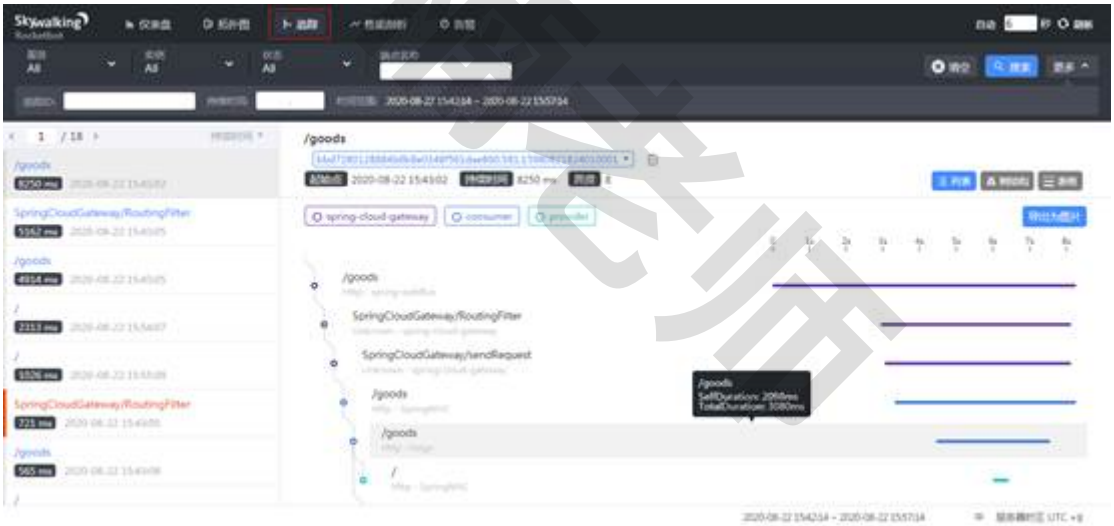
1：选择不同的服务关联拓扑；

2：查看单个服务相关内容；

3：服务间连接情况；

4：分组展示服务拓扑；

7、追踪

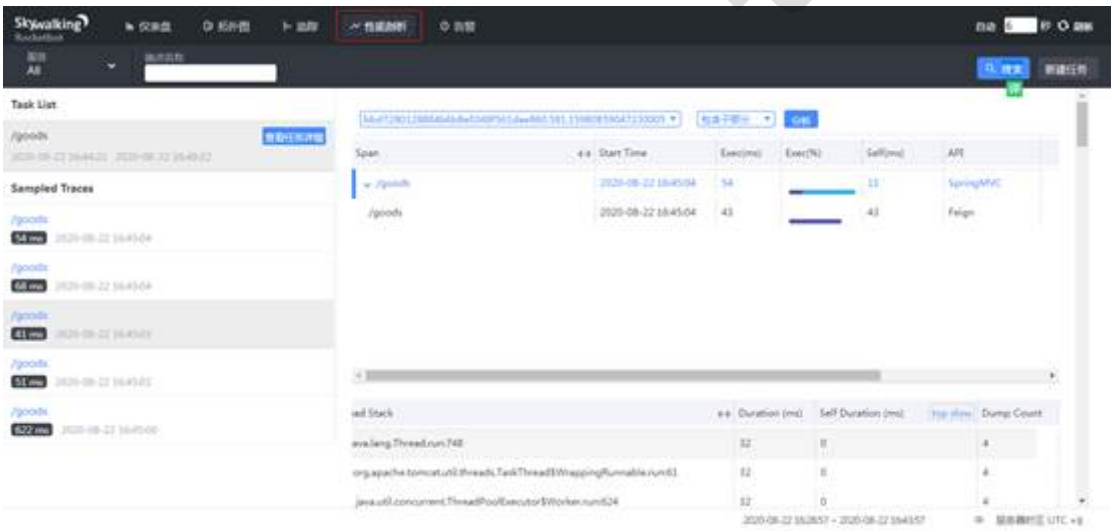


左侧：api接口列表，红色-异常请求，蓝色-正常请求；

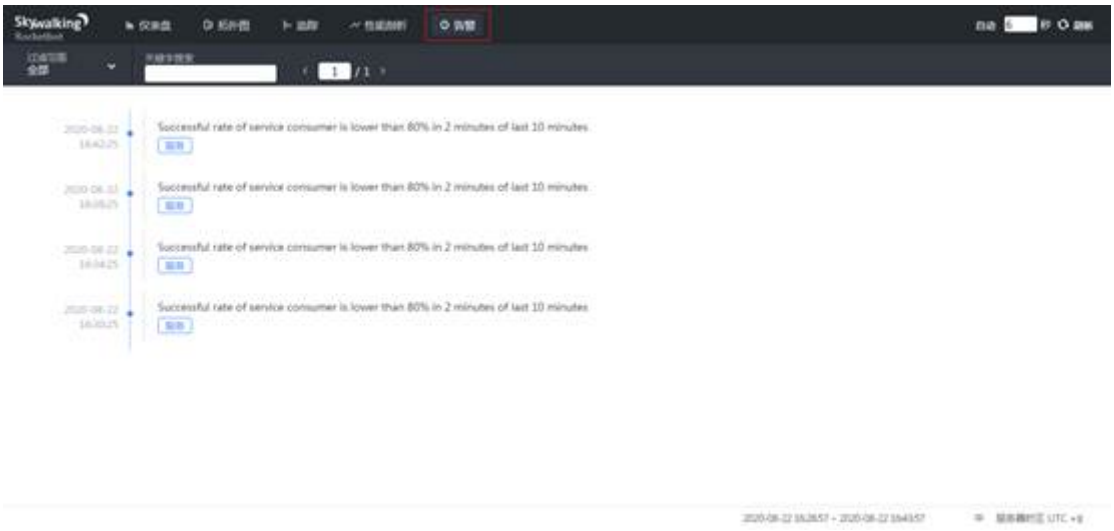
右侧：api追踪列表，api请求连接各端点的先后顺序和时间；

8、性能剖析

- 服务：需要分析的服务；
- 端点：链路监控中端点的名称，可以在链路追踪中查看端点名称；
- 监控时间：采集数据的开始时间；
- 监控持续时间：监控采集多长时间；
- 起始监控时间：多少秒后进行采集；
- 监控间隔：多少秒采集一次；
- 最大采集数：最大采集多少样本；



9、告警



不同维度告警列表，可分为服务、端点和实例；

五、Skywalking告警通知

1、告警配置

skywalking告警的核心由一组规则驱动，这些规则定义在config/alarm-settings.yml文件中，告警规则的定义分为三部分；

1、告警规则：它们定义了应该如何触发度量警报，应该考虑什么条件；

2、网络钩子(Webhook}：当警告触发时，哪些服务终端需要被通知；

3、gRPC钩子：远程gRPC方法的主机和端口，告警触发后调用；

为了方便，skywalking发行版中提供了默认的alarm-setting.yml文件，包括一些规则，每个规则有英文注释，可以根据注释得知每个规则的作用；

比如service_resp_time_rule规则：

```
rules:
  # Rule unique name, must be ended with `_rule`.
  service_resp_time_rule:
    metrics-name: service_resp_time
    op: ">"
    threshold: 1000
    period: 10
    count: 3
    silence-period: 5
    message: Response time of service {name} is more than 1000ms in 3
minutes of last 10 minutes.
```

该规则表示服务{name}的响应时间在最近10分钟的3分钟内超过1000ms；

只有我们的服务请求符合alarm-setting.yml文件中的某一条规则就会触发告警；

2、Webhook回调通知

SkyWalking告警Webhook回调要求接收方是一个Web容器（比如tomcat服务），告警的消息会通过HTTP请求进行发送, 请求方法为POST, Content-Type为application/json, JSON格式基于

修改配置文件：

```
webhooks:
   - http://127.0.0.1:8022/timeout
#  - http://127.0.0.1/go-wechat/
```

List<org.apache.skywalking.oap.server.core.alarm.AlarmMessage>的集合对象数据, 集合中的每个AlarmMessage包含以下信息：

```
1、scopeId. 所有可用的Scope请查阅：
org.apache.skywalking.oap.server.core.source.DefaultScopeDefine；
2、name. 目标 Scope 的实体名称；
3、id0. Scope 实体的 ID；
4、id1. 未使用；
5、ruleName. 您在 alarm-settings.yml 中配置的规则名；
6、alarmMessage. 报警消息内容；
7、startTime. 告警时间，位于当前时间与 UTC 1970/1/1 之间；
```

```
[
    {
        "scopeId":2,
        "scope":"SERVICE_INSTANCE",
        "name":"a33e573c1df2495ca18d4d7239202aa5@169.254.80.114 of
provider",
 "id0":"cHJvdmlkZXI=.1_YTMzZTU3M2MxZGYyNDk1Y2ExOGQ0ZDcyMzkyMDJhYTVAMTY5LjI1N
C44MC4xMTQ=",
        "id1":"",
        "ruleName":"service_instance_resp_time_rule",
        "alarmMessage":"Response time of service instance
a33e573c1df2495ca18d4d7239202aa5@169.254.80.114 of provider is more than
1000ms in 2 minutes of last 10 minutes",
        "startTime":1619526800214
    },
    {
        "scopeId":6,
        "scope":"ENDPOINT_RELATION",
        "name":"User in User to {GET}/goods/hello in provider",
        "id0":"VXNlcg==.0_VXNlcg==",
        "id1":"cHJvdmlkZXI=.1_e0dFVH0vZ29vZHMvaGVsbG8=",
        "ruleName":"endpoint_relation_resp_time_rule",
        "alarmMessage":"Response time of endpoint relation User in User to
{GET}/goods/hello in provider is more than 1000ms in 2 minutes of last 10
minutes",
        "startTime":1619526800214
    }
]
```

六、Skywalking持久化到mysql

Skywalking跟踪数据默认是存放在内嵌式数据库H2中的，重启skywalking，跟踪数据就丢失了，我们可以把跟踪数据持久化到mysql或者elasticsearch中

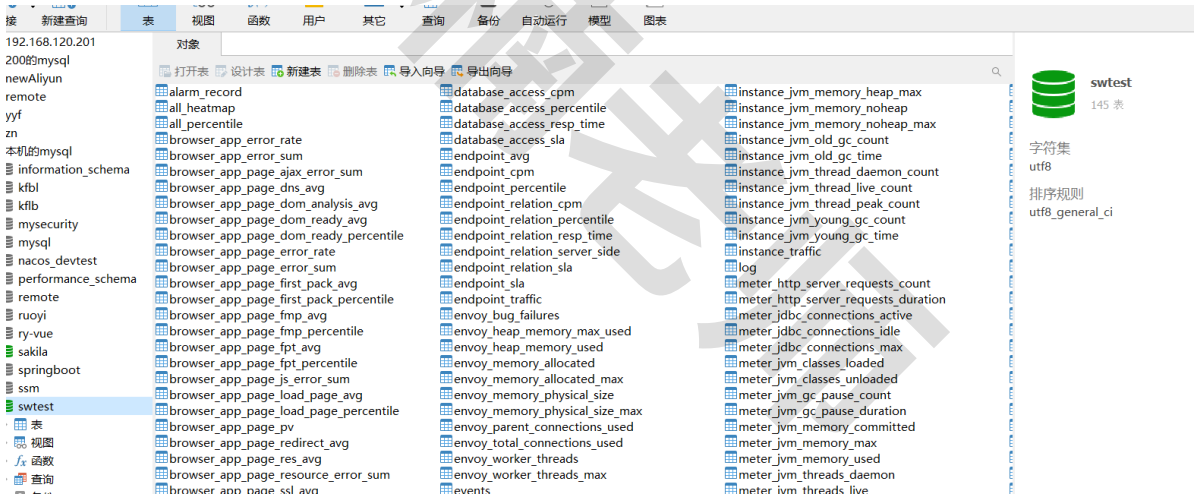
1、修改application.yml配置文件：

```
storage:
 selector: ${SW_STORAGE:mysql}
mysql:
 properties:
   jdbcUrl: ${SW_JDBC_URL:"jdbc:mysql://localhost:3306/swtest?useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai"}
   dataSource.user: ${SW_DATA_SOURCE_USER:root}
   dataSource.password: ${SW_DATA_SOURCE_PASSWORD:root}
```

2、启动skywalking

./startup.sh

3、启动应用程序，查看跟踪数据是否已经持久化到mysql的索引中，数据库中已经存在大量的表



4、然后重启skywalking，验证跟踪数据会不会丢失；