

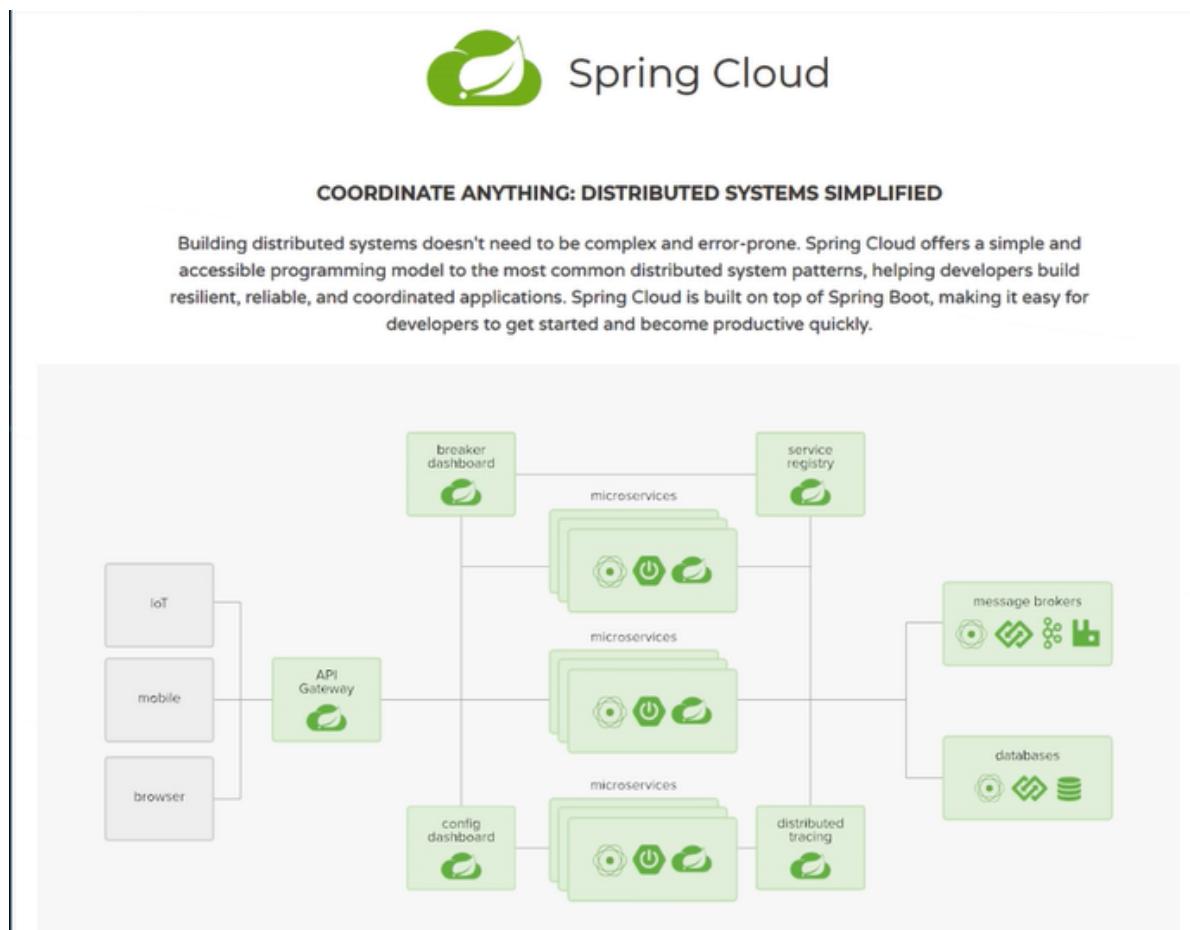
基础概念

01、微服务架构

微服务架构师一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的Restful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

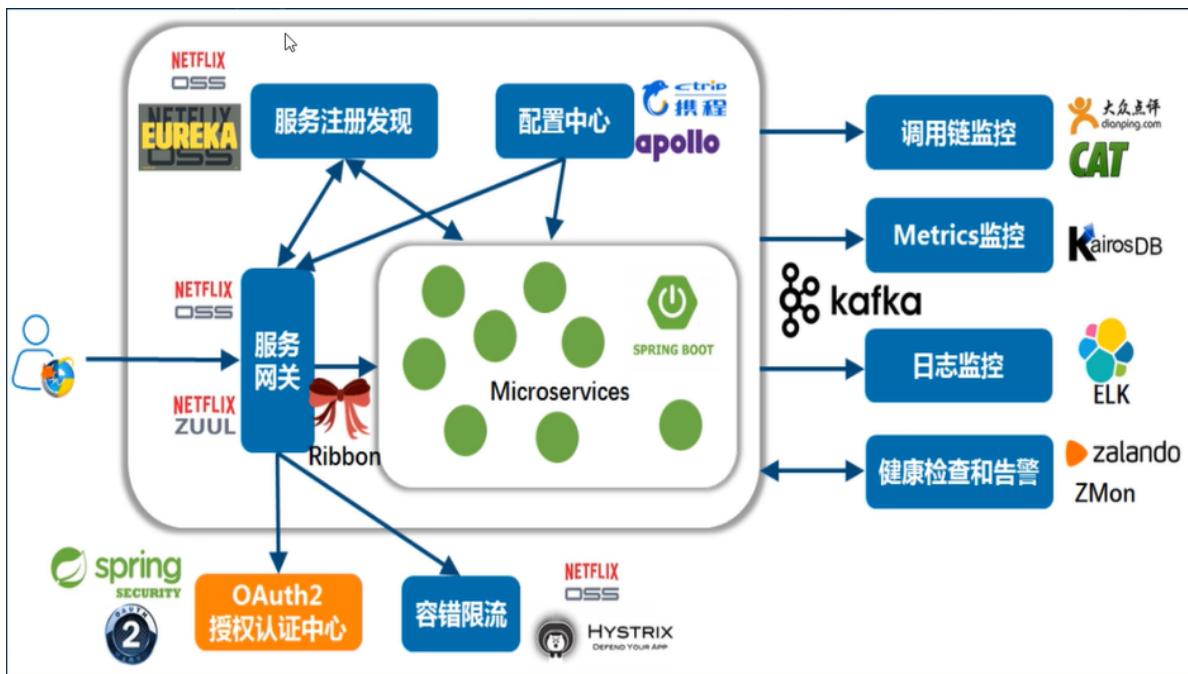
02、SpringCloud

SpringCloud=分布式微服务架构的一站式解决方案，是多种微服务架构落地技术的集合体，速成微服务全家桶

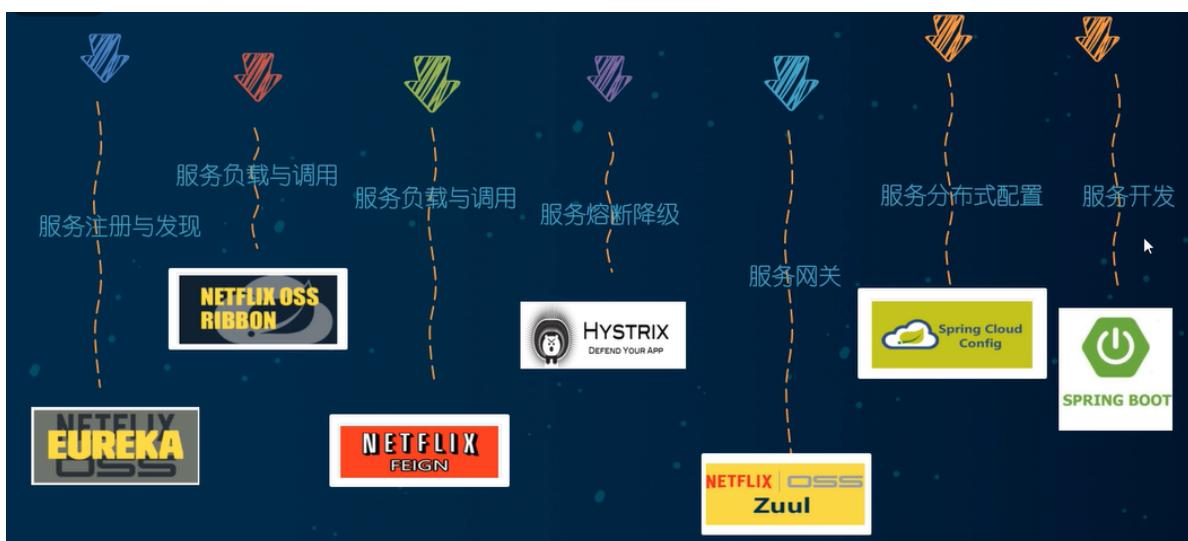




03、SpringCloud 技术栈



具体实现用到的技术



04、boot和cloud的版本选型

Spring Cloud采用了英国伦敦地铁站的名称来命名，并由地铁站名称字母A-Z依次类推的形式来发布迭代版本

SpringCloud是一个由许多子项目组成的综合项目，各子项目有不同的发布节奏。为了管理SpringCloud与各子项目的版本依赖关系，发布了一个清单，其中包括了某个SpringCloud版本对应的子项目版本。为了避免SpringCloud版本号与子项目版本号混淆，SpringCloud版本采用了名称而非版本号的命名，这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序。例如Angel是第一个版本，Brixton是第二个版本。当SpringCloud的发布内容积累到临界点或者一个重大BUG被解决后，会发布一个“service releases”版本，简称SRX版本，比如Greenwich.SR2就是SpringCloud发布的Greenwich版本的第2个SRX版本。

cloud Hoxton.SR1

boot 2.2.2.RELEASE

cloud alibaba 2.1.0.RELEASE

java java8

Maven 3.5及以上

Mysql 5.7及以上

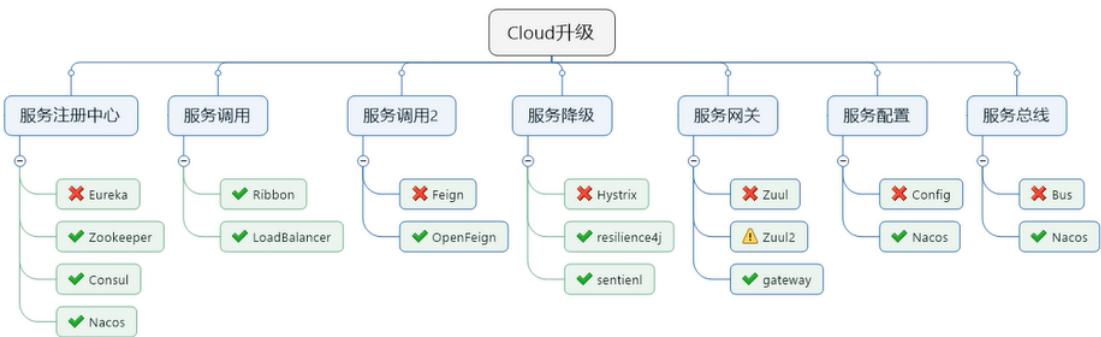
05、cloud 组件停更说明

停更不停用

- 被动修复bugs
- 不再接受合并请求
- 不再发布新版本

cloud升级

- 服务注册中心
 - Eureka停用
 - 可以使用zookeeper作为服务注册中心
 - consul
 - Nacos
- 服务调用
 - Ribbon准备停更,
 - 代替为LoadBalance
- 服务调用2
 - Feign改为OpenFeign
- 服务降级
 - Hystrix停更,改为resilience4j
 - 或者阿里巴巴的Sentinel
- 服务网关
 - Zuul改为gateway
- 服务配置
 - Config改为 Nacos
- 服务总线
 - Bus改为Nacos



环境搭建

1. New Project
2. 聚合总父工程名字
3. Maven选版本
4. 工程名字
5. 字符编码
6. 注解生效激活
7. java 编码版本选8
8. File Type 过滤

01、新建maven父工程

在pom文件里添加

```

1 <properties>
2   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3   <maven.compiler.source>1.8</maven.compiler.source>
4   <maven.compiler.target>1.8</maven.compiler.target>
5   <junit.version>4.12</junit.version>
6   <log4j.version>1.2.17</log4j.version>
7   <lombok.version>1.16.18</lombok.version>
8   <mysql.version>5.1.47</mysql.version>
9   <druid.version>1.1.16</druid.version>
10  <mybatis.spring.boot.version>1.3.0</mybatis.spring.boot.version>
11 </properties>
12
13  <!-- 1、只是声明依赖，并不实际引入，子项目按需声明使用的依赖 -->
14  <!-- 2、子项目可以继承父项目的 version 和 scope -->
15  <!-- 3、子项目若指定了 version 和 scope，以子项目为准 -->
16 <dependencyManagement>
17   <dependencies>
18     <!--spring boot 2.2.2-->
19     <dependency>
20       <groupId>org.springframework.boot</groupId>
21       <artifactId>spring-boot-dependencies</artifactId>
22       <version>2.2.2.RELEASE</version>
23       <type>pom</type>
24       <scope>import</scope>
25     </dependency>
26     <!--spring cloud Hoxton.SR1-->
27     <dependency>
28       <groupId>org.springframework.cloud</groupId>
29       <artifactId>spring-cloud-dependencies</artifactId>
30       <version>Hoxton.SR1</version>

```

```
31         <type>pom</type>
32         <scope>import</scope>
33     </dependency>
34     <!--spring cloud alibaba 2.1.0.RELEASE-->
35     <dependency>
36         <groupId>com.alibaba.cloud</groupId>
37         <artifactId>spring-cloud-alibaba-dependencies</artifactId>
38         <version>2.1.0.RELEASE</version>
39         <type>pom</type>
40         <scope>import</scope>
41     </dependency>
42
43     <dependency>
44         <groupId>mysql</groupId>
45         <artifactId>mysql-connector-java</artifactId>
46         <version>${mysql.version}</version>
47     </dependency>
48     <dependency>
49         <groupId>com.alibaba</groupId>
50         <artifactId>druid</artifactId>
51         <version>${druid.version}</version>
52     </dependency>
53     <dependency>
54         <groupId>org.mybatis.spring.boot</groupId>
55         <artifactId>mybatis-spring-boot-starter</artifactId>
56         <version>${mybatis.spring.boot.version}</version>
57     </dependency>
58     <dependency>
59         <groupId>junit</groupId>
60         <artifactId>junit</artifactId>
61         <version>${junit.version}</version>
62     </dependency>
63     <dependency>
64         <groupId>log4j</groupId>
65         <artifactId>log4j</artifactId>
66         <version>${log4j.version}</version>
67     </dependency>
68     <dependency>
69         <groupId>org.projectlombok</groupId>
70         <artifactId>lombok</artifactId>
71         <version>${lombok.version}</version>
72         <optional>true</optional>
73     </dependency>
74   </dependencies>
75 </dependencyManagement>
76
77 <build>
78   <plugins>
79     <plugin>
80       <groupId>org.springframework.boot</groupId>
81       <artifactId>spring-boot-maven-plugin</artifactId>
82       <configuration>
83         <fork>true</fork>
84         <addResources>true</addResources>
85       </configuration>
86     </plugin>
87   </plugins>
88 </build>
```

dependencyManagement

Maven 使用dependencyManagement元素来提供了一种管理依赖版本号的方式。

通常会在一个组织或者项目的最顶层的父pom中看到。

使用pom.xml 中的dependencyManagement元素能让所有在子项目中引用一个依赖而不用显式的列出版本号。

Maven 会沿着父子层次向上走，直到找到一个拥有dependencyManagement元素的项目，然后它就会使用这个dependencyManagement元素中指定的版本号。

这样做的好处就是:如果有多个子项目都引用同一样依赖，则可以避免在每个使用的子项目里都声明一个版本号，这样当想升级或切换到另一个版本时，只需要在

顶层父容器里更新，而不需要一个一个子项目的修改：另外如果某个子项目需要另外的一个版本，只需要声明version就可。

dependencyManagement里只是声明依赖，并不实现引入，因此子项目需要显示的声明需要用的依赖。

02、创建子模块



1、子模块名字

cloud_provider_payment8001

2、pom依赖

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6   <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-actuator</artifactId>
9   </dependency>
10  <dependency>
11    <groupId>org.mybatis.spring.boot</groupId>
12    <artifactId>mybatis-spring-boot-starter</artifactId>
13  </dependency>
14  <dependency>
15    <groupId>com.alibaba</groupId>
16    <artifactId>druid-spring-boot-starter</artifactId>
17    <version>1.1.10</version>
18  </dependency>
19  <dependency>
```

```

1          <groupId>mysql</groupId>
2          <artifactId>mysql-connector-java</artifactId>
3      </dependency>
4      <dependency>
5          <groupId>org.springframework.boot</groupId>
6          <artifactId>spring-boot-starter-jdbc</artifactId>
7      </dependency>
8      <dependency>
9          <groupId>org.springframework.boot</groupId>
10         <artifactId>spring-boot-devtools</artifactId>
11         <scope>runtime</scope>
12         <optional>true</optional>
13     </dependency>
14     <dependency>
15         <groupId>org.projectlombok</groupId>
16         <artifactId>lombok</artifactId>
17         <optional>true</optional>
18     </dependency>
19 </dependencies>

```

3、创建application.yml

```

1 server:
2   port: 8001
3 spring:
4   application:
5     name: cloud-payment-service
6   datasource:
7     type: com.alibaba.druid.pool.DruidDataSource
#当前数据源操作类型
8     driver-class-name: com.mysql.jdbc.Driver
#mysql驱动
9     url: jdbc:mysql://localhost:3306/db2022?
useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=GMT%2B8
10    username: root
11    password: mysql729
12 mybatis:
13   mapper-locations: classpath:mapper/*.xml
14   type-aliases-package: com.example.springcloud.pojo           # 所有的
entity (pojo) 别名类所在包

```

4、主启动类

```

1 @SpringBootApplication
2 public class PaymentMain8001 {
3     public static void main(String[] args) {
4         SpringApplication.run(PaymentMain8001.class, args);
5     }
6 }

```

5、业务类

1、sql

```
1 CREATE TABLE `payment`  
2 (  
3     `id`      bigint(20) NOT NULL AUTO_INCREMENT COMMENT '主键',  
4     `serial`  varchar(200) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT  
NULL COMMENT '支付流水号',  
5     PRIMARY KEY (`id`) USING BTREE  
6 ) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci COMMENT =  
'支付表' ROW_FORMAT = Dynamic;  
7  
8 INSERT INTO `payment`  
9 VALUES (31, '尚硅谷111');  
10 INSERT INTO `payment`  
11 VALUES (32, 'atguigu002');  
12 INSERT INTO `payment`  
13 VALUES (34, 'atguigu002');  
14 INSERT INTO `payment`  
15 VALUES (35, 'atguigu002');
```

2、实体类

```
1 @Data  
2 @AllArgsConstructor  
3 @NoArgsConstructor  
4 public class Payment implements Serializable {  
5     private Long id;  
6     private String serial;  
7 }
```

3、通用返回类

```
1 @Data  
2 @NoArgsConstructor  
3 @AllArgsConstructor  
4 public class CommonResult<T>{  
5     private Integer code;  
6     private String message;  
7     private T data;  
8  
9     public CommonResult(Integer code, String message){  
10         this(code,message,null);  
11     }  
12 }
```

4、dao层

```
1 @Mapper  
2 public interface PaymentDao {  
3     /**  
4      * 插入操作  
5      * @param payment 实体  
6      * @return 受影响的行数
```

```

7   */
8   int create(Payment payment);
9
10 /**
11  * 通过id查订单
12  * @param id 订单id
13  * @return 订单
14  */
15 Payment getPaymentById(@Param("id") Long id);
16 }

```

5、mapper 配置文件类

在resource下,创建mapper/PaymentMapper.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4 <mapper namespace="com.example.springcloud.dao.PaymentDao">
5   <resultMap id="BaseResultMap"
6     type="com.example.springcloud.pojo.Payment">
7     <id column="id" property="id" jdbcType="BIGINT"/>
8     <id column="serial" property="serial" jdbcType="VARCHAR"/>
9   </resultMap>
10
11   <insert id="create" parameterType="Payment" useGeneratedKeys="true"
12     keyProperty="id">
13     insert into payment(serial) values (#{serial})
14   </insert>
15
16   <select id="getPaymentById" parameterType="Long"
17     resultMap="BaseResultMap">
18     select * from payment where id = #{id}
19   </select>
20
21 </mapper>

```

6、写service和serviceImpl

service

```

1 public interface PaymentService {
2   /**
3    * 插入操作
4    * @param payment 实体
5    * @return 受影响的行数
6    */
7   int create(Payment payment);
8
9   /**
10    * 通过id查订单
11    * @param id 订单id
12    * @return 订单
13    */
14   Payment getPaymentById(@Param("id") Long id);
15 }

```

servicelmpl

```
1 @Service
2 public class PaymentServiceImpl implements PaymentService {
3
4     @Resource
5     private PaymentDao paymentDao;
6
7     @Override
8     public int create(Payment payment) {
9         return paymentDao.create(payment);
10    }
11
12     @Override
13     public Payment getPaymentById(Long id) {
14         return paymentDao.getPaymentById(id);
15     }
16 }
```

7、controller

```
1 @Slf4j
2 @RestController
3 @RequiredArgsConstructor(onConstructor_ = @Autowired)
4 @RequestMapping("/payment")
5 public class PaymentController {
6     private final PaymentService paymentService;
7
8     @PostMapping("/create")
9     public CommonResult create(Payment payment){
10         int result = paymentService.create(payment);
11         log.info("****插入结果*****:"+result);
12         if(result > 0){
13             return new CommonResult(200,"插入数据库成功",result);
14         }
15         return new CommonResult(444,"插入数据库失败",result);
16     }
17
18     @PostMapping("/get/{id}")
19     public CommonResult getPaymentById(@PathVariable("id") Long id){
20         Payment payment = paymentService.getPaymentById(id);
21         log.info("****查询结果*****:"+payment);
22         if(payment != null){
23             return new CommonResult(200,"查询成功",payment);
24         }
25         return new CommonResult(444,"没有对应的记录, 查询id"+id);
26     }
27
28 }
```

03、热部署

1.Adding devtools to your project

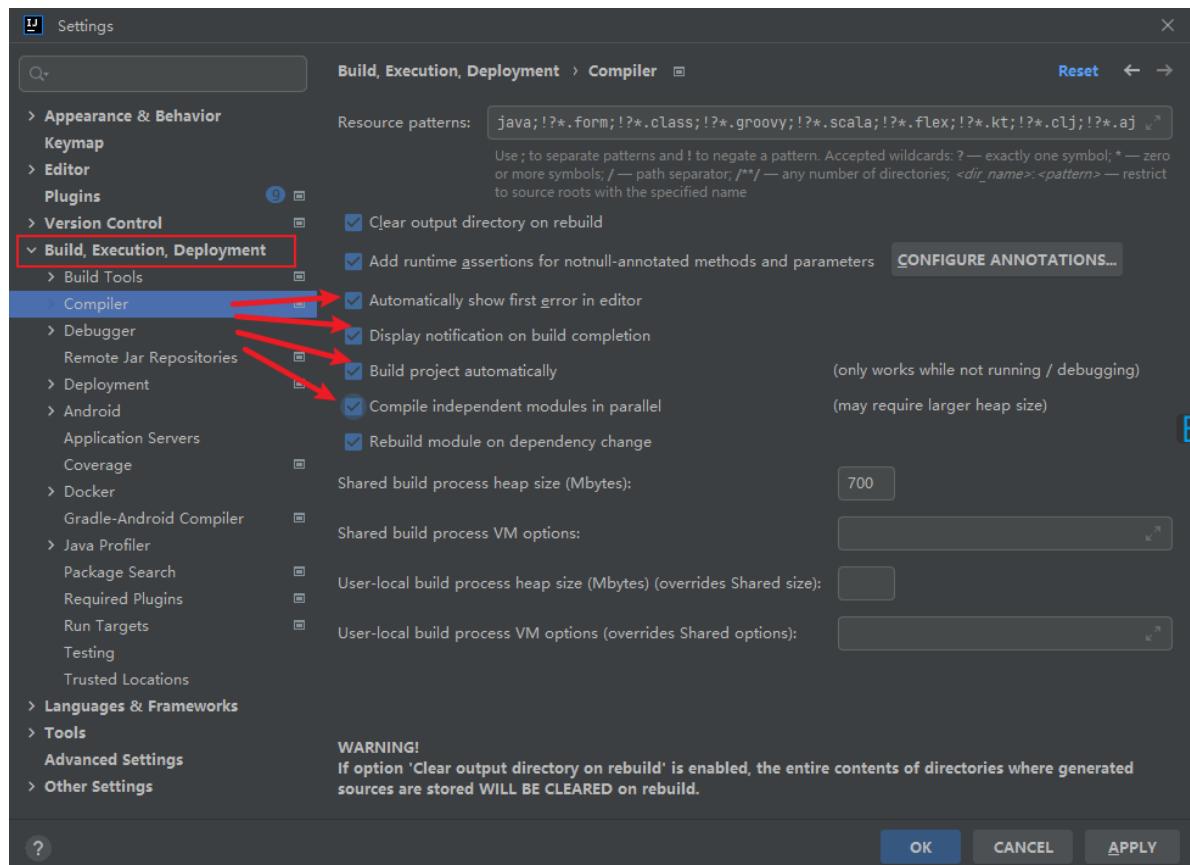
```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-devtools</artifactId>
4   <scope>runtime</scope>
5   <optional>true</optional>
6 </dependency>
```

2.Adding plugin to your pom.xml

父类总工程

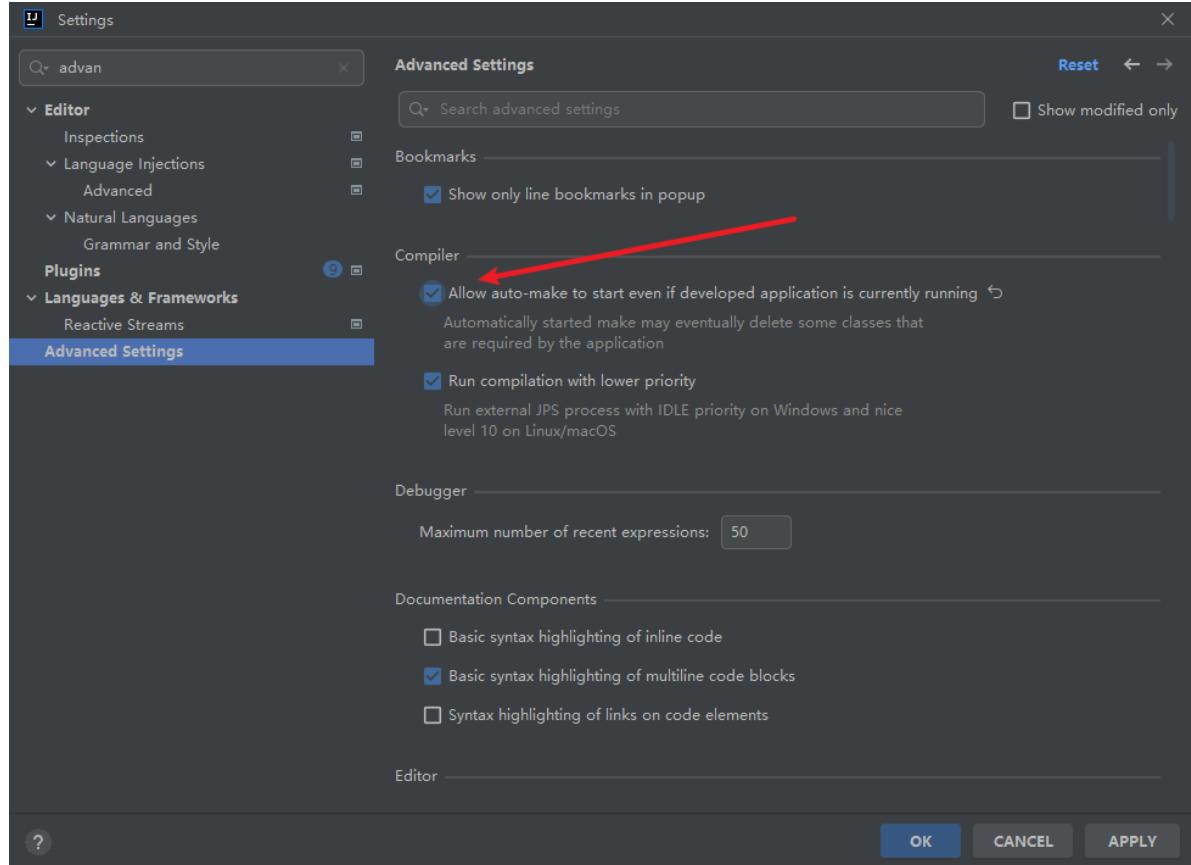
```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-maven-plugin</artifactId>
6       <configuration>
7         <fork>true</fork>
8         <addResources>true</addResources>
9       </configuration>
10      </plugin>
11    </plugins>
12  </build>
```

3.Enabling automatic build



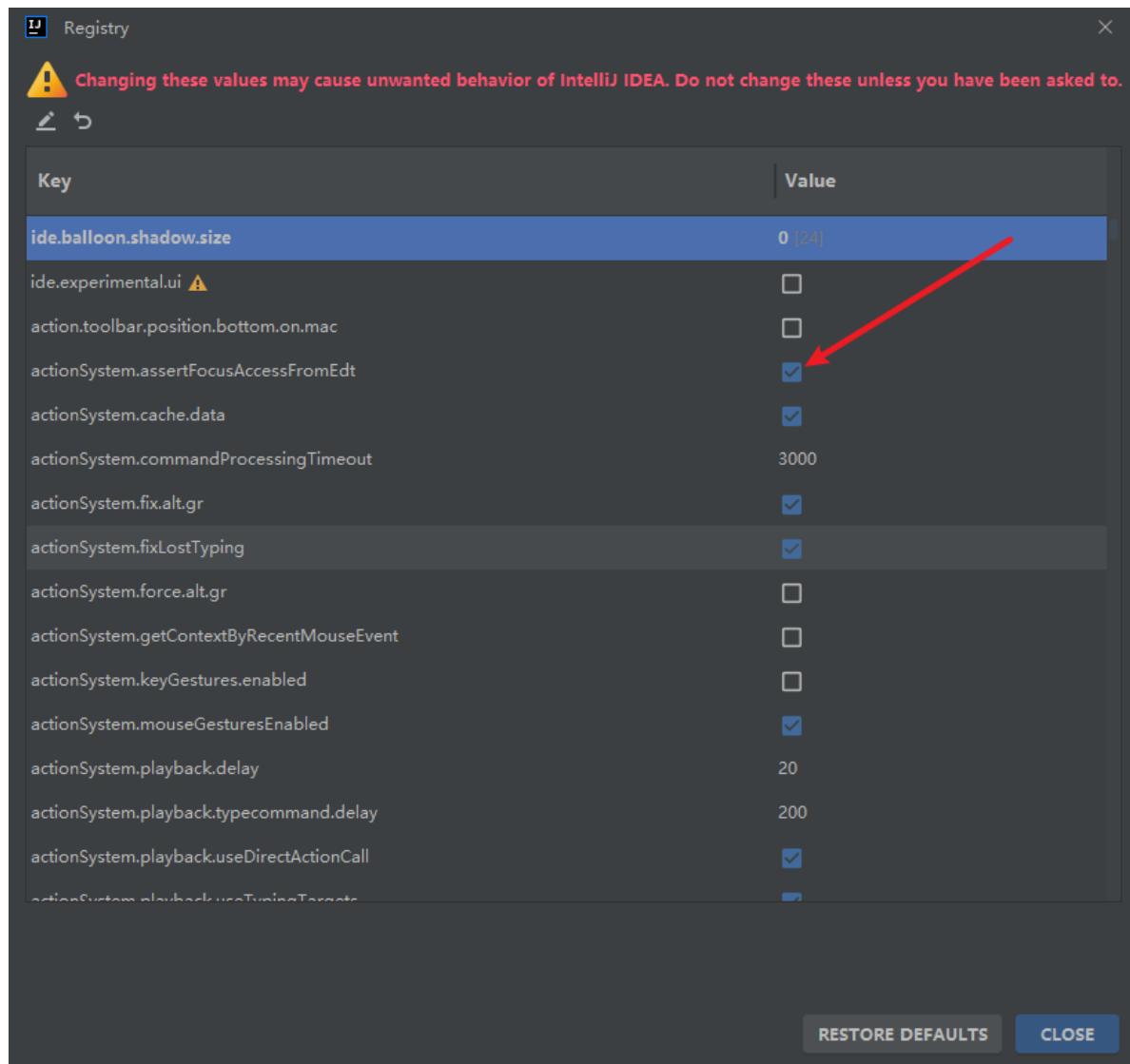
4. Update the value of

在settings里将此勾选



按ctrl+shift+alt+/ 选择register

将此勾选



5.重启idea

04、order模块

1.pom

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-web</artifactId>
5     </dependency>
6     <dependency>
7         <groupId>org.springframework.boot</groupId>
8         <artifactId>spring-boot-starter-actuator</artifactId>
9     </dependency>
10    <dependency>
11        <groupId>org.springframework.boot</groupId>
12        <artifactId>spring-boot-devtools</artifactId>
13        <scope>runtime</scope>
14        <optional>true</optional>
15    </dependency>
16    <dependency>
17        <groupId>org.projectlombok</groupId>
18        <artifactId>lombok</artifactId>
19        <optional>true</optional>
```

```
20      </dependency>
21      <dependency>
22          <groupId>org.springframework.boot</groupId>
23          <artifactId>spring-boot-starter-test</artifactId>
24          <scope>test</scope>
25      </dependency>
26  </dependencies>
```

2.yml配置文件

```
1 server:
2   port: 80
```

3.主启动类

```
1 @SpringBootApplication
2 public class OrderMain80 {
3     public static void main(String[] args) {
4         SpringApplication.run(OrderMain80.class, args);
5     }
6 }
```

4.复制paymant模块的实体类 (pojo)

5.写controller层

因为这里是消费者类,主要是消费,那么就没有service和dao,需要调用pay模块的方法

并且这里还没有微服务的远程调用,那么如果要调用另外一个模块,则需要使用基本的api调用

使用RestTemplate调用pay模块

RestTemplate提供了多种便捷访问远程Http服务的方法,是一种简单便捷的访问restful服务模板类,是Spring提供的用于访问Rest服务的客户端模板工具集

使用:

使用restTemplate访问restful接口非常的简单粗暴无脑。(url, requestMap, ResponseBean.class)这三个参数分别代表REST请求地址、请求参数、HTTP响应转成

被转换成的对象类型。

```
1 @Slf4j
2 @RestController
3 public class OrderController {
4     public static final String PAYMENT_URL = "http://localhost:8001";
5
6     @Resource
7     private RestTemplate restTemplate;
8
9     @GetMapping("/consumer/payment/create")
10    public CommonResult<Payment> create(Payment payment){
11        return
12            restTemplate.postForObject(PAYMENT_URL+"/payment/create", payment,
CommonResult.class);
13    }
14}
```

```
13
14     @GetMapping("/consumer/payment/get/{id}")
15     public CommonResult<Payment> getPayment(@PathVariable("id") Long id){
16         return restTemplate.getForObject(PAYMENT_URL+"/payment/get/"+id,
17             CommonResult.class);
18     }
19 }
```

05、重构

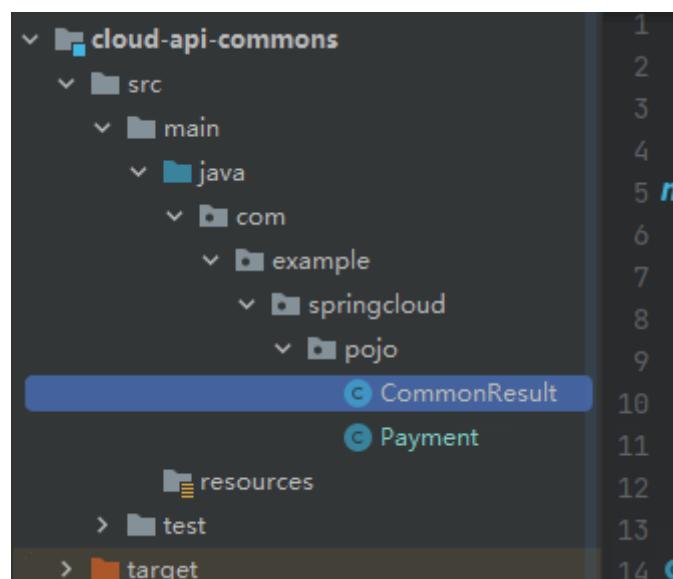
新建一个模块，将重复的代码抽取到一个公共的模块中

1. 创建commons模块

2. 抽取公共的pom

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-devtools</artifactId>
5         <scope>runtime</scope>
6         <optional>true</optional>
7     </dependency>
8
9     <dependency>
10        <groupId>org.projectlombok</groupId>
11        <artifactId>lombok</artifactId>
12        <optional>true</optional>
13    </dependency>
14    <dependency>
15        <groupId>cn.hutool</groupId>
16        <artifactId>hutool-all</artifactId>
17        <version>5.7.22</version>
18    </dependency>
19 </dependencies>
```

3. pojo类放入到commons中



4. 使用maven, 将commons打包 (install)

其他模块删除pojo, 引入commons

```
1 <dependency>
2   <groupId>org.example</groupId>
3   <artifactId>cloud-api-commons</artifactId>
4   <version>1.0-SNAPSHOT</version>
5 </dependency>
```

服务注册与发现

前面我们没有服务注册中心,也可以服务间调用,为什么还要服务注册?

当服务很多时,单靠代码手动管理是很麻烦的,需要一个公共组件,统一管理多服务,包括服务是否正常运行,等

Eureka用于**服务注册**,目前官网已经停止更新

1、Eureka

服务治理

Spring Cloud封装了Netflix公司开发的Eureka模块来实现服务治理

在传统的rpc远程调用框架中,管理每个服务与服务之间依赖关系比较复杂,管理比较复杂,所以需要使用服务治理,管理服务于服务之间依赖关系,可以实现

服务调用、负载均衡、容错等,实现服务发现与注册。

服务注册

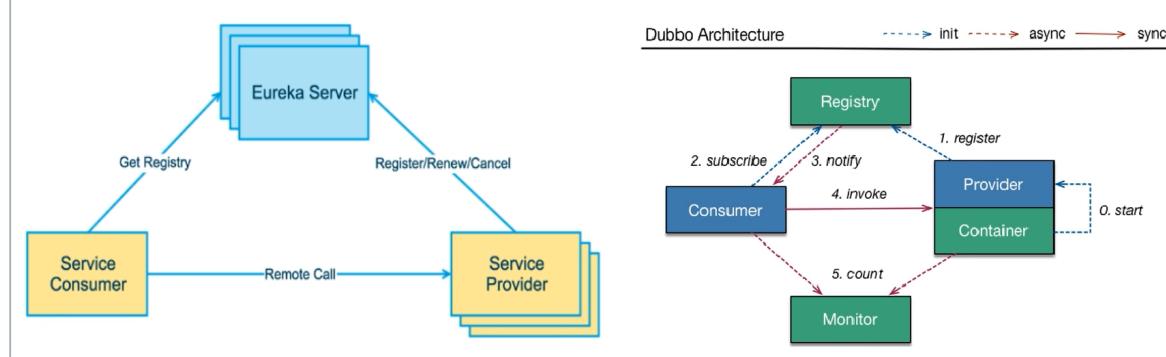
Eureka采用了CS的设计架构, Eureka sever作为服务注册功能的服务器,它是服务注册中心。而系统中的其他做服务,使用eureka的客户端连接到Eureka

Server并维持心跳连接。这样系统的维护人员就可以通过Eureka Server来监控系统中各个微服务是否正常运行。

在服务注册与发现中,有一个注册中心。当服务器启动的时候,会把当前自己服务器的信息,比如:服务地址通讯地址等以别名方式注册到注册中心上。另一方(消费者 | 服务提供者),以该别名的方式去注册中心上获取到实际的服务通讯地址,然后再实现本地RPC调用

RPC远程调用框架核心设计思想:在于注册中心,因为使用注册中心管理每个服务与服务之间的一个依赖关系(服务治理概念)。在任何rpc远程框架中,都会有一个注册中心(存放服务地址相关信息(接口地址))

下左图是Eureka系统架构,右图是Dubbo的架构,请对比



Eureka的两个组件:

Eureka Server

Eureka Server 提供服务注册服务

各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。

Eureka Client

Eureka Client 通过注册中心进行访问

是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认周期为30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除 (默认90秒)

单机版

EurekaServer端服务注册中心类似物业中心

EurekaClient端cloud-provider-payment8001将注册进EurekaServer 成为服务提供者provider

EurekaClient端cloud-consumer-order80将注册进EurekaServer 成为服务消费者consumer

1.创建cloud-eureka-server7001

2引入pom依赖

eureka最新的依赖变了

1.X和2.X的对比说明

以前的老版本 (当前使用2018)

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

现在新版本 (当前使用2020.2)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

3.配置文件

```

1 server:
2   port: 7001
3
4 # 单机版
5 eureka:
6   instance:
7     hostname: localhost #eureka服务端的实例名字
8   client:
9     register-with-eureka: false #表示不向注册中心注册自己
10    fetch-registry: false #表示自己就是注册中心，职责是维护服务实例，并不需要去检索
11      服务
12    service-url:
13      #设置与eureka server交互的地址查询服务和注册服务都需要依赖这个地址
14      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

```

4.主启动类

```

1 @SpringBootApplication
2 @EnableEurekaServer
3 public class EurekaMain7001 {
4   public static void main(String[] args) {
5     SpringApplication.run(EurekaMain7001.class,args);
6   }
7 }

```

5.此时就可以启动当前项目了

浏览器输入：localhost:7001

The screenshot shows the Spring Eureka dashboard. At the top, it displays 'spring Eureka' and 'HOME LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section provides details about the environment (test), data center (default), and system metrics like current time (2022-07-17T16:00:39+0800), uptime (00:03), lease expiration enabled (true), renew threshold (1), and renew interval (2). The 'DS Replicas' section shows 'Instances currently registered with Eureka' but lists 'No instances available'. The 'General Info' section contains various system metrics such as total available memory (487mb), environment (test), number of CPUs (8), current memory usage (167mb, 34%), server uptime (00:03), registered replicas, unavailable replicas, and available replicas. The 'Instance Info' section is partially visible at the bottom.

6.其他服务注册到eureka

将payment模块加入eureka

1.在pom中添加依赖

```

1 <!--Eureka-client-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5 </dependency>

```

2.主启动类上，加注解。表示当前是eureka客户端

```

1 @SpringBootApplication
2 @EnableEurekaClient
3 public class PaymentMain8001 {
4     public static void main(String[] args) {
5         SpringApplication.run(PaymentMain8001.class, args);
6     }
7 }

```

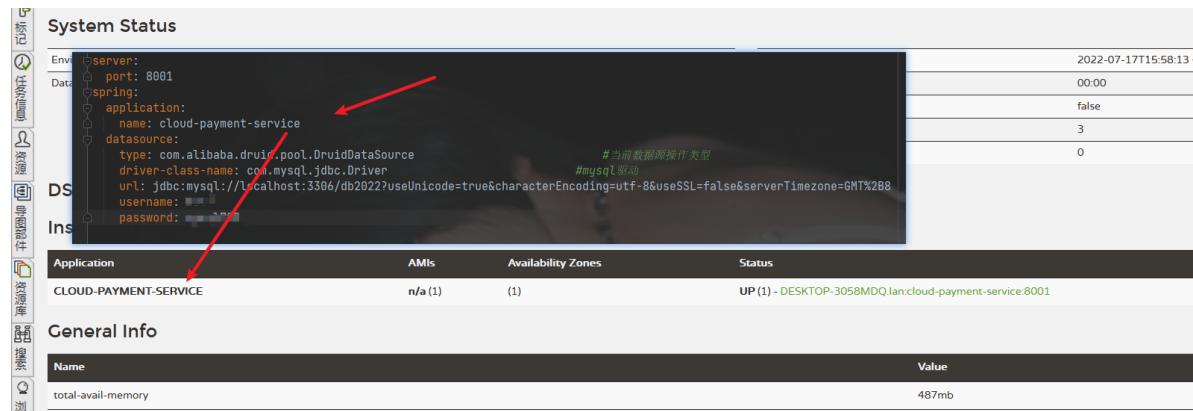
3.修改yml

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer 默认为true
4     register-with-eureka: true
5     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓，集群必须设置为
6     #true才能配合ribbon 使用负载均衡
7     fetch-registry: true
8     service-url:
9       defaultZone: http://localhost:7001/eureka

```

重启



成功注册

将order模块加入eureka

1.在pom中添加依赖

```

1 <!--Eureka-client-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5 </dependency>

```

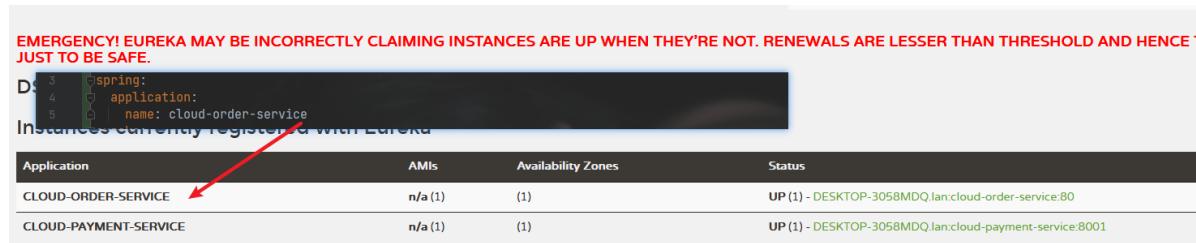
2.主启动类上，加注解。表示当前是eureka客户端

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class OrderMain80 {
4     public static void main(String[] args) {
5         SpringApplication.run(OrderMain80.class, args);
6     }
7 }
```

3.修改yml

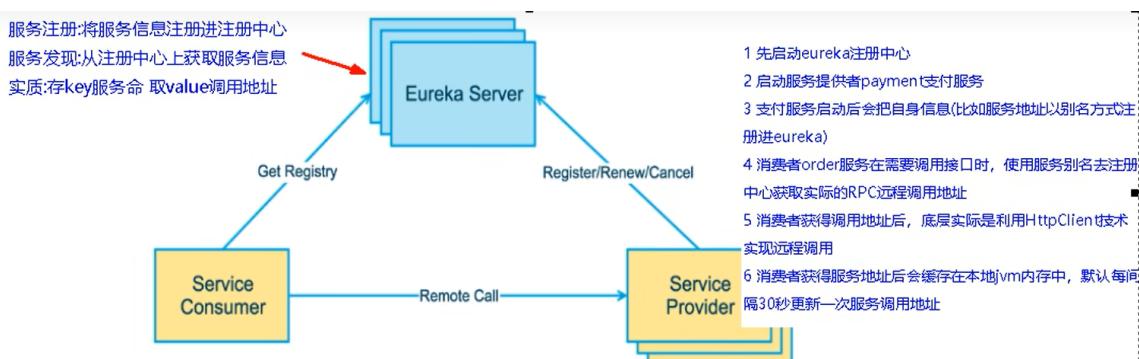
```
1 spring:
2   application:
3     name: cloud-order-service
4 eureka:
5   client:
6     # 表示是否将自己注册进EurekaServer 默认为true
7     register-with-eureka: true
8     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓，集群必须设置为
9     # true才能配合ribbon 使用负载均衡
10    fetch-registry: true
11    service-url:
12      defaultZone: http://localhost:7001/eureka
```

重启



集群

集群原理：



1. 就是pay模块启动时，注册自己，并且自身信息也放入eureka
2. order模块，首先也注册自己，放入信息，当要调用pay时，先从eureka拿到pay的调用地址
3. 通过HttpClient调用
4. 并且还会缓存一份到本地，每30秒更新一次

微服务RPC远程服务调用最核心的是什么？

高可用：如果只有一个注册中心，出故障整个系统就瘫痪了。会导致整个微服务环境不可用。

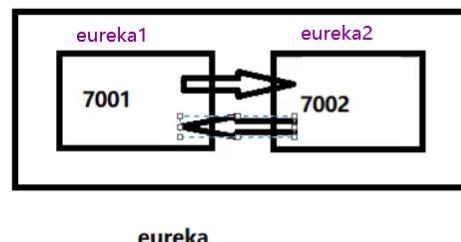
解决办法：搭建Eureka注册中心集群，实现负载均衡+故障容错。

集群搭建原理：

互相注册

如果有三台,那么就是1注册23,2->13, 3->12这样

互相注册，相互守望



构建新Eureka项目

cloud-eureka-server7002

1.pom文件

和7001一样即可

2.配置文件

在写配置文件之前，修改本机的hosts文件(C:\Windows\System32\drivers\etc)

```
1 # eureka
2 127.0.0.1 eureka7001.com
3 127.0.0.1 eureka7002.com
4 127.0.0.1 eureka7003.com
```

首先修改之前的7001的eureka项目,因为多个eureka需要互相注册

```
1 server:
2   port: 7001
3 #集群版
4 eureka:
5   instance:
6     hostname: eureka7001.com      #eureka服务端的实例名字
7   client:
8     register-with-eureka: false    #表示不向注册中心注册自己
9     fetch-registry: false        #表示自己就是注册中心，职责是维护服务实例，并不需要去检索
10    service-url:
11      #设置与eureka server交互的地址查询服务和注册服务都需要依赖这个地址
12    # defaultZone: http://eureka7001.com:7001/eureka/
13    defaultZone: http://eureka7002.com:7002/eureka/  #这个是集群版开启 互相注
册
```

修改7002

```

1 server:
2   port: 7002
3 #集群版
4 eureka:
5   instance:
6     hostname: eureka7002.com      #eureka服务端的实例名字
7   client:
8     register-with-eureka: false    #表示不向注册中心注册自己
9     fetch-registry: false        #表示自己就是注册中心，职责是维护服务实例，并不需要去检索
10    services
11    service-url:
12      #设置与eureka server交互的地址查询服务和注册服务都需要依赖这个地址
13    #      defaultZone: http://eureka7002.com:7002/eureka/
14      defaultZone: http://eureka7001.com:7001/eureka/ #这个是集群版开启 互相注
15    册

```

3,主启动类:

复制7001的即可

4,然后启动7001,7002即可

The image contains two screenshots of the Spring Eureka web interface, each showing the 'System Status' and 'DS Replicas' sections.

Screenshot 1 (Top):

- System Status:**
 - Environment: test
 - Data center: default
 - Current time: 2022-07-17T16:34:33 +0800
 - Uptime: 00:00
 - Lease expiration enabled: false
 - Renews threshold: 1
 - Renews (last min): 0
- DS Replicas:** A table showing a single entry: eureka7001.com

Screenshot 2 (Bottom):

- System Status:**
 - Environment: test
 - Data center: default
 - Current time: 2022-07-17T16:32:27 +0800
 - Uptime: 00:00
 - Lease expiration enabled: true
 - Renews threshold: 1
 - Renews (last min): 0
- DS Replicas:** A table showing a single entry: eureka7002.com

A red arrow points from the URL bar of the top screenshot to the URL bar of the bottom screenshot, indicating they are both running at localhost:7002.

将pay,order模块注册到eureka集群中:

1,只需要修改配置文件即可:

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer 默认为true
4     register-with-eureka: true
5     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓，集群必须设置为
6     # true才能配合ribbon 使用负载均衡
7     fetch-registry: true
8     service-url:
9       #      defaultZone: http://localhost:7001/eureka    # 单机版
       defaultZone:
          http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka    # 集群
          版

```

2,两个模块都修改上面的都一样即可

然后启动两个模块

要先启动7001,7002,然后是pay模块8001,然后是order(80)

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-payment-service:8001

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-payment-service:8001

将payment模块也配置为集群模式

1,创建新模块,8002

名称: cloud-provider-payment8002

2.pom文件

复制8001的

3,yml配置文件

复制8001的

端口修改成8002

服务名称不用改，用一样的

4.主启动类

复制8001的

修改为8002

5,mapper,service,controller都复制一份

为了区分，分别打印端口号：

```
1  @Slf4j
2  @RestController
3  @RequestMapping("/payment")
4  public class PaymentController {
5      @Autowired
6      private PaymentService paymentService;
7
8      @Value("${server.port}")
9      private String servicePort;
10
11     @PostMapping("/create")
12     public CommonResult<Payment> create(@RequestBody Payment payment){
13         int result = paymentService.create(payment);
14         log.info("****插入结果*****:"+result);
15         if(result > 0){
16             return new CommonResult(200,"插入数据库成功,serverPort:
17             "+servicePort,result);
18         }
19         return new CommonResult(444,"插入数据库失败",result);
20     }
21
22     @GetMapping("/get/{id}")
23     public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id)
24     {
25         Payment payment = paymentService.getPaymentById(id);
26         if(payment != null){
27             return new CommonResult(200,"查询成功,serverPort:
28             "+servicePort,payment);
29         }
30     }
31 }
```

然后就启动服务即可

The screenshot displays two browser windows side-by-side, both showing the Eureka service registry interface.

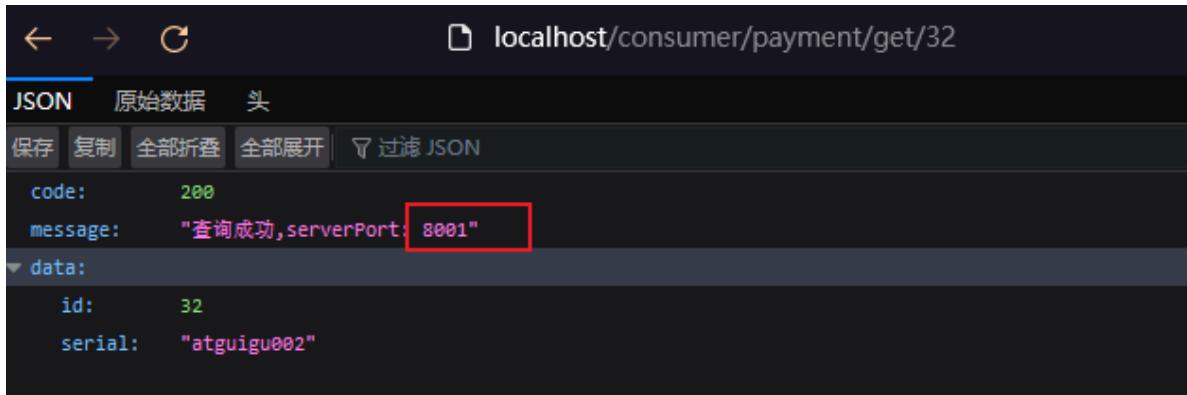
The top window's address bar shows `eureka7002.com:7002`. The page title is "DS Replicas". The main content table lists instances registered with Eureka:

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-3058MDQ.lan:cloud-payment-service:8001 , DESKTOP-3058MDQ.lan:cloud-payment-service:8002

The bottom window's address bar shows `eureka7001.com:7001`. The page title is "DS Replicas". The main content table lists instances registered with Eureka:

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-3058MDQ.lan:cloud-payment-service:8001 , DESKTOP-3058MDQ.lan:cloud-payment-service:8002

此时访问order模块,发现并没有负载均衡到两个pay模块中,而是只访问8001



```
JSON 原始数据 头  
保存 复制 全部折叠 全部展开 过滤 JSON  
code: 200  
message: "查询成功,serverPort: 8001"  
data:  
id: 32  
serial: "atguigu002"
```

虽然我们是使用RestTemplate访问的微服务,但是也可以负载均衡的

修改order80的访问地址

```
1 //public static final String PAYMENT_URL = "http://localhost:8001";  
2     public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";
```

注意这样还不行,需要让RestTemplate开启负载均衡注解,还可以指定负载均衡算法,默认轮询

修改RestTemplate的配置 添加@LoadBalanced

```
1 @Configuration  
2 public class ApplicationContextConfig {  
3     @Bean  
4     @LoadBalanced  
5     public RestTemplate getRestTemplate(){  
6         return new RestTemplate();  
7     }  
8 }
```

Ribbon和Eureka 整合后Consumer可以直接调用服务而不用再关心地址和端口号,且改该服务还有负载能力了。

修改服务主机名和ip在eureka的web上显示

修改服务主机名

修改yml配置文件

添加instance.instance-id

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer 默认为true
4     register-with-eureka: true
5     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓，集群必须设置为
6     # true才能配合ribbon 使用负载均衡
7     fetch-registry: true
8     service-url:
9       #     defaultZone: http://localhost:7001/eureka    # 单机版
10      defaultZone:
11        http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka      # 集群
12      instance:
13        instance-id: payment8001

```

同样的修改8002的

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002, payment8001

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-3058MDQ.lan:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002, payment8001

添加ip显示

修改yml配置文件

添加instance.prefer-ip-address

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer 默认为true
4     register-with-eureka: true
5     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓，集群必须设置为
6     # true才能配合ribbon 使用负载均衡
7     fetch-registry: true
8     service-url:
9       #     defaultZone: http://localhost:7001/eureka    # 单机版
10      defaultZone:
11        http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka      # 集群
12      instance:
13        instance-id: payment8001
14        prefer-ip-address: true          # 访问路径可以显示IP地址

```

同样的修改8002的

服务发现

对于注册进eureka里面的微服务，可以通过服务发现来获得该服务的信息
以payment模块为例

1. 引入DiscoveryClient

在controller里面

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController {
5     @Autowired
6     private PaymentService paymentService;
7
8     @Value("${server.port}")
9     private String servicePort;
10
11    @Resource
12    private DiscoveryClient discoveryClient;
13
14    @GetMapping("/discovery")
15    public Object discovery(){
16        List<String> services = discoveryClient.getServices();
17        for (String element : services) {
18            log.info("*****element: " + element);
19        }
20
21        List<ServiceInstance> instances =
22        discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
23        for (ServiceInstance instance : instances) {
24
25            log.info("instance:" + instance.getInstanceId() + "\t" + instance.getHost() + "\t" +
26            instance.getPort() + "\t" + instance.getUri());
27        }
28        return this.discoveryClient;
29    }
30}
```

2. 在主启动类上添加一个注解

@EnableDiscoveryClient

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableDiscoveryClient
4 public class PaymentMain8001 {
5     public static void main(String[] args) {
6         SpringApplication.run(PaymentMain8001.class, args);
7     }
8 }
```

然后重启8001。访问/payment/discovery

Eureka的自我保护机制

自我保护机制原理

保护模式主要用于一组客户端和Eureka Server之间存在网络分区场景下的保护。

一旦进入保护模式，Eureka Server将会尝试保护其服务注册表中的信息，不再删除服务注册表中的数据，也就是不会注销任何微服务。

如果在Eureka Server的首页看到以下这段提示，则说明Eureka进入了保护模式：

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

即：某时刻某一个微服务不可用了。Eureka不会立刻清理，依旧会对该微服务的信息进行保存。（属于CAP里面的AP分支）

- 为什么会产生Eureka自我保护机制？

为了防EurekaClient可以正常运行，但是与EurekaServer网络不通情况下，EurekaServer不会立刻将EurekaClient服务剔除

- 什么是自我保护机制？

默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生（延时、卡

顿、拥挤）时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。

Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模

式。

总结：

在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。

它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话讲解：好死不如赖活着

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留）也不盲目注销

任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

禁止自我保护

以7001和8001为例

修改7001的yml：

```
1 eureka:  
2   instance:  
3     hostname: eureka7001.com      #eureka服务端的实例名字  
4     client:  
5       register-with-eureka: false    #表示不向注册中心注册自己  
6       fetch-registry: false        #表示自己就是注册中心，职责是维护服务实例，并不需要去检索  
7       service-url:  
8         #设置与eureka server交互的地址查询服务和注册服务都需要依赖这个地址  
9         defaultZone: http://eureka7001.com:7001/eureka/
```

```

10 #      defaultZone: http://eureka7002.com:7002/eureka/ #这个是集群版开启 互相注
  册 集群就是指向其他的eureka
11   server:
12 #      关闭自我保护机制, 保证不可用服务被及时踢除
13   enable-self-preservation: false
14   eviction-interval-timer-in-ms: 2000

```

修改8001的yml

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer 默认为true
4     register-with-eureka: true
5     # 是否从EurekaServer 抓取已有的注册信息。默认为true。单节点无所谓, 集群必须设置为
6     # true才能配合ribbon 使用负载均衡
7     fetch-registry: true
8     service-url:
9       #      defaultZone: http://localhost:7001/eureka    # 单机版
10      defaultZone:
11        http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka    # 集群
12        版
13
14   instance:
15     instance-id: payment8001
16     prefer-ip-address: true          # 访问路径可以显示IP地址
17     # Eureka 客户端向服务端发送心跳的时间间隔, 单位为秒 (默认为30秒)
18     lease-renewal-interval-in-seconds: 1
19     # Eureka 客户端在收到最后一次心跳后等待的时间上限, 单位为秒 (默认为90秒), 超时将剔除
20     服务
21     lease-expiration-duration-in-seconds: 2

```

先关闭8001

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-SERVICE	n/a (1)	{1}	DOWN (1) - payment8001

此时启动erueka和pay.此时如果直接关闭了pay,那么erueka会直接删除其注册信息

2、Zookeeper

zookeeper是一个分布式协调工具，可以实现注册中心功能

关闭Linux服务器防火墙后启动zookeeper服务器

Zookeeper服务器取代Eureka服务器，zk作为服务注册中心

1.安装zookeeper并启动

2.创建新的pay模块

单独用于注册到zk中

名字：cloud-provider-zookeeper-payment8004

1.pom依赖

```
1 <dependencies>
2     <!--springboot 整合 zookeeper 客户端-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-zookeeper-
6         discovery</artifactId>
7     </dependency>
8
9     <dependency>
10        <groupId>org.example</groupId>
11        <artifactId>cloud-api-commons</artifactId>
12        <version>${project.version}</version>
13    </dependency>
14    <dependency>
15        <groupId>org.springframework.boot</groupId>
16        <artifactId>spring-boot-starter-web</artifactId>
17    </dependency>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-starter-actuator</artifactId>
21    </dependency>
22    <dependency>
23        <groupId>org.springframework.boot</groupId>
24        <artifactId>spring-boot-devtools</artifactId>
25        <scope>runtime</scope>
26        <optional>true</optional>
27    </dependency>
28    <dependency>
29        <groupId>org.projectlombok</groupId>
30        <artifactId>lombok</artifactId>
31        <optional>true</optional>
32    </dependency>
33    <dependency>
34        <groupId>org.springframework.boot</groupId>
35        <artifactId>spring-boot-starter-test</artifactId>
36    </dependency>
37 </dependencies>
```

2.yml配置文件

```
1 # 8004 表示注册到zookeeper服务器的支付服务提供者端口号
2 server:
3     port: 8004
4
5 # 服务别名----注册到zookeeper 到注册中心的名称
6 spring:
7     application:
8         name: cloud-provider-payment
9     cloud:
10        zookeeper:
11            connect-string: 192.168.160.128:2181
```

3,主启动类

```
1 @SpringBootApplication
2 // 该注解用于向使用consul或者zookeeper 作为注册中心时的注册服务
3 @EnableDiscoveryClient
4 public class PaymentMain8004 {
5     public static void main(String[] args) {
6         SpringApplication.run(PaymentMain8004.class,args);
7     }
8 }
```

4.controller

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController {
5     @Value("${server.port}")
6     private String serverPort;
7
8     @RequestMapping("/zk")
9     public String paymentZk(){
10         return "Springcloud with zookeeper: "+serverPort +"\t"+
11         UUID.randomUUID();
12     }
13 }
```

5,启动

此时启动,会报错,因为jar包与我们的zk版本不匹配

修改pom文件, 改成与我们zk相匹配的jar包

eg:

```
<!-- SpringBoot整合zookeeper客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
    <!-- 先排除自带的zookeeper3.5.3-->
    <exclusions>
        <exclusion>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<!-- 添加zookeeper3.4.9版本-->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.9</version>
</dependency>
```

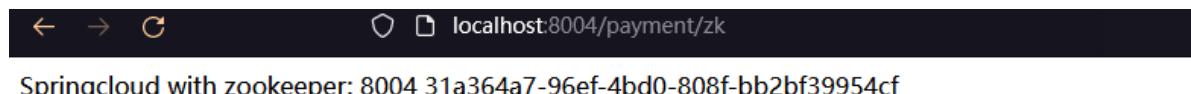
再次重启:

```
[zk: localhost:2181(CONNECTED) 2] ls /
[services, zookeeper]
[zk: localhost:2181(CONNECTED) 3] ls /services
[cloud-provider-payment]
[zk: localhost:2181(CONNECTED) 4]
```

此时8004就已经注册到zk中了。

6. 测试

- 验证测试：浏览器 - <http://localhost:8004/payment/zk>



- 验证测试2：接着用zookeeper客户端操作

```
[zk: localhost:2181(CONNECTED) 3] ls /services/cloud-provider-payment
Node does not exist: /services/cloud-provider-payment
[zk: localhost:2181(CONNECTED) 4] ls /services
[cloud-provider-payment]
[zk: localhost:2181(CONNECTED) 5] ls /services/cloud-provider-payment
[9e03f796-9672-4136-ae32-84f0de4ccacc]
[zk: localhost:2181(CONNECTED) 6] ls /services/cloud-provider-payment/9e03f796-9672-4136-ae32-84f0de4ccacc
[]
[zk: localhost:2181(CONNECTED) 7] get /services/cloud-provider-payment/9e03f796-9672-4136-ae32-84f0de4ccacc
{"name":"cloud-provider-payment","id":"9e03f796-9672-4136-ae32-84f0de4ccacc","address":"DESKTOP-3058MDQ.lan","port":8004,"sslPort":null,"payload":{"@class":"org.springframework.cloud.zookeeper.discovery.ZookeeperInstance","id":"application-1","name":"cloud-provider-payment","metadata":{}}, "registrationTimeUTC":1658057554392,"serviceType":"DYNAMIC","uriSpec":{"parts":[{"value":"scheme","variable":true},{ "value":"://","variable":false},{ "value":"address","variable":true},{ "value":":","variable":false},{ "value":"port","variable":true}]}}
[zk: localhost:2181(CONNECTED) 8]
```

```
1 [zk: localhost:2181(CONNECTED) 5] ls /services/cloud-provider-payment
2 [9e03f796-9672-4136-ae32-84f0de4ccacc]
3 [zk: localhost:2181(CONNECTED) 6] ls /services/cloud-provider-
4 payment/9e03f796-9672-4136-ae32-84f0de4ccacc
5 []
6 [zk: localhost:2181(CONNECTED) 7] get /services/cloud-provider-
7 payment/9e03f796-9672-4136-ae32-84f0de4ccacc
8 {"name":"cloud-provider-payment","id":"9e03f796-9672-4136-ae32-
9 84f0de4ccacc","address":"DESKTOP-
10 3058MDQ.lan","port":8004,"sslPort":null,"payload":
11 {"@class":"org.springframework.cloud.zookeeper.discovery.ZookeeperInstance",
12 "id":"application-1","name":"cloud-provider-payment","metadata":{}},
13 "registrationTimeUTC":1658057554392,"serviceType":"DYNAMIC","uriSpec":
14 {"parts":[{"value":"scheme","variable":true},
15 {"value":"://","variable":false},{ "value":"address","variable":true},
16 {"value":":","variable":false},{ "value":"port","variable":true}]}}
```

json格式化 `get /services/cloud-provider-payment/a4567f50-6ad9-47a3-9fbb-7391f41a9f3d` 的结果：

```
1 {
2     "name": "cloud-provider-payment",
3     "id": "9e03f796-9672-4136-ae32-84f0de4ccacc",
4     "address": "DESKTOP-3058MDQ.lan",
5     "port": 8004,
6     "sslPort": null,
7     "payload": {
8         "@class":
9             "org.springframework.cloud.zookeeper.discovery.ZookeeperInstance",
10            "id": "application-1",
11            "name": "cloud-provider-payment",
12            "metadata": {}}
```

```

12 },
13 "registrationTimeUTC": 1658057554392,
14 "serviceType": "DYNAMIC",
15 "uriSpec": {
16     "parts": [
17         {
18             "value": "scheme",
19             "variable": true
20         },
21         {
22             "value": "://",
23             "variable": false
24         },
25         {
26             "value": "address",
27             "variable": true
28         },
29         {
30             "value": ":",
31             "variable": false
32         },
33         {
34             "value": "port",
35             "variable": true
36         }
37     ]
38 }
39 }

```

ZooKeeper的服务节点是临时节点，没有Eureka那含情脉脉。

3.创建order消费模块注册到zk

1,创建项目

名字: cloud-consum-zookeeper-order80

2,pom

```

1 <dependencies>
2     <!--springboot 整合 zookeeper 客户端-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-zookeeper-
6 discovery</artifactId>
7     </dependency>
8
8     <dependency>
9         <groupId>org.example</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <dependency>
14        <groupId>org.springframework.boot</groupId>
15        <artifactId>spring-boot-starter-web</artifactId>
16    </dependency>
17    <dependency>

```

```

18         <groupId>org.springframework.boot</groupId>
19         <artifactId>spring-boot-starter-actuator</artifactId>
20     </dependency>
21     <dependency>
22         <groupId>org.springframework.boot</groupId>
23         <artifactId>spring-boot-devtools</artifactId>
24         <scope>runtime</scope>
25         <optional>true</optional>
26     </dependency>
27     <dependency>
28         <groupId>org.projectlombok</groupId>
29         <artifactId>lombok</artifactId>
30         <optional>true</optional>
31     </dependency>
32
33     <dependency>
34         <groupId>org.springframework.boot</groupId>
35         <artifactId>spring-boot-starter-test</artifactId>
36     </dependency>
37 </dependencies>

```

3.配置文件

```

1 server:
2   port: 80
3
4 # 服务别名----注册到zookeeper 到注册中心的名称
5 spring:
6   application:
7     name: cloud-consumer-order
8   cloud:
9     zookeeper:
10      connect-string: 127.0.0.1:2181 # 192.168.160.128:2181

```

4.主启动类

```

1 @SpringBootApplication
2 // 该注解用于向使用consul或者zookeeper 作为注册中心时的注册服务
3 @EnableDiscoveryClient
4 public class OrderZKMain80 {
5   public static void main(String[] args) {
6     SpringApplication.run(OrderZKMain80.class,args);
7   }
8 }

```

5.配置类

```

1 @Configuration
2 public class ApplicationContextConfig {
3   @Bean
4   @LoadBalanced
5   public RestTemplate getRestTemplate(){
6     return new RestTemplate();
7   }
8 }

```

6.业务类

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class OrderZKController {
5     public static final String INVOKE_URL = "http://cloud-provider-payment";
6
7     @Resource
8     private RestTemplate restTemplate;
9
10    @GetMapping(value = "/consumer/payment/zk")
11    public String paymentInfo()
12    {
13        String result =
14            restTemplate.getForObject(INVOKE_URL+"/payment/zk",String.class);
15        return result;
16    }
17}
```

7.启动

可以发现已经注册

```
[zk: localhost:2181(CONNECTED) 12] ls /
[services, zookeeper]
[zk: localhost:2181(CONNECTED) 13] ls /services
[cloud-consumer-order, cloud-provider-payment]
[zk: localhost:2181(CONNECTED) 14]
```

8. 集群版的zk注册

只需修改配置文件

这个connect-string指定多个zk地址即可

connect-string: 1.2.3.4,2.3.4.5

3、consul

简介

[Consul官网](#)

[Consul下载地址](#)

What is Consul?

Consul is a service mesh solution providing a full featured control plane with service discovery, configuration, and segmentation functionality. Each of these features can be used individually as needed, or they can be used together to build a full service mesh. Consul ships with a simple built-in proxy so that everything works out of the box, but also supports 3rd party proxy integrations such as Envoy, link

Consul是一个服务网格解决方案，它提供了一个功能齐全的控制平面，具有服务发现、配置和分段功能。这些特性中的每一个都可以根据需要单独使用，也可以一起用于构建全服务网格。

Consul需要一个数据平面，并支持代理和本机集成模型。Consul船与一个简单的内置代理，使一切工作的开箱即用，但也支持第三方代理集成，如Envoy。

基本概念

Consul是一套开源的分布式服务发现和配置管理系统，由HashiCorp公司用Go语言开发。

提供了微服务系统中的服务治理、配置中心、控制总线等功能。这些功能中的每一个都可以根据需要单独使用，也可以一起使用以构建全方位的服务网格，总之

Consul提供了一种完整的服务网格解决方案。

它具有很多优点。包括：基于raft协议，比较简洁；支持健康检查，同时支持HTTP和DNS协议；支持跨数据中心的WAN集群；提供图形界面跨平台，支持Linux、Mac、Windows。

功能

- 服务发现
 - 提供http和dns两种发现方式
- 健康检测
 - 支持多种方式，http、tcp、docker、shell脚本定制化
- kv存储
 - key、value的存储方式
- 多数据中心
 - consul支持多数据中心
- 可视化web界面

怎么玩

安装并运行consul

官网安装说明

windows版解压缩后，得consul.exe，打开cmd

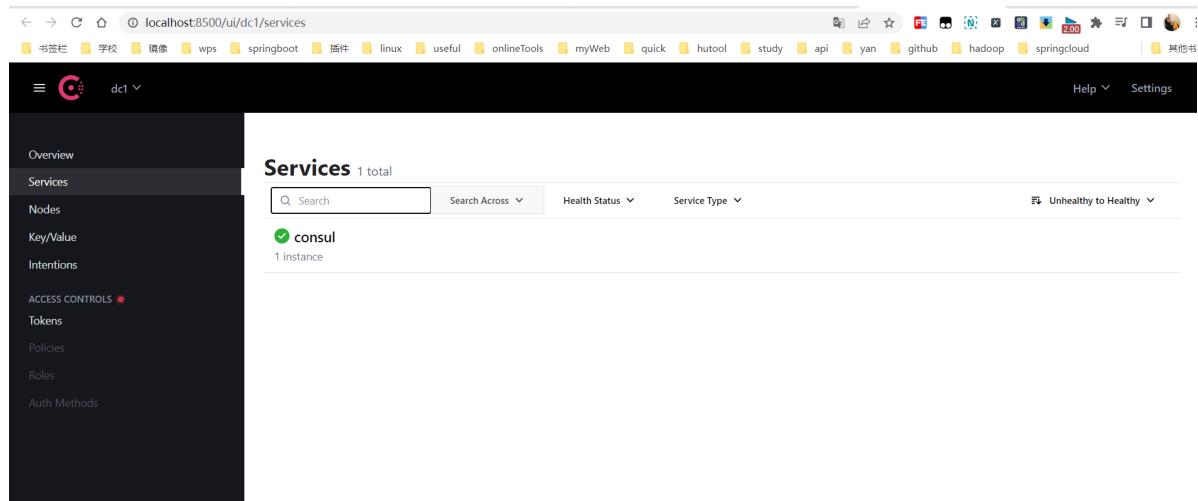
查看版本 `consul -v`：

```
D:\studytools\springcloud\consul>consul -v
Consul v1.12.3
Revision 2308c75e
Protocol 2 spoken by default, understands 2 to 3 (agent will automatically use protocol >2 when speaking to compatible agents)
```

- 1 D:\studytools\springcloud\consul>consul -v
- 2 Consul v1.12.3
- 3 Revision 2308c75e
- 4 Protocol 2 spoken by default, understands 2 to 3 (agent will automatically use protocol >2 when speaking to compatible agents)

开发模式启动 `consul agent -dev`：

浏览器输入 - <http://localhost:8500/> - 打开Consul控制页。



服务提供者注册进consul

1. 新建Module支付服务provider8006

名字cloud-provider-consul-payment8006

2.pom依赖

```
1 <dependencies>
2     <!--springboot 整合 consul 客户端-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-consul-discovery</artifactId>
6     </dependency>
7     <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
8     <dependency>
9         <groupId>org.example</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <dependency>
14        <groupId>org.springframework.boot</groupId>
15        <artifactId>spring-boot-starter-web</artifactId>
16    </dependency>
17    <dependency>
18        <groupId>org.springframework.boot</groupId>
19        <artifactId>spring-boot-starter-actuator</artifactId>
20    </dependency>
21    <dependency>
22        <groupId>org.springframework.boot</groupId>
23        <artifactId>spring-boot-devtools</artifactId>
24        <scope>runtime</scope>
25        <optional>true</optional>
26    </dependency>
27    <dependency>
28        <groupId>org.projectlombok</groupId>
29        <artifactId>lombok</artifactId>
30        <optional>true</optional>
31    </dependency>
32
33    <dependency>
34        <groupId>org.springframework.boot</groupId>
35        <artifactId>spring-boot-starter-test</artifactId>
```

```
36      </dependency>
37  </dependencies>
```

3.yml配置

```
1 # consul 服务端口号
2 server:
3   port: 8006
4
5 spring:
6   application:
7     name: cloud-provider-payment
8   cloud:
9     # consul注册中心地址
10    consul:
11      host: localhost
12      port: 8500
13    discovery:
14      #hostname 127.0.0.1
15      service-name: ${spring.application.name}
```

4.主启动类

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class PaymentMain8006 {
4   public static void main(String[] args) {
5     SpringApplication.run(PaymentMain8006.class,args);
6   }
7 }
```

5.业务类controller

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController {
5
6   @Value("${server.port}")
7   private String serverPort;
8
9   @RequestMapping("/consul")
10  public String paymentZk(){
11    return "SpringCloud with consul: "+serverPort +"\t"+
12    UUID.randomUUID();
13  }
14 }
```

6.验证测试

Services 2 total

Search	Search Across	Health Status	Service Type
<input checked="" type="checkbox"/> consul			
1 instance			
<input checked="" type="checkbox"/> consul-provider-payment		secure=false	
1 instance			

服务消费者注册进consul

1.新建Module消费服务order80

名字cloud-consumer-consul-order80

2.pom依赖

```
1 <dependencies>
2     <!--springboot 整合 consul 客户端-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-consul-discovery</artifactId>
6     </dependency>
7     <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
8     <dependency>
9         <groupId>org.example</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <dependency>
14        <groupId>org.springframework.boot</groupId>
15        <artifactId>spring-boot-starter-web</artifactId>
16    </dependency>
17    <dependency>
18        <groupId>org.springframework.boot</groupId>
19        <artifactId>spring-boot-starter-actuator</artifactId>
20    </dependency>
21    <dependency>
22        <groupId>org.springframework.boot</groupId>
23        <artifactId>spring-boot-devtools</artifactId>
24        <scope>runtime</scope>
25        <optional>true</optional>
26    </dependency>
27    <dependency>
28        <groupId>org.projectlombok</groupId>
29        <artifactId>lombok</artifactId>
30        <optional>true</optional>
31    </dependency>
32
33    <dependency>
34        <groupId>org.springframework.boot</groupId>
35        <artifactId>spring-boot-starter-test</artifactId>
36    </dependency>
37 </dependencies>
```

3.yml配置

```
1 # consul 服务端口号
2 server:
3     port: 80
4
5 spring:
6     application:
7         name: cloud-consul-order
8     cloud:
9         # consul注册中心地址
10        consul:
11            host: localhost
12            port: 8500
13            discovery:
14                #hostname 127.0.0.1
15                service-name: ${spring.application.name}
```

4.主启动类

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class PaymentMain8006 {
4     public static void main(String[] args) {
5         SpringApplication.run(PaymentMain8006.class,args);
6     }
7 }
```

5配置类

```
1 @Configuration
2 public class ApplicationContextConfig {
3     @Bean
4     @LoadBalanced
5     public RestTemplate getRestTemplate(){
6         return new RestTemplate();
7     }
8 }
```

6.业务类controller

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class OrderConsulController {
5     public static final String INVOKE_URL = "http://cloud-provider-payment";
6
7     @Resource
8     private RestTemplate restTemplate;
9
10    @GetMapping(value = "/payment/consul")
11    public String paymentInfo()
12    {
13        return
14            restTemplate.getForObject(INVOKE_URL+"/payment/consul",String.class);
```

```
14 }
15
16 }
```

7. 验证测试

运行consul, cloud-provider-consul-payment8006, cloud-consumer-consul-order80

<http://localhost:8500/> 主页会显示出consul, cloud-providerconsul-payment8006, cloud-consumerconsul-order80三服务。

Services 3 total

Search	Search Across	Health Status	Service Type
consul			
1 instance			
cloud-consumer-order			
1 instance	secure=false		
consul-provider-payment			
1 instance	secure=false		

三个注册中心异同点

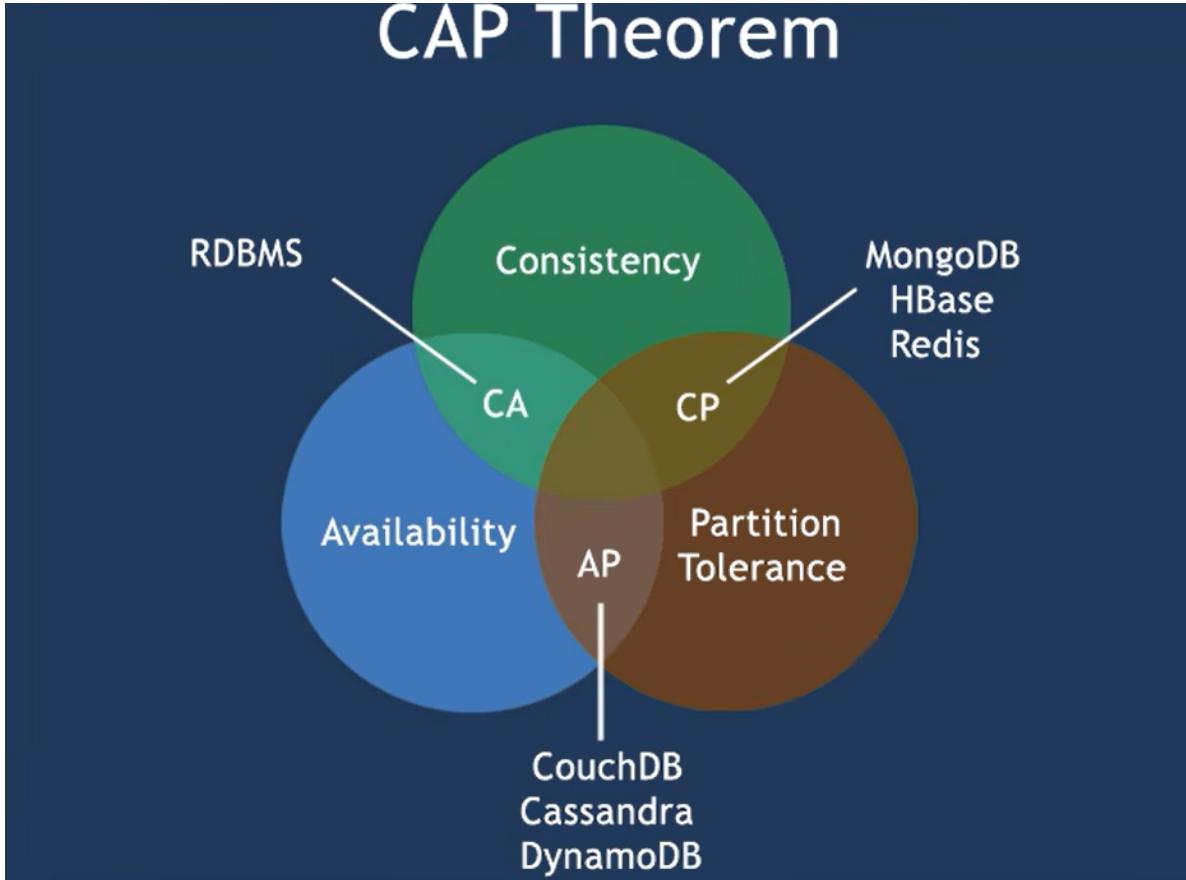
组件名	语言	CAP	服务健康检查	对外暴露接口	Spring Cloud集成
Eureka	Java	AP	可配支持	HTTP	已集成
Consul	Go	CP	支持	HTTP/DNS	已集成
Zookeeper	Java	CP	支持	客户端	已集成

CAP:

- C: Consistency (强一致性)
- A: Availability (可用性)
- P: Partition tolerance (分区容错性)

CAP理论关注粒度是数据，而不是整体系统设计的策略

CAP Theorem



最多只能同时较好的满足两个。

CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求。

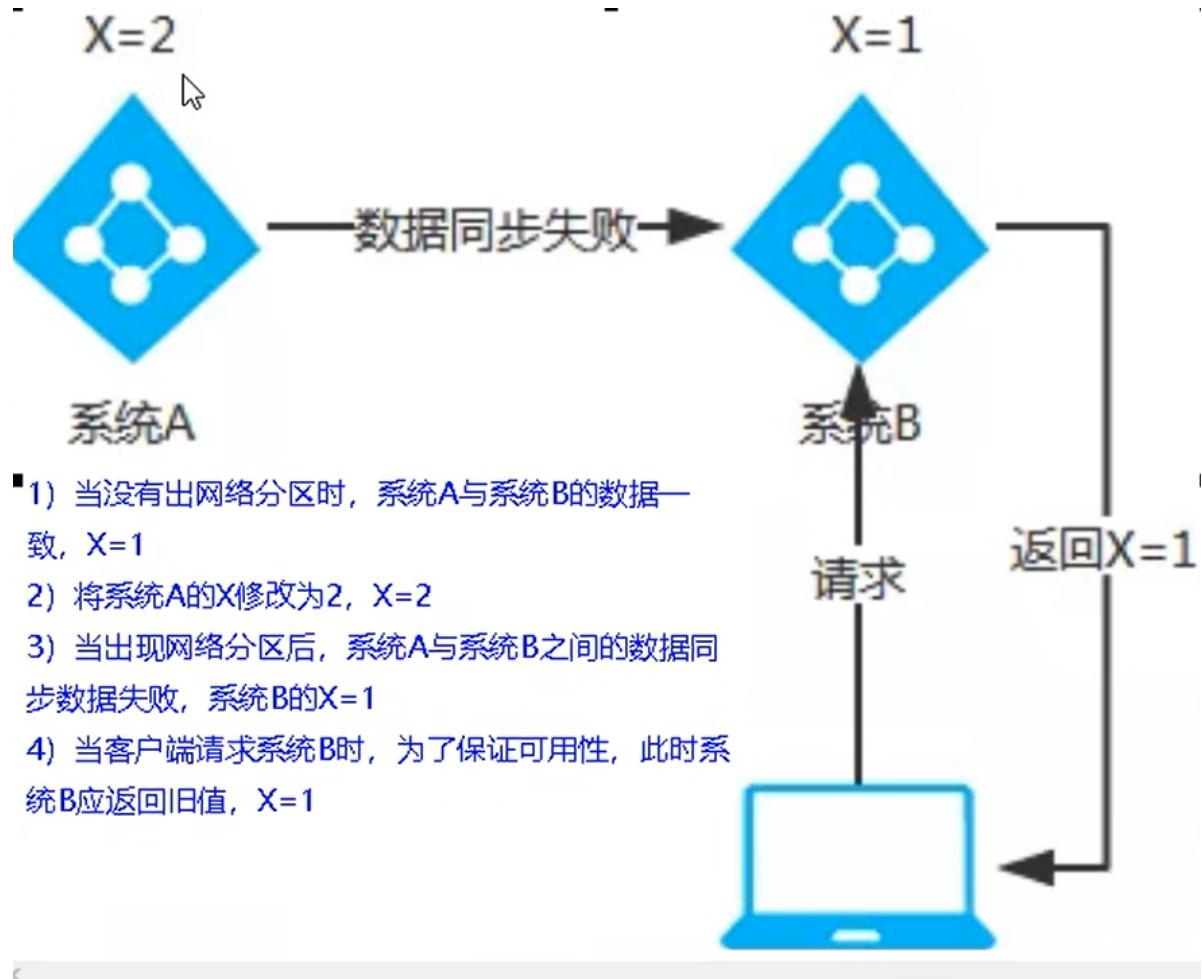
因此，根据CAP原理将NoSQL数据库分成了满足CA原则、满足CP原则和满足AP原则三大类：

- CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。
- CP - 满足一致性，分区容忍的系统，通常性能不是特别高。
- AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。

AP架构 (Eureka)

当网络分区出现后，为了保证可用性，系统B可以返回旧值，保证系统的可用性。

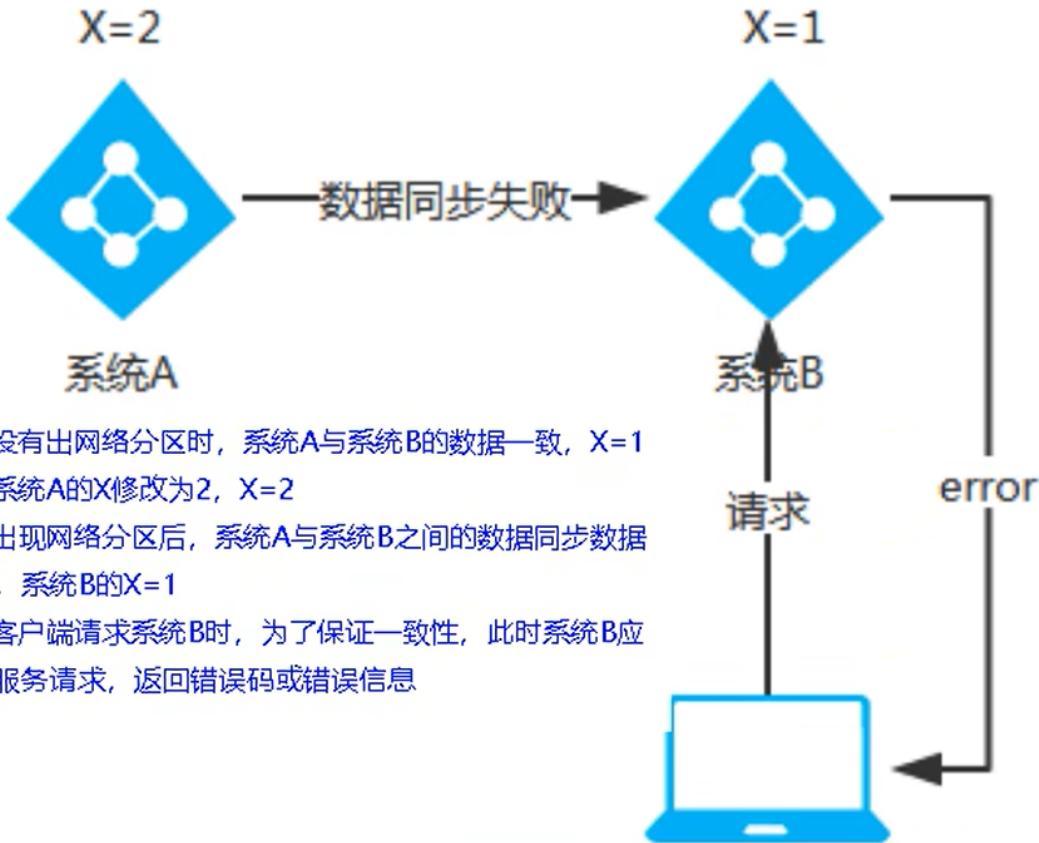
结论：违背了一致性C的要求，只满足可用性和分区容错，即AP



CP架构 (ZooKeeper/Consul)

当网络分区出现后, 为了保证一致性, 就必须拒接请求, 否则无法保证一致性。

结论: 违背了可用性A的要求, 只满足一致性和分区容错, 即CP。



CP 与 AP 对立同一的矛盾关系。

服务调用

Ribbon负载均衡

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套**客户端 负载均衡的工具**。

简单的说, Ribbon是Netflix发布的开源项目, 主要功能是提供**客户端的软件负载均衡算法和服务调用**。Ribbon客户端组件提供一系列完善的配置项如连接超时, 重试等。

简单的说, 就是在配置文件中列出Load Balancer(简称LB)后面所有的机器, Ribbon会自动的帮助你基于某种规则(如简单轮询, 随机连接等)去连接这些机器。我们很容易使用Ribbon实现自定义的负载均衡算法。

ribbon

英 [ˈrɪbən] 美 [ˈrɪbən]

n. (用于捆绑或装饰的)带子;丝带;带状物;狭长的东西;绶带;勋带

[Github - Ribbon](#)

Ribbon目前也进入维护模式。

Ribbon未来可能被Spring Cloud LoadBalacer替代。

LB负载均衡(Load Balance)是什么

简单的说就是将用户的请求平摊的分配到多个服务上, 从而达到系统的HA (高可用)。

常见的负载均衡有软件Nginx, LVS, 硬件F5等。

Ribbon本地负载均衡客户端 VS Nginx服务端负载均衡区别

Nginx是服务器负载均衡，客户端所有请求都会交给nginx，然后由nginx实现转发请求。即负载均衡是由服务端实现的。

Ribbon本地负载均衡，在调用微服务接口时候，会在注册中心上获取注册信息服务列表之后缓存到JVM本地，从而在本地实现RPC远程服务调用技术。

集中式LB

即在服务的消费方和提供方之间使用独立的LB设施(可以是硬件，如F5, 也可以是软件，如nginx)，由该设施负责把访问请求通过某种策略转发至服务的提供方；

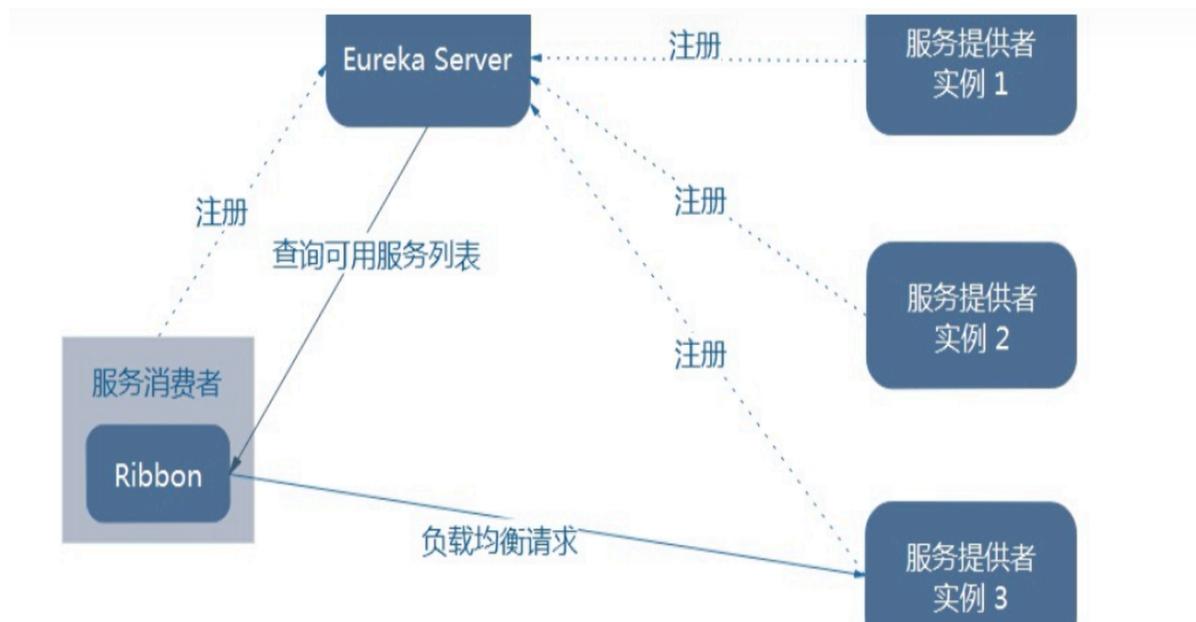
进程内LB

将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

一句话 负载均衡 + RestTemplate调用

Ribbon的负载均衡和Rest调用



总结：Ribbon其实就是一个软负载均衡的客户端组件，它可以和其他所需请求的客户端结合使用，和Eureka结合只是其中的一个实例。

Ribbon在工作时分成两步：

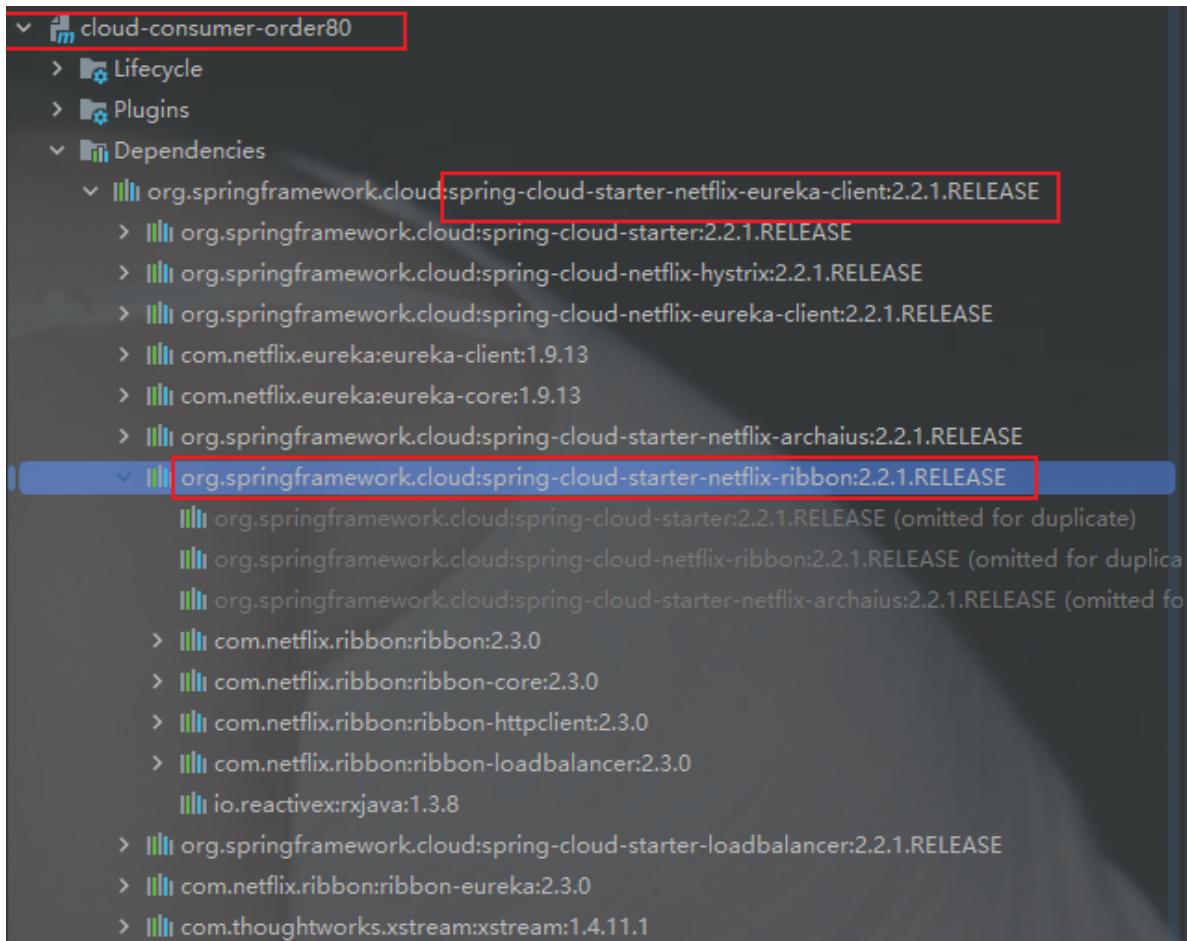
- 第一步先选择EurekaServer, 它优先选择在同一个区域内负载较少的server。
- 第二步再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。

其中Ribbon提供了多种策略：比如轮询、随机和根据响应时间加权。

使用Ribbon：

1. 默认我们使用eureka的新版本时，它默认集成了ribbon：

这是因为spring-cloud-starter-netflix-eureka-client自带了spring-cloud-starter-ribbon引用。



2. 我们也可以手动引入ribbon

放到order模块中，因为只有order访问pay时需要负载均衡

先前工程项目没有引入spring-cloud-starter-ribbon也可以使用ribbon。

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
4 </dependency>
```

RestTemplate的使用

[RestTemplate Java Doc](#)

getForObject() / getForEntity() - GET请求方法

getForObject(): 返回对象为响应体中数据转化成的对象，基本上可以理解为Json。

getForEntity(): 返回对象为 ResponseEntity 对象，包含了响应中的一些重要信息，比如响应头、响应状态码、响应体等。

```

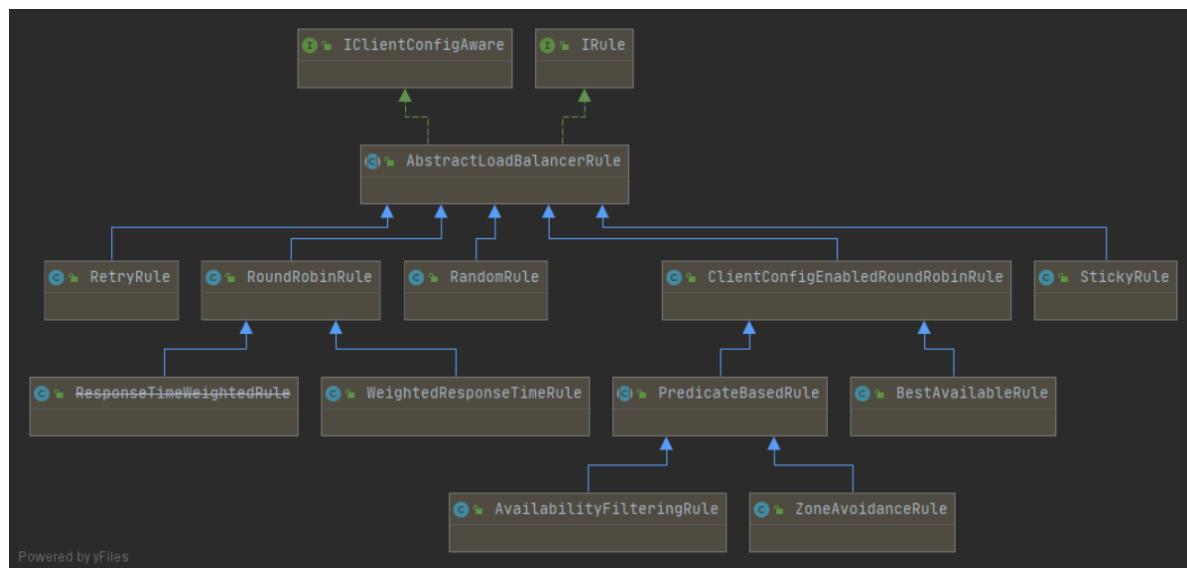
1  @GetMapping("/payment/getForEntity/{id}")
2  public CommonResult<Payment> getPayment2(@PathVariable("id") Long id)
3  {
4      ResponseEntity<CommonResult> entity =
5      restTemplate.getForEntity(PAYMENT_URL+"/payment/get/"+id,CommonResult.class);
6
7      if(entity.getStatusCode().is2xxSuccessful()){
8          return entity.getBody(); //getForObject()
9      }else{
10         return new CommonResult<>(444,"操作失败");
11     }
12 }

```

postForObject() / postForEntity() - POST请求方法

Ribbon默认自带的负载规则

IRule：根据特定算法中从服务列表中选取一个要访问的服务



- com.netflix.loadbalancer.RoundRobinRule 轮询
- com.netflix.loadbalancer.RandomRule 随机
- com.netflix.loadbalancer.RetryRule 先按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定时间内会进行重试
- WeightedResponseTimeRule 对RoundRobinRule的扩展，响应速度越快的实例选择权重越大，越容易被选择
- BestAvailableRule 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务
- AvailabilityFilteringRule 先过滤掉故障实例，再选择并发较小的实例
- ZoneAvoidanceRule 默认规则，复合判断server所在区域的性能和server的可用性选择服务器

Ribbon负载规则替换

1.修改cloud-consumer-order80

2.注意配置细节

官方文档明确给出了警告:

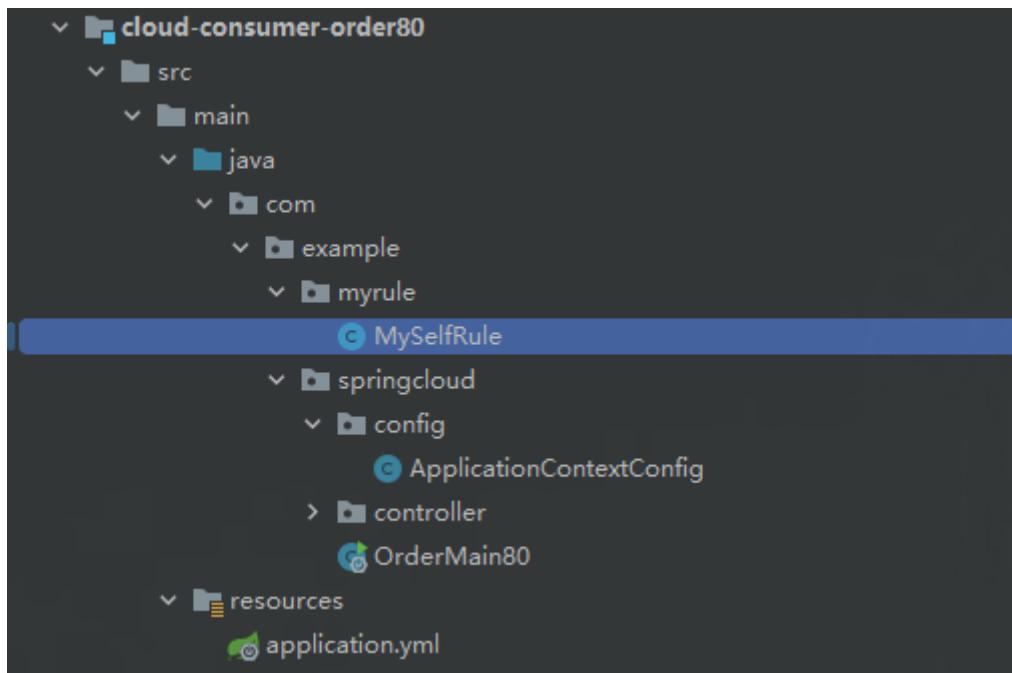
这个自定义配置类不能放在@ComponentScan所扫描的当前包下以及子包下,

否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享, 达不到特殊化定制的目的了。 (也就是说不要将Ribbon配置类与主启动类同包)

3.新建package - myrule

4.在com.example.myrule下新建MySelfRule规则类

```
1 import com.netflix.loadbalancer.IRule;
2 import com.netflix.loadbalancer.RandomRule;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class MySelfRule {
8
9     @Bean
10    public IRule myRule(){
11        return new RandomRule();
12    }
13 }
```



5.主启动类添加@RibbonClient

表示,访问CLOUD_PAYMENT_SERVICE的服务时,使用我们自定义的负载均衡算法

```
1 import com.example.myrule.MySelfRule;
2 import org.springframework.boot.SpringApplication;
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
5 import org.springframework.cloud.netflix.ribbon.RibbonClient;
6
7 @SpringBootApplication
```

```

8  @EnableEurekaClient
9  @RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration =
10 MySelfRule.class)
11 public class OrderMain80 {
12     public static void main(String[] args) {
13         SpringApplication.run(OrderMain80.class, args);
14     }

```

6. 测试

开启cloud-eureka-server7001, cloud-consumer-order80, cloud-provider-payment8001, cloud-provider-payment8002

浏览器输入<http://localhost/consumer/payment/get/31>

返回结果中的serverPort在8001与8002两种间反复横跳。

Ribbon默认负载轮询算法原理

默认负载轮训算法: rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标, 每次服务重启后rest接口计数从1开始。

List instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");

如:

- List [0] instances = 127.0.0.1:8002
- List [1] instances = 127.0.0.1:8001

8001+ 8002组合成为集群, 它们共计2台机器, 集群总数为2, 按照轮询算法原理:

- 当总请求数为1时: $1\%2=1$ 对应下标位置为1, 则获得服务地址为127.0.0.1:8001
- 当总请求数位2时: $2\%2=0$ 对应下标位置为0, 则获得服务地址为127.0.0.1:8002
- 当总请求数位3时: $3\%2=1$ 对应下标位置为1, 则获得服务地址为127.0.0.1:8001
- 当总请求数位4时: $4\%2=0$ 对应下标位置为0, 则获得服务地址为127.0.0.1:8002
- 如此类推...

RoundRobinRule源码分析

```

1  public interface IRule{
2      /*
3      * choose one alive server from lb.allServers or
4      * lb.upServers according to key
5      *
6      * @return chosen Server object. NULL is returned if none
7      * server is available
8      */
9
10 //重点关注这方法
11 public Server choose(Object key);
12
13 public void setLoadBalancer(ILoadBalancer lb);
14
15 public ILoadBalancer getLoadBalancer();
16 }

```

```
1 package com.netflix.loadbalancer;
2
3 import com.netflix.client.config.IClientConfig;
4 import org.slf4j.Logger;
5 import org.slf4j.LoggerFactory;
6
7 import java.util.List;
8 import java.util.concurrent.atomic.AtomicInteger;
9
10 /**
11 * The most well known and basic load balancing strategy, i.e. Round Robin
12 * Rule.
13 *
14 * @author stonse
15 * @author Nikos Michalakis <nikos@netflix.com>
16 *
17 */
18 public class RoundRobinRule extends AbstractLoadBalancerRule {
19
20     private AtomicInteger nextServerCyclicCounter;
21     private static final boolean AVAILABLE_ONLY_SERVERS = true;
22     private static final boolean ALL_SERVERS = false;
23
24     private static Logger log =
25 LoggerFactory.getLogger(RoundRobinRule.class);
26
27     public RoundRobinRule() {
28         nextServerCyclicCounter = new AtomicInteger(0);
29     }
30
31     public RoundRobinRule(ILoadBalancer lb) {
32         this();
33         setLoadBalancer(lb);
34     }
35
36     //重点关注这方法。
37     public Server choose(ILoadBalancer lb, Object key) {
38         if (lb == null) {
39             log.warn("no load balancer");
40             return null;
41         }
42
43         Server server = null;
44         int count = 0;
45         while (server == null && count++ < 10) {
46             List<Server> reachableServers = lb.getReachableServers();
47             List<Server> allServers = lb.getAllServers();
48             int upCount = reachableServers.size();
49             int serverCount = allServers.size();
50
51             if ((upCount == 0) || (serverCount == 0)) {
52                 log.warn("No up servers available from load balancer: " +
53                         lb);
54                 return null;
55             }
56
57             int nextServerIndex = incrementAndGetModulo(serverCount);
58             server = allServers.get(nextServerIndex);
59         }
60     }
61
62     private int incrementAndGetModulo(int serverCount) {
63         int result = nextServerCyclicCounter.getAndIncrement();
64         if (result >= serverCount) {
65             result = 0;
66         }
67         return result;
68     }
69 }
```

```

56
57     if (server == null) {
58         /* Transient. */
59         Thread.yield();
60         continue;
61     }
62
63     if (server.isAlive() && (server.isReadyToServe())) {
64         return (server);
65     }
66
67     // Next.
68     server = null;
69 }
70
71     if (count >= 10) {
72         log.warn("No available alive servers after 10 tries from load
73 balancer: " + 1b);
74     }
75     return server;
76 }
77
78 /**
79 * Inspired by the implementation of {@link
80 AtomicInteger#incrementAndGet()}.
81 *
82 * @param modulo The modulo to bound the value of the counter.
83 * @return The next value.
84 */
85 private int incrementAndGetModulo(int modulo) {
86     for (;;) {
87         int current = nextServerCyclicCounter.get();
88         int next = (current + 1) % modulo;//求余法
89         if (nextServerCyclicCounter.compareAndSet(current, next))
90             return next;
91     }
92 }
93
94 @Override
95 public Server choose(Object key) {
96     return choose(getLoadBalancer(), key);
97 }
98
99 @Override
100 public void initWithNwscConfig(IClientConfig clientConfig) {
101 }

```

Ribbon之手写轮询算法

自己试着写一个类似RoundRobinRule的本地负载均衡器。

- 7001/7002集群启动
- 8001/8002微服务改造- controller

```

1  @RestController
2  @Slf4j
3  public class PaymentController{
4
5      ...
6
7      @GetMapping(value = "/payment/lb")
8      public String getPaymentLB() {
9          //返回服务接口
10         return serverPort;
11     }
12     ...
13 }

```

- 80订单微服务改造

1.ApplicationContextConfig去掉注解@LoadBalanced， OrderMain80去掉注解@RibbonClient

```

1 import org.springframework.cloud.client.loadbalancer.LoadBalanced;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.web.client.RestTemplate;
5
6 @Configuration
7 public class ApplicationContextConfig {
8
9     @Bean
10     //@LoadBalanced
11     public RestTemplate getRestTemplate(){
12         return new RestTemplate();
13     }
14 }
15 }

```

2.创建LoadBalancer接口

```

1 import org.springframework.cloud.client.ServiceInstance;
2 import java.util.List;
3
4 public interface LoadBalancer{
5     ServiceInstance instances(List<ServiceInstance> serviceInstances);
6 }

```

3.MyLB

实现LoadBalancer接口

```

1 import org.springframework.cloud.client.ServiceInstance;
2 import org.springframework.stereotype.Component;
3
4 import java.util.List;
5 import java.util.concurrent.atomic.AtomicInteger;
6
7 /**
8 */
9 @Component//需要跟主启动类同包，或者在其子孙包下。

```

```

10 public class MyLB implements LoadBalancer
11 {
12
13     private AtomicInteger atomicInteger = new AtomicInteger(0);
14
15     public final int getAndIncrement()
16     {
17         int current;
18         int next;
19
20         do {
21             current = this.atomicInteger.get();
22             next = current >= 2147483647 ? 0 : current + 1;
23             }while(!this.atomicInteger.compareAndSet(current,next));
24             log.info("*****第几次访问，次数: next: "+next);
25             return next;
26     }
27
28     //负载均衡算法: rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标 ，
29     //每次服务重启动后rest接口计数从1开始。
30     @Override
31     public ServiceInstance instances(List<ServiceInstance> serviceInstances)
32     {
33         int index = getAndIncrement() % serviceInstances.size();
34
35         return serviceInstances.get(index);
36     }

```

4.OrderController

```

1 import org.springframework.cloud.client.ServiceInstance;
2 import org.springframework.cloud.client.discovery.DiscoveryClient;
3 import com.lun.springcloud.lb.LoadBalancer;
4
5 @Slf4j
6 @RestController
7 public class OrderController {
8
9     //public static final String PAYMENT_URL = "http://localhost:8001";
10    public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";
11
12    ...
13
14    @Resource
15    private LoadBalancer loadBalancer;
16
17    @Resource
18    private DiscoveryClient discoveryClient;
19
20    ...
21
22    @GetMapping(value = "/consumer/payment/lb")
23    public String getPaymentLB()
24    {
25        List<ServiceInstance> instances =
discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");

```

```
26
27     if(instances == null || instances.size() <= 0){
28         return null;
29     }
30
31     ServiceInstance serviceInstance = loadBalancer.instances(instances);
32     URI uri = serviceInstance.getUri();
33
34     return restTemplate.getForObject(uri+"/payment/lb",String.class);
35
36 }
37 }
```

5. 测试 不停地刷新<http://localhost/consumer/payment/lb>, 可以看到8001/8002交替出现。

OpenFeign

基本概念

[官方文档](#)

[Github地址](#)

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka, as well as Spring Cloud LoadBalancer to provide a load-balanced http client when using Feign. link

Feign是一个声明式WebService客户端。使用Feign能让编写Web Service客户端更加简单。它的使用方法是[定义一个服务接口然后在上面添加注解](#)。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

Feign能干什么

Feign旨在使编写Java Http客户端变得更容易。

前面在使用Ribbon + RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，[往往一个接口会被多处调用](#)，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，[我们只需创建一个接口并使用注解的方式来配置它\(以前是Dao接口上面标注Mapper注解,现在是一个微服务接口上面标注一个Feign注解即可\)](#)，即可完成对服务提供方的接口绑定，简化了使用Spring cloud Ribbon时，自动封装服务调用客户端的开发量。

Feign集成了Ribbon

利用Ribbon维护了Payment的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，[通过feign只需要定义服务绑定接口且以声明式的方法](#)，优雅而简单的实现了服务调用。

Feign和OpenFeign两者区别

Feign	OpenFeign
Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。Feign的使用方式是：使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务。	OpenFeign是Spring Cloud在Feign的基础上支持了SpringMVC的注解，如@RequesMapping等等。OpenFeign的@Feignclient可以解析SpringMVC的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。
org.springframework.cloud spring-cloud-starter-feign	org.springframework.cloud spring-cloud-starter-openfeign

feign
 英 [feɪn] 美 [feɪn]
 v. 假装，装作，佯装(有某种感觉或生病、疲倦等)

OpenFeign服务调用

接口+注解：微服务调用接口 + @FeignClient

1.新建模块

名字：cloud-consumer-feign-order80

2.POM

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5   http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <parent>
7     <artifactId>LearnCloud</artifactId>
8     <groupId>com.lun.springcloud</groupId>
9     <version>1.0.0-SNAPSHOT</version>
10    </parent>
11    <modelVersion>4.0.0</modelVersion>
12
13    <artifactId>cloud-consumer-feign-order80</artifactId>
14
15    <dependencies>
16      <!--openfeign-->
17      <dependency>
18        <groupId>org.springframework.cloud</groupId>
19        <artifactId>spring-cloud-starter-openfeign</artifactId>
20      </dependency>
21      <!--eureka client-->
22      <dependency>
23        <groupId>org.springframework.cloud</groupId>
24        <artifactId>spring-cloud-starter-netflix-eureka-
25        client</artifactId>
26      </dependency>
27      <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
28      <dependency>
29        <groupId>org.example</groupId>

```

```
28         <artifactId>cloud-api-commons</artifactId>
29         <version>${project.version}</version>
30     </dependency>
31     <!--web-->
32     <dependency>
33         <groupId>org.springframework.boot</groupId>
34         <artifactId>spring-boot-starter-web</artifactId>
35     </dependency>
36     <dependency>
37         <groupId>org.springframework.boot</groupId>
38         <artifactId>spring-boot-starter-actuator</artifactId>
39     </dependency>
40     <!--一般基础通用配置-->
41     <dependency>
42         <groupId>org.springframework.boot</groupId>
43         <artifactId>spring-boot-devtools</artifactId>
44         <scope>runtime</scope>
45         <optional>true</optional>
46     </dependency>
47     <dependency>
48         <groupId>org.projectlombok</groupId>
49         <artifactId>lombok</artifactId>
50         <optional>true</optional>
51     </dependency>
52     <dependency>
53         <groupId>org.springframework.boot</groupId>
54         <artifactId>spring-boot-starter-test</artifactId>
55         <scope>test</scope>
56     </dependency>
57   </dependencies>
58 </project>
```

3.YML

```
1 server:
2   port: 80
3
4 eureka:
5   client:
6     register-with-eureka: false
7     service-url:
8       defaultZone:
      http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/
```

4.主启动

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.openfeign.EnableFeignClients;
4
5 @SpringBootApplication
6 @EnableFeignClients
7 public class OrderFeignMain80 {
8     public static void main(String[] args) {
9         SpringApplication.run(OrderFeignMain80.class, args);
10    }
11 }

```

5.业务类

业务逻辑接口+@FeignClient配置调用provider服务

新建PaymentFeignService接口并新增注解@FeignClient

```

1 import com.lun.springcloud.entities.CommonResult;
2 import com.lun.springcloud.entities.Payment;
3 import org.springframework.cloud.openfeign.FeignClient;
4 import org.springframework.stereotype.Component;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7
8
9 @Component
10 @FeignClient(value = "CLOUD-PAYMENT-SERVICE")
11 public interface PaymentFeignService
12 {
13     @GetMapping(value = "/payment/get/{id}")
14     public CommonResult<Payment> getPaymentById(@PathVariable("id") Long
15     id);
16 }

```

控制层Controller

```

1 import com.example.springcloud.pojo.CommonResult;
2 import com.example.springcloud.pojo.Payment;
3 import com.example.springcloud.service.PaymentFeignService;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RestController;
9
10 import javax.annotation.Resource;
11
12 @Slf4j
13 @RestController
14 @RequestMapping("/consumer")
15 public class OrderFeignController {
16
17     @Resource
18     private PaymentFeignService paymentFeignService;

```

```

19
20     /**
21      * 通过id查订单
22      * @param id 订单id
23      * @return 订单
24     */
25     @GetMapping("/payment/get/{id}")
26     public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id)
27     {
28         return paymentFeignService.getPaymentById(id);
29     }

```

6. 测试

先启动2个eureka集群7001/7002

再启动2个微服务8001/8002

启动OpenFeign启动

<http://localhost/consumer/payment/get/1>

Feign自带负载均衡配置项



OpenFeign超时控制

超时设置，故意设置超时演示出错情况

1. 服务提供方8001/8002故意写暂停程序

```

1  @RestController
2  @Slf4j
3  @RequestMapping("/payment")
4  public class PaymentController {
5
6      ...
7
8      @Value("${server.port}")
9      private String serverPort;
10
11     ...
12
13     @GetMapping(value = "/feign/timeout")
14     public String paymentFeignTimeout()

```

```
15    {
16        // 业务逻辑处理正确，但是需要耗费3秒钟
17        try {
18            TimeUnit.SECONDS.sleep(3);
19        } catch (InterruptedException e) {
20            e.printStackTrace();
21        }
22        return serverPort;
23    }
24
25    ...
26}
```

2.服务消费方80添加超时方法PaymentFeignService

```
1 @Component
2 @FeignClient(value = "CLOUD-PAYMENT-SERVICE")
3 public interface PaymentFeignService{
4
5     ...
6
7     @GetMapping(value = "/payment/feign/timeout")
8     public String paymentFeignTimeout();
9 }
```

3.服务消费方80添加超时方法OrderFeignController

```
1 @RestController
2 @Slf4j
3 @RequestMapping("/consumer")
4 public class OrderFeignController
5 {
6     @Resource
7     private PaymentFeignService paymentFeignService;
8
9     ...
10
11    @GetMapping(value = "/payment/feign/timeout")
12    public String paymentFeignTimeout()
13    {
14        // OpenFeign客户端一般默认等待1秒钟
15        return paymentFeignService.paymentFeignTimeout();
16    }
17}
```

4.测试：

多次刷新 <http://localhost/consumer/payment/feign/timeout>

将会跳出错误Spring Boot默认错误页面，主要异常： feign.RetryableException:Read timed out executing GET <http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout>。

OpenFeign默认等待1秒钟，超过后报错

YML文件里需要开启OpenFeign客户端超时控制

```
1 #设置feign客户端超时时间(OpenFeign默认支持ribbon)(单位: 毫秒)
2 ribbon:
3     #指的是建立连接所用的时间, 适用于网络状况正常的情况下,两端连接所用的时间
4     ReadTimeout: 5000
5     #指的是建立连接后从服务器读取到可用资源所用的时间
6     ConnectTimeout: 5000
```

OpenFeign日志增强

日志打印功能

Feign提供了日志打印功能，我们可以通过配置来调整日志级别，从而了解Feign中Http请求的细节。

说白了就是对Feign接口的调用情况进行监控和输出

日志级别

- NONE：默认的，不显示任何日志；
- BASIC：仅记录请求方法、URL、响应状态码及执行时间；
- HEADERS：除了BASIC中定义的信息之外，还有请求和响应的头信息；
- FULL：除了HEADERS中定义的信息之外，还有请求和响应的正文及元数据。

配置日志bean

```
1 import feign.Logger;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4
5 @Configuration
6 public class FeignConfig{
7     @Bean
8     Logger.Level feignLoggerLevel(){
9         {
10             return Logger.Level.FULL;
11         }
12     }
13 }
```

YML文件里需要开启日志的Feign客户端

```
1 logging:
2     level:
3         # feign日志以什么级别监控哪个接口
4         com.example.springcloud.service.PaymentFeignService: debug
```

后台日志查看

得到更多日志信息。

```
[J]ServerList.org.springframework.framework.net.Helix.ribbon.EurekaDomainExtractingServerList@055cades7
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] <--- HTTP/1.1 200 (248ms)
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] connection: keep-alive
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] content-type: application/json
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] date: Mon, 18 Jul 2022 07:37:14 GMT
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] keep-alive: timeout=60
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] transfer-encoding: chunked
2022-07-18 15:37:14.110 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById]
2022-07-18 15:37:14.112 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] {"code":200,"message":"查询成功,serverPort: 8001","data":{"id":31,"serial":"尚硅谷111"}}
2022-07-18 15:37:14.112 DEBUG 19852 --- [p-nio-80-exec-1] c.e.s.service.PaymentFeignService      :
[PaymentFeignService#getPaymentById] <--- END HTTP (95-byte body)
```

服务降级

Hystrix

概述

分布式系统面临的问题

复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免地失败。

服务雪崩

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它的微服务，这就是所谓的“**扇出**”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“**雪崩效应**”。

对于高流量的应用来说，单一的后避依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

所以，通常当你发现一个模块下的某个实例失败后，这时候这个模块依然还会接收流量，然后这个有问题的模块还调用了其他的模块，这样就会发生级联故障，或者叫雪崩。

Hystrix是什么

Hystrix是一个用于处理分布式系统的**延迟**和**容错**的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性**。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），**向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常**，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

hystrix
n. 豪猪属;猬草属;豪猪;豪猪亚属

Hystrix停更进维

能干嘛

- 服务降级
- 服务熔断
- 接近实时的监控
- ...

官网资料

[link](#)

Hystrix官宣，停更进维

[link](#)

- 被动修bugs
- 不再接受合并请求
- 不再发布新版本

Hystrix的服务降级熔断限流概念初讲

服务降级

服务器忙，请稍后再试，不让客户端等待并立刻返回一个友好提示，fallback

哪些情况会触发降级

- 程序运行异常
- 超时
- 服务熔断触发服务降级
- 线程池/信号量打满也会导致服务降级

服务熔断

类比保险丝达到最大服务访问后，直接拒绝访问，拉闸限电，然后调用服务降级的方法并返回友好提示。

服务的降级 -> 进而熔断 -> 恢复调用链路

服务限流

秒杀高并发等操作，严禁一窝蜂的过来拥挤，大家排队，一秒钟N个，有序进行。

Hystrix支付微服务构建

订单微服务提供者

将cloud-eureka-server7001改配置成单机版

1.新建模块

名字：cloud-provider-hygtrix-payment8001

2.POM

```
1 <dependencies>
2     <!--hystrix-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
6     </dependency>
7     <!--eureka client-->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-netflix-eureka-
11        client</artifactId>
12    </dependency>
```

```

12      <!--web-->
13      <dependency>
14          <groupId>org.springframework.boot</groupId>
15          <artifactId>spring-boot-starter-web</artifactId>
16      </dependency>
17      <dependency>
18          <groupId>org.springframework.boot</groupId>
19          <artifactId>spring-boot-starter-actuator</artifactId>
20      </dependency>
21      <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
22      <dependency>
23          <groupId>org.example</groupId>
24          <artifactId>cloud-api-commons</artifactId>
25          <version>${project.version}</version>
26      </dependency>
27      <dependency>
28          <groupId>org.springframework.boot</groupId>
29          <artifactId>spring-boot-devtools</artifactId>
30          <scope>runtime</scope>
31          <optional>true</optional>
32      </dependency>
33      <dependency>
34          <groupId>org.projectlombok</groupId>
35          <artifactId>lombok</artifactId>
36          <optional>true</optional>
37      </dependency>
38      <dependency>
39          <groupId>org.springframework.boot</groupId>
40          <artifactId>spring-boot-starter-test</artifactId>
41          <scope>test</scope>
42      </dependency>
43  </dependencies>

```

3.YML

```

1 server:
2     port: 8001
3
4 spring:
5     application:
6         name: cloud-provider-hystrix-payment
7
8 eureka:
9     client:
10        register-with-eureka: true
11        fetch-registry: true
12        service-url:
13            #defaultZone:
14                http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
15                defaultZone: http://eureka7001.com:7001/eureka

```

4.主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
4
5 @SpringBootApplication
6 @EnableEurekaClient
7 public class PaymentHystrixMain8001{
8     public static void main(String[] args) {
9         SpringApplication.run(PaymentHystrixMain8001.class, args);
10    }
11 }
```

5.业务类

service

```
1 import org.springframework.stereotype.Service;
2 import java.util.concurrent.TimeUnit;
3
4 @Service
5 public class PaymentService {
6
7     /**
8      * 正常访问 ok
9      * @param id id
10     * @return 字符串
11     */
12     public String paymentInfo_OK(Integer id) {
13         return "线程池: "+Thread.currentThread().getName()+""
14         paymentInfo_OK,id: "+id+"\t"+0(n_n)0哈哈~";
15     }
16
17     /**
18      * 超时访问 error
19      * @param id id
20      * @return 字符串
21      */
22     public String paymentInfo_TimeOut(Integer id) {
23         try { TimeUnit.MILLISECONDS.sleep(3000); } catch
24         (InterruptedException e) { e.printStackTrace(); }
25         return "线程池: "+Thread.currentThread().getName()+" id:
26         "+id+"\t"+0(n_n)0哈哈~"+" 耗时(秒): 3";
27     }
28 }
```

controller

```
1 import com.lun.springcloud.service.PaymentService;
2 import lombok.extern.slf4j.Slf4j;
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RestController;
7
```

```

8 import javax.annotation.Resource;
9
10 @Slf4j
11 @RestController
12 @RequestMapping("/payment")
13 public class PaymentController {
14
15     @Resource
16     private PaymentService paymentService;
17
18     @Value("${server.port}")
19     private String serverPort;
20
21     @GetMapping("/hystrix/ok/{id}")
22     public String paymentInfo_OK(@PathVariable("id") Integer id) {
23         String result = paymentService.paymentInfo_OK(id);
24         log.info("*****result: "+result);
25         return result;
26     }
27
28     @GetMapping("/hystrix/timeout/{id}")
29     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
30         String result = paymentService.paymentInfo_TimeOut(id);
31         log.info("*****result: "+result);
32         return result;
33     }
34 }
```

6.正常测试

启动eureka7001

启动cloud-provider-hystrix-payment8001

访问

success的方法 - <http://localhost:8001/payment/hystrix/ok/1>

每次调用耗费5秒钟 - <http://localhost:8001/payment/hystrix/timeout/1>

上述module均OK

以上述为根基平台，从正确 -> 错误 -> 降级熔断 -> 恢复。

7.JMeter高并发压测后卡顿

上述在非高并发情形下，还能勉强满足

Jmeter压测测试

[JMeter官网](#)

The Apache JMeter™ application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.

开启Jmeter，来20000个并发压死8001，20000个请求都去访问paymentInfo_TimeOut服务

1.测试计划中右键添加->线程->线程组（线程组202102，线程数：200，线程数：100，其他参数默认）

2.刚刚新建线程组202102, 右键它-》添加-》取样器-》Http请求-》基本输入<http://localhost:8001/payment/hystrix/ok/1>

3.点击绿色三角形图标启动。

看演示结果：拖慢，原因：tomcat的默认的工作线程数被打满了，没有多余的线程来分解压力和处理。

Jmeter压测结论

上面还是服务提供者8001自己测试，假如此时外部的消费者80也来访问，那消费者只能干等，最终导致消费端80不满意，服务端8001直接被拖慢。

订单微服务调用支付服务出现卡顿

看热闹不嫌事大，80新建加入

1.新建模块

名字：cloud-consumer-feign-hystrix-order80

2.POM

```
1 <dependencies>
2     <!--openfeign-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-openfeign</artifactId>
6     </dependency>
7     <!--hystrix-->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
11    </dependency>
12    <!--eureka client-->
13    <dependency>
14        <groupId>org.springframework.cloud</groupId>
15        <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
16    </dependency>
17    <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
18    <dependency>
19        <groupId>org.example</groupId>
20        <artifactId>cloud-api-commons</artifactId>
21        <version>${project.version}</version>
22    </dependency>
23    <!--web-->
24    <dependency>
25        <groupId>org.springframework.boot</groupId>
26        <artifactId>spring-boot-starter-web</artifactId>
27    </dependency>
28    <dependency>
29        <groupId>org.springframework.boot</groupId>
30        <artifactId>spring-boot-starter-actuator</artifactId>
31    </dependency>
32    <!--一般基础通用配置-->
33    <dependency>
34        <groupId>org.springframework.boot</groupId>
35        <artifactId>spring-boot-devtools</artifactId>
36        <scope>runtime</scope>
```

```

37      <optional>true</optional>
38    </dependency>
39    <dependency>
40      <groupId>org.projectlombok</groupId>
41      <artifactId>lombok</artifactId>
42      <optional>true</optional>
43    </dependency>
44    <dependency>
45      <groupId>org.springframework.boot</groupId>
46      <artifactId>spring-boot-starter-test</artifactId>
47      <scope>test</scope>
48    </dependency>
49  </dependencies>

```

3.YML

```

1 server:
2   port: 80
3
4 eureka:
5   client:
6     register-with-eureka: false
7     service-url:
8       defaultZone: http://eureka7001.com:7001/eureka/

```

4.主启动

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.hystrix.EnableHystrix;
4 import org.springframework.cloud.openfeign.EnableFeignClients;
5
6 @SpringBootApplication
7 @EnableFeignClients
8 //@EnableHystrix
9 public class OrderHystrixMain80{
10   public static void main(String[] args){
11     SpringApplication.run(OrderHystrixMain80.class,args);
12   }
13 }

```

5.业务类

```

1 import org.springframework.cloud.openfeign.FeignClient;
2 import org.springframework.stereotype.Component;
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5
6 @Component
7 @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT" /*,fallback =
PaymentFallbackService.class*/)
8 public interface PaymentHystrixService{
9   @GetMapping("/payment/hystrix/ok/{id}")
10  String paymentInfo_OK(@PathVariable("id") Integer id);
11
12  @GetMapping("/payment/hystrix/timeout/{id}")

```

```
13     String paymentInfo_TimeOut(@PathVariable("id") Integer id);  
14 }
```

controller

```
1 import com.lun.springcloud.service.PaymentHystrixService;  
2 import lombok.extern.slf4j.Slf4j;  
3 import org.springframework.web.bind.annotation.GetMapping;  
4 import org.springframework.web.bind.annotation.PathVariable;  
5 import org.springframework.web.bind.annotation.RestController;  
6 import javax.annotation.Resource;  
7  
8 @Slf4j  
9 @RestController  
10 @RequestMapping("/consumer")  
11 public class OrderHystrixController {  
12     @Resource  
13     private PaymentHystrixService paymentHystrixService;  
14  
15     @GetMapping("/payment/hystrix/ok/{id}")  
16     public String paymentInfo_OK(@PathVariable("id") Integer id) {  
17         return paymentHystrixService.paymentInfo_OK(id);  
18     }  
19  
20     @GetMapping("/payment/hystrix/timeout/{id}")  
21     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {  
22         return paymentHystrixService.paymentInfo_TimeOut(id);  
23     }  
24 }
```

6.正常测试

<http://localhost/consumer/payment/hystrix/ok/1>

7.高并发测试

2W个线程压8001

消费端80微服务再去访问正常的Ok微服务8001地址

<http://localhost/consumer/payment/hystrix/ok/32>

消费者80被拖慢

原因：8001同一层次的其它接口服务被困死，因为tomcat线程池里面的工作线程已经被挤占完毕。

正因为有上述故障或不佳表现才有我们的降级/容错/限流等技术诞生。

Hystrix,服务降级

超时导致服务器变慢(转圈) - 超时不再等待

出错(宕机或程序运行出错) - 出错要有兜底

解决：

- 对方服务(8001)超时了，调用者(80)不能一直卡死等待，必须有服务降级。
- 对方服务(8001)down机了，调用者(80)不能一直卡死等待，必须有服务降级。

- 对方服务(8001)OK, 调用者(80)自己出故障或有自我要求(自己的等待时间小于服务提供者), 自己处理降级。

服务降级支付测fallback(8001)

降级配置 - @HystrixCommand

8001先从自身找问题

设置自身调用超时时间的峰值, 峰值内可以正常运行, 超过了需要有兜底的方法处理, 作服务降级 fallback。

8001fallback

业务类启用 - **@HystrixCommand**报异常后如何处理

一旦调用服务方法失败并抛出了错误信息后, 会自动调用@HystrixCommand标注好的fallbackMethod 调用类中的指定方法

```

1  @Service
2  public class PaymentService {
3      /**
4          * 超时访问 error
5          * @param id id
6          * @return 字符串
7          */
8      @HystrixCommand(fallbackMethod =
9          "paymentInfo_TimeOutHandler", commandProperties = {
10             @HystrixProperty(name =
11                 "execution.isolation.thread.timeoutInMilliseconds", value = "3000")
12         })
13      public String paymentInfo_TimeOut(Integer id) {
14          //int age = 10/0;
15          int timeNumber = 5;
16          try { TimeUnit.SECONDS.sleep(timeNumber); } catch
17          (InterruptedException e) { e.printStackTrace(); }
18          return "线程池: "+Thread.currentThread().getName()+" id:
19          "+id+"\t"+"o(∩_∩)o哈哈~"+" 耗时(秒): "+timeNumber;
20      }
21
22      /**
23          * 用来善后的办法
24          * @param id id
25          * @return 字符串
26          */
27      public String paymentInfo_TimeOutHandler(Integer id) {
28
29          return "线程池: "+Thread.currentThread().getName()+" 8001系统繁忙或者
30          运行报错, 请稍后再试, id: "+id+"\t"+"o(╥﹏╥)o";
31      }
32  }
```

上面故意制造两种异常:

1. int age = 10/0, 计算异常
2. 我们能接受3秒钟, 它运行5秒钟, 超时异常。

当前服务不可用了, 做服务降级, 兜底的方案都是paymentInfo_TimeOutHandler

主启动类激活

添加新注解@EnableCircuitBreaker

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
4 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
5
6 @SpringBootApplication
7 @EnableEurekaClient
8 @EnableCircuitBreaker//添加到此处
9 public class PaymentHystrixMain8001{
10     public static void main(String[] args) {
11         SpringApplication.run(PaymentHystrixMain8001.class, args);
12     }
13 }
```

服务降级订单测fallback(80)

80订单微服务，也可以更好的保护自己，自己也依样画葫芦进行客户端降级保护

题外话，切记 - 我们自己配置过的热部署方式对java代码的改动明显

但对@HystrixCommand内属性的修改建议重启微服务

YML

```
1 server:
2   port: 80
3
4 eureka:
5   client:
6     register-with-eureka: false
7     service-url:
8       defaultZone: http://eureka7001.com:7001/eureka/
9
10 #开启
11 feign:
12   hystrix:
13     enabled: true
```

主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.hystrix.EnableHystrix;
4 import org.springframework.cloud.openfeign.EnableFeignClients;
5
6 @SpringBootApplication
7 @EnableFeignClients
8 @EnableHystrix//添加到此处
9 public class OrderHystrixMain80{
10     public static void main(String[] args){
11         SpringApplication.run(OrderHystrixMain80.class,args);
12     }
13 }
```

业务类

```
1 import com.example.springcloud.service.PaymentHystrixService;
2 import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
3 import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RestController;
9 import javax.annotation.Resource;
10
11 @Slf4j
12 @RestController
13 @RequestMapping("/consumer")
14 public class OderHystrixController {
15     @Resource
16     private PaymentHystrixService paymentHystrixService;
17
18     @GetMapping("/payment/hystrix/ok/{id}")
19     public String paymentInfo_OK(@PathVariable("id") Integer id) {
20         return paymentHystrixService.paymentInfo_OK(id);
21     }
22
23
24     @GetMapping("/payment/hystrix/timeout/{id}")
25     @HystrixCommand(fallbackMethod =
26 "paymentTimeOutFallbackMethod", commandProperties = {
27         @HystrixProperty(name =
28 "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
29     })
30     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
31         return paymentHystrixService.paymentInfo_TimeOut(id);
32     }
33
34     public String paymentTimeOutFallbackMethod(@PathVariable("id") Integer id) {
35         return "我是消费者80,对方支付系统繁忙请10秒钟后再试或者自己运行出错请检查自己,o(╥﹏╥)o";
36     }
37 }
```

全局服务降级DefaultProperties

目前问题1 每个业务方法对应一个兜底的方法，代码膨胀

解决方法

1:1 每个方法配置一个服务降级方法，技术上可以，但是不聪明

1:N 除了个别重要核心业务有专属，其它普通的可以通过@DefaultProperties(defaultFallback = "")统一跳转到统一处理结果页面

通用的和独享的各自分开，避免了代码膨胀，合理减少了代码量

```
1 import com.example.springcloud.service.PaymentHystrixService;
2 import com.netflix.hystrix.contrib.javanica.annotation.DefaultProperties;
3 import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
```

```

4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RestController;
9
10 import javax.annotation.Resource;
11
12 @Slf4j
13 @RestController
14 @RequestMapping("/consumer")
15 @DefaultProperties(defaultFallback = "paymentGlobalFallbackMethod")
16 public class OderHystrixController {
17     @Resource
18     private PaymentHystrixService paymentHystrixService;
19
20     @GetMapping("/payment/hystrix/ok/{id}")
21     public String paymentInfo_OK(@PathVariable("id") Integer id) {
22         return paymentHystrixService.paymentInfo_OK(id);
23     }
24
25     @GetMapping("/payment/hystrix/timeout/{id}")
26     //@HystrixCommand(fallbackMethod =
27     //    "paymentTimeOutFallbackMethod", commandProperties = {
28     //        //          @HystrixProperty(name =
29     //            "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
30     //    })
31     @HystrixCommand      //用全局的fallback方法
32     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
33         return paymentHystrixService.paymentInfo_TimeOut(id);
34     }
35
36     public String paymentTimeOutFallbackMethod(@PathVariable("id") Integer id) {
37         return "我是消费者80,对方支付系统繁忙请10秒钟后再试或者自己运行出错请检查自己,o(╥﹏╥)o";
38     }
39
40     /**
41      *下面是全局fallback方法
42      */
43     public String paymentGlobalFallbackMethod(){
44         return "Global异常处理信息, 请稍后再试, /(ㄒoㄒ)/~~";
45     }
46 }

```

通配服务降级FeignFallback

目前问题2 统一和自定义的分开，代码混乱

服务降级，客户端去调用服务端，碰上服务端宕机或关闭

本次案例服务降级处理是在客户端80实现完成的，与服务端8001没有关系，只需要为Feign客户端定义的接口添加一个服务降级处理的实现类即可实现解耦

未来我们要面对的异常

- 运行

- 超时
- 容机

修改cloud-consumer-feign-hystrix-order80

根据cloud-consumer-feign-hystrix-order80已经有的PaymentHystrixService接口，

重新新建一个类(PaymentFallbackService)实现该接口，统一为接口里面的方法进行异常处理

PaymentFallbackService类实现PaymentHystrixService接口

```

1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class PaymentFallbackService implements PaymentHystrixService{
5     @Override
6     public String paymentInfo_OK(Integer id) {
7         return "-----PaymentFallbackService fall back-paymentInfo_OK ,o(╥﹏╥)o";
8     }
9
10    @Override
11    public String paymentInfo_TimeOut(Integer id) {
12        return "-----PaymentFallbackService fall back-paymentInfo_TimeOut
13        ,o(╥﹏╥)o";
14    }

```

YML

```

1 server:
2   port: 80
3
4 eureka:
5   client:
6     register-with-eureka: false
7     service-url:
8       defaultZone: http://eureka7001.com:7001/eureka/
9
10 #开启
11 feign:
12   hystrix:
13     enabled: true

```

PaymentHystrixService接口

```

1 import org.springframework.cloud.openfeign.FeignClient;
2 import org.springframework.stereotype.Component;
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5
6 @Component
7 @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT",
8             fallback = PaymentFallbackService.class) //指定
9 PaymentFallbackService类
10 public interface PaymentHystrixService {
11     @GetMapping("/payment/hystrix/ok/{id}")

```

```

11     String paymentInfo_OK(@PathVariable("id") Integer id);
12
13     @GetMapping("/payment/hystrix/timeout/{id}")
14     String paymentInfo_TimeOut(@PathVariable("id") Integer id);
15 }

```

测试

单个eureka先启动7001

PaymentHystrixMain8001启动

正常访问测试 - <http://localhost/consumer/payment/hystrix/ok/31>

故意关闭微服务8001

客户端自己调用提示 - 此时服务端provider已经down了，但是我们做了服务降级处理，让客户端在服务端不可用时也会获得提示信息而不会挂起耗死服务器。

Hystrix服务熔断

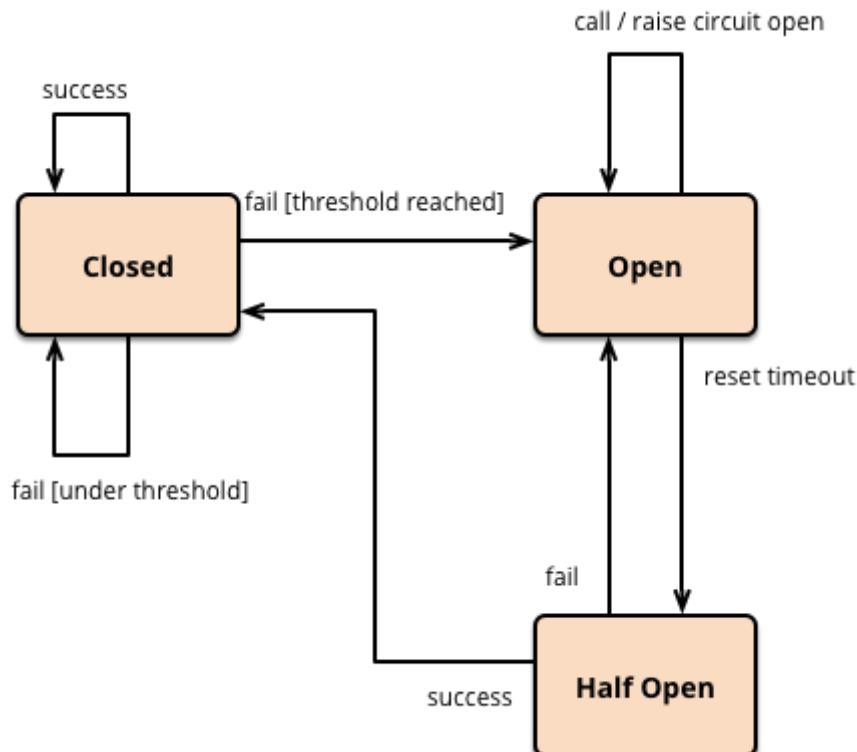
断路器，相当于保险丝。

熔断机制概述

熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务出错不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息。**当检测到该节点微服务调用响应正常后，恢复调用链路。**

在Spring Cloud框架里，熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败，就会启动熔断机制。熔断机制的注解是@HystrixCommand。

[Martin Fowler的相关论文](#)



服务熔断案例

Hutool国产工具类

修改cloud-provider-hystrix-payment8001

```
1 import cn.hutool.core.util.IdUtil;
2 import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
3 import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
4 import org.springframework.stereotype.Service;
5 import org.springframework.web.bind.annotation.PathVariable;
6
7 import java.util.concurrent.TimeUnit;
8
9 @Service
10 public class PaymentService{
11
12     ...
13
14     //=====服务熔断
15     @HystrixCommand(fallbackMethod =
16         "paymentCircuitBreaker_fallback", commandProperties = {
17             @HystrixProperty(name = "circuitBreaker.enabled", value =
18                 "true"), // 是否开启断路器
19             @HystrixProperty(name =
20                 "circuitBreaker.requestVolumeThreshold", value = "10"), // 请求次数
21             @HystrixProperty(name =
22                 "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), // 时间窗口期
23             @HystrixProperty(name =
24                 "circuitBreaker.errorThresholdPercentage", value = "60") // 失败率达到多少后跳闸
25         })
26     public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
27         if(id < 0) {
28             throw new RuntimeException("*****id 不能负数");
29         }
30         String serialNumber = IdUtil.simpleUUID();
31
32         return Thread.currentThread().getName()+"\t"+调用成功，流水号：" +
33             serialNumber;
34     }
35     public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id) {
36         return "id 不能负数，请稍后再试，/(ㄒoㄒ)/~~    id: " +id;
37     }
38 }
```

The precise way that the circuit opening and closing occurs is as follows:

1. Assuming the volume across a circuit meets a certain threshold :

```
HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()
```

2. And assuming that the error percentage, as defined above exceeds the error percentage defined in :

```
HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()
```

3. Then the circuit-breaker transitions from CLOSED to OPEN.

4. While it is open, it short-circuits all requests made against that circuit-breaker.
5. After some amount of time (`HystrixCommandProperties.circuitBreakersSleepWindowInMilliseconds()`), the next request is let through. If it fails, the command stays OPEN for the sleep window. If it succeeds, it transitions to CLOSED and the logic in 1) takes over again.

[link](#)

HystrixCommandProperties配置类

```

1 package com.netflix.hystrix;
2
3 ...
4
5 public abstract class HystrixCommandProperties {
6     private static final Logger logger =
7         LoggerFactory.getLogger(HystrixCommandProperties.class);
8
9     /* defaults */
10    /* package */ static final Integer
11        defaultMetricsRollingStatisticalWindow = 10000;// default =>
12        statisticalWindow: 10000 = 10 seconds (and default of 10 buckets so each
13        bucket is 1 second)
14        private static final Integer
15        defaultMetricsRollingStatisticalWindowBuckets = 10;// default =>
16        statisticalWindowBuckets: 10 = 10 buckets in a 10 second window so each
17        bucket is 1 second
18        private static final Integer
19        defaultCircuitBreakerRequestVolumeThreshold = 20;// default =>
20        statisticalWindowVolumeThreshold: 20 requests in 10 seconds must occur
21        before statistics matter
22        private static final Integer
23        defaultCircuitBreakersSleepWindowInMilliseconds = 5000;// default =>
24        sleepwindow: 5000 = 5 seconds that we will sleep before trying again after
25        tripping the circuit
26        private static final Integer
27        defaultCircuitBreakerErrorThresholdPercentage = 50;// default =>
28        errorThresholdPercentage = 50 = if 50%+ of requests in 10 seconds are
29        failures or latent then we will trip the circuit
30        private static final Boolean defaultCircuitBreakerForceOpen = false;// default =>
31        forceCircuitOpen = false (we want to allow traffic)
32        /* package */ static final Boolean defaultCircuitBreakerForceClosed =
33        false;// default => ignoreErrors = false
34        private static final Integer defaultExecutionTimeoutInMilliseconds =
35        1000; // default => executionTimeoutInMilliseconds: 1000 = 1 second
36        private static final Boolean defaultExecutionTimeoutEnabled = true;
37
38        ...
39    }

```

controller

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController

```

```

5  {
6      @Resource
7      private PaymentService paymentService;
8
9      ...
10
11     //====服务熔断
12     @GetMapping("/circuit/{id}")
13     public String paymentCircuitBreaker(@PathVariable("id") Integer id){
14         String result = paymentService.paymentCircuitBreaker(id);
15         log.info("****result: "+result);
16         return result;
17     }
18 }

```

测试

自测 cloud-provider-hystrix-payment8001

正确 - <http://localhost:8001/payment/circuit/31>

错误 - <http://localhost:8001/payment/circuit/-1>

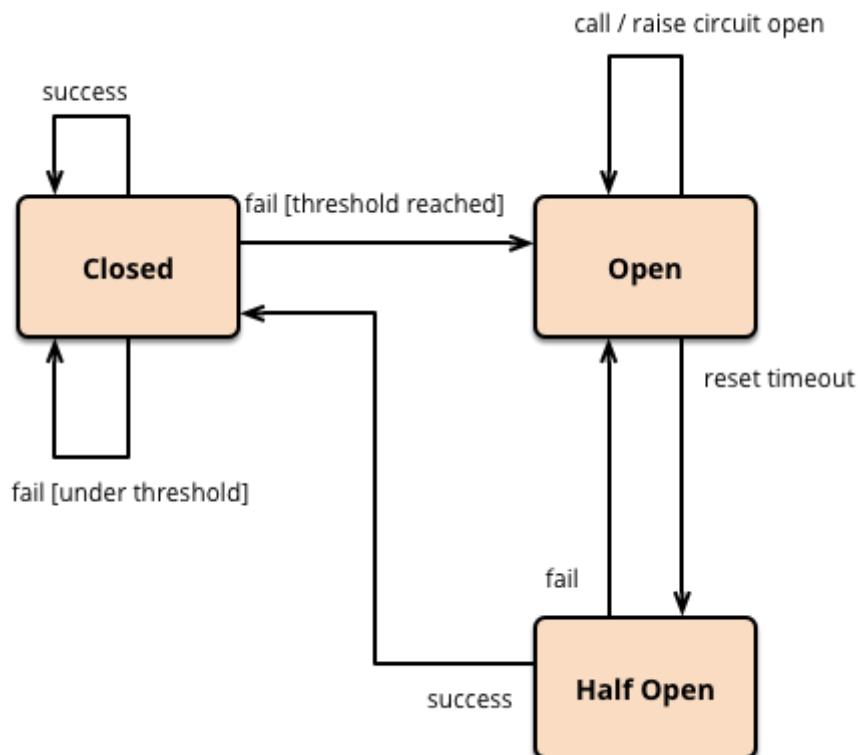
多次错误，再来次正确，但显示的是错误的

重点测试 - 多次错误，然后慢慢正确，发现刚开始不满足条件，就算是正确的访问地址也不能进行

服务熔断总结

大神结论

[Martin Fowler的相关论文](#)

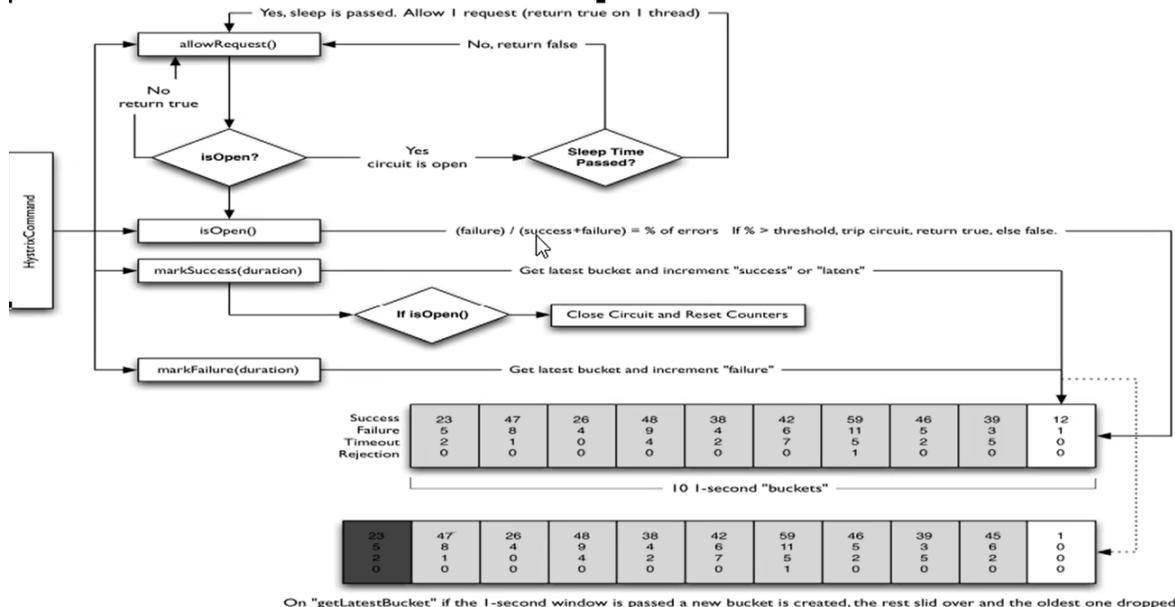


熔断类型

- 熔断打开：请求不再进行调用当前服务，内部设置时钟一般为MTTR(平均故障处理时间)，当打开时长达到所设时钟则进入半熔断状态。

- 熔断关闭：熔断关闭不会对服务进行熔断。
- 熔断半开：部分请求根据规则调用当前服务，如果请求成功且符合规则则认为当前服务恢复正常，关闭熔断。

官网断路器流程图



官网步骤

The precise way that the circuit opening and closing occurs is as follows:

1. Assuming the volume across a circuit meets a certain threshold :

`HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()`

2. And assuming that the error percentage, as defined above exceeds the error percentage defined in :

`HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()`

3. Then the circuit-breaker transitions from CLOSED to OPEN.
4. While it is open, it short-circuits all requests made against that circuit-breaker.
5. After some amount of time

(`HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()`), the next request is let through. If it fails, the command stays OPEN for the sleep window. If it succeeds, it transitions to CLOSED and the logic in 1) takes over again.

[link](#)

断路器在什么情况下开始起作用

```

1 //=====服务熔断
2 @HystrixCommand(fallbackMethod =
3     "paymentCircuitBreaker_fallback", commandProperties = {
4         @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),// 是否
5         @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value =
6             "10"),// 请求次数
7         @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value =
8             "10000"), // 时间窗口期
9         @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value =
10            "60")},// 失败率达到多少后跳闸
11     ...
12 }
13 public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
14     ...
15 }
```

错误百分比阀值

- **快照时间窗**: 断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。
- **请求数总阀值**: 在快照时间窗内，必须满足请求数总阀值才有资格熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用次数不足20次，则即使所有的请求都超时或其他原因失败，断路器都不会打开。
- **错误百分比阀值**: 当请求数在快照时间窗内超过了阀值，比如发生了30次调用，如果在这30次调用中，有15次发生了超时异常，也就是超过50%的错误百分比，在默认设定50%阀值情况下，这时候就会将断路器打开。

断路器开启或者关闭的条件

- 到达以下阀值，断路器将会开启：
 - 当满足一定的阀值的时候（默认10秒内超过20个请求数）
 - 当失败率达到一定的时候（默认10秒内超过50%的请求失败）
- 当开启的时候，所有请求都不会进行转发
- 一段时间之后（默认是5秒），这个时候断路器是半开状态，会让其中一个请求进行转发。如果成功，断路器会关闭，若失败，继续开启。

断路器打开之后

1: 再有请求调用的时候，将不会调用主逻辑，而是直接调用降级fallback。通过断路器，实现了自动地发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果。

2: 原来的主逻辑要如何恢复呢？

对于这一问题，hystrix也为我们实现了自动恢复功能。

当断路器打开，对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑上，如果此次请求正常返回，那么断路器将继续闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

All配置

```

1 @HystrixCommand(fallbackMethod = "fallbackMethod",
2     groupKey = "strGroupCommand",
3     commandKey = "strCommand",
4     threadPoolKey = "strThreadPool",
5     ...)
```

```
6         commandProperties = {  
7             // 设置隔离策略, THREAD 表示线程池 SEMAPHORE: 信号池隔离  
8             @HystrixProperty(name = "execution.isolation.strategy",  
9                 value = "THREAD"),  
10                // 当隔离策略选择信号池隔离的时候, 用来设置信号池的大小(最大并发  
11                // 数)  
12                @HystrixProperty(name =  
13                    "execution.isolation.semaphore.maxConcurrentRequests", value = "10"),  
14                    // 配置命令执行的超时时间  
15                    @HystrixProperty(name =  
16                        "execution.isolation.thread.timeoutinMilliseconds", value = "10"),  
17                        // 是否启用超时时间  
18                        @HystrixProperty(name =  
19                            "execution.timeout.enabled",  
20                                value = "true"),  
21                                // 执行超时的时候是否中断  
22                                @HystrixProperty(name =  
23                                    "execution.isolation.thread.interruptOnTimeout", value = "true"),  
24                                    // 执行被取消的时候是否中断  
25                                    @HystrixProperty(name =  
26                                        "execution.isolation.thread.interruptOnCancel", value = "true"),  
27                                        // 允许回调方法执行的最大并发数  
28                                        @HystrixProperty(name =  
29                                            "fallback.isolation.semaphore.maxConcurrentRequests", value = "10"),  
30                                            // 服务降级是否启用, 是否执行回调函数  
31                                            @HystrixProperty(name = "fallback.enabled", value =  
32                                                "true"),  
33                                                // 是否启用断路器  
34                                                @HystrixProperty(name =  
35                                                    "circuitBreaker.enabled", value = "true"),  
36                                                    // 该属性用来设置在滚动时间窗中, 断路器熔断的最小请求数。例如, 默认  
37                                                    // 该值为 20 的时候, 如果滚动时间窗(默认10秒)内仅收到了19个请求, 即使这19个请求都失败了,  
38                                                    // 断路器也不会打开。  
39                                                    @HystrixProperty(name =  
40                                                        "circuitBreaker.requestVolumeThreshold", value = "20"),  
41  
42                                                    // 该属性用来设置在滚动时间窗中, 表示在滚动时间窗中, 在请求数量超过  
43                                                    // circuitBreaker.requestVolumeThreshold 的情况下, 如果错误请求数的百分比超过50, 就把  
44                                                    // 断路器设置为 "打开" 状态, 否则就设置为 "关闭" 状态。  
45                                                    @HystrixProperty(name =  
46                                                        "circuitBreaker.errorThresholdPercentage", value = "50"),  
47                                                        // 该属性用来设置当断路器打开之后的休眠时间窗。休眠时间窗结束之  
48                                                        // 后, 会将断路器置为 "半开" 状态, 尝试熔断的请求命令, 如果依然失败就将断路器继续设置为 "打  
49                                                        // 开" 状态, 如果成功就设置为 "关闭" 状态。  
50                                                        @HystrixProperty(name =  
51                                                            "circuitBreaker.sleepwindowinMilliseconds", value = "5000"),  
52                                                            // 断路器强制打开  
53                                                            @HystrixProperty(name = "circuitBreaker.forceOpen",  
54                                                                value = "false"),  
55                                                                // 断路器强制关闭  
56                                                                @HystrixProperty(name = "circuitBreaker.forceClosed",  
57                                                                    value = "false"),  
58                                                                    // 滚动时间窗设置, 该时间用于断路器判断健康度时需要收集信息的持续时  
59                                                                   间  
60                                                                    @HystrixProperty(name =  
61                                                        "metrics.rollingstats.timeinMilliseconds", value = "10000"),  
62
```

```
40 // 该属性用来设置滚动时间窗统计指标信息时划分"桶"的数量，断路器在收  
41 //集指标信息的时候会根据设置的时间窗长度拆分成多个 "桶" 来累计各度量值，每个"桶"记录了一段时间  
42 //内的采集指标。  
43 // 比如 10 秒内拆分成 10 个"桶"收集这样，所以  
timeInMilliseconds 必须能被 numBuckets 整除。否则会抛异常  
44 @HystrixProperty(name =  
"metrics.rollingstats.numBuckets", value = "10"),  
45 // 该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算。如果  
46 // 设置为 false，那么所有的概要统计都将返回 -1。  
47 @HystrixProperty(name =  
"metrics.rollingPercentile.enabled", value = "false"),  
48 // 该属性用来设置百分位统计的滚动窗口的持续时间，单位为毫秒。  
49 @HystrixProperty(name =  
"metrics.rollingPercentile.timeInMilliseconds", value = "60000"),  
50 // 该属性用来设置百分位统计滚动窗口中使用 " 桶 " 的数量。  
51 @HystrixProperty(name =  
"metrics.rollingPercentile.numBuckets", value = "60000"),  
52 // 该属性用来设置在执行过程中每个 "桶" 中保留的最大执行次数。如果  
53 // 在滚动时间窗内发生超过该设定值的执行次数，  
54 // 就从最初的位置开始重写。例如，将该值设置为100，滚动窗口为10  
55 // 秒，若在10秒内一个 "桶 " 中发生了500次执行，  
56 // 那么该 "桶" 中只保留 最后的100次执行的统计。另外，增加该值的大  
57 // 小将会增加内存量的消耗，并增加排序百分位数所需的计算时间。  
58 @HystrixProperty(name =  
"metrics.rollingPercentile.bucketSize", value = "100"),  
59 // 该属性用来设置采集影响断路器状态的健康快照（请求的成功、 错误百  
60 // 分比）的间隔等待时间。  
61 @HystrixProperty(name =  
"metrics.healthSnapshot.intervalInMilliseconds", value = "500"),  
62 // 是否开启请求缓存  
63 @HystrixProperty(name = "requestCache.enabled", value =  
"true"),  
64 // HystrixCommand的执行和事件是否打印日志到  
HystrixRequestLog 中  
65 @HystrixProperty(name = "requestLog.enabled", value =  
"true"),  
66 },  
67 threadPoolProperties = {  
68 // 该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最  
69 // 大并发量  
70 @HystrixProperty(name = "coreSize", value = "10"),  
71 // 该参数用来设置线程池的最大队列大小。当设置为 -1 时，线程池将使  
72 // 用 SynchronousQueue 实现的队列，否则将使用 LinkedBlockingQueue 实现的队列。  
73 @HystrixProperty(name = "maxQueueSize", value = "-1"),  
74 // 该参数用来为队列设置拒绝阈值。 通过该参数， 即使队列没有达到最大  
// 值也能拒绝请求。  
75 @HystrixProperty(name = "queueSizeRejectionThreshold",  
value = "5"),  
76 }  
77 })  
78 public String doSomething() {  
79     ...  
80 }
```

工作流程最后总结

服务限流 - 后面高级篇讲解alibaba的Sentinel说明

[官方解释](#)

[官网图例](#)

步骤说明

1. 创建HystrixCommand (用在依赖的服务返回单个操作结果的时候) 或 HystrixObservableCommand (用在依赖的服务返回多个操作结果的时候) 对象。
2. 命令执行。
3. 其中 HystrixCommand实现了下面前两种执行方式
 - execute(): 同步执行，从依赖的服务返回一个单一的结果对象或是在发生错误的时候抛出异常。
 - queue(): 异步执行，直接返回一个Future对象，其中包含了服务执行结束时要返回的单一结果对象。
4. 而 HystrixObservableCommand实现了后两种执行方式：
 - observe(): 返回Observable对象，它代表了操作的多个结果，它是一个Hot Observable (不论“事件源”是否有“订阅者”，都会在创建后对事件进行发布，所以对于Hot Observable的每一个“订阅者”都有可能是从“事件源”的中途开始的，并可能只是看到了整个操作的局部过程)。
 - toObservable(): 同样会返回Observable对象，也代表了操作的多个结果，但它返回的是一个Cold Observable (没有“订阅者”的时候并不会发布事件，而是进行等待，直到有“订阅者”之后才发布事件，所以对于Cold Observable 的订阅者，它可以保证从一开始看到整个操作的全部过程)。
5. 若当前命令的请求缓存功能是被启用的，并且该命令缓存命中，那么缓存的结果会立即以 Observable对象的形式返回。
6. 检查断路器是否为打开状态。如果断路器是打开的，那么Hystrix不会执行命令，而是转接到 fallback处理逻辑(第8步)；如果断路器是关闭的，检查是否有可用资源来执行命令(第5步)。
7. 线程池/请求队列信号量是否占满。如果命令依赖服务的专有线程池和请求队列，或者信号量 (不使用线程的时候) 已经被占满，那么Hystrix也不会执行命令，而是转接到fallback处理理辑(第8步)。
8. Hystrix会根据我们编写的方法来决定采取什么样的方式去请求依赖服务。
 - HystrixCommand.run(): 返回一个单一的结果，或者抛出异常。
 - HystrixObservableCommand.construct(): 返回一个Observable对象来发射多个结果，或通过onError发送错误通知。
9. Hystrix会将“成功”、“失败”、“拒绝”、“超时” 等信息报告给断路器，而断路器会维护一组计数器来统计这些数据。断路器会使用这些统计数据来决定是否要将断路器打开，来对某个依赖服务的请求进行“熔断/短路”。
10. 当命令执行失败的时候，Hystrix会进入fallback尝试回退处理，我们通常也称波操作为“服务降级”。而能够引起服务降级处理的情况有下面几种：
 - 第4步：当前命令处于“熔断/短路”状态，断路器是打开的时候。
 - 第5步：当前命令的钱程池、请求队列或者信号量被占满的时候。
 - 第6步：HystrixObsevableCommand.construct()或HystrixCommand.run()抛出异常的时候。
11. 当Hystrix命令执行成功之后，它会将处理结果直接返回或是以Observable的形式返回。

tips: 如果我们没有为命令实现降级逻辑或者在降级处理逻辑中抛出了异常，Hystrix依然会返回一个Obsevable对象，但是它不会发射任结果数据，而是通过

onError方法通知命令立即中断请求，并通过onError方法将引起命令失败的异常发送给调用者。

服务监控HystrixDashboard

概述

除了隔离依赖服务的调用以外，Hystrix还提供了准实时的调用监控(Hystrix Dashboard)，Hystrix会持续地记录所有通过Hystrix发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求多少成功，多少失败等。

Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。Spring Cloud也提供了Hystrix Dashboard的整合，对监控内容转化成可视化界面。

实践

1.新建模块

名字：cloud-consumer-hystrix-dashboard9001

2.POM

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-netflix-hystrix-
5       dashboard</artifactId>
6   </dependency>
7   <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-actuator</artifactId>
10    </dependency>
11   <dependency>
12     <groupId>org.springframework.boot</groupId>
13     <artifactId>spring-boot-devtools</artifactId>
14     <scope>runtime</scope>
15     <optional>true</optional>
16   </dependency>
17   <dependency>
18     <groupId>org.projectlombok</groupId>
19     <artifactId>lombok</artifactId>
20     <optional>true</optional>
21   </dependency>
22   <dependency>
23     <groupId>org.springframework.boot</groupId>
24     <artifactId>spring-boot-starter-test</artifactId>
25     <scope>test</scope>
26   </dependency>
27 </dependencies>
```

3.YML

```
1 server:  
2   port: 9001
```

4.HystrixDashboardMain9001+新注解@EnableHystrixDashboard

```
1 import org.springframework.boot.SpringApplication;  
2 import org.springframework.boot.autoconfigure.SpringBootApplication;  
3 import  
4   org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  
5  
6 @SpringBootApplication  
7 @EnableHystrixDashboard  
8 public class HystrixDashboardMain9001{  
9     public static void main(String[] args) {  
10         SpringApplication.run(HystrixDashboardMain9001.class, args);  
11     }  
12 }
```

5.所有Provider微服务提供类(8001/8002/8003)都需要监控依赖配置

```
1 <dependency>  
2   <groupId>org.springframework.boot</groupId>  
3   <artifactId>spring-boot-starter-actuator</artifactId>  
4 </dependency>
```

6.启动cloud-consumer-hystrix-dashboard9001该微服务后续将监控微服务8001

浏览器输入<http://localhost:9001/hystrix>



Hystrix图形化Dashboard监控实战

修改cloud-provider-hystrix-payment8001

注意：新版本Hystrix需要在主启动类PaymentHystrixMain8001中指定监控路径

```
1 import  
2   com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet;  
3 import org.springframework.boot.SpringApplication;  
4 import org.springframework.boot.autoconfigure.SpringBootApplication;  
5 import org.springframework.boot.web.servlet.ServletRegistrationBean;  
6 import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;  
7 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
8 import org.springframework.context.annotation.Bean;  
9  
10 @SpringBootApplication  
11 @EnableEurekaClient  
12 @EnableCircuitBreaker  
13 public class PaymentHystrixMain8001 {  
14     public static void main(String[] args) {  
15         SpringApplication.run(PaymentHystrixMain8001.class, args);  
16     }  
17     /**
```

```

18     *此配置是为了服务监控而配置，与服务容错本身无关，springcloud升级后的坑
19     *ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
20     *只要在自己的项目里配置上下面的servlet就可以了
21     *否则，Unable to connect to Command Metric Stream 404
22     */
23     @Bean
24     public ServletRegistrationBean getServlet() {
25         HystrixMetricsStreamServlet streamServlet = new
26         HystrixMetricsStreamServlet();
27         ServletRegistrationBean registrationBean = new
28         ServletRegistrationBean(streamServlet);
29         registrationBean.setLoadOnStartup(1);
30         registrationBean.addUrlMappings("/hystrix.stream");
31         registrationBean.setName("HystrixMetricsStreamServlet");
32         return registrationBean;
33     }
34 }
```

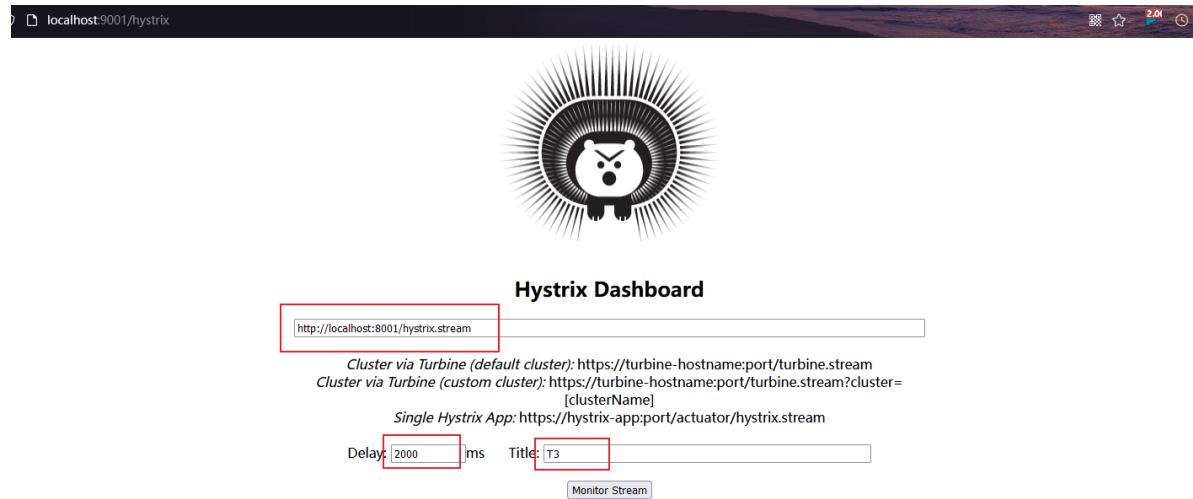
监控测试

启动1个eureka

启动8001, 9001

观察监控窗口

9001监控8001 - 填写监控地址 - <http://localhost:8001/hystrix.stream> 到 <http://localhost:9001/hystrix> 页面的输入框。



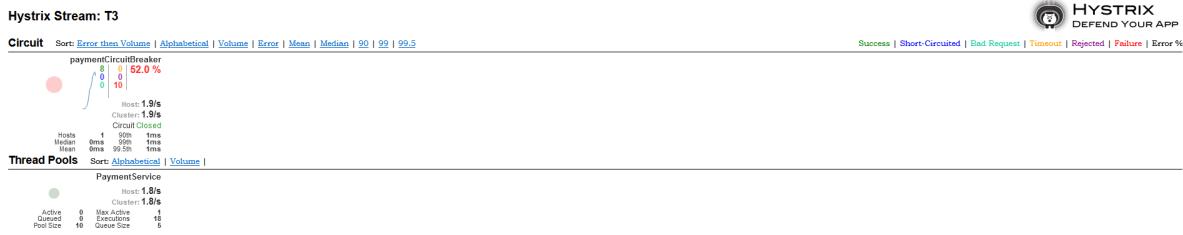
测试地址

<http://localhost:8001/payment/circuit/31>

<http://localhost:8001/payment/circuit/-1>

测试通过

先访问正确地址，再访问错误地址，再正确地址，会发现图示断路器都是慢慢放开的。



如何看?

- 7色



- 1圈

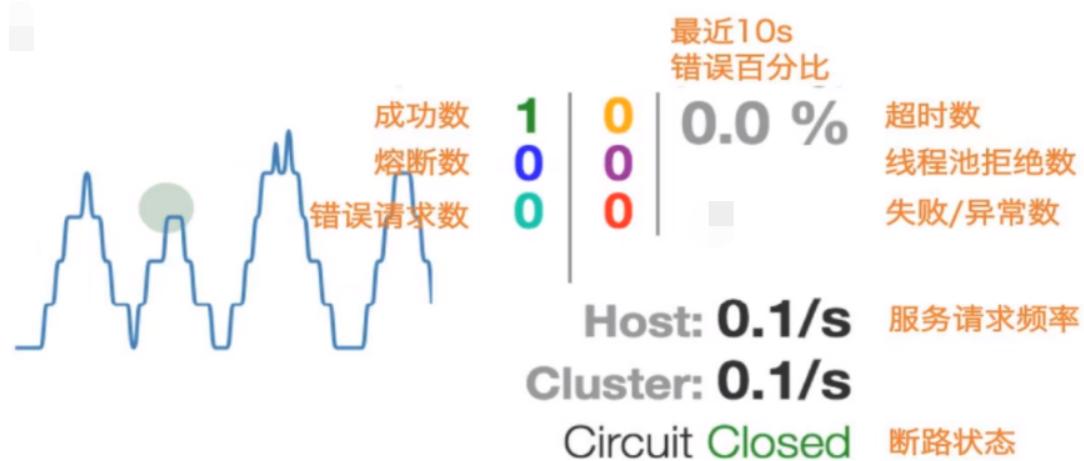
实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康度从绿色<黄色<橙色<红色递减。

该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，**流量越大该实心圆就越大**。所以通过该实心圆的展示，就可以在大量的实例中**快速的发现故障实例和高压力实例**。

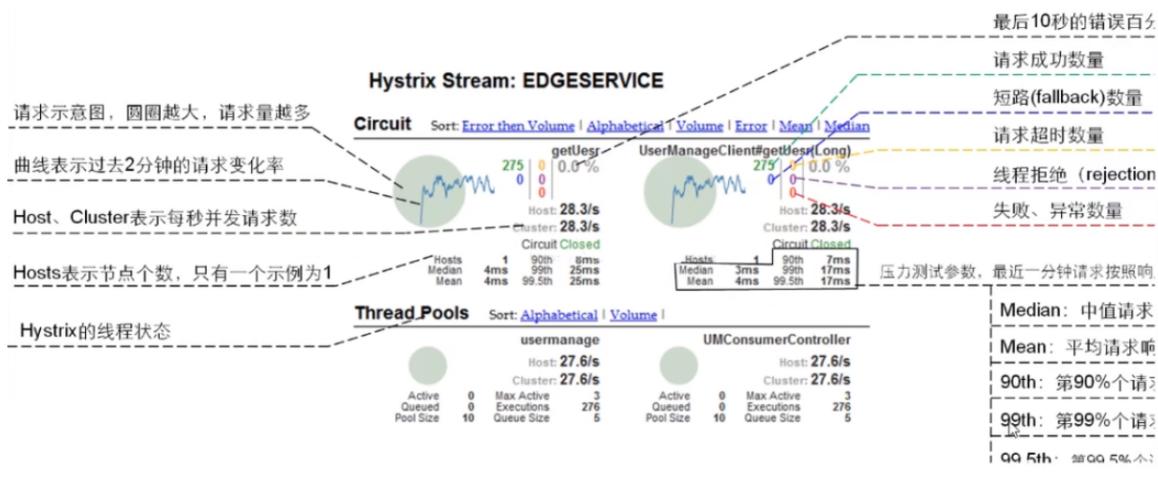
- 1线

曲线：用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

- 整图说明



- 整图说明2



服务网关

GateWay

核心简介

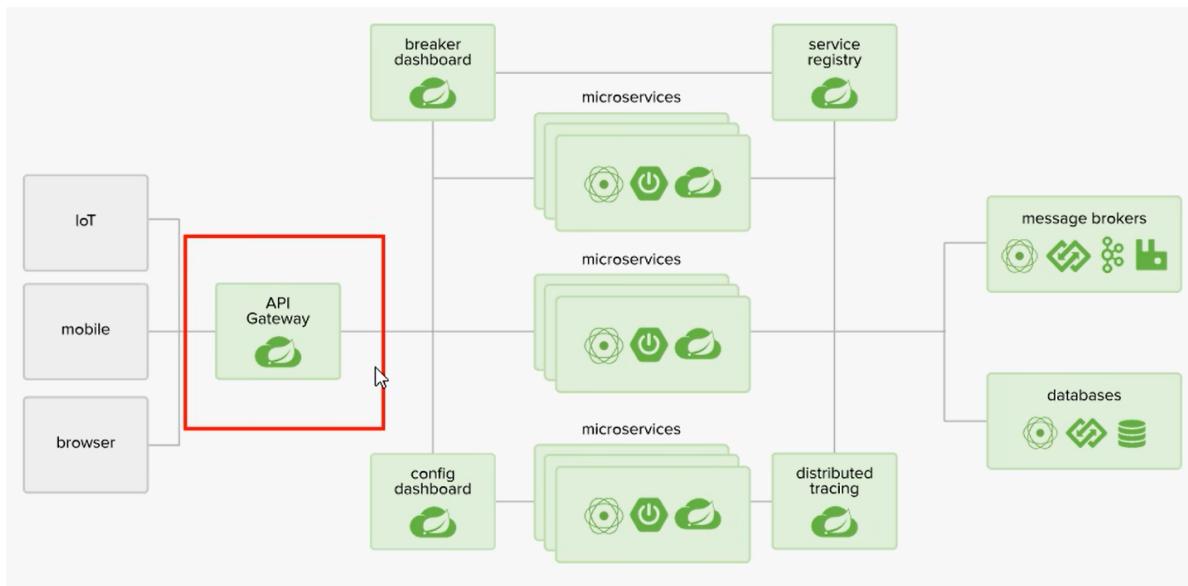
[上一代zuul 1.x官网](#)

[Gateway官网](#)

概述

Cloud全家桶中有个很重要的组件就是网关，在1.x版本中都是采用的Zuul网关；

但在2.x版本中，zuul的升级一直跳票，SpringCloud最后自己研发了一个网关替代Zuul，那就是SpringCloud Gateway一句话：gateway是原zuul1.x版的替代



Gateway是在Spring生态系统之上构建的API网关服务，基于Spring 5, Spring Boot 2和Project Reactor等技术。

Gateway旨在提供一种简单而有效的方式来对API进行路由，以及提供一些强大的过滤器功能，例如熔断、限流、重试等。

SpringCloud Gateway是Spring Cloud的一个全新项目，基于Spring 5.0+Spring Boot 2.0和Project Reactor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路由管理方式。

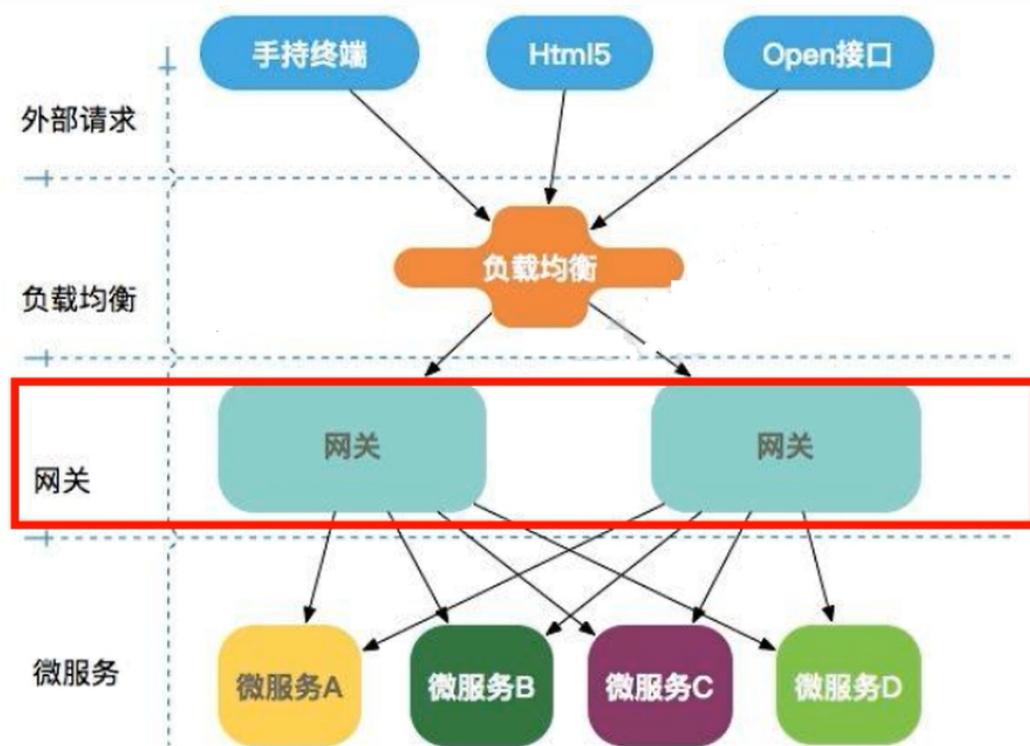
SpringCloud Gateway作为Spring Cloud 生态系统中的网关，目标是替代Zuul，在Spring Cloud 2.0以上版本中，没有对新版本的Zuul 2.0以上最新高性能版本进行集成，仍然还是使用的Zuul 1.x非Reactor模式的老版本。而为了提升网关的性能，SpringCloud Gateway是基于WebFlux框架实现的，而WebFlux框架底层则使用了高性能的Reactor模式通信框架Netty。

Spring Cloud Gateway的目标提供统一的路由方式且基于 Filter链的方式提供了网关基本的功能，例如：安全，监控/指标，和限流。

作用

- 方向代理
- 鉴权
- 流量控制
- 熔断
- 日志监控
- ...

微服务架构中网关的位置



有Zuul 1了怎么又出来Gateway?

我们为什么选择Gateway?

1.netflix不太靠谱， zuul2.0一直跳票，迟迟不发布。

- 一方面因为Zuul1.0已经进入了维护阶段，而且Gateway是SpringCloud团队研发的，是亲儿子产品，值得信赖。而且很多功能Zuul都没有用起来也非常的简单便捷。
- Gateway是基于异步非阻塞模型上进行开发的，性能方面不需要担心。虽然Netflix早就发布了最新的Zuul 2.x，但Spring Cloud貌似没有整合计划。而且Netflix相关组件都宣布进入维护期；不知前景如何？
- 多方面综合考虑Gateway是很理想的网关选择。

2.SpringCloud Gateway具有如下特性

- 基于Spring Framework 5, Project Reactor和Spring Boot 2.0进行构建；
- 动态路由：能够匹配任何请求属性；

- 可以对路由指定Predicate (断言)和Filter(过滤器);
- 集成Hystrix的断路器功能;
- 集成Spring Cloud 服务发现功能;
- 易于编写的Predicate (断言)和Filter (过滤器);
- 请求限流功能;
- 支持路径重写。

3.SpringCloud Gateway与Zuul的区别

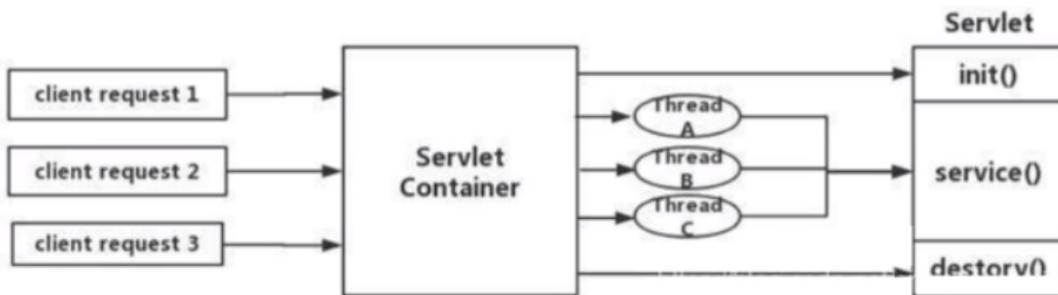
- 在SpringCloud Finchley正式版之前，Spring Cloud推荐的网关是Netflix提供的Zuul。
- Zuul 1.x，是一个基于阻塞I/O的API Gateway。
- Zuul 1.x基于Servlet 2.5使用阻塞架构它不支持任何长连接(如WebSocket)Zuul的设计模式和Nginx较像，每次I/O操作都是从工作线程中选择一个执行，请求线程被阻塞到工作线程完成，但是差别是Nginx用C++实现，Zuul用Java实现，而JVM本身会有第-次加载较慢的情况，使得Zuul的性能相对较差。
- Zuul 2.x理念更先进，想基于Netty非阻塞和支持长连接，但SpringCloud目前还没有整合。Zuul .x的性能较Zuul 1.x有较大提升。在性能方面，根据官方提供的基准测试，Spring Cloud Gateway的RPS(每秒请求数)是Zuul的1.6倍。
- Spring Cloud Gateway建立在Spring Framework 5、Project Reactor和Spring Boot2之上，使用非阻塞API。
- Spring Cloud Gateway还支持WebSocket，并且与Spring紧密集成拥有更好的开发体验

Zuul1.x模型

Springcloud中所集成的Zuul版本，采用的是Tomcat容器，使用的是传统的Servlet IO处理模型。

Servlet的生命周期？servlet由servlet container进行生命周期管理。

- container启动时构造servlet对象并调用servlet init()进行初始化；
- container运行时接受请求，并为每个请求分配一个线程（一般从线程池中获取空闲线程）然后调用service();
- container关闭时调用servlet destroy()销毁servlet。



上述模式的缺点：

Servlet是一个简单的网络IO模型，当请求进入Servlet container时，Servlet container就会为其绑定一个线程，在并发不高的场景下这种模型是适用的。但是一旦高并发(如抽风用Jmeter压)，线程数量就会上涨，而线程资源代价是昂贵的（上线文切换，内存消耗大）严重影响请求的处理时间。在一些简单业务场景下，不希望为每个request分配一个线程，只需要1个或几个线程就能应对极大并发的请求，这种业务场景下servlet模型没有优势。

所以Zuul 1.X是基于servlet之上一个阻塞式处理模型，即Spring实现了处理所有request请求的一个servlet (DispatcherServlet)并由该servlet阻塞式处理处理。所以SpringCloud Zuul无法摆脱servlet模型的弊端。

Gateway模型

WebFlux是什么? [官方文档](#)

传统的Web框架,比如说: Struts2, SpringMVC等都是基于Servlet API与Servlet容器基础之上运行的。

但是在[Servlet3.1之后有了异步非阻塞的支持](#)。而WebFlux是一个典型非阻塞异步的框架,它的核心是基于Reactor的相关API实现的。相对于传统的web框架来说,它可以运行在诸如Netty, Undertow及支持Servlet3.1的容器上。非阻塞式+函数式编程(Spring 5必须让你使用Java 8)。

Spring WebFlux是Spring 5.0 引入的新的响应式框架,区别于Spring MVC, 它不需要依赖Servlet API, 它是完全异步非阻塞的, 并且基于Reactor来实现响应式流规范。

Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work in a traditional Servlet Container or when built as a WAR.[link](#)

三大核心概念

Route(路由)

路由是构建网关的基本模块,它由ID,目标URI,一系列的断言和过滤器组成,如断言为true则匹配该路由;

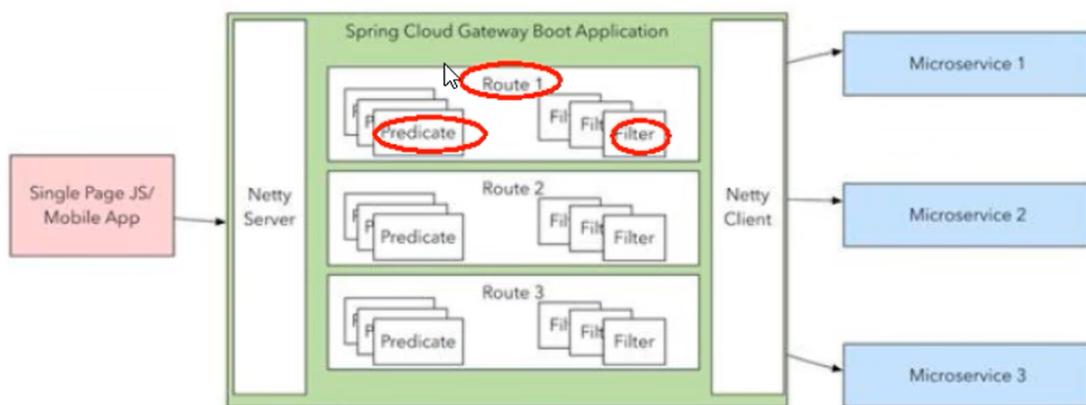
Predicate(断言)

参考的是Java8的java.util.function.Predicate, 开发人员可以匹配HTTP请求中的所有内容(例如请求头或请求参数),如果请求与断言相匹配则进行路由;

Filter(过滤)

指的是Spring框架中GatewayFilter的实例,使用过滤器,可以在请求被路由前或者之后对请求进行修改。

总体

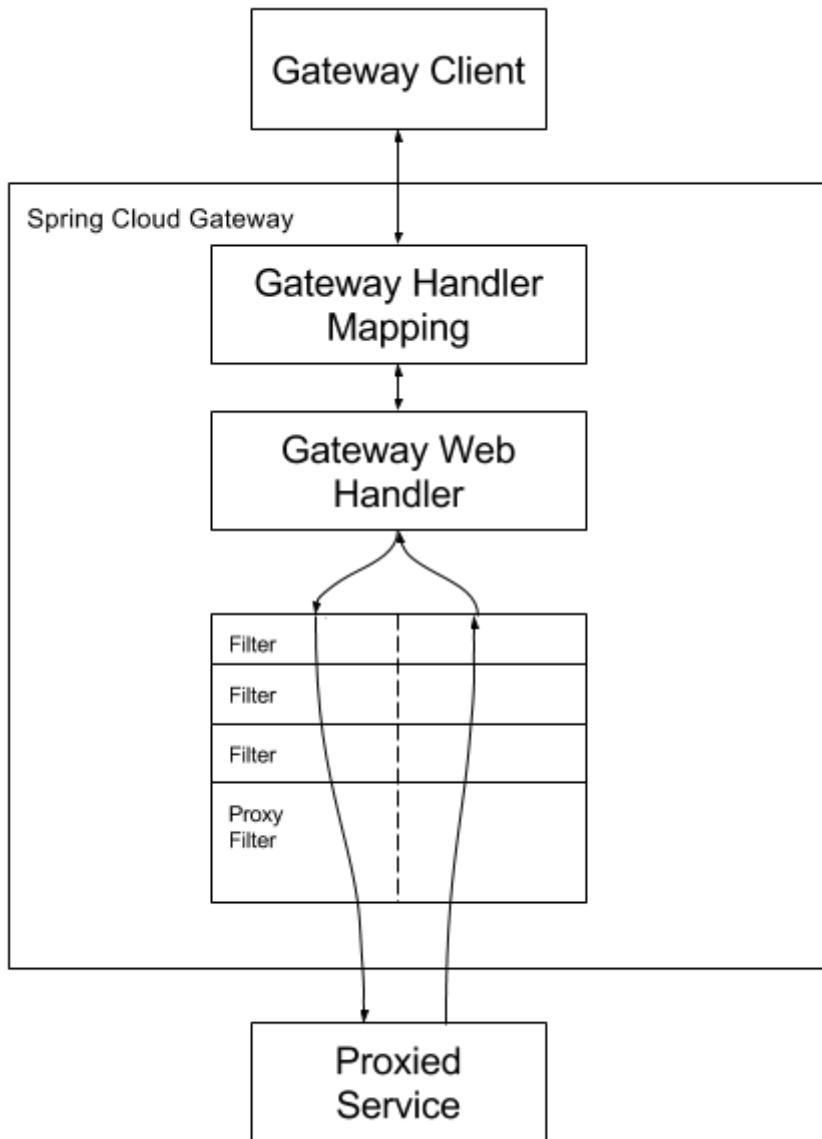


web请求,通过一些匹配条件,定位到真正的服务节点。并在这个转发过程的前后,进行一些精细化控制。

predicate就是我们的匹配条件;而filter,就可以理解为一个无所不能的拦截器。有了这两个元素,再加上目标uri,就可以实现一个具体的路由了

Gateway工作流程

[官网总结](#)



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This handler runs the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent. All "pre" filter logic is executed. Then the proxy request is made. After the proxy request is made, the "post" filter logic is run. [link](#)

客户端向Spring Cloud Gateway发出请求。然后在Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到GatewayWeb Handler。

Handler再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。

过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前("pre")或之后("post") 执行业务逻辑。

Filter在“pre”类型的过滤器可以做参数校验、权限校验、流量监控、日志输出、协议转换等，在“post”类型的过滤器中可以做响应内容、响应头的修改，日志的输出，流量监控等有着非常重要的作用。

核心逻辑：路由转发 + 执行过滤器链。

入门配置

1.新建模块

名字: cloud-gateway-gateway9527

2.POM

网关这不需要引入web和actuator

```
1 <dependencies>
2     <!--gateway-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-gateway</artifactId>
6     </dependency>
7     <!--eureka-client-->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
11    </dependency>
12    <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
13    <dependency>
14        <groupId>org.example</groupId>
15        <artifactId>cloud-api-commons</artifactId>
16        <version>${project.version}</version>
17    </dependency>
18    <!--一般基础配置类-->
19    <dependency>
20        <groupId>org.springframework.boot</groupId>
21        <artifactId>spring-boot-devtools</artifactId>
22        <scope>runtime</scope>
23        <optional>true</optional>
24    </dependency>
25    <dependency>
26        <groupId>org.projectlombok</groupId>
27        <artifactId>lombok</artifactId>
28        <optional>true</optional>
29    </dependency>
30    <dependency>
31        <groupId>org.springframework.boot</groupId>
32        <artifactId>spring-boot-starter-test</artifactId>
33        <scope>test</scope>
34    </dependency>
35 </dependencies>
```

3.YML

```
1 server:
2   port: 9527
3
4 spring:
5   application:
6     name: cloud-gateway
7
```

```

8 eureka:
9   instance:
10    hostname: cloud-gateway-service
11    client: #服务提供者provider注册进eureka服务列表内
12    service-url:
13      register-with-eureka: true
14      fetch-registry: true
15      defaultZone: http://eureka7001.com:7001/eureka

```

4.主启动类

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
4
5 @SpringBootApplication
6 @EnableEurekaClient
7 public class GatewayMain9527{
8     public static void main(String[] args) {
9         SpringApplication.run(GatewayMain9527.class, args);
10    }
11 }

```

5.业务类

无

6.9527网关如何做路由映射?

cloud-provider-payment8001看看controller的访问地址

- get
- lb

我们目前不想暴露8001端口，希望在8001外面套一层9527

7.YML新增网关配置

```

1 server:
2   port: 9527
3
4 spring:
5   application:
6     name: cloud-gateway
7 #####新增网关配置#####
8 cloud:
9   gateway:
10  routes:
11    - id: payment_routh #payment_route      #路由的ID, 没有固定规则但要求唯一,
建议配合服务名
12      uri: http://localhost:8001          #匹配后提供服务的路由地址
13      #uri: lb://cloud-payment-service #匹配后提供服务的路由地址
14      predicates:
15        - Path=/payment/get/**       # 断言, 路径相匹配的进行路由
16
17        - id: payment_routh2 #payment_route      #路由的ID, 没有固定规则但要求唯
一, 建议配合服务名

```

```

18     uri: http://localhost:8001          #匹配后提供服务的路由地址
19     #uri: lb://cloud-payment-service #匹配后提供服务的路由地址
20     predicates:
21       - Path=/payment/lb/**           # 断言, 路径相匹配的进行路由
22 #####
23
24 eureka:
25   instance:
26     hostname: cloud-gateway-service
27     client: #服务提供者provider注册进eureka服务列表内
28     service-url:
29       register-with-eureka: true
30       fetch-registry: true
31       defaultZone: http://eureka7001.com:7001/eureka

```

8. 测试

启动7001

启动8001-cloud-provider-payment8001

启动9527网关

访问说明

- 添加网关前 - <http://localhost:8001/payment/get/31>
- 添加网关后 - <http://localhost:9527/payment/get/31>
- 两者访问成功, 返回相同结果

9. YML配置说明

Gateway网关路由有两种配置方式

在配置文件yml中配置, 见上一章节

代码中注入RouteLocator的Bean

官方案例 - [link](#)

```

1 RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
2   .maxTrustedIndex(1);
3
4 ...
5
6 .route("direct-route",
7   r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
8     .uri("https://downstream1")
9 .route("proxied-route",
10   r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
11     .uri("https://downstream2")
12 )
13

```

百度国内新闻网址, 需要外网 - <http://news.baidu.com/guonei>

自己写一个

业务需求 - 通过9527网关访问到外网的百度新闻网址

编码

cloud-gateway-gateway9527 业务实现

```
1 import org.springframework.cloud.gateway.route.RouteLocator;
2 import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class GatewayConfig{
8     @Bean
9     public RouteLocator customRouteLocator(RouteLocatorBuilder
routeLocatorBuilder){
10         RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
11
12         routes.route("path_route_atguigu",
13             r -> r.path("/guonei")
14             .uri("http://news.baidu.com/guonei")).build();
15
16         return routes.build();
17     }
18 }
```

测试

浏览器输入<http://localhost:9527/guonei>, 返回<http://news.baidu.com/guonei>相同的页面。

通过微服务名实现动态路由

默认情况下Gateway会根据注册中心注册的服务列表，以注册中心上微服务名为路径创建**动态路由**进行转发，从而实现**动态路由**的功能（不写死一个地址）。

启动

- eureka7001
- payment8001/8002

POM

```
1 <!--eureka-client-->
2 <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5 </dependency>
```

YML

需要注意的是uri的协议为lb，表示启用Gateway的负载均衡功能。

lb://serviceName是spring cloud gateway在微服务中自动为我们创建的负载均衡uri。

```
1 server:
2   port: 9527
3
4 spring:
5   application:
6     name: cloud-gateway
7 #####新增网关配置#####
```

```

8  cloud:
9    gateway:
10   discovery:
11     locator:
12       enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由
13     routes:
14       - id: payment_routh #payment_route      #路由的ID，没有固定规则但要求唯一,
建议配合服务名
15         #uri: http://localhost:8001          #匹配后提供服务的路由地址
16         uri: lb://cloud-payment-service #匹配后提供服务的路由地址
17         predicates:
18           - Path=/payment/get/**        # 断言，路径相匹配的进行路由
19
20       - id: payment_routh2 #payment_route     #路由的ID，没有固定规则但要求唯
一，建议配合服务名
21         #uri: http://localhost:8001          #匹配后提供服务的路由地址
22         uri: lb://cloud-payment-service #匹配后提供服务的路由地址
23         predicates:
24           - Path=/payment/lb/**        # 断言，路径相匹配的进行路由
25 #####
26
27 eureka:
28   instance:
29     hostname: cloud-gateway-service
30     client: #服务提供者provider注册进eureka服务列表内
31     service-url:
32       register-with-eureka: true
33       fetch-registry: true
34     defaultZone: http://eureka7001.com:7001/eureka

```

测试

浏览器输入 - <http://localhost:9527/payment/lb>

结果

不停刷新页面，8001/8002两个端口切换。

Predicate的使用

[官方文档](#)

Route Predicate Factories这个是什么

Spring Cloud Gateway matches routes as part of the Spring WebFlux HandlerMapping infrastructure. Spring Cloud Gateway includes many built-in route predicate factories. All of these predicates match on different attributes of the HTTP request. You can combine multiple route predicate factories with logical and statements. [link](#)

Spring Cloud Gateway将路由匹配作为Spring WebFlux HandlerMapping基础架构的一部分。

Spring Cloud Gateway包括许多内置的Route Predicate工厂。所有这些Predicate都与HTTP请求的不同属性匹配。多个RoutePredicate工厂可以进行组合。

Spring Cloud Gateway创建Route 对象时，使用RoutePredicateFactory 创建 Predicate对象，Predicate 对象可以赋值给Route。Spring Cloud Gateway包含许多内置的Route Predicate Factories，所有这些谓词都匹配HTTP请求的不同属性。多种谓词工厂可以组合，并通过逻辑and。

[predicate](#)

美: ['predɪkeɪt] 英: ['predɪkət]

v. 断言; 使基于; 使以...为依据; 表明

adj. 述语的; 谓项的

n. 谓语 (句子成分, 对主语加以陈述, 如 John went home 中的 went home)

常用的Route Predicate Factory

1. The After Route Predicate Factory
2. The Before Route Predicate Factory
3. The Between Route Predicate Factory
4. The Cookie Route Predicate Factory
5. The Header Route Predicate Factory
6. The Host Route Predicate Factory
7. The Method Route Predicate Factory
8. The Path Route Predicate Factory
9. The Query Route Predicate Factory
10. The RemoteAddr Route Predicate Factory
11. The weight Route Predicate Factory

```
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [After]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Before]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Between]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Cookie]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Header]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Host]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Method]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Path]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Query]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [RequestBodyPredicateFactory]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [RemoteAddr]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Weight]
2022-07-19 11:03:02.879 INFO 18308 --- [ restartedMain] o.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [CloudFoundryRouteService]
```

讨论几个Route Predicate Factory

The After Route Predicate Factory

```
1  spring:
2    application:
3      name: cloud-gateway
4      #####新增网关配置#####
5    cloud:
6      gateway:
7        discovery:
8          locator:
9            enabled: true #开启从注册中心动态创建路由的功能, 利用微服务名进行路由
10         routes:
11           - id: payment_routh2 #payment_route      #路由的ID, 没有固定规则但要求唯一, 建议配合服务名
12             uri: http://localhost:8001           #匹配后提供服务的路由地址
13             uri: lb://cloud-payment-service #匹配后提供服务的路由地址
14           predicates:
15             - Path=/payment/lb/**          # 断言, 路径相匹配的进行路由
16             - After=2022-07-19T12:12:00.736+08:00[Asia/Shanghai]
```

可以通过下述方法获得上述格式的时间戳字符串

```

1 import java.time.ZonedDateTime;
2
3 public class T2{
4     public static void main(String[] args)
5     {
6         ZonedDateTime zbj = ZonedDateTime.now(); // 默认时区
7         System.out.println(zbj);
8
9         //2022-07-19T11:12:00.736+08:00[Asia/Shanghai]
10    }
11 }

```

The Between Route Predicate Factory

```

1 spring:
2   application:
3     name: cloud-gateway
4 #####新增网关配置#####
5 cloud:
6   gateway:
7     discovery:
8       locator:
9         enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由
10      routes:
11        - id: payment_routh2 #payment_route      #路由的ID，没有固定规则但要求唯一，建议配合服务名
12          uri: http://localhost:8001           #匹配后提供服务的路由地址
13          uri: lb://cloud-payment-service #匹配后提供服务的路由地址
14          predicates:
15            - Path=/payment/lb/**           # 断言，路径相匹配的进行路由
16            - Between=2022-07-19T12:12:00.736+08:00[Asia/Shanghai],2022-07-20T12:12:00.736+08:00[Asia/Shanghai]

```

The Cookie Route Predicate Factory

```

1 spring:
2   application:
3     name: cloud-gateway
4 #####新增网关配置#####
5 cloud:
6   gateway:
7     discovery:
8       locator:
9         enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由
10      routes:
11        - id: payment_routh2 #payment_route      #路由的ID，没有固定规则但要求唯一，建议配合服务名
12          uri: http://localhost:8001           #匹配后提供服务的路由地址
13          uri: lb://cloud-payment-service #匹配后提供服务的路由地址
14          predicates:
15            - Path=/payment/lb/**           # 断言，路径相匹配的进行路由
16            - Cookie=username,jay

```

The cookie route predicate factory takes two parameters, the cookie name and a regular expression.

This predicate matches cookies that have the given name and whose values match the regular expression.

测试：

```
1 # 该命令相当于发get请求，且没带cookie
2 curl http://localhost:9527/payment/lb
3
4 # 带cookie的
5 curl http://localhost:9527/payment/lb --cookie "username=jay"
```

```
C:\Users\Serendipity>curl http://localhost:9527/payment/lb
{"timestamp": "2022-07-19T03:22:15.421+0000", "path": "/payment/lb", "status": 404, "error": "Not Found", "message": null, "requestId": "878ef09c", "trace": "org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND\r\n\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\t\tSuppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: \nError has been observed at the following site(s):\n\t|_ checkpoint □ org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]\n\t|_ checkpoint □ HTTP GET \"/payment/lb\" [ExceptionHandlingWebHandler]\n\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\t\tat reactor.core.publisher.MonoDefer.subscribe(MonoDefer.java:4
C:\Users\Serendipity>curl http://localhost:9527/payment/lb --cookie "username=jay"
8002
C:\Users\Serendipity>curl http://localhost:9527/payment/lb --cookie "username=jay"
8001
C:\Users\Serendipity>curl http://localhost:9527/payment/lb --cookie "username=jay"
8002
C:\Users\Serendipity>curl http://localhost:9527/payment/lb --cookie "username=jay"
8001
```

The Header Route Predicate Factory

```
1 spring:
2   application:
3     name: cloud-gateway
4 #####新增网关配置#####
5 cloud:
6   gateway:
7     discovery:
8       locator:
9         enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由
10      routes:
11        - id: payment_routh2 #payment_route      #路由的ID，没有固定规则但要求唯一，建议配合服务名
12          #uri: http://localhost:8001           #匹配后提供服务的路由地址
13          uri: lb://cloud-payment-service #匹配后提供服务的路由地址
14          predicates:
15            - Path=/payment/lb/**          # 断言，路径相匹配的进行路由
16            - Header=X-Request-Id,\d+      #请求头要有X-Request-Id 属性并且值为整数的正则表达式
```

The header route predicate factory takes two parameters, the header name and a regular expression.

This predicate matches with a header that has the given name whose value matches the regular expression.

测试

```
1 # 带指定请求头的参数的CURL命令
2 curl http://localhost:9527/payment/lb -H "X-Request-Id:123"
```

其它的，举一反三。

小结

说白了，Predicate就是为了实现一组匹配规则，让请求过来找到对应的Route进行处理。

Filter的使用

官方文档

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

路由过滤器可用于修改进入的HTTP请求和返回的HTTP响应，路由过滤器只能指定路由进行使用。Spring Cloud Gateway内置了多种路由过滤器，他们都由GatewayFilter的工厂类来产生。

Spring Cloud Gateway的Filter:

生命周期:

- pre
- post

种类 (具体看官方文档) :

- GatewayFilter - 有31种
- GlobalFilter - 有10种

常用的GatewayFilter: AddRequestParameter GatewayFilter

自定义全局GlobalFilter:

两个主要接口介绍:

1. GlobalFilter
2. Ordered

能干什么:

1. 全局日志记录
2. 统一网关鉴权
3. ...

代码案例:

GateWay9527项目添加MyLogGateWayFilter类:

```
1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.cloud.gateway.filter.GatewayFilterChain;
3 import org.springframework.cloud.gateway.filter.GlobalFilter;
4 import org.springframework.core.Ordered;
5 import org.springframework.http.HttpStatus;
6 import org.springframework.stereotype.Component;
7 import org.springframework.web.server.ServerWebExchange;
8 import reactor.core.publisher.Mono;
9
10 import java.util.Date;
11
12 @Slf4j
13 @Component
14 public class MyLogGatewayFilter implements GlobalFilter, Ordered {
15     @Override
```

```

16     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
17         chain) {
18             log.info("*****come in MyLogGatewayFilter: "+new Date());
19
20             String uname =
21                 exchange.getRequest().getQueryParams().getFirst("uname");
22
23             if(uname == null) {
24                 log.info("*****用户名为null, 非法用户, o(╥﹏╥)o");
25                 exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
26                 return exchange.getResponse().setComplete();
27             }
28
29
30             return chain.filter(exchange);
31
32         @Override
33         public int getOrder() {
34             return 0;
35         }
36     }

```

测试：

启动：

- EurekaMain7001
- PaymentMain8001
- GateWayMain9527
- PaymentMain8002

浏览器输入：

- <http://localhost:9527/payment/lb> - 反问异常
- <http://localhost:9527/payment/lb?uname=abc> - 正常反问

服务配置

Spring Cloud Config分布式配置中心

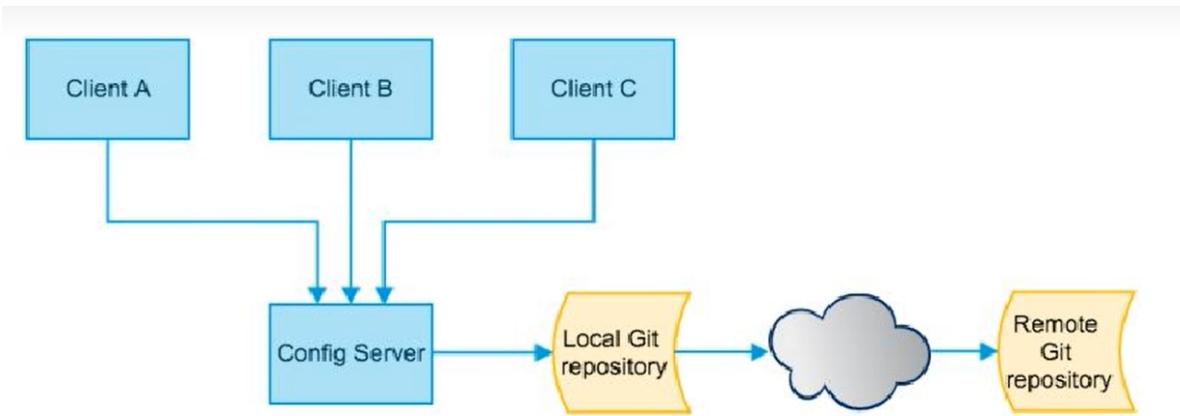
概述

分布式系统面临的配置问题

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务。由于每个服务都需要必要的配置信息才能运行，所以一套集中式的、动态的配置管理设施是必不可少的。

SpringCloud提供了ConfigServer来解决这个问题，我们每一个微服务自己带着一个application.yml，上百个配置文件的管理.....

是什么



SpringCloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为**各个不同微服务应用**的所有环境提供了一个**中心化的外部配置**。

怎么玩

SpringCloud Config分为**服务端**和**客户端**两部分。

- 服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。
- 客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

能干嘛

- 集中管理配置文件
- 不同环境不同配置，动态化的配置更新，分环境部署比如dev/test/prod/beta/release
- 运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息
- 当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置
- 将配置信息以REST接口的形式暴露 - post/crud访问刷新即可...

与GitHub整合配置

由于SpringCloud Config默认使用Git来存储配置文件(也有其它方式,比如支持SVN和本地文件)，但最推荐的还是Git，而且使用的是http/https访问的形式。

官网

<https://cloud.spring.io/spring-cloud-static/spring-cloud-config/2.2.1.RELEASE/reference/html/>

config服务端配置与测试

用你自己的账号在GitHub上新建一个名为springcloud-config的新Repository。

由上一步获得刚新建的git地址 - git@github.com:chengdashia/springcloud-config.git。

本地硬盘目录上新建git仓库并clone。

- 工作目录为E:\gitProject\github
- git clone git@github.com:chengdashia/springcloud-config.git

此时在工作目录会创建名为springcloud-config的文件夹。

在springcloud-config的文件夹中创建三个配置文件（为本次教学使用的），随后 `git add .`, `git commit -m "sth"` 等一系列上传操作上传到springcloud-config的新Repository。

- config-dev.yml

```

1 | config:
2 |   info: "master branch,springcloud-config/config-dev.yml version=7"

```

- config-prod.yml

```

1 | config:
2 |   info: "master branch,springcloud-config/config-prod.yml version=1"

```

- config-test.yml

```

1 | config:
2 |   info: "master branch,springcloud-config/config-test.yml version=1"

```

新建Module模块

名字: cloud-config-center-3344

它即为Cloud的配置中心模块CloudConfig Center

POM

```

1 <dependencies>
2   <!--添加消息总线RabbitMQ支持-->
3   <dependency>
4     <groupId>org.springframework.cloud</groupId>
5     <artifactId>spring-cloud-starter-bus-amqp</artifactId>
6   </dependency>
7   <dependency>
8     <groupId>org.springframework.cloud</groupId>
9     <artifactId>spring-cloud-config-server</artifactId>
10  </dependency>
11  <dependency>
12    <groupId>org.springframework.cloud</groupId>
13    <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
14  </dependency>
15  <dependency>
16    <groupId>org.springframework.boot</groupId>
17    <artifactId>spring-boot-starter-web</artifactId>
18  </dependency>
19
20  <dependency>
21    <groupId>org.springframework.boot</groupId>
22    <artifactId>spring-boot-starter-actuator</artifactId>
23  </dependency>
24  <dependency>
25    <groupId>org.springframework.boot</groupId>
26    <artifactId>spring-boot-devtools</artifactId>
27    <scope>runtime</scope>
28    <optional>true</optional>
29  </dependency>
30  <dependency>
31    <groupId>org.projectlombok</groupId>
32    <artifactId>lombok</artifactId>

```

```

33         <optional>true</optional>
34     </dependency>
35     <dependency>
36         <groupId>org.springframework.boot</groupId>
37         <artifactId>spring-boot-starter-test</artifactId>
38         <scope>test</scope>
39     </dependency>
40 </dependencies>

```

YML

```

1 server:
2   port: 3344
3
4 spring:
5   application:
6     name: cloud-config-center #注册进Eureka服务器的微服务名
7   cloud:
8     config:
9       server:
10      git:
11        #uri: git@github.com:chengdashia/springcloud-config.git #GitHub上面
12        #的git仓库名字 使用ssh可能有问题
13        url: https://github.com/chengdashia/springcloud-config.git
14        #####搜索目录
15        search-paths:
16          - springcloud-config
17        #####读取分支
18        label: master
19
20 #服务注册到eureka地址
21 eureka:
22   client:
23     service-url:
24       defaultZone: http://localhost:7001/eureka

```

主启动类

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.config.server.EnableConfigServer;
4
5 @SpringBootApplication
6 @EnableConfigServer
7 public class ConfigCenterMain3344{
8     public static void main(String[] args) {
9         SpringApplication.run(ConfigCenterMain3344.class, args);
10    }
11 }

```

windows下修改hosts文件，增加映射

```
1 | 127.0.0.1 config-3344.com
```

测试通过Config微服务是否可以从GitHub上获取配置内容

- 启动ConfigCenterMain3344
- 浏览器访问 - <http://config-3344.com:3344/master/config-dev.yml>
- 页面返回结果：

配置读取规则

- [官方文档](#)
- /{label}/{application}-{profile}.yml (推荐)
 - master分支
 - <http://config-3344.com:3344/master/config-dev.yml>
 - <http://config-3344.com:3344/master/config-test.yml>
 - <http://config-3344.com:3344/master/config-prod.yml>
 - dev 分支
 - <http://config-3344.com:3344/dev/config-dev.yml>
 - <http://config-3344.com:3344/dev/config-test.yml>
 - <http://config-3344.com:3344/dev/config-prod.yml>
- <http://config-3344.com:3344/dev/config-dev.yml>
- <http://config-3344.com:3344/dev/config-test.yml>
- <http://config-3344.com:3344/dev/config-prod.yml>
- /{application}-{profile}.yml
 - <http://config-3344.com:3344/config-dev.yml>
 - <http://config-3344.com:3344/config-test.yml>
 - <http://config-3344.com:3344/config-prod.yml>
 - <http://config-3344.com:3344/config-xxxx.yml>(不存在的配置)
- /{application}/{profile}[/{label}]
 - <http://config-3344.com:3344/config/dev/master>
 - <http://config-3344.com:3344/config/test/master>
 - <http://config-3344.com:3344/config/test/dev>
- 重要配置细节总结
 - /{name}-{profiles}.yml
 - /{label}-{name}-{profiles}.yml
 - label: 分支(branch)
 - name: 服务名
 - profiles: 环境(dev/test/prod)

成功实现了用SpringCloud Config通过GitHub获取配置信息

config客户端配置与测试

新建module

名字：cloud-config-client-3355

POM

```

1 <dependencies>
2   <!--添加消息总线RabbitMQ支持-->
3   <dependency>
4     <groupId>org.springframework.cloud</groupId>
5     <artifactId>spring-cloud-starter-bus-amqp</artifactId>
6   </dependency>

```

```

7   <dependency>
8     <groupId>org.springframework.cloud</groupId>
9       <artifactId>spring-cloud-starter-config</artifactId>
10      </dependency>
11      <dependency>
12        <groupId>org.springframework.cloud</groupId>
13          <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
14        </dependency>
15        <dependency>
16          <groupId>org.springframework.boot</groupId>
17            <artifactId>spring-boot-starter-web</artifactId>
18          </dependency>
19          <dependency>
20            <groupId>org.springframework.boot</groupId>
21              <artifactId>spring-boot-starter-actuator</artifactId>
22            </dependency>
23
24          <dependency>
25            <groupId>org.springframework.boot</groupId>
26              <artifactId>spring-boot-devtools</artifactId>
27              <scope>runtime</scope>
28              <optional>true</optional>
29            </dependency>
30            <dependency>
31              <groupId>org.projectlombok</groupId>
32                <artifactId>lombok</artifactId>
33                <optional>true</optional>
34            </dependency>
35            <dependency>
36              <groupId>org.springframework.boot</groupId>
37                <artifactId>spring-boot-starter-test</artifactId>
38                <scope>test</scope>
39            </dependency>
40        </dependencies>

```

bootstrap.yml

- application.yml 是用户级的资源配置项
- bootstrap.yml 是系统级的，优先级更加高

Spring Cloud会创建一个Bootstrap Context，作为Spring应用的Application Context的父上下文。

初始化的时候，BootstrapContext负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取的Environment。

Bootstrap属性有高优先级，默认情况下，它们不会被本地配置覆盖。Bootstrap context和Application Context有着不同的约定，所以新增了一个bootstrap.yml文件，保证Bootstrap Context和Application Context配置的分离。

要将Client模块下的application.yml文件改为bootstrap.yml,这是很关键的，因为bootstrap.yml是比application.yml先加载的。

bootstrap.yml优先级高于application.yml。

```

1 server:
2   port: 3355
3

```

```

4 spring:
5   application:
6     name: config-client
7   cloud:
8     #Config客户端配置
9   config:
10    label: master #分支名称
11    name: config #配置文件名称
12    profile: dev #读取后缀名称 上述3个综合: master分支上config-dev.yml的配置文
件被读取http://config-3344.com:3344/master/config-dev.yml
13    uri: http://localhost:3344 #配置中心地址
14
15
16 #服务注册到eureka地址
17 eureka:
18   client:
19     service-url:
20       defaultZone: http://localhost:7001/eureka

```

修改config-dev.yml配置并提交到GitHub中，比如加个变量age或者版本号version

主启动

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
4
5 @EnableEurekaClient
6 @SpringBootApplication
7 public class ConfigClientMain3355{
8   public static void main(String[] args) {
9     SpringApplication.run(ConfigClientMain3355.class, args);
10  }
11 }

```

业务类

```

1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.cloud.context.config.annotation.RefreshScope;
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class ConfigClientController{
8   @Value("${config.info}")
9   private String configInfo;
10
11   @GetMapping("/configInfo")
12   public String getConfigInfo(){
13     return configInfo;
14   }
15 }

```

测试

启动Config配置中心3344微服务并自测

- <http://config-3344.com:3344/master/config-prod.yml>
- <http://config-3344.com:3344/master/config-dev.yml>

启动3355作为Client准备访问

- <http://localhost:3355/configInfo>

The screenshot shows two browser tabs. The top tab is titled 'localhost:3355/configInfo' and displays the configuration details: 'master branch,springcloud-config/config-dev.yml version=7'. The bottom tab is titled 'config-3344.com:3344/master/config-dev.yml' and shows the raw configuration content: 'config: info: master branch,springcloud-config/config-dev.yml version=7'.

成功实现了客户端3355访问SpringCloud Config3344通过GitHub获取配置信息

分布式配置的动态刷新问题

- Linux运维修改GitHub上的配置文件内容做调整
- 刷新3344，发现ConfigServer配置中心立刻响应
- 刷新3355，发现ConfigClient客户端没有任何响应
- 3355没有变化除非自己重启或者重新加载
- 难到每次运维修改配置文件，客户端都需要重启??噩梦

config客户端之动态刷新

避免每次更新配置都要重启客户端微服务3355

动态刷新步骤：

修改3355模块

POM引入actuator监控

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```

修改YML，添加暴露监控端口配置：

```
1 # 暴露监控端点
2 management:
3   endpoints:
4     web:
5       exposure:
6         include: "*"
```

@RefreshScope业务类Controller修改

```
1 import org.springframework.cloud.context.config.annotation.RefreshScope;
2 ...
3
4 @RestController
5 @RefreshScope//<----->
6 public class ConfigClientController{
7 ...
8 }
9
```

测试

此时修改github配置文件内容 -> 访问3344 -> 访问3355

<http://localhost:3355/configInfo>

3355改变没有??? **没有**, 还需一步

How

需要运维人员发送Post请求刷新3355

```
1 | curl -X POST "http://localhost:3355/actuator/refresh"
```

```
C:\Users\Serendipity>curl -X POST "http://localhost:3355/actuator/refresh"
[{"config.client.version", "config.info"}]
C:\Users\Serendipity>
```

再次测试

<http://localhost:3355/configInfo>

3355改变没有??? **改了**。

成功实现了客户端3355刷新到最新配置内容, 避免了服务重启

想想还有什么问题?

- 假如有多个微服务客户端3355/3366/3377
- 每个微服务都要执行一次post请求, 手动刷新?
- 可否广播, 一次通知, 处处生效?
- 我们想大范围的自动刷新, 求方法

消息总线

Spring Cloud Bus

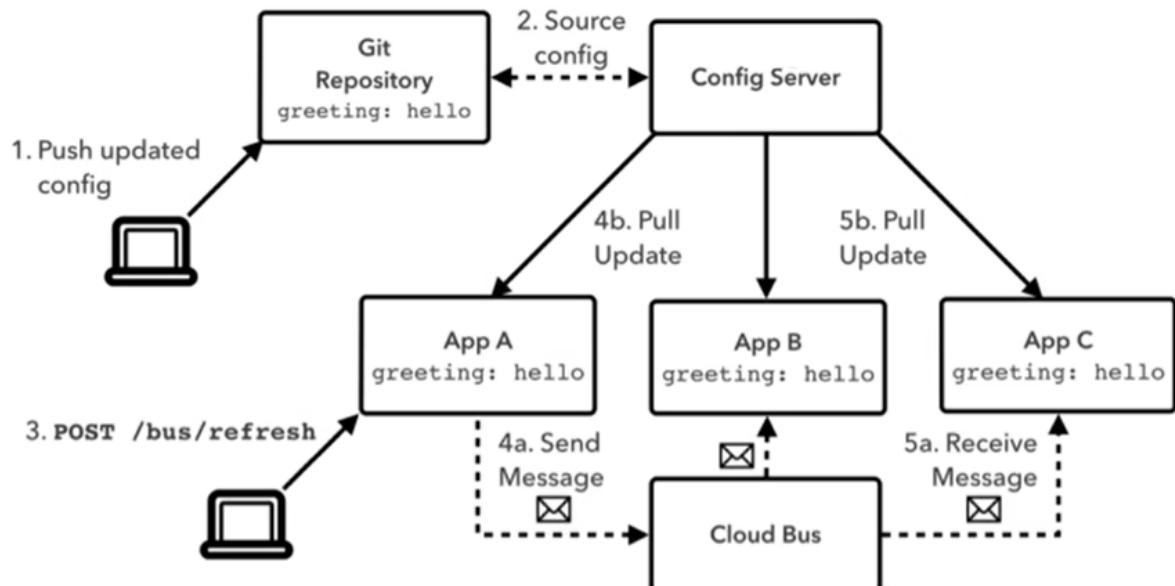
概述

一言以蔽之, 分布式自动刷新配置功能。

Spring Cloud Bus配合Spring Cloud Config使用可以实现配置的动态刷新。

是什么

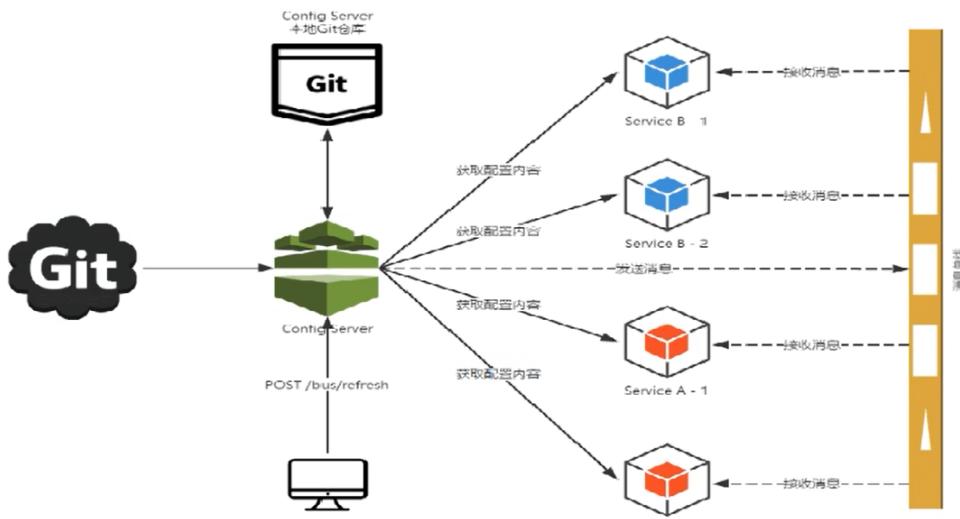
Spring Cloud Bus 配合Spring Cloud Config 使用可以实现配置的动态刷新。



Spring Cloud Bus是用来将分布式系统的节点与轻量级消息系统链接起来的框架，它整合了Java的事件处理机制和消息中间件的功能。Spring Cloud Bus目前支持RabbitMQ和Kafka。

能干嘛

Spring Cloud Bus能管理和传播分布式系统间的消息，就像一个分布式执行器，可用于广播状态更改、事件推送等，也可以当作微服务间的通信通道。



为何被称为总线

- 什么是总线

在微服务架构的系统中，通常会使用轻量级的消息代理来构建一个共用的消息主题，并让系统中所有微服务实例都连接上来。由于该主题中产生的消息会被所有实例监听和消费，所以称它为消息总线。在总线上的各个实例，都可以方便地广播一些需要让其他连接在该主题上的实例都知道的消息。

- 基本原理

ConfigClient实例都监听MQ中同一个topic(默认是Spring Cloud Bus)。当一个服务刷新数据的时候，它会把这个信息放入到Topic中，这样其它监听同一Topic的服务就能得到通知，然后去更新自身的配置。

Rabbit MQ环境配置

- 安装Erlang，下载地址：http://erlang.org/download/otp_win64_21.3.exe
- 安装RabbitMQ，下载地址：<https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.3/rabbitmq-server-3.8.3.exe>
- 打开cmd进入RabbitMQ安装目录下的sbin目录，如：D:\devSoft\RabbitMQ\Scrverk\rabbitmq_server-3.7.14\sbin
- 输入以下命令启动管理功能

```
1 | rabbitmq-plugins enable rabbitmq_management
```

这样就可以添加可视化插件。

- 访问地址查看是否安装成功：<http://localhost:15672/>
- 输入账号密码并登录：guest guest

动态刷新全局广播

设计思想和选型

必须先具备良好的RabbitMQ环境先

演示广播效果，增加复杂度，再以3355为模板再制作一个3366

1.新建module

名字：cloud-config-client-3366

2.POM

```
1 <dependencies>
2     <!--添加消息总线RabbitMQ支持-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-bus-amqp</artifactId>
6     </dependency>
7     <dependency>
8         <groupId>org.springframework.cloud</groupId>
9         <artifactId>spring-cloud-starter-config</artifactId>
10    </dependency>
11    <dependency>
12        <groupId>org.springframework.cloud</groupId>
13        <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
14    </dependency>
15    <dependency>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-web</artifactId>
18    </dependency>
19    <dependency>
20        <groupId>org.springframework.boot</groupId>
21        <artifactId>spring-boot-starter-actuator</artifactId>
22    </dependency>
23
24    <dependency>
25        <groupId>org.springframework.boot</groupId>
26        <artifactId>spring-boot-devtools</artifactId>
```

```

27         <scope>runtime</scope>
28         <optional>true</optional>
29     </dependency>
30     <dependency>
31         <groupId>org.projectlombok</groupId>
32         <artifactId>lombok</artifactId>
33         <optional>true</optional>
34     </dependency>
35     <dependency>
36         <groupId>org.springframework.boot</groupId>
37         <artifactId>spring-boot-starter-test</artifactId>
38         <scope>test</scope>
39     </dependency>
40 </dependencies>

```

3.YML

```

1 server:
2   port: 3366
3
4 spring:
5   application:
6     name: config-client
7   cloud:
8     #Config客户端配置
9     config:
10       label: master #分支名称
11       name: config #配置文件名称
12       profile: dev #读取后缀名称 上述3个综合: master分支上config-dev.yml的配置文
件被读取http://config-3344.com:3344/master/config-dev.yml
13       uri: http://localhost:3344 #配置中心地址
14
15 #rabbitmq相关配置 15672是web管理界面的端口; 5672是MQ访问的端口
16 rabbitmq:
17   host: localhost
18   port: 5672
19   username: guest
20   password: guest
21
22 #服务注册到eureka地址
23 eureka:
24   client:
25     service-url:
26       defaultZone: http://localhost:7001/eureka
27
28 # 暴露监控端点
29 management:
30   endpoints:
31     web:
32       exposure:
33         include: "*"

```

4.主启动

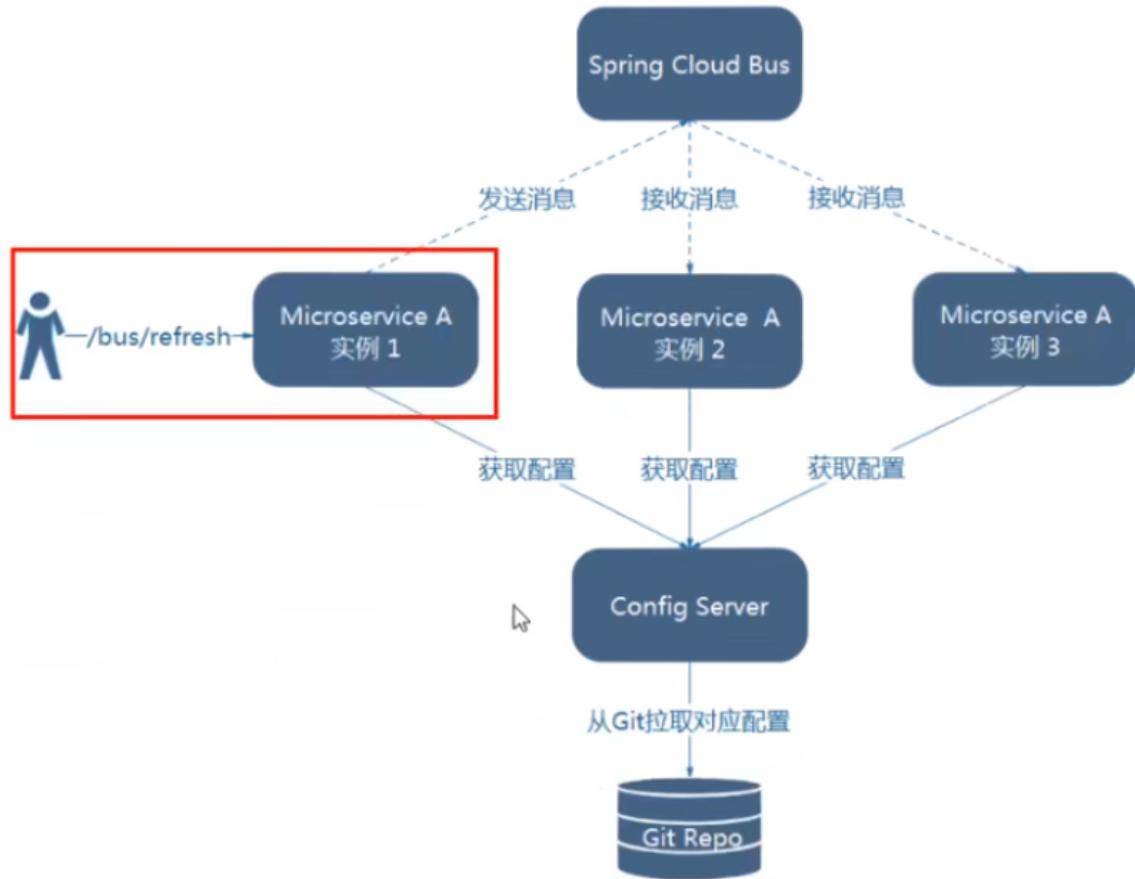
```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
4
5 @EnableEurekaClient
6 @SpringBootApplication
7 public class ConfigClientMain3366{
8     public static void main(String[] args)
9     {
10         SpringApplication.run(ConfigClientMain3366.class,args);
11     }
12 }
```

5.业务类

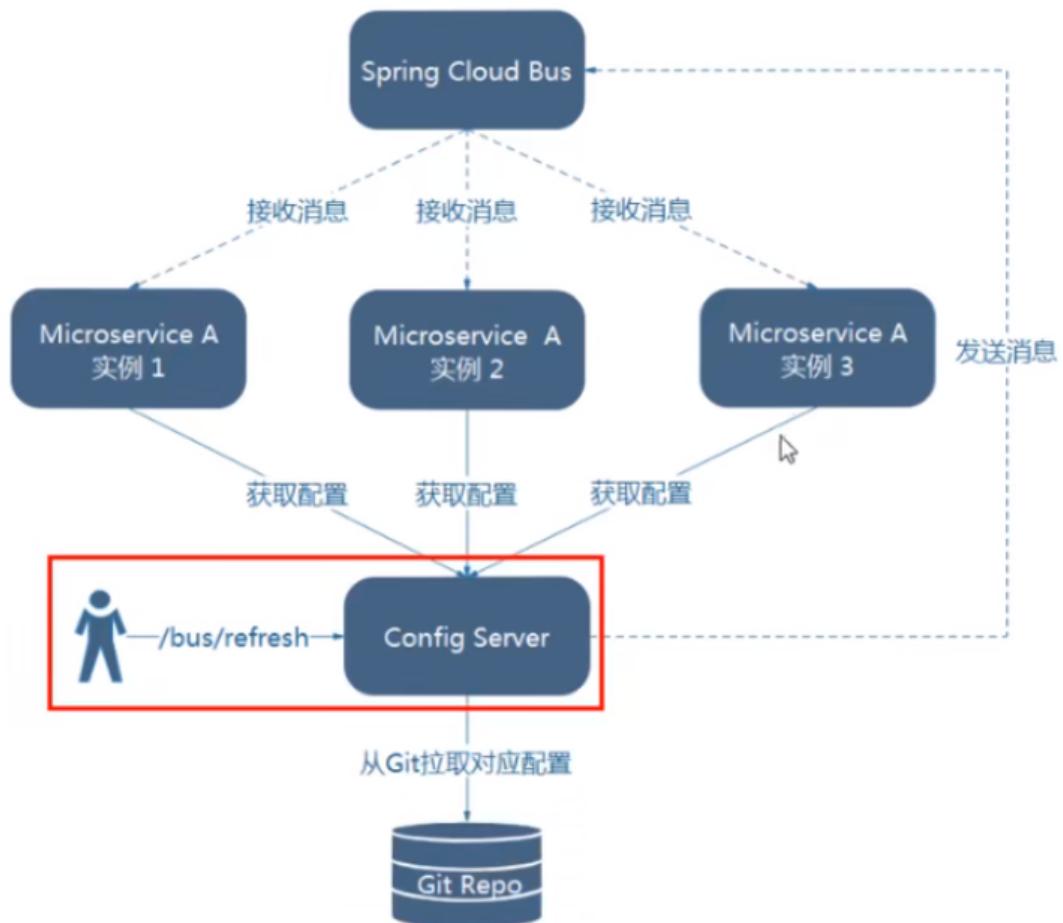
```
1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.cloud.context.config.annotation.RefreshScope;
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 @RefreshScope
8 public class ConfigClientController{
9     @Value("${server.port}")
10    private String serverPort;
11
12    @Value("${config.info}")
13    private String configInfo;
14
15    @GetMapping("/configInfo")
16    public String configInfo(){
17        return "serverPort: "+serverPort+"\t\n\n configInfo: "+configInfo;
18    }
19 }
```

设计思想

1.利用消息总线触发一个客户端/bus/refresh,而刷新所有客户端的配置



2.利用消息总线触发一个服务端ConfigServer的/bus/refresh端点，而刷新所有客户端的配置



图二的架构显然更加适合，图一不适合的原因如下：

- 打破了微服务的职责单一性，因为微服务本身是业务模块，它本不应该承担配置刷新的职责。
- 破坏了微服务各节点的对等性。
- 有一定的局限性。例如，微服务在迁移时，它的网络地址常常会发生变化，此时如果想要做到自动刷新，那就会增加更多的修改。

动态刷新全局广播配置实现

给cloud-config-center-3344配置中心服务端添加消息总线支持

```

1 <!--添加消息总线RabbitMQ支持-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-bus-amqp</artifactId>
5 </dependency>
6 <dependency>
7   <groupId>org-springframework.boot</groupId>
8   <artifactId>spring-boot-starter-actuator</artifactId>
9 </dependency>
```

yml

```

1 server:
2   port: 3344
3
4 spring:
5   application:
6     name: cloud-config-center #注册进Eureka服务器的微服务名
7   cloud:
8     config:
9       server:
10      git:
11        uri: git@github.com:zzyybs/springcloud-config.git #GitHub上面的git
仓库名字
12      #####搜索目录
13      search-paths:
14        - springcloud-config
15      #####读取分支
16      label: master
17 #rabbitmq相关配置<-----
18 rabbitmq:
19   host: localhost
20   port: 5672
21   username: guest
22   password: guest
23
24 #服务注册到eureka地址
25 eureka:
26   client:
27     service-url:
28       defaultZone: http://localhost:7001/eureka
29
30 ##rabbitmq相关配置，暴露bus刷新配置的端点<-----
31 management:
32   endpoints: #暴露bus刷新配置的端点
33   web:
34     exposure:
```

```
35 |     include: 'bus-refresh'
```

给cloud-config-client-3355客户端添加消息总线支持

POM

```
1 <!--添加消息总线RabbitMQ支持-->
2 <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-bus-amqp</artifactId>
5 </dependency>
6 <dependency>
7     <groupId>org-springframework.boot</groupId>
8     <artifactId>spring-boot-starter-actuator</artifactId>
9 </dependency>
```

YML

```
1 server:
2   port: 3355
3
4 spring:
5   application:
6     name: config-client
7   cloud:
8     #Config客户端配置
9     config:
10       label: master #分支名称
11       name: config #配置文件名称
12       profile: dev #读取后缀名称 上述3个综合: master分支上config-dev.yml的配置文
件被读取http://config-3344.com:3344/master/config-dev.yml
13       uri: http://localhost:3344 #配置中心地址
14
15 #rabbitmq相关配置 15672是web管理界面的端口; 5672是MQ访问的端口<-----
16 ---
17   rabbitmq:
18     host: localhost
19     port: 5672
20     username: guest
21     password: guest
22
23   #服务注册到eureka地址
24   eureka:
25     client:
26       service-url:
27         defaultZone: http://localhost:7001/eureka
28
29   # 暴露监控端点
30   management:
31     endpoints:
32       web:
33         exposure:
34           include: "*"
```

给cloud-config-client-3366客户端添加消息总线支持

POM

```

1 <!--添加消息总线RabbitMQ支持-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-bus-amqp</artifactId>
5 </dependency>
6 <dependency>
7   <groupId>org-springframework.boot</groupId>
8   <artifactId>spring-boot-starter-actuator</artifactId>
9 </dependency>

```

YML

```

1 server:
2   port: 3366
3
4 spring:
5   application:
6     name: config-client
7   cloud:
8     #Config客户端配置
9     config:
10    label: master #分支名称
11    name: config #配置文件名称
12    profile: dev #读取后缀名称 上述3个综合: master分支上config-dev.yml的配置文
件被读取http://config-3344.com:3344/master/config-dev.yml
13    uri: http://localhost:3344 #配置中心地址
14
15 #rabbitmq相关配置 15672是web管理界面的端口; 5672是MQ访问的端口<-----
16 -----
17   rabbitmq:
18     host: localhost
19     port: 5672
20     username: guest
21     password: guest
22
23 #服务注册到eureka地址
24 eureka:
25   client:
26     service-url:
27       defaultZone: http://localhost:7001/eureka
28
29 # 暴露监控端点
30 management:
31   endpoints:
32     web:
33       exposure:
34         include: "*"

```

测试

- 启动
 - EurekaMain7001
 - ConfigcenterMain3344
 - ConfigclientMain3355
 - ConfigclientMain3366
- 运维工程师

- 修改Github上配置文件内容，增加版本号
- 发送POST请求
 - curl -X POST "<http://localhost:3344/actuator/bus-refresh>"
 - 一次发送，处处生效
- 配置中心
 - <http://config-3344.com:3344/config-dev.yml>
- 客户端
 - <http://localhost:3355/configInfo>
 - <http://localhost:3366/configInfo>
 - 获取配置信息，发现都已经刷新了

一次修改，广播通知，处处生效

动态刷新定点通知

不想全部通知，只想定点通知

- 只通知3355
- 不通知3366

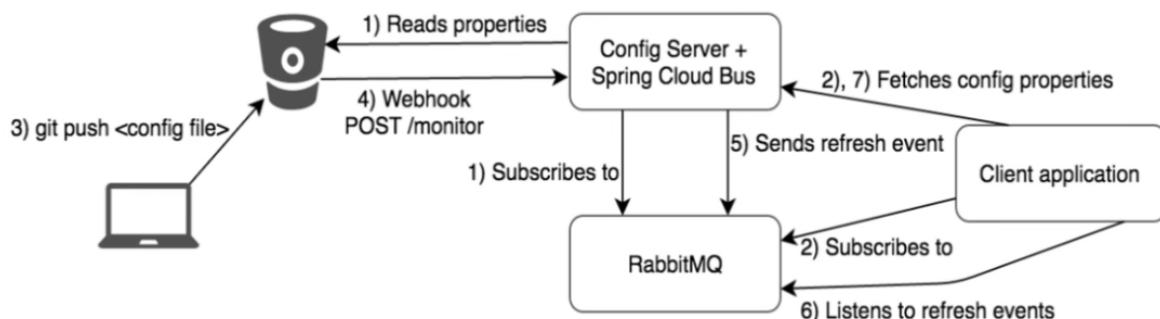
简单一句话 - 指定具体某一个实例生效而不是全部

- 公式：<http://localhost:3344/actuator/bus-refresh/{destination}>
- /bus/refresh请求不再发送到具体的服务实例上，而是发给config server通过destination参数类指定需要更新配置的服务或实例

案例

- 我们这里以刷新运行在3355端口上的config-client（配置文件中设定的应用名称）为例，只通知3355，不通知3366
- curl -X POST "<http://localhost:3344/actuator/bus-refresh/config-client:3355>"

通知总结



消息驱动

Spring Cloud Stream

概述

常见MQ(消息中间件):

- ActiveMQ
- RabbitMQ
- RocketMQ

- Kafka

有没有一种新的技术诞生，让我们不再关注具体MQ的细节，我们只需要用一种适配绑定的方式，自动的给我们在各种MQ内切换。（类似于Hibernate）

Cloud Stream是什么？屏蔽底层消息中间件的差异，降低切换成本，统一消息的编程模型。

是什么

Stream是什么及Binder介绍

[官方文档1](#)

[官方文档2](#)

[Cloud Stream中文指导手册](#)

什么是Spring Cloud Stream？

官方定义Spring Cloud Stream是一个构建消息驱动微服务的框架。

应用程序通过inputs或者 outputs 来与Spring Cloud Stream中binder对象交互。

通过我们配置来binding(绑定)，而Spring Cloud Stream 的binder对象负责与消息中间件交互。所以，我们只需要搞清楚如何与Spring Cloud Stream交互就可以方便使用消息驱动的方式。

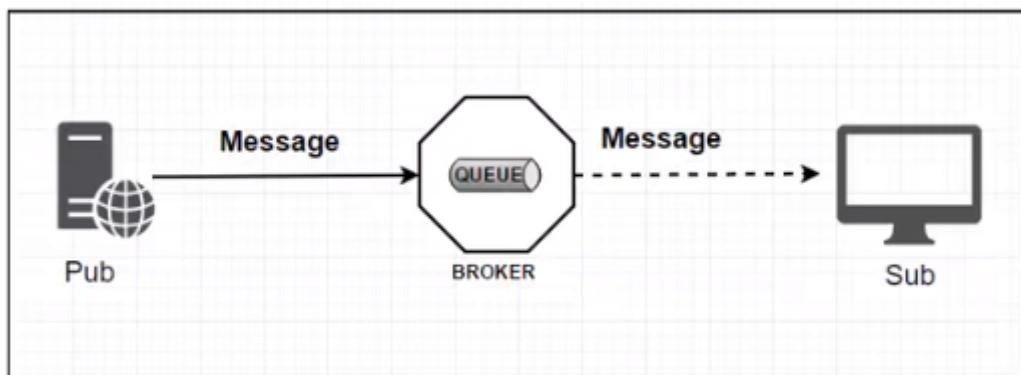
通过使用Spring Integration来连接消息代理中间件以实现消息事件驱动。

Spring Cloud Stream为一些供应商的消息中间件产品提供了个性化的自动化配置实现，引用了发布-订阅、消费组、分区的三个核心概念。

目前仅支持RabbitMQ、 Kafka。

设计思想

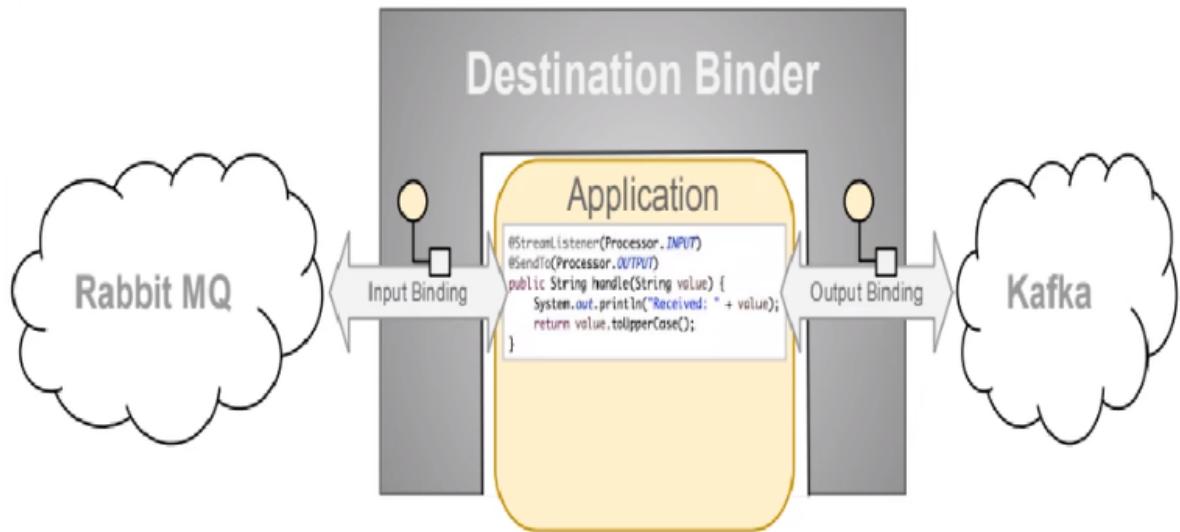
标准MQ



- 生产者/消费者之间靠**消息**媒介传递信息内容
- 消息必须走特定的**通道** - 消息通道 Message Channel
- 消息通道里的消息如何被消费呢，谁负责收发处理 - 消息通道MessageChannel的子接口 SubscribableChannel，由MessageHandler消息处理器所订阅。

为什么用Cloud Stream？

比方说我们用到了RabbitMQ和Kafka，由于这两个消息中间件的架构上的不同，像RabbitMQ有exchange，kafka有Topic和Partitions分区。



这些中间件的差异性导致我们实际项目开发给我们造成了一定的困扰，我们如果用了两个消息队列的其中一种，后面的业务需求，我想往另外一种消息队列进行迁移，这时候无疑就是一个灾难性的，一大堆东西都要重新推倒重新做，因为它跟我们的系统耦合了，这时候Spring Cloud Stream给我们提供了一种解耦合的方式。

Stream凭什么可以统一底层差异？

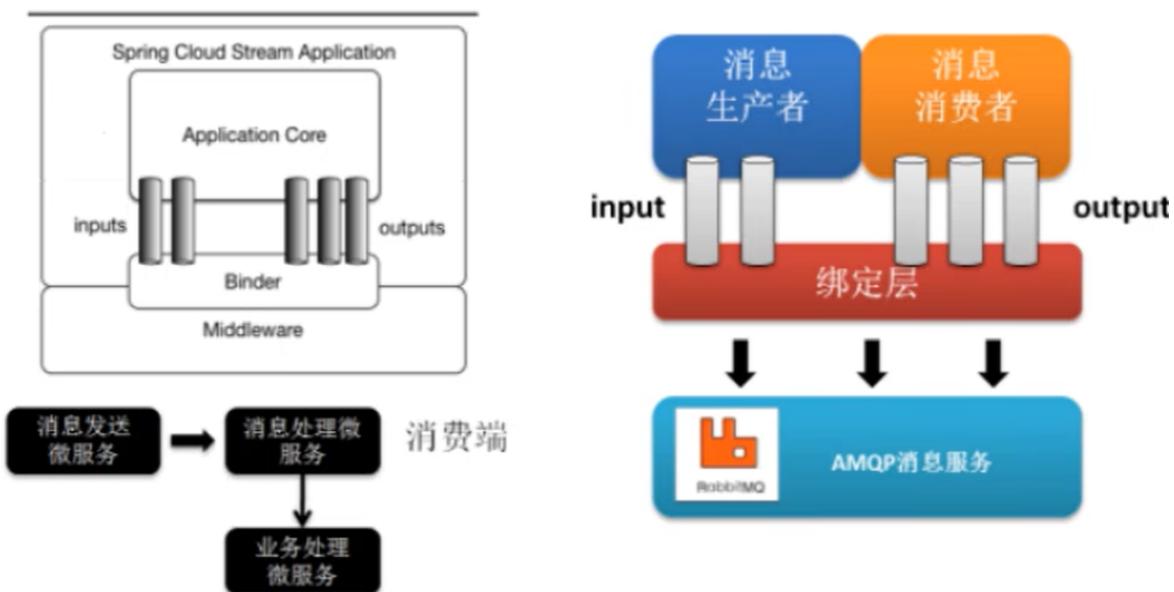
在没有绑定器这个概念的情况下，我们的SpringBoot应用要直接与消息中间件进行信息交互的时候，由于各消息中间件构建的初衷不同，它们的实现细节上会有较大的差异性通过定义绑定器作为中间层，完美地实现了应用程序与消息中间件细节之间的隔离。通过向应用程序暴露统一的Channel通道，使得应用程序不需要再考虑各种不同的消息中间件实现。

通过定义绑定器Binder作为中间层，实现了应用程序与消息中间件细节之间的隔离。

Binder:

- INPUT对应于消费者
- OUTPUT对应于生产者

SpringCloudStream处理架构

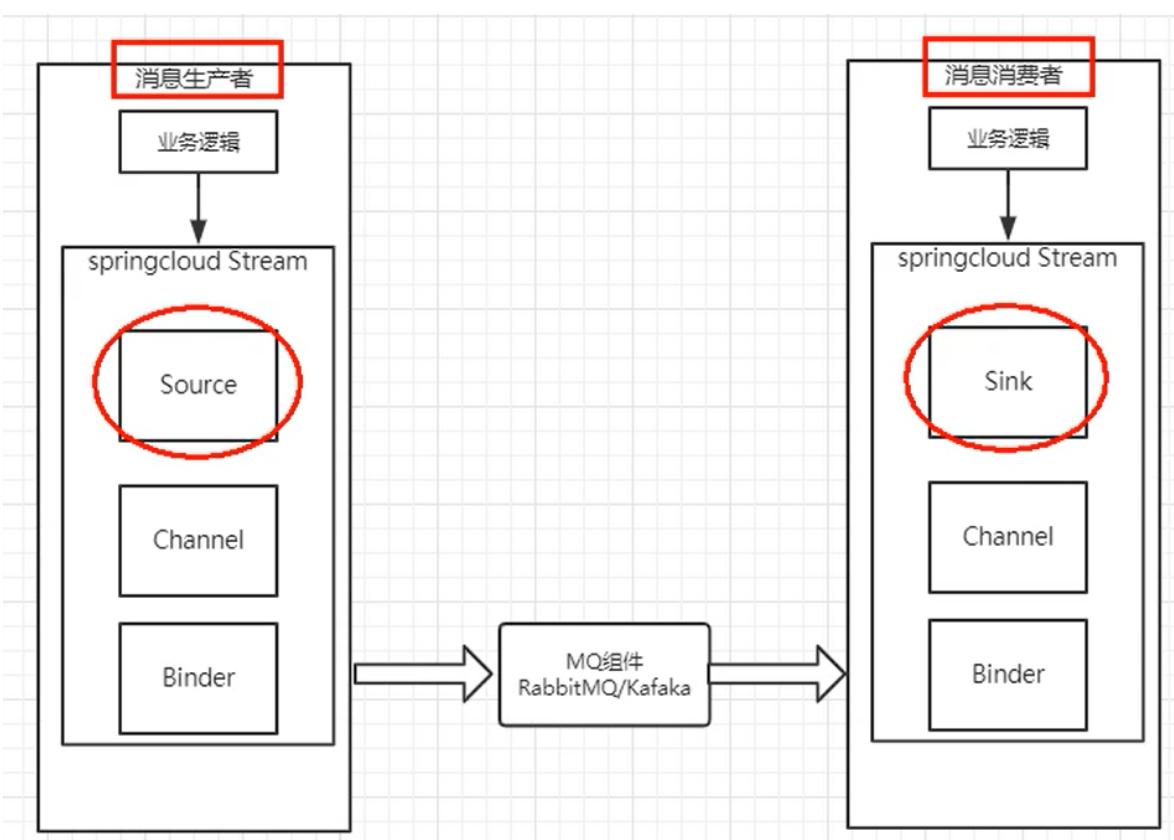
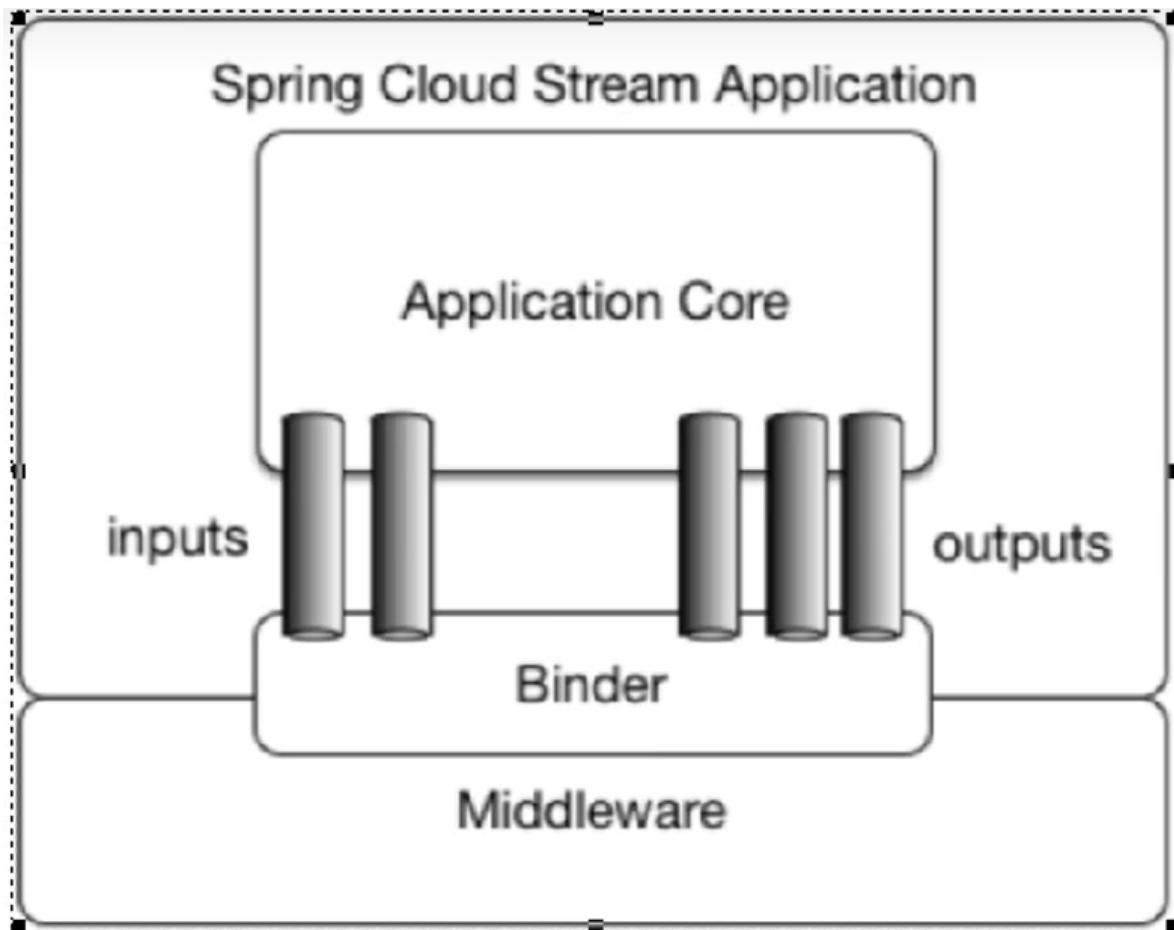


Stream中的消息通信方式遵循了发布-订阅模式

Topic主题进行广播

- 在RabbitMQ就是Exchange
- 在Kafka中就是Topic

标准流程套路



Binder - 很方便的连接中间件，屏蔽差异。

Channel - 通道，是队列Queue的一种抽象，在消息通讯系统中就是实现存储和转发的媒介，通过Channel对队列进行配置。

Source和Sink - 简单的可理解为参照对象是Spring Cloud Stream自身，从Stream发布消息就是输出，接受消息就是输入。

编码API和常用注解

组成	说明
Middleware	中间件，目前只支持RabbitMQ和Kafka
Binder	Binder是应用与消息中间件之间的封装，目前实行了Kafka和RabbitMQ的Binder，通过Binder可以很方便的连接中间件，可以动态的改变消息类型(对于Kafka的topic,RabbitMQ的exchange)，这些都可以通过配置文件来实现
@Input	注解标识输入通道，通过该通道接收到的消息进入应用程序
@Output	注解标识输出通道，发布的消息将通过该通道离开应用程序
@StreamListener	监听队列，用于消费者的队列的消息接收
@EnableBinding	指信道channel和exchange绑定在一起

案例说明

准备RabbitMQ环境

工程中新建三个子模块

- cloud-stream-rabbitmq-provider8801，作为生产者进行发消息模块
- cloud-stream-rabbitmq-consumer8802，作为消息接收模块
- cloud-stream-rabbitmq-consumer8803，作为消息接收模块

消息驱动之生产者

新建Module

名字：cloud-stream-rabbitmq-provider8801

POM

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4             <artifactId>spring-boot-starter-web</artifactId>
5         </dependency>
6     <dependency>
7         <groupId>org.springframework.boot</groupId>
8             <artifactId>spring-boot-starter-actuator</artifactId>
9         </dependency>
10    <dependency>
11        <groupId>org.springframework.cloud</groupId>
12            <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
```

```

13      </dependency>
14  <dependency>
15      <groupId>org.springframework.cloud</groupId>
16      <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
17  </dependency>
18  <!--基础配置-->
19  <dependency>
20      <groupId>org.springframework.boot</groupId>
21      <artifactId>spring-boot-devtools</artifactId>
22      <scope>runtime</scope>
23      <optional>true</optional>
24  </dependency>
25  <dependency>
26      <groupId>org.projectlombok</groupId>
27      <artifactId>lombok</artifactId>
28      <optional>true</optional>
29  </dependency>
30  <dependency>
31      <groupId>org.springframework.boot</groupId>
32      <artifactId>spring-boot-starter-test</artifactId>
33      <scope>test</scope>
34  </dependency>
35 </dependencies>

```

YML

```

1 server:
2   port: 8801
3
4 spring:
5   application:
6     name: cloud-stream-provider
7   cloud:
8     stream:
9       binders: # 在此处配置要绑定的rabbitmq的服务信息;
10      defaultRabbit: # 表示定义的名称, 用于于binding整合
11      type: rabbit # 消息组件类型
12      environment: # 设置rabbitmq的相关的环境配置
13        spring:
14          rabbitmq:
15            host: localhost
16            port: 5672
17            username: guest
18            password: guest
19      bindings: # 服务的整合处理
20        output: # 这个名字是一个通道的名称
21          destination: studyExchange # 表示要使用的Exchange名称定义
22          content-type: application/json # 设置消息类型, 本次为json, 文本则设置“text/plain”
23        binder: defaultRabbit # 设置要绑定的消息服务的具体设置
24
25 eureka:
26   client: # 客户端进行Eureka注册的配置
27     service-url:
28       defaultZone: http://localhost:7001/eureka
29     instance:
30       lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔(默认是30秒)

```

```
31     lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90  
秒）  
32     instance-id: send-8801.com # 在信息列表时显示主机名称  
33     prefer-ip-address: true # 访问的路径变为IP地址
```

主启动类

```
1 import org.springframework.boot.SpringApplication;  
2 import org.springframework.boot.autoconfigure.SpringBootApplication;  
3  
4 @SpringBootApplication  
5 public class StreamMQMain8801 {  
6     public static void main(String[] args) {  
7         SpringApplication.run(StreamMQMain8801.class, args);  
8     }  
9 }
```

业务类

1.发送消息接口

```
1 public interface IMessageProvider {  
2     public String send();  
3 }
```

2.发送消息接口实现类

```
1 import com.lun.springcloud.service.IMessageProvider;  
2 import org.springframework.cloud.stream.annotation.EnableBinding;  
3 import org.springframework.cloud.stream.messaging.Source;  
4 import org.springframework.integration.support.MessageBuilder;  
5 import org.springframework.messaging.MessageChannel;  
6  
7 import javax.annotation.Resource;  
8 import java.util.UUID;  
9  
10 @EnableBinding(Source.class) //定义消息的推送管道  
11 public class MessageProviderImpl implements IMessageProvider{  
12     @Resource  
13     private MessageChannel output; // 消息发送管道  
14  
15     @Override  
16     public String send(){  
17         String serial = UUID.randomUUID().toString();  
18         output.send(MessageBuilder.withPayload(serial).build());  
19         System.out.println("*****serial: "+serial);  
20         return null;  
21     }  
22 }
```

3.Controller

```
1 import com.lun.springcloud.service.IMessageProvider;
2 import org.springframework.web.bind.annotation.GetMapping;
3 import org.springframework.web.bind.annotation.RestController;
4
5 import javax.annotation.Resource;
6
7 @RestController
8 public class SendMessageController{
9     @Resource
10    private IMessageProvider messageProvider;
11
12    @GetMapping(value = "/sendMessage")
13    public String sendMessage() {
14        return messageProvider.send();
15    }
16
17 }
```

测试

- 启动 7001eureka
- 启动 RabpitMq (79_Bus之RabbitMQ环境配置)
 - rabbitmq-plugins enable rabbitmq_management
 - <http://localhost:15672/>

The screenshot shows the RabbitMQ Management Interface with the 'Exchanges' tab selected. The page displays a table of exchanges, with one row, 'studyExchange', highlighted by a red border. The table columns are: Name, Type, Features, Message rate in, and Message rate out. The 'studyExchange' row has 'topic' in the Type column and 'D' in the Features column.

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
springCloudBus	topic	D	0.00/s	0.00/s	
studyExchange	topic	D			

- 启动 8801
- 访问 - <http://localhost:8801/sendMessage>
 - 后台将打印serial: UUID字符串

消息驱动之消费者

新建Module

名字: cloud-stream-rabbitmq-consumer8802

POM

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-web</artifactId>
5     </dependency>
6     <dependency>
7         <groupId>org.springframework.cloud</groupId>
8         <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
9     </dependency>
10    <dependency>
11        <groupId>org.springframework.cloud</groupId>
12        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
13    </dependency>
14    <dependency>
15        <groupId>org.springframework.boot</groupId>
16        <artifactId>spring-boot-starter-actuator</artifactId>
17    </dependency>
18    <!--基础配置-->
19    <dependency>
20        <groupId>org.springframework.boot</groupId>
21        <artifactId>spring-boot-devtools</artifactId>
22        <scope>runtime</scope>
23        <optional>true</optional>
24    </dependency>
25    <dependency>
26        <groupId>org.projectlombok</groupId>
27        <artifactId>lombok</artifactId>
28        <optional>true</optional>
29    </dependency>
30    <dependency>
31        <groupId>org.springframework.boot</groupId>
32        <artifactId>spring-boot-starter-test</artifactId>
33        <scope>test</scope>
34    </dependency>
35 </dependencies>
```

YML

```
1 server:
2   port: 8802
3
4 spring:
5   application:
6     name: cloud-stream-consumer
7   cloud:
8     stream:
9       binders: # 在此处配置要绑定的rabbitmq的服务信息;
```

```

10      defaultRabbit: # 表示定义的名称, 用于于binding整合
11      type: rabbit # 消息组件类型
12      environment: # 设置rabbitmq的相关的环境配置
13          spring:
14              rabbitmq:
15                  host: localhost
16                  port: 5672
17                  username: guest
18                  password: guest
19      bindings: # 服务的整合处理
20          input: # 这个名字是一个通道的名称
21          destination: studyExchange # 表示要使用的Exchange名称定义
22          content-type: application/json # 设置消息类型, 本次为对象json, 如果是
23          文本则设置“text/plain”
24          binder: defaultRabbit # 设置要绑定的消息服务的具体设置
25
26 eureka:
27     client: # 客户端进行Eureka注册的配置
28         service-url:
29             defaultZone: http://localhost:7001/eureka
30         instance:
31             lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔(默认是30秒)
32             lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔(默认是90
33             秒)
34             instance-id: receive-8802.com # 在信息列表时显示主机名称
35             prefer-ip-address: true      # 访问的路径变为IP地址

```

主启动类

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3
4 @SpringBootApplication
5 public class StreamMQMain8802 {
6     public static void main(String[] args) {
7         SpringApplication.run(StreamMQMain8802.class, args);
8     }
9 }

```

业务类

```

1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.cloud.stream.annotation.EnableBinding;
3 import org.springframework.cloud.stream.annotation.StreamListener;
4 import org.springframework.cloud.stream.messaging.Sink;
5 import org.springframework.messaging.Message;
6 import org.springframework.stereotype.Component;
7
8 @Slf4j
9 @Component
10 @EnableBinding(Sink.class)
11 public class ReceiveMessageListenerController{
12     @Value("${server.port}")
13     private String serverPort;
14
15     @StreamListener(Sink.INPUT)

```

```
16     public void input(Message<String> message){  
17         log.info("消费者1号,----->接受到的消息: "+message.getPayload()+"\t  
18             port: "+serverPort);  
19     }  
}
```

测试

- 启动EurekaMain7001
- 启动StreamMQMain8801
- 启动StreamMQMain8802
- 8801发送8802接收消息

分组消费与持久化

消息重复消费

依照8802，克隆出来一份运行8803 - cloud-stream-rabbitmq-consumer8803。

启动

- RabbitMQ
- 服务注册 - 8801
- 消息生产 - 8801
- 消息消费 - 8802
- 消息消费 - 8802

运行后有两个问题

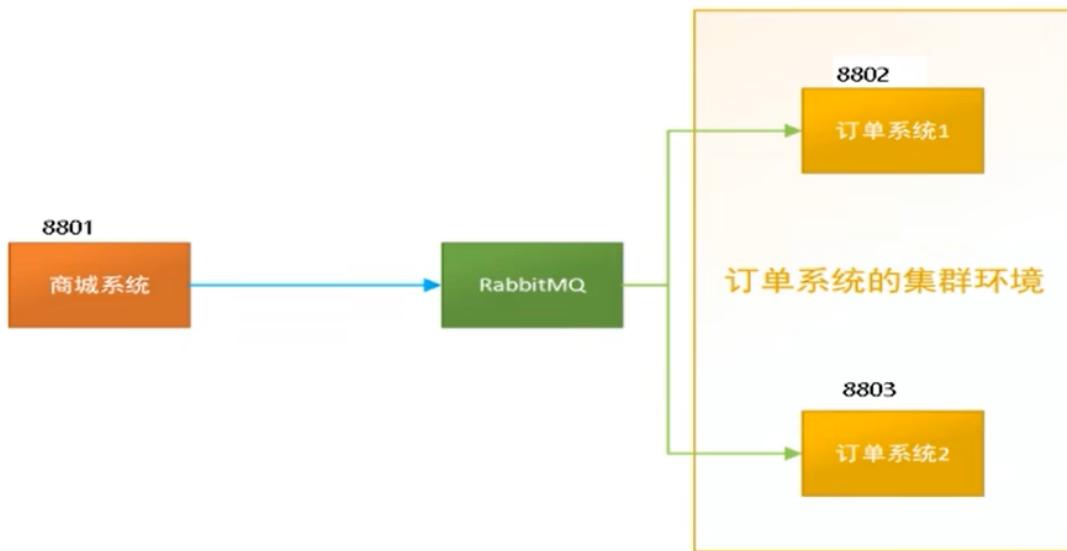
- 有重复消费问题
- 消息持久化问题

消费

- <http://localhost:8801/sendMessage>
- 目前是8802/8803同时都收到了，存在重复消费问题
- 如何解决：分组和持久化属性group（重要）

生产实际案例

比如在如下场景中，订单系统我们做集群部署，都会从RabbitMQ中获取订单信息，那如果一个订单同时被两个服务获取到，那么就会造成数据错误，我们得避免这种情况。这时我们就可以使用Stream中的消息分组来解决。



注意在Stream中处于同一个group中的多个消费者是竞争关系，就能够保证消息只会被其中一个应用消费一次。

不同组是可以全面消费的(重复消费)。

同一组内会发生竞争关系，只要其中一个可以消费。

group解决消息重复消费

原理

微服务应用放置于同一个group中，就能够保证消息只会被其中一个应用消费一次。

不同的组是可以重复消费的，同一个组内会发生竞争关系，只有其中一个可以消费。

8802/8803都变成不同组，group两个不同

group: A_Group、B_Group

8802修改YML

```

1  spring:
2    application:
3      name: cloud-stream-provider
4    cloud:
5      stream:
6        binders: # 在此处配置要绑定的rabbitmq的服务信息;
7          defaultRabbit: # 表示定义的名称，用于于binding整合
8            type: rabbit # 消息组件类型
9            environment: # 设置rabbitmq的相关的环境配置
10           spring:
11             rabbitmq:
12               host: localhost
13               port: 5672
14               username: guest
15               password: guest
16         bindings: # 服务的整合处理
17           output: # 这个名字是一个通道的名称
18             destination: studyExchange # 表示要使用的Exchange名称定义
19             content-type: application/json # 设置消息类型，本次为json，文本则设置“text/plain”
20             binder: defaultRabbit # 设置要绑定的消息服务的具体设置
21             group: A_Group #<-----关键

```

8803修改YML (与8802的类似位置 group: B_Group)

This exchange			
To	Routing key	Arguments	
studyExchange.A_Group	#		<button>Unbind</button>
studyExchange.B_Group	#		<button>Unbind</button>

结论：还是重复消费

8802/8803实现了轮询分组，每次只有一个消费者

8801模块发的消息只能被8802或8803其中一个接收到，这样避免了重复消费。

8802/8803都变成相同组，group两个相同

group: A_Group

8802修改YML group: A_Group

8803修改YML group: A_Group

结论：同一个组的多个微服务实例，每次只会有一个拿到

持久化

停止8802/8803并去除掉8802的分组 group: A_Group，8803的分组 group: A_Group 没有去掉。

8801先发送4条消息到RabbitMq。

先启动8802，无分组属性配置，后台没有打出来消息。

再启动8803，有分组属性配置，后台打出来了MQ上的消息。(消息持久化体现)

链路追踪

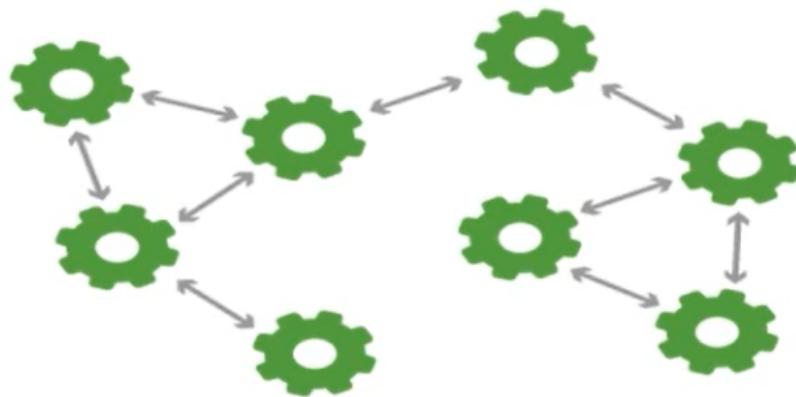
Sleuth

概述

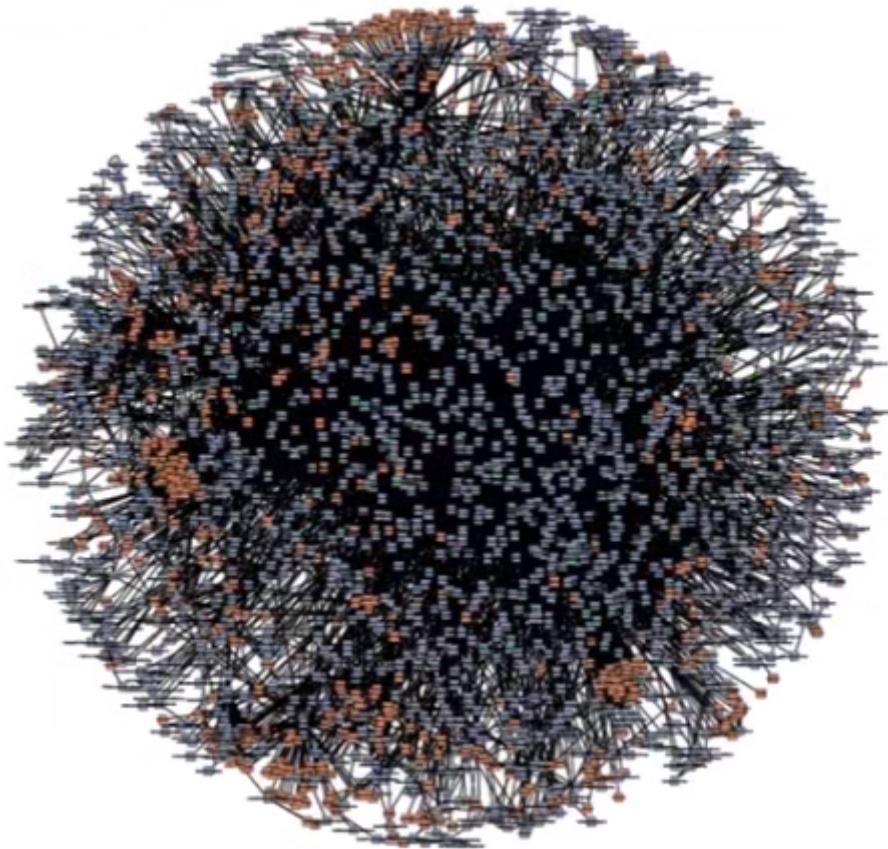
为什么会出现这个技术？要解决哪些问题？

在微服务框架中，一个由客户端发起的请求在后端系统中会经过多个不同的的服务节点调用来协同产生最后的请求结果，每一个前段请求都会形成一条复杂的分布式服务调用链路，链路中的任何一环出现高延时或错误都会引起整个请求最后的失败。

MICROSERVICES ARCHITECTURE



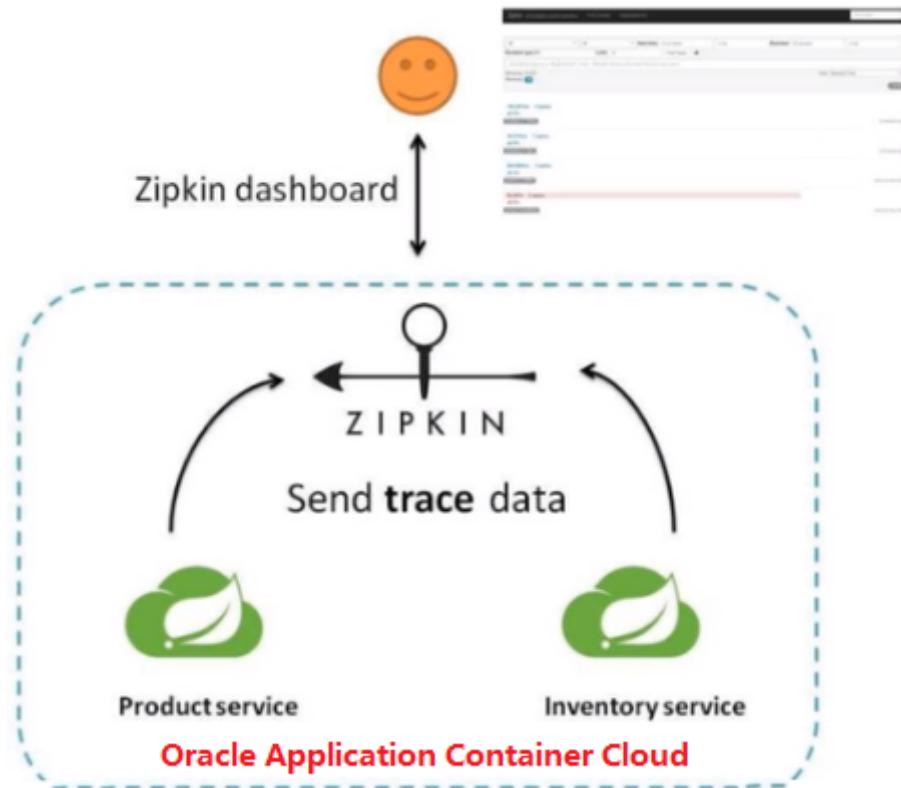
Microservices are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.



是什么

- <https://github.com/spring-cloud/spring-cloud-sleuth>
- Spring Cloud Sleuth提供了一套完整的服务跟踪的解决方案
- 在分布式系统中提供追踪解决方案并且兼容支持了zipkin

解决



sleuth

英 [slu:θ] 美 [slu:θ]

n. 侦探

链路搭建

zipkin搭建安装

1.zipkin

下载

- SpringCloud从F版起已不需要自己构建Zipkin Server了，只需调用jar包即可
- https://search.maven.org/remote_content?g=io.zipkin&a=zipkin-server&v=LATEST&c=exec
- zipkin-server-2.12.9-exec.jar

运行jar

```
1 | java -jar zipkin-server-2.12.9-exec.jar
```

Microsoft Windows [版本 10.0.19044.1826]
(c) Microsoft Corporation。保留所有权利。

D:\studytools\springcloud\zipkin>java -jar zipkin-server-2.12.9-exec.jar

** * *

** ** *

** ** **

:: Powered by Spring Boot :: (v2.1.4.RELEASE)

运行控制台

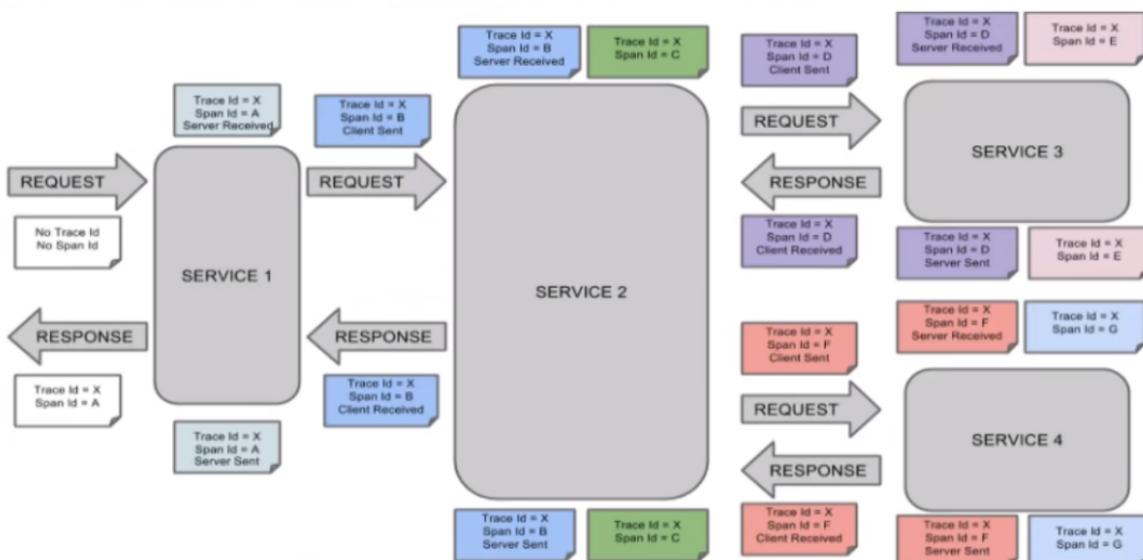
<http://localhost:9411/zipkin/>

The screenshot shows the Zipkin web interface at <http://localhost:9411/zipkin/>. The search bar includes dropdowns for '服务名' (Service Name) set to 'all', 'Span Name', and '时间' (Time) set to '1 小时'. Below the search bar is a query input field: '根据Annotation查询' (Query by Annotation), containing 'For example: http.path=/foo/bar/ and cluster=foo and cache.miss'. To the right of this are filters for '持续时间 (μs) >=' (Duration (μs) >=) with 'Ex: 100ms or 5s', '数量' (Count) with '10', and '排序' (Sort) with '耗时降序' (Sort by Duration Descending). At the bottom left is a blue '查找' (Search) button.

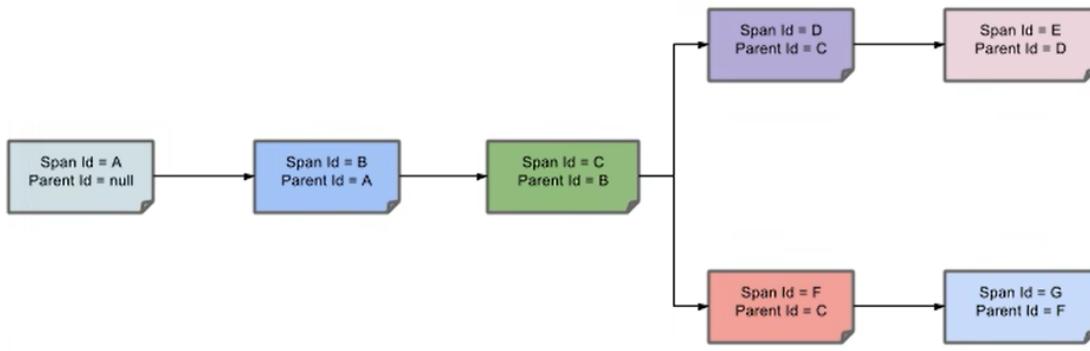
术语

完整的调用链路

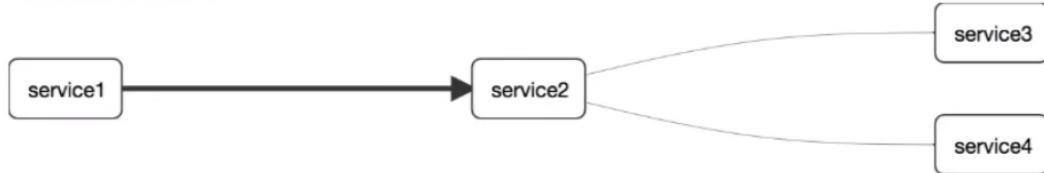
表示一请求链路，一条链路通过Trace Id唯一标识，Span标识发起的请求信息，各span通过parent id关联起来



一条链路通过Trace Id唯一标识，Span标识发起的请求信息，各span通过parent id关联起来。



整个链路的依赖关系如下：



名词解释

- Trace: 类似于树结构的Span集合，表示一条调用链路，存在唯一标识
- span: 表示调用链路来源，通俗的理解span就是一次请求信息

链路监控展现

2.服务提供者

cloud-provider-payment8001

POM

```

1 <!--包含了sleuth+zipkin-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-zipkin</artifactId>
5 </dependency>

```

YML

```

1 spring:
2   application:
3     name: cloud-payment-service
4
5   zipkin: #<-----关键
6     base-url: http://localhost:9411
7   sleuth: #<-----关键
8     sampler:
9       #采样率值介于 0 到 1 之间，1 则表示全部采集
10      probability: 1
11
12   datasource:
13     type: com.alibaba.druid.pool.DruidDataSource          # 当前数据源操作类
14       driver-class-name: org.gjt.mm.mysql.Driver           # mysql驱动包

```

```
15     url: jdbc:mysql://localhost:3306/db2019?
16     useUnicode=true&characterEncoding=utf-8&useSSL=false
17     username: root
18     password: 123456
```

业务类PaymentController

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/payment")
4 public class PaymentController {
5
6     ...
7
8     @GetMapping("/zipkin")
9     public String paymentZipkin() {
10         return "hi ,i'am paymentzipkin server fall back, welcome to here,
11         o(∩_∩)o哈哈~";
12     }
13 }
```

3.服务消费者(调用方)

cloue-consumer-order80

POM

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-zipkin</artifactId>
4 </dependency>
```

YML

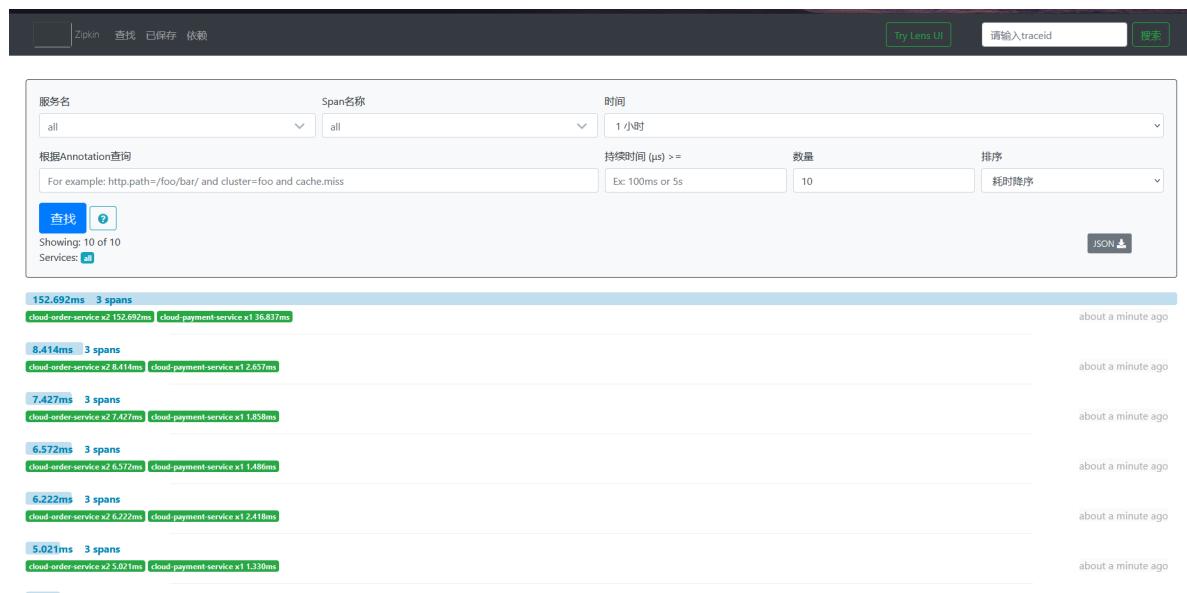
```
1 spring:
2     application:
3         name: cloud-order-service
4     zipkin:
5         base-url: http://localhost:9411
6     sleuth:
7         sampler:
8             probability: 1
```

业务类OrderController

```
1 // =====> zipkin+sleuth
2 @GetMapping("/payment/zipkin")
3 public String paymentZipkin(){
4     String result =
5     restTemplate.getForObject("http://localhost:8001"+"/payment/zipkin/",
6     String.class);
7     return result;
8 }
```

4.依次启动eureka7001/8001/80 - 80调用8001几次测试下

5.打开浏览器访问: <http://localhost:9411>



Spring Cloud Alibaba

概述

为什么会出现SpringCloud alibaba

Spring Cloud Netflix项目进入维护模式

<https://spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now>

什么是维护模式?

将模块置于维护模式, 意味着Spring Cloud团队将不会再向模块添加新功能。

他们将修复block级别的 bug 以及安全问题, 他们也会考虑并审查社区的小型pull request。

SpringCloud alibaba带来了什么 是什么

官网

Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件, 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。

诞生: 2018.10.31, Spring Cloud Alibaba 正式入驻了Spring Cloud官方孵化器, 并在Maven 中央库发布了第一个版本。

能干嘛

- 服务限流降级: 默认支持 WebServlet、WebFlux、OpenFeign、RestTemplate、Spring Cloud Gateway、Zuul、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: 适配 Spring Cloud 服务注册与发现标准, 默认集成了 Ribbon 的支持。
- 分布式配置管理: 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。

- 阿里云对象存储：阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- 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。
- 阿里云短信服务：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

去哪下

如果需要使用已发布的版本，在 dependencyManagement 中添加如下配置。

```

1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>com.alibaba.cloud</groupId>
5       <artifactId>spring-cloud-alibaba-dependencies</artifactId>
6       <version>2.2.5.RELEASE</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10    </dependencies>
11 </dependencyManagement>

```

然后在 dependencies 中添加自己所需使用的依赖即可使用。

怎么玩

- Sentinel：把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。
- Nacos：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- RocketMQ：一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的的消息发布与订阅服务。
- Dubbo：Apache Dubbo™ 是一款高性能 Java RPC 框架。
- Seata：阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。
- Alibaba Cloud OSS：阿里云对象存储服务（Object Storage Service，简称 OSS），是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- Alibaba Cloud SchedulerX：阿里中间件团队开发的一款分布式任务调度产品，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。
- Alibaba Cloud SMS：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

Spring Cloud Alibaba学习资料获取

- 官网
 - <https://spring.io/projects/spring-cloud-alibaba#overview>
- 中文
 - <https://github.com/alibaba/spring-cloud-alibaba/blob/master/README-zh.md>
- 英文
 - <https://github.com/alibaba/spring-cloud-alibaba>
 - <https://spring-cloud-alibaba-group.github.io/github-pages/greenwich/spring-cloud-alibaba.html>

Nacos

简介

为什么叫Nacos

- 前四个字母分别为Naming和Configuration的前两个字母，最后的s为Service。

是什么

- 一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- Nacos: Dynamic Naming and Configuration Service
- Nacos就是注册中心 + 配置中心的组合 -> Nacos = Eureka+Config+Bus

能干嘛

- 替代Eureka做服务注册中心
- 替代Config做服务配置中心

去哪下

- <https://github.com/alibaba/nacos/releases>
- <https://nacos.io>
- [官网文档](#)

各种注册中心

服务注册与发现框架	CAP模型	控制台管理	社区活跃度
Eureka	AP	支持	低(2.x版本闭源)
Zookeeper	CP	不支持	中
consul	CP	支持	高
Nacos	AP	支持	高

安装Nacos

- 本地Java8+Maven环境已经OK先
- 从[官网](#)下载Nacos
- 解压安装包，直接运行bin目录下的startup.cmd
- 使用sublime Text 将集群启动改成单机启动 把 set MODE="cluster" 改为 set MODE="standalone" 即可

```
nacos is starting with cluster
[...]
[Nacos 2.0.3] Running in cluster mode, All function modules
[Nacos 2.0.3] Port: 8848
[Nacos 2.0.3] Pid: 8676
[Nacos 2.0.3] Console: http://192.168.2.133:8848/nacos/index.html
[Nacos 2.0.3] https://nacos.io
```

- 命令运行成功后直接访问<http://localhost:8848/nacos>，默认账号密码都是nacos
- 结果页面

The screenshot shows the Nacos 2.0.3 web interface. The top navigation bar includes links for Home, Documentation, Issues, Community, and Nacos Enterprise Edition. On the left, a sidebar lists modules: Configuration Management, History Version, Monitoring, Service Management, Permission Control, Namespace, and Cluster Management. The main content area is titled 'public' and displays a search bar with placeholder text '添加匹配符“进行模糊查询”' and a dropdown menu for 'Group'. Below the search bar are buttons for 'Add Configuration' and 'Import Configuration'. A table header row includes columns for 'Data ID', 'Group', and '归属应用'. The table body is empty, showing the message '没有数据'.

使用nacos

服务提供者注册

[官方文档](#)

新建Module

名字： cloud-alibaba-provider-payment9001

父POM

```
1 <dependencyManagement>
2   <dependencies>
3     <!--spring cloud alibaba 2.1.0.RELEASE-->
4     <dependency>
5       <groupId>com.alibaba.cloud</groupId>
6       <artifactId>spring-cloud-alibaba-dependencies</artifactId>
7       <version>2.1.0.RELEASE</version>
8       <type>pom</type>
9       <scope>import</scope>
10      </dependency>
11    </dependencies>
12  </dependencyManagement>
```

本模块POM

```
1 <dependencies>
2   <!--SpringCloud alibaba nacos -->
3   <dependency>
4     <groupId>com.alibaba.cloud</groupId>
5     <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
6   </dependency>
7   <!-- SpringBoot整合Web组件 -->
8   <dependency>
9     <groupId>org.springframework.boot</groupId>
10    <artifactId>spring-boot-starter-web</artifactId>
11  </dependency>
12  <dependency>
13    <groupId>org.springframework.boot</groupId>
14    <artifactId>spring-boot-starter-actuator</artifactId>
15  </dependency>
16  <!--日常通用jar包配置-->
17  <dependency>
18    <groupId>org.springframework.boot</groupId>
19    <artifactId>spring-boot-devtools</artifactId>
```

```
20         <scope>runtime</scope>
21         <optional>true</optional>
22     </dependency>
23     <dependency>
24         <groupId>org.projectlombok</groupId>
25         <artifactId>lombok</artifactId>
26         <optional>true</optional>
27     </dependency>
28     <dependency>
29         <groupId>org.springframework.boot</groupId>
30         <artifactId>spring-boot-starter-test</artifactId>
31         <scope>test</scope>
32     </dependency>
33 </dependencies>
```

YML

```
1 server:
2   port: 9001
3
4 spring:
5   application:
6     name: nacos-payment-provider
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #配置Nacos地址
11
12 management:
13   endpoints:
14     web:
15       exposure:
16         include: '*'
```

主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
4
5 @EnableDiscoveryClient
6 @SpringBootApplication
7 public class PaymentMain9001 {
8     public static void main(String[] args) {
9         SpringApplication.run(PaymentMain9001.class, args);
10    }
11 }
```

业务类

```
1 import org.springframework.beans.factory.annotation.Value;
2 import org.springframework.web.bind.annotation.GetMapping;
3 import org.springframework.web.bind.annotation.PathVariable;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
```

```

7  @RequestMapping("/payment")
8  public class PaymentController {
9      @Value("${server.port}")
10     private String serverPort;
11
12     @GetMapping(value = "/nacos/{id}")
13     public String getPayment(@PathVariable("id") Integer id) {
14         return "nacos registry, serverPort: " + serverPort + "\t id" + id;
15     }
16 }

```

测试

- <http://localhost:9001/payment/nacos/1>
- nacos控制台
- nacos服务注册中心+服务提供者9001都OK了

为了下一章节演示nacos的负载均衡，参照9001新建9002

- 新建cloud-alibaba-provider-payment9002
- 9002其它步骤你懂的
- 或者取巧不想新建重复体力劳动，可以利用IDEA功能，直接拷贝虚拟端口映射

启动9002

服务消费者注册和负载

新建Module

名字: cloud-alibaba-consumer-nacos-order83

POM

```

1 <dependencies>
2     <!--SpringCloud alibaba nacos -->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
6     </dependency>
7     <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
8     <dependency>
9         <groupId>org.example</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <!-- SpringBoot整合Web组件 -->
14    <dependency>
15        <groupId>org.springframework.boot</groupId>
16        <artifactId>spring-boot-starter-web</artifactId>
17    </dependency>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-starter-actuator</artifactId>
21    </dependency>
22    <!--日常通用jar包配置-->
23    <dependency>
24        <groupId>org.springframework.boot</groupId>
25        <artifactId>spring-boot-devtools</artifactId>
26        <scope>runtime</scope>
27        <optional>true</optional>
28    </dependency>
29    <dependency>
30        <groupId>org.projectlombok</groupId>
31        <artifactId>lombok</artifactId>
32        <optional>true</optional>
33    </dependency>
34    <dependency>
```

```
35         <groupId>org.springframework.boot</groupId>
36         <artifactId>spring-boot-starter-test</artifactId>
37         <scope>test</scope>
38     </dependency>
39 </dependencies>
```

为什么nacos支持负载均衡？因为spring-cloud-starter-alibaba-nacos-discovery内含netflix-ribbon包。

YML

```
1 server:
2   port: 83
3
4 spring:
5   application:
6     name: nacos-order-consumer
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848
11
12 #消费者将要去访问的微服务名称(注册成功进nacos的微服务提供者)
13 service-url:
14   nacos-user-service: http://nacos-payment-provider
```

主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
4
5 @EnableDiscoveryClient
6 @SpringBootApplication
7 public class OrderNacosMain83{
8     public static void main(String[] args){
9         SpringApplication.run(OrderNacosMain83.class,args);
10    }
11 }
```

业务类

ApplicationContextConfig

```

1 import org.springframework.cloud.client.loadbalancer.LoadBalanced;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.web.client.RestTemplate;
5
6 @Configuration
7 public class ApplicationContextConfig{
8     @Bean
9     @LoadBalanced
10    public RestTemplate getRestTemplate(){
11        return new RestTemplate();
12    }
13}

```

OrderNacosController

```

1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.beans.factory.annotation.Value;
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5 import org.springframework.web.bind.annotation.RestController;
6 import org.springframework.web.client.RestTemplate;
7
8 import javax.annotation.Resource;
9
10 @Slf4j
11 @RestController
12 @RequestMapping("/consumer")
13 public class OrderNacosController {
14
15     @Resource
16     private RestTemplate restTemplate;
17
18     @Value("${service-url.nacos-user-service}")
19     private String serverURL;
20
21     @GetMapping(value = "/payment/nacos/{id}")
22     public String paymentInfo(@PathVariable("id") Long id){
23         return
24             restTemplate.getForObject(serverURL+"/payment/nacos/"+id,String.class);
25     }
26 }

```

测试

- 启动nacos控制台

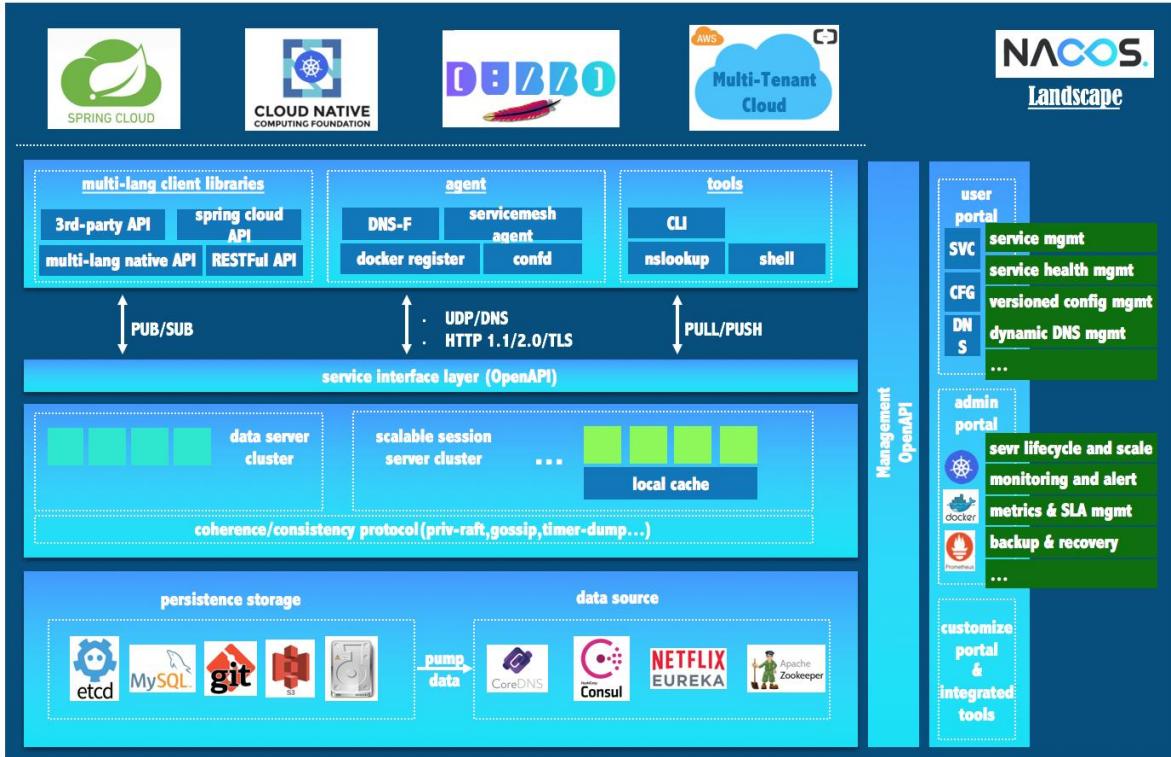
服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
nacos-order-consumer	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
nacos-payment-provider	DEFAULT_GROUP	1	2	2	false	详情 示例代码 订阅者 删除

- <http://localhost:83/Eonsumer/payment/nacos/13>

- 83访问9001/9002，轮询负载OK

Nacos服务注册中心对比提升

Nacos全景图



Nacos和CAP

Nacos与其他注册中心特性对比

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

Nacos服务发现实例模型



Nacos支持AP和CP模式的切换

C是所有节点在同一时间看到的数据是一致的; 而A的定义是所有的请求都会收到响应。

何时选择使用何种模式?

一般来说, 如果不需要存储服务级别的信息且服务实例是通过nacos-client注册, 并能够保持心跳上报, 那么就可以选择AP模式。当前主流的服务如Spring cloud和Dubbo服务, 都适用于AP模式, AP模式为了服务的可能性而减弱了一致性, 因此AP模式下只支持注册临时实例。

如果需要在服务级别编辑或者存储配置信息, 那么CP是必须, K8S服务和DNS服务则适用于CP模式。CP模式下则支持注册持久化实例, 此时则是以Raft协议为集群运行模式, 该模式下注册实例之前必须先注冊服务, 如果服务不存在, 则会返回错误。

切换命令:

```
1 | curl -X PUT '$NACOS_SERVER:8848/nacos/v1/ns/operator/switches?
entry=serverMode&value=CP'
```

服务配置中心

基础配置

新建module

名字: cloud-alibaba-config-nacos-client3377

POM

```
1 <dependencies>
2   <!--nacos-config-->
3   <dependency>
4     <groupId>com.alibaba.cloud</groupId>
5     <artifactId>spring-cloud-starter-alibaba-nacos-
config</artifactId>
6   </dependency>
7   <!--nacos-discovery-->
8   <dependency>
9     <groupId>com.alibaba.cloud</groupId>
10    <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
```

```

11      </dependency>
12      <!--web + actuator-->
13      <dependency>
14          <groupId>org.springframework.boot</groupId>
15          <artifactId>spring-boot-starter-web</artifactId>
16      </dependency>
17      <dependency>
18          <groupId>org.springframework.boot</groupId>
19          <artifactId>spring-boot-starter-actuator</artifactId>
20      </dependency>
21      <!--一般基础配置-->
22      <dependency>
23          <groupId>org.springframework.boot</groupId>
24          <artifactId>spring-boot-devtools</artifactId>
25          <scope>runtime</scope>
26          <optional>true</optional>
27      </dependency>
28      <dependency>
29          <groupId>org.projectlombok</groupId>
30          <artifactId>lombok</artifactId>
31          <optional>true</optional>
32      </dependency>
33      <dependency>
34          <groupId>org.springframework.boot</groupId>
35          <artifactId>spring-boot-starter-test</artifactId>
36          <scope>test</scope>
37      </dependency>
38  </dependencies>

```

YML

Nacos同springcloud-config一样，在项目初始化时，要保证先从配置中心进行配置拉取，拉取配置之后，才能保证项目的正常启动。

springboot中配置文件的加载是存在优先级顺序的，bootstrap优先级高于application

bootstrap

```

1 # nacos配置
2 server:
3     port: 3377
4
5 spring:
6     application:
7         name: nacos-config-client
8     cloud:
9         nacos:
10            discovery:
11                server-addr: localhost:8848 #Nacos服务注册中心地址
12            config:
13                server-addr: localhost:8848 #Nacos作为配置中心地址
14                file-extension: yaml #指定yaml格式的配置
15
16
17 #
${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}

```

```
18 # nacos-config-client-dev.yaml  
19  
20 # nacos-config-client-test.yaml ----> config.info
```

application

```
1 spring:  
2   profiles:  
3     active: dev # 表示开发环境  
4     #active: test # 表示测试环境  
5     #active: info
```

主启动

```
1 import org.springframework.boot.SpringApplication;  
2 import org.springframework.boot.autoconfigure.SpringBootApplication;  
3 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
4  
5 @EnableDiscoveryClient  
6 @SpringBootApplication  
7 public class NacosConfigClientMain3377{  
8     public static void main(String[] args) {  
9         SpringApplication.run(NacosConfigClientMain3377.class, args);  
10    }  
11 }
```

业务类

```
1 import org.springframework.beans.factory.annotation.Value;  
2 import org.springframework.cloud.context.config.annotation.RefreshScope;  
3 import org.springframework.web.bind.annotation.GetMapping;  
4 import org.springframework.web.bind.annotation.RestController;  
5  
6 @RestController  
7 @RefreshScope //支持Nacos的动态刷新功能。  
8 public class ConfigclientController{  
9     @Value("${config.info}")  
10     private String configInfo;  
11  
12     @GetMapping("/config/info")  
13     public String getConfigInfo() {  
14         return configInfo;  
15     }  
16 }
```

在Nacos中添加配置信息

Nacos中的dataid的组成格式及与SpringBoot配置文件中的匹配规则

官方文档

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：

```
1 | ${prefix}-${spring-profile.active}.${file-extension}
```

- `prefix` 默认为 `spring.application.name` 的值，也可以通过配置项 `spring.cloud.nacos.config.prefix` 来配置。
- `spring.profile.active` 即为当前环境对应的 profile，详情可以参考 Spring Boot 文档。注意：当 `spring.profile.active` 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 `${prefix}.${file-extension}`
- `file-extension` 为配置内容的数据格式，可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。目前只支持 properties 和 yaml 类型。
- 通过 Spring Cloud 原生注解 @RefreshScope 实现配置自动更新

最后公式：

```
1 | ${spring.application.name}-${spring.profiles.active}.${spring.cloud.nacos.config.file-extension}
```

配置新增

The screenshot shows the Nacos 2.0.3 configuration management interface. On the left, there's a sidebar with options like Configuration Management, History Version, Listener Query, Service Management, Application Control, Namespace, and Cluster Management. The main area is titled 'public' and shows a table of configurations. One row is selected, showing 'Data ID: nacos-config-client-dev.yaml' and 'Group: DEFAULT_GROUP'. At the bottom right of the table, there are buttons for 'Details', 'Example Code', 'Edit', 'Delete', and 'More'. Below the table, there are pagination controls ('每页显示: 10', '< 上一页', '下一页 >'). A red arrow points to the '+' button in the top right corner of the main content area.

Nacos 界面配置对应-设置 DataId

The screenshot shows the Nacos configuration editor. It has fields for 'Data ID' (set to 'nacos-config-client-dev.yaml'), 'Group' (set to 'DEFAULT_GROUP'), and a 'Description' field. Below these, there's a 'Format' section with radio buttons for TEXT, JSON, XML, YAML (which is selected), HTML, and Properties. Underneath is a code editor with a YAML snippet:
```yaml  
1 config:  
2 info: nacos config center.version=1  
```  
At the bottom right, there are 'Publish' and 'Cancel' buttons. A red box highlights the 'YAML' button in the format section, and a red arrow points to the 'Publish' button.

注意：在 nacos 中，一定写成 yaml。和 bootstrap 里的保持一致。

配置小结



测试

- 启动前需要在nacos客户端-配置管理-配置管理栏目下有对应的yaml配置文件
- 运行cloud-config-nacos-client3377的主启动类
- 调用接口查看配置信息 - <http://localhost:3377/config/info>

自带动态刷新

修改下Nacos中的yaml配置文件，再次调用查看配置的接口，就会发现配置已经刷新。

命名空间、分组和DataID三者关系

问题 - 多环境多项目管理

问题1：

实际开发中，通常一个系统会准备

- dev开发环境
- test测试环境
- prod生产环境。

如何保证指定环境启动时服务能正确读取到Nacos上相应环境的配置文件呢？

问题2：

一个大型分布式微服务系统会有很多微服务子项目，每个微服务项目又都会有相应的开发环境、测试环境、预发环境、正式环境...那怎么对这些微服务配置进行管理呢？

Nacos的图形化管理界面

The screenshot shows the Nacos 2.0.3 graphical management interface. The left sidebar has a '命名空间' (Namespace) section highlighted with a red arrow. The main content area shows the '配置管理 | public' tab selected, also highlighted with a red arrow. A configuration entry for 'data id: it' is shown, with its 'Group: it' also highlighted with a red arrow. The interface includes search and filter options like 'Data ID' and 'Group'.

NACOS 2.0.3

命名空间

配置管理

配置列表

历史版本

监听查询

服务管理

权限控制

命名空间

集群管理

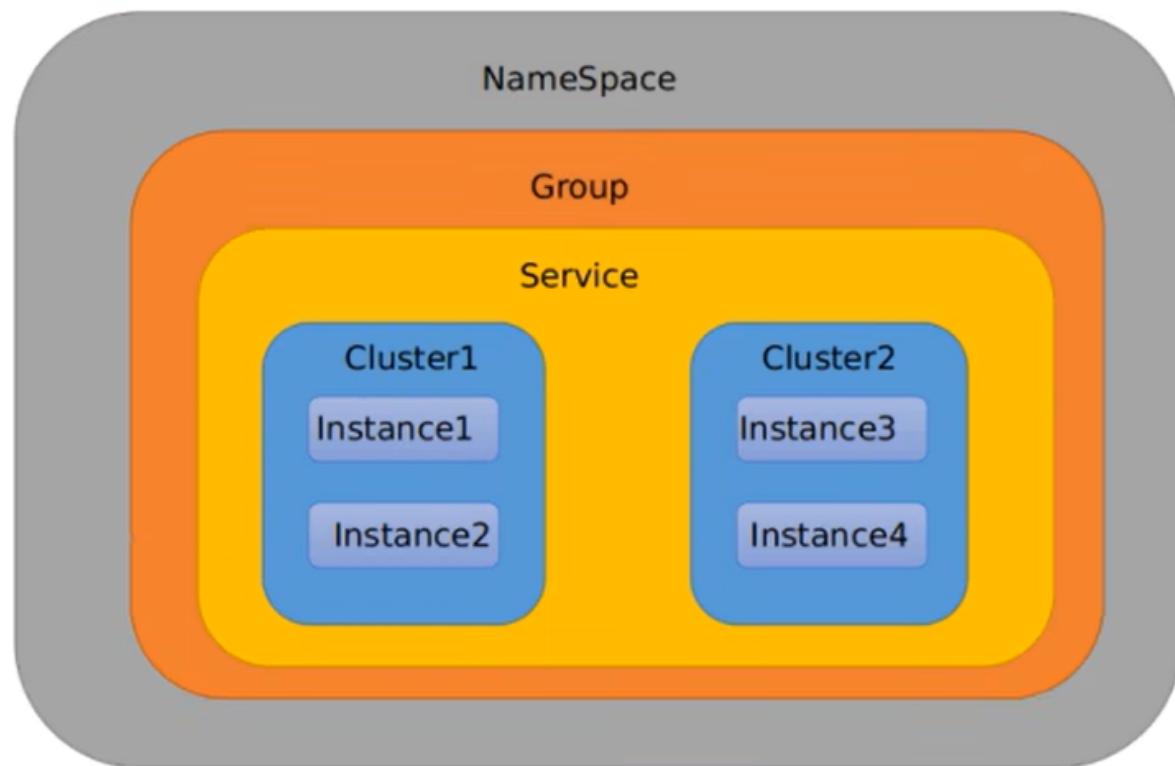
命名空间名称
public(保留空间)

Namespace+Group+Data ID三者关系? 为什么这么设计?

1、是什么

类似Java里面的package名和类名最外层的namespace是可以用于区分部署环境的，Group和DataID逻辑上区分两个目标对象。

2、三者情况



默认情况: Namespace=public, Group=DEFAULT_GROUP, 默认Cluster是DEFAULT

- Nacos默认的Namespace是public, Namespace主要用来实现隔离。
 - 比方说我们现在有三个环境：开发、测试、生产环境，我们就可以创建三个Namespace，不同的Namespace之间是隔离的。

- Group默认是DEFAULT_GROUP, Group可以把不同的微服务划分到同一个分组里面去
- Service就是微服务:一个Service可以包含多个Cluster (集群), Nacos默认Cluster是DEFAULT, Cluster是对指定微服务的一个虚拟划分。
 - 比方说为了容灾, 将Service微服务分别部署在了杭州机房和广州机房, 这时就可以给杭州机房的Service微服务起一个集群名称(HZ), 给广州机房的Service微服务起一个集群名称(GZ), 还可以尽量让同一个机房的微服务互相调用, 以提升性能。
- 最后是Instance, 就是微服务的实例。

DataID配置

指定spring.profile.active和配置文件的DataID来使不同环境下读取不同的配置

默认空间+默认分组+新建dev和test两个DataID

- 新建dev配置DataID

The screenshot shows the Nacos 2.0.3 configuration management interface. On the left, there's a sidebar with 'NACOS 2.0.3' at the top, followed by sections like '配置管理' (Configuration Management), '配置列表' (Configuration List) which is highlighted with a red box and has a red arrow pointing to it, '历史版本' (History Versions), '监听查询' (Listen Query), '服务管理' (Service Management), '权限控制' (Permission Control), '命名空间' (Namespace), and '集群管理' (Cluster Management). The main area is titled '编辑配置' (Edit Configuration). It shows fields for 'Data ID': 'nacos-config-client-dev.yaml' (highlighted with a red box), 'Group': 'DEFAULT_GROUP' (highlighted with a red box), '描述' (Description), 'Beta发布' (Beta Release) with a checkbox, and '配置格式' (Configuration Format) with radio buttons for TEXT, JSON, XML, YAML (which is selected and highlighted with a red box), HTML, and Properties. Below these is a code editor labeled '配置内容' (Configuration Content) containing the following YAML code:

```

1 config:
2   info: nacos config center.version=2

```

- 新建test配置DataId

新建配置

This screenshot shows the 'New Configuration' dialog in the Nacos 2.0.3 interface. It includes fields for 'Data ID': 'nacos-config-client-test.yaml' (highlighted with a red box), 'Group': 'DEFAULT_GROUP', '更多高级选项' (More Advanced Options), '描述' (Description), '配置格式' (Configuration Format) with radio buttons for TEXT, JSON, XML, YAML (selected and highlighted with a red box), HTML, and Properties, and a '配置内容' (Configuration Content) code editor. The code editor contains the following YAML code:

```

* 配置内容: ① :
1 config:
2   info: from nacos config center,nacos-config-test.yaml,version=2

```

同一个Group下两个配置文件

The screenshot shows the Nacos 2.0.3 graphical interface. On the left, there's a sidebar with options like '配置管理', '历史版本', '监听查询', '服务管理', '权限控制', '命名空间', and '集群管理'. The main area is titled 'public' and contains a table for '配置管理'. The table has columns for 'Data ID', 'Group', '归属应用', and '操作'. There are two rows: one for 'nacos-config-client-dev.yaml' with 'Group: DEFAULT_GROUP' and another for 'nacos-config-client-test.yaml' also with 'Group: DEFAULT_GROUP'. Both rows have a red box around them.

Group配置

通过Group实现环境区分 - 新建Group

新建配置

This screenshot shows the 'Create Configuration' dialog. It includes fields for 'Data ID' (nacos-config-client-info.yaml), 'Group' (DEV_GROUP, highlighted with a red box), and a 'Description' field. Below these are configuration format options (TEXT, JSON, XML, YAML, HTML, Properties) and a code editor. The code editor shows a single line of YAML: 'config: info: nacos-config-client-info.yaml,DEV_GROUP'. The 'YAML' option is selected.

在nacos图形界面控制台上面新建配置文件DataID

This screenshot shows the 'public' configuration list again. A new row for 'nacos-config-client-info.yaml' is visible, with 'Group: TEST_GROUP' highlighted by a red box. The 'Data ID' column also has a red box around it. The table structure is identical to the first screenshot, with columns for Data ID, Group, Application, and Operations.

bootstrap+application

在config下增加一条group的配置即可。可配置为DEV_GROUP或TEST GROUP

```

bootstrap.yml
1 # nacos配置
2 server:
3   port: 3377
4
5 spring:
6   application:
7     name: nacos-config-client
8   cloud:
9     nacos:
10    discovery:
11      server-addr: localhost:8848 #Nacos服务注册中心地址
12    config:
13      server-addr: localhost:8848 #Nacos作为配置中心地址
14      file-extension: yaml #指定yaml格式的配置
15      group: TEST_GROUP
16
17
18 # ${spring.application.name}-${spring.profile.active}.${spring.config.format}
19 # nacos-config-client-dev.yaml
20
21 # nacos-config-client-test.yaml ----> config.info

```

```

application.yml
1 spring:
2   profiles:
3     # active: dev # 表示开发环境
4     # active: test # 表示测试环境
5     active: info

```

Namespace配置

新建dev/test的Namespace

命名空间名称	命名空间ID	配对数	操作
public(保留空间)		4	详情 删除 编辑
dev	a5019df9-577b-4b87-a5a4-98158dc84405	0	详情 删除 编辑

回到服务管理-服务列表查看

服务名	分组名	提供数目	实例数	健康实例数	触发保护值	操作
nacos-order-consumer	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
nacos-payment-provider	DEFAULT_GROUP	1	2	2	false	详情 示例代码 订阅者 删除
nacos-config-client	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除

按照域名配置填写

Data ID & Group	归属应用	操作
nacos-config-client-dev.yaml	DEFAULT_GROUP	详情 示例代码 编辑 删除 更多
nacos-config-client-dev.yaml	DEV_GROUP	详情 示例代码 编辑 删除 更多
nacos-config-client-dev.yaml	TEST_GROUP	详情 示例代码 编辑 删除 更多

yml

```

1 # nacos配置
2 server:
3   port: 3377
4

```

```

5  spring:
6    application:
7      name: nacos-config-client
8    cloud:
9      nacos:
10        discovery:
11          server-addr: localhost:8848 #Nacos服务注册中心地址
12        config:
13          server-addr: localhost:8848 #Nacos作为配置中心地址
14          file-extension: yaml #指定yaml格式的配置
15          group: TEST_GROUP
16          namespace: a5019df9-577b-4b87-a5a4-98158dc84405 #<-----指定
17          namespace
18
19  #
20  ${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.con
21  fig.file-extension}
22  # nacos-config-client-dev.yaml
23
24  # nacos-config-client-test.yaml ----> config.info

```

集群架构与持久化

[官方文档](#)

[官网架构图](#)

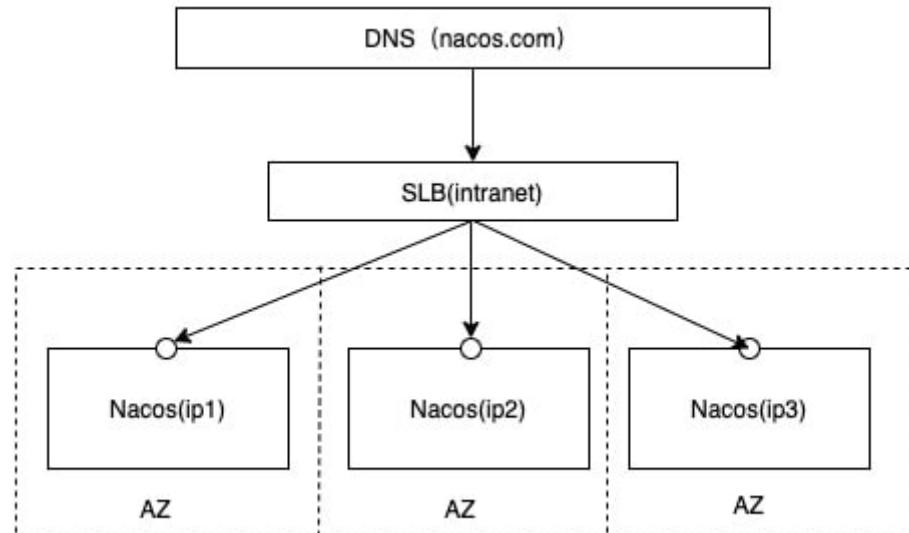
[集群部署架构图](#)

因此开源的时候推荐用户把所有服务列表放到一个vip下面，然后挂到一个域名下面

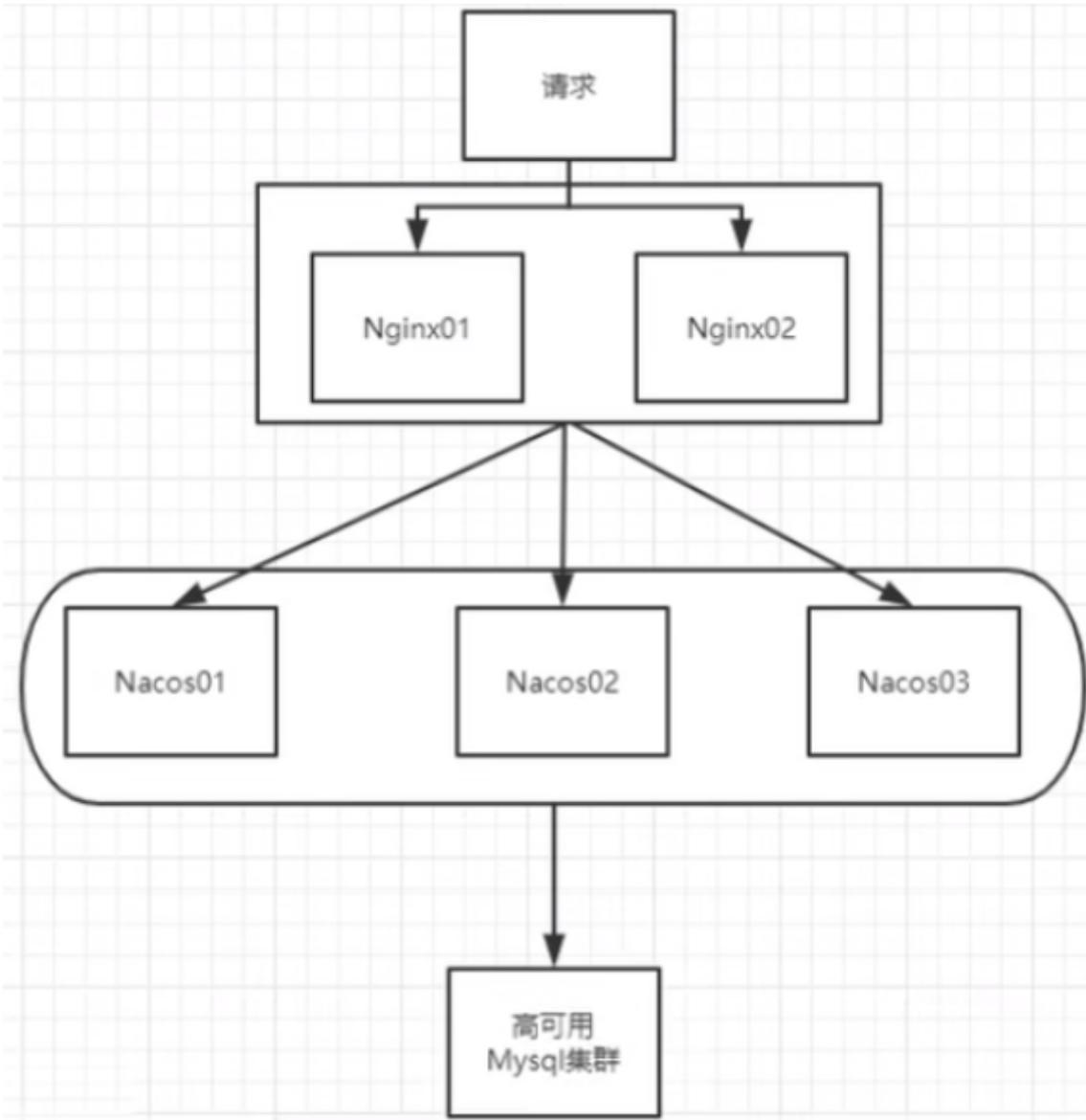
<http://ip1:port/openAPI>直连ip模式，机器挂则需要修改ip才可以使用。

<http://VIP:port/openAPI>挂载VIP模式，直连vip即可，下面挂server真实ip，可读性不好。

<http://nacos.com:port/openAPI>域名 + VIP模式，可读性好，而且换ip方便，推荐模式



上图官网翻译，真实情况



按照上述，我们需要mysql数据库。

[官网说明](#)

默认Nacos使用嵌入式数据库实现数据的存储。所以，如果启动多个默认配置下的Nacos节点，数据存储是存在一致性问题的。为了解决这个问题，Nacos采用了集中式存储的方式来支持集群化部署，目前只支持MySQL的存储。

Nacos支持三种部署模式

- 单机模式-用于测试和单机试用。
- 集群模式-用于生产环境，确保高可用。
- 多集群模式-用于多数据中心场景。

单机模式

Windows

cmd startup.cmd或者双击startup.cmd文件

在0.7版本之前，在单机模式时nacos使用嵌入式数据库实现数据的存储，不方便观察数据存储的基本情况。0.7版本增加了支持mysql数据源能力，具体的操作步骤：

- 安装数据库，版本要求:5.6.5+
- 初始化mysql数据库，数据库初始化文件: nacos-mysql.sql

- 修改conf/application.properties文件，增加支持mysql数据源配置（目前只支持mysql），添加mysql数据源的url、用户名和密码。

```

1 spring.datasource.platform=mysql
2
3 db.num=1
4 db.url.0=jdbc:mysql://127.0.0.1:3306/nacos_config?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&serverTimezone=UTC
5 db.user=nacos_devtest
6 db.password=youdontknow

```

再以单机模式启动nacos，nacos所有写嵌入式数据库的数据都写到了mysql。

集群

预计需要，1个Nginx+3个[nacos](#)注册中心+1个mysql

官方要求

请确保是在环境中安装使用：

- 64 bit OS Linux/Unix/Mac，推荐使用Linux系统。
- 64 bit JDK 1.8+；[下载配置](#)。
- Maven 3.2.x+；[下载配置](#)。
- 3个或3个以上Nacos节点才能构成集群。

[link](#)

Nacos下载Linux版

- <https://github.com/alibaba/nacos/releases/tag/2.0.3>
- nacos-server-2.0.3.tar.gz 解压后安装

集群配置步骤(重点)

1. Linux服务器上mysql数据库配置

SQL脚本在哪里 - 目录nacos/conf/nacos-mysql.sql

```

[root@node1 conf]# pwd
/root/soft/nacos/conf
[root@node1 conf]# ls
application.properties application.properties.example cluster.conf.example nacos-logback.xml nacos-mysql.sql schema.sql
[root@node1 conf]#

```

自己Linux机器上的Mysql数据库上运行

2. application.properties配置

添加以下内容，设置数据源

```

1 spring.datasource.platform=mysql
2
3 db.num=1
4 db.url.0=jdbc:mysql://localhost:3306/nacos_devtest?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&serverTimezone=UTC
5 db.user=root
6 db.password=1234

```

3. Linux服务器上nacos的集群配置cluster.conf

梳理出3台nacos集器的不同服务端口号，设置3个端口：

- 3333
- 4444
- 5555

可以先将cluster.conf复制一份。一方配置错误

```
1 | cp cluster.conf.example cluster.conf
```

修改cluster.conf

内容

```
1 | 192.168.111.144:3333
2 | 192.168.111.144:4444
3 | 192.168.111.144:5555
```

注意，这个IP不能写127.0.0.1，必须是Linux命令 hostname -i 能够识别的IP

```
[root@node1 conf]# hostname -i
192.168.160.128
```

4.编辑Nacos的启动脚本startup.sh，使它能够接受不同的启动端口

/nacos/bin目录下有startup.sh

```
[root@node1 bin]# pwd
/root/soft/nacos/bin
[root@node1 bin]# ll
total 28
-rwxr-xr-x. 1 502 games 954 Oct 11 2019 shutdown.cmd
-rwxr-xr-x. 1 502 games 949 Nov  3 2019 shutdown.sh
-rwxr-xr-x. 1 502 games 2854 Nov  3 2019 startup.cmd
-rwxr-xr-x. 1 502 games 4801 Nov  5 2019 startup.sh
-rwxr-xr-x. 1 root root  4801 Jul 20 04:30 startup.sh.bk
[root@node1 bin]#
```

平时单机版的启动，都是./startup.sh即可

但是，集群启动，我们希望可以类似其它软件的shell命令，传递不同的端口号启动不同的nacos实例。
命令：./startup.sh -p 3333表示启动端口号为3333的nacos服务器实例，和上一步的cluster.conf配置的一致。

修改内容

```

57 while getopts ":m:f:s:p:" opt
58 do
59     case $opt in
60         m)
61             MODE=$OPTARG;;
62         f)
63             FUNCTION_MODE=$OPTARG;;
64         s)
65             SERVER=$OPTARG;;
66         p)
67             PORT=$OPTARG;;
68         ?)
69             echo "Unknown parameter"
70             exit 1;;
71     esac
72 done

```

```

132 # start
133 echo "$JAVA ${JAVA_OPT}" > ${BASE_DIR}/logs/start.out 2>&1 &
134 nohup $JAVA -Dserver.port=${PORT} ${JAVA_OPT} nacos.nacos >> ${BASE_DIR}/logs/start.out 2>&1 &
135 echo "nacos is starting, you can check the ${BASE_DIR}/logs/start.out"
-- INSERT --

```

执行方式 - `startup.sh - p 端口号`

```

[root@node1 bin]# ./startup.sh -p 3333
/root/soft/jdk/bin/java -server -Xms2g -Xmx2g -Xmn1g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m
mpPath=/root/soft/nacos/logs/java_heapdump.hprof -XX:-UseLargePages -Djava.ext.dirs=/root/soft/jdk/jre
acos/plugins/mysql -Xloggc:/root/soft/nacos/logs/nacos_gc.log -verbose:gc -XX:+PrintGCDetails -XX:+Pri
OfGCLogFile=10 -XX:GCLogFile=100M -Dnacos.home=/root/soft/nacos -Dloader.path=/root/soft/nacos/pl
g.location=classpath:/,classpath:/config/,file:./,file:./config/,file:/root/soft/nacos/conf/ --logging
ize=524288
nacos is starting with cluster
nacos is starting, you can check the /root/soft/nacos/logs/start.out
[root@node1 bin]# ./startup.sh -p 4444
/root/soft/jdk/bin/java -server -Xms2g -Xmx2g -Xmn1g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m
mpPath=/root/soft/nacos/logs/java_heapdump.hprof -XX:-UseLargePages -Djava.ext.dirs=/root/soft/jdk/jre
acos/plugins/mysql -Xloggc:/root/soft/nacos/logs/nacos_gc.log -verbose:gc -XX:+PrintGCDetails -XX:+Pri
OfGCLogFile=10 -XX:GCLogFile=100M -Dnacos.home=/root/soft/nacos -Dloader.path=/root/soft/nacos/pl
g.location=classpath:/,classpath:/config/,file:./,file:./config/,file:/root/soft/nacos/conf/ --logging
ize=524288
nacos is starting with cluster
nacos is starting, you can check the /root/soft/nacos/logs/start.out
[root@node1 bin]# ./startup.sh -p 5555
/root/soft/jdk/bin/java -server -Xms2g -Xmx2g -Xmn1g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m
mpPath=/root/soft/nacos/logs/java_heapdump.hprof -XX:-UseLargePages -Djava.ext.dirs=/root/soft/jdk/jre
acos/plugins/mysql -Xloggc:/root/soft/nacos/logs/nacos_gc.log -verbose:gc -XX:+PrintGCDetails -XX:+Pri
OfGCLogFile=10 -XX:GCLogFile=100M -Dnacos.home=/root/soft/nacos -Dloader.path=/root/soft/nacos/pl
g.location=classpath:/,classpath:/config/,file:./,file:./config/,file:/root/soft/nacos/conf/ --logging
ize=524288
nacos is starting with cluster
nacos is starting, you can check the /root/soft/nacos/logs/start.out
[root@node1 bin]#

```

使用命令查看是否启动成功：

```
ps -ef | grep nacos|grep -v grep|wc -l
```

```
[root@node1 bin]# ps -ef | grep nacos|grep -v grep|grep|wc -l
3
```

Sentinel

概述

[官方Github](#)

[官方文档](#)

Sentinel 是什么？

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。[Sentinel](#) 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保护服务的稳定性。

Sentinel 具有以下特征：

- 丰富的应用场景：**Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。
- 完备的实时监控：**Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- 广泛的开源生态：**Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。
- 完善的 SPI 扩展点：**Sentinel 提供简单易用、完善的 SPI 扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

Sentinel 的主要特性：

[link](#)



一句话解释，之前我们讲解过的Hystrix。

Hystrix与Sentinel比较：

- Hystrix
 - 需要我们程序员自己手工搭建监控平台
 - 没有一套 web 界面可以给我们进行更加细粒度得配置流控、速率控制、服务熔断、服务降级
- Sentinel
 - 单独一个组件，可以独立出来。
 - 直接界面化的细粒度统一配置。

约定 > 配置 > 编码

都可以写在代码里面，但是我们本次还是大规模的学习使用配置和注解的方式，尽量少写代码

sentinel

英 ['sentɪnl] 美 ['sentɪnl]

n. 哨兵

下载安装运行

官方文档

服务使用中的各种问题：

- 服务雪崩
- 服务降级
- 服务熔断
- 服务限流

Sentinel 分为两个部分：

- 核心库 (Java 客户端) 不依赖任何框架/库，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，不需要额外的 Tomcat 等应用容器。

安装步骤：

- 下载
 - <https://github.com/alibaba/Sentinel/releases>
 - 下载到本地 sentinel-dashboard-1.8.4.jar
- 运行命令
 - 前提
 - Java 8 环境
 - 8080端口不能被占用
 - 命令
 - `java -jar sentinel-dashboard-1.8.4.jar`

```
D:\studytools\springcloud\sentinel>java -jar sentinel-dashboard-1.8.4.jar
INFO: Sentinel log output type is: file
INFO: Sentinel log charset is: utf-8
INFO: Sentinel log base directory is: C:\Users\Serendipity\logs\csp\
INFO: Sentinel log name use pid is: false
INFO: Sentinel log level is: INFO


:: Spring Boot ::      (v2.5.12)

2022-07-20 17:54:48.474  INFO 2872 --- [           main] c.a.c.s.dashboard.DashboardApplication : Starting DashboardApplication using Java 1.8.0_311 on DESKTOP-3058MDQ with PID 2872 (D:\studytools\springcloud\sentinel)
2022-07-20 17:54:48.477  INFO 2872 --- [           main] c.a.c.s.dashboard.DashboardApplication : No active profile set, falling back to 1 default profile: "default"
2022-07-20 17:54:49.622  INFO 2872 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-07-20 17:54:49.631  INFO 2872 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-07-20 17:54:49.632  INFO 2872 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.60]
2022-07-20 17:54:49.771  INFO 2872 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring embedded WebApplicationContext
2022-07-20 17:54:49.771  INFO 2872 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1256 ms
```

- 访问Sentinel管理界面
 - localhost:8080
 - 登录账号密码均为sentinel



初始化监控

启动Nacos8848成功

新建工程

名称: cloud-alibaba-sentinel-service8401

POM

```
1 <dependencies>
2     <dependency><!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
3         <groupId>org.example</groupId>
4         <artifactId>cloud-api-commons</artifactId>
5         <version>${project.version}</version>
6     </dependency>
7     <!--SpringCloud alibaba nacos -->
8     <dependency>
9         <groupId>com.alibaba.cloud</groupId>
10        <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
11    </dependency>
12    <!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
13    <dependency>
14        <groupId>com.alibaba.csp</groupId>
15        <artifactId>sentinel-datasource-nacos</artifactId>
16    </dependency>
17    <!--SpringCloud alibaba sentinel -->
18    <dependency>
19        <groupId>com.alibaba.cloud</groupId>
20        <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
21    </dependency>
22    <!--openfeign-->
23    <dependency>
24        <groupId>org.springframework.cloud</groupId>
25        <artifactId>spring-cloud-starter-openfeign</artifactId>
26    </dependency>
27    <!-- SpringBoot整合Web组件+actuator -->
28    <dependency>
29        <groupId>org.springframework.boot</groupId>
30        <artifactId>spring-boot-starter-web</artifactId>
31    </dependency>
32    <dependency>
33        <groupId>org.springframework.boot</groupId>
34        <artifactId>spring-boot-starter-actuator</artifactId>
35    </dependency>
36    <!--日常通用jar包配置-->
37    <dependency>
38        <groupId>org.springframework.boot</groupId>
39        <artifactId>spring-boot-devtools</artifactId>
40        <scope>runtime</scope>
41        <optional>true</optional>
```

```
42      </dependency>
43      <dependency>
44          <groupId>cn.hutool</groupId>
45          <artifactId>hutool-all</artifactId>
46          <version>4.6.3</version>
47      </dependency>
48      <dependency>
49          <groupId>org.projectlombok</groupId>
50          <artifactId>lombok</artifactId>
51          <optional>true</optional>
52      </dependency>
53      <dependency>
54          <groupId>org.springframework.boot</groupId>
55          <artifactId>spring-boot-starter-test</artifactId>
56          <scope>test</scope>
57      </dependency>
58  </dependencies>
```

YML

```
1 server:
2   port: 8401
3
4 spring:
5   application:
6     name: cloudalibaba-sentinel-service
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #Nacos服务注册中心地址
11   sentinel:
12     transport:
13       dashboard: localhost:8080 #配置Sentinel dashboard地址
14       #默认8719端口, 假如被占用会自动从8719开始依次加1 扫描, 直至找到未被占用的端口
15       port: 8719
16
17 management:
18   endpoints:
19     web:
20       exposure:
21         include: '*'
22
23 feign:
24   sentinel:
25     enabled: true # 激活Sentinel对Feign的支持
```

主启动

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
4
5 @EnableDiscoveryClient
6 @SpringBootApplication
7 public class MainApp8401 {
8     public static void main(String[] args) {
9         SpringApplication.run(MainApp8401.class, args);
10    }
11 }

```

业务类

FlowLimitController

```

1 import com.alibaba.csp.sentinel.annotation.SentinelResource;
2 import com.alibaba.csp.sentinel.slots.block.BlockException;
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import java.util.concurrent.TimeUnit;
9
10 @Slf4j
11 @RestController
12 public class FlowLimitController {
13     @GetMapping("/testA")
14     public String testA(){
15         return "-----testA";
16     }
17
18     @GetMapping("/testB")
19     public String testB(){
20         log.info(Thread.currentThread().getName()+"\t"+...testB");
21         return "-----testB";
22     }
23 }

```

启动Sentinel8080 - `java -jar sentinel-dashboard-1.8.4.jar`

启动微服务8401

启动8401微服务后查看sentienl控制台

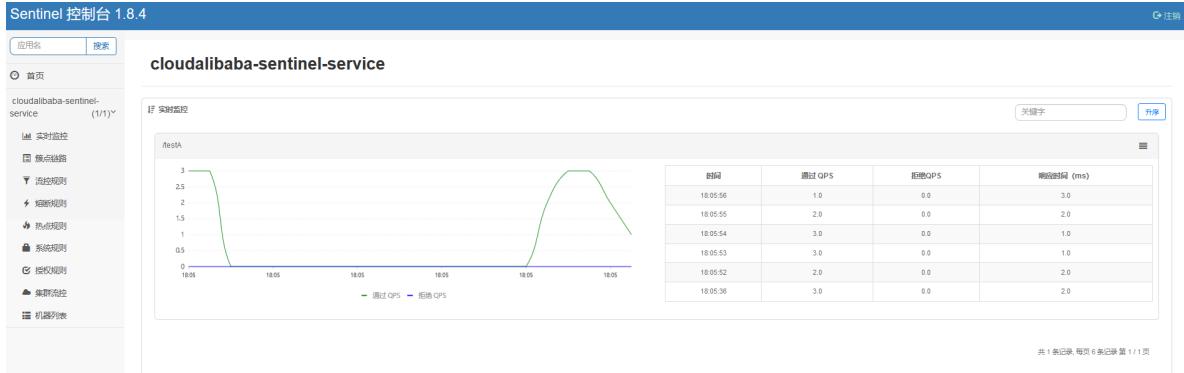
- 刚启动，空空如也，啥都没有



- Sentinel采用的懒加载说明
 - 执行一次访问即可
 - <http://localhost:8401/testA>

■ <http://localhost:8401/testB>

- 效果 - sentinel8080正在监控微服务8401



流控规则

基本介绍



进一步解释说明：

- 资源名：唯一名称，默认请求路径。
- 针对来源：Sentinel可以针对调用者进行限流，填写微服务名，默认default（不区分来源）。
- 阈值类型/单机阈值：
 - QPS(每秒钟的请求数量)：当调用该API的QPS达到阈值的时候，进行限流。
 - 线程数：当调用该API的线程数达到阈值的时候，进行限流。
- 是否集群：不需要集群。
- 流控模式：
 - 直接：API达到限流条件时，直接限流。
 - 关联：当关联的资源达到阈值时，就限流自己。
 - 链路：只记录指定链路上的流量（指定资源从入口资源进来的流量，如果达到阈值，就进行限流）【API级别的针对来源】。
- 流控效果：
 - 快速失败：直接失败，抛异常。
 - Warm up：根据Code Factor（冷加载因子，默认3）的值，从阈值/codeFactor，经过预热时长，才达到设置的QPS阈值。
 - 排队等待：匀速排队，让请求以匀速的速度通过，阈值类型必须设置为QPS，否则无效。

QPS直接失败

直接 -> 快速失败 (系统默认)

配置及说明

表示1秒钟内查询1次就是OK，若超过次数1，就直接->快速失败，报默认错误



测试

快速多次点击访问<http://localhost:8401/testA>

结果

返回页面 Blocked by Sentinel (flow limiting)

源码

com.alibaba.csp.sentinel.slots.block.flow.controller.DefaultController

思考

直接调用默认报错信息，技术方面OK，但是，是否应该有我们自己的后续处理？类似有个fallback的兜底方法？

线程数直接失败

线程数：当调用该API的线程数达到阈值的时候，进行限流。

编辑流控规则

资源名	/testA
针对来源	default
阈值类型	<input type="radio"/> QPS <input checked="" type="radio"/> 并发线程数
单机阈值	1
是否集群	<input type="checkbox"/>
高级选项	
保存	取消

关联

是什么？

- 当自己关联的资源达到阈值时，就限流自己
- 当与A关联的资源B达到阈值后，就限流A自己（B惹事，A挂了）

设置testA

当关联资源/testB的QPS阈值超过1时，就限流/testA的Rest访问地址，**当关联资源到阈值后限制配置好的资源名。**

编辑流控规则

资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
单机阈值	1
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路
关联资源	/testB
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待
关闭高级选项	
保存 取消	

ApiPost模拟密集访问testB

ApiPost - API 文档、调试、Mock、测试一体化协作平台 V6.0.3

菜单 克隆模型的私有团队 + 新建 ... ○ Cookie管理器 □ 归档管理 □ 分享项目 四 全局参数 ○ 参数筛选库 四 默认环境 ...

最近 APIs 分享 测试 笔记

搜索目录或接口 全部 > 切换流程 + 新建流程 保存测试流程 重新开始 0 通过 0 失败 接口通过: 259

新建流程 2022-07-20 18:30:54

执行 1000 次

间隔 0 ms

遇到错误继续执行 沙盒模式运行

第1次 GET TEST http://localhost:8401/testB 200 OK 5ms 1KB 查看请求

第2次 GET TEST http://localhost:8401/testB 200 OK 3ms 1KB 查看请求

第3次 GET TEST http://localhost:8401/testB 200 OK 2ms 1KB 查看请求

第4次 GET TEST http://localhost:8401/testB 200 OK 3ms 1KB 查看请求

第5次

点击左侧列表接口，可以添加指定接口到以下待测试接口列表

GET TEST localhost:8401/testB

ApiPost 官网 内置 Mock 变量 问题反馈 帮助文档 控制台 上下分屏 分屏显示 深灰模式 缩放 设置 检查更新

ApiPost运行后，点击访问<http://localhost:8401/testA>，发现testA挂了

- 结果 Blocked by Sentinel(flow limiting)

链路

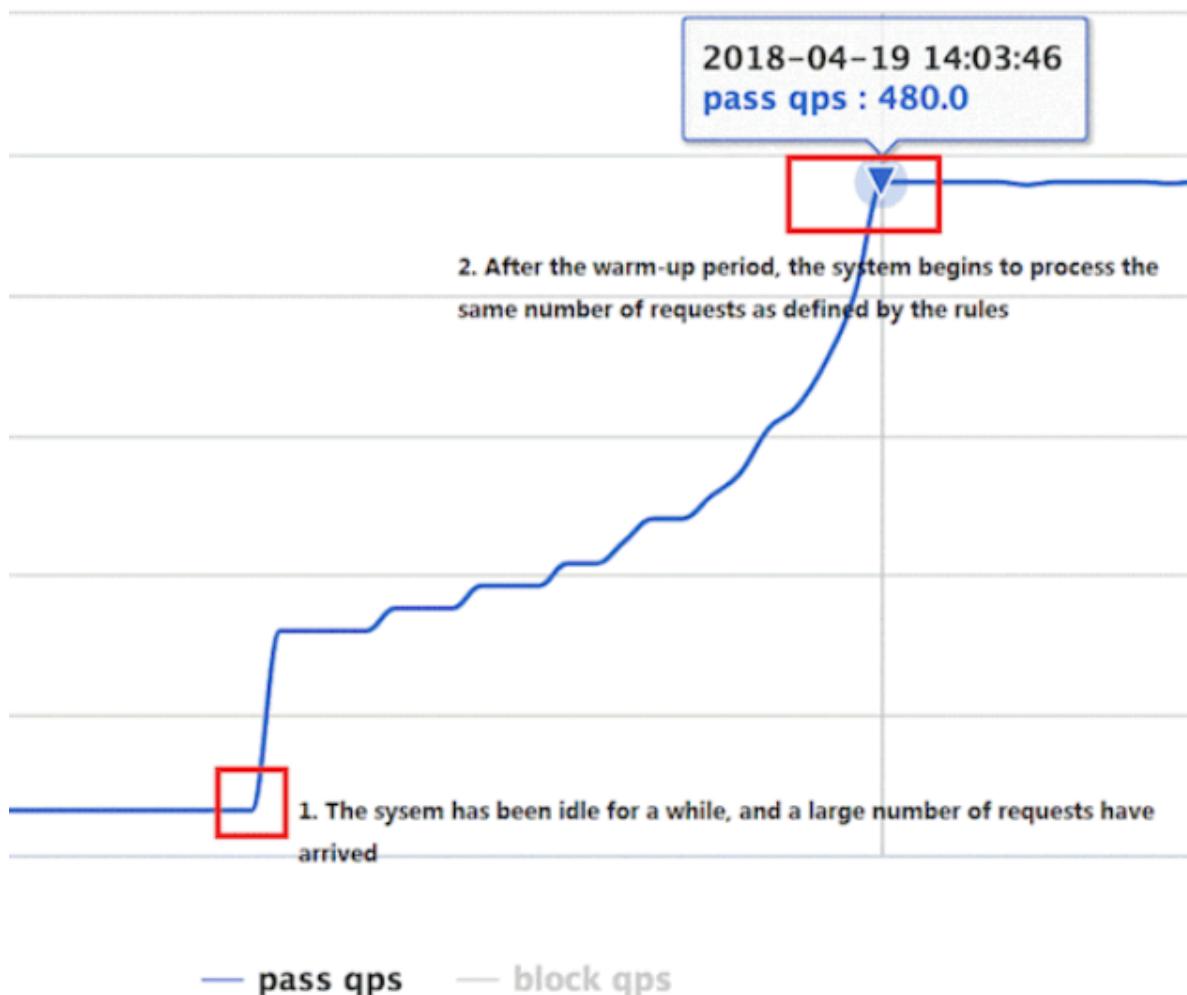
只记录指定链路上的流量（指定资源从入口资源进来的流量，如果达到阈值，就进行限流）【API级别的针对来源】

预热

Warm Up

Warm Up (`RuleConstant.CONTROL_BEHAVIOR_WARM_UP`) 方式，即预热/冷启动方式。当系统长期处于低水位的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。通过“冷启动”，让通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，给冷系统一个预热的时间，避免冷系统被压垮。详细文档可以参考[流量控制 - Warm Up 文档](#)，具体的例子可以参见[WarmUpFlowDemo](#)。

通常冷启动的过程系统允许通过的 QPS 曲线如下图所示：[link](#)



默认coldFactor为3，即请求QPS从`threshold / 3`开始，经预热时长逐渐升至设定的QPS阈值。[link](#)

源码 - com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController

WarmUp配置

案例，阀值为10+预热时长设置5秒。

系统初始化的阀值为 $10 / 3$ 约等于3,即阀值刚开始为3; 然后过了5秒后阀值才慢慢升高恢复到10

编辑流控规则

资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
单机阈值	10
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路
关联资源	/testB
流控效果	<input type="radio"/> 快速失败 <input checked="" type="radio"/> Warm Up <input type="radio"/> 排队等待
预热时长	5
关闭高级选项	
<input type="button" value="保存"/>	<input type="button" value="取消"/>

测试

多次快速点击<http://localhost:8401/testB> - 刚开始不行，后续慢慢OK

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，慢慢的把阀值增长到设置的阀值。

排队等待

匀速排队，让请求以均匀的速度通过，阀值类型必须设成QPS，否则无效。

设置：/testA每秒1次请求，超过的话就排队等待，等待的超时时间为20000毫秒。

编辑流控规则

资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数
单机阈值	1
是否集群	<input type="checkbox"/>
流控模式	<input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路
流控效果	<input type="radio"/> 快速失败 <input type="radio"/> Warm Up <input checked="" type="radio"/> 排队等待
超时时间	20000

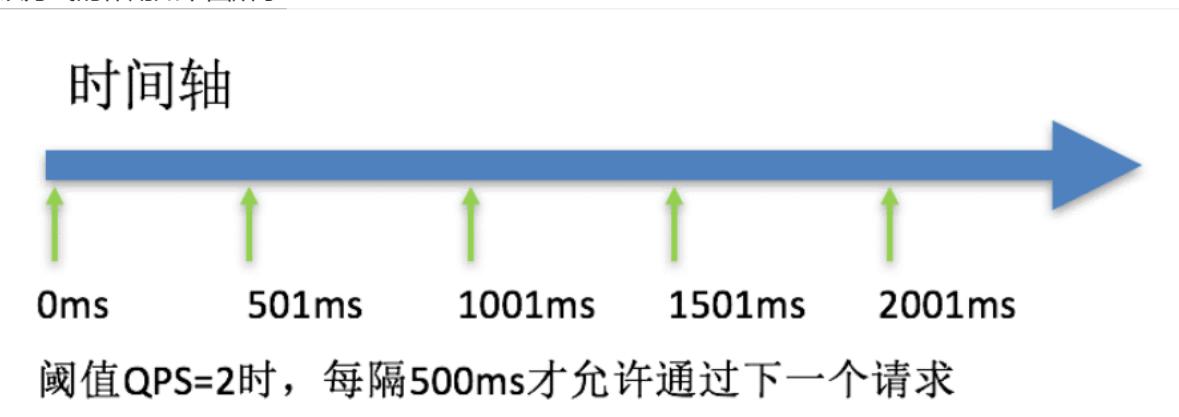
[关闭高级选项](#)

[保存](#) [取消](#)

匀速排队 [link](#)

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

该方式的作用如下图所示：



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

注意：匀速排队模式暂时不支持 QPS > 1000 的场景。

源码 - com.alibaba.csp.sentinel.slots.block.flow.controller.RateLimiterController

测试

- 添加日志记录代码到FlowLimitController的testA方法

```
1 @Slf4j
2 @RestController
3 public class FlowLimitController {
4     @GetMapping("/testA")
5     public String testA(){
6         log.info(Thread.currentThread().getName()+"\t"+"...testA");//<-----
7         return "-----testA";
8     }
9
10    ...
11 }
```

ApiPost模拟密集访问testA

后台结果

```
2022-07-20 18:58:19.120 INFO 20160 --- [nio-8401-exec-2] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-2 ...testA
2022-07-20 18:58:20.121 INFO 20160 --- [nio-8401-exec-4] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-4 ...testA
2022-07-20 18:58:21.120 INFO 20160 --- [nio-8401-exec-6] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-6 ...testA
2022-07-20 18:58:22.121 INFO 20160 --- [nio-8401-exec-8] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-8 ...testA
2022-07-20 18:58:23.120 INFO 20160 --- [io-8401-exec-10] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-10 ...testA
2022-07-20 18:58:24.120 INFO 20160 --- [nio-8401-exec-2] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-2 ...testA
2022-07-20 18:58:25.121 INFO 20160 --- [nio-8401-exec-4] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-4 ...testA
2022-07-20 18:58:26.121 INFO 20160 --- [nio-8401-exec-6] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-6 ...testA
2022-07-20 18:58:27.122 INFO 20160 --- [nio-8401-exec-8] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-8 ...testA
2022-07-20 18:58:28.121 INFO 20160 --- [io-8401-exec-10] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-10 ...testA
2022-07-20 18:58:29.121 INFO 20160 --- [nio-8401-exec-2] c.e.s.a.controller.FlowLimitController : http-nio-8401-exec-2 ...testA
```

降级规则

基本介绍

[官方文档](#)

熔断降级概述

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。一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。例如，支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。然而，这个被依赖服务的稳定性是不能保证的。如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。

现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此我们需要对不稳定的弱依赖服务调用进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。熔断降级作为保护自身的手段，通常在客户端（调用端）进行配置。

[link](#)



- RT (平均响应时间, 秒级)
 - 平均响应时间 超出阈值 且 在时间窗口内通过的请求 ≥ 5 , 两个条件同时满足后触发降级。
 - 窗口期过后关闭断路器。
 - RT最大4900 (更大的需要通过-Dcsp.sentinel.statistic.max.rt=XXXX才能生效)。
- 异常比例 (秒级)
 - QPS ≥ 5 且异常比例 (秒级统计) 超过阈值时, 触发降级;时间窗口结束后, 关闭降级。
- 异常数(分钟级)
 - 异常数(分钟统计) 超过阈值时, 触发降级;时间窗口结束后, 关闭降级

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的断路器是没有类似Hystrix半开状态的。(Sentinel 1.8.0 已有半开状态)

半开的状态系统自动去检测是否请求有异常, 没有异常就关闭断路器恢复使用, 有异常则继续打开断路器不可用。

降级策略

RT

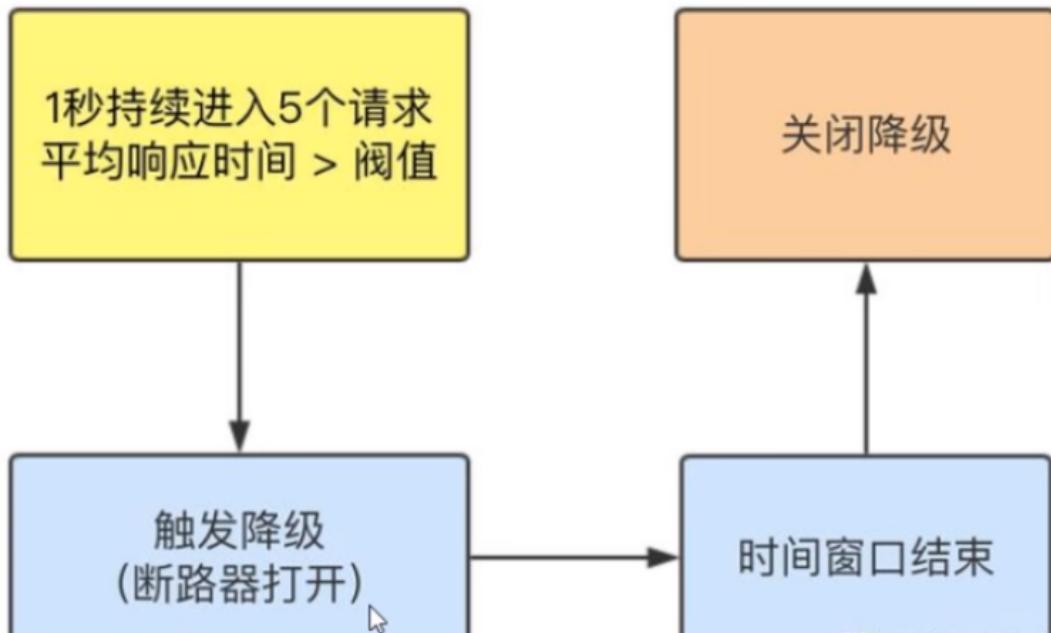
是什么?

平均响应时间(DEGRADE_GRADE_RT): 当1s内持续进入5个请求, 对应时刻的平均响应时间 (秒级) 均超过阈值 (count, 以ms为单位), 那么在接下的时间窗口 (DegradeRule 中的 timewindow, 以s为单位) 之内, 对这个方法的调用都会自动地熔断(抛出 DegradeException)。注意Sentinel 默认统计的RT上限是4900 ms, 超出此阈值的都会算作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。

注意: Sentinel 1.7.0才有平均响应时间 (DEGRADE_GRADE_RT), Sentinel 1.8.0的没有这项, 取而代之的是慢调用比例 (SLOW_REQUEST_RATIO)。

慢调用比例 (SLOW_REQUEST_RATIO): 选择以慢调用比例作为阈值, 需要设置允许的慢调用 RT (即最大的响应时间), 请求的响应时间大于该值则统计为慢调用。当单位统计时长 (statIntervalMs) 内请求数目大于设置的最小请求数目, 并且慢调用的比例大于阈值, 则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态 (HALF-OPEN 状态)。

态)，若接下来的一个请求响应时间小于设置的慢调用 RT 则结束熔断，若大于设置的慢调用 RT 则会再次被熔断。[link](#)



测试

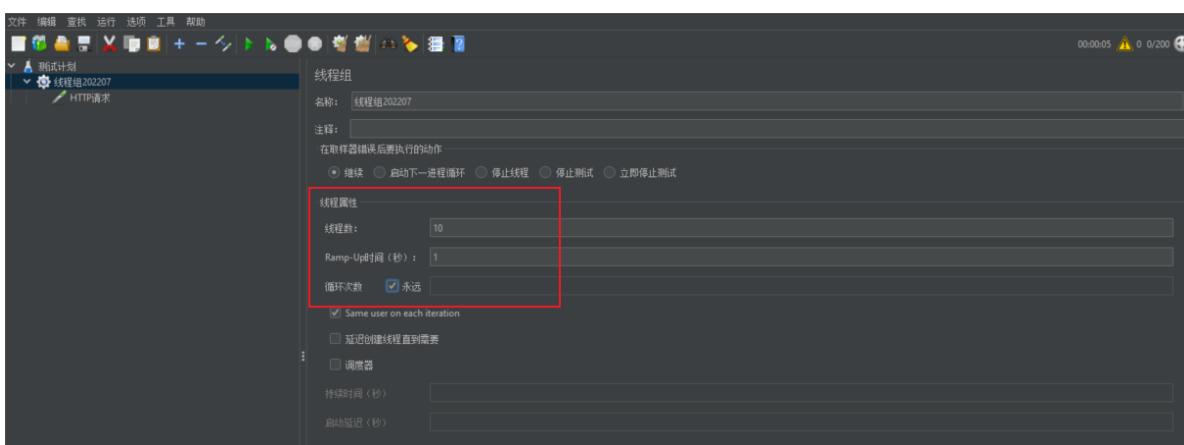
代码

```
1 @Slf4j
2 @RestController
3 public class FlowLimitController {
4     ...
5
6     @GetMapping("/testD")
7     public String testD() {
8         try {
9             TimeUnit.SECONDS.sleep(1);
10        } catch (InterruptedException e) {
11            e.printStackTrace();
12        }
13        log.info("testD 测试RT");
14        return "-----testD";
15    }
16}
```

配置



Jmeter压测



结论

按照上述配置，永远一秒钟打进来10个线程（大于5个了）调用testD，我们希望200毫秒处理完本次任务，如果超过200毫秒还没处理完，在未来1秒钟的时间窗口内，断路器打开（保险丝跳闸）微服务不可用，保险丝跳闸断电了后续我停止jmeter，没有这么大的访问量了，断路器关闭（保险丝恢复），微服务恢复OK。

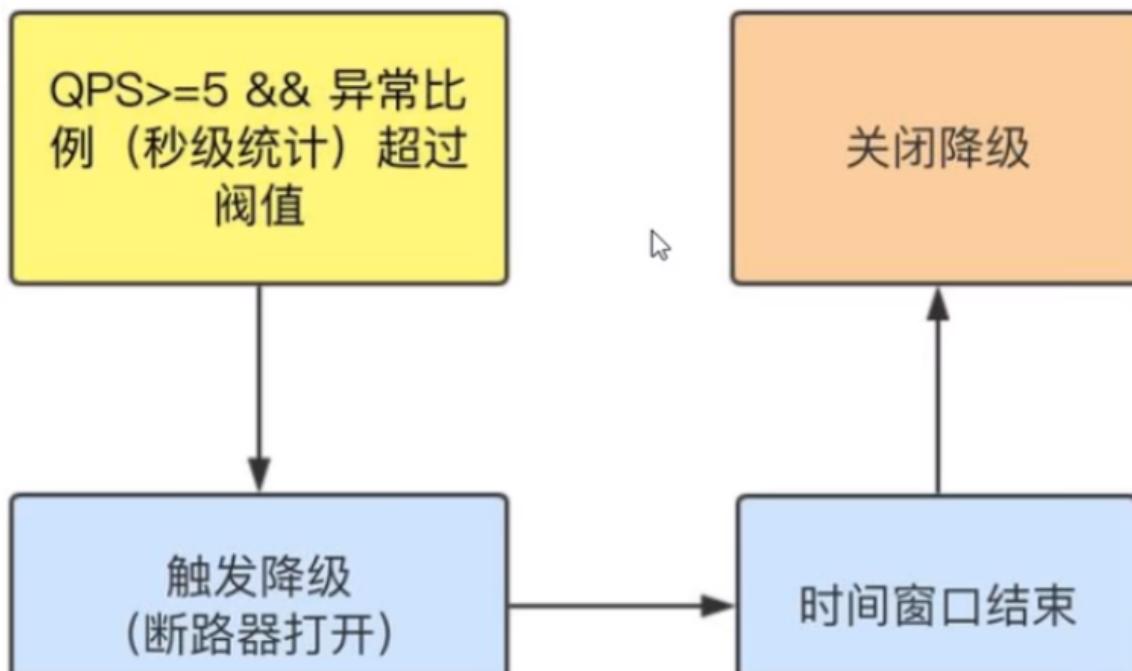
异常比例

是什么？

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

注意，与Sentinel 1.8.0相比，有些不同 (Sentinel 1.8.0才有的半开状态)，Sentinel 1.8.0的如下：

异常比例 (`ERROR_RATIO`)：当单位统计时长 (`statIntervalMs`) 内请求数目大于设置的最小请求数目，并且异常的比例大于阈值，则接下来的熔断时长内请求会自动被熔断。经过熔断时长后熔断器会进入探测恢复状态 (HALF-OPEN 状态)，若接下来的一个请求成功完成 (没有错误) 则结束熔断，否则会再次被熔断。异常比率的阈值范围是 $[0.0, 1.0]$ ，代表 0% - 100%。[link](#)



测试

代码

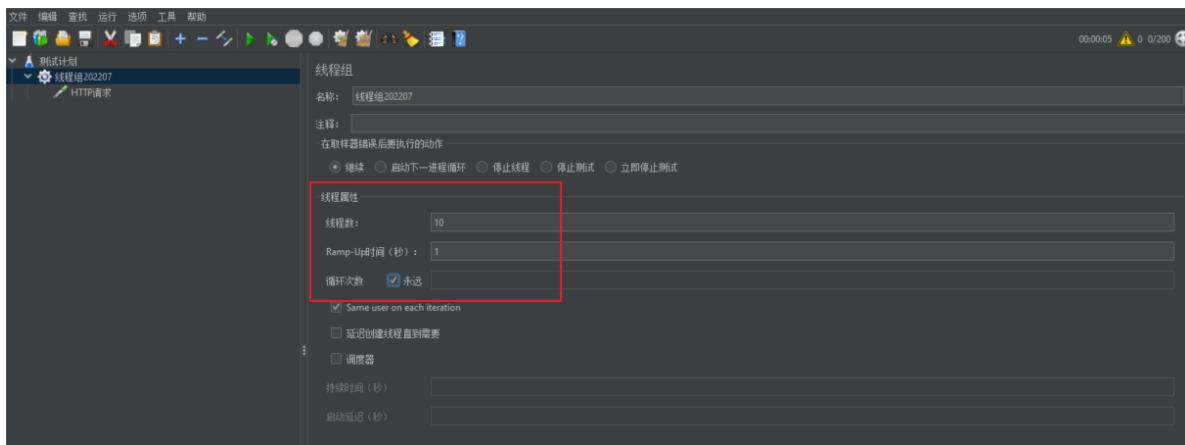
```

1  @Slf4j
2  @RestController
3  public class FlowLimitController {
4
5      ...
6
7      @GetMapping("/testE")
8      public String testE() {
9          log.info("testE 异常比例");
10         int age = 10/0;
11         return "-----testD";
12     }
13 }
  
```

配置

新增降级规则

资源名	/testE
降级策略	<input type="radio"/> RT <input checked="" type="radio"/> 异常比例 <input type="radio"/> 异常数
异常比例	0.2
时间窗口	3



结论

按照上述配置，单独访问一次，必然来一次报错一次(`int age = 10/0;`)，调一次错一次。

开启jmeter后，直接高并发发送请求，多次调用达到我们的配置条件了。断路器开启(保险丝跳闸)，微服务不可用了，不再报错error而是服务降级了。

异常数

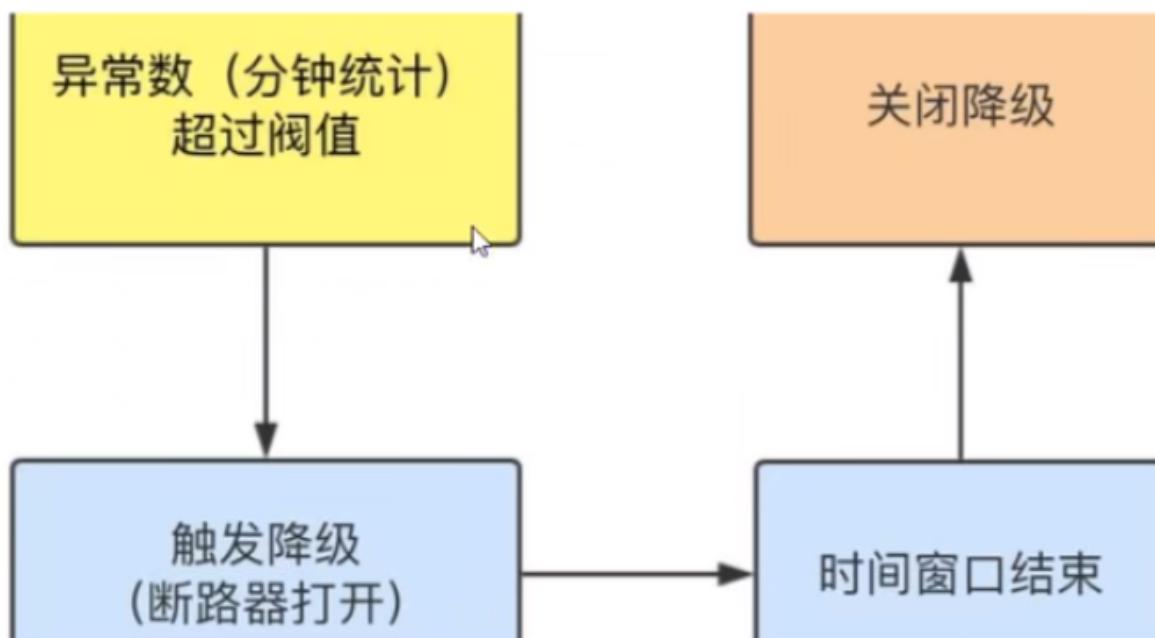
是什么？

异常数(`DEGRADE_GRADF_EXCEPTION_COUNT`)：当资源近1分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若`timewindow`小于60s，则结束熔断状态后码可能再进入熔断状态。

注意，与Sentinel 1.8.0相比，有些不同 (Sentinel 1.8.0才有的半开状态)，Sentinel 1.8.0的如下：

异常数 (`ERROR_COUNT`)：当单位统计时长内的异常数目超过阈值之后会自动进行熔断。经过熔断时长后熔断器会进入探测恢复状态 (HALF-OPEN 状态)，若接下来的一个请求成功完成 (没有错误) 则结束熔断，否则会再次被熔断。

异常数是按照分钟统计的，时间窗口一定要大于等于60秒。



测试

代码

```

1  @Slf4j
2  @RestController
3  public class FlowLimitController {
4
5      ...
6
7      @GetMapping("/testF")
8      public String testF() {
9          log.info("testF 异常数");
10         int age = 10/0;
11         return "-----testF 测试异常数";
12     }
13 }

```

配置



访问<http://localhost:8401/testF>, 第一次访问绝对报错, 因为除数不能为零, 我们看到error窗口, 但是达到5次报错后, 进入熔断后降级。

热点key限流

基本介绍



官网

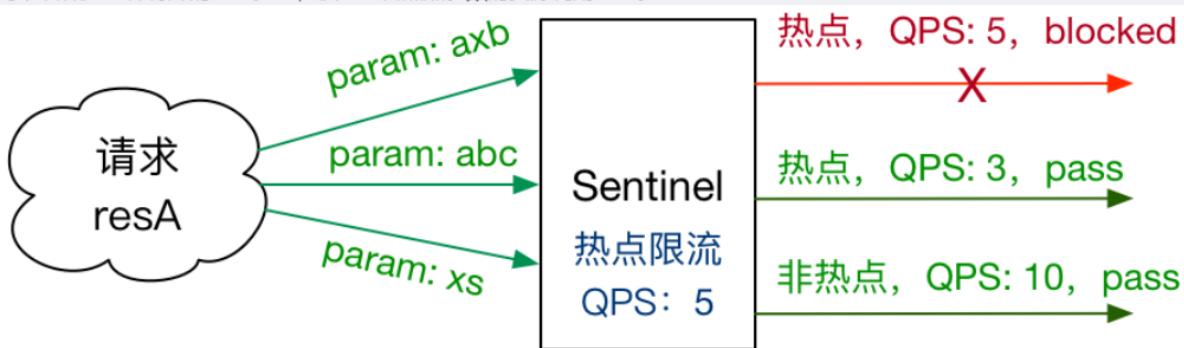
官方文档

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 K 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参数，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。



Sentinel 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。热点参数限流支持集群模式。

[link](#)

承上启下复习start

兜底方法，分为系统默认和客户自定义，两种

之前的case，限流出问题后，都是用sentinel系统默认的提示: Blocked by Sentinel (flow limiting)

我们能不能自定？类似hystrix，某个方法出问题了，就找对应的兜底降级方法？

结论 - 从 `HystrixCommand` 到 `@SentinelResource`

代码

`com.alibaba.csp.sentinel.slots.block.BlockException`

```

1  @Slf4j
2  @RestController
3  public class FlowLimitController{
4
5      ...
6
7      @GetMapping("/testHotKey")
8      @SentinelResource(value = "testHotKey",blockHandler/*兜底方法*/ =
9      "deal_testHotKey")
10     public String testHotKey(@RequestParam(value = "p1",required = false)
11                             String p1,
12                             @RequestParam(value = "p2",required = false)
13                             String p2) {
14         return "-----testHotKey";
15     }
16
17     /**兜底方法*/
18     public String deal_testHotKey (String p1, String p2, BlockException
19     exception) {
20         //sentinel系统默认的提示: Blocked by sentinel (flow limiting)
21         return "-----deal_testHotKey,o(╥﹏╥)o";
22     }
23 }

```

配置



- 一
- @SentinelResource(value = "testHotKey")
 - 异常(error page)打到了前台用户界面看到，不友好

- 二
- @SentinelResource(value = "testHotKey", blockHandler = "dealHandler_testHotKey")
 - 方法testHotKey里面第一个参数只要QPS超过每秒1次，马上降级处理
 - 异常用了我们自己定义的兜底方法

测试

- error
 - <http://localhost:8401/testHotKey?p1=abc>
 - <http://localhost:8401/testHotKey?p1=abc&p2=33>
- right
 - <http://localhost:8401/testHotKey?p2=abc>

上面的例子演示了第一个参数p1，当QPS超过1秒1次点击后马上被限流。

参数例外项

- 普通 - 超过1秒钟一个后，达到阈值1后马上被限流
- 我们期望p1参数当它是某个特殊值时，它的限流值和平时不一样
- 特例 - 假如当p1的值等于5时，它的阈值可以达到200

配置

编辑热点规则

资源名	testHotKey		
限流模式	QPS 模式		
参数索引	0		
单机阈值	1	统计窗口时长	1 秒
是否集群	<input type="checkbox"/>		
参数例外项			
参数类型	<input type="button" value="例外项参数值"/> <input type="button" value="限流阈值"/> <input type="button" value="添加"/>		
参数值	参数类型	限流阈值	操作
5	java.lang.String	200	<input type="button" value="删除"/>
关闭高级选项			
<input type="button" value="保存"/> <input type="button" value="取消"/>			

测试

- right - <http://localhost:8401/testHotKey?p1=5>
- error - <http://localhost:8401/testHotKey?p1=3>
- 当p1等于5的时候，阈值变为200
- 当p1不等于5的时候，阈值就是平常的1

前提条件 - 热点参数的注意点，参数必须是基本类型或者String

其它

在方法体抛异常

```

1  @Slf4j
2  @RestController
3  public class FlowLimitController{
4      ...
5
6      @GetMapping("/testHotKey")
7      @SentinelResource(value = "testHotKey",blockHandler/*兜底方法*/ =
"deal_testHotKey")

```

```

8     public String testHotKey(@RequestParam(value = "p1", required = false)
9         String p1,
10            @RequestParam(value = "p2", required = false)
11        String p2) {
12            int age = 10 / 0;
13            return "-----testHotKey";
14        }
15
16    /**兜底方法*/
17    public String deal_testHotKey (String p1, String p2, BlockException
18 exception) {
19        //sentinel系统默认的提示: Blocked by sentinel (flow limiting)
20        return "-----deal_testHotKey,o(╥﹏╥)o";
21    }
22 }
```

将会抛出Spring Boot 2的默认异常页面，而不是兜底方法。

- `@SentinelResource` - 处理的是sentinel控制台配置的违规情况，有`blockHandler`方法配置的兜底处理；
- `RuntimeException` `int age = 10/0`，这个是java运行时报出的运行时异常`RunTimeException`，`@SentinelResource`不管

总结 - `@SentinelResource`主管配置出错，运行出错该走异常走异常

系统规则

官方文档

Sentinel 系统自适应限流从整体维度对**应用入口流量**进行控制，结合应用的 Load、CPU 使用率、总体平均 RT、入口 QPS 和并发线程数等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。[link](#)

系统规则

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 load、CPU 使用率、平均 RT、入口 QPS 和并发线程数等几个维度监控应用指标，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统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 服务或 Dubbo 服务端接收的请求，都属于入口流量。

系统规则支持以下的模式：

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 `load1` 作为启发指标，进行自适应系统保护。当系统 `load1` 超过设定的启发值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护（BBR 阶段）。系统容量由系统的 `maxQps * minRT` 估算得出。设定参考值一般是 `CPU cores * 2.5`。
- **CPU usage**（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0），比较灵敏。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。

[link](#)

配置

资源名称限流 + 后续处理

启动Nacos成功

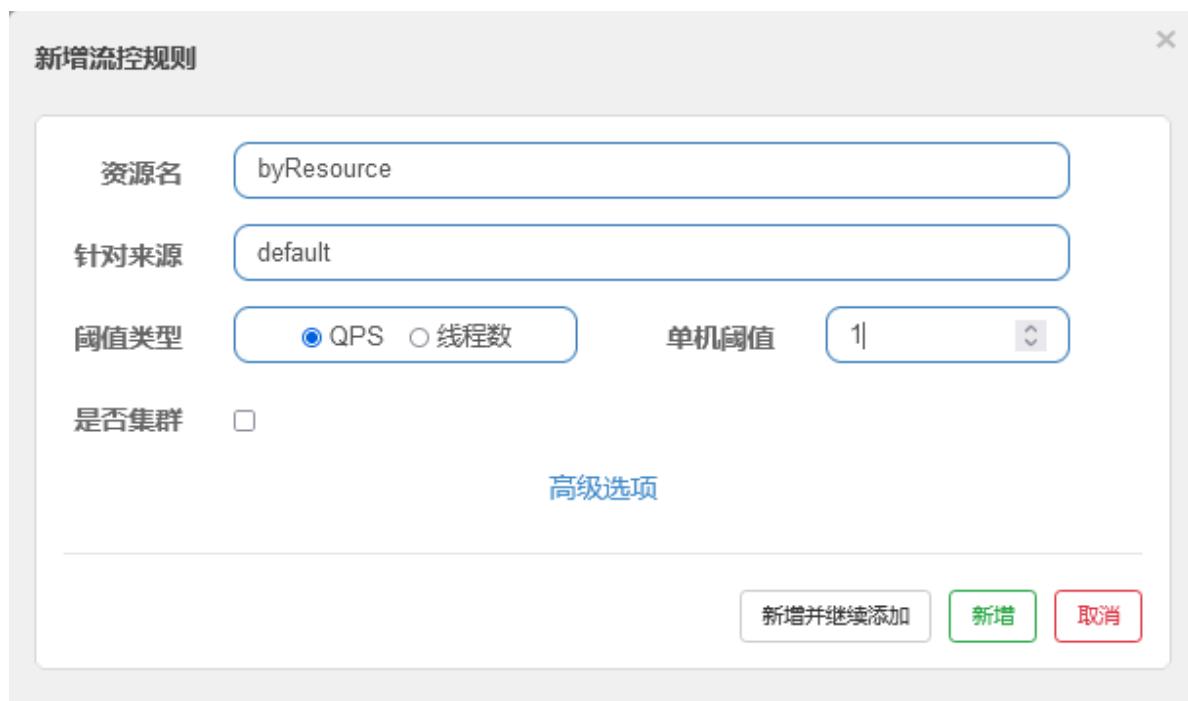
启动Sentinel成功

Module - cloudalibaba-sentinel-service8401

```
1 import com.alibaba.csp.sentinel.annotation.SentinelResource;
2 import com.alibaba.csp.sentinel.slots.block.BlockException;
3 import com.atguigu.springcloud.alibaba.myhandler.CustomerBlockHandler;
4 import com.atguigu.springcloud.entities.CommonResult;
5 import com.atguigu.springcloud.entities.Payment;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 @RestController
10 public class RateLimitController {
11
12     @GetMapping("/byResource")
13     @SentinelResource(value = "byResource",blockHandler = "handleException")
14     public CommonResult byResource() {
15         return new CommonResult(200,"按资源名称限流测试OK",new
16 Payment(2020L,"serial001"));
17     }
18
19     public CommonResult handleException(BlockException exception) {
20         return new
21 CommonResult(444,exception.getClass().getCanonicalName()+"\t 服务不可用");
22     }
23 }
```

配置流控规则

配置步骤



图形配置和代码关系

表示1秒钟内查询次数大于1，就跑到我们自定义的处流，限流

测试

1秒钟点击1下，OK

超过上述，疯狂点击，返回了自己定义的限流处理信息，限流发生

```
1 | {"code":444,
  "message": "com.alibaba.csp.sentinel.slots.block.flow.FlowException\\t 服务不可用", "data":null}
```

额外问题

此时关闭问服务8401 -> Sentinel控制台，流控规则消失了

按照Url地址限流 + 后续处理

通过访问的URL来限流+后续处理会返回Sentinel自带默认的限流处理信息

业务类RateLimitController

```
1 | @RestController
2 | public class RateLimitController{
3 |     ...
4 |
5 |     @GetMapping("/rateLimit/byUrl")
6 |     @SentinelResource(value = "byUrl")
7 |     public CommonResult byUrl(){
8 |         return new CommonResult(200,"按url限流测试OK",new
9 |             Payment(2020L,"serial002"));
10 |     }
10 | }
```

Sentinel控制台配置



测试

- 快速点击<http://localhost:8401/rateLimit/byUrl>
- 结果 - 会返回Sentinel自带的限流处理结果 Blocked by Sentinel (flow limiting)

上面兜底方案面临的问题

- 系统默认的，没有体现我们自己的业务要求。
- 依照现有条件，我们自定义的处理方法又和业务代码耦合在一块，不直观。
- 每个业务方法都添加一个兜底的，那代码膨胀加剧。
- 全局统一的处理方法没有体现。

客户自定义限流处理逻辑

客户自定义限流处理逻辑

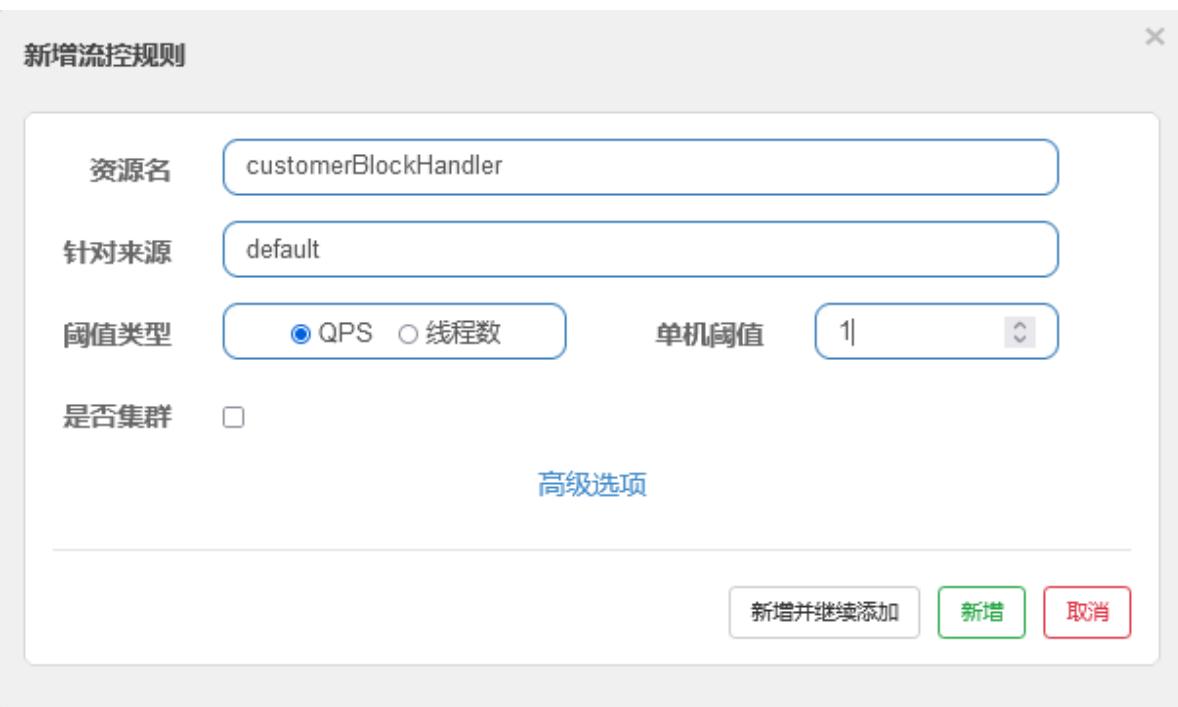
自定义限流处理类 - 创建 `CustomerBlockHandler` 类用于自定义限流处理逻辑

```
1 import com.alibaba.csp.sentinel.slots.block.BlockException;
2 import com.atguigu.springcloud.entities.CommonResult;
3 import com.atguigu.springcloud.entities.Payment;
4
5 public class CustomerBlockHandler {
6     public static CommonResult handlerException(BlockException exception) {
7         return new CommonResult(4444,"按客户自定义,global handlerException--1");
8     }
9
10    public static CommonResult handlerException2(BlockException exception) {
11        return new CommonResult(4444,"按客户自定义,global handlerException--2");
12    }
13 }
```

RateLimitController

```
1 @RestController
2 public class RateLimitController {
3     ...
4
5     @GetMapping("/rateLimit/customerBlockHandler")
6     @SentinelResource(value = "customerBlockHandler",
7         blockHandlerClass = CustomerBlockHandler.class, //----- 自定义
8         blockHandler = "handlerException2") //-----
9     public CommonResult customerBlockHandler(){
10         return new CommonResult(200,"按客户自定义",new
11             Payment(2020L,"serial003"));
12     }
13 }
```

Sentinel控制台配置



启动微服务后先调用一次 - <http://localhost:8401/rateLimit/customerBlockHandler>。然后，多次快速刷新<http://localhost:8401/rateLimit/customerBlockHandler>。刷新后，我们自定义兜底方法的字符串信息就返回到前端。

更多注解属性说明

@SentinelResource 注解

注意：注解方式埋点不支持 private 方法。

`@SentinelResource` 用于定义资源，并提供可选的异常处理和 fallback 配置项。

`@SentinelResource` 注解包含以下属性：

- `value`：资源名称，必需项（不能为空）
- `entryType`：entry 类型，可选项（默认为 `EntryType.OUT`）
- `blockHandler / blockHandlerClass`: `blockHandler` 对应处理 `BlockException` 的函数名称，可选项。`blockHandler` 函数访问范围需要是 `public`，返回类型需要与原方法相匹配，参数类型需要和原方法相匹配并且最后加一个额外的参数，类型为 `BlockException`。`blockHandler` 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 `blockHandlerClass` 为对应的类的 Class 对象，注意对应的函数必需为 `static` 函数，否则无法解析。
- `fallback / fallbackClass`: `fallback` 函数名称，可选项，用于在抛出异常的时候提供 fallback 处理逻辑。`fallback` 函数可以针对所有类型的异常（除了 `exceptionsToIgnore` 里面排除掉的异常类型）进行处理。`fallback` 函数签名和位置要求：
 - 返回值类型必须与原函数返回值类型一致；
 - 方法参数列表需要和原函数一致，或者可以额外多一个 `Throwable` 类型的参数用于接收对应的异常。
 - `fallback` 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 `fallbackClass` 为对应的类的 `Class` 对象，注意对应的函数必需为 `static` 函数，否则无法解析。
- `defaultFallback` (since 1.6.0) : 默认的 `fallback` 函数名称，可选项，通常用于通用的 `fallback` 逻辑（即可以用于很多服务或方法）。默认 `fallback` 函数可以针对所有类型的异常（除了 `exceptionsToIgnore` 里面排除掉的异常类型）进行处理。若同时配置了 `fallback` 和 `defaultFallback`，则只有 `fallback` 会生效。`defaultFallback` 函数签名要求：

- 返回值类型必须与原函数返回值类型一致；
- 方法参数列表需要为空，或者可以额外多一个 `Throwable` 类型的参数用于接收对应的异常。
- `defaultFallback` 函数默认需要和原方法在同一个类中。若希望使用其他类的函数，则可以指定 `fallbackClass` 为对应的类的 `class` 对象，注意对应的函数必需为 `static` 函数，否则无法解析。
- `exceptionsToIgnore` (since 1.6.0) : 用于指定哪些异常被排除掉，不会计入异常统计中，也不会进入 fallback 逻辑中，而是会原样抛出。

[link](#)

Sentinel主要有三个核心Api:

1. SphU定义资源
2. Tracer定义统计
3. ContextUtil定义了上下文

服务熔断

sentinel整合ribbon+openFeign+fallback

Ribbon系列

- 启动nacos和sentinel
- 提供者9003/9004
- 消费者84

提供者9003/9004

新建Module

名字: cloud-alibaba-provider-payment9003/9004, 两个一样的做法

POM

```

1 <dependencies>
2     <!--SpringCloud alibaba nacos -->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
6     </dependency>
7     <dependency><!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
8         <groupId>org.example</groupId>
9         <artifactId>cloud-api-commons</artifactId>
10        <version>${project.version}</version>
11    </dependency>
12    <!-- SpringBoot整合Web组件 -->
13    <dependency>
14        <groupId>org.springframework.boot</groupId>
15        <artifactId>spring-boot-starter-web</artifactId>
16    </dependency>
17    <dependency>
18        <groupId>org.springframework.boot</groupId>
19        <artifactId>spring-boot-starter-actuator</artifactId>
20    </dependency>
21    <!--日常通用jar包配置-->
22    <dependency>
```

```
13         <groupId>org.springframework.boot</groupId>
14         <artifactId>spring-boot-devtools</artifactId>
15         <scope>runtime</scope>
16         <optional>true</optional>
17     </dependency>
18     <dependency>
19         <groupId>org.projectlombok</groupId>
20         <artifactId>lombok</artifactId>
21         <optional>true</optional>
22     </dependency>
23     <dependency>
24         <groupId>org.springframework.boot</groupId>
25         <artifactId>spring-boot-starter-test</artifactId>
26         <scope>test</scope>
27     </dependency>
28   </dependencies>
```

YML

```
1 server:
2   port: 9003
3
4 spring:
5   application:
6     name: nacos-payment-provider
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #配置Nacos地址
11
12 management:
13   endpoints:
14     web:
15       exposure:
16         include: '*'
```

记得修改不同的端口号

主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
4
5 @SpringBootApplication
6 @EnableDiscoveryClient
7 public class PaymentMain9003 {
8     public static void main(String[] args) {
9         SpringApplication.run(PaymentMain9003.class, args);
10    }
11 }
```

业务类

```
1 import com.atguigu.springcloud.entities.CommonResult;
2 import com.atguigu.springcloud.entities.Payment;
```

```

3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import java.util.HashMap;
9
10 @RestController
11 public class PaymentController {
12     @Value("${server.port}")
13     private String serverPort;
14
15     //模拟数据库
16     public static HashMap<Long,Payment> hashMap = new HashMap<>();
17     static
18     {
19         hashMap.put(1L,new Payment(1L,"28a8c1e3bc2742d8848569891fb42181"));
20         hashMap.put(2L,new Payment(2L,"bba8c1e3bc2742d8848569891ac32182"));
21         hashMap.put(3L,new Payment(3L,"6ua8c1e3bc2742d8848569891xt92183"));
22     }
23
24     @GetMapping(value = "/paymentSQL/{id}")
25     public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id){
26         Payment payment = hashMap.get(id);
27         CommonResult<Payment> result = new CommonResult(200,"from
mysql,serverPort: "+serverPort,payment);
28         return result;
29     }
30
31 }

```

测试地址 - <http://localhost:9003/paymentSQL/1>

消费者84

新建Module

名字: cloud-alibaba-consumer-nacos-order84

POM

```

1 <dependencies>
2     <!--SpringCloud openfeign -->
3     <!--
4     <dependency>
5         <groupId>org.springframework.cloud</groupId>
6         <artifactId>spring-cloud-starter-openfeign</artifactId>
7     </dependency>
8     <!--
9     <!--SpringCloud alibaba nacos -->
10    <dependency>
11        <groupId>com.alibaba.cloud</groupId>
12        <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
13    </dependency>
14    <!--SpringCloud alibaba sentinel -->
15    <dependency>
16        <groupId>com.alibaba.cloud</groupId>

```

```

17      <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
18  </dependency>
19  <!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
20  <dependency>
21      <groupId>org.example</groupId>
22      <artifactId>cloud-api-commons</artifactId>
23      <version>${project.version}</version>
24  </dependency>
25  <!-- SpringBoot整合Web组件 -->
26  <dependency>
27      <groupId>org.springframework.boot</groupId>
28      <artifactId>spring-boot-starter-web</artifactId>
29  </dependency>
30  <dependency>
31      <groupId>org.springframework.boot</groupId>
32      <artifactId>spring-boot-starter-actuator</artifactId>
33  </dependency>
34  <!--日常通用jar包配置-->
35  <dependency>
36      <groupId>org.springframework.boot</groupId>
37      <artifactId>spring-boot-devtools</artifactId>
38      <scope>runtime</scope>
39      <optional>true</optional>
40  </dependency>
41  <dependency>
42      <groupId>org.projectlombok</groupId>
43      <artifactId>lombok</artifactId>
44      <optional>true</optional>
45  </dependency>
46  <dependency>
47      <groupId>org.springframework.boot</groupId>
48      <artifactId>spring-boot-starter-test</artifactId>
49      <scope>test</scope>
50  </dependency>
51 </dependencies>

```

YML

```

1 server:
2   port: 84
3
4 spring:
5   application:
6     name: nacos-order-consumer
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848
11   sentinel:
12     transport:
13       #配置Sentinel dashboard地址
14       dashboard: localhost:8080
15       #默认8719端口，假如被占用会自动从8719开始依次+1扫描，直至找到未被占用的端口
16       port: 8719
17
18   #消费者将要去访问的微服务名称(注册成功进nacos的微服务提供者)
19   service-url:

```

```
20 nacos-user-service: http://nacos-payment-provider
21
22 # 激活Sentinel对Feign的支持
23 feign:
24   sentinel:
25     enabled: false
```

主启动

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 //import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
4 import org.springframework.cloud.openfeign.EnableFeignClients;
5
6 @EnableDiscoveryClient
7 @SpringBootApplication
8 //@EnableFeignClients
9 public class OrderNacosMain84 {
10   public static void main(String[] args) {
11     SpringApplication.run(OrderNacosMain84.class, args);
12   }
13 }
```

业务类

ApplicationContextConfig

```
1 import org.springframework.cloud.client.loadbalancer.LoadBalanced;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.web.client.RestTemplate;
5
6 @Configuration
7 public class ApplicationContextConfig {
8
9   @Bean
10  @LoadBalanced
11  public RestTemplate getRestTemplate() {
12    return new RestTemplate();
13  }
14 }
```

CircleBreakerController

```
1 import com.alibaba.csp.sentinel.annotation.SentinelResource;
2 import com.alibaba.csp.sentinel.slots.block.BlockException;
3 import com.atguigu.springcloud.alibaba.service.PaymentService;
4 import com.atguigu.springcloud.entities.CommonResult;
5 import com.atguigu.springcloud.entities.Payment;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11 import org.springframework.web.client.RestTemplate;
```

```

13 import javax.annotation.Resource;
14
15 @Slf4j
16 @RestController
17 @RequestMapping("/consumer")
18 public class CircleBreakerController {
19     public static final String SERVICE_URL = "http://nacos-payment-
provider";
20
21     @Resource
22     private RestTemplate restTemplate;
23
24     @RequestMapping("/fallback/{id}")
25     @SentinelResource(value = "fallback")//没有配置
26     public CommonResult<Payment> fallback(@PathVariable Long id){
27         CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
+ "/paymentsSQL/" + id, CommonResult.class, id);
28
29         if (id == 4) {
30             throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
31         } else if (result.getData() == null) {
32             throw new NullPointerException ("NullPointerException,该ID没有对应记录,空指针异常");
33         }
34
35         return result;
36     }
37 }

```

修改后请重启微服务

- 热部署对java代码级生效及时
- 对@SentinelResource注解内属性，有时效果不好

目的

- fallback管运行异常
- blockHandler管配置违规

测试地址 - <http://localhost:84/consumer/fallback/1>

没有任何配置

只配置fallback

只配置blockHandler

fallback和blockHandler都配置

忽略属性

无配置

没有任何配置 - 给用户error页面，不友好

```

1 @Slf4j
2 @RestController
3 public class CircleBreakerController {

```

```

4     public static final String SERVICE_URL = "http://nacos-payment-
5     provider";
6
7     @Resource
8     private RestTemplate restTemplate;
9
10    @RequestMapping("/consumer/fallback/{id}")
11    @SentinelResource(value = "fallback")//没有配置
12    public CommonResult<Payment> fallback(@PathVariable Long id){
13        CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
14        + "/paymentsSQL/" + id, CommonResult.class, id);
15
16        if (id == 4) {
17            throw new IllegalArgumentException ("IllegalArgumentException,非
18            法参数异常....");
19        } else if (result.getData() == null) {
20            throw new NullPointerException ("NullPointerException,该ID没有对应
21            记录,空指针异常");
22        }
23    }

```

只配置fallback

fallback只负责业务异常

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class CircleBreakerController {
5
6     public static final String SERVICE_URL = "http://nacos-payment-
7     provider";
8
9     @Resource
10    private RestTemplate restTemplate;
11
12    @RequestMapping("/fallback/{id}")
13    //@SentinelResource(value = "fallback")//没有配置
14    @SentinelResource(value = "fallback", fallback = "handlerFallback")
15    //fallback只负责业务异常
16    public CommonResult<Payment> fallback(@PathVariable Long id) {
17        CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
18        + "/paymentsSQL/" + id, CommonResult.class, id);
19
20        if (id == 4) {
21            throw new IllegalArgumentException ("IllegalArgumentException,非
22            法参数异常....");
23        } else if (result.getData() == null) {
24            throw new NullPointerException ("NullPointerException,该ID没有对应
25            记录,空指针异常");
26        }
27
28    }
29
30    return result;

```

```

24     }
25
26     //本例是fallback
27     public CommonResult handlerFallback(@PathVariable Long id, Throwable e)
28     {
29         Payment payment = new Payment(id,"null");
30         return new CommonResult<>(444,"兜底异常handlerFallback,exception内容
31         "+e.getMessage(),payment);
32     }

```

测试地址 - <http://localhost:84/consumer/fallback/4>

页面返回结果：

```

1 {"code":444,"message":"兜底异常handlerFallback, exception内容
illegalArgumentEBxception,非法参数异常.....","data":{"id":4,"seria:"null"}}

```

只配置blockHandler

blockHandler只负责sentinel控制台配置违规

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class CircleBreakerController{
5     public static final String SERVICE_URL = "http://nacos-payment-
provider";
6
7     @Resource
8     private RestTemplate restTemplate;
9
10    @RequestMapping("/fallback/{id}")
11    //@SentinelResource(value = "fallback") //没有配置
12    //@SentinelResource(value = "fallback",fallback = "handlerFallback")
13    //fallback只负责业务异常
14    @SentinelResource(value = "fallback",blockHandler = "blockHandler")
15    //blockHandler只负责sentinel控制台配置违规
16    public CommonResult<Payment> fallback(@PathVariable Long id)
17    {
18        CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
19        + "/paymentsQL/" + id,CommonResult.class,id);
20
21        if (id == 4) {
22            throw new IllegalArgumentException ("IllegalArgumentException,非
23            法参数异常....");
24        }else if (result.getData() == null) {
25            throw new NullPointerException ("NullPointerException,该ID没有对应
26            记录,空指针异常");
27        }
28
29        return result;
30    }
31    //本例是fallback
32    /* public CommonResult handlerFallback(@PathVariable Long id,Throwable
e) {

```

```

28         Payment payment = new Payment(id,"null");
29         return new CommonResult<>(444,"兜底异常handlerFallback,exception内容
30 "+e.getMessage(),payment);
31     }*/
32
33     //本例是blockHandler
34     public CommonResult blockHandler(@PathVariable Long id,BlockException
35 blockException) {
36         Payment payment = new Payment(id,"null");
37         return new CommonResult<>(445,"blockHandler-sentinel限流,无此流水:
38 blockException "+blockException.getMessage(),payment);
39     }
40 }
```

sentinel 配置



测试地址 - <http://localhost:84/consumer/fallback/4>

第一次直接error page。

后面属于sentinel的配置出错了。直接。fallback。

fallback和blockHandler都配置

若blockHandler和fallback 都进行了配置，则被限流降级而抛出BlockException时只会进入blockHandler处理逻辑。

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class CircleBreakerController{
5     public static final String SERVICE_URL = "http://nacos-payment-
6 provider";
7
8     @Resource
9     private RestTemplate restTemplate;
10
11    @RequestMapping("/fallback/{id}")
12    //@SentinelResource(value = "fallback") //没有配置
13    //@SentinelResource(value = "fallback",fallback = "handlerFallback")
14    //fallback只负责业务异常
```

```

13     //@SentinelResource(value = "fallback",blockHandler = "blockHandler")
//blockHandler只负责sentinel控制台配置违规
14     @SentinelResource(value = "fallback",fallback =
"handlerFallback",blockHandler = "blockHandler")
15     public CommonResult<Payment> fallback(@PathVariable Long id){
16         CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
+ "/paymentsSQL/" + id,CommonResult.class,id);
17
18         if (id == 4) {
19             throw new IllegalArgumentException ("IllegalArgumentException,非
法参数异常....");
20         }else if (result.getData() == null) {
21             throw new NullPointerException ("NullPointerException,该ID没有对应
记录,空指针异常");
22         }
23
24         return result;
25     }
26     //本例是fallback
27     public CommonResult handlerFallback(@PathVariable Long id,Throwable e)
{
28         Payment payment = new Payment(id,"null");
29         return new CommonResult<>(444,"兜底异常handlerFallback,exception内容
"+e.getMessage(),payment);
30     }
31     //本例是blockHandler
32     public CommonResult blockHandler(@PathVariable Long id,BlockException
blockException) {
33         Payment payment = new Payment(id,"null");
34         return new CommonResult<>(445,"blockHandler-sentinel限流,无此流水:
blockException "+blockException.getMessage(),payment);
35     }
36 }
```

exceptionsToIgnore

exceptionsToIgnore, 忽略指定异常, 即这些异常不用兜底方法处理。

```

1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class CircleBreakerController
5
6     ...
7
8     @RequestMapping("/fallback/{id}")
9     @SentinelResource(value = "fallback",fallback =
"handlerFallback",blockHandler = "blockHandler",
10         exceptionsToIgnore = {IllegalArgumentException.class})//<-----
----->
11     public CommonResult<Payment> fallback(@PathVariable Long id){
12         CommonResult<Payment> result = restTemplate.getForObject(SERVICE_URL
+ "/paymentsSQL/" + id,CommonResult.class,id);
13
14         if (id == 4) {
15             //exceptionsToIgnore属性有IllegalArgumentException.class,
16             //所以IllegalArgumentException不会跳入指定的兜底程序。
17         }
18     }
19 }
```

```
17         throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");
18     }else if (result.getData() == null) {
19         throw new NullPointerException ("NullPointerException,该ID没有对应记录,空指针异常");
20     }
21
22     return result;
23 }
24
25 ...
26 }
```

Feign系列

修改84模块

- 84消费者调用提供者9003
- Feign组件一般是消费侧

POM

```
1 <!--SpringCloud openfeign -->
2 <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-openfeign</artifactId>
5 </dependency>
```

YML

```
1 # 激活Sentinel对Feign的支持
2 feign:
3     sentinel:
4         enabled: true
```

业务类

带@Feignclient注解的业务接口， fallback = PaymentFallbackService.class

```
1 import com.atguigu.springcloud.entities.CommonResult;
2 import com.atguigu.springcloud.entities.Payment;
3 import org.springframework.cloud.openfeign.FeignClient;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.PathVariable;
6
7 @FeignClient(value = "nacos-payment-provider",fallback =
8 PaymentFallbackService.class)
9 public interface PaymentService{
10     @GetMapping(value = "/paymentsSQL/{id}")
11     CommonResult<Payment> paymentsSQL(@PathVariable("id") Long id);
12 }
```

```
1 import com.atguigu.springcloud.entities.CommonResult;
2 import com.atguigu.springcloud.entities.Payment;
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class PaymentFallbackService implements PaymentService {
7     @Override
8     public CommonResult<Payment> paymentSQL(Long id){
9         return new CommonResult<>(4444,"服务降级返回,--"
10            + "PaymentFallbackService",new Payment(id,"errorSerial"));
11    }
12 }
```

Controller

```
1 @Slf4j
2 @RestController
3 @RequestMapping("/consumer")
4 public class CircleBreakerController {
5
6     ...
7
8     //=====OpenFeign
9     @Resource
10    private PaymentService paymentService;
11
12    @GetMapping(value = "/paymentSQL/{id}")
13    public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id){
14        return paymentService.paymentSQL(id);
15    }
16 }
```

主启动

```
1 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
2
3 @EnabledDiscoveryClient
4 @SpringBootApplication
5 @EnableFeignClients//<-----
6 public class OrderNacosMain84 {
7     public static void main(String[] args) {
8         SpringApplication.run(OrderNacosMain84.class, args);
9     }
10 }
```

测试 - <http://localhost:84/consumer/paymentSQL/1>

测试84调用9003，此时故意关闭9003微服务提供者，**84消费侧自动降级**，不会被耗死。

熔断框架比较

-	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离（并发线程数限流）	线程池隔商/ 信号量隔离	信号量隔离
熔断降级策略	基于响应时间、异常比率、异常数	基于异常比率	基于异常比率、响应时间
实时统计实现	滑动窗口（LeapArray）	滑动窗口（基于RxJava）	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
流量整形	支持预热模式匀速器模式、预热排队模式	不支持	简单的Rate Limiter模式
系统自适应保护	支持	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、查看秒级监控，机器发观等	简单的监控查看	不提供控制台，可对接其它监控系统

持久化规则

是什么

一旦我们重启应用，sentinel规则将消失，生产环境需要将配置规则进行持久化。

怎么玩

将限流配置规则持久化进Nacos保存，只要刷新8401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8401上sentinel上的流控规则持续有效。

步骤

修改cloud-alibaba-sentinel-service8401

POM

```

1 <!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
2 <dependency>
3   <groupId>com.alibaba.csp</groupId>
4   <artifactId>sentinel-datasource-nacos</artifactId>
5 </dependency>
```

YML

```

1 server:
```

```
2 port: 8401
3
4 spring:
5   application:
6     name: cloud-alibaba-sentinel-service
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #Nacos服务注册中心地址
11 sentinel:
12   transport:
13     dashboard: localhost:8080 #配置Sentinel dashboard地址
14     port: 8719
15   datasource: #<-----关注点，添加Nacos数据源配置
16     ds1:
17       nacos:
18         server-addr: localhost:8848
19         dataId: cloud-alibaba-sentinel-service
20         groupId: DEFAULT_GROUP
21         data-type: json
22         rule-type: flow
23
24 management:
25   endpoints:
26     web:
27       exposure:
28         include: '*'
29
30 feign:
31   sentinel:
32     enabled: true # 激活Sentinel对Feign的支持
```

添加Nacos业务规则配置

新建配置

* Data ID: 微服务名称

* Group:

[更多高级选项](#)

描述:

配置格式: TEXT JSON XML YAML HTML Properties

* 配置内容: [?](#) :

```
1 [{}  
2   "resource": "/rateLimit/byUrl",  
3   "limitApp": "default",  
4   "grade": 1,  
5   "count": 1,  
6   "strategy": 0,  
7   "controlBehavior": 0,  
8   "clusterMode": false  
9 ]
```

配置内容解析

```
1 [{  
2   "resource": "/rateLimit/byUrl",  
3   "limitApp": "default",  
4   "grade": 1,  
5   "count": 1,  
6   "strategy": 0,  
7   "controlBehavior": 0,  
8   "clusterMode": false  
9 }]
```

- 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Warm Up, 2表示排队等待;
- clusterMode: 是否集群。

启动8401后刷新sentinel发现业务规则有了

Sentinel 控制台 1.7.0

cloud-alibaba-sentinel-service

流控规则

资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
/rateLimit/byUrl	default	直连	QPS	1	单机	快速失败	

快速访问测试接口 - <http://localhost:8401/rateLimit/byUrl> - 页面返回 blocked by sentinel (flow limiting)

停止8401再看sentinel - 停机后发现流控规则没有了

Sentinel 控制台 1.7.0

cloud-alibaba-sentinel-service

流控规则

资源名	来源应用	流控模式	阈值类型	阈值	阈值模式
-----	------	------	------	----	------

停机后发现流控规则没有了

重新启动8401再看sentinel

- 乍一看还是没有，稍等一会儿
- 多次调用 - <http://localhost:8401/rateLimit/byUrl>
- 重新配置出现了，持久化验证通过

Sentinel 控制台 1.7.0

cloud-alibaba-sentinel-service

流控规则

资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
/rateLimit/byUrl	default	直连	QPS	1	单机	快速失败	

Seata

分布式事务

分布式前

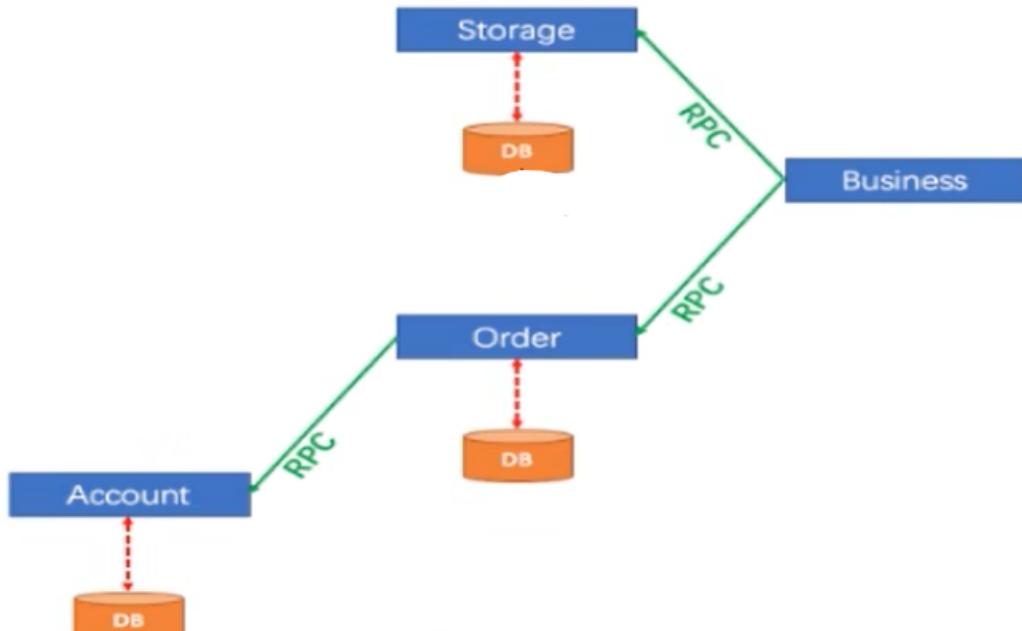
- 单机单库没这个问题
- 从1:1 -> 1:N -> N:N

单体应用被拆分成微服务应用，原来的三个模块被拆分成三个独立的应用，分别使用三个独立的数据源，业务操作需要调用三个服务来完成。此时每个服务内部的数据一致性由本地事务来保证，但是全局的数据一致性问题没法保证。

用户购买商品的业务逻辑。整个业务逻辑由3个微服务提供支持：

- 仓储服务：对给定的商品扣除仓储数量。
- 订单服务：根据采购需求创建订单。
- 帐户服务：从用户帐户中扣除余额。

架构图



一句话：一次业务操作需要跨多个数据源或需要跨多个系统进行远程调用，就会产生分布式事务问题。

Seata简介

是什么

Seata是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。

[官方网址](#)

能干嘛

一个典型的分布式事务过程

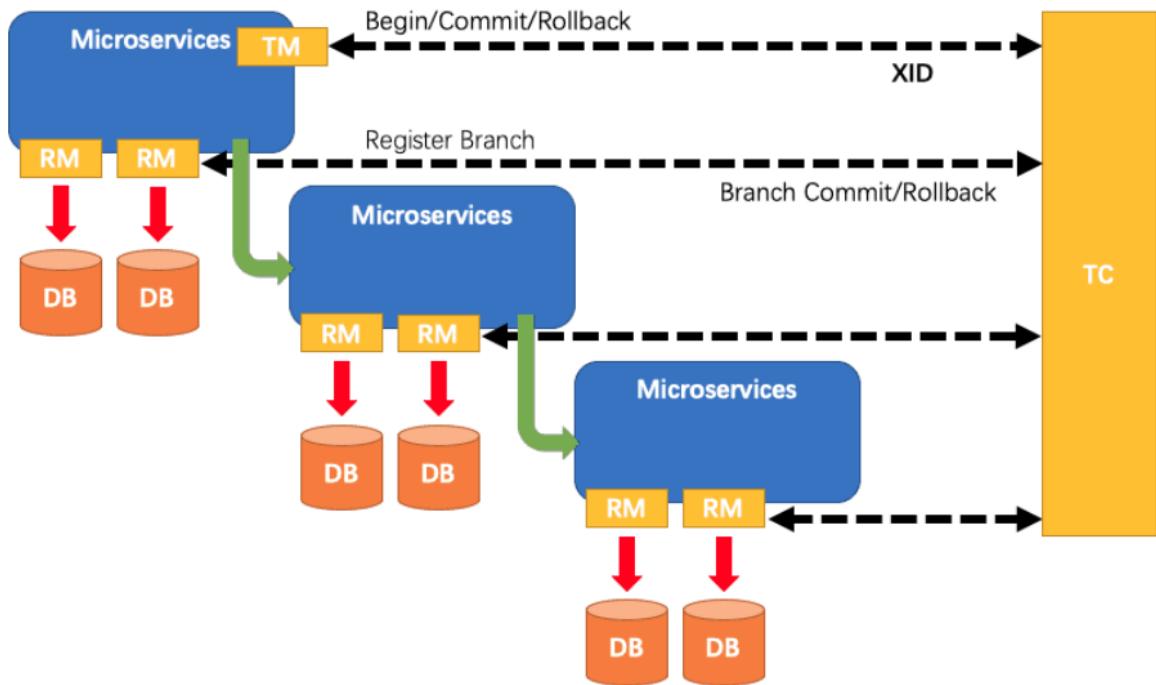
分布式事务处理过程的一ID+三组件模型：

- Transaction ID XID 全局唯一的事务ID
- 三组件概念
 - TC (Transaction Coordinator) - 事务协调者：维护全局和分支事务的状态，驱动全局事务提交或回滚。
 - TM (Transaction Manager) - 事务管理器：定义全局事务的范围：开始全局事务、提交或回滚全局事务。
 - RM (Resource Manager) - 资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

处理过程：

1. TM向TC申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的XID；
2. XID在微服务调用链路的上下文中传播；
3. RM向TC注册分支事务，将其纳入XID对应全局事务的管辖；

4. TM向TC发起针对XID的全局提交或回滚决议；
5. TC调度XID下管辖的全部分支事务完成提交或回滚请求。



去哪里

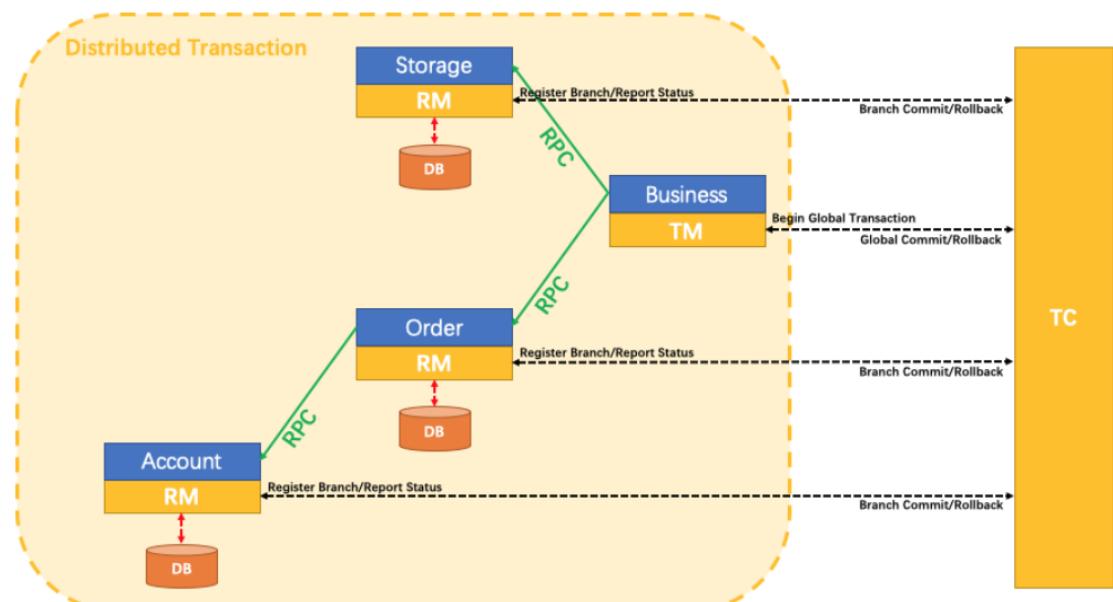
发布说明: <https://github.com/seata/seata/releases>

怎么玩

本地@Transactional

全局@GlobalTransactional

SEATA 的分布式交易解决方案



我们只需要使用一个 `@GlobalTransactional` 注解在业务方法上:

Seata-Server安装

1、官网地址 - <http://seata.io/zh-cn/>

2、下载版本 - 0.9.0

3、seata-server-0.9.0.zip解压到指定目录并修改conf目录下的file.conf配置文件

1. 先备份原始file.conf文件
2. 主要修改：自定义事务组名称+事务日志存储模式为db +数据库连接信息
3. file.conf

service模块

```
1 service {  
2     #vgroup->rgroup #fsp_tx_group是自定义的  
3     vgroup_mapping.my_test_tx_group = "fsp_tx_group"  
4     #only support single node  
5     default.grouplist = "127.0.0.1:8091"  
6     #degrade current not support  
7     enableDegrade = false  
8     #disable  
9     disable = false  
10    #unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days,  
11    default permanent  
12    max.commit.retry.timeout = "-1"  
13    max.rollback.retry.timeout = "-1"  
14 }
```

store模块

```
1 ## transaction log store  
2 store {  
3     ## store mode: file, db  
4     ## 改成db  
5     mode = "db"  
6  
7     ## file store  
8     file {  
9         dir = "sessionStore"  
10  
11         # branch session size, if exceeded first try compress lockkey, still  
12         exceeded throws exceptions  
13         max-branch-session-size = 16384  
14         # globe session size, if exceeded throws exceptions  
15         max-global-session-size = 512  
16         # file buffer size, if exceeded allocate new buffer  
17         file-write-buffer-cache-size = 16384  
18         # when recover batch read size  
19         session.reload.read_size= 100  
20         # async, sync  
21         flush-disk-mode = async  
22     }  
23  
24     # database store  
25     db {
```

```

25     ## the implement of javax.sql.DataSource, such as
DruidDataSource(druid)/BasicDataSource(dbcp) etc.
26     datasource = "dbcp"
27     ## mysql/oracle/h2/oceanbase etc.
28     ## 配置数据源
29     db-type = "mysql"
30     driver-class-name = "com.mysql.jdbc.Driver"
31     url = "jdbc:mysql://127.0.0.1:3306/seata"
32     user = "root"
33     password = "你自己密码"
34     min-conn= 1
35     max-conn = 3
36     global.table = "global_table"
37     branch.table = "branch_table"
38     lock-table = "lock_table"
39     query-limit = 100
40   }
41 }
```

4、mysql5.7数据库新建库seata

5、在seata库里建表

建表db_store.sql在\seata-server-0.9.0\seata\conf目录里面

```

1 -- the table to store GlobalSession data
2 drop table if exists `global_table`;
3 create table `global_table` (
4   `xid` varchar(128) not null,
5   `transaction_id` bigint,
6   `status` tinyint not null,
7   `application_id` varchar(32),
8   `transaction_service_group` varchar(32),
9   `transaction_name` varchar(128),
10  `timeout` int,
11  `begin_time` bigint,
12  `application_data` varchar(2000),
13  `gmt_create` datetime,
14  `gmt_modified` datetime,
15  primary key (`xid`),
16  key `idx_gmt_modified_status` (`gmt_modified`, `status`),
17  key `idx_transaction_id` (`transaction_id`)
18 );
19
20 -- the table to store BranchSession data
21 drop table if exists `branch_table`;
22 create table `branch_table` (
23   `branch_id` bigint not null,
24   `xid` varchar(128) not null,
25   `transaction_id` bigint ,
26   `resource_group_id` varchar(32),
27   `resource_id` varchar(256) ,
28   `lock_key` varchar(128) ,
29   `branch_type` varchar(8) ,
30   `status` tinyint,
31   `client_id` varchar(64),
32   `application_data` varchar(2000),
33   `gmt_create` datetime,
```

```

34   `gmt_modified` datetime,
35   primary key (`branch_id`),
36   key `idx_xid` (`xid`)
37 );
38
39 -- the table to store lock data
40 drop table if exists `lock_table`;
41 create table `lock_table` (
42   `row_key` varchar(128) not null,
43   `xid` varchar(96),
44   `transaction_id` long ,
45   `branch_id` long,
46   `resource_id` varchar(256) ,
47   `table_name` varchar(32) ,
48   `pk` varchar(36) ,
49   `gmt_create` datetime ,
50   `gmt_modified` datetime,
51   primary key(`row_key`)
52 );

```

6、修改seata-server-0.9.0\seata\conf目录下的registry.conf配置文件

```

1 registry {
2   # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
3   # 改用为nacos
4   type = "nacos"
5
6   nacos {
7     ## 加端口号
8     serverAddr = "localhost:8848"
9     namespace = ""
10    cluster = "default"
11  }
12  ...
13 }

```

目的是：指明注册中心为nacos，及修改nacos连接信息

7、先启动Nacos端口号8848

nacos\bin\startup.cmd

8、再启动seata-server

seata-server-0.9.0\seata\bin\seata-server.bat

订单/库存/账户业务数据库准备

以下演示都需要先启动Nacos后启动Seata,保证两个都OK。

分布式事务业务说明

这里我们会创建三个服务，一个订单服务，一个库存服务，一个账户服务。

当用户下单时,会在订单服务中创建一个订单,然后通过远程调用库存服务来扣减下单商品的库存,再通过远程调用账户服务来扣减用户账户里面的余额,最后在订单服务中修改订单状态为已完成。

该操作跨越三个数据库,有两次远程调用,很明显会有分布式事务问题。

一言蔽之, 下订单—>扣库存—>减账户(余额)。

创建业务数据库

- seata_order: 存储订单的数据库;
- seata_storage: 存储库存的数据库;
- seata_account: 存储账户信息的数据库。

建库SQL

```
1 CREATE DATABASE seata_order;
2 CREATE DATABASE seata_storage;
3 CREATE DATABASE seata_account;
```

按照上述3库分别建对应业务表

- seata_order库下建t_order表

```
1 CREATE TABLE t_order (
2     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
3     `user_id` BIGINT(11) DEFAULT NULL COMMENT '用户id',
4     `product_id` BIGINT(11) DEFAULT NULL COMMENT '产品id',
5     `count` INT(11) DEFAULT NULL COMMENT '数量',
6     `money` DECIMAL(11,0) DEFAULT NULL COMMENT '金额',
7     `status` INT(1) DEFAULT NULL COMMENT '订单状态: 0:创建中; 1:已完结'
8 ) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
9
10 SELECT * FROM t_order;
```

- seata_storage库下建t_storage表

```
1 CREATE TABLE t_storage (
2     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
3     `product_id` BIGINT(11) DEFAULT NULL COMMENT '产品id',
4     `total` INT(11) DEFAULT NULL COMMENT '总库存',
5     `used` INT(11) DEFAULT NULL COMMENT '已用库存',
6     `residue` INT(11) DEFAULT NULL COMMENT '剩余库存'
7 ) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
8
9 INSERT INTO seata_storage.t_storage(`id`, `product_id`, `total`, `used`,
10 `residue`)
11 VALUES ('1', '1', '100', '0', '100');
12
13 SELECT * FROM t_storage;
```

- seata_account库下建t_account表

```

1 CREATE TABLE t_account(
2     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY COMMENT 'id',
3     `user_id` BIGINT(11) DEFAULT NULL COMMENT '用户id',
4     `total` DECIMAL(10,0) DEFAULT NULL COMMENT '总额度',
5     `used` DECIMAL(10,0) DEFAULT NULL COMMENT '已用余额',
6     `residue` DECIMAL(10,0) DEFAULT '0' COMMENT '剩余可用额度'
7 ) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
8
9 INSERT INTO seata_account.t_account(`id`, `user_id`, `total`, `used`,
10 `residue`)
11 VALUES ('1', '1', '1000', '0', '1000');
12 SELECT * FROM t_account;

```

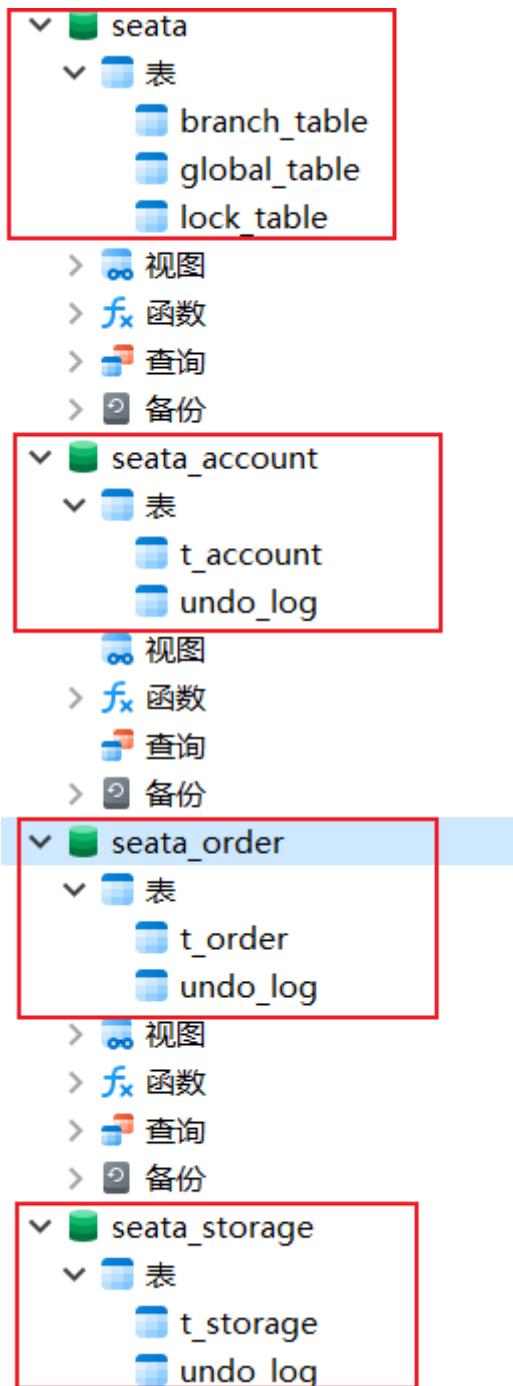
按照上述3库分别建对应的回滚日志表

- 订单-库存-账户3个库下都需要建各自的回滚日志表
- \seata-server-0.9.0\seata\conf目录下的db_undo_log.sql
- 建表SQL

```

1 -- the table to store seata xid data
2 -- 0.7.0+ add context
3 -- you must to init this sql for your business database. the seata server not
need it.
4 -- 此脚本必须初始化在你当前的业务数据库中，用于AT 模式XID记录。与server端无关（注：业务数
据库）
5 -- 注意此处0.3.0+ 增加唯一索引 ux_undo_log
6 drop table IF EXISTS `undo_log`;
7 CREATE TABLE `undo_log` (
8     `id` bigint(20) NOT NULL AUTO_INCREMENT,
9     `branch_id` bigint(20) NOT NULL,
10    `xid` varchar(100) NOT NULL,
11    `context` varchar(128) NOT NULL,
12    `rollback_info` longblob NOT NULL,
13    `log_status` int(11) NOT NULL,
14    `log_created` datetime NOT NULL,
15    `log_modified` datetime NOT NULL,
16    `ext` varchar(100) DEFAULT NULL,
17    PRIMARY KEY (`id`),
18    UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
19 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

```



订单/库存/账户业务微服务准备

业务需求：下订单 -> 减库存 -> 扣余额 -> 改（订单）状态

新建Order-Module

新建Module

名字：seata-order-service2001

POM

```

1 <dependencies>
2     <!--nacos-->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-starter-alibaba-nacos-
6         discovery</artifactId>
7     </dependency>

```

```
7      <!--seata-->
8      <dependency>
9          <groupId>com.alibaba.cloud</groupId>
10         <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
11         <exclusions>
12             <exclusion>
13                 <artifactId>seata-all</artifactId>
14                 <groupId>io.seata</groupId>
15             </exclusion>
16         </exclusions>
17     </dependency>
18     <dependency>
19         <groupId>io.seata</groupId>
20         <artifactId>seata-all</artifactId>
21         <version>0.9.0</version>
22     </dependency>
23     <!--feign-->
24     <dependency>
25         <groupId>org.springframework.cloud</groupId>
26         <artifactId>spring-cloud-starter-openfeign</artifactId>
27     </dependency>
28     <!--web-actuator-->
29     <dependency>
30         <groupId>org.springframework.boot</groupId>
31         <artifactId>spring-boot-starter-web</artifactId>
32     </dependency>
33     <dependency>
34         <groupId>org.springframework.boot</groupId>
35         <artifactId>spring-boot-starter-actuator</artifactId>
36     </dependency>
37     <!--mysql-druid-->
38     <dependency>
39         <groupId>mysql</groupId>
40         <artifactId>mysql-connector-java</artifactId>
41         <version>5.1.37</version>
42     </dependency>
43     <dependency>
44         <groupId>com.alibaba</groupId>
45         <artifactId>druid-spring-boot-starter</artifactId>
46         <version>1.1.10</version>
47     </dependency>
48     <dependency>
49         <groupId>org.mybatis.spring.boot</groupId>
50         <artifactId>mybatis-spring-boot-starter</artifactId>
51         <version>2.0.0</version>
52     </dependency>
53     <dependency>
54         <groupId>org.springframework.boot</groupId>
55         <artifactId>spring-boot-starter-test</artifactId>
56         <scope>test</scope>
57     </dependency>
58     <dependency>
59         <groupId>org.projectlombok</groupId>
60         <artifactId>lombok</artifactId>
61         <optional>true</optional>
62     </dependency>
63 </dependencies>
```

配置文件

YML

```
1 server:
2   port: 2001
3
4 spring:
5   application:
6     name: seata-order-service
7   cloud:
8     alibaba:
9       seata:
10      #自定义事务组名称需要与seata-server中的对应
11      tx-service-group: fsp_tx_group
12   nacos:
13     discovery:
14       server-addr: localhost:8848
15   datasource:
16     driver-class-name: com.mysql.jdbc.Driver
17     url: jdbc:mysql://localhost:3306/seata_order
18     username: root
19     password: 123456
20
21 feign:
22   hystrix:
23     enabled: false
24
25 logging:
26   level:
27     io:
28       seata: info
29
30 mybatis:
31   mapperLocations: classpath:mapper/*.xml
```

file.conf

```
1 transport {
2   # tcp udt unix-domain-socket
3   type = "TCP"
4   #NIO NATIVE
5   server = "NIO"
6   #enable heartbeat
7   heartbeat = true
8   #thread factory for netty
9   thread-factory {
10     boss-thread-prefix = "NettyBoss"
11     worker-thread-prefix = "NettyServerNIOWorker"
12     server-executor-thread-prefix = "NettyServerBizHandler"
13     share-boss-worker = false
14     client-selector-thread-prefix = "NettyClientSelector"
15     client-selector-thread-size = 1
16     client-worker-thread-prefix = "NettyClientWorkerThread"
17     # netty boss thread size,will not be used for UDT
18     boss-thread-size = 1
19     #auto default pin or 8
```

```
20     worker-thread-size = 8
21 }
22 shutdown {
23     # when destroy server, wait seconds
24     wait = 3
25 }
26 serialization = "seata"
27 compressor = "none"
28 }
29
30 service {
31
32     vgroup_mapping.fsp_tx_group = "default" #修改自定义事务组名称
33
34     default.grouplist = "127.0.0.1:8091"
35     enableDegrade = false
36     disable = false
37     max.commit.retry.timeout = "-1"
38     max.rollback.retry.timeout = "-1"
39     disableGlobalTransaction = false
40 }
41
42
43 client {
44     async.commit.buffer.limit = 10000
45     lock {
46         retry.internal = 10
47         retry.times = 30
48     }
49     report.retry.count = 5
50     tm.commit.retry.count = 1
51     tm.rollback.retry.count = 1
52 }
53
54 ## transaction log store
55 store {
56     ## store mode: file、db
57     mode = "db"
58
59     ## file store
60     file {
61         dir = "sessionStore"
62
63         # branch session size , if exceeded first try compress lockkey, still
64         # exceeded throws exceptions
65         max-branch-session-size = 16384
66         # globe session size , if exceeded throws exceptions
67         max-global-session-size = 512
68         # file buffer size , if exceeded allocate new buffer
69         file-write-buffer-cache-size = 16384
70         # when recover batch read size
71         session.reload.read_size = 100
72         # async, sync
73         flush-disk-mode = async
74     }
75
76     ## database store
77     db {
```

```
77     ## the implement of javax.sql.DataSource, such as
78     DruidDataSource(druid)/BasicDataSource(dbcp) etc.
79     datasource = "dbcp"
80     ## mysql/oracle/h2/oceanbase etc.
81     db-type = "mysql"
82     driver-class-name = "com.mysql.jdbc.Driver"
83     url = "jdbc:mysql://127.0.0.1:3306/seata"
84     user = "root"
85     password = "123456"
86     min-conn = 1
87     max-conn = 3
88     global.table = "global_table"
89     branch.table = "branch_table"
90     lock-table = "lock_table"
91     query-limit = 100
92 }
93
94 lock {
95     ## the lock store mode: local、remote
96     mode = "remote"
97
98     local {
99         ## store locks in user's database
100    }
101
102     remote {
103         ## store locks in the seata's server
104    }
105
106 recovery {
107     #schedule committing retry period in milliseconds
108     committing-retry-period = 1000
109     #schedule asyn committing retry period in milliseconds
110     asyn-committing-retry-period = 1000
111     #schedule rollbacks retry period in milliseconds
112     rollbacks-retry-period = 1000
113     #schedule timeout retry period in milliseconds
114     timeout-retry-period = 1000
115 }
116
117 transaction {
118     undo.data.validation = true
119     undo.log.serialization = "jackson"
120     undo.log.save.days = 7
121     #schedule delete expired undo_log in milliseconds
122     undo.log.delete.period = 86400000
123     undo.log.table = "undo_log"
124 }
125
126 ## metrics settings
127 metrics {
128     enabled = false
129     registry-type = "compact"
130     # multi exporters use comma divided
131     exporter-list = "prometheus"
132     exporter-prometheus-port = 9898
133 }
```

```
134 support {
135     ## spring
136     spring {
137         # auto proxy the DataSource bean
138         datasource.autoproxy = false
139     }
140 }
```

registry.conf

```
1 registry {
2     # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
3     type = "nacos"
4
5     nacos {
6         serverAddr = "localhost:8848"
7         namespace = ""
8         cluster = "default"
9     }
10    eureka {
11        serviceUrl = "http://localhost:8761/eureka"
12        application = "default"
13        weight = "1"
14    }
15    redis {
16        serverAddr = "localhost:6379"
17        db = "0"
18    }
19    zk {
20        cluster = "default"
21        serverAddr = "127.0.0.1:2181"
22        session.timeout = 6000
23        connect.timeout = 2000
24    }
25    consul {
26        cluster = "default"
27        serverAddr = "127.0.0.1:8500"
28    }
29    etcd3 {
30        cluster = "default"
31        serverAddr = "http://localhost:2379"
32    }
33    sofa {
34        serverAddr = "127.0.0.1:9603"
35        application = "default"
36        region = "DEFAULT_ZONE"
37        datacenter = "DefaultDataCenter"
38        cluster = "default"
39        group = "SEATA_GROUP"
40        addresswaitForTime = "3000"
41    }
42    file {
43        name = "file.conf"
44    }
45 }
46
47 config {
```

```

48 # file、nacos 、apollo、zk、consul、etcd3
49 type = "file"
50
51 nacos {
52     serverAddr = "localhost"
53     namespace = ""
54 }
55 consul {
56     serverAddr = "127.0.0.1:8500"
57 }
58 apollo {
59     app.id = "seata-server"
60     apollo.meta = "http://192.168.1.204:8801"
61 }
62 zk {
63     serverAddr = "127.0.0.1:2181"
64     session.timeout = 6000
65     connect.timeout = 2000
66 }
67 etcd3 {
68     serverAddr = "http://localhost:2379"
69 }
70 file {
71     name = "file.conf"
72 }
73 }
```

domain

```

1 import lombok.AllArgsConstructor;
2 import lombok.Data;
3 import lombok.NoArgsConstructor;
4
5 @Data
6@AllArgsConstructor
7@NoArgsConstructor
8 public class CommonResult<T>{
9     private Integer code;
10    private String message;
11    private T data;
12
13    public CommonResult(Integer code, String message){
14        this(code,message,null);
15    }
16}
```

```

1 import lombok.AllArgsConstructor;
2 import lombok.Data;
3 import lombok.NoArgsConstructor;
4
5 import java.math.BigDecimal;
6
7 @Data
8@AllArgsConstructor
9@NoArgsConstructor
10 public class Order{
```

```

11     private Long id;
12
13     private Long userId;
14
15     private Long productId;
16
17     private Integer count;
18
19     private BigDecimal money;
20
21     private Integer status; //订单状态: 0: 创建中; 1: 已完结
22 }

```

Dao接口及实现

```

1 import com.atguigu.springcloud.alibaba.domain.Order;
2 import org.apache.ibatis.annotations.Mapper;
3 import org.apache.ibatis.annotations.Param;
4
5 @Mapper
6 public interface OrderDao{
7     //1 新建订单
8     void create(Order order);
9
10    //2 修改订单状态, 从零改为1
11    void update(@Param("userId") Long userId,@Param("status") Integer
12    status);
13 }

```

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3 "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5 <mapper namespace="com.example.springcloud.alibaba.dao.OrderDao">
6
7     <resultMap id="BaseResultMap"
8     type="com.example.springcloud.alibaba.domain.Order">
9         <id column="id" property="id" jdbcType="BIGINT"/>
10        <result column="user_id" property="userId" jdbcType="BIGINT"/>
11        <result column="product_id" property="productId" jdbcType="BIGINT"/>
12        <result column="count" property="count" jdbcType="INTEGER"/>
13        <result column="money" property="money" jdbcType="DECIMAL"/>
14        <result column="status" property="status" jdbcType="INTEGER"/>
15     </resultMap>
16
17     <insert id="create">
18         insert into t_order (id,user_id,product_id,count,money,status)
19             values (null,#{userId},#{productId},#{count},#{money},0);
20     </insert>
21
22     <update id="update">
23         update t_order set status = 1
24             where user_id=#{userId} and status = #{status};
25     </update>

```

Service接口及实现

- OrderService
 - OrderServiceImpl
- StorageService
- AccountService

```

1 import com.example.springcloud.alibaba.domain.Order;
2
3 public interface OrderService{
4     void create(Order order);
5 }
```

```

1 import com.atguigu.springcloud.alibaba.domain.CommonResult;
2 import org.springframework.cloud.openfeign.FeignClient;
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestParam;
5
6 import java.math.BigDecimal;
7
8 @FeignClient(value = "seata-storage-service")
9 public interface StorageService{
10     @PostMapping(value = "/storage/decrease")
11     CommonResult decrease(@RequestParam("productId") Long productId,
12     @RequestParam("count") Integer count);
13 }
```

```

1 import com.atguigu.springcloud.alibaba.domain.CommonResult;
2 import org.springframework.cloud.openfeign.FeignClient;
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestParam;
5
6 import java.math.BigDecimal;
7
8 @FeignClient(value = "seata-account-service")
9 public interface AccountService{
10     @PostMapping(value = "/account/decrease")
11     CommonResult decrease(@RequestParam("userId") Long userId,
12     @RequestParam("money") BigDecimal money);
13 }
```

```

1 import com.example.springcloud.alibaba.dao.AccountService;
2 import com.example.springcloud.alibaba.dao.OrderDao;
3 import com.example.springcloud.alibaba.dao.StorageService;
4 import com.example.springcloud.alibaba.domain.Order;
5 import com.example.springcloud.alibaba.service.Orderservice;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.stereotype.Service;
8
9 import javax.annotation.Resource;
10
11 @Service
```

```

12 @Slf4j
13 public class OrderServiceImpl implements OrderService{
14     @Resource
15     private OrderDao orderDao;
16     @Resource
17     private StorageService storageService;
18     @Resource
19     private AccountService accountService;
20
21     /**
22      * 创建订单->调用库存服务扣减库存->调用账户服务扣减账户余额->修改订单状态
23      * 简单说：下订单->扣库存->减余额->改状态
24      */
25     @Override
26     //@GlobalTransactional(name = "fsp-create-order", rollbackFor =
Exception.class)
27     public void create(Order order){
28         log.info("----->开始新建订单");
29         //1 新建订单
30         orderDao.create(order);
31
32         //2 扣减库存
33         log.info("----->订单微服务开始调用库存，做扣减count");
34         storageService.decrease(order.getProductId(),order.getCount());
35         log.info("----->订单微服务开始调用库存，做扣减end");
36
37         //3 扣减账户
38         log.info("----->订单微服务开始调用账户，做扣减Money");
39         accountService.decrease(order.getUserId(),order.getMoney());
40         log.info("----->订单微服务开始调用账户，做扣减end");
41
42         //4 修改订单状态，从零到1,1代表已经完成
43         log.info("----->修改订单状态开始");
44         orderDao.update(order.getUserId(),0);
45         log.info("----->修改订单状态结束");
46
47         log.info("----->下订单结束了，o(∩_∩)o哈哈~");
48
49     }
50 }
```

Controller

```

1 import com.example.springcloud.alibaba.domain.CommonResult;
2 import com.example.springcloud.alibaba.domain.Order;
3 import com.example.springcloud.alibaba.service.OrderService;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.RequestBody;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import javax.annotation.Resource;
9
10 @RestController
11 public class OrderController{
12     @Resource
13     private OrderService orderService;
```

```
15
16     @GetMapping("/order/create")
17     public CommonResult create(@RequestBody Order order){
18         orderService.create(order);
19         return new CommonResult(200,"订单创建成功");
20     }
21 }
```

Config配置

- MyBatisConfig
- DataSourceProxyConfig

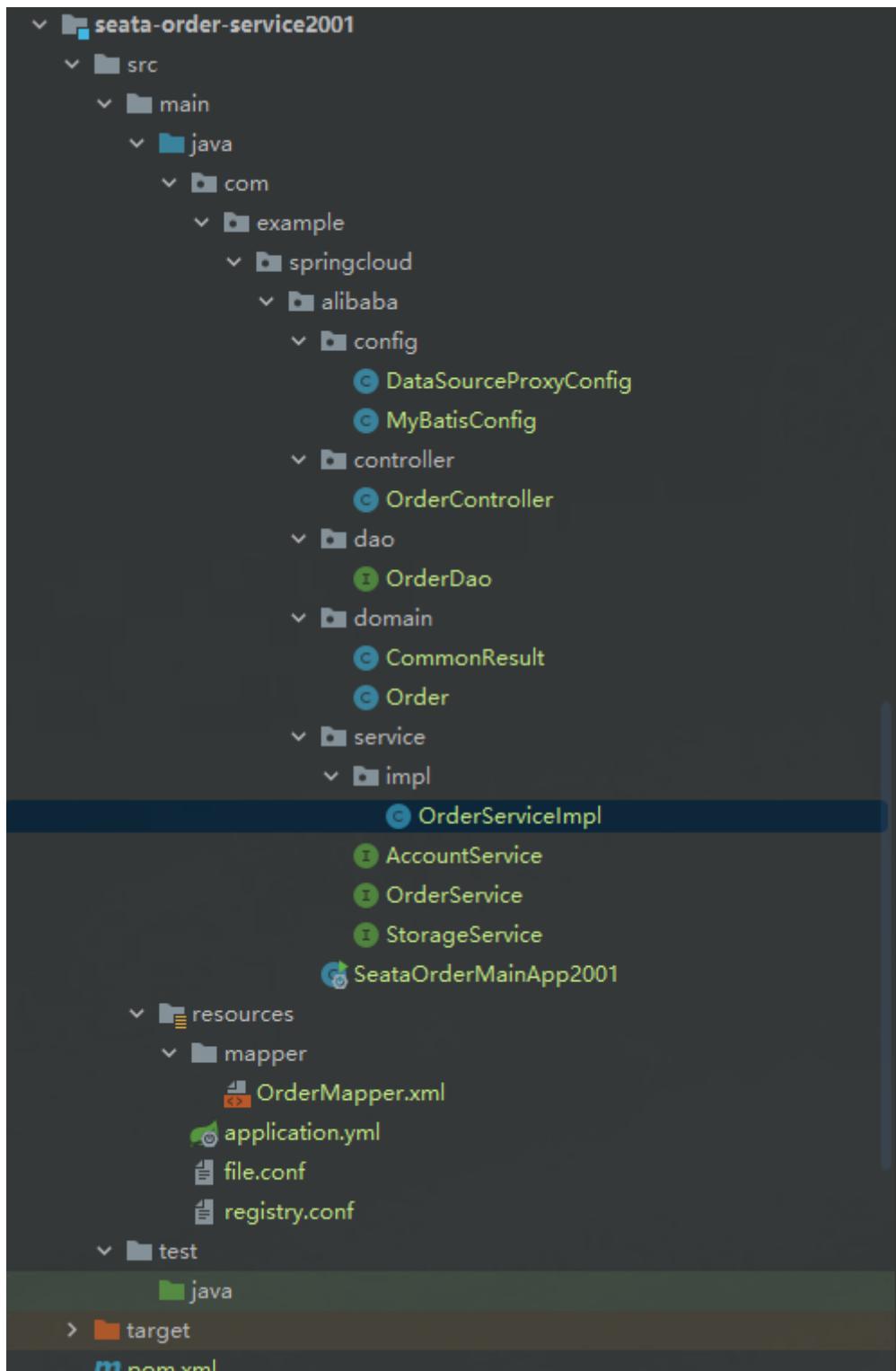
```
1 import org.mybatis.spring.annotation.MapperScan;
2 import org.springframework.context.annotation.Configuration;
3
4 @Configuration
5 @MapperScan({"com.example.springcloud.alibaba.dao"})
6 public class MyBatisConfig {
7 }
```

```
1 import com.alibaba.druid.pool.DruidDataSource;
2 import io.seata.rm.datasource.DataSourceProxy;
3 import org.apache.ibatis.session.SqlSessionFactory;
4 import org.mybatis.spring.SqlSessionFactoryBean;
5 import org.mybatis.spring.transaction.SpringManagedTransactionFactory;
6 import org.springframework.beans.factory.annotation.Value;
7 import org.springframework.boot.context.properties.ConfigurationProperties;
8 import org.springframework.context.annotation.Bean;
9 import org.springframework.context.annotation.Configuration;
10 import
11     org.springframework.core.io.support.PathMatchingResourcePatternResolver;
12 import javax.sql.DataSource;
13 /**
14  * 使用Seata对数据源进行代理
15 */
16 @Configuration
17 public class DataSourceProxyConfig {
18
19     @Value("${mybatis.mapperLocations}")
20     private String mapperLocations;
21
22     @Bean
23     @ConfigurationProperties(prefix = "spring.datasource")
24     public DataSource druidDataSource(){
25         return new DruidDataSource();
26     }
27
28     @Bean
29     public DataSourceProxy dataSourceProxy(DataSource dataSource) {
30         return new DataSourceProxy(dataSource);
31     }
32
33     @Bean
```

```
34     public SqlSessionFactory sqlSessionFactory(DataSourceProxy  
35         dataSourceProxy) throws Exception {  
36         SqlSessionFactoryBean sqlSessionFactoryBean = new  
37             SqlSessionFactoryBean();  
38             sqlSessionFactoryBean.setDataSource(dataSourceProxy);  
39             sqlSessionFactoryBean.setMapperLocations(new  
40                 PathMatchingResourcePatternResolver().getResources(mapperLocations));  
41             sqlSessionFactoryBean.setTransactionFactory(new  
42                 SpringManagedTransactionFactory());  
43             return sqlSessionFactoryBean.getObject();  
44     }  
45 }
```

主启动

```
1 import org.springframework.boot.SpringApplication;  
2 import org.springframework.boot.autoconfigure.SpringBootApplication;  
3 import  
4     org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;  
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
6 import org.springframework.cloud.openfeign.EnableFeignClients;  
7  
8 @EnableDiscoveryClient  
9 @EnableFeignClients  
10 //取消数据源的自动创建，而是使用自己定义的  
11 @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)  
12 public class SeataOrderMainApp2001{  
13  
14     public static void main(String[] args){  
15         SpringApplication.run(SeataorderMainApp2001.class, args);  
16     }  
17 }
```



新建Storage-Module

与seata-order-service2001模块大致相同

新建Module

名称: seata-storage-service2002

POM

与seata-order-service2001模块大致相同

YML

```
1 server:
2   port: 2002
3
4 spring:
5   application:
6     name: seata-storage-service
7   cloud:
8     alibaba:
9       seata:
10      tx-service-group: fsp_tx_group
11
12 nacos:
13   discovery:
14     server-addr: localhost:8848
15
16 datasource:
17   driver-class-name: com.mysql.jdbc.Driver
18   url: jdbc:mysql://localhost:3306/seata_storage
19   username: root
20   password: 123456
21
22 logging:
23   level:
24     io:
25       seata: info
26
27 mybatis:
28   mapperLocations: classpath:mapper/*.xml
```

file.conf (与seata-order-service2001模块大致相同)

registry.conf (与seata-order-service2001模块大致相同)

domain

```
1 import lombok.Data;
2
3 @Data
4 public class Storage {
5
6   private Long id;
7
8   /**
9    * 产品id
10   */
11   private Long productId;
12
13   /**
14    * 总库存
15   */
16   private Integer total;
17
18   /**
19    * 已用库存
20   */
21   private Integer used;
```

```
23     /**
24      * 剩余库存
25      */
26     private Integer residue;
27 }
```

CommonResult (与seata-order-service2001模块大致相同)

Dao接口及实现

```
1 import org.apache.ibatis.annotations.Mapper;
2 import org.apache.ibatis.annotations.Param;
3
4 @Mapper
5 public interface StorageDao {
6
7     //扣减库存
8     void decrease(@Param("productId") Long productId, @Param("count") Integer
9 count);
9 }
```

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3 "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5 <mapper namespace="com.example.springcloud.alibaba.dao.StorageDao">
6
7     <resultMap id="BaseResultMap"
8         type="com.example.springcloud.alibaba.domain.Storage">
9         <id column="id" property="id" jdbcType="BIGINT"/>
10        <result column="product_id" property="productId" jdbcType="BIGINT"/>
11        <result column="total" property="total" jdbcType="INTEGER"/>
12        <result column="used" property="used" jdbcType="INTEGER"/>
13        <result column="residue" property="residue" jdbcType="INTEGER"/>
14    </resultMap>
15
16    <update id="decrease">
17        UPDATE
18            t_storage
19        SET
20            used = used + #{count}, residue = residue - #{count}
21        WHERE
22            product_id = #{productId}
23    </update>
24
25 </mapper>
```

Service接口及实现

```
1 public interface StorageService {
2     /**
3      * 扣减库存
4      */
5     void decrease(Long productId, Integer count);
6 }
```

```

1 import com.example.springcloud.alibaba.dao.StorageDao;
2 import com.example.springcloud.alibaba.service.StorageService ;
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import javax.annotation.Resource;
9
10 @Service
11 public class StorageServiceImpl implements StorageService {
12
13     private static final Logger LOGGER =
14         LoggerFactory.getLogger(StorageServiceImpl.class);
15
16     @Resource
17     private StorageDao storageDao;
18
19     /**
20      * 扣减库存
21      */
22     @Override
23     public void decrease(Long productId, Integer count) {
24         LOGGER.info("----->storage-service中扣减库存开始");
25         storageDao.decrease(productId, count);
26         LOGGER.info("----->storage-service中扣减库存结束");
27     }

```

Controller

```

1 import com.example.springcloud.alibaba.domain.CommonResult ;
2 import com.example.springcloud.alibaba.service.StorageService ;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class StorageController {
9
10     @Autowired
11     private StorageService storageService;
12
13     /**
14      * 扣减库存
15      */
16     @RequestMapping("/storage/decrease")
17     public CommonResult decrease(Long productId, Integer count) {
18         storageService.decrease(productId, count);
19         return new CommonResult(200, "扣减库存成功! ");
20     }
21 }

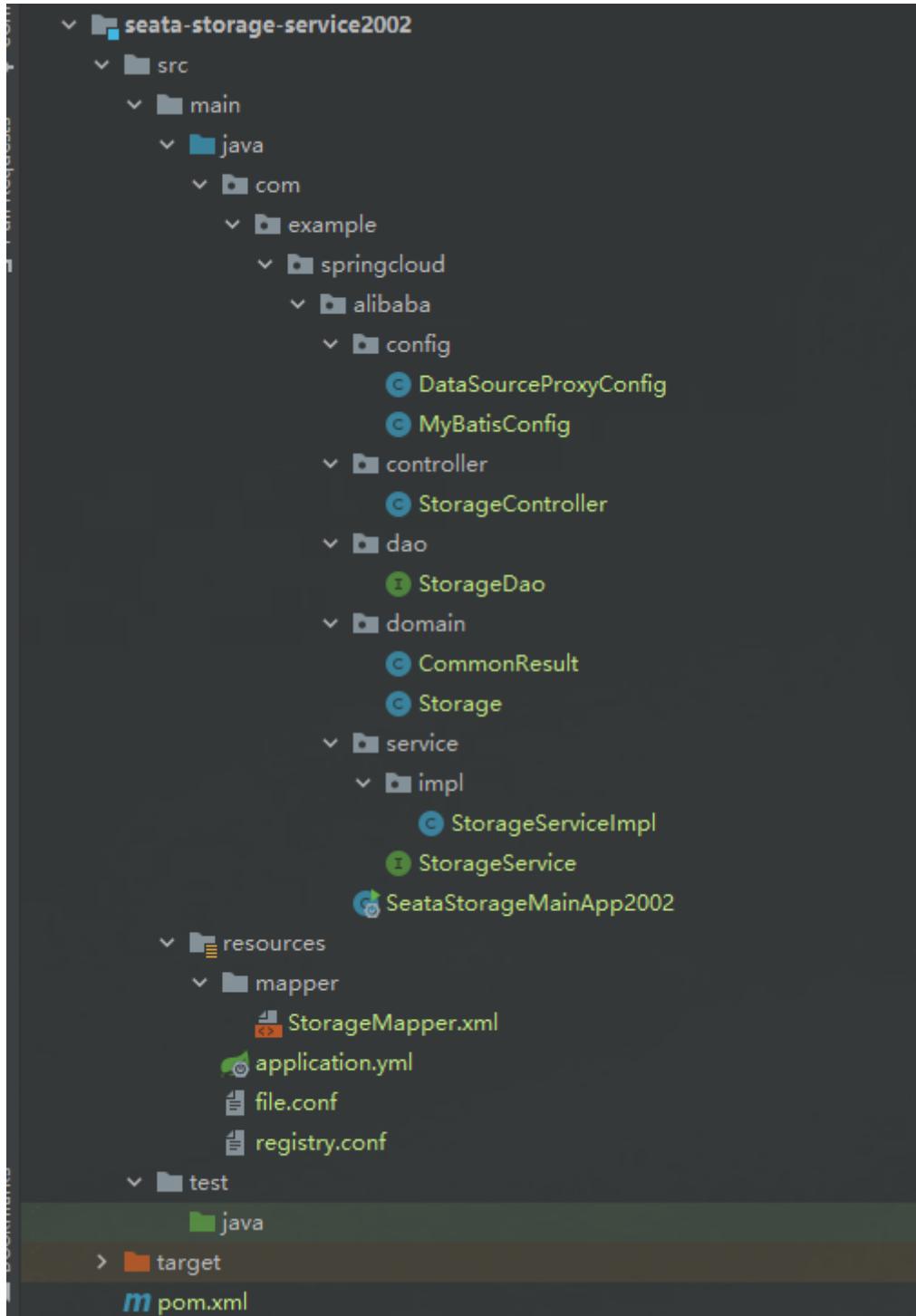
```

Config配置

与seata-order-service2001模块大致相同

主启动

与seata-order-service2001模块大致相同



新建Account-Module

与seata-order-service2001模块大致相同

新建Module

名称: seata-account-service2003

POM

与seata-order-service2001模块大致相同

YML

```
1 server:
2   port: 2003
3
4 spring:
5   application:
6     name: seata-account-service
7   cloud:
8     alibaba:
9       seata:
10      tx-service-group: fsp_tx_group
11   nacos:
12     discovery:
13       server-addr: localhost:8848
14   datasource:
15     driver-class-name: com.mysql.jdbc.Driver
16     url: jdbc:mysql://localhost:3306/seata_account
17     username: root
18     password: 123456
19
20 feign:
21   hystrix:
22     enabled: false
23
24 logging:
25   level:
26     io:
27       seata: info
28
29 mybatis:
30   mapperLocations: classpath:mapper/*.xml
```

file.conf (与seata-order-service2001模块大致相同)

registry.conf (与seata-order-service2001模块大致相同)

domain

```
1 import lombok.AllArgsConstructor;
2 import lombok.Data;
3 import lombok.NoArgsConstructor;
4
5 import java.math.BigDecimal;
6
7 @Data
8@AllArgsConstructor
9@NoArgsConstructor
10 public class Account {
```

```

12     private Long id;
13
14     /**
15      * 用户id
16      */
17     private Long userId;
18
19     /**
20      * 总额度
21      */
22     private BigDecimal total;
23
24     /**
25      * 已用额度
26      */
27     private BigDecimal used;
28
29     /**
30      * 剩余额度
31      */
32     private BigDecimal residue;
33 }

```

CommonResult (与seata-order-service2001模块大致相同)

Dao接口及实现

```

1 import org.apache.ibatis.annotations.Mapper;
2 import org.apache.ibatis.annotations.Param;
3 import org.springframework.stereotype.Component;
4 import org.springframework.stereotype.Repository;
5
6 import java.math.BigDecimal;
7
8 @Mapper
9 public interface AccountDao {
10
11     /**
12      * 扣减账户余额
13      */
14     void decrease(@Param("userId") Long userId, @Param("money") BigDecimal
15     money);
16 }

```

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3 "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4
5 <mapper namespace="com.example.springcloud.alibaba.dao.AccountDao">
6
7     <resultMap id="BaseResultMap"
8     type="com.example.springcloud.alibaba.domain.Account">
9         <id column="id" property="id" jdbcType="BIGINT"/>
10        <result column="user_id" property="userId" jdbcType="BIGINT"/>
11        <result column="total" property="total" jdbcType="DECIMAL"/>
12        <result column="used" property="used" jdbcType="DECIMAL"/>
13        <result column="residue" property="residue" jdbcType="DECIMAL"/>

```

```

12    </resultMap>
13
14    <update id="decrease">
15        UPDATE t_account
16        SET
17            residue = residue - #{money}, used = used + #{money}
18        WHERE
19            user_id = #{userId};
20    </update>
21
22 </mapper>

```

Service接口及实现

```

1 import org.springframework.web.bind.annotation.RequestParam;
2 import org.springframework.web.bind.annotation.ResponseBody;
3
4 import java.math.BigDecimal;
5
6 public interface AccountService {
7
8     /**
9      * 扣减账户余额
10     * @param userId 用户id
11     * @param money 金额
12     */
13     void decrease(@RequestParam("userId") Long userId,
14                   @RequestParam("money") BigDecimal money);
15 }

```

```

1 import com.example.springcloud.alibaba.dao.AccountDao;
2 import com.example.springcloud.alibaba.service.AccountService ;
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import javax.annotation.Resource;
9 import java.math.BigDecimal;
10 import java.util.concurrent.TimeUnit;
11
12 @Service
13 public class AccountServiceImpl implements AccountService {
14
15     private static final Logger LOGGER =
16 LoggerFactory.getLogger(AccountServiceImpl.class);
17
18     @Resource
19     AccountDao accountDao;
20
21     /**
22      * 扣减账户余额
23      */
24     @Override
25     public void decrease(Long userId, BigDecimal money) {

```

```
26     LOGGER.info("----->account-service中扣减账户余额开始");
27     accountDao.decrease(userId,money);
28     LOGGER.info("----->account-service中扣减账户余额结束");
29 }
30 }
```

Controller

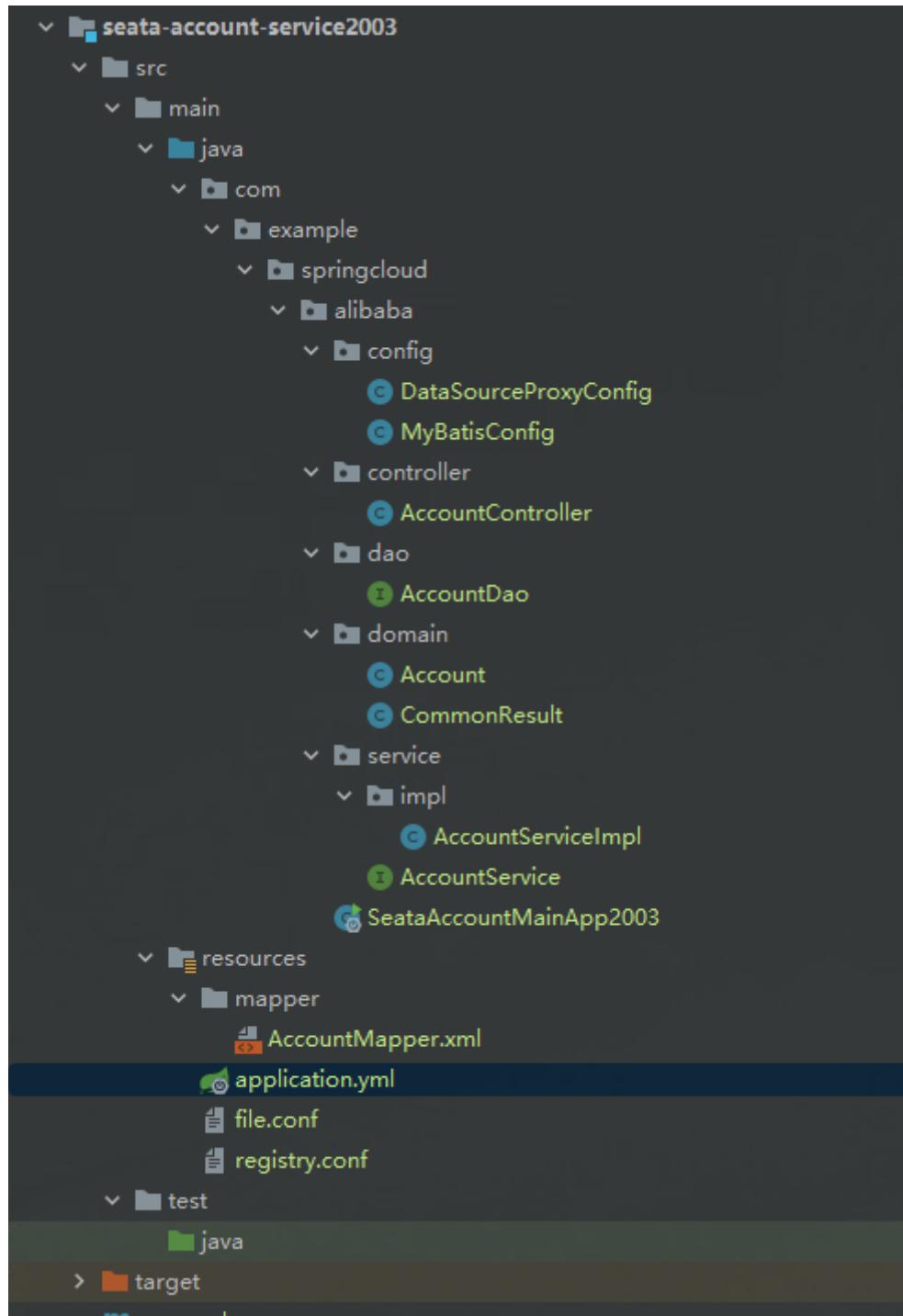
```
1 import com.atguigu.springcloud.alibaba.domain.CommonResult ;
2 import com.atguigu.springcloud.alibaba.service.AccountService ;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.web.bind.annotation.PostMapping;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7 import org.springframework.web.bind.annotation.RestController;
8
9 import javax.annotation.Resource;
10 import java.math.BigDecimal;
11
12 @RestController
13 public class AccountController {
14
15     @Resource
16     AccountService accountService;
17
18     /**
19      * 扣减账户余额
20      */
21     @RequestMapping("/account/decrease")
22     public CommonResult decrease(@RequestParam("userId") Long userId,
23 @RequestParam("money") BigDecimal money){
24         accountService.decrease(userId,money);
25         return new CommonResult(200,"扣减账户余额成功!");
26     }
27 }
```

Config配置

与seata-order-service2001模块大致相同

主启动

与seata-order-service2001模块大致相同



TEST

下订单 -> 减库存 -> 扣余额 -> 改 (订单) 状态

数据库初始情况

```
SELECT * from seata_order.t_order;
```

```
1 SELECT * from seata_order.t_order;
```

信息	摘要	结果 1	剖析	状态	
id	user_id	product_id	count	money	status
(N/A)	(N/A)	(N/A)	(N/A)	(N/A)	(N/A)

```
SELECT * from seata_storage.t_storage;
```

```
1 SELECT * from seata_storage.t_storage;
```

id	product_id	total	used	residue
1	1	100	0	100

```
SELECT * from seata_account.t_account;
```

```
1 SELECT * from seata_account.t_account;
```

id	user_id	total	used	residue
1	1	1000	0	1000

正常下单

- <http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>

数据库正常下单后状况：

```
SELECT * from seata_order.t_order;
```

```
1 SELECT * from seata_order.t_order;
```

id	user_id	product_id	count	money	status
1	1	1	1	10	100

```
SELECT * from seata_storage.t_storage;
```

```
1 SELECT * from seata_storage.t_storage;
```

信息	摘要	结果 1	剖析	状态
id	product_id	total	used	residue
▶ 1	1	100	10	90

```
SELECT * from seata_account.t_account;
```

```
1 SELECT * from seata_account.t_account;
```

信息	摘要	结果 1	剖析	状态
id	user_id	total	used	residue
▶ 1	1	1000	100	900

超时异常

没加@GlobalTransactional

模拟AccountServiceImpl添加超时

```
1 @Service
2 public class AccountServiceImpl implements AccountService {
3
4     private static final Logger LOGGER =
5     LoggerFactory.getLogger(AccountServiceImpl.class);
6
7
8     @Resource
9     AccountDao accountDao;
10
11    /**
12     * 扣减账户余额
13     */
14    @Override
15    public void decrease(Long userId, BigDecimal money) {
16        LOGGER.info("----->account-service中扣减账户余额开始");
17        //模拟超时异常，全局事务回滚
18        //暂停几秒钟线程
19        try { TimeUnit.SECONDS.sleep(20); } catch (InterruptedException e) {
20            e.printStackTrace();
21            accountDao.decrease(userId,money);
22            LOGGER.info("----->account-service中扣减账户余额结束");
23        }
24    }
25}
```

另外，OpenFeign的调用默认时间是1s以内，所以最后会抛异常。

数据库情况

SELECT * from seata_order.t_order;

The screenshot shows the results of a SELECT query on the t_order table. The table has columns: id, user_id, product_id, count, money, and status. There are two rows: one with id 1, user_id 1, product_id 1, count 10, money 100, and status 1; and another with id 2, user_id 1, product_id 1, count 10, money 100, and status 0.

id	user_id	product_id	count	money	status
1	1	1	10	100	1
2	1	1	10	100	0

SELECT * from seata_storage.t_storage;

The screenshot shows the results of a SELECT query on the t_storage table. The table has columns: id, product_id, total, used, and residue. There is one row with id 1, product_id 1, total 100, used 20, and residue 80.

id	product_id	total	used	residue
1	1	100	20	80

SELECT * from seata_account.t_account;

The screenshot shows the results of a SELECT query on the t_account table. The table has columns: id, user_id, total, used, and residue. There is one row with id 1, user_id 1, total 1000, used 200, and residue 800.

id	user_id	total	used	residue
1	1	1000	200	800

故障情况

- 当库存和账户金额扣减后，订单状态并没有设置为已经完成，没有从零改为1
- 而且由于feign的重试机制，账户余额还有可能被多次扣减

加了@GlobalTransactional

用@GlobalTransactional标注OrderServiceImpl的create()方法。

```
1 @Service  
2 @Slf4j
```

```

3  public class OrderServiceImpl implements Orderservice {
4
5      ...
6
7      /**
8       * 创建订单->调用库存服务扣减库存->调用账户服务扣减账户余额->修改订单状态
9       * 简单说：下订单->扣库存->减余额->改状态
10      */
11     @Override
12     //rollbackFor = Exception.class表示对任意异常都进行回滚
13     @GlobalTransactional(name = "fsp-create-order", rollbackFor =
Exception.class)
14     public void create(Order order){
15         ...
16     }
17 }

```

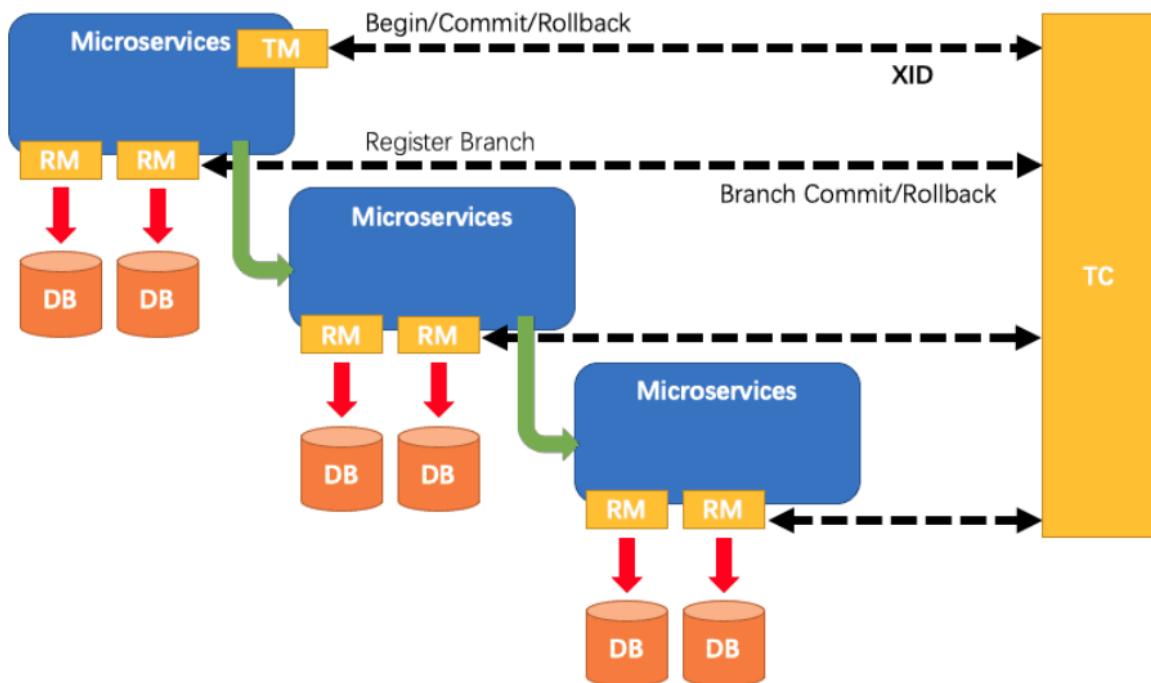
还是模拟AccountServiceImpl添加超时，下单后数据库数据并没有任何改变，记录都添加不进来，**达到出异常，数据库回滚的效果。**

原理简介

2019年1月份蚂蚁金服和阿里巴巴共同开源的分布式事务解决方案。

Simple Extensible Autonomous Transaction Architecture，简单可扩展自治事务框架。

2020起始，用1.0以后的版本。Alina Gingertail



分布式事务的执行流程

- TM开启分布式事务(TM向TC注册全局事务记录)；
- 按业务场景，编排数据库、服务等事务内资源(RM向TC汇报资源准备状态)；
- TM结束分布式事务，事务一阶段结束(TM通知TC提交/回滚分布式事务)；
- TC汇总事务信息，决定分布式事务是提交还是回滚；
- TC通知所有RM提交/回滚资源，事务二阶段结束。

AT模式如何做到对业务的无侵入

- 是什么

- 前提

- 基于支持本地 ACID 事务的关系型数据库。
 - Java 应用，通过 JDBC 访问数据库。

- 整体机制（两阶段提交协议的演变：）

- 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。
 - 二阶段：

- 提交异步化，非常快速地完成。
 - 回滚通过一阶段的回滚日志进行反向补偿。

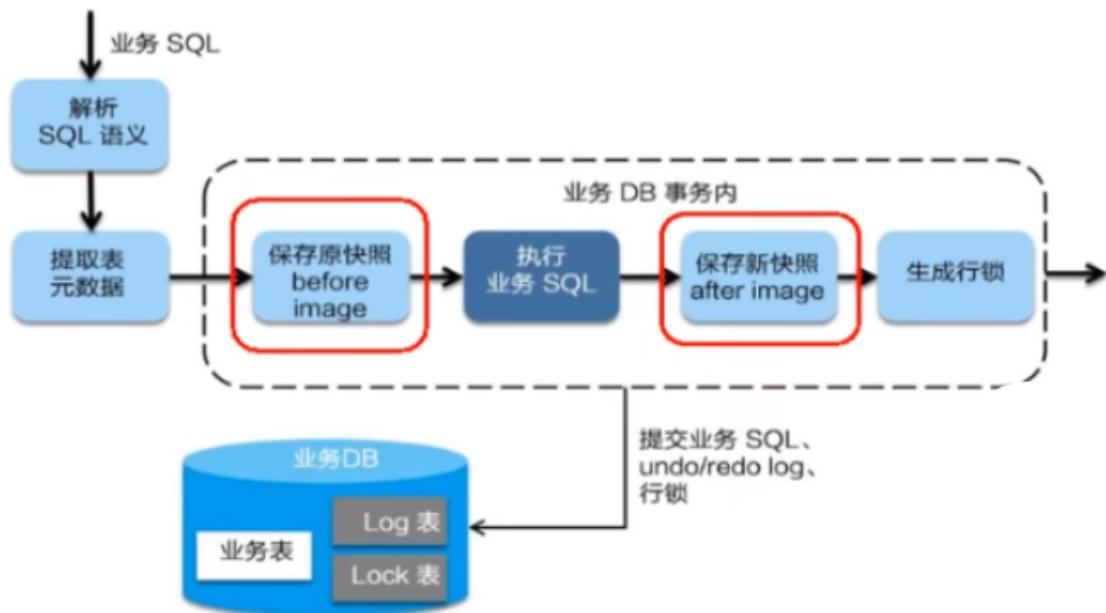
[link](#)

- 一阶段加载

在一阶段，Seata会拦截“业务SQL”

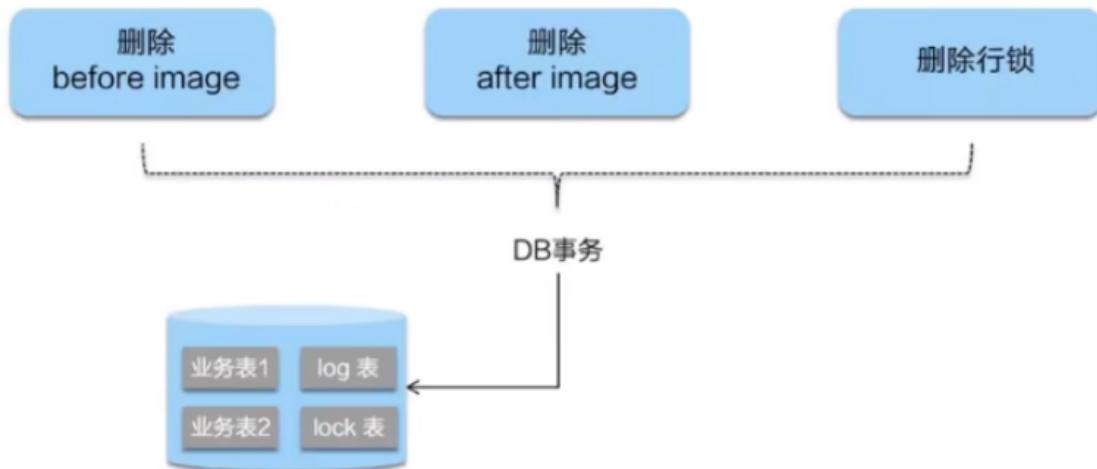
1. 解析SQL语义，找到“业务SQL”要更新的业务数据，在业务数据被更新前，将其保存成"before image"
2. 执行“业务SQL”更新业务数据，在业务数据更新之后，
3. 其保存成"after image"，最后生成行锁。

以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



- 二阶段提交

二阶段如果顺利提交的话，因为“业务SQL”在一阶段已经提交至数据库，所以Seata框架只需将一阶段保存的快照数据和行锁删掉，完成数据清理即可。

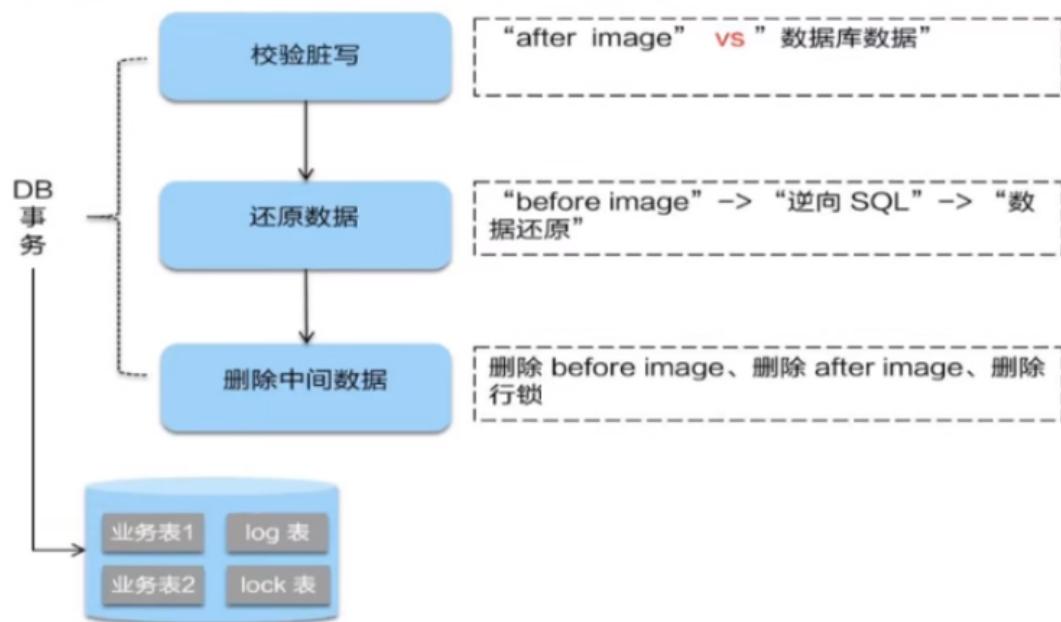


- 二阶段回滚

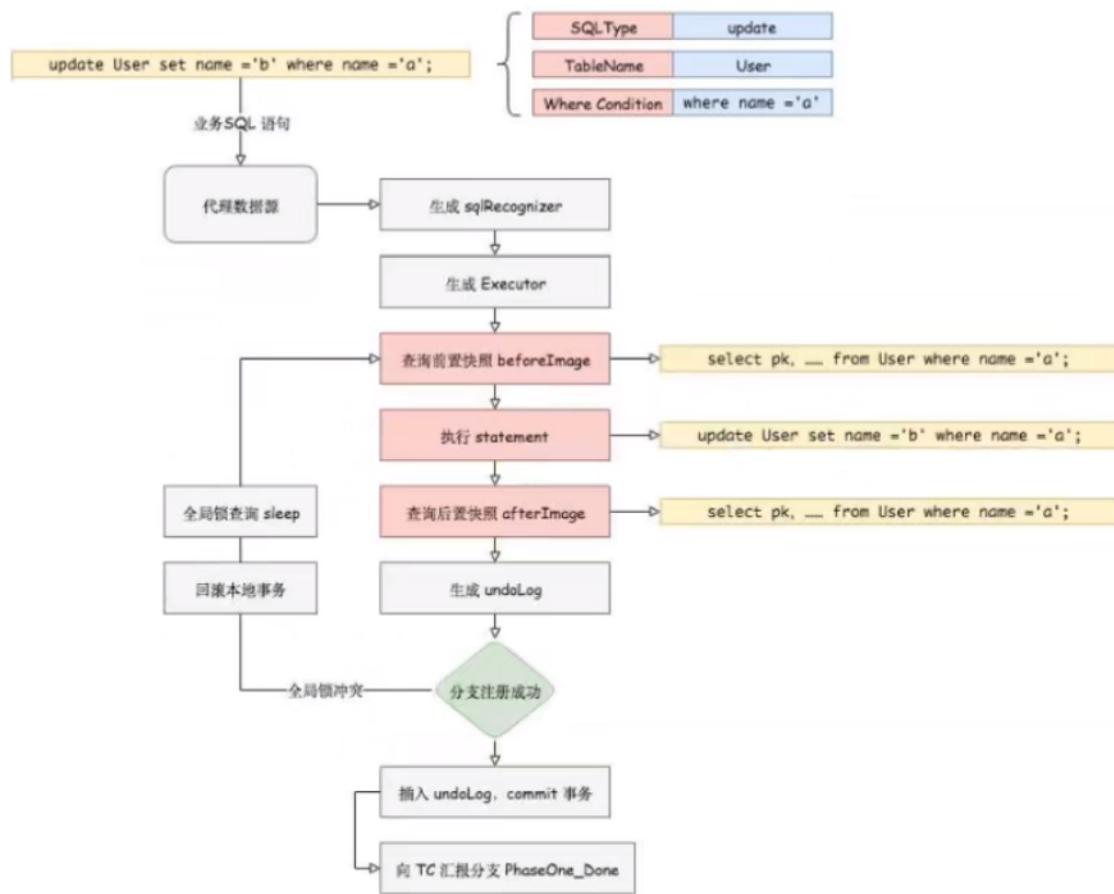
二阶段如果是回滚的话，Seata 就需要回滚一阶段已经执行的“业务SQL”，还原业务数据。

回滚方式便是用"before image"还原业务数据；但在还原前要首先要校验脏写，对比"数据库当前业务数据"和"after image"。

如果两份数据完全一致就说明没有脏写，可以还原业务数据，如果不一致就说明有脏写，出现脏写就需要转人工处理。



补充



雪花算法

为什么需要分布式全局唯一ID以及分布式ID的业务需求？集群高并发情况下如何保证分布式唯一全局Id生成？

在复杂分布式系统中，往往需要对大量的数据和消息进行唯一标识，如在美团点评的金融、支付、餐饮、酒店，猫眼电影等产品的系统中数据日渐增长，对数据分库分表后需要有一个唯一ID来标识一条数据或消息。特别一点的如订单、骑手、优惠券也都需要有唯一ID做标识。此时一个能够生成全局唯一ID的系统是非常必要的。

ID生成规则部分硬性要求

- 全局唯一：不能出现重复的ID号，既然是唯一-标识，这是最基本的要求
- 趋势递增：在MySQL的InnoDB引擎中使用的是聚集索引，由于多数RDBMS使用Btree的数据结构来存储索引数据，在主键的选择上面我们应该尽量使用有序的主键保证写入性能。
- 单调递增：保证下一个ID一定大于上一个ID，例如事务版本号、IM增量消息、排序等特殊需求
- 信息安全：如果ID是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定URL即可。如果是订单号就更危险了，竞对可以直接知道我们一天的单量。所以在一些应用场景下，需要ID无规则不规则，让竞争对手否好猜。
- 含时间戳：这样就能够在开发中快速了解这个分布式id的生成时间。

ID号生成系统的可用性要求

- 高可用：发一个获取分布式ID的请求，服务器就要保证99.999%的情况下给我创建一个唯一分布式ID。
- 低延迟：发一个获取分布式ID的请求，服务器就要快，极速。
- 高QPS：假如并发一口气10万个创建分布式ID请求同时杀过来，服务器要顶得住且一下子成功创建10万个分布式ID。

一般通用方案

UUID

UUID(Universally Unique Identifier)的标准型式包含32个16进制数字，以连了号分为五段，形式为8-4-4-4-12的36个字符，示例：550e8400-e29b-41d4-a716-446655440000

- 性能非常高：本地生成，没有网络消耗
- 如果只是考虑唯一性，那就选用它吧
- 但是，入数据库性能差

为什么无序的UUID会导致入库性能变差呢？

- 无序，无法预测他的生成顺序，不能生成递增有序的数字。首先分布式ID一般都会作为主键，但是安装MySQL官方推荐主键要尽量越短越好，UUID每一个都很长，所以不是很推荐。
- 主键，ID作为主键时在特定的环境会存在一些问题。比如做DB主键的场景下，UUID就非常不适用 MySQL官方有明确的建议主键要尽量越短越好36个字符长度的UUID不符合要求。
- 索引，既然分布式ID是主键，然后主键是包含索引的，然后MySQL的索引是通过B+树来实现的，每一次新的UUID数据的插入，为了查询的优化，都会对索引底层的B+树进行修改，因为UUID数据是无序的，所以每一次UUID数据的插入都会对主键地械的B+树进行很大的修改，这一点很不好。插入完全无序，不但会导致一些中间节点产生分裂，也会白白创造出很多不饱和的节点，这样大大降低了数据库插入的性能。

All indexes other than the clustered index are known as [secondary indexes](#). In [InnoDB](#), each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. [InnoDB](#) uses this primary key value to search for the row in the clustered index.

If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.

[link](#)

数据库自增主键

单机

在单机里面，数据库的自增ID机制的主要原理是：数据库自增ID和MySQL数据库的replace into实现的。

REPLACE INTO的含义是插入一条记录，如果表中唯一索引的值遇到冲突，则替换老数据。

这里的replace into跟insert功能类似，不同点在于：replace into首先尝试插入数据列表中，如果发现表中已经有此行数据（根据主键或唯一索引判断）则先删除，再插入。否则直接插入新数据。

```
1 CREATE TABLE t_test(
2     id BIGINT(20) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
3     stub CHAR(1) NOT NULL DEFAULT '',
4     UNIQUE KEY stub(stub)
5 )
6
7 SELECT * FROM t_test;
8
9 REPLACE INTO t_test (stub) VALUES('b');
10
11 SELECT LAST_INSERT_ID();
```

集群分布式

那数据库自增ID机制适合作分布式ID吗？答案是不太适合

1: 系统水平扩展比较困难，比如定义好了步长和机器台数之后，如果要添加机器该怎么做？假设现在只有一台机器发号是1, 2, 3, 4, 5（步长是1），这个时候需要扩容机器一台。可以这样做：把第二台机器的初始值设置得比第一台超过很多，貌似还好，现在想象一下如果我们线上有100台机器，这个时候要扩容该怎么做？简直是噩梦，所以系统水平扩展方案复杂难以实现。

2: 数据库压力还是很大，每次获取ID都得读写一次数据库，非常影响性能，不符合分布式ID里面的延迟低和要高QPS的规则（在高并发下，如果都去数据库里面获取id，那是非常影响性能的）

基于Redis生成全局ID策略

因为Redis是单线程的天生保证原子性，可以使用原子操作INCR和INCRBY来实现

注意：在Redis集群情况下，同样和MySQL一样需要设置不同的增长步长，同时key一定要设置有效期可以使用Redis集群来获取更高的吞吐量。

假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。

各个Redis生成的ID为：

A: 1, 6, 11, 16, 21

B: 2, 7, 12, 17, 22

C: 3, 8, 13, 18, 23

D: 4, 9, 14, 19, 24

E: 5, 10, 15, 20, 25

Twitter的分布式自增ID算法snowflake

概述

Twitter的snowflake解决了这种需求，最初Twitter把存储系统从MySQL迁移到Cassandra（由Facebook开发一套开源分布式NoSQL数据库系统）。因为Cassandra没有顺序ID生成机制，所以开发了这样一套全局唯一生成服务。

Twitter的分布式雪花算法SnowFlake，经测试snowflake 每秒能够产生26万个自增可排序的ID

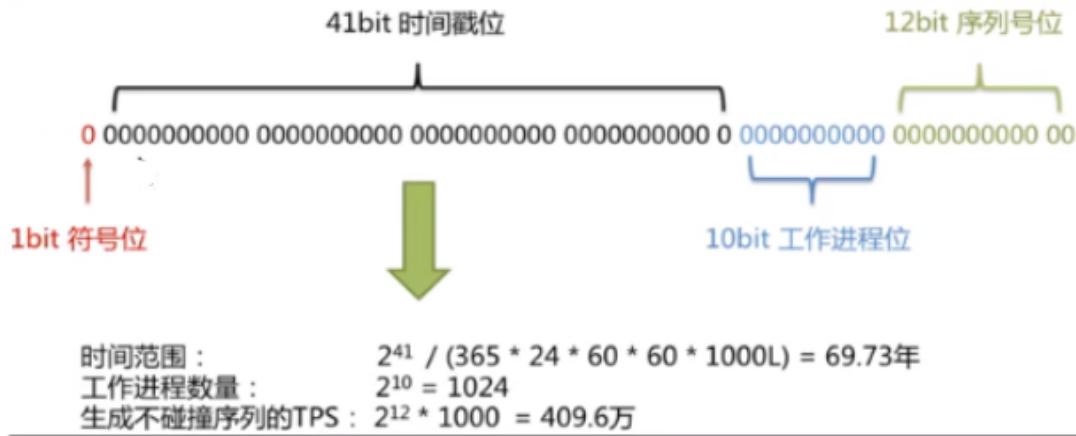
1. Twitter的SnowFlake生成ID能够按照时间有序生成。
2. SnowFlake算法生成ID的结果是一个64bit大小的整数，为一个Long型（转换成字符串后长度最多19）。
3. 分布式系统内不会产生ID碰撞（由datacenter和workerId作区分）并且效率较高。

分布式系统中，有一些需要使用全局唯一ID的场景，生成ID的基本要求：

1. 在分布式的环境下必须全局且唯一。
2. 一般都需要单调递增，因为一般唯一ID都会存到数据库，而Innodb的特性就是将内容存储在主键索引树上的叶子节点而且是从左往右，递增的，所以考虑到数据库性能，一般生成的ID也最好是单调递增。为了防止ID冲突可以使用36位的UUID，但是UUID有一些缺点，首先他相对比较长，另外UUID一般是无序的。
3. 可能还会需要无规则，因为如果使用唯一ID作为订单号这种，为了不然别人知道一天的订单量是多少，就需要这个规则。

结构

雪花算法的几个核心组成部分：



在Java中64bit的整数是long类型，所以在SnowFlake算法生成的id就是long来存储的。

字段解析：

1bit:

不用，因为二进制中最高位是符号位，1表示负数，0表示正数。生成的id一般都是用整数，所以最高位固定为0。

41bit - 时间戳，用来记录时间戳，毫秒级：

- 41位可以表示 $2^{41} - 1$ 个数字
- 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0至 $2^{41} - 1$, **减1是因为可表示的数值范围是从0开始算的，而不是1。**
- 也就是说41位可以表示 $2^{41} - 1$ 个毫秒的值，转化成单位年则是 $(2^{41} - 1 - 1) \% (1000 * 60 * 60 * 24 * 365) = 69\text{年}$

10bit - 工作机器ID，用来记录工作机器ID：

- 可以部署在 $2^{10} = 1024$ 个节点，包括5位DataCenterId和WorkerId。
- 5位(bit)可以表示的最大正整数是 $2^5 - 1 = 31$ ，即可以用0、1、2、3... 31这32个数字，来表示不同的DataCenterId和WorkerId。

12bit - 序列号，用来记录同毫秒内产生的不同ID。

- 12位(bit)可以表示的最大正整数是 $2^{12} - 1 = 4095$ ，即可以用0、1、2、3... 4095这4095个数字，来表示同一机器同一时间截(毫秒)内产生的4095个ID序号。

SnowFlake可以保证：

- 所有生成的ID按时间趋势递增。
- 整个分布式系统内不会产生重复id（因为有DataCenterId和WorkerId来做区分）

源码

以下代码仅供学习：

```

1 /**
2  * Twitter_Snowflake
3  * SnowFlake的结构如下(每部分用-分开):
4  * 0 - 0000000000 0000000000 0000000000 0 - 00000 - 00000 -
5  * 000000000000
6  * 1位标识, 由于long基本类型在Java中是带符号的, 最高位是符号位, 正数是0, 负数是1, 所以
7  * id一般是正数, 最高位是0
8  * 41位时间戳(毫秒级), 注意, 41位时间戳不是存储当前时间的时间戳, 而是存储时间戳的差值
9  * (当前时间戳 - 开始时间戳)

```

```
7  * 得到的值），这里的开始时间戳，一般是我们的id生成器开始使用的时间，由我们程序来指定的  
8  （如下程序SnowflakeIdworker类的startTime属性）。41位的时间戳，可以使用69年，年T =  
9  (1L << 41) / (1000L * 60 * 60 * 24 * 365) = 69  
10 * 10位的数据机器位，可以部署在1024个节点，包括5位datacenterId和5位workerId  
11 * 12位序列，毫秒内的计数，12位的计数顺序号支持每个节点每毫秒（同一机器，同一时间戳）产生  
12 4096个ID序号  
13 * 加起来刚好64位，为一个Long型。  
14 */  
15 public class SnowflakeIdworker {  
16     /** 开始时间戳 (2015-01-01) */  
17     private final long twepoch = 1420041600000L;  
18  
19     /** 机器id所占的位数 */  
20     private final long workerIdBits = 5L;  
21  
22     /** 数据标识id所占的位数 */  
23     private final long datacenterIdBits = 5L;  
24  
25     /** 支持的最大机器id，结果是31（这个移位算法可以很快的计算出几位二进制数所能表示的  
26     最大十进制数） */  
27     private final long maxWorkerId = -1L ^ (-1L << workerIdBits);  
28  
29     /** 支持的最大数据标识id，结果是31 */  
30     private final long maxDatacenterId = -1L ^ (-1L << datacenterIdBits);  
31  
32     /** 序列在id中占的位数 */  
33     private final long sequenceBits = 12L;  
34  
35     /** 机器ID向左移12位 */  
36     private final long workerIdShift = sequenceBits;  
37  
38     /** 数据标识id向左移17位(12+5) */  
39     private final long datacenterIdShift = sequenceBits + workerIdBits;  
40  
41     /** 时间戳向左移22位(5+5+12) */  
42     private final long timestampLeftShift = sequenceBits + workerIdBits +  
43     datacenterIdBits;  
44  
45     /** 生成序列的掩码，这里为4095 (0b111111111111=0xffff=4095) */  
46     private final long sequenceMask = -1L ^ (-1L << sequenceBits);  
47  
48     /** 工作机器ID(0~31) */  
49     private long workerId;  
50  
51     /** 数据中心ID(0~31) */  
52     private long datacenterId;  
53  
54     /** 毫秒内序列(0~4095) */  
55     private long sequence = 0L;  
56  
57     /** 上次生成ID的时间戳 */  
58     private long lastTimestamp = -1L;  
59  
59 //=====Constructors=====  
60 ======  
61     /**  
62      * 构造函数  
63     */
```

```
58     * @param workerId 工作ID (0~31)
59     * @param datacenterId 数据中心ID (0~31)
60     */
61     public SnowflakeIdWorker(long workerId, long datacenterId) {
62         if (workerId > maxWorkerId || workerId < 0) {
63             throw new IllegalArgumentException(String.format("worker Id
can't be greater than %d or less than 0", maxWorkerId));
64         }
65         if (datacenterId > maxDatacenterId || datacenterId < 0) {
66             throw new IllegalArgumentException(String.format("datacenter Id
can't be greater than %d or less than 0", maxDatacenterId));
67         }
68         this.workerId = workerId;
69         this.datacenterId = datacenterId;
70     }
71
72     /**
73      * 获得下一个ID (该方法是线程安全的)
74      * @return SnowflakeId
75      */
76     public synchronized long nextId() {
77         long timestamp = timeGen();
78
79         //如果当前时间小于上一次ID生成的时间戳，说明系统时钟回退过这个时候应当抛出异常
80         if (timestamp < lastTimestamp) {
81             throw new RuntimeException(
82                 String.format("Clock moved backwards. Refusing to
generate id for %d milliseconds", lastTimestamp - timestamp));
83         }
84
85         //如果是同一时间生成的，则进行毫秒内序列
86         if (lastTimestamp == timestamp) {
87             sequence = (sequence + 1) & sequenceMask;
88             //毫秒内序列溢出
89             if (sequence == 0) {
90                 //阻塞到下一个毫秒，获得新的时间戳
91                 timestamp = tilNextMillis(lastTimestamp);
92             }
93         }
94         //时间戳改变，毫秒内序列重置
95         else {
96             sequence = 0L;
97         }
98
99         //上次生成ID的时间戳
100        lastTimestamp = timestamp;
101
102        //移位并通过或运算拼到一起组成64位的ID
103        return ((timestamp - twepoch) << timestampLeftShift) ||
104            | (datacenterId << datacenterIdShift) ||
105            | (workerId << workerIdShift) ||
106            | sequence;
107    }
108
109    /**
110     *
```

```

111     * 阻塞到下一个毫秒，直到获得新的时间戳
112     * @param lastTimestamp 上次生成ID的时间戳
113     * @return 当前时间戳
114     */
115     protected long tilNextMillis(long lastTimestamp) {
116         long timestamp = timeGen();
117         while (timestamp <= lastTimestamp) {
118             timestamp = timeGen();
119         }
120         return timestamp;
121     }
122
123 /**
124 * 返回以毫秒为单位的当前时间
125 * @return 当前时间(毫秒)
126 */
127 protected long timeGen() {
128     return System.currentTimeMillis();
129 }
130
131 /**
132 public static void main(String[] args) {
133     System.out.println("开始: "+System.currentTimeMillis());
134     SnowflakeIdworker idworker = new SnowflakeIdworker(0, 0);
135     for (int i = 0; i < 50; i++) {
136         long id = idworker.nextId();
137         System.out.println(id);
138         System.out.println(Long.toBinaryString(id));
139     }
140     System.out.println("结束: "+System.currentTimeMillis());
141 }
142 }
```

工程落地经验

[Hutool的Snowflake文档](#)

添加依赖

```

1 <dependency>
2     <groupId>cn.hutool</groupId>
3     <artifactId>hutool-captcha</artifactId>
4     <version>7.2.2</version>
5 </dependency>
```

示例程序:

```

1 import cn.hutool.core.lang.Snowflake;
2 import cn.hutool.core.net.NetUtil;
3 import cn.hutool.core.util.IdUtil;
4 import lombok.extern.slf4j.Slf4j;
5 import org.springframework.stereotype.Component;
6
7 import javax.annotation.PostConstruct;
8
9 @Slf4j
10 @Component
```

```

11 public class IdGeneratorSnowflake{
12     private long workerId = 0;
13     private long datacenterId = 1;
14     private Snowflake snowflake = IdUtil.createSnowflake(workerId,
15     datacenterId);
16
17     public synchronized long snowflakeId(){
18         return snowflake.nextId();
19     }
20
21     public synchronized long snowflakeId(long workerId, long datacenterId){
22         Snowflake snowflake = IdUtil.createSnowflake(workerId,
23         datacenterId);
24         return snowflake.nextId();
25     }
26
27     public static void main(String[] args){
28         IdGeneratorSnowflake idGenerator = new IdGeneratorSnowflake();
29         System.out.println(idGenerator.snowflakeId());
30
31         ExecutorService threadPool = Executors.newFixedThreadPool(5);
32         for (int i = 1; i <= 20; i++){
33             threadPool.submit(() -> {
34                 System.out.println(idGenerator.snowflakeId());
35             });
36         }
37
38     }
39 }
40

```

优缺点

- 优点：
 - 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。
 - 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的。
 - 可以根据自身业务特性分配bit位，非常灵活。
- 缺点：
 - 依赖机器时钟，如果机器时钟回拨，会导致重复ID生成。
 - 在单机上是递增的，但是由于设计到分布式环境，每台机器上的时钟不可能完全同步，有时候会出现不是全局递增的情况。（此缺点可以认为无所谓，一般分布式ID只要求趋势递增，并不会严格要求递增，90%的需求都只要求趋势递增）

其他补充

百度开源的分布式唯一ID生成器UidGenerator

美团点评分布式ID生成系统Leaf

Spring Cloud组件总结

组件	简介	分类	官网	笔记	备注
Eureka	Eureka is the Netflix Service Discovery Server and Client.	服务注册中心	link	link	eureka中文解释: int.(因找到某物, 尤指问题的答案而高兴)我发现了, 我找到了
Zookeeper	ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.	服务注册中心	link	link	zookeeper中文解释: n.动物园管理员
Consul	Consul is a service mesh solution providing a full featured control plane with service discovery, configuration, and segmentation functionality.	服务注册中心	link	link	consul中文解释: n.领事
Ribbon	Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients.	服务调用	link	link	ribbon中文解释: n.(用于捆绑或装饰的)带子;丝带;带状物;
OpenFeign	Feign is a declarative web service client. It makes writing web service clients easier.	服务调用	link	link	feign中文意思: v.假装, 装作, 佯装(有某种感觉或生病、疲倦等)
Hystrix	Netflix has created a library called Hystrix that implements the circuit breaker pattern.	服务降级	link	link	hystrix中文意思: n.豪猪属; 猬属;豪猪;豪猪亚属
GateWay	Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.	服务网关	link	link	gateway中文意思: n.网关; 途径;门道;手段
Config	Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system.	服务配置	link	link	-

组件	简介	分类	官网	笔记	备注
Bus	Spring Cloud Bus links nodes of a distributed system with a lightweight message broker.	服务总线	link	link	-
Stream	Spring Cloud Stream is a framework for building message-driven microservice applications.	消息队列	link	link	-
Sleuth	Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud.	服务跟踪	link	link	sleuth中文意思：n.侦探
Nacos	Nacos致力于帮助您发现、配置和管理微服务。	服务注册中心、服务配置、服务总线	link	link	NAme + COnfiguration + Service
Sentinel	Sentinel是面向分布式服务架构的流量控制组件，主要以流量为切入点，从流量控制、熔断降级、系统自适应保护等多个维度来帮助您保障微服务的稳定性。	服务降级	link	link	sentinel中文意思：n.哨兵
Seata	Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。	分布式事务	link	link	-