

参考视频: https://www.bilibili.com/video/BV18E411x7eT?spm_id_from=333.999.0.0

技术要求:

java8+maven+git、github+Nginx+RabbitMQ+Spring Boot2.0

一、SpringCloud 简介

springcloud官网: <https://spring.io/cloud>

springcloud官方文档: <https://spring.io/projects/spring-cloud>

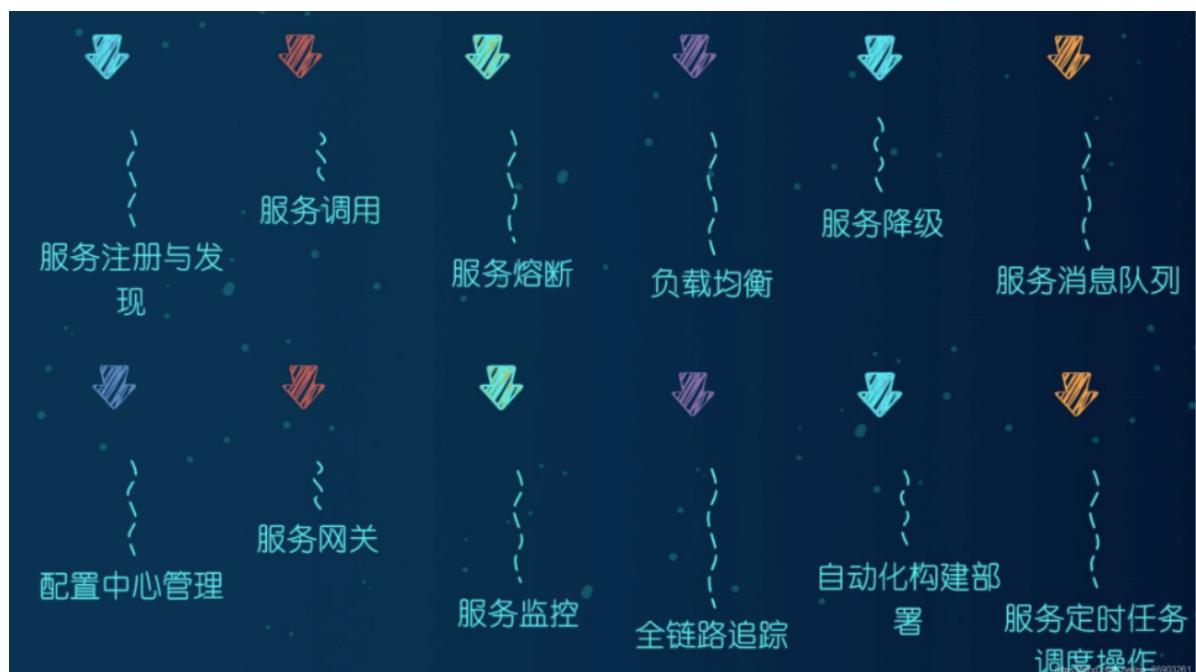
springcloud官方文档 (Hoxton SR5) : <https://cloud.spring.io/spring-cloud-static/Hoxton.SR5/reference/htmlsingle/>

springcloud中文文档: <https://www.springcloud.cc/>

springcloud中国社区文档: <http://docs.springcloud.cn/>

<https://www.bookstack.cn/read/spring-cloud-docs/docs-index.md>

**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，俗称微服务全家桶**



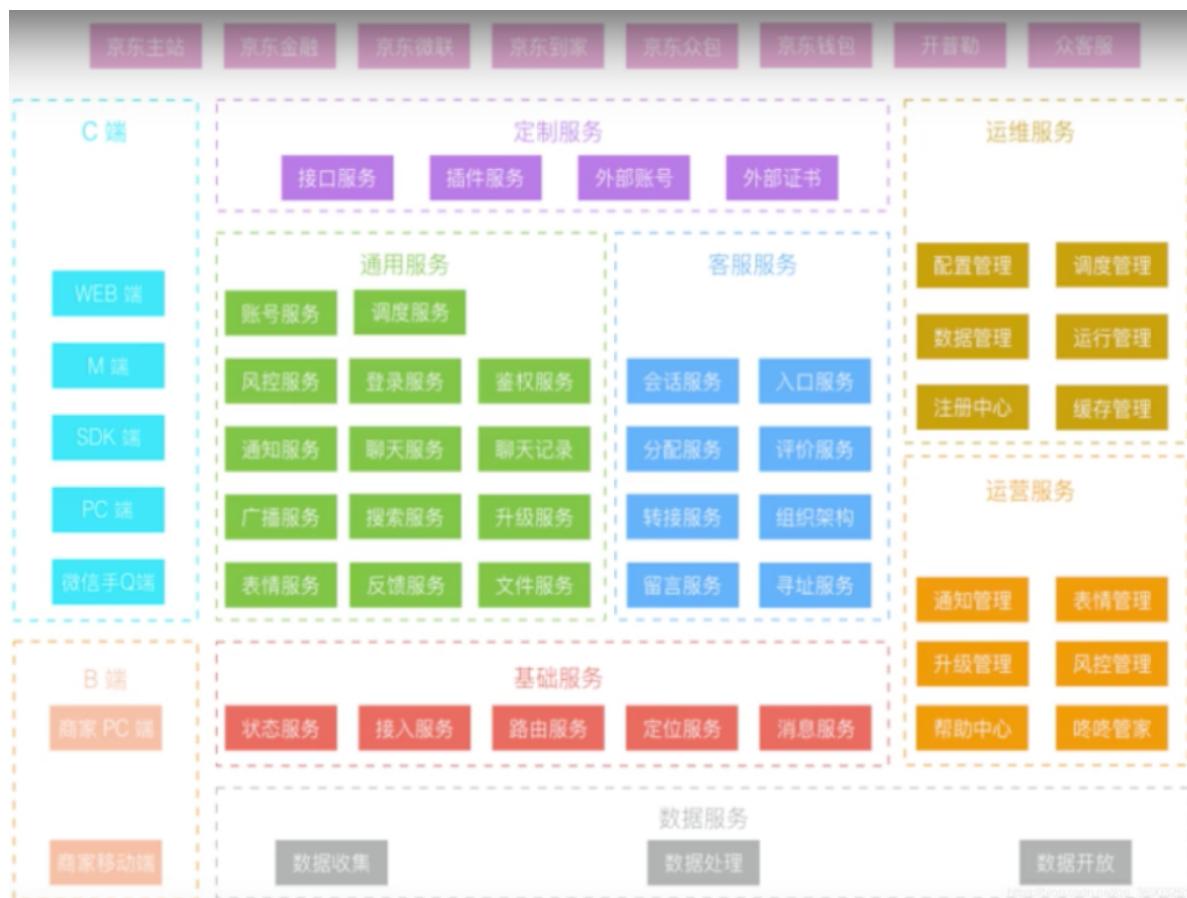
京东的促销节架构:



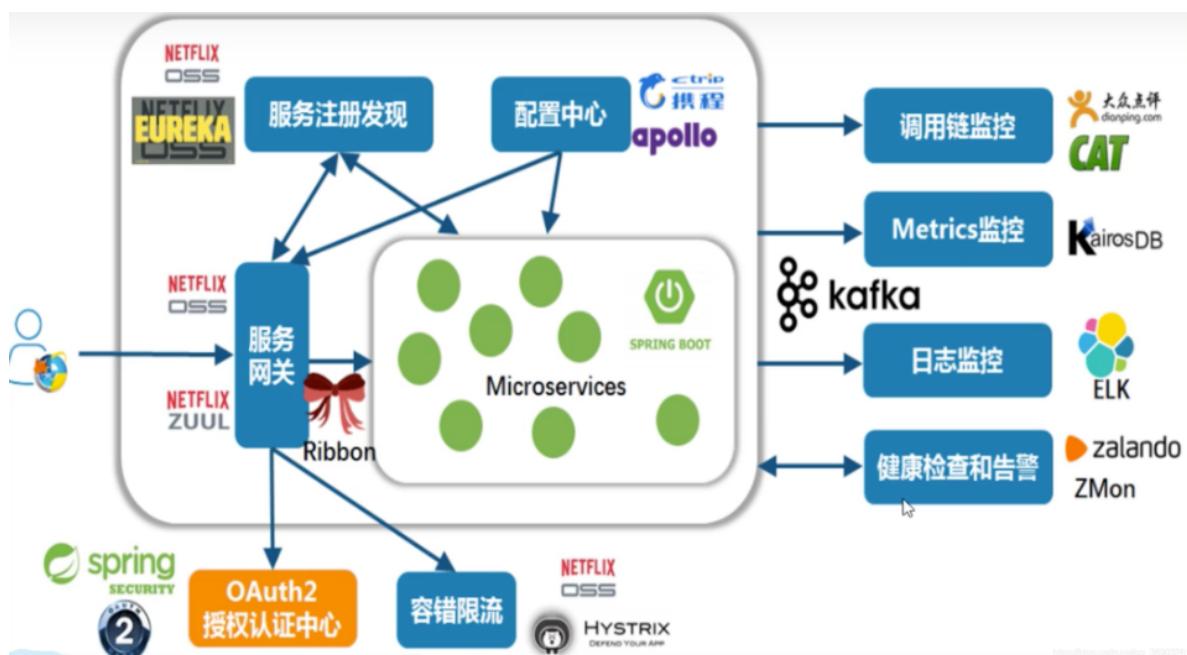
阿里的架构：



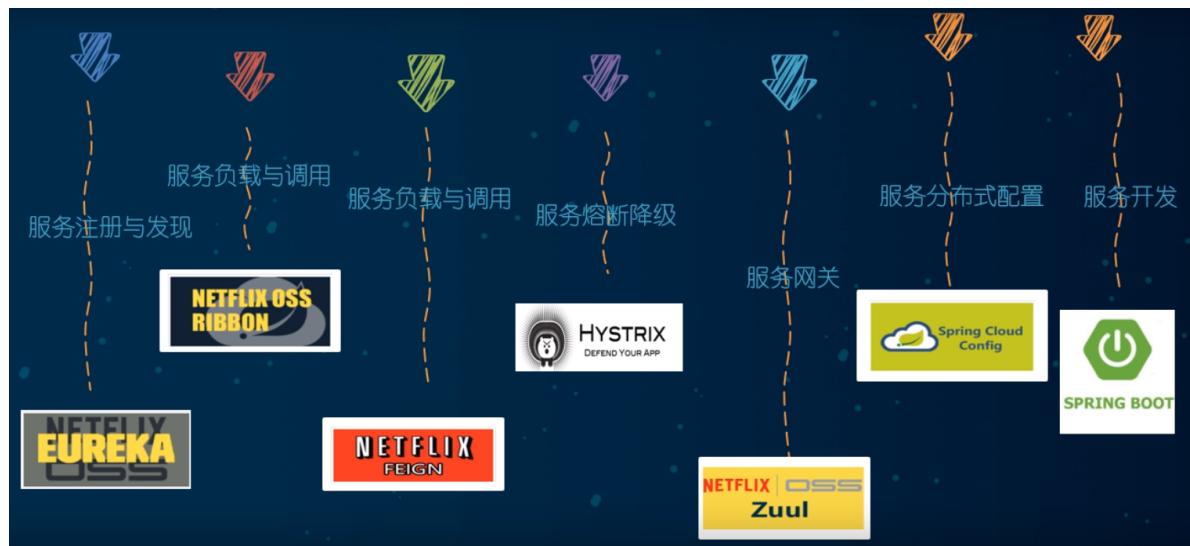
京东物流的架构：



SpringCloud技术栈



主流技术：



- 服务注册与发现
- 服务负载与调用
- 服务熔断降级
- 服务网关
- 服务分布式配置
- 服务开发

SpringCloud与SpringBoot版本的对应关系

尚硅谷使用：SpringBoot2.x版 和 SpringCloud H版

查看SpringCloud与SpringBoot版本对应关系的网址：<https://spring.io/projects/spring-cloud>

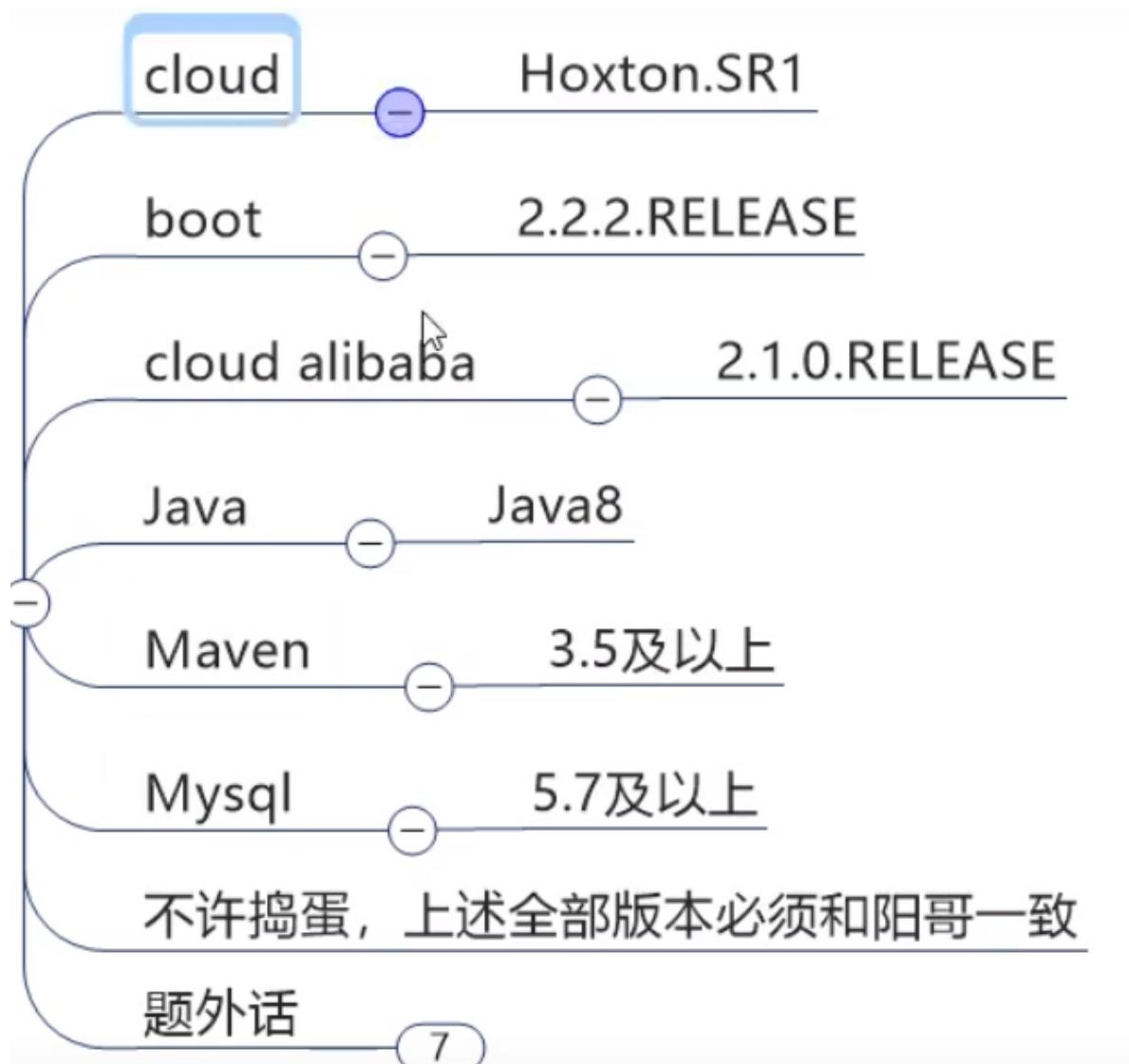
Table 1. Release train Spring Boot compatibility

Release Train	Boot Version
2021.0.x aka Jubilee	2.6.x
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

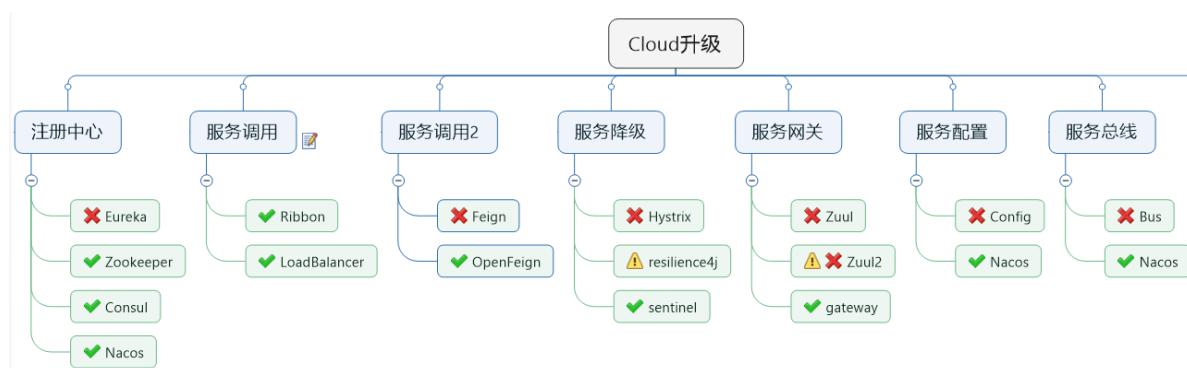
更详细的版本对应查看方法：<https://start.spring.io/actuator/info>

```
        },
      - bom-ranges: {
        - codecentric-spring-boot-admin: {
          2.4.3: "Spring Boot >=2.3.0.M1 and <2.5.0-M1",
          2.5.6: "Spring Boot >=2.5.0.M1 and <2.6.0-M1",
          2.6.5: "Spring Boot >=2.6.0.M1 and <2.7.0-M1"
        },
        - solace-spring-boot: {
          1.1.0: "Spring Boot >=2.3.0.M1 and <2.6.0-M1",
          1.2.1: "Spring Boot >=2.6.0.M1 and <2.7.0-M1"
        },
        - solace-spring-cloud: {
          1.1.1: "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
          2.1.0: "Spring Boot >=2.4.0.M1 and <2.6.0-M1",
          2.3.0: "Spring Boot >=2.6.0.M1 and <2.7.0-M1"
        },
        - spring-cloud: {
          Hoxton.SR12: "Spring Boot >=2.2.0.RELEASE and <2.4.0.M1",
          2020.0.5: "Spring Boot >=2.4.0.M1 and <2.6.0-M1",
          2021.0.0-M1: "Spring Boot >=2.6.0-M1 and <2.6.0-M3",
          2021.0.0-M3: "Spring Boot >=2.6.0-M3 and <2.6.0-RC1",
          2021.0.0-RC1: "Spring Boot >=2.6.0-RC1 and <2.6.1",
          2021.0.1: "Spring Boot >=2.6.1 and <2.6.7-SNAPSHOT",
          2021.0.2-SNAPSHOT: "Spring Boot >=2.6.7-SNAPSHOT and <3.0.0-M1",
          2022.0.0-M1: "Spring Boot >=3.0.0-M1 and <3.0.0-M2",
          2022.0.0-M2: "Spring Boot >=3.0.0-M2 and <3.1.0-M1"
        },
        - spring-cloud-azure: {
          4.0.0: "Spring Boot >=2.5.0.M1 and <2.7.0-M1"
        },
        - spring-cloud-gcp: {
          2.0.10: "Spring Boot >=2.4.0-M1 and <2.6.0-M1",
          3.2.1: "Spring Boot >=2.6.0-M1 and <2.7.0-M1"
        },
        - spring-cloud-services: {
          2.3.0.RELEASE: "Spring Boot >=2.3.0.RELEASE and <2.4.0-M1",
          ...
        }
      }
    }
  }
}
```

尚硅谷使用的版本：



关于Cloud各种组件的停更/升级/替换



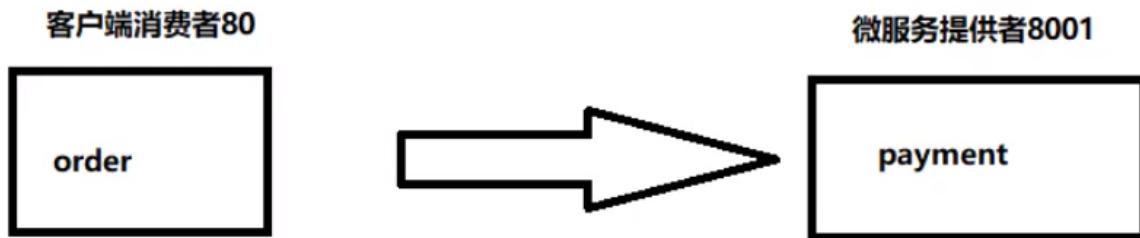
建议多看看官方文档：

springcloud官方文档 (Hoxton SR5) : <https://cloud.spring.io/spring-cloud-static/Hoxton.SR5/reference/htmlsingle/>

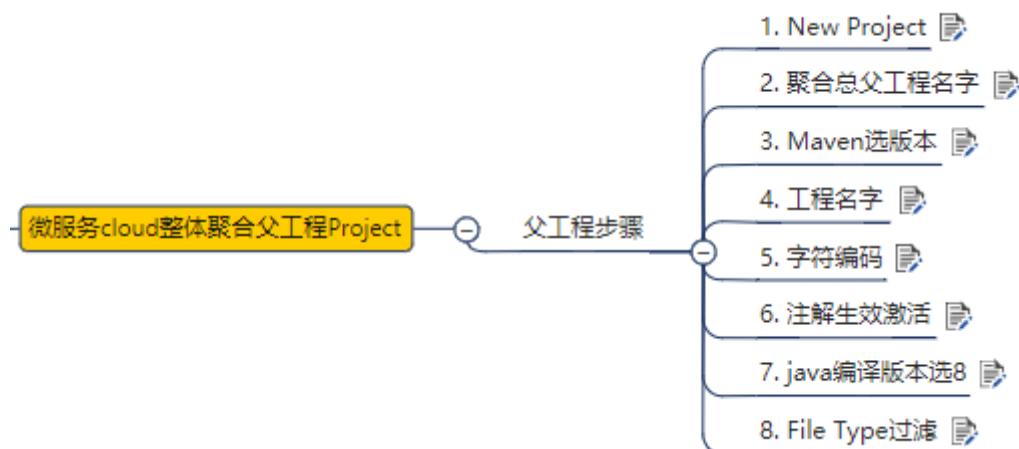
springcloud中文文档: <https://www.springcloud.cc/>

二、微服务架构编码构建

做一个最简单的 订单-支付模块微服务

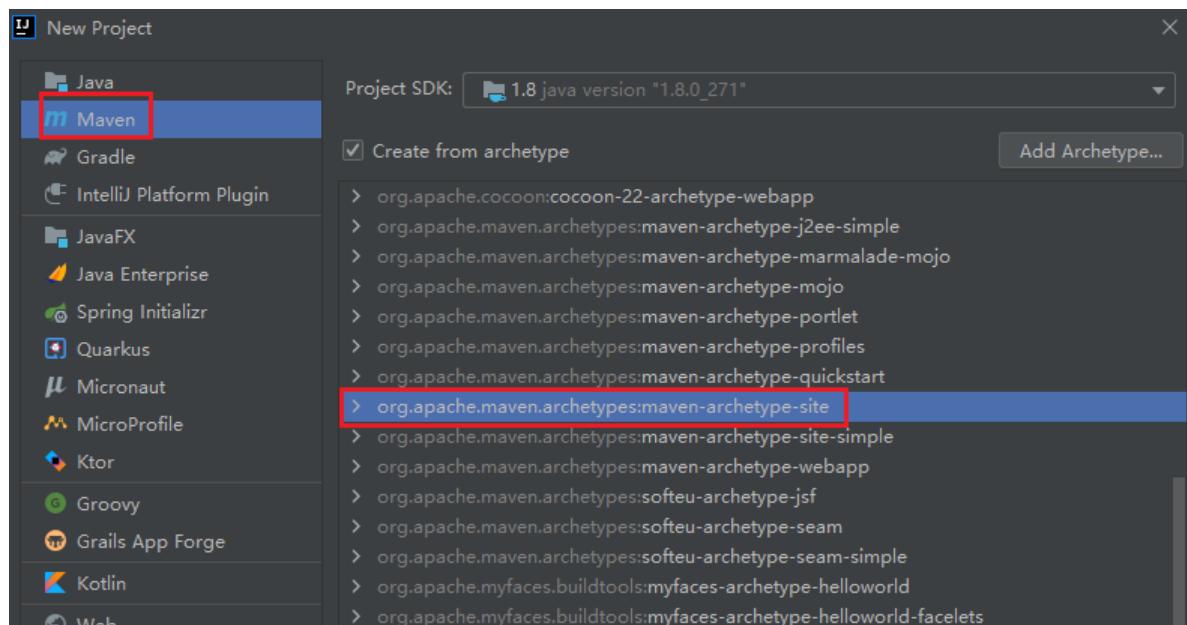


大概有以下步骤：

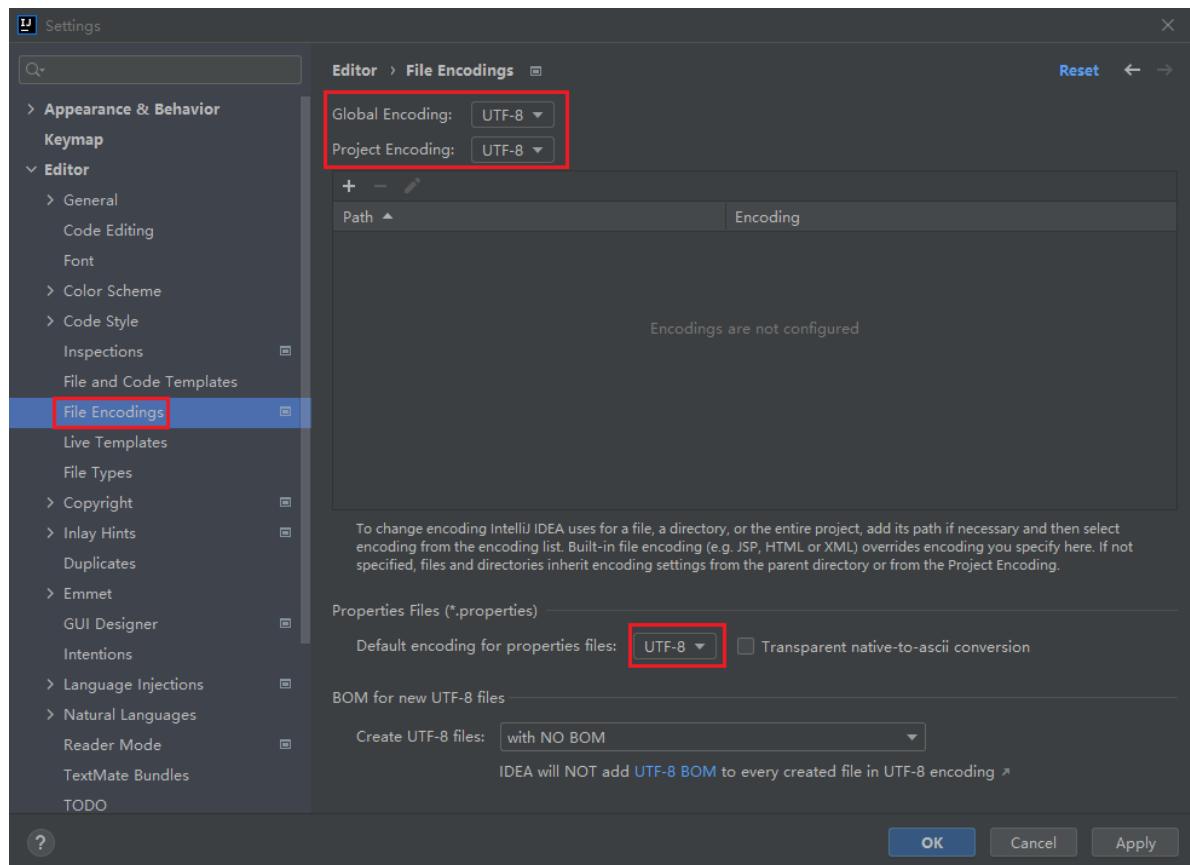


构建一个父工程

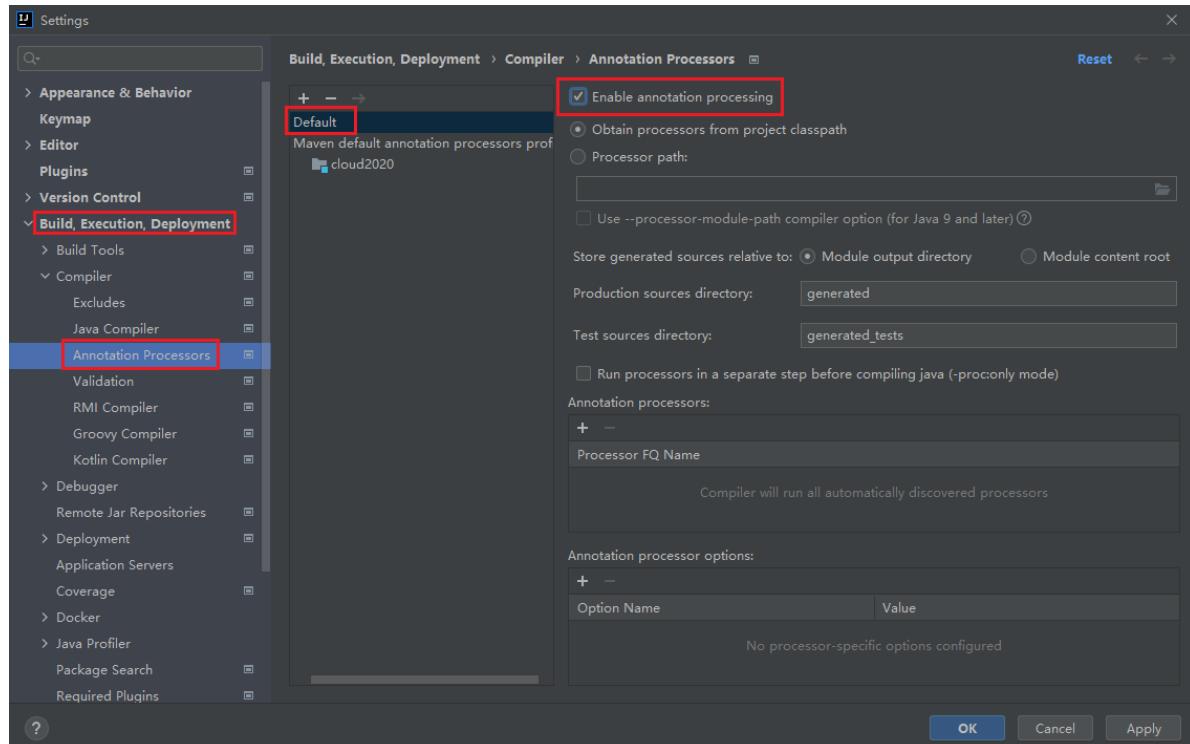
1. 新建父工程



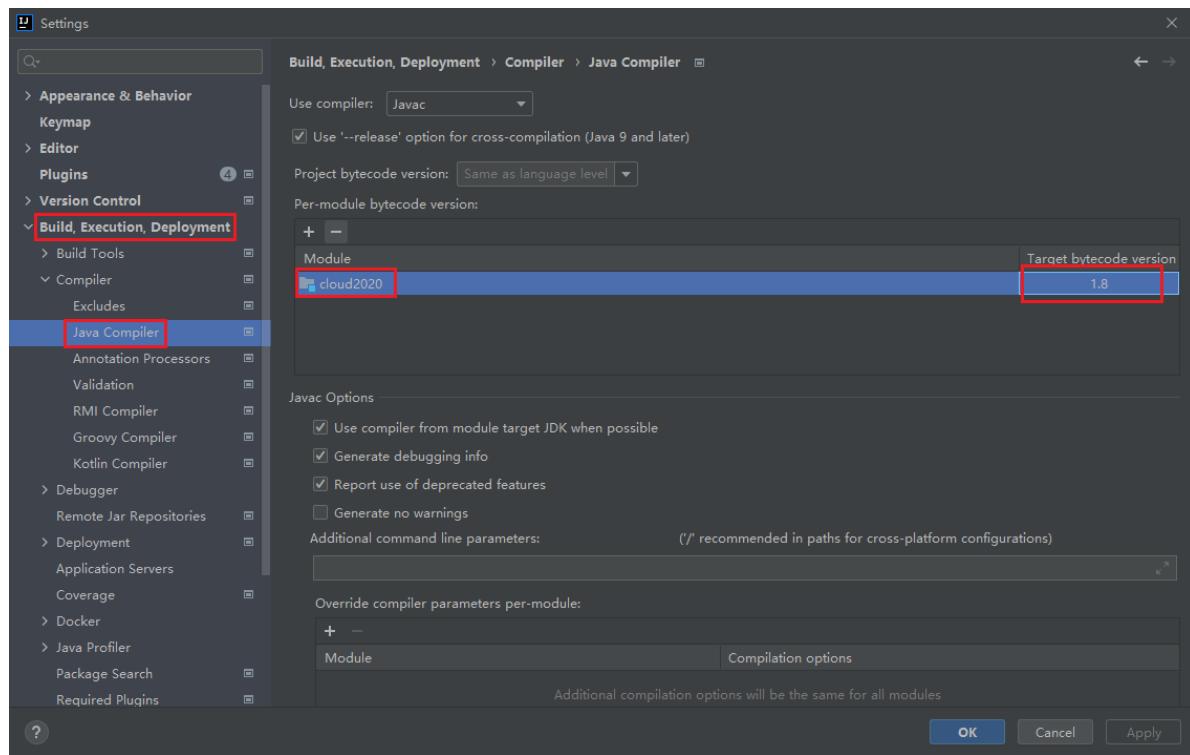
5. 字符编码，都选择 UTF-8



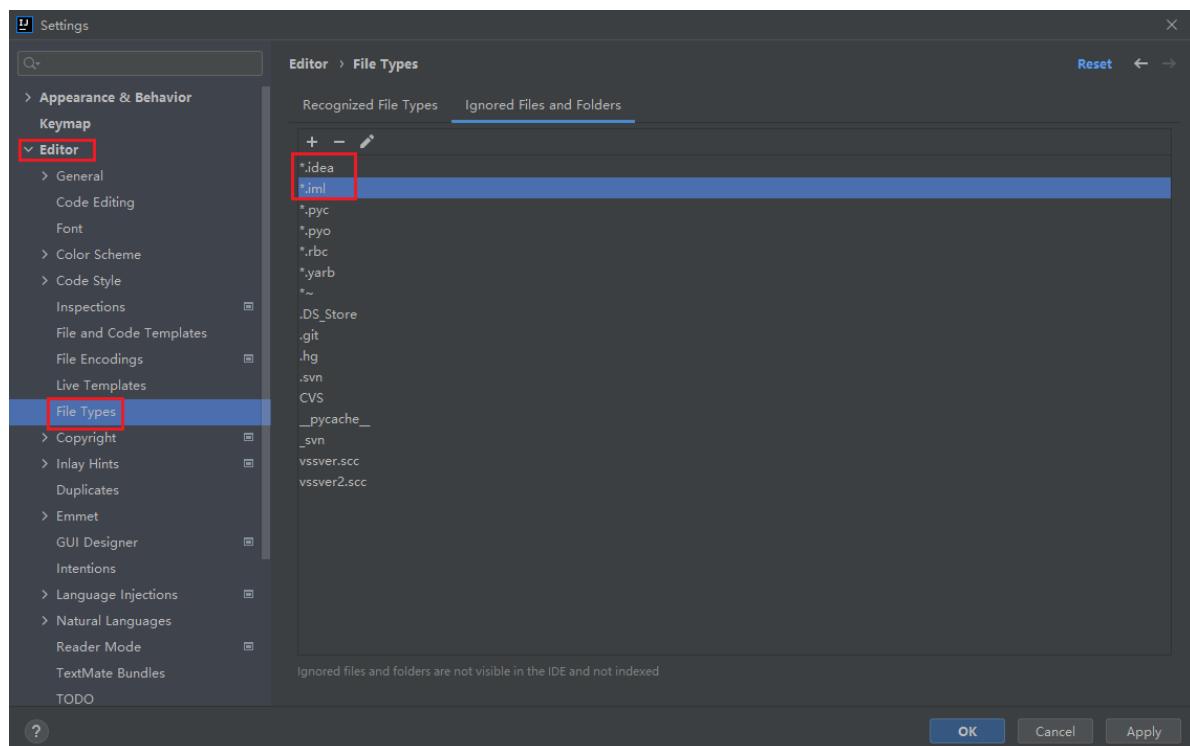
6. 注解生效激活



7. Java 编译版本



8. File Type 过滤



父工程pom文件

```
1 <groupId>com.atguigu.springcloud</groupId>
2 <artifactId>cloud2020</artifactId>
3 <version>1.0-SNAPSHOT</version>
4 <packaging>pom</packaging>
5
```

```
6   <!-- 统一管理jar包版本 -->
7   <properties>
8     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
9     <maven.compiler.source>1.8</maven.compiler.source>
10    <maven.compiler.target>1.8</maven.compiler.target>
11    <junit.version>4.12</junit.version>
12    <log4j.version>1.2.17</log4j.version>
13    <lombok.version>1.18.10</lombok.version>
14    <mysql.version>5.1.47</mysql.version>
15    <druid.version>1.1.17</druid.version>
16    <mybatis.spring.boot.version>1.3.1</mybatis.spring.boot.version>
17  </properties>
18
19  <!-- 子模块继承之后，提供作用：锁定版本+子modlue不用写groupId和version -->
20  <dependencyManagement>
21    <dependencies>
22      <!--spring boot 2.2.2-->
23      <dependency>
24        <groupId>org.springframework.boot</groupId>
25        <artifactId>spring-boot-dependencies</artifactId>
26        <version>2.2.2.RELEASE</version>
27        <type>pom</type>
28        <scope>import</scope>
29      </dependency>
30      <!--spring cloud Hoxton.SR1-->
31      <dependency>
32        <groupId>org.springframework.cloud</groupId>
33        <artifactId>spring-cloud-dependencies</artifactId>
34        <version>Hoxton.SR1</version>
35        <type>pom</type>
36        <scope>import</scope>
37      </dependency>
38      <!--spring cloud alibaba 2.1.0.RELEASE-->
39      <dependency>
40        <groupId>com.alibaba.cloud</groupId>
41        <artifactId>spring-cloud-alibaba-dependencies</artifactId>
42        <version>2.1.0.RELEASE</version>
43        <type>pom</type>
44        <scope>import</scope>
45      </dependency>
46      <dependency>
47        <groupId>mysql</groupId>
48        <artifactId>mysql-connector-java</artifactId>
49        <version>${mysql.version}</version>
50      </dependency>
51      <dependency>
52        <groupId>com.alibaba</groupId>
53        <artifactId>druid</artifactId>
54        <version>${druid.version}</version>
55      </dependency>
56      <dependency>
57        <groupId>org.mybatis.spring.boot</groupId>
58        <artifactId>mybatis-spring-boot-starter</artifactId>
59        <version>${mybatis.spring.boot.version}</version>
60      </dependency>
61      <dependency>
62        <groupId>junit</groupId>
63        <artifactId>junit</artifactId>
```

```
64          <version>${junit.version}</version>
65      </dependency>
66      <dependency>
67          <groupId>log4j</groupId>
68          <artifactId>log4j</artifactId>
69          <version>${log4j.version}</version>
70      </dependency>
71      <dependency>
72          <groupId>org.projectlombok</groupId>
73          <artifactId>lombok</artifactId>
74          <version>${lombok.version}</version>
75          <optional>true</optional>
76      </dependency>
77  </dependencies>
78 </dependencyManagement>
79
80 <build>
81     <plugins>
82         <plugin>
83             <groupId>org.springframework.boot</groupId>
84             <artifactId>spring-boot-maven-plugin</artifactId>
85             <configuration>
86                 <fork>true</fork>
87                 <addResources>true</addResources>
88             </configuration>
89         </plugin>
90     </plugins>
91 </build>
```

注：这里有部分版本号根据本地maven有的稍作了一些调整。

dependencyManagement 和 dependencies 的区别：

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dependencyManagement 元素。

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。

举例：

例如在父项目里：

Xml代码 ★ ★

```
1. <dependencyManagement>
2.   <dependencies>
3.     <dependency>
4.       <groupId>mysql</groupId>
5.       <artifactId>mysql-connector-java</artifactId>
6.       <version>5.1.2</version>
7.     </dependency>
8.     ...
9.   <dependencies>
10.  </dependencyManagement>
```

然后在子项目里就可以添加mysql-connector时可以不指定版本号，例如：

Xml代码 ★ ★

```
1. <dependencies>
2.   <dependency>
3.     <groupId>mysql</groupId>
4.     <artifactId>mysql-connector-java</artifactId>
5.   </dependency>
6. </dependencies>
```

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。
- 如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且version和scope都读取自父pom；
- 如果子项目中指定了版本号，那么会使用子项目中指定的jar版本。

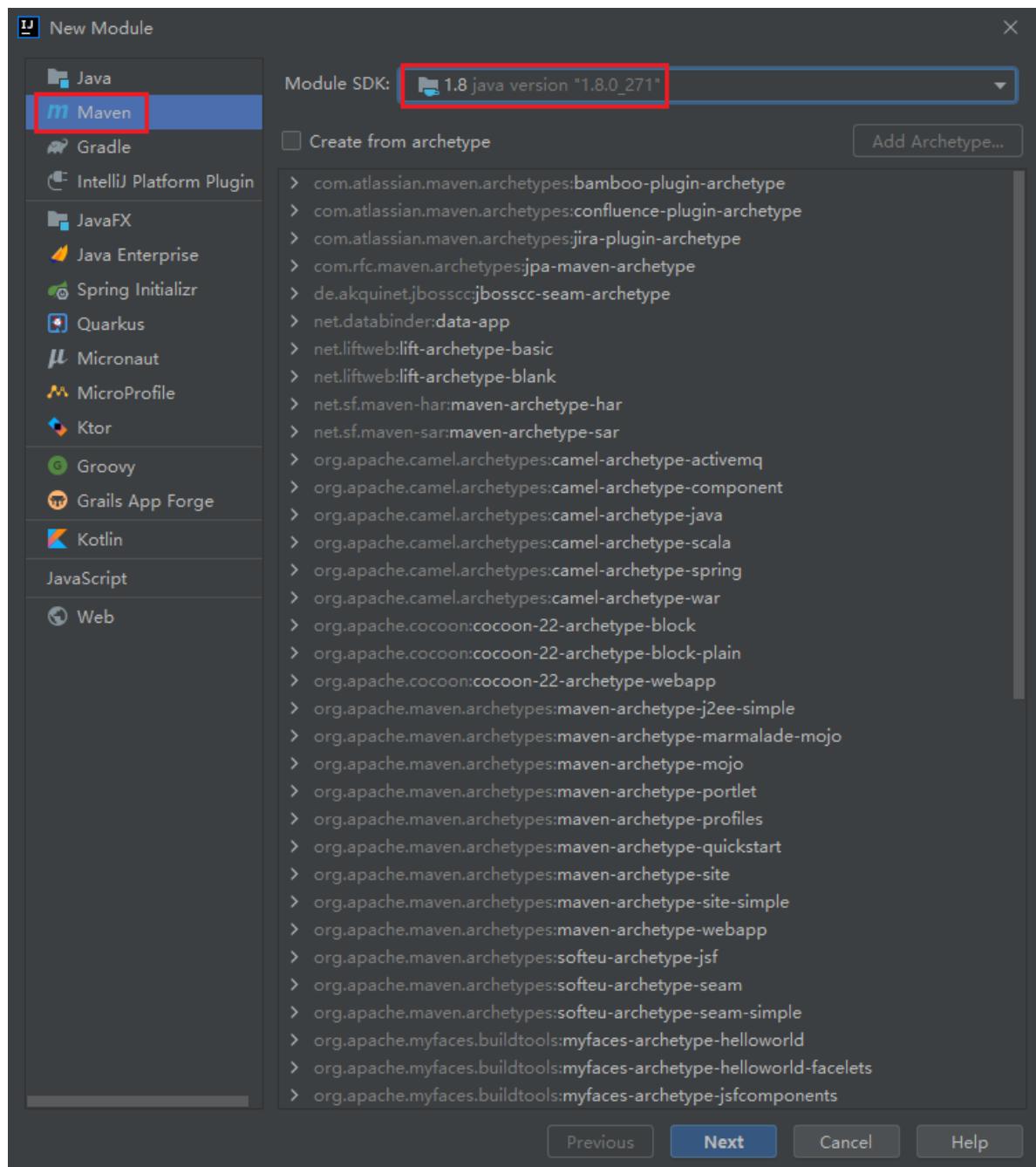
支付模块构建

如何构建微服务模块？

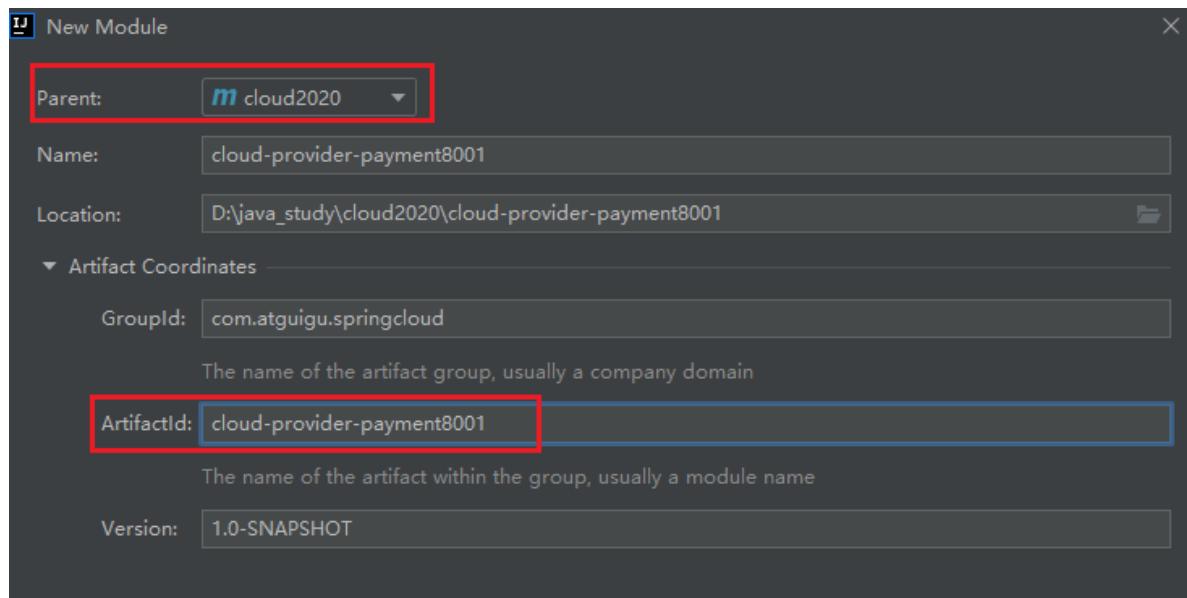
1. 建module
2. 改pom
3. 写yaml
4. 主启动
5. 业务类

1. 建module

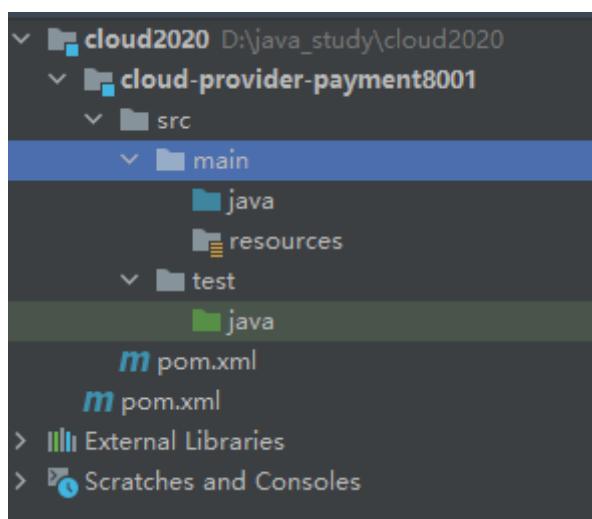
1) 在父工程中 新建 子模块



2) 子模块命名为 `cloud-provider-payment8001`



3) 建完后，项目目录结构如下。



2. 改pom

在 `cloud-provider-payment8001` 中修改它的 `pom.xml` 文件

```
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      <!--mysql-connector-java-->
20      <dependency>
21          <groupId>mysql</groupId>
22          <artifactId>mysql-connector-java</artifactId>
23      </dependency>
24      <!--jdbc-->
25      <dependency>
26          <groupId>org.springframework.boot</groupId>
27          <artifactId>spring-boot-starter-jdbc</artifactId>
28      </dependency>
29      <dependency>
30          <groupId>org.springframework.boot</groupId>
31          <artifactId>spring-boot-devtools</artifactId>
32          <scope>runtime</scope>
33          <optional>true</optional>
34      </dependency>
35      <dependency>
36          <groupId>org.projectlombok</groupId>
37          <artifactId>lombok</artifactId>
38          <optional>true</optional>
39      </dependency>
40      <dependency>
41          <groupId>org.springframework.boot</groupId>
42          <artifactId>spring-boot-starter-test</artifactId>
43          <scope>test</scope>
44      </dependency>
45  </dependencies>
```

3. 写YAML

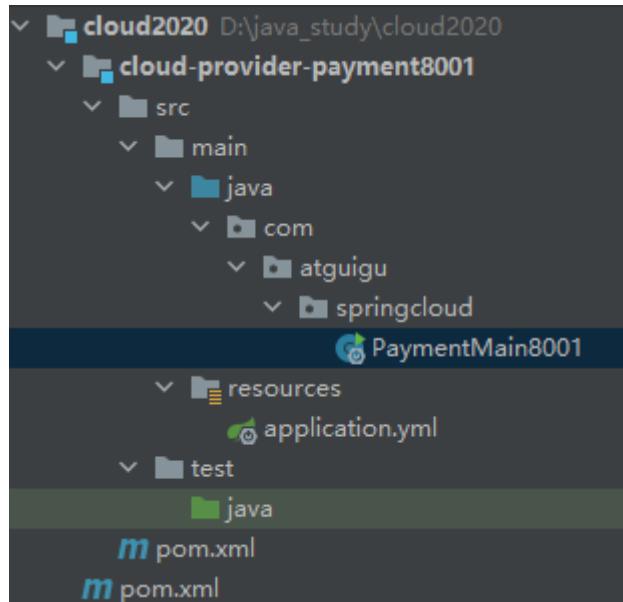
```
1  server:
2      port: 8001
3  spring:
4      application:
5          name: cloud-payment-service
6      datasource:
7          type: com.alibaba.druid.pool.DruidDataSource
8          driver-class-name: com.mysql.jdbc.Driver
9          url: jdbc:mysql://localhost:3306/db2019?
10         useUnicode=true&characterEncoding=utf-8&useSSL=false
11         username: root
12         password: 123456
13     mybatis:
14         mapper-locations: classpath:mapper/*.xml
15         type-aliases-package: com.atguigu.springcloud.entities
```

4. 主启动

创建一个包 `com.atguigu.springcloud`，在其中创建一个Class类，名为 `PaymentMain8001` 作为主启动类

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

完成后，目录结构如下：



5. 业务类

构建业务类，主要五步：

1. 建表SQL
2. entities
3. dao
4. service
5. controller

1) 建表SQL

在MySQL中建立一个 `db2019` 的数据库，在该库中建立一张表：

```
1 CREATE TABLE `payment` (
2     `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'ID',
3
4     `serial` varchar(200) DEFAULT '',
5
6     PRIMARY KEY (`id`)
7
8 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8
```

2) 建entities

0. 新建一个包 com.atguigu.springcloud.entities
1. 构建主实体 Payment

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

2. 构建Json封装体 CommonResult (用于返回给前端)

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
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 }
```

3) dao

0. 新建一个包 com.atguigu.springcloud.dao
1. 建立一个 PaymentDao 接口

```
1 @Mapper
2 public interface PaymentDao {
3
4     int create(Payment payment);
5
6     Payment getPaymentById(@Param("id") Long id);
7
8 }
```

2. 新建一个 PaymentMapper.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
```

```
3 <mapper namespace="com.atguigu.springcloud.dao.PaymentDao">
4
5     <insert id="create" parameterType="Payment" useGeneratedKeys="true"
6         keyColumn="id" keyProperty="id">
7         insert into payment(serial) values(#{serial})
8     </insert>
9
9     <resultMap id="BaseResultMap"
10        type="com.atguigu.springcloud.entities.Payment">
11         <id column="id" property="id" jdbcType="BIGINT"/>
12         <id column="serial" property="serial" jdbcType="VARCHAR"/>
13     </resultMap>
14     <select id="getPaymentById" parameterType="long" resultMap="BaseResultMap">
15         select * from payment where id=#{id}
16     </select>
17 </mapper>
```

4) service

1. 编写一个 `PaymentService` 接口

```
1 public interface PaymentService {
2
3     int create(Payment payment);
4
5     Payment getPaymentById(@Param("id") Long id);
6 }
```

2. 编写一个 `PaymentServiceImpl` 类

```
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 }
```

注:

`@Autowired` 和 `@Resource` 注解的区别:

https://blog.csdn.net/weixin_40423597/article/details/80643990

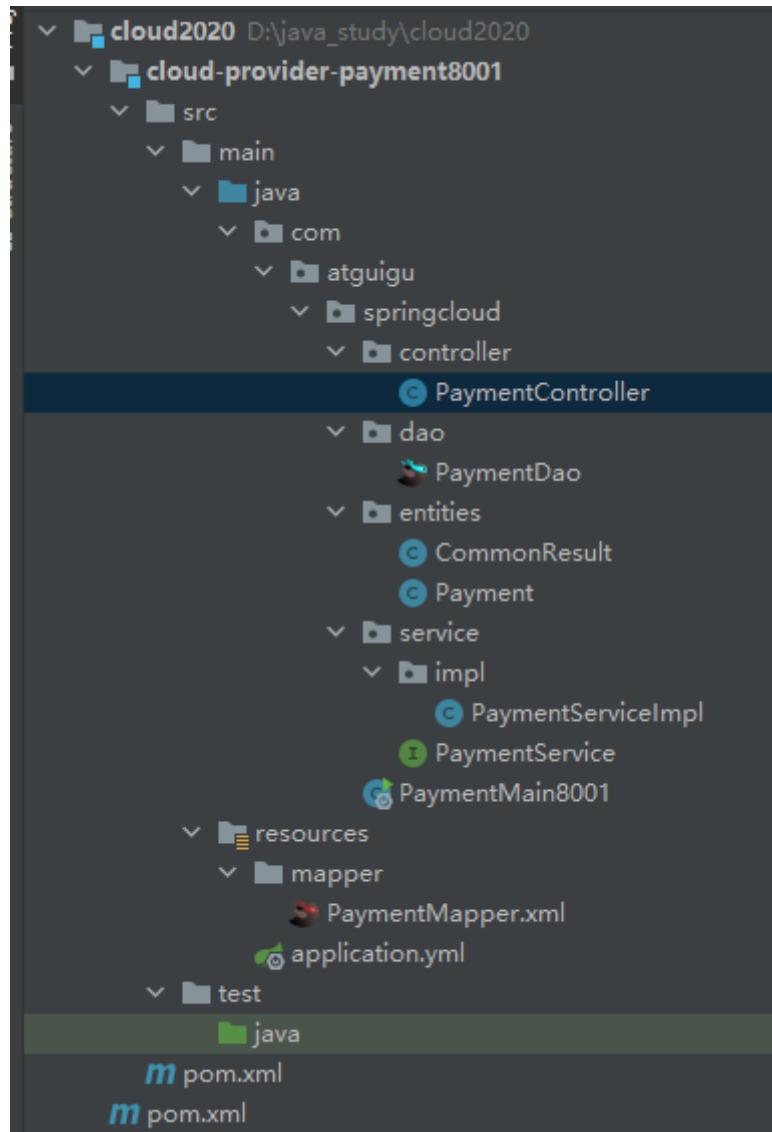
5) controller

编写一个 `PaymentController` 类

```
1  @RestController
2  @RequestMapping("/payment")
3  @Slf4j
4  public class PaymentController {
5      @Resource
6      private PaymentService paymentService;
7
8      @PostMapping("/create")
9      public CommonResult create(@RequestBody 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         } else {
15             return new CommonResult(444, "插入数据库失败");
16         }
17     }
18
19     @GetMapping("/get/{id}")
20     public CommonResult getPaymentById(@PathVariable("id") Long id) {
21         Payment paymentById = paymentService.getPaymentById(id);
22         log.info("*****查询结果: " + paymentById);
23         if (paymentById != null) {
24             return new CommonResult(200, "查询成功", paymentById);
25         } else {
26             return new CommonResult(444, "没有对应记录, 查询失败");
27         }
28     }
29
30 }
```

测试

当前目录结构如下：



在数据库中输入一条记录：

	id	serial
▶	31	尚硅谷001

启动程序，利用postman进行测试

消费者订单模块构建

1. 建module

构建一个名为 `cloud-consumer-order80` 的模块

2. 改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
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. 写YAML

```
1 server:
2   port: 80
3
```

4. 主启动

在 com.atguigu.springcloud 目录下构建一个 OrderMain80 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 }
```

5. 业务类

由于订单模块的业务，主要是调用支付模块的功能。所以，订单模块不需要连接数据库什么的！

1) 建entities

订单模块使用的实体类和支付模块的实体类相同，直接拷贝 `Payment` 和 `CommonResult` 类过来即可

2) controller

现在要考虑，如何在订单模块的controller中调用支付模块的controller？

最原始的方案是：`httpClient`

现在可以使用：`restTemplate`

RestTemplate是什么？

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。

- 官网地址：<https://docs.spring.io/spring-framework/docs/5.2.2.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html>
- 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响应转换被转换成的对象类型。

为了使用restTemplate，我们先写一个config配置类，将它注入到容器中。

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

编写controller

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

```
12         return restTemplate.postForObject(PAYMENT_URL + "/payment/create",
13             payment, CommonResult.class);
14
15     @GetMapping("/payment/get/{id}")
16     public CommonResult<Payment> getPayment(@PathVariable("id") Long id) {
17         return restTemplate.getForObject(PAYMENT_URL + "payment/get/" + id,
18             CommonResult.class);
19     }
20 }
```

3) 测试

开启订单模块和支付模块这两个子项目，

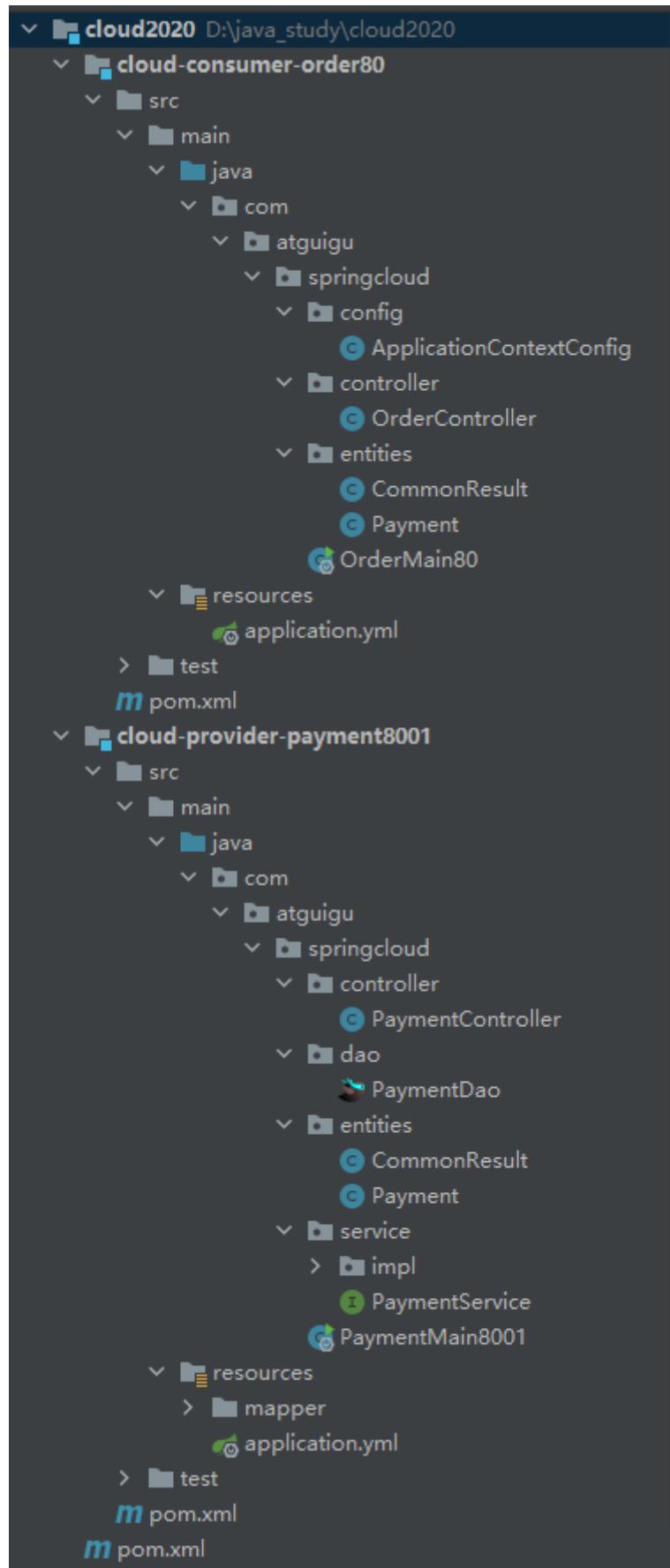
使用浏览器输入

- <http://localhost/consumer/payment/get/31>
- <http://localhost/consumer/payment/create?serial=111>

测试，是否成功

工程重构

在上述章节，我们构建了一个支付模块和订单模块。项目结构如下：



能够发现，两个模块中都有一个 **entities** 包，里面的实体类完全一样！所以造成了冗余！

所以，我们要进行工程重构，将重复部分重构

1. 新建一个子工程

新建子工程 `cloud-api-commons`

2. 改pom

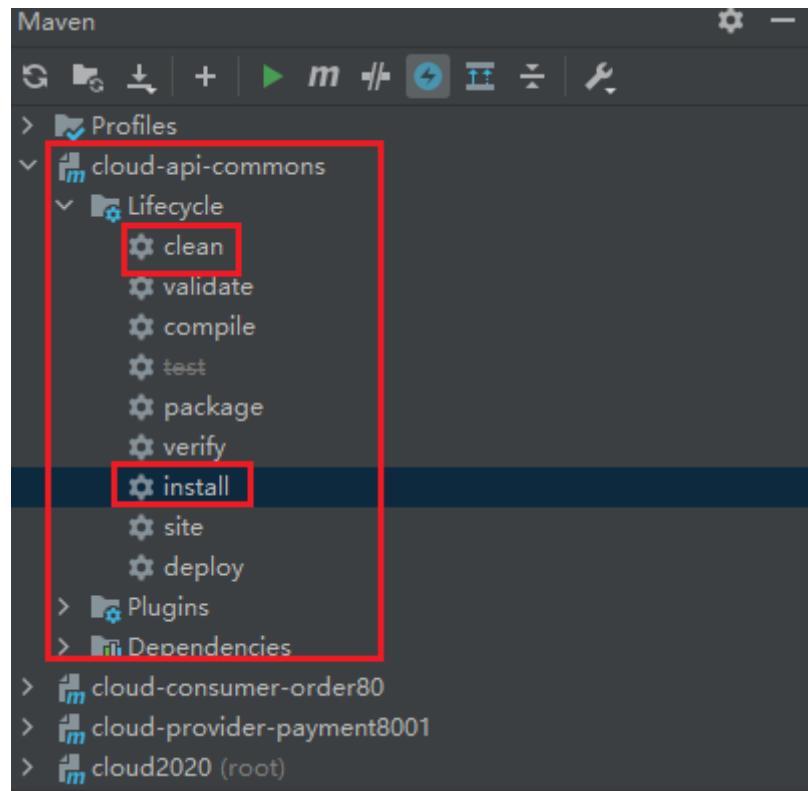
```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-devtools</artifactId>
5     <scope>runtime</scope>
6     <optional>true</optional>
7   </dependency>
8   <dependency>
9     <groupId>org.projectlombok</groupId>
10    <artifactId>lombok</artifactId>
11    <optional>true</optional>
12  </dependency>
13  <dependency>
14    <groupId>cn.hutool</groupId>
15    <artifactId>hutool-all</artifactId>
16    <version>5.1.0</version>
17  </dependency>
18 </dependencies>
```

3. entities

直接将订单模块或支付模块中的实体类全部复制过来即可

4. 打包

利用maven的命令 `clean`、`install` 将 `cloud-api-commons` 下载到本地库



5. 订单模块和支付模块分别改造

1. 删除各自原先的entities目录
2. 在两个模块的pom.xml文件中加入以下依赖

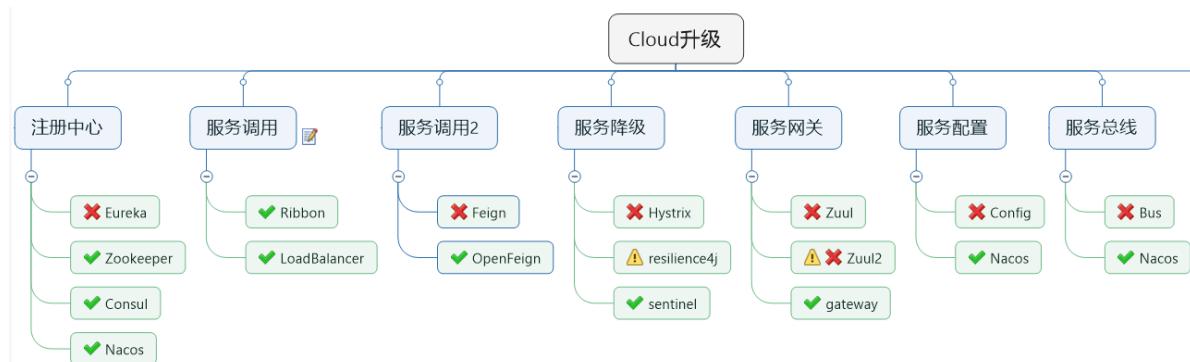
```

1 <!-- 引用自己定义的 api 通用包 -->
2 <dependency>
3   <groupId>com.atguigu.springcloud</groupId>
4   <artifactId>cloud-api-commons</artifactId>
5   <version>${project.version}</version>
6 </dependency>

```

总结

上述实现的工程，并没有用到以下各种组件！而在下面的章节，我们需要依次学习。



三、Eureka服务注册与发现

Eureka 基础知识

Eureka 源代码：<https://github.com/Netflix/eureka>

什么是服务治理

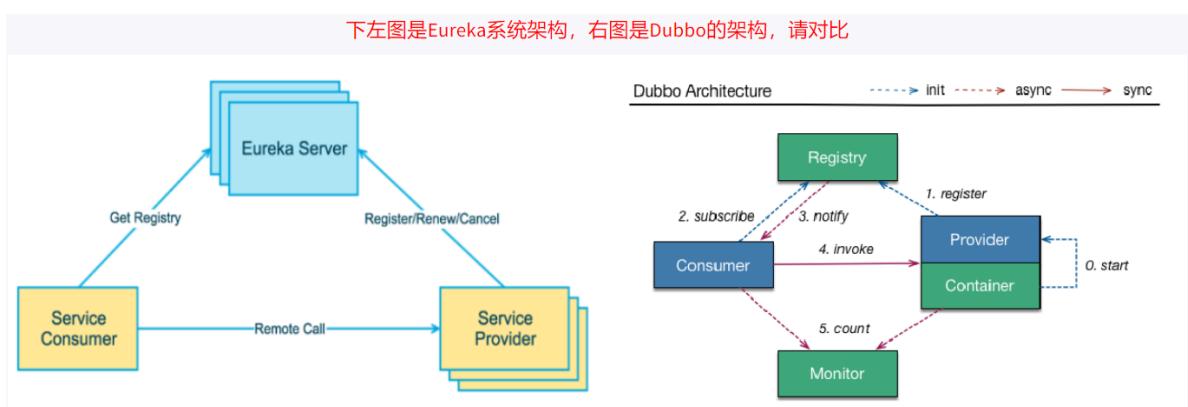
Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务治理

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，管理服务与服务之间依赖关系，可以实现服务调用、负载均衡、容错等，实现服务发现与注册

什么是服务注册

Eureka采用了CS的设计架构，Eureka Server 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，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))



Eureka 两个组件

Eureka包含两个组件：**Eureka Server** 和 **Eureka Client**

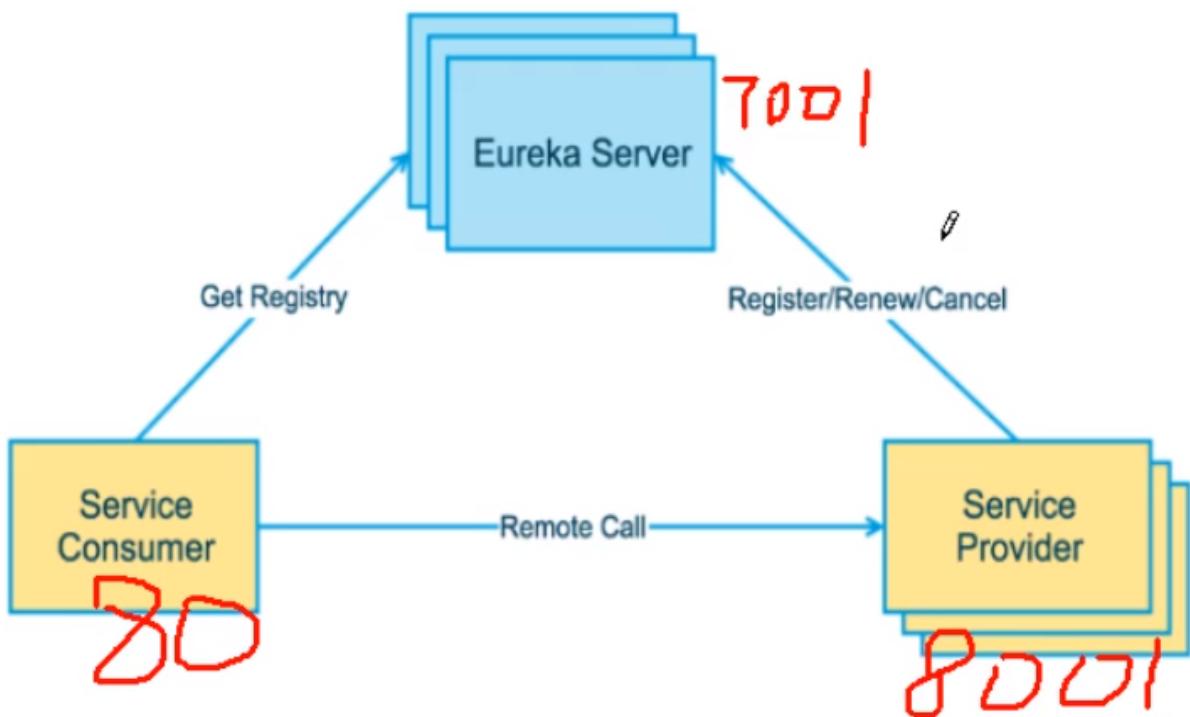
1. **Eureka Server** 提供服务注册服务

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有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。

2. **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Eureka构建步骤



IDEA 生成 Eureka Server 端服务注册中心

1. 建module

构建一个 `cloud-eureka-server7001` 模块

2. 改pom

```
1 <dependencies>
2     <!--eureka-server-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-netflix-eureka-server</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>com.atguigu.springcloud</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <!--boot web actuator-->
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  <!--一般通用配置-->
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  </dependency>
33  <dependency>
34      <groupId>org.springframework.boot</groupId>
35      <artifactId>spring-boot-starter-test</artifactId>
36      <scope>test</scope>
37  </dependency>
38  <dependency>
39      <groupId>junit</groupId>
40      <artifactId>junit</artifactId>
41  </dependency>
42 </dependencies>
```

注：这里和eureka有关的依赖之所以不用写版本号，是因为父工程中引入的 `spring-cloud-dependencies` 包含了eureka相关版本

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

3. 写YAML

在module中新建一个 `application.yml`，并填写相关配置

```
1 server:
2   port: 7001
3 eureka:
4   instance:
5     hostname: localhost #eureka服务端的实例名称
6   client:
7     #false表示不向注册中心注册自己。
8     register-with-eureka: false
9     #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
10    fetch-registry: false
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://${eureka.instance.hostname}:${server.port}/eureka/
```

4. 主启动

编写一个主启动类

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

注：

`@EnableEurekaServer` 注解表示该类是Eureka的服务注册中心

还有一个注解 `@EnableEurekaClient` 表示一个类是Eureka的客户端

5. 测试

1. 将该类启动
2. 打开浏览器，输入：<http://localhost:7001/>

出现以下画面，则成功

The screenshot shows the Spring Eureka dashboard. At the top, it displays 'spring Eureka' and 'HOME LAST 1000 SINCE STARTUP'. Below this, there are two main sections: 'System Status' and 'DS Replicas'.

System Status:

Environment	test	Current time	2022-04-20T19:10:27 +0800
Data center	default	Uptime	00:01
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas:

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info:

Name	Value
total-avail-memory	464mb
environment	test
num-of-cpus	16
current-memory-usage	310mb (66%)
server-upptime	00:01
registered-replicas	
unavailable-replicas	
available-replicas	

支付模块入驻进eureka

将 `cloud-provider-payment8001` 注册进Eureka Server，成为服务提供者provider.

因此，下面我们需要修改 `cloud-provider-payment8001` 模块的代码

1. 修改pom

在 `pom.xml` 文件中，加入以下依赖：

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

2. 写YML

在原来的 `application.yml` 配置文件中，加入以下配置

```
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true才能
6     # 配合ribbon使用负载均衡
7     fetch-registry: true
8     service-url:
9       defaultZone: http://localhost:7001/eureka
```

3. 主启动

在主启动类上加上`@EnableEurekaClient`注解

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

4. 测试

1. 先启动EurekaServer
2. 在浏览器中进入：<http://localhost:7001/>
3. 查看是否有应用注册进eureka

DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-payment-service:8001

可以看到有一个叫 `cloud-payment-service` 的应用注册进eurekaServer了。

这个名字，其实就是模块中 `application.yaml` 中配置的 `spring.application.name`

```
server:
  port: 8001
spring:
  application:
    name: cloud-payment-service
```

订单模块注册进eureka

将订单模块注册进EurekaServer，成为服务消费者consumer

1. 修改pom

在 `pom.xml` 文件中，加入以下依赖：

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

2. 写YML

在原来的 `application.yml` 配置文件中，加入以下配置

```
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true才能
6     # 配合ribbon使用负载均衡
7     fetch-registry: true
8     service-url:
9       defaultZone: http://localhost:7001/eureka
10
11 spring:
12   application:
13     name: cloud-order-service
```

3. 主启动

在主启动类上加上`@EnableEurekaClient`注解

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

4. 测试

1. 先启动EurekaServer
2. 在浏览器中进入：<http://localhost:7001/>
3. 查看是否有应用注册进eureka

DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-payment-service:8001

4. 在浏览器中测试：<http://localhost/consumer/payment/get/31>

```

    {
      code: 200,
      message: "查询成功",
      - data: {
        id: 31,
        serial: "尚硅谷001"
      }
    }
  
```

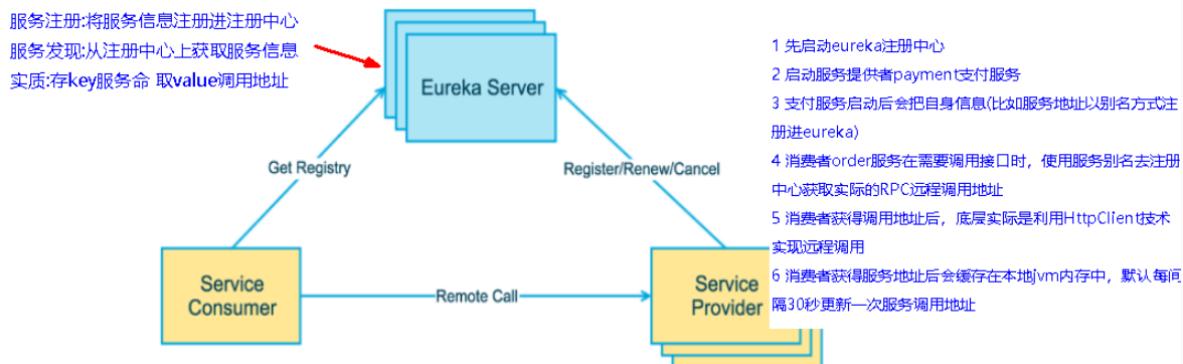
5. 如果不想将订单模块注册进 eureka，则修改yaml文件

```

1 eureka:
2   client:
3     # 表示是否将自己注册进EurekaServer, 默认为true
4     register-with-eureka: false
  
```

Eureka集群构建步骤

Eureka集群原理说明



问题：微服务RPC远程服务调用最核心的是什么

高可用

试想你的注册中心只有一个only one，它出故障了那就呵呵了，会导致整个服务环境不可用，所以解决办法：搭建Eureka注册中心集群，实现负载均衡+故障容错

EurekaServer集群环境构建步骤

1. 新建模块

注：参考 [cloud-eureka-server7001](#)，新建 [cloud-eureka-server7002](#) 模块，代码基本一致

2. 写YAML

这里和以前单机的Eureka不太一样。

在修改YAML之前，需要修改映射配置：

1. 找到C:\Windows\System32\drivers\etc路径下的hosts文件
2. 修改映射配置添加进hosts文件

```
1 127.0.0.1 eureka7001.com
2 127.0.0.1 eureka7002.com
```

修改完后，在修改yaml文件

7001对应的yaml文件如下：

```
1 server:
2   port: 7001
3   eureka:
4     instance:
5       hostname: eureka7001.com #eureka服务端的实例名称
6     client:
7       #false表示不向注册中心注册自己
8       register-with-eureka: false
9       #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
10      fetch-registry: false
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
```

7002对应的yaml文件如下：

```
1 server:
2   port: 7002
3   eureka:
4     instance:
5       hostname: eureka7002.com #eureka服务端的实例名称
6     client:
7       #false表示不向注册中心注册自己。
8       register-with-eureka: false
9       #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
10      fetch-registry: false
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://eureka7001.com:7001/eureka/
14
```

测试

构建好集群后，测试

- 在浏览器中打开：<http://localhost:7001> 或 <http://eureka7001.com:7001>

The screenshot shows the Spring Eureka interface for port 7001. At the top, it says "spring Eureka" and "HOME LAST 1000 SINCE STARTUP". Below that is a "System Status" section with environment information:

Environment	test
Data center	default

On the right, there's a table of system metrics:

Current time	2022-04-21T15:44:19 +0800
Uptime	00:01
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

Below this is a "DS Replicas" section with a table:

Application	AMIs	Availability Zones	Status
No instances available			

A red box highlights the URL "eureka7002.com" in the "DS Replicas" table.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

- 在浏览器中打开：<http://localhost:7002> 或 <http://eureka7002.com:7002>

The screenshot shows the Spring Eureka interface for port 7002. At the top, it says "spring Eureka" and "HOME LAST 1000 SINCE STARTUP". Below that is a "System Status" section with environment information:

Environment	test
Data center	default

On the right, there's a table of system metrics:

Current time	2022-04-21T15:44:57 +0800
Uptime	00:01
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

Below this is a "DS Replicas" section with a table:

Application	AMIs	Availability Zones	Status
No instances available			

A red box highlights the URL "eureka7001.com" in the "DS Replicas" table.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

将支付模块发布到Eureka集群中

只需要修改yaml文件

```
eureka:  
  client:  
    # 表示是否将自己注册进EurekaServer, 默认为true  
    register-with-eureka: true  
    # 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true 才能配合ribbon使用负载均衡  
    fetch-registry: true  
    service-url:  
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
```

和单机版相比，只需要改 `eureka.client.service-utl.defaultZone` 这里！

将订单模块发布到Eureka集群中

和支付模块一模一样！只需要修改yaml文件

测试

测试集群的构建是否对原有功能有影响



支付模块集群环境构建

之前只有一个支付服务提供者8001，现在要再加一个支付服务提供者！

1. 建module

参考 `cloud-provider-payment8001` 模块，新建一个 `cloud-provider-payment8002` 模块。

2. 改pom

这里直接复制8001中的pom依赖即可

3. 写YAML

可以直接复制8001的yaml文件，但是记得修改服务端口号 `server.port`

4. 主启动类&业务类

直接拷贝8001的

5. 修改controller

修改8001和8002中的controller代码

```

public class PaymentController {
    @Resource
    private PaymentService paymentService;

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

    @PostMapping("/create")
    public CommonResult create(@RequestBody Payment payment) {
        int result = paymentService.create(payment);
        log.info("*****插入结果: " + result);
        if (result > 0) {
            return new CommonResult(code: 200, message: "插入数据库成功, serverPort: " + serverPort, result);
        } else {
            return new CommonResult(code: 444, message: "插入数据库失败");
        }
    }

    @GetMapping("/get/{id}")
    public CommonResult getPaymentById(@PathVariable("id") Long id) {
        Payment paymentById = paymentService.getPaymentById(id);
        log.info("*****查询结果: " + paymentById);
        if (paymentById != null) {
            return new CommonResult(code: 200, message: "查询成功, serverPort: " + serverPort, paymentById);
        }
    }
}

```

测试

将所有模块按顺序启动

1. 打开浏览器，输入：<http://eureka7001.com: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 general information like environment (test), data center (default), current time (2022-04-21T16:43:36 +0800), and uptime (00:04). The 'DS Replicas' section shows two instances registered under 'eureka7002.com': 'CLOUD-ORDER-SERVICE' with 1 replica and 'CLOUD-PAYMENT-SERVICE' with 2 replicas.

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-7CK6BAS:cloud-payment-service:8002 , DESKTOP-7CK6BAS:cloud-payment-service:8001

2. 打开浏览器，访问：<http://localhost:8001/payment/get/31>

```

{
    code: 200,
    message: "查询成功, serverPort: 8001",
    - data: [
        id: 31,
        serial: "尚硅谷001"
    ]
}

```

3. 打开浏览器，访问：<http://localhost:8002/payment/get/31>

```
{  
    code: 200,  
    message: "查询成功, serverPort: 8002",  
    - data: {  
        id: 31,  
        serial: "尚硅谷001"  
    }  
}
```

4. 打开浏览器，访问：<http://localhost/consumer/payment/get/31>

```
{  
    code: 200,  
    message: "查询成功, serverPort: 8001",  
    - data: {  
        id: 31,  
        serial: "尚硅谷001"  
    }  
}
```

其实这里能发现，通过订单模块去访问支付模块时，还是只能访问一个固定的支付模块，这样导致两个支付模块集群就没什么作用。之所以这样，是因为在订单模块中将地址写死了，如下图所示。

```
@RestController  
@Slf4j  
@RequestMapping("/consumer")  
public class OrderController {  
  
    public static final String PAYMENT_URL = "http://localhost:8001";  
  
    @Resource  
    private RestTemplate restTemplate;  
    @GetMapping("/payment/create")  
    public CommonResult<Payment> create(Payment payment) {  
        return restTemplate.postForObject(url: PAYMENT_URL + "/payment/create", payment, CommonResult.class);  
    }  
  
    @GetMapping("/payment/get/{id}")  
    public CommonResult<Payment> getPayment(@PathVariable("id") Long id) {  
        return restTemplate.getForObject(url: PAYMENT_URL + "payment/get/" + id, CommonResult.class);  
    }  
}
```

针对这个bug，在下面一小节 负载均衡 中进行修复！

负载均衡

针对上面的bug，我们在这里进行改进！

1. 修改controller

这里修改订单模块中的 `OrderController` 类

```

@RestController
@Slf4j
@RequestMapping("/consumer")
public class OrderController {
    // public static final String PAYMENT_URL = "http://localhost:8001";
    public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";

    @Resource
    private RestTemplate restTemplate;
    @GetMapping("/payment/create")
    public CommonResult<Payment> create(Payment payment) {
        return restTemplate.postForObject(url: PAYMENT_URL + "/payment/create", payment, CommonResult.class);
    }

    @GetMapping("/payment/get/{id}")
    public CommonResult<Payment> getPayment(@PathVariable("id") Long id) {
        return restTemplate.getForObject(url: PAYMENT_URL + "payment/get/" + id, CommonResult.class);
    }
}

```

如上述代码，将URL换成服务名，而不是指定的ip+端口号

```

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

```

2. 测试

使用浏览器访问：<http://localhost/consumer/payment/get/31>，可以发现报错

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

```

Thu Apr 21 18:30:11 CST 2022
There was an unexpected error (type=Internal Server Error, status=500).
I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/31": CLOUD-PAYMENT-SERVICE; nested exception is
java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE
org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://CLOUD-PAYMENT-SERVICE/payment/get/31"
CLOUD-PAYMENT-SERVICE; nested exception is java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE
    at org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:751)
    at org.springframework.web.client.RestTemplate.execute(RestTemplate.java:677)
    at org.springframework.web.client.RestTemplate.getForObject(RestTemplate.java:318)
    at com.atguigu.springcloud.controller.OrderController.getPayment(OrderController.java:35)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.springframework.web.method.support.InvocableHandlerMethod.invoke(InvocableHandlerMethod.java:190)
    at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)
    at

```

可以发现，还是需要继续修改！

因为我们还没有开启负载均衡功能！

3. 开启负载均衡功能@LoadBalanced

使用@LoadBalanced注解赋予RestTemplate负载均衡的能力

即，rest加上了@LoadBalanced才能服务发现

在订单模块的 ApplicationContext 中，加上注解：

```

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

```

4. 测试

多次使用浏览器访问：<http://localhost/consumer/payment/get/31>

```

{
    code: 200,
    message: "查询成功, serverPort: 8001",
    - data: [
        id: 31,
        serial: "尚硅谷001"
    ]
}

{
    code: 200,
    message: "查询成功, serverPort: 8002",
    - data: [
        id: 31,
        serial: "尚硅谷001"
    ]
}

```

多次访问就能看到，会轮询访问两个支付模块！

后期会使用Ribbon做负载均衡！这里只是使用了@LoadBalanced注解达到类似效果而已

actuator 微服务信息完善

1. 主机名称:服务名称修改

目前问题是：目前注册的应用的Status栏中含有主机名称。如下图所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-7CK6BAS:cloud-payment-service:8002 , DESKTOP-7CK6BAS:cloud-payment-service:8001

现在需要按照规范要求，只暴露服务名！不要出现主机名称

可以做如下修改：

1. 修改 `cloud-provider-payment8001` 模块的yaml文件，加上以下配置即可！

```
1 eureka:  
2   instance:  
3     instance-id: payment8001
```

```
eureka:  
  client:  
    # 表示是否将自己注册进EurekaServer, 默认为true  
    register-with-eureka: true  
    # 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true才能配合ribbon使用负载均衡  
    fetch-registry: true  
    service-url:  
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka  
  instance:  
    instance-id: payment8001
```

2. `cloud-provider-payment8002` 做类似修改!

修改之后:

Instances currently registered with Eureka				
Application	AMIs	Availability Zones	Status	修改之前
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-order-service:80	
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) payment8002 , payment8001	修改之后

2. 访问信息有IP信息提示

目前的问题：当前的访问信息不会有IP地址提示。如下图所示。当鼠标移动到超链接所在处，浏览器左下角不会提示该连接所访问的IP

CLOUD-PAYMENT-SERVICE	n/a (2)	UP (2) - DESKTOP-7CK6BAS:cloud-payment-service:8002, DESKTOP-7CK6BAS:cloud-payment-service:8001														
General Info																
<table border="1"><thead><tr><th>Name</th><th>Value</th></tr></thead><tbody><tr><td>total-avail-memory</td><td>432mb</td></tr><tr><td>environment</td><td>test</td></tr><tr><td>num-of-cpus</td><td>16</td></tr><tr><td>current-memory-usage</td><td>195mb (45%)</td></tr><tr><td>www第一名</td><td>00:01</td></tr><tr><td>desktop-7ck6bas:8002/actuator/info</td><td></td></tr></tbody></table>			Name	Value	total-avail-memory	432mb	environment	test	num-of-cpus	16	current-memory-usage	195mb (45%)	www第一名	00:01	desktop-7ck6bas:8002/actuator/info	
Name	Value															
total-avail-memory	432mb															
environment	test															
num-of-cpus	16															
current-memory-usage	195mb (45%)															
www第一名	00:01															
desktop-7ck6bas:8002/actuator/info																

可以做如下修改：

1. 修改 `cloud-provider-payment8001` 模块的yaml文件，加上以下配置即可！

```
1 eureka:  
2   instance:  
3     prefer-ip-address: true
```

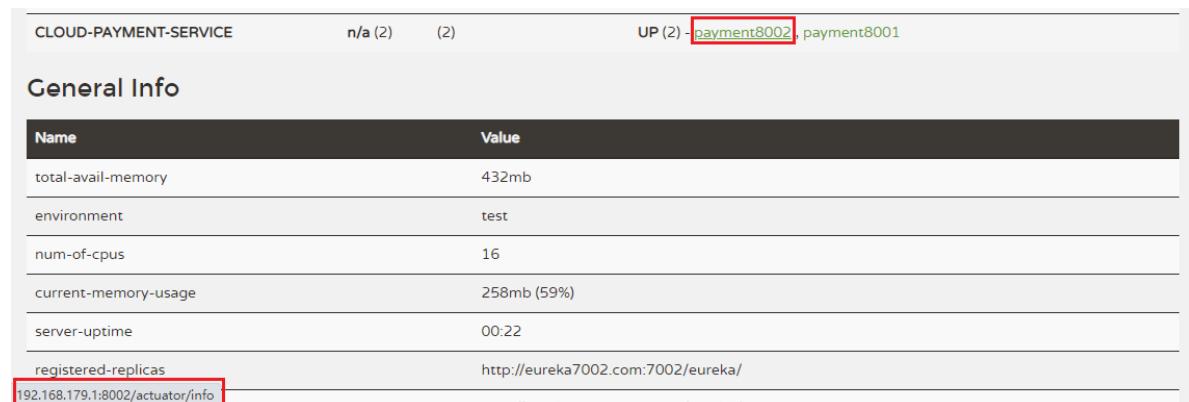
```

eureka:
  client:
    # 表示是否将自己注册进EurekaServer, 默认为true
    register-with-eureka: true
    # 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true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
  instance:
    instance-id: payment8001
    prefer-ip-address: true

```

- cloud-provider-payment8002 做类似修改即可!

修改后：



Name	Value
total-avail-memory	432mb
environment	test
num-of-cpus	16
current-memory-usage	258mb (59%)
server-uptime	00:22
registered-replicas	http://eureka7002.com:7002/eureka/
192.168.179.1:8002/actuator/info	

服务发现 Discovery

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

修改微服务的controller

对注册进eureka里的微服务，比如 cloud-provider-payment8001，我们对它的Controller进行修改

```

1  @RestController
2  @RequestMapping("/payment")
3  @Slf4j
4  public class PaymentController {
5      @Resource
6      private PaymentService paymentService;
7
8      @Value("${server.port}")
9      private String serverPort;

```

```

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         List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-
21 PAYMENT-SERVICE");
22         for (ServiceInstance instance : instances) {
23             log.info(instance.getServiceId() + "\t" + instance.getHost() + "\t" +
24             instance.getPort() + "\t" + instance.getUri());
25         }
26         return this.discoveryClient;
27     }
28
29     @PostMapping("/create")
30     public CommonResult create(@RequestBody Payment payment) {
31         int result = paymentService.create(payment);
32         log.info("****插入结果: " + result);
33         if (result > 0) {
34             return new CommonResult(200, "插入数据库成功, serverPort: " +
35             serverPort, result);
36         } else {
37             return new CommonResult(444, "插入数据库失败");
38         }
39     }
40
41     @GetMapping("/get/{id}")
42     public CommonResult getPaymentById(@PathVariable("id") Long id) {
43         Payment paymentById = paymentService.getPaymentById(id);
44         log.info("****查询结果: " + paymentById);
45         if (paymentById != null) {
46             return new CommonResult(200, "查询成功, serverPort: " + serverPort,
47             paymentById);
48         } else {
49             return new CommonResult(444, "没有对应记录, 查询失败");
50         }
51     }

```

注意：这里导入的 `DiscoveryClient` 是 `cloud` 包下的，而不是 `netflix` 的！

```
import org.springframework.cloud.client.discovery.DiscoveryClient;
```

上述 `Controller` 类中主要是添加了如下代码：

```

public class PaymentController {
    @Resource
    private PaymentService paymentService;

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

    @Resource
    private DiscoveryClient discoveryClient;

    @GetMapping("/discovery")
    public Object discovery() {
        List<String> services = discoveryClient.getServices();
        for (String element : services) {
            log.info("*****element:" + element);
        }
        List<ServiceInstance> instances = discoveryClient.getInstances(serviceId: "CLOUD-PAYMENT-SERVICE");
        for (ServiceInstance instance : instances) {
            log.info(instance.getServiceId() + "\t" + instance.getHost() + "\t" + instance.getPort() + "\t" + instance.getUri());
        }
        return this.discoveryClient;
    }
}

```

修改主启动类

在微服务，如 `cloud-provider-payment8001` 中，修改它的主启动类！

在主启动类上添加 `@EnableDiscoveryClient` 注解。如下：

```

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  }

```

注：新版本的spring-cloud 可以不需要添加该注解

测试

重启 `cloud-provider-payment8001`

在浏览器上访问：<http://localhost:8001/payment/discovery>

会在浏览器上显示：

```
{
  - services: [
      "cloud-payment-service",
      "cloud-order-service"
    ],
  order: 0
}
```

控制台会输出：

```
: *****element:cloud-payment-service
: *****element:cloud-order-service
: CLOUD-PAYMENT-SERVICE    192.168.179.1    8001      http://192.168.179.1:8001
: CLOUD-PAYMENT-SERVICE    192.168.179.1    8002      http://192.168.179.1:8002
```

Eureka 自我保护

概述

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

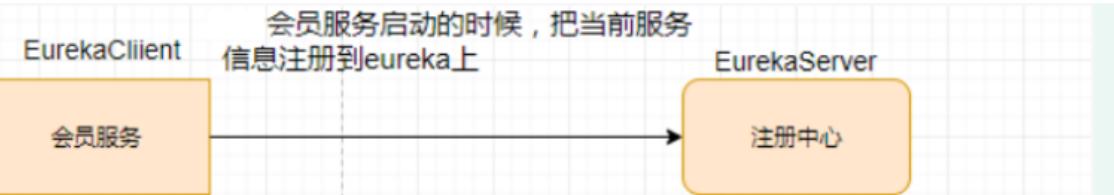
EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

为什么会产生Eureka自我保护机制？

为了防止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

什么是自我保护模式？

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。



自我保护机制：默认情况下EurekaClient定时向EurekaServer端发送心跳包。如果Eureka在server端在一定时间内(默认90秒)没有收到EurekaClient发送心跳包，便会直接从服务注册列表中剔除该服务，但是在短时间(90秒中)内丢失了大量的服务实例心跳，这时候EurekaServer会开启自我保护机制，不会剔除该服务(该现象可能出现在如果网络不通但是EurekaClient为出现宕机，此时如果换做别的注册中心如果一定时间内没有收到心跳会将剔除该服务，这样就出现了严重失误，因为客户端还能正常发送心跳，只是网络延迟问题，而保护机制是为了解决此问题而产生的)

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。

Eureka自我保护的设计思想属于CAP里面的AP

一句话总结：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

怎么禁止自我保护

只需要修改注册中心的配置即可！

注册中心修改

首先，在注册中心中，自我保护机制是默认开启的：`eureka.server.enable-self-preservation=true`

在注册中心 EurekaServer 端7001修改yaml文件，将上述机制修改为false即可！

```

1 eureka:
2   server:
3     # 关闭自我保护机制，保证不可用服务被及时剔除
4     enable-self-preservation: false
5     # 修改了默认发送心跳的间隔时间。(这里只为了做测试)
6     eviction-interval-timer-in-ms: 2000

```

修改完毕后，进行测试：

在浏览器中访问：<http://eureka7001.com:7001/>

The screenshot shows the Spring Eureka system status page. At the top, it displays the Spring logo and the word "Eureka". On the right, there are links for "HOME" and "LAST 1000 SINCE STARTUP". Below this, the "System Status" section contains two tables. The first table shows environment details: Environment is "test", Data center is "default". The second table shows system metrics: Current time is "2022-04-22T15:32:32 +0800", Uptime is "00:00", Lease expiration enabled is "true", Renews threshold is "1", and Renews (last min) is "0". A red-bordered box highlights a warning message: "THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS." Below this, the "DS Replicas" section is titled "Instances currently registered with Eureka".

表示eureka自我保护机制关闭了！

修改生产者客户端

为了测试禁止eureka自我保护的效果，这里对 `cloud-provider-payment8001` 进行修改。

修改该模块的yaml文件：

```
1 eureka:
2     # Eureka 客户端向服务端发送心跳的时间间隔，单位是秒（默认30s）
3     lease-renewal-interval-in-seconds: 1
4     # Eureka 服务端在收到最后一次心跳后等待时间上限，单位为秒（默认90s），超时将剔除服务
5     lease-expiration-duration-in-seconds: 2
```

测试：

1. 开启eureserver7001和payment8001

可以看到eureka server上可以发现payment8001

The screenshot shows the "DS Replicas" section of the Eureka interface. It has a title "Instances currently registered with Eureka". Below it is a table with columns: Application, AMIs, Availability Zones, and Status. There is one entry: CLOUD-PAYMENT-SERVICE, with n/a (1) in AMIs, (1) in Availability Zones, and UP (1) - payment8001 in Status.

2. 假设payment8001宕机，比如关闭该服务

可以看到eureka server上立即删除了payment8001

The screenshot shows the "DS Replicas" section again. The table now shows "No instances available", indicating that the previously registered instance has been removed.

四、Zookeeper服务注册与发现

Eureka停更了怎么办

<https://github.com/Netflix/eureka/wiki>

Eureka 1.0

- Eureka at a glance
- Configuring Eureka
- Building Eureka Client and Server
- Running the Demo Application
- Deploying-Eureka-Servers-in-EC2
- Understanding Eureka Client/Server Communication
- Server Self Preservation Mode
- Eureka REST operations
- Understanding Eureka Peer to Peer communication
- Overriding Default Configurations
- FAQ

Eureka 2.0 (Discontinued)

The existing open source work on eureka 2.0 is discontinued. The code base and artifacts that were released as part of the existing repository of work on the 2.x branch is considered use at your own risk.

Eureka 1.x is a core part of Netflix's service discovery system and is still an active project.

下面学习SpringCloud整合Zookeeper代替Eureka

注：

学习整合之前最好学习以下zookeeper，并将zookeeper安装到一台Linux虚拟机上。

后续整合，会使用这台虚拟机当作zookeeper服务器。

我这里在本地已配置好一台Linux虚拟机（192.168.179.130）

SpringCloud整合Zookeeper

环境构建

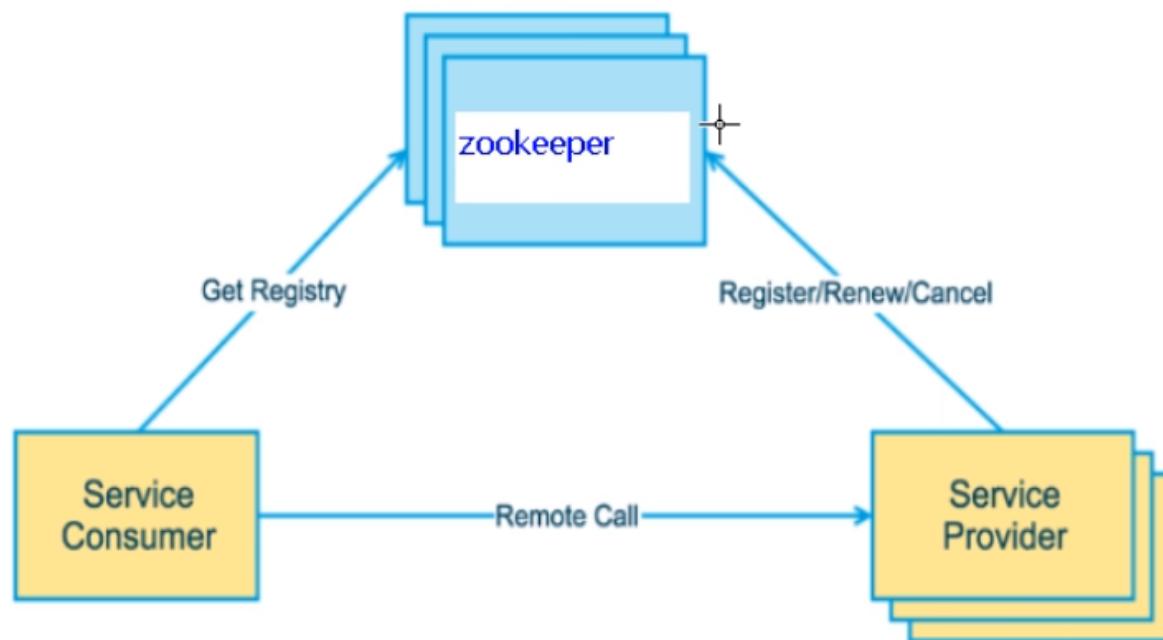
1. 在一台Linux虚拟机上安装zookeeper（虚拟机IP为：192.168.179.130）

我这里使用该链接下载了zookeeper3.5.7

<https://archive.apache.org/dist/zookeeper/zookeeper-3.5.7/apache-zookeeper-3.5.7-bin.tar.gz>

2. 本地Windows10系统的IP为：10.1.125.73

下面会构建一个这样的系统！



服务提供者

1. 新建模块

新建 `cloud-provider-payment8004` 作为服务提供者

2. 改pom

```
1 <dependencies>
2     <!-- SpringBoot整合Web组件 -->
3     <dependency>
4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-starter-web</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>com.atguigu.springcloud</groupId>
10        <artifactId>cloud-api-commons</artifactId>
11        <version>${project.version}</version>
12    </dependency>
13    <!-- SpringCloud整合zookeeper客户端 -->
14    <dependency>
15        <groupId>org.springframework.cloud</groupId>
16        <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
17    </dependency>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-devtools</artifactId>
21        <scope>runtime</scope>
22        <optional>true</optional>
23    </dependency>
```

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

注意：一定要引入springcloud整合zookeeper的包：

```
<!-- SpringCloud整合zookeeper客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
```

3. 写yaml

编写 `application.yml` 配置文件

```
1  server:
2      port: 8004
3  spring:
4      application:
5          name: cloud-provider-payment
6      cloud:
7          zookeeper:
8              connect-string: 192.168.179.130:2181
```

4. 主启动类

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

5. Controller

```

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

```

注：由于这里主要是要学习springcloud和zookeeper的整合，所以对于dao和mysql等一些东西没有配置！因为它们不是重点！

6. 启动zookeeper

- 在linux中启动zookeeper

```
1 ./zkServer.sh start
```

- 启动zookeeper客户端

```
1 ./bin/zkCli.sh
```

- 在zookeeper客户端中查看根节点所包含的内容

```

1 [zk: localhost:2181(CONNECTED) 0] ls /
2 [zookeeper]

```

7. 启动服务提供者

将 `cloud-provider-payment8004` 启动

注：如果zookeeper版本 < 你导入的 `spring-cloud-starter-zookeeper-discovery` 版本，则会报错

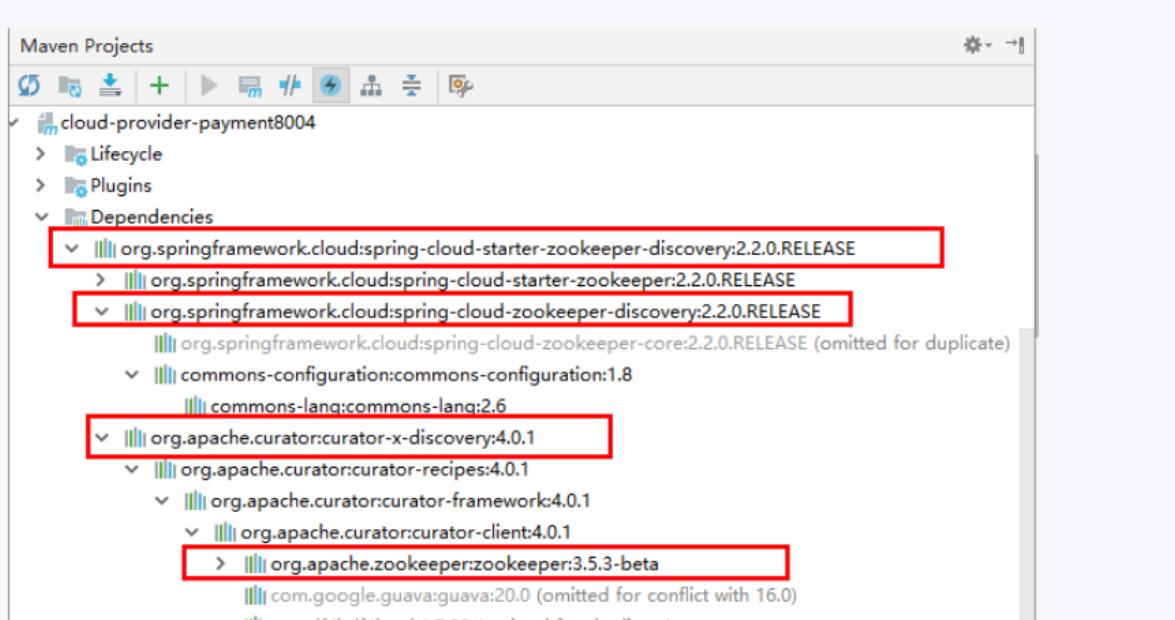
```

at org.springframework.boot.SpringApplication.run(SpringApplication.java:1215) [spring-boot-2.2.2.RELEASE.jar:2.2.2.RELEASE]
at com.atguigu.springcloud.PaymentMain8004.main(PaymentMain8004.java:17) [classes/:na] <4 internal calls>
at org.springframework.boot.devtools.restart.RestartLauncher.run(RestartLauncher.java:49) [spring-boot-devtools-2.2.2.RELEASE.jar:2.2.2.RELEASE]
Caused by: org.apache.zookeeper.KeeperException$UnimplementedException: KeeperErrorCode = Unimplemented for /services/cloud-provider-payment/4d21b38a-b16c-
at org.apache.zookeeper.KeeperException.create(KeeperException.java:103) ~[zookeeper-3.5.3-beta.jar:3.5.3-beta-8ce24f9e675cbefffb8f21a47e06b42864475a6]
at org.apache.zookeeper.KeeperException.create(KeeperException.java:51) ~[zookeeper-3.5.3-beta.jar:3.5.3-beta-8ce24f9e675cbefffb8f21a47e06b42864475a6]
at org.apache.zookeeper.ZooKeeper.create(ZooKeeper.java:1525) ~[zookeeper-3.5.3-beta.jar:3.5.3-beta-8ce24f9e675cbefffb8f21a47e06b42864475a6]
at org.apache.zookeeper.framework.imm.CuratorBuilder$17.call(CuratorBuilder.java:110) ~[curator-framework-1.0.1.jar:1.0.1]

```

这是因为zookeeper版本和jar包冲突造成的

```
zookeeper.KeeperException$UnimplementedException: KeeperErrorCode = Unimplemented for /services/s  
apache.zookeeper.KeeperException.create(KeeperException.java:103) [zookeeper-3.5.3-beta.jar|3.5.
```



解决方案：

如果zookeeper服务器中的版本是3.4.9，而原先的jar包版本是3.5.3，则可以在pom.xml文件重新引入另一个zookeeper版本。

```
<!-- 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--&gt;<br/><dependency>  
    <groupId>org.apache.zookeeper</groupId>  
    <artifactId>zookeeper</artifactId>  
    <version>3.4.9</version>  
</dependency>
```

8. 测试1

- 启动服务提供者后，在zookeeper客户端上可以发现多了一个services节点，该节点中有一个cloud-provider-payment

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

2. 在浏览器中访问: <http://localhost:8004/payment/zk>

springcloud with zookeeper: 8004 548792e8-28bd-4525-989e-0b2b0d0bc735

9. 测试2

在zookeeper客户端中发现:

```
[root@cloud-provider-payment ~]
[zk: localhost:2181(CONNECTED) 5] ls /services/cloud-provider-payment
[f69508fe-c50a-4b4f-9e1b-bf5c3e3bbce6]
[zk: localhost:2181(CONNECTED) 6]
```

```
1 [zk: localhost:2181(CONNECTED) 5] ls /services/cloud-provider-payment
2 [f69508fe-c50a-4b4f-9e1b-bf5c3e3bbce6]
```

那么这个是什么?

获取其中的节点数据, 可以发现如下信息:

```
1 [zk: localhost:2181(CONNECTED) 6] get /services/cloud-provider-payment/f69508fe-
c50a-4b4f-9e1b-bf5c3e3bbce6
2 {"name": "cloud-provider-payment", "id": "f69508fe-c50a-4b4f-9e1b-
bf5c3e3bbce6", "address": "DESKTOP-7CK6BAS", "port": 8004, "sslPort": null, "payload": {
    "@class": "org.springframework.cloud.zookeeper.discovery.ZookeeperInstance", "id": "application-1", "name": "cloud-provider-payment", "metadata": {}}, "registrationTimeUTC": 1650624733530, "serviceType": "DYNAMIC", "uriSpec": {"parts": [{"value": "scheme", "variable": true}, {"value": "://", "variable": false}, {"value": "address", "variable": true}, {"value": ":", "variable": false}, {"value": "port", "variable": true}]}}
3 [zk: localhost:2181(CONNECTED) 7]
```

上述json字符串即:

```
1 {
2     "name": "cloud-provider-payment",
3     "id": "f69508fe-c50a-4b4f-9e1b-bf5c3e3bbce6",
4     "address": "DESKTOP-7CK6BAS",
5     "port": 8004,
6     "sslPort": null,
7     "payload": {
8         "@class": "org.springframework.cloud.zookeeper.discovery.ZookeeperInstance",
9         "id": "application-1",
10        "name": "cloud-provider-payment",
11        "metadata": {}
12    },
13    "registrationTimeUTC": 1650624733530,
14    "serviceType": "DYNAMIC",
15    "uriSpec": {
16        "parts": [
17            {
18                "value": "scheme",
19                "variable": true
20            },
21            {
22                "value": "://",
23                "variable": false
24            }
25        ]
26    }
27}
```

```

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 }
```

10. 思考

服务节点是临时节点还是持久节点?

是临时节点!

服务消费者

1. 新建module

新建一个 **cloud-consumerzk-order80**

2. 改pom

```

1 <dependencies>
2     <!-- SpringBoot整合Web组件 -->
3     <dependency>
4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-starter-web</artifactId>
6     </dependency>
7     <!-- SpringBoot整合zookeeper客户端 -->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
11    </dependency>
12    <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>
```

4. 写YAML

```
1 server:
2   port: 80
3 spring:
4   application:
5     name: cloud-consumer-order
6   cloud:
7     #注册到zookeeper地址
8     zookeeper:
9       connect-string: 192.168.179.130:2181
```

5. 主启动

```
1 @SpringBootApplication
2 public class OrderZKMain80 {
3   public static void main(String[] args) {
4     SpringApplication.run(OrderZKMain80.class, args);
5   }
6 }
```

6. 业务类

1. 配置Bean

```
1 @Configuration
2 public class ApplicationContextConfig {
3
4   @Bean
5   @LoadBalanced
6   public RestTemplate getRestTemplate() {
7     return new RestTemplate();
8   }
9 }
```

2. controller

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

7. 测试

1. 将zookeeper启动
2. 将服务提供者启动
3. 将服务消费者启动

在zookeeper客户端上可以发现，服务消费者成功注册进去了

```
[zk: localhost:2181(CONNECTED) 11] ls /services
[cloud-consumer-order, cloud-provider-payment]
[zk: localhost:2181(CONNECTED) 12]
```

4. 在浏览器中，访问地址：<http://localhost/consumer/payment/zk>

成功显示：

```
springcloud with zookeeper: 8004 b3f45d3e-acef-4c5f-9ce1-3456df5e80fa
```

五、Consul 服务注册与发现

Consul简介

官网：<https://www.consul.io/docs/intro>

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 requires a data plane and supports both a proxy and native integration model. 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.

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

Consul 具有如下特性：

The key features of Consul are:

- **Service Discovery:** Clients of Consul can register a service, such as `api` or `mysql`, and other clients can use Consul to discover providers of a given service. Using either DNS or HTTP, applications can easily find the services they depend upon.
- **Health Checking:** Consul clients can provide any number of health checks, either associated with a given service ("is the webserver returning 200 OK"), or with the local node ("is memory utilization below 90%"). This information can be used by an operator to monitor cluster health, and it is used by the service discovery components to route traffic away from unhealthy hosts.
- **KV Store:** Applications can make use of Consul's hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination, leader election, and more. The simple HTTP API makes it easy to use.
- **Secure Service Communication:** Consul can generate and distribute TLS certificates for services to establish mutual TLS connections. **Intentions** can be used to define which services are allowed to communicate. Service segmentation can be easily managed with intentions that can be changed in real time instead of using complex network topologies and static firewall rules.
- **Multi Datacenter:** Consul supports multiple datacenters out of the box. This means users of Consul do not have to worry about building additional layers of abstraction to grow to multiple regions.

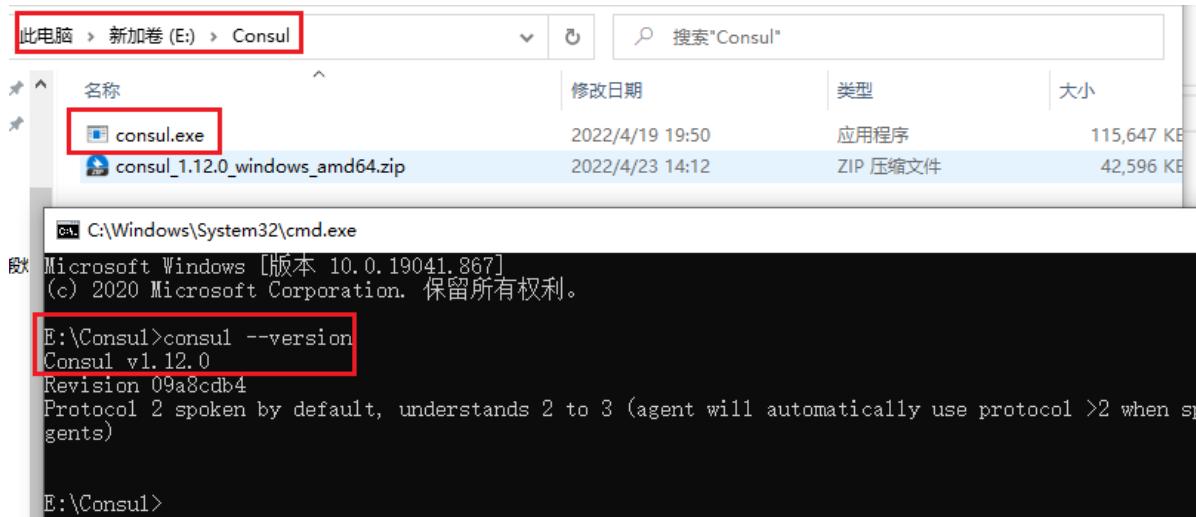
Consul中文教程: <https://www.springcloud.cc/spring-cloud-consul.html>

安装并运行Consul

<https://www.consul.io/downloads>

这里我下载的是windows系统下的Consul

下载完成后，是一个zip压缩包。解压后，利用cmd命令可以查看版本号



使用开发模式启动：

```
1 consul agent -dev
```

启动后，通过以下地址可以访问Consul的首页：<http://localhost:8500>

访问后，页面如下：

The screenshot shows the 'Cluster Overview' page of the Consul UI. On the left is a sidebar with navigation links: Overview, Services, Nodes, Key/Value, Intentions, ACCESS CONTROLS (with a red dot), Tokens, Policies, Roles, Auth Methods, and a footer note 'Consul v1.12.0'. The main content area is titled 'Cluster Overview' and contains two sections: 'Server fault tolerance' and 'Servers'. Under 'Server fault tolerance', it says 'Immediate' and 'the number of healthy active voting servers that can fail at once without causing an outage' with a value of '0'. Under 'Servers', there is a card for 'DESKTOP-7...' which is marked as an 'Active voter'.

服务提供者

1. 新建module

新建 `cloud-providerconsul-payment8006` 作为服务提供者

2. 改POM

```
1 <dependencies>
2     <!--SpringCloud consul-server -->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-consul-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>
```

3. 写YAML

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

4. 主启动

```
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

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

6. 测试

1. 启动consul
2. 启动8006微服务
3. 访问consul的页面: <http://localhost:8500>

The screenshot shows the Consul UI interface. On the left, there's a sidebar with options like Overview, Services (which is selected and highlighted with a red box), Nodes, Key/Value, Intentions, ACCESS CONTROLS, Tokens, Policies, Roles, and Auth Methods. The main area is titled 'Services' and shows '2 total'. It includes a search bar and filters for Health Status (set to 'Healthy') and Service Type. Two services are listed: 'consul' (1 instance, healthy) and 'consul-provider-payment' (1 instance, healthy). A red box highlights the 'consul-provider-payment' entry.

4. 访问: <http://localhost:8006/payment/consul>

springcloud with consul: 8006 a9ae7b26-71e5-48ad-ae3b-8db6a0ac7320

服务消费者

1. 新建module

新建 `cloud-consumerconsul-order80` 模块作为服务消费者

2. 改POM

```
1 <dependencies>
2     <!--SpringCloud consul-server -->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-consul-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>
```

3. 写YAML

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

4. 主启动

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class OrderConsulMain80 {
4     public static void main(String[] args) {
5         SpringApplication.run(OrderConsulMain80.class, args);
6     }
7 }
```

5. 配置Bean

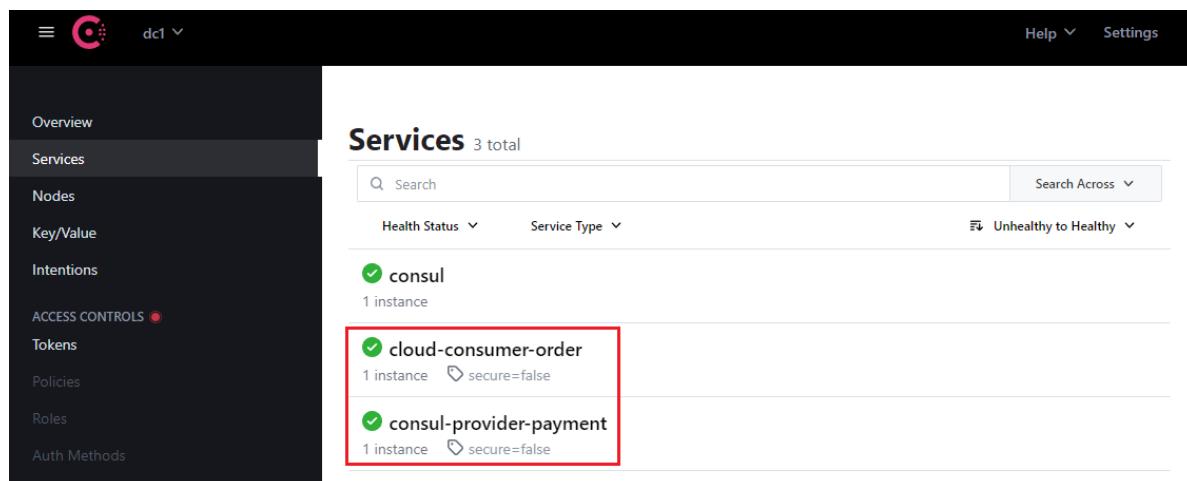
```
1 @Configuration
2 public class ApplicationContextConfig {
3     @Bean
4     @LoadBalanced
5     public RestTemplate getRestTemplate() {
6         return new RestTemplate();
7     }
8 }
```

6. 业务类Controller

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

7. 测试

1. 启动consul
2. 启动服务提供者和服务消费者
3. 查看consul页面，可以看到：



The screenshot shows the Consul UI interface. On the left is a sidebar with navigation links: Overview, Services (which is selected), Nodes, Key/Value, Intentions, ACCESS CONTROLS (with a red dot), Tokens, Policies, Roles, and Auth Methods. The main area is titled 'Services' and shows '3 total'. It includes a search bar and filters for 'Health Status' (set to 'Unhealthy to Healthy') and 'Service Type'. Below the filters, there is a table with three rows:

Service	Status	Instances	Secure
consul	green checkmark	1 instance	
cloud-consumer-order	green checkmark	1 instance	secure=false
consul-provider-payment	green checkmark	1 instance	secure=false

The last two rows, 'cloud-consumer-order' and 'consul-provider-payment', are highlighted with a red box.

4. 在浏览器中访问：<http://localhost/consumer/payment/consul>

springcloud with consul: 8006 f4d58197-fcd1-4554-ae91-299492b6d5ec

六、注册中心总结

在上述，我们学了 Eureka、Zookeeper 和 Consul 三种注册中心。

以下是它们的异同：

组件名	语言	CAP	服务健康检查	对外暴露接口	SpringCloud集合
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理论是分布式系统、特别是分布式存储领域中被讨论的最多的理论。其中C代表一致性 (Consistency)，A代表可用性 (Availability)，P代表分区容错性 (Partition tolerance)。CAP理论告诉我们C、A、P三者不能同时满足，最多只能满足其中两个。

- **一致性 (Consistency)** :一个写操作返回成功，那么之后的读请求都必须读到这个新数据；如果返回失败，那么所有读操作都不能读到这个数据。所有节点访问同一份最新的数据。
- **可用性 (Availability)** :对数据更新具备高可用性，请求能够及时处理，不会一直等待，即使出现节点失效。
- **分区容错性 (Partition tolerance)** :能容忍网络分区，在网络断开的情况下，被分隔的节点仍能正常对外提供服务。

对CAP理论的理解：

理解CAP理论最简单的方式是想象两个副本处于分区两侧，即两个副本之间的网络断开，不能通信。

- 如果允许其中一个副本更新，则会导致数据不一致，即丧失了C性质。
- 如果为了保证一致性，将分区某一侧的副本设置为不可用，那么又丧失了A性质。
- 除非两个副本可以互相通信，才能既保证C又保证A，这又会导致丧失P性质。

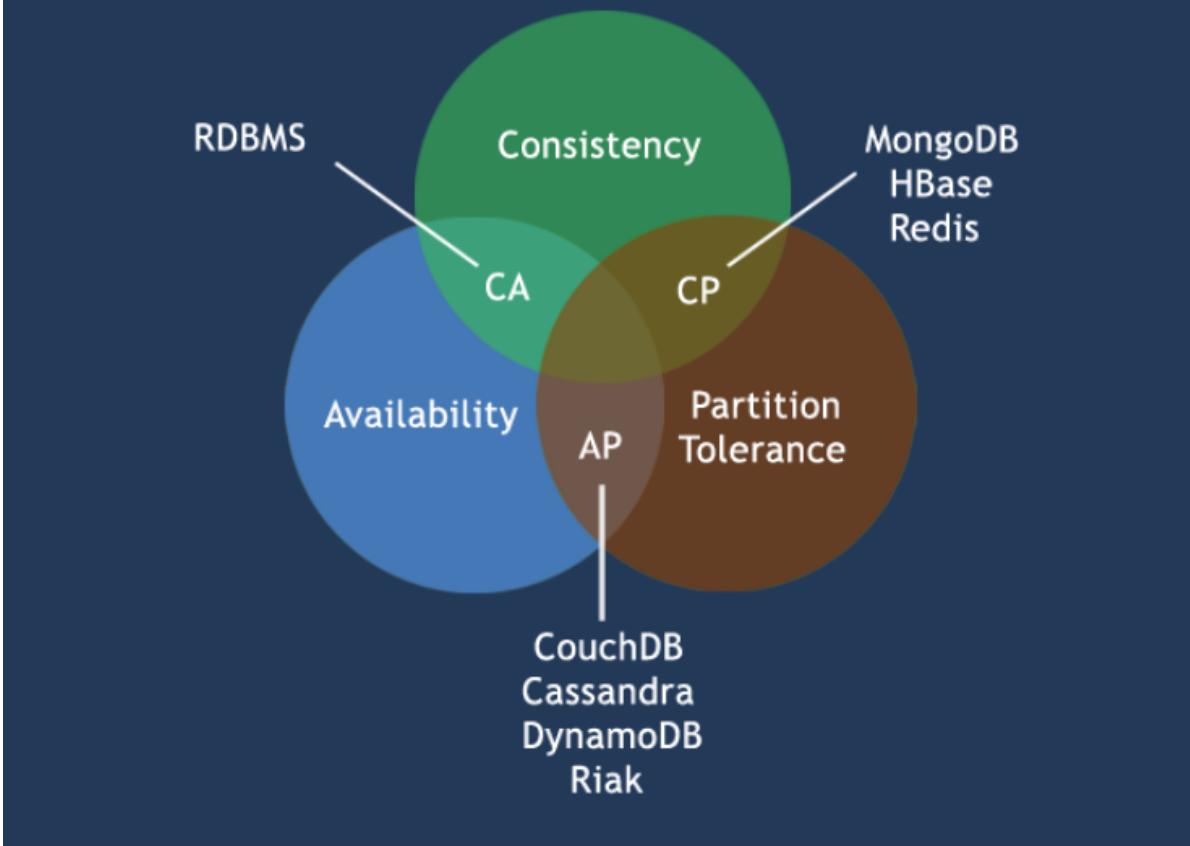
一般来说使用网络通信的分布式系统，无法舍弃P性质，那么就只能在一致性和可用性上做一个艰难的选择。

CAP理论关注粒度是数据，而不是整体系统设计的。

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，分区容忍性的系统，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。

CAP Theorem

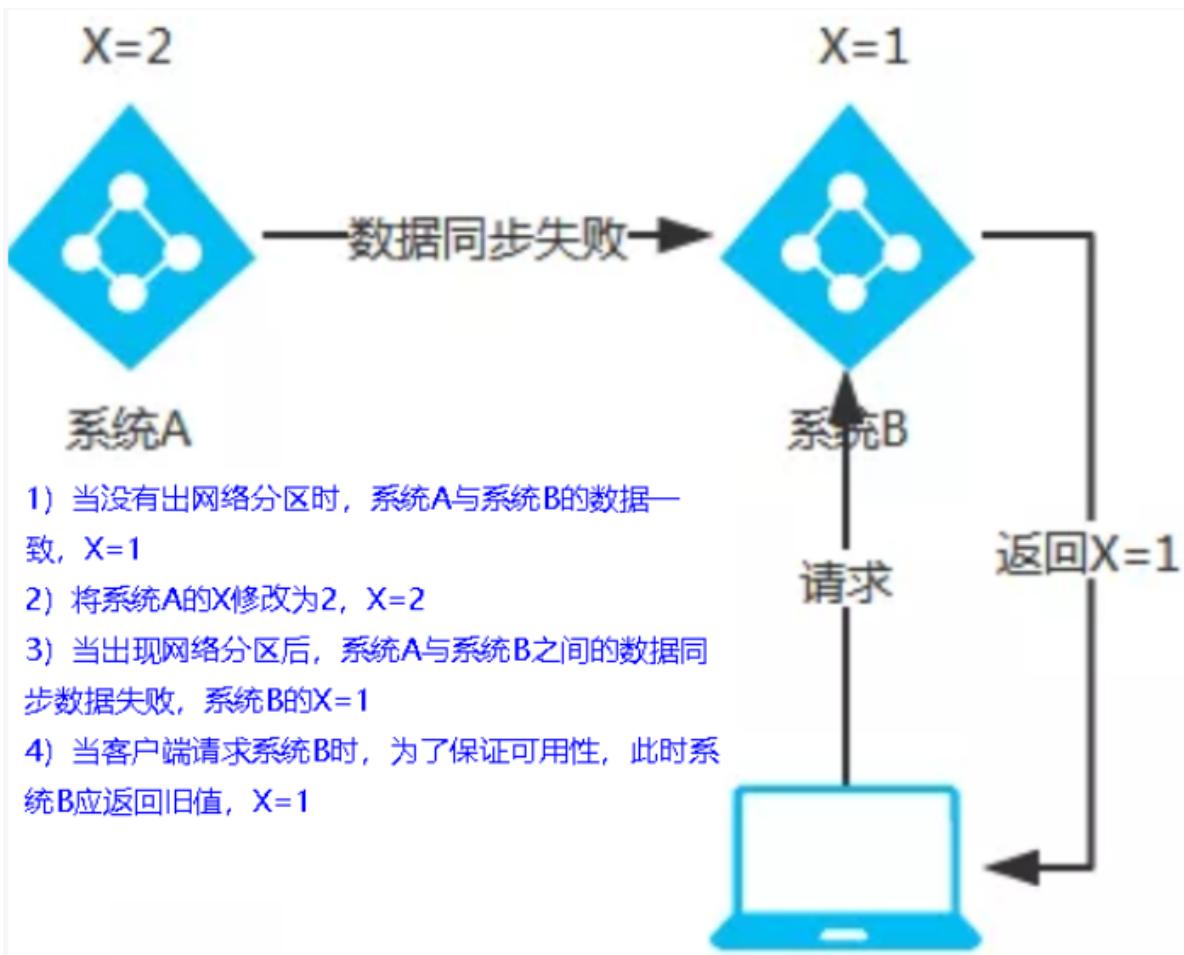


在注册中心三大组件中，Eureka满足AP，Zookeeper / Consul 满足CP。

1. AP架构

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。

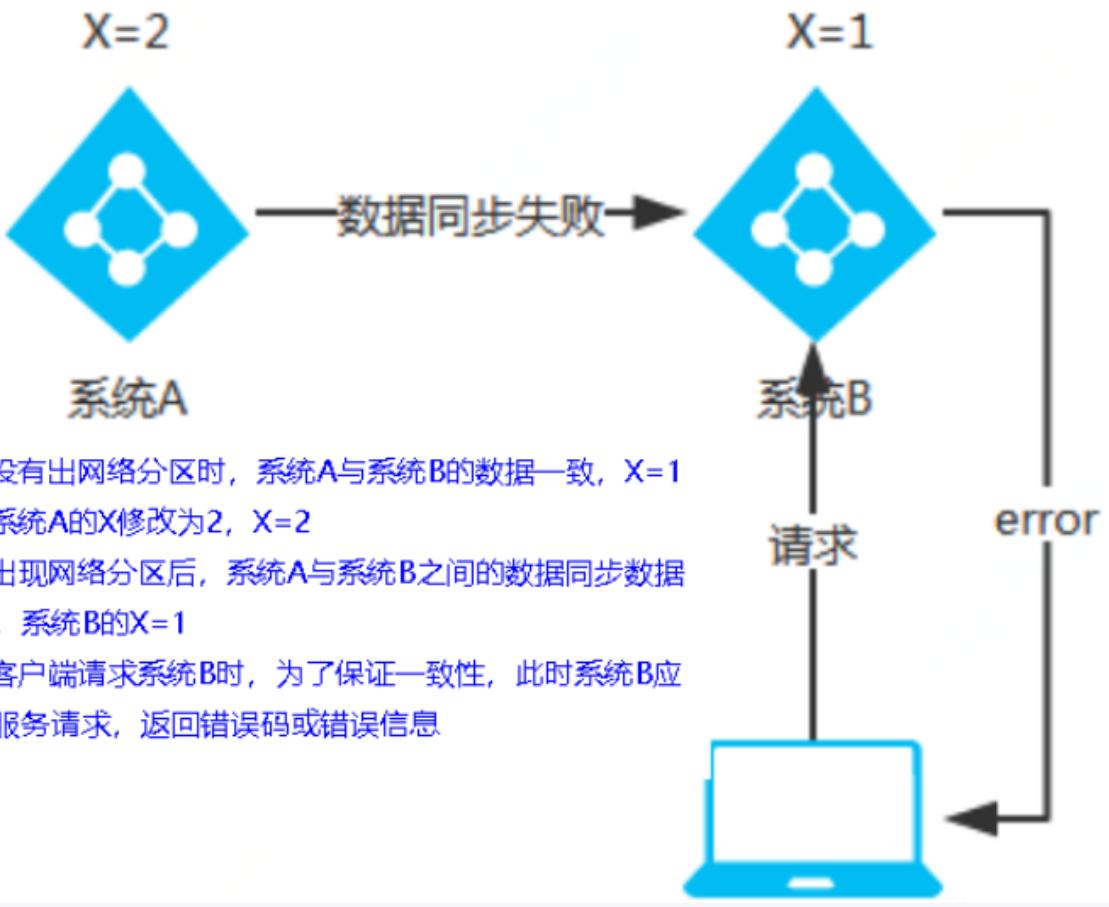
结论：违背了一致性C的要求，只满足可用性和分区容错，即AP



2. CP架构

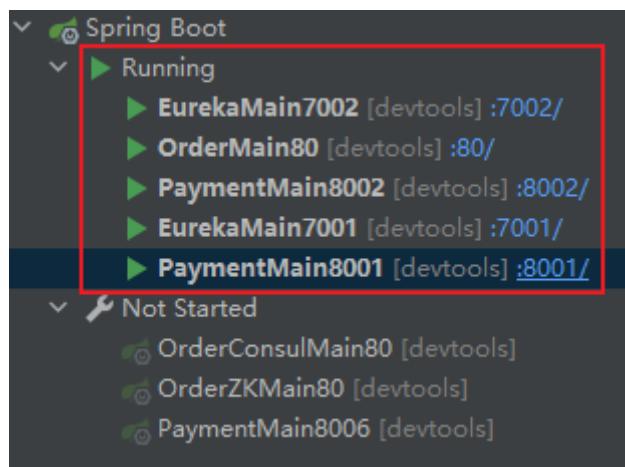
当网络分区出现后，为了保证一致性，就必须拒接请求，否则无法保证一致性

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



七、 Ribbon 负载均衡服务调用

为了演示负载均衡，这里还原了之前Eureka集群以及当时的代码。



概述

Ribbon 是什么

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。

官网

Ribbon： <https://github.com/Netflix/ribbon>

Ribbon目前进入了维护模式！

The screenshot shows a portion of the [GitHub README.md](https://github.com/Netflix/ribbon) page for the Netflix/ribbon repository. A red box highlights the heading **Project Status: On Maintenance**. Below it, a paragraph explains the status: "Ribbon comprises of multiple components some of which are used in production internally and some of which were replaced by non-OSS solutions over time. This is because Netflix started moving into a more componentized architecture for RPC with a focus on single-responsibility modules. So each Ribbon component gets a different level of attention at this moment." Another red box highlights a note: "Recognizing these realities and deficiencies, we are placing Ribbon in maintenance mode. This means that if an external user submits a large feature request, internally we wouldn't prioritize it highly. However, if someone were to do work on their own and submit complete pull requests, we'd be happy to review and accept. Our team has instead started building an RPC solution on top of gRPC. We are doing this transition for two main reasons: multi-language support and better extensibility/composability through request interceptors. That's our current plan moving forward."

Ribbon 的作用

Ribbon主要作用是负载均衡(LB)

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服务端负载均衡的区别？

- 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。

负载均衡主要分为两种：

1. 集中式负载均衡

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；

2. 进程内负载均衡

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。

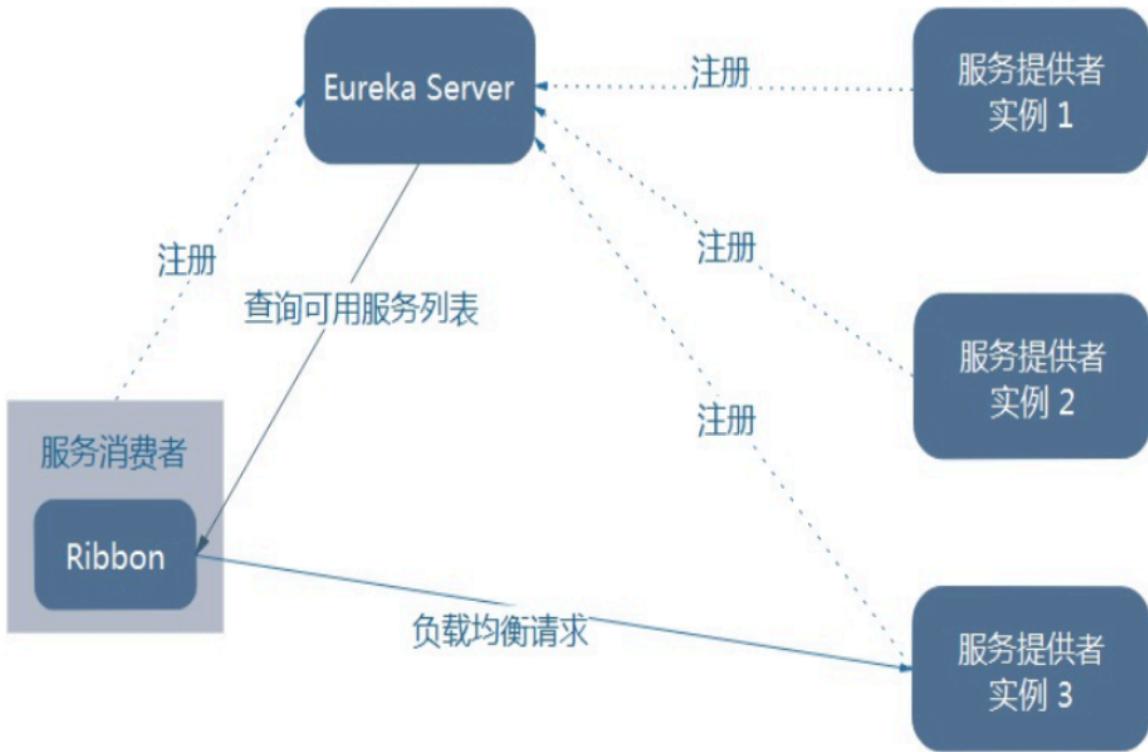
之前我们在Eureka中搭建集群，让服务消费者通过轮询负载访问8001/8002，就是负载均衡。

小结

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结合只是其中一个实例。

Ribbon负载均衡演示

架构说明



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。

POM

在模块 `cloud-consumer-order80` 模块的pom.xml中，我们发现我们没有引入`spring-cloud-starter-ribbon`也可以使用ribbon，

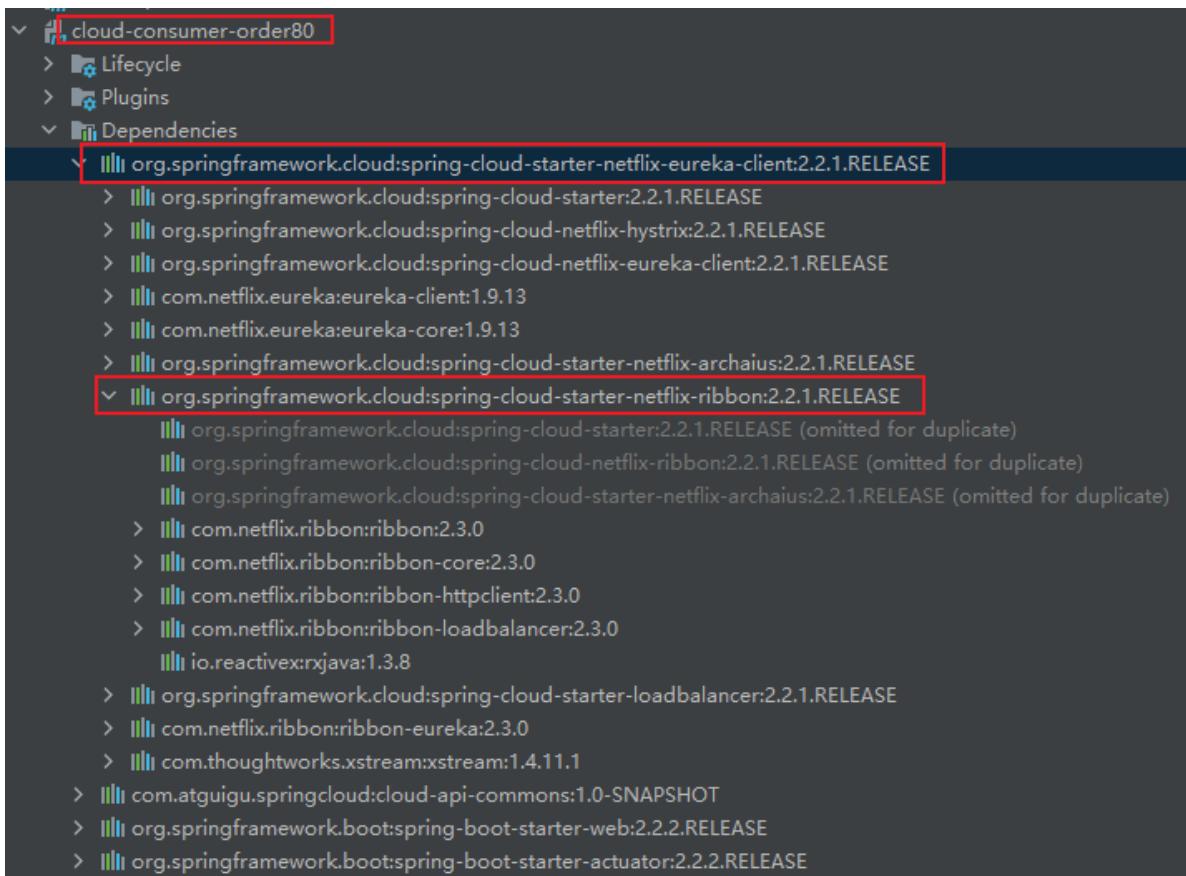
```

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

```

这是因为：

`spring-cloud-starter-netflix-eureka-client`自带了`spring-cloud-starter-ribbon`引用



RestTemplate 的使用

RestTemplate 官方文档: <https://docs.spring.io/spring-framework/docs/5.2.2.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html>

getForObject方法/getForEntity方法

我们先演示一下 getForObject方法/getForEntity方法。

<T> ResponseEntity<T>	<code>getForEntity(String url, Class<T> responseType, Map<String,?> uriVariables)</code> Retrieve a representation by doing a GET on the URI template.
<T> ResponseEntity<T>	<code>getForEntity(String url, Class<T> responseType, Object... uriVariables)</code> Retrieve an entity by doing a GET on the specified URL.
<T> ResponseEntity<T>	<code>getForEntity(URI url, Class<T> responseType)</code> Retrieve a representation by doing a GET on the URL .
<T> T	<code>getForObject(String url, Class<T> responseType, Map<String,?> uriVariables)</code> Retrieve a representation by doing a GET on the URI template.
<T> T	<code>getForObject(String url, Class<T> responseType, Object... uriVariables)</code> Retrieve a representation by doing a GET on the specified URL.
<T> T	<code>getForObject(URI url, Class<T> responseType)</code> Retrieve a representation by doing a GET on the URL .

1. getForObject方法

返回对象为响应体中数据转换成的对象，基本上可以理解为JSON

```
@GetMapping("支付对象/{id}")
public CommonResult<Payment> getForObject(@PathVariable("id") Long id) {
    return restTemplate.getForObject(url: PAYMENT_URL + "/payment/get/" + id, CommonResult.class);
}
```

2. getForEntity方法

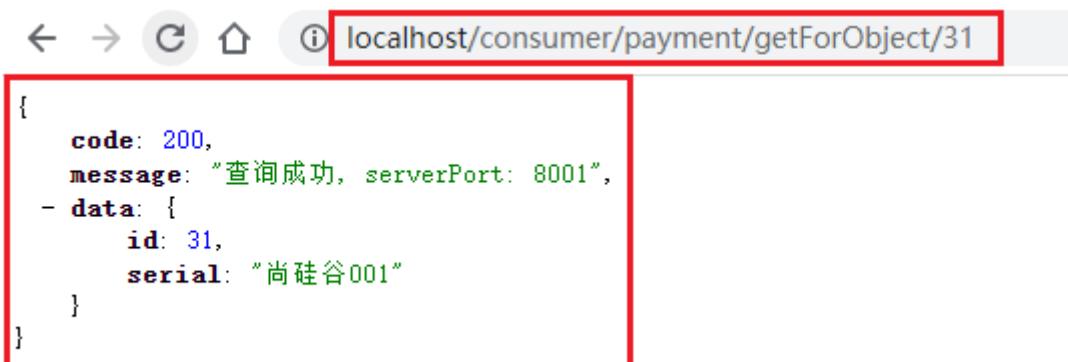
返回对象为 ResponseEntity 对象，包含了响应中的一些重要信息，比如响应头、响应状态码、响应体等

```
@GetMapping("支付对象/{id}")
public CommonResult<Payment> getForEntity(@PathVariable("id") Long id) {
    ResponseEntity<CommonResult> entity = restTemplate.getForEntity(url: PAYMENT_URL + "/payment/get/" + id, CommonResult.class);
    if (entity.getStatusCode().is2xxSuccessful()) {
        return entity.getBody();
    } else {
        return new CommonResult<>(code: 444, message: "操作失败");
    }
}
```

按上述两种方法写出两个http接口后，进行测试。

测试：

1. 浏览器访问：<http://localhost/consumer/payment/getForObject/31>



2. 浏览器访问：<http://localhost/consumer/payment/getForEntity/31>



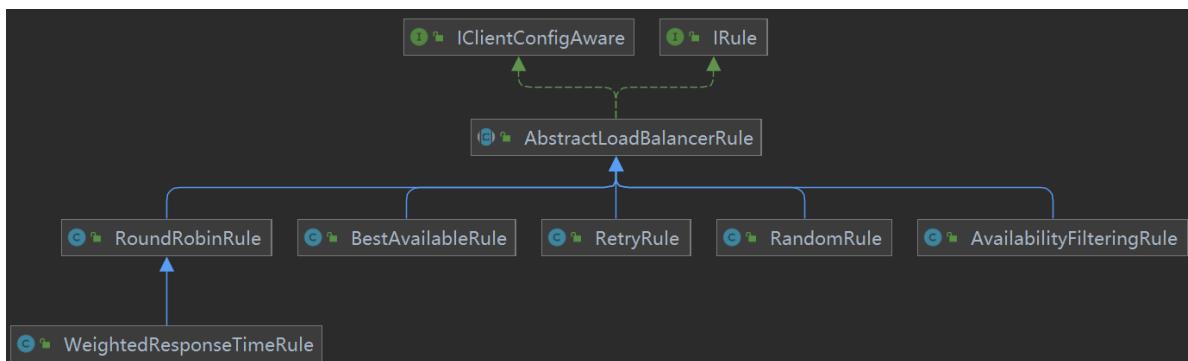
Ribbon 核心组件 IRule

IRule：根据特定算法从服务列表中选取一个要访问的服务。

Interface that defines a "Rule" for a LoadBalancer. A Rule can be thought of as a Strategy for loadbalancing. Well known loadbalancing strategies include Round Robin, Response Time based etc.

Author: stonse

```
public interface IRule{  
    /*  
     * choose one alive server from lb.allServers or  
     * lb.upServers according to key  
     *  
     * @return chosen Server object. NULL is returned if none  
     * server is available  
     */  
  
    public Server choose(Object key);  
  
    public void setLoadBalancer(ILoadBalancer lb);  
  
    public ILoadBalancer getLoadBalancer();  
}
```



实现`IRule`接口的类主要有：

- `RoundRobinRule`：轮询
- `RandomRule`：随机
- `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，获取可用的服务
- `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

在之前 `cloud-consumer-order80` 中，我们默认的Ribbon规则是 `RoundRobinRule` 轮询。

下面我们学习一下如何替换Ribbon负载规则！

主要是对 `cloud-consumer-order80` 模块进行修改！需要创建一个配置类！

0. 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。

Customizing the Ribbon Client

You can configure some bits of a Ribbon client using external properties in `<client>.ribbon.*`, which is no different than using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in `CommonClientConfigKey` (part of `ribbon-core`).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`. Example:

```
@Configuration  
{@RibbonClient(name = "foo", configuration = FooConfiguration.class)  
public class TestConfiguration {  
}}
```

In this case the client is composed from the components already in `RibbonClientConfiguration` together with any in `FooConfiguration` (where the latter generally will override the former).

WARNING

The `FooConfiguration` has to be `@Configuration` but take care that it is not in a `@ComponentScan` for the main application context, otherwise it will be shared by all the `@RibbonClients`. If you use `@ComponentScan` (or `@SpringBootApplication`) you need to take steps to avoid it being included (for instance put it in a separate, non-overlapping package, or specify the packages to scan explicitly in the `@ComponentScan`).

1. 新建package

由于配置细节中发现，官方说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。因此，我们自定义一个package名为 `com/atguigu/myrule`，这样就与主启动类不在同一包下！

2. 在 `com/atguigu/myrule` 下新建 `MySelfRule` 类

```
1  @Configuration  
2  public class MySelfRule {  
3      @Bean  
4      public IRule myRule() {  
5          return new RandomRule();    // 定义为随机选择  
6      }  
7  }
```

3. 主启动类上添加`@RibbonClient`注解

```

1  @SpringBootApplication
2  @EnableEurekaClient
3  @RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)
4  public class OrderMain80 {
5      public static void main(String[] args) {
6          SpringApplication.run(OrderMain80.class, args);
7      }
8  }

```

4. 测试

用浏览器访问: <http://localhost/consumer/payment/get/31>

Ribbon负载均衡算法

以 RoundRobinRule 轮询规则为例!

原理

举例：目前Eureka中已注册的服务提供者有2个。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002 , payment8001

为什么能实现轮询？

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，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

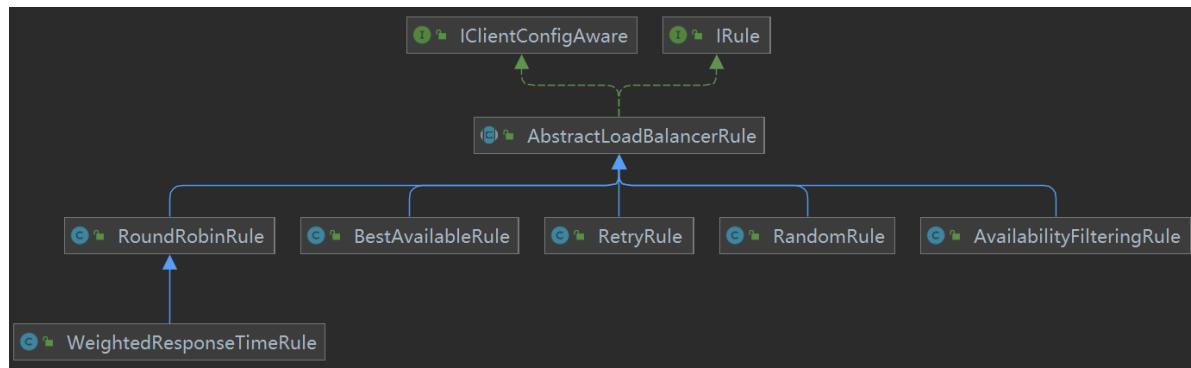
List [1] instances = 127.0.0.1:8001

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



RoundRobinRule源码：

```
public class RoundRobinRule extends AbstractLoadBalancerRule {

    private AtomicInteger nextServerCyclicCounter;
    private static final boolean AVAILABLE_ONLY_SERVERS = true;
    private static final boolean ALL_SERVERS = false;

    private static Logger log = LoggerFactory.getLogger(RoundRobinRule.class);

    public RoundRobinRule() {
        nextServerCyclicCounter = new AtomicInteger( initialValue: 0);
    }

    public RoundRobinRule(ILoadBalancer lb) {
        this();
        setLoadBalancer(lb);
    }
}
```

RoundRobinRule中选择服务器的规则是通过 `choose()` 方法实现的：

```

public Server choose(ILoadBalancer lb, Object key) {
    if (lb == null) {
        log.warn("no load balancer");
        return null;
    }
    Server server = null;
    int count = 0;
    while (server == null && count++ < 10) {
        List<Server> reachableServers = lb.getReachableServers();
        List<Server> allServers = lb.getAllServers();
        int upCount = reachableServers.size();
        int serverCount = allServers.size();

        if ((upCount == 0) || (serverCount == 0)) {
            log.warn("No up servers available from load balancer: " + lb);
            return null;
        }

        int nextServerIndex = incrementAndGetModulo(serverCount);
        server = allServers.get(nextServerIndex);

        if (server == null) {
            /* Transient. */
            Thread.yield();
            continue;
        }

        if (server.isAlive() && (server.isReadyToServe())) {
            return (server);
        }
        // Next.
        server = null;
    }
    if (count >= 10) {
        log.warn("No available alive servers after 10 tries from load balancer: "
                + lb);
    }
    return server;
}

```

手写负载均衡算法

试着写一个本地负载均衡器

在写之前先对8001/8002微服务进行改造，在它们的controller中新建一个http接口，用于后续测试：

```

@GetMapping("/lb")
public String getPaymentLB() {
    return serverPort;
}

```

然后对 `cloud-consumer-order80` 进行如下改造

1. 去掉@LoadBalanced注解

在 `ApplicationContextBean` 类中去掉注解`@LoadBalanced`

2. 新建LoadBalancer接口

```
1  public interface LoadBalancer {
2      ServiceInstance instances(List<ServiceInstance> serviceInstances);
3  }
```

3. 新建实现类

新建一个实现类 `MyLB` 来实现 `LoadBalancer` 接口

```
1  @Component
2  public class MyLB implements LoadBalancer{
3
4      private AtomicInteger atomicInteger = new AtomicInteger(0);
5
6      public final int getAndIncrement() {
7          int current;
8          int next;
9          do {
10              current = this.atomicInteger.get();
11              next = current >= Integer.MAX_VALUE ? 0 : current + 1;
12          } while (!this.atomicInteger.compareAndSet(current, next));
13          System.out.println("*****第几次访问，次数next: " + next);
14          return next;
15      }
16
17      @Override
18      public ServiceInstance instances(List<ServiceInstance> serviceInstances) {
19          int index = getAndIncrement() % serviceInstances.size();
20          return serviceInstances.get(index);
21      }
22  }
```

4. OrderController修改

在 `OrderController` 中添加如下代码：

```

public class OrderController {
    // public static final String PAYMENT_URL = "http://localhost:8001";
    public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";

    @Resource
    private RestTemplate restTemplate;

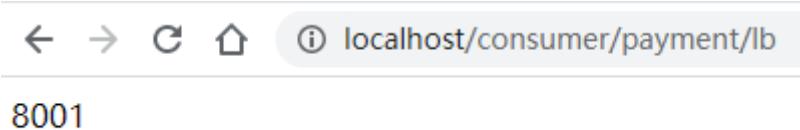
    @Resource
    private LoadBalancer loadBalancer;
    @Resource
    private DiscoveryClient discoveryClient;

    @GetMapping("/payment/lb")
    public String getPaymentLB() {
        List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
        if (instances == null || instances.size() <= 0) {
            return null;
        }
        ServiceInstance serviceInstance = loadBalancer.instances(instances);
        URI uri = serviceInstance.getUri();
        return restTemplate.getForObject(uri + "/payment/lb", String.class);
    }
}

```

5. 测试

- 启动eureka集群和8001/8002这两个微服务
- 启动 cloud-consumer-order80
- 在浏览器中访问: <http://localhost/consumer/payment/lb>



八、OpenFeign服务接口调用

已经有Ribbon了，为什么还要出一个OpenFeign？

Feign和OpenFeign的区别？

简介

OpenFeign是什么

官网：<https://cloud.spring.io/spring-cloud-static/Hoxton.SR1/reference/htmlsingle/#spring-cloud-openfeign>

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。

6. Spring Cloud OpenFeign

Hoxton.SR1

This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

6.1. Declarative REST Client: Feign

[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.

6.1.1. How to Include Feign

To include Feign in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-openfeign`. See the [Spring Cloud Project page](#) for details on setting up your build system with the current Spring Cloud Release Train.

总的来说，Feign是一个声明式的Web服务客户端，让编写Web服务客户端变得非常容易，**只需创建一个接口并在接口上添加注解即可。**

Feign能干什么

已经有Ribbon可以进行负载均衡了，为什么还要出一个OpenFeign？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 的区别

Feign	OpenFeign
<p>Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端</p> <p>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。</p> <p>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</p>	<p>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。</p>
<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter- feign</artifactId> </dependency></pre>	<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-openfeign</artifactId> </dependency></pre>

OpenFeign 使用步骤

OpenFeign使用就是：接口+注解（微服务调用接口+@FeignClient注解）

1. 新建服务消费者模块

Feign是在消费端使用的。

因此，新建一个 `cloud-consumer-feign-order80` 模块

2. 改POM

```

1 <dependencies>
2   <!--openfeign-->
3   <dependency>
4     <groupId>org.springframework.cloud</groupId>
5     <artifactId>spring-cloud-starter-openfeign</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>com.atguigu.springcloud</groupId>
15    <artifactId>cloud-api-commons</artifactId>
16    <version>${project.version}</version>
17  </dependency>
18  <!--web-->
19  <dependency>
20    <groupId>org.springframework.boot</groupId>
```

```
21         <artifactId>spring-boot-starter-web</artifactId>
22     </dependency>
23     <dependency>
24         <groupId>org.springframework.boot</groupId>
25         <artifactId>spring-boot-starter-actuator</artifactId>
26     </dependency>
27     <!--一般基础通用配置-->
28     <dependency>
29         <groupId>org.springframework.boot</groupId>
30         <artifactId>spring-boot-devtools</artifactId>
31         <scope>runtime</scope>
32         <optional>true</optional>
33     </dependency>
34     <dependency>
35         <groupId>org.projectlombok</groupId>
36         <artifactId>lombok</artifactId>
37         <optional>true</optional>
38     </dependency>
39     <dependency>
40         <groupId>org.springframework.boot</groupId>
41         <artifactId>spring-boot-starter-test</artifactId>
42         <scope>test</scope>
43     </dependency>
44 </dependencies>
```

3. 编写YAML

```
1 server:
2   port: 80
3 eureka:
4   client:
5     register-with-eureka: false
6     service-url:
7       defaultZone:
8         http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
9
```

4. 主启动

```
1 @SpringBootApplication
2 @EnableFeignClients // 激活并开启Feign
3 public class OrderFeignMain80 {
4     public static void main(String[] args) {
5         SpringApplication.run(OrderFeignMain80.class, args);
6     }
7 }
```

注意注解 @EnableFeignClients

5. 业务类

业务逻辑接口 + @FeignClient配置调用provider服务

新建接口并新增注解@FeignClient

1. 新建一个 com.atguigu.springcloud.service 包
2. 在包下新建一个接口 PaymentFeignService

```
1  @Component
2  @FeignClient(value = "CLOUD-PAYMENT-SERVICE") // 指定要调用的服务名
3  public interface PaymentFeignService {
4      // 这里定义的方法 是 服务提供者中 controller 中的!
5      // 相当于把服务提供者中 controller 层的方法直接复制过来用!
6      @GetMapping("/payment/get/{id}")
7      CommonResult getPaymentById(@PathVariable("id") Long id);
8  }
```

注意：

1. 使用 @FeignClient 前，一定要在主启动类上添加 @EnableFeignClients
2. 一定要在接口对应的方法上添加@GetMapping等注解！并且请求参数要求和服务提供者上的完全一致！不然报错！

新建controller

```
1  @RestController
2  @Slf4j
3  public class OrderFeignController {
4
5      @Resource
6      private PaymentFeignService paymentFeignService;
7
8      @GetMapping("/consumer/payment/get/{id}")
9      public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id) {
10         return paymentFeignService.getPaymentById(id);
11     }
12 }
13 }
```

6. 测试

1. 先启动2个rureka集群7001/7002
2. 再启动两个微服务提供者8001/8002
3. 再启动服务消费者80
4. 在浏览器中访问：



注意：这里进行多次访问时会发现它默认具有负载均衡功能！默认规则是 RoundRobinRule ！

小总结

Application	注册中心中已注册的服务名	AMIs
CLOUD-PAYMENT-SERVICE		n/a (2)

OpenFeign 超时控制

默认Feign客户端只等待一秒钟，但是如果服务端处理需要超过1秒钟，会导致Feign客户端不想等待了，直接返回报错。

为了避免这样的情况，有时候我们需要设置Feign客户端的超时控制。

演示超时出错的情况

1. 服务提供方8001故意写暂停程序

在8001的controller中加上如下代码即可

```
@GetMapping("payment/feign/timeout")
public String paymentFeignTimeout() {
    try {
        TimeUnit.SECONDS.sleep(timeout: 3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return serverPort;
}
```

2. 服务消费方80在 `PaymentFeignService` 中添加超时方法

```
@Component
@FeignClient(value = "CLOUD-PAYMENT-SERVICE") // 指定要调用的服务名
public interface PaymentFeignService {
    // 这里定义的方法 是 服务提供者中 controller 中的!
    // 相当于把服务提供者中 controller 层的方法直接复制过来用!
    @GetMapping("payment/get/{id}")
    CommonResult<Payment> getPaymentById(@PathVariable("id") Long id);

    @GetMapping("payment/feign/timeout")
    String paymentFeignTimeout();
}
```

3. 服务消费者80在 `OrderFeignController` 类中添加超时方法

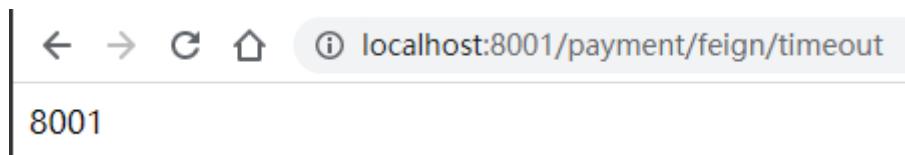
```
@RestController
@Slf4j
public class OrderFeignController {
    @Resource
    private PaymentFeignService paymentFeignService;
    @GetMapping("consumer/payment/get/{id}")
    public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id) {
        return paymentFeignService.getPaymentById(id);
    }

    @GetMapping("consumer/payment/feign/timeout")
    public String paymentFeignTimeout() {
        return paymentFeignService.paymentFeignTimeout();
    }
}
```

4. 测试

- 先在浏览器中直接访问服务提供者的接口：<http://localhost:8001/payment/feign/timeout>

虽然耗时长一点，但是可以访问



- 在浏览器中用服务消费者的接口调用服务提供者：

发现直接报错！

The screenshot shows a browser displaying a "Whitelabel Error Page". The page content includes:

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Apr 26 10:57:47 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

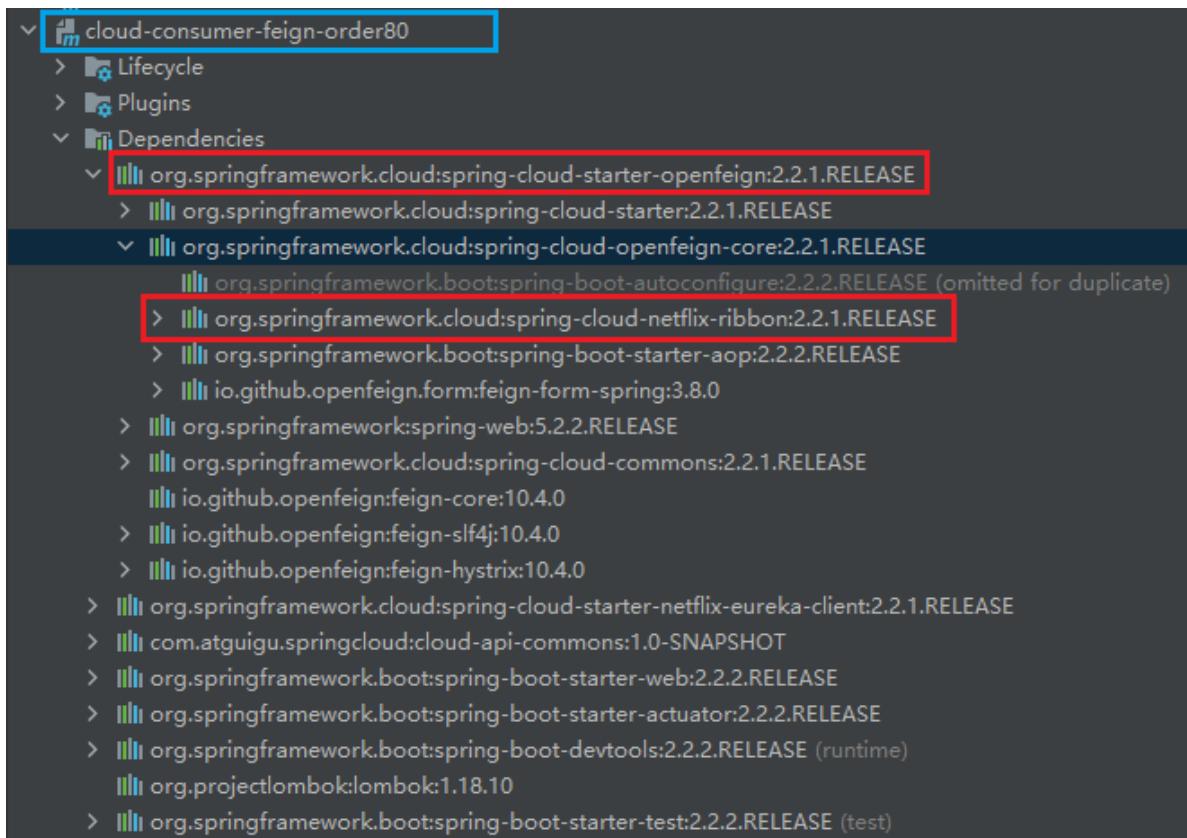
```
status 404 reading PaymentFeignService#paymentFeignTimeout()  
feign.FeignException$NotFound: status 404 reading PaymentFeignService#paymentFeignTimeout()  
at feign.FeignException.clientErrorStatus(FeignException.java:165)  
at feign.FeignException.errorStatus(FeignException.java:141)  
at feign.FeignException.errorStatus(FeignException.java:133)  
at feign.codec.ErrorDecoder$Default.decode(ErrorDecoder.java:92)  
at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:151)  
at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:80)
```

如上演示可以知道：

OpenFeign默认等待1秒钟，超过后报错！

开启超时控制

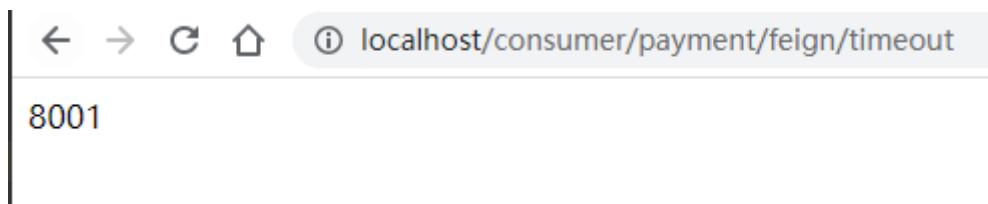
在我们使用OpenFeign的时候，其实也整合了Ribbon。 **OpenFeign的超时控制，也是由Ribbon来进行控制的！**



所以，进行超时控制需要在 `cloud-consumer-feign-order80` 中的 `application.yml` 文件加入如下配置：

```
1 # 设置feign客户端超时时间（OpenFeign默认支持ribbon）
2 ribbon:
3     # 指的是建立连接后从服务器读取到可用资源所用的时间，单位ms
4     ReadTimeout: 5000
5     # 指的是建立链接所用的时间，适用于网络状况正常的情况下，两端连接所用的时间，单位ms
6     ConnectTimeout: 5000
7
```

之后测试，即可访问服务提供者：



OpenFeign 日志打印功能

日志打印功能是什么

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

OpenFeign 的日志级别有：

- **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

配置及使用

1. 配置日志 bean

新建一个 `com.atguigu.springcloud.config` 包，在其中新建一个配置类 `FeignConfig`

```
1  @Configuration
2  public class FeignConfig {
3      @Bean
4      Logger.Level feignLoggerLevel() {
5          return Logger.Level.FULL;
6      }
7  }
```

注：配置日志，还有另外一种方法。如下：

在 `cloud-consumer-feign-order80` 中的 `application.yml` 中添加如下代码：

```
1  # 可以直接在yaml中配置: feign.client.config.default.loggerLevel: FULL
2  # 其中"default"可以换成FeignClient中配置的name属性，也可以直接用default
3  # 对应的是FeignClientProperties类中的config属性。该类为Feign自动配置类引入的配置项类
4  feign:
5    client:
6      config:
7        default:
8          loggerLevel: FULL
```

2. YML 文件里需要开启日志的Feign客户端

在 `cloud-consumer-feign-order80` 中的 `application.yml` 中添加如下代码：

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

3. 测试

1. 开启各个微服务
2. 用浏览器访问：<http://localhost/consumer/payment/get/31>



3. 查看 `cloud-consumer-feign-order80` 的后台日志：

```
framework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@390d31d2
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] <--- HTTP/1.1 200 (234ms)
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] connection: keep-alive
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] content-type: application/json
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] date: Tue, 26 Apr 2022 06:51:57 GMT
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] keep-alive: timeout=60
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] transfer-encoding: chunked
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById]
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] {"code":200,"message":"查询成功, serverPort: 8001"}
DEBUG 11924 --- [p-nio-80-exec-1] c.a.s.service.PaymentFeignService : [PaymentFeignService#getPaymentById] <--- END HTTP (96-byte body)
INFO 11924 --- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty : Flipping property: CLOUD-PAYMENT-SERVICE.ribbon.ActiveConnectionsLimit to use NEX
```

九、Hystrix断路器

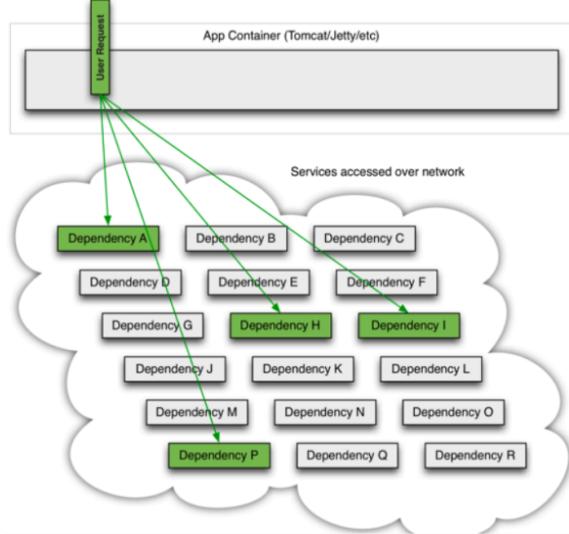
断路器：circuit breaker

Hystrix 简介

分布式系统面临的问题

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。



左图中的请求需要调用A, P, H, I四个服务，如果一切顺利则没有什么问题，关键是如何I服务超时会出现什么情况呢？



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，单一的后端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**。

所以，为了避免发生级联故障，就有了一种兜底的方案，或者一种链路中断方案，即服务降级。

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 能做什么

Hystrix主要做的是：

- 服务降级
- 服务熔断
- 接近实时的监控

-

官网资料

<https://github.com/Netflix/Hystrix>

<https://github.com/Netflix/Hystrix/wiki>

Hystrix停止更新

☞ Hystrix: Latency and Fault Tolerance for Distributed Systems

[oss lifecycle](#) [maintenance](#) [build](#) [error](#) [maven central](#) [1.5.18](#) [License](#) [Apache 2](#)

Hystrix Status

Hystrix is no longer in active development, and is currently in maintenance mode.

Hystrix (at version 1.5.18) is stable enough to meet the needs of Netflix for our existing applications. Meanwhile, our focus has shifted towards more adaptive implementations that react to an application's real time performance rather than pre-configured settings (for example, through [adaptive concurrency limits](#)). For the cases where something like Hystrix makes sense, we intend to continue using Hystrix for existing applications, and to leverage open and active projects like [resilience4j](#) for new internal projects. We are beginning to recommend others do the same.

Netflix Hystrix is now officially in maintenance mode, with the following expectations to the greater community: Netflix will no longer actively review issues, merge pull-requests, and release new versions of Hystrix. We have made the final release of Hystrix (1.5.18) per [issue 1891](#) so that the latest version in Maven Central is aligned with the last known stable version used internally at Netflix (1.5.11). If members of the community are interested in taking ownership of Hystrix and moving it back into active mode, please reach out to hystrixoss@googlegroups.com.

Hystrix has served Netflix and the community well over the years, and the transition to maintenance mode is in no way an indication that the concepts and ideas from Hystrix are no longer valuable. On the contrary, Hystrix has inspired many great ideas and projects. We thank everyone at Netflix, and in the greater community, for all the contributions made to Hystrix over the years.

Hystrix重要概念

服务降级

服务降级 fallback

在某个服务单元发生故障之后，通过断路器的故障监控，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。

比如，可以返回：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

哪些情况下会发生 服务降级？

- 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

服务熔断 break

服务熔断：类比保险丝，达到最大服务访问量后，直接拒绝访问，然后调用服务降级的方法并返回友好提示

服务限流

服务限流 flowlimit

在秒杀、高并发等操作中，服务限流可以严禁请求一窝蜂的过来，让请求排队，一秒钟N个，有序进行。

Hystrix 案例

构建

构建模块

在构建模块之前，将7001模块eureka改成单机版，演示不用集群。

然后，

新建一个 `cloud-provider-hystrix-payment8001` 模块

改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-client</artifactId>
11  </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     <dependency><!-- 引入自己定义的api通用包，可以使用Payment支付Entity -->
22         <groupId>com.atguigu.springcloud</groupId>
23         <artifactId>cloud-api-commons</artifactId>
24         <version>${project.version}</version>
25     </dependency>
26     <dependency>
27         <groupId>org.springframework.boot</groupId>
28         <artifactId>spring-boot-devtools</artifactId>
29         <scope>runtime</scope>
30         <optional>true</optional>
31     </dependency>
32     <dependency>
33         <groupId>org.projectlombok</groupId>
34         <artifactId>lombok</artifactId>
35         <optional>true</optional>
36     </dependency>
37     <dependency>
38         <groupId>org.springframework.boot</groupId>
39         <artifactId>spring-boot-starter-test</artifactId>
40         <scope>test</scope>
41     </dependency>
42 </dependencies>
```

写YML

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

主启动

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class PaymentHystrixMain8001 {
4     public static void main(String[] args) {
5         SpringApplication.run(PaymentHystrixMain8001.class, args);
6     }
7 }
```

业务类 - service

注：正常情况下是要写一个接口和对应的实现类

这里为了节约时间，只创建一个类！

```
1  @Service
2  public class PaymentService {
3
4      /**
5       * 正常访问
6       * @param id
7       * @return
8       */
9      public String paymentInfo_OK(Integer id) {
10         return "线程池：" + Thread.currentThread().getName() + " paymentInfo_OK,
11             id：" + id;
12     }
13
14     /**
15      * 可能出现访问异常
16      * @param id
17      * @return
18      */
19      public String paymentInfo_TimeOut(Integer id) {
20         int timeNumber = 3;
21         try {
22             TimeUnit.SECONDS.sleep(timeNumber);
23         } catch (InterruptedException e) {
24             e.printStackTrace();
25         }
26         return "线程池：" + Thread.currentThread().getName() + "
27             paymentInfo_TimeOut, id：" + id + " 耗时(秒)" + timeNumber;
28     }
29 }
```

业务类 - controller

```
1  @RestController
2  @Slf4j
3  public class PaymentController {
4      @Resource
5      private PaymentService paymentService;
6      @Value("${server.port}")
7      private String serverPort;
8
9      @GetMapping("/payment/hystrix/ok/{id}")
10     public String paymentInfo_OK(@PathVariable("id") Integer id) {
11         String result = paymentService.paymentInfo_OK(id);
12         log.info("****result: " + result);
13         return result;
14     }
15
16     @GetMapping("/payment/hystrix/timeout/{id}")
17     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
18         String result = paymentService.paymentInfo_TimeOut(id);
19         log.info("****result: " + result);
20         return result;
21 }
```

```
21      }
22 }
```

测试

1. 启动eureka7001
2. 启动 **cloud-provider-hystrix-payment8001**
3. 访问 success 方法: <http://localhost:8001/payment/hystrix/ok/31>

很快能返回。可以正常访问



← → ⌛ ⌂ ⓘ localhost:8001/payment/hystrix/ok/31

线程池: http-nio-8001-exec-1 paymentInfo_OK, id: 31

4. 访问 timeout 方法: <http://localhost:8001/payment/hystrix/timeout/31>

耗时约3秒后返回。可以正常访问



← → ⌛ ⌂ ⓘ localhost:8001/payment/hystrix/timeout/31

线程池: http-nio-8001-exec-7 paymentInfo_TimeOut, id: 31 耗时(秒)3

上述module均正常，以上述为根基，后续我们从正常 -> 错误 -> 降级熔断 -> 恢复 进行演示。

高并发测试

上一小节能正常访问是因为在非高并发的情况下，但是如果在高并发条件下，可能就会出现问题。下面进行演示。

JMeter压测测试

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

jmeter创建一个线程组：



jmeter在线程组中创建一个HTTP请求，访问timeout方法：



测试结果：

在经历高并发测试之后，立马使用浏览器访问 ok 方法和 timeout 方法，发现在一段时间内都**存在访问缓慢现象**！

分析演示结果

在上述高并发测试中，存在访问缓慢的现象！

为什么会这样？

tomcat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

上面测试还是服务提供者8001自己测试，假设此时外部的消费者80也来访问，那消费者只能干等，最终导致消费者端80不满意，服务端8001直接被拖死

下一小节我们演示消费者端80的加入，在高并发测试下会出现什么问题！

消费端80加入高并发测试

1. 新建module

新建一个 `cloud-consumer-feign-hystrix-order80`

2. 改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>com.atguigu.springcloud</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. 写YML

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

4. 主启动

```
1 @SpringBootApplication
2 @EnableFeignClients
3 public class OrderHystrixMain80 {
4     public static void main(String[] args) {
5         SpringApplication.run(OrderHystrixMain80.class, args);
6     }
7 }
```

5. 业务类 - service

```
1 @Component
2 @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")
3 public interface PaymentHystrixService {
4
5     @GetMapping("/payment/hystrix/ok/{id}")
6     String paymentInfo_OK(@PathVariable("id") Integer id);
7
8     @GetMapping("/payment/hystrix/timeout/{id}")
9     String paymentInfo_TimeOut(@PathVariable("id") Integer id);
10 }
```

6. 业务类 - controller

```
1 @RestController
2 @Slf4j
3 public class OrderHystrixController {
4
5     @Resource
6     private PaymentHystrixService paymentHystrixService;
7
8     @GetMapping("/consumer/payment/hystrix/ok/{id}")
9     String paymentInfo_OK(@PathVariable("id") Integer id) {
10         return paymentHystrixService.paymentInfo_OK(id);
11     }
12 }
```

```
12
13     @GetMapping("/consumer/payment/hystrix/timeout/{id}")
14     String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
15         return paymentHystrixService.paymentInfo_TimeOut(id);
16     }
17 }
```

7. 正常测试

- 在浏览器上访问：<http://localhost/consumer/payment/hystrix/ok/31>

可以正常访问

8. 高并发测试

- 再次用jmeter制造2w个线程压测8001
- 立即在浏览器上访问：<http://localhost/consumer/payment/hystrix/ok/31>

可能出现两种现象：

- 要么访问卡顿
- 要么消费端报超时错误（因为消费端目前没有利用OpenFeign做超时控制）

故障现象和导致原因

- 8001同一层次的其他接口服务被困死，因为tomcat线程池里面的工作线程已经被挤占完毕
- 80此时调用8001，客户端访问响应缓慢，浏览器一致转圈圈

正是因为有了上述故障或不佳表现，才有了我们的降级/容错/限流等技术诞生

如何解决？解决的要求

- 超时导致服务器变慢 ---> 解决：超时不再等待
- 出错（宕机或程序运行出错） ---> 解决：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

降级配置：@HystrixCommand 注解

先解决8001自身的问题：可以设置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

在 `cloud-provider-hystrix-payment8001` 模块中修改service类。主要加了如下代码：

```
public class PaymentService {
    /** 正常访问 ... */
    public String paymentInfo_OK(Integer id) {...}
    /** 可能出现访问异常 ... */
    @HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")
    })
    public String paymentInfo_TimeOut(Integer id) {
        int timeNumber = 5;
        // 故意写的一行，为了让该方法执行时抛出异常
        // int age = 10 / 0;
        try {
            TimeUnit.SECONDS.sleep(timeNumber);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "线程池：" + Thread.currentThread().getName() + " paymentInfo_TimeOut, id:" + id + " 耗时(秒)" + timeNumber;
    }
    public String paymentInfo_TimeOutHandler(Integer id) {
        return "线程池：" + Thread.currentThread().getName() + " paymentInfo_TimeOutHandler, id:" + id + " 服务降级处理方法";
    }
}
```

主启动类激活

由于上述添加了一个新注解 `@HystrixCommand`，因此我们需要在主启动类上激活

```
1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableCircuitBreaker
4  public class PaymentHystrixMain8001 {
5      public static void main(String[] args) {
6          SpringApplication.run(PaymentHystrixMain8001.class, args);
7      }
8  }
```

上面代码中，添加了新注解 `@EnableCircuitBreaker`

当然，上述注解目前已过时了！请用注解 `@EnableHystrix`

```

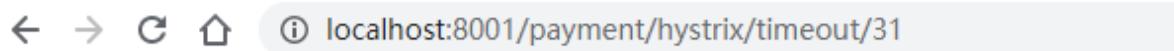
1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableHystrix
4  public class PaymentHystrixMain8001 {
5      public static void main(String[] args) {
6          SpringApplication.run(PaymentHystrixMain8001.class, args);
7      }
8  }

```

测试1

访问：<http://localhost:8001/payment/hystrix/timeout/31>

页面显示结果：



线程池：HystrixTimer-1 paymentInfo_TimeOutHandler, id: 31 服务降级处理方法

可以发现，进行了服务降级！调用了 `paymentInfo_TimeOutHandler` 方法

并且，可以发现，这里的 `HystrixTimer`！说明服务降级时使用了一个单独的线程池，起到一个隔离的效果。

测试2

- 在 `PaymentService` 类的 `paymentInfo_TimeOut` 方法中故意制造异常

```

@Service
public class PaymentService {
    /** 正常访问 ...*/
    public String paymentInfo_OK(Integer id) {...}

    /** 可能出现访问异常 ...*/
    @HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")
    })
    public String paymentInfo_TimeOut(Integer id) {
        int timeNumber = 5;
        // 故意写的一行，为了让该方法执行时抛出异常
        int age = 10 / 0;
        // try {
        //     TimeUnit.SECONDS.sleep(timeNumber); 取消睡眠
        // } catch (InterruptedException e) {
        //     e.printStackTrace();
        // }
        return "线程池：" + Thread.currentThread().getName() + " paymentInfo_TimeOut, id:" + id + " 耗时(秒)" + timeNumber;
    }

    public String paymentInfo_TimeOutHandler(Integer id) {

```

- 访问：<http://localhost:8001/payment/hystrix/timeout/31>



线程池：hystrix-PaymentService-1 paymentInfo_TimeOutHandler, id: 31 服务降级处理方法

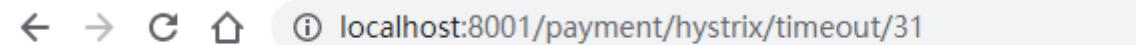
可以看到，如果是方法抛出异常，也是会进行服务降级处理的

测试3

- 在 `PaymentService` 类的 `paymentInfo_TimeOut` 方法中设置超时时间5s，线程睡眠时间3s。
(这说明执行该方法时不会超时)

```
@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {  
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")  
})  
public String paymentInfo_TimeOut(Integer id) {  
    int timeNumber = 3;  
    // 故意写的一行，为了让该方法执行时抛出异常  
    // int age = 10 / 0;  
    try {  
        TimeUnit.SECONDS.sleep(timeNumber);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    return "线程池: " + Thread.currentThread().getName() + " paymentInfo_TimeOut, id: " + id + " 耗时(秒)" + timeNumber;  
}  
public String paymentInfo_TimeOutHandler(Integer id) {  
    return "线程池: " + Thread.currentThread().getName() + " paymentInfo_TimeOutHandler, id: " + id + " 服务降级处理方法";  
}
```

- 访问：<http://localhost:8001/payment/hystrix/timeout/31>



线程池: hystrix-PaymentService-2 paymentInfo_TimeOut, id: 31 耗时(秒)3

可以看到，能够正常访问该方法，没有进行服务降级！

结论

只要是当前服务不可用了，无论是计算异常还是超时异常，都会做服务降级。

80fallback

80订单微服务，当它去调用8001的微服务时，也可以更好的保护自己，自己也可以进行客户端降级保护

Hystrix的服务降级，既可以放在服务端，也可以放在消费端！

对 `cloud-consumer-feign-hystrix-order80` 进行修改

改YAML

注意：消费端要想使用hystrix，需要在 `application.yml` 中开启！因为消费端使用了feign！

```
1 feign:  
2     hystrix:  
3         enabled: true
```

主启动

在主启动类上添加 `@EnableHystrix` 注解

```
1  @SpringBootApplication
2  @EnableFeignClients
3  @EnableHystrix
4  public class OrderHystrixMain80 {
5      public static void main(String[] args) {
6          SpringApplication.run(OrderHystrixMain80.class, args);
7      }
8  }
```

业务类

对controller类进行修改

```
1  @RestController
2  @Slf4j
3  public class OrderHystrixController {
4
5      @Resource
6      private PaymentHystrixService paymentHystrixService;
7
8      @GetMapping("/consumer/payment/hystrix/ok/{id}")
9      String paymentInfo_OK(@PathVariable("id") Integer id) {
10         return paymentHystrixService.paymentInfo_OK(id);
11     }
12
13     /**
14      * 可能出现访问异常
15      * @param id
16      * @return
17      */
18     @HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod",
19     commandProperties = {
20         @HystrixProperty(name =
21             "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
22     })
23     @GetMapping("/consumer/payment/hystrix/timeout/{id}")
24     public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
25         return paymentHystrixService.paymentInfo_TimeOut(id);
26     }
27     public String paymentTimeOutFallbackMethod(Integer id) {
28         return "我是消费者80，对方支付系统繁忙，请10s后再试或者自己运行出错请检查自己！";
29     }
30 }
```

主要修改了如下代码：

```
    /**
     * 可能出现访问异常 ...
     */
    @HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod", commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
    })
    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
        return paymentHystrixService.paymentInfo_TimeOut(id);
    }
    public String paymentTimeOutFallbackMethod(Integer id) {
        return "我是消费者80，对方支付系统繁忙，请10s后再试或者自己运行出错请检查自己!";
    }
}
```

测试1

注：

上述代码还是存在问题：消费端controller中超时时间配置其实是不生效的！

原因：

关键在于 `feign:hystrix:enabled: true` 的作用，官网解释“Feign将使用断路器包装所有方法”，也就是将 `@FeignClient` 标记的那个service接口下所有的方法进行了hystrix包装（类似于在这些方法上加了一个`@HystrixCommand`），这些方法会应用一个默认的超时时间为1s，所以你的service方法也有一个1s的超时时间，service1s就会报异常，controller立马进入备用方法，controller上那个3秒那超时时间就没有效果了。

解决方法：

在测试前需要在 `application.yml` 文件中修改默认超时时间

```
1 hystrix:
2   command:
3     default:
4       execution:
5         isolation:
6           thread:
7             timeoutInMilliseconds: 4000
8   # ribbon的超时时间也需要加上
9   ribbon:
10  ReadTimeout: 5000
11  ConnectTimeout: 5000
```

注：

配置文件这里的 `timeoutInMilliseconds` 并不是覆盖注解中的设置，而是两者取较低值，同时也会算上 `ribbon: ReadTimeout` 的值，也就是**三者取最低值**。

三者，即：注解中的设置、配置文件中的`timeoutMilliseonds` 以及 `ribbon.ReadTimeout`。

因此，上述配置后，消费端的超时时间为4s。

1. 假设在支付端（即，服务提供者）调用一个方法，需要执行3s，且若执行超过5s有服务降级兜底

```

@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000")
})
public String paymentInfo_TimeOut(Integer id) {
    int timeNumber = 3;
    // 故意写的一行，为了让该方法执行时抛出异常
    // int age = 10 / 0;
    try {
        TimeUnit.SECONDS.sleep(timeNumber);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "线程池: " + Thread.currentThread().getName() + " paymentInfo_TimeOut, id: " + id + " 耗时(秒)" + timeNumber;
}
public String paymentInfo_TimeOutHandler(Integer id) {
    return "线程池: " + Thread.currentThread().getName() + " paymentInfo_TimeOutHandler, id: " + id + " 服务降级处理方法";
}

```

2. 消费端，调用自己的一个方法，如果执行该方法超过1.5s，则有服务降级兜底。消费端的这个方法需要调用上述支付端的方法

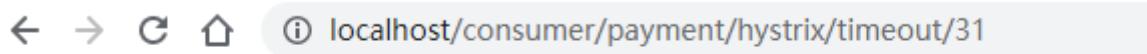
```

/** 可能出现访问异常 ...*/
@HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
})
@GetMapping("/consumer/payment/hystrix/timeout/{id}")
public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
    return paymentHystrixService.paymentInfo_TimeOut(id);
}
public String paymentTimeOutFallbackMethod(Integer id) {
    return "我是消费者80，对方支付系统繁忙，请10s后再试或者自己运行出错请检查自己!";
}

```

3. 启动微服务，访问地址：<http://localhost/consumer/payment/hystrix/timeout/31>

显示如下。说明消费端调用了自己的服务降级方法！



测试2

1. 修改消费端的方法，让方法内抛出计算异常

```

/** 可能出现访问异常 ...*/
@HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod", commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
})
@GetMapping("/consumer/payment/hystrix/timeout/{id}")
public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
    int age = 10 / 0;
    return paymentHystrixService.paymentInfo_TimeOut(id);
}
public String paymentTimeOutFallbackMethod(Integer id) {
    return "我是消费者80，对方支付系统繁忙，请10s后再试或者自己运行出错请检查自己!";
}

```

2. 然后其他配置和测试1中一样
3. 运行微服务，访问地址：<http://localhost/consumer/payment/hystrix/timeout/31>

我是消费者80, 对方支付系统繁忙, 请10s后再试或者自己运行出错请检查自己!

依旧会使用消费端的服务降级方法进行兜底!

说明, 无论计算异常还是超时异常, 都会使用服务降级方法进行兜底!

全局服务降级

在上述的例子中, 我们可以发现, 每个业务方法都需要一个兜底 (服务降级) 方法 (容易导致代码膨胀), 并且兜底方法和业务方法混合在一块 (代码混乱)。

因此, 我们需要解决上述问题!

这里我们利用全局服务降级 (`@DefaultProperties`) 来解决代码膨胀问题!

`@DefaultProperties(defaultFallback = "")` 的作用:

```
① @DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
public class PaymentControllerFeign
{
    @Autowired
    private PaymentService paymentService;

    @GetMapping("/consumer/payment/{id}")
    public String paymentInfo(@PathVariable("id") Integer id)
    {
        return paymentService.getPaymentInfo(id);
    }

    @GetMapping("/consumer/paymentTimeOut/{id}")
    @HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod")
    public String paymentTimeOut(@PathVariable("id") Integer id)
    {
        return paymentService.paymentTimeOut(id);
        //throw new RuntimeException("*****Exception 2001 ");
    }
    public String paymentTimeOutFallbackMethod(@PathVariable("id") Integer id)
    {
        return "对方系统繁忙或者已经down机, 请10秒钟后再次尝试";
    }

    ② public String payment_Global_FallbackMethod()
    {
        return "Global 222|对方系统繁忙或者已经down机, 请10秒钟后再次尝试,/(ㄒoㄒ)/~~";
    }
}
```

没有特别指明就用统一的

注意: 全局服务降级配置的统一方法 `defaultFallback` 不需要有参数!

配置全局服务降级

1. 在消费端的controller写上如下代码:

```
1  @RestController
2  @Slf4j
3  @DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
4  public class OrderHystrixController {
5
6      @Resource
```

```

7     private PaymentHystrixService paymentHystrixService;
8
9     @GetMapping("/consumer/payment/hystrix/ok/{id}")
10    String paymentInfo_OK(@PathVariable("id") Integer id) {
11        return paymentHystrixService.paymentInfo_OK(id);
12    }
13
14    /**
15     * 可能出现访问异常
16     * @param id
17     * @return
18     */
19    // @HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod",
20    commandProperties = {
21        //          @HystrixProperty(name =
22        "execution.isolation.thread.timeoutInMilliseconds", value = "1500")
23        // })
24    @HystrixCommand // 类上加了@DefaultProperties属性注解，并且方法上的@HystrixCommand
25    没有写具体方法名字，就用统一全局的
26    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
27    public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
28        int age = 10 / 0;
29        return paymentHystrixService.paymentInfo_TimeOut(id);
30    }
31    // 下面是全局fallback方法
32    public String payment_Global_FallbackMethod() {
33        return "Global异常处理信息，请稍后再试！";
34    }
35
36 }

```

主要加了如下代码：

```

@DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
public class OrderHystrixController {

    @Resource
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/consumer/payment/hystrix/ok/{id}")
    String paymentInfo_OK(@PathVariable("id") Integer id) { return paymentHystrixService.paymentInfo_OK(id); }

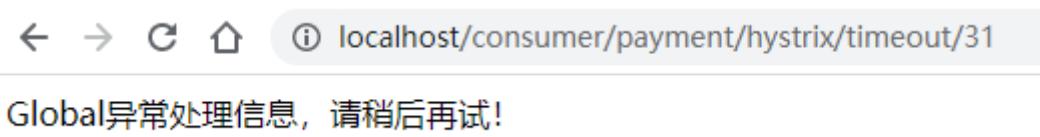
    /**
     * 可能出现访问异常 ...
     */
    @HystrixCommand // 类上加了@DefaultProperties属性注解，并且方法上的@HystrixCommand没有写具体方法名字，就用统一全局的
    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id) {
        int age = 10 / 0;
        return paymentHystrixService.paymentInfo_TimeOut(id);
    }

    // 下面是全局fallback方法
    public String payment_Global_FallbackMethod() {
        return "Global异常处理信息，请稍后再试！";
    }
}

```

测试

1. 访问: <http://localhost/consumer/payment/hystrix/timeout/31>



通配服务降级

目前，我们的服务降级方法和业务逻辑放在同一个类下，会造成代码混乱等情况。这一小结我们就要解决这种情况。

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：

- 运行时可能抛出异常
- 超时
- 宕机可能出现的异常（比如，服务端如果突然宕机）

目前我们的客户端大部分是以下类型的代码：

```
    ...
    ...
    ...
}

@GetMapping("/payment/hystrix/{id}")
@HystrixCommand(fallbackMethod = "paymentInfoHandlerException")
public String paymentInfo(@PathVariable("id") Integer id)
{
    if(id >= 0)
    {
        return "调用支付接口hystrix服务: \t"+serverPort+"\t entity ID: "+id;
    }else{
        throw new RuntimeException("ID不能是负数");
    }
}
public String paymentInfoHandlerException(Integer id)
{
    return "调用支付接口hystrix服务出现异常: \t"+"ID不能是负数";
}
```

可以看到服务降级方法和业务方法容易混合在一起。

下面我们就来改进这种方法。（只要修改 `cloud-consumer-feign-hystrix-order80` 即可）

新建一个service类实现PaymentHystrixService接口

在 `cloud-consumer-feign-hystrix-order80` 中，我们目前已经有一个 `PaymentHystrixService` 接口，如下：

```
1 @Component
2 @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")
3 public interface PaymentHystrixService {
4
5     @GetMapping("/payment/hystrix/ok/{id}")
6     String paymentInfo_OK(@PathVariable("id") Integer id);
7
8     @GetMapping("/payment/hystrix/timeout/{id}")
9     String paymentInfo_TimeOut(@PathVariable("id") Integer id);
10
11 }
```

现在，我们新建一个类 `PaymentFallbackService` 实现该接口，统一为接口里的方法进行异常处理

```
1 @Component
2 public class PaymentFallbackService implements PaymentHystrixService{
3     @Override
4     public String paymentInfo_OK(Integer id) {
5         return "-----PaymentFallbackService fall back--paymentInfo_OK";
6     }
7
8     @Override
9     public String paymentInfo_TimeOut(Integer id) {
10        return "-----PaymentFallbackService fall back--paymentInfo_TimeOut";
11    }
12 }
```

YAML

保证YAML中开启了Hystrix

```
1 feign:
2     hystrix:
3         enabled: true
```

修改PaymentHystrixService接口

```
1  @Component
2  @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT", fallback =
3      PaymentFallbackService.class)
4
5  public interface PaymentHystrixService {
6
7
8      @GetMapping("/payment/hystrix/ok/{id}")
9      String paymentInfo_OK(@PathVariable("id") Integer id);
10
11 }
```

主要修改处：

```
1  @Component
2  @FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT", [fallback = PaymentFallbackService.class])
3  public interface PaymentHystrixService {
4
5      @GetMapping("/payment/hystrix/ok/{id}")
6      String paymentInfo_OK(@PathVariable("id") Integer id);
7
8      @GetMapping("/payment/hystrix/timeout/{id}")
9      String paymentInfo_TimeOut(@PathVariable("id") Integer id);
10
11 }
```

注：

`@FeignClient` 注解中的 `fallback` 属性，指定客户端接口的fallback类

测试

在修改后的 `cloud-consumer-feign-hystrix-order80` 中，运行逻辑是这样的：

1. http请求会访问消费端的controller类中的方法
2. controller类中的方法会调用 `PaymentHystrixService` 接口中的方法
3. `PaymentHystrixService` 接口中的方法会调用远程服务端的方法
4. 如果 `PaymentHystrixService` 接口调用方法出现异常，则会调用 `PaymentFallbackService` 类中相对应的方法作为服务降级方法，进行兜底；如果未出现异常，则正常执行

下面我们进行测试：

1. 启动单个eureka服务器7001
2. `PaymentHystrixMain8001`启动
3. `OrderHystrixMain80`启动
4. 正常测试：访问地址 <http://localhost/consumer/payment/hystrix/ok/31>

线程池: http-nio-8001-exec-1 paymentInfo_OK, id: 31

5. 故意关闭微服务8001，模拟服务端突然宕机的状态
6. 访问地址 <http://localhost/consumer/payment/hystrix/ok/31>

-----PaymentFallbackService fall back--paymentInfo_OK

可以看到，这里访问的是 `PaymentFallbackService` 类中的 `paymentInfo_OK` 方法。它是通配服务降级方法。

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。

服务熔断

什么是熔断

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。

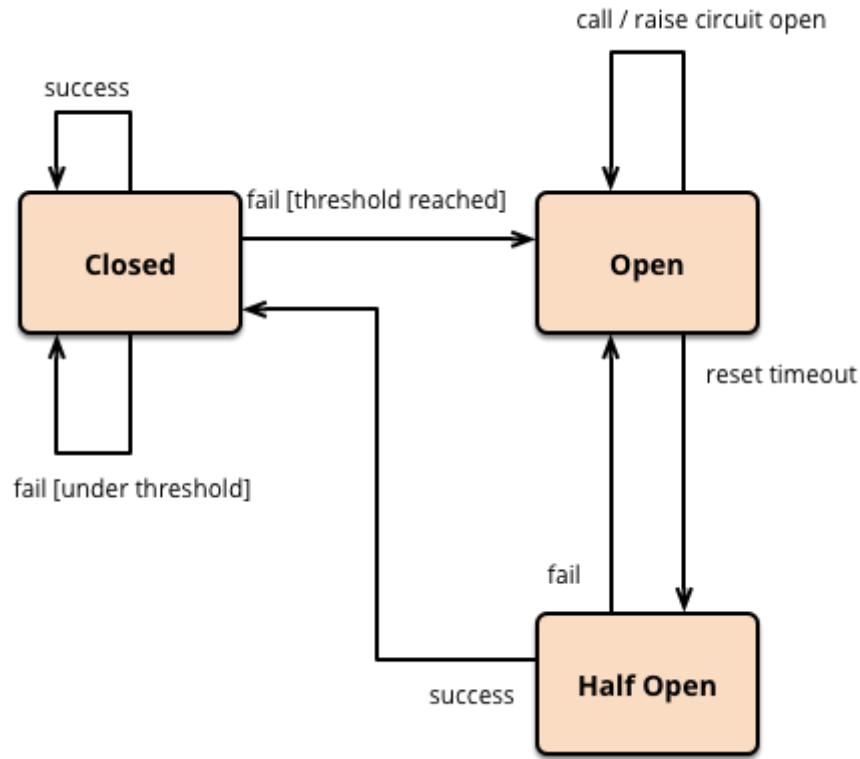
当检测到该节点微服务调用响应正常后，恢复调用链路。

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`。

服务熔断和服务降级出现的顺序：

1. 调用失败会触发降级，而降级会调用fallback方法
2. 但无论如何，降级的流程一定会先调用正常方法，再调用fallback方法
3. 假如单位时间内调用失败次数过多，也就是降级次数过多，则触发熔断
4. 熔断以后，就会跳过正常方法，直接调用fallback方法
5. 所谓“熔断后服务不可用”就是因为跳过了正常方法执行fallback

论文：<https://martinfowler.com/bliki/CircuitBreaker.html>



实操

修改cloud-provider-hystrix-payment8001

修改 `cloud-provider-hystrix-payment8001` 服务提供者模块

1. 修改 PaymentService

```

1  @Service
2  public class PaymentService {
3      /**
4      * 正常访问
5      * @param id
6      * @return
7      */
8      public String paymentInfo_OK(Integer id) {
9          return "线程池: " + Thread.currentThread().getName() + " paymentInfo_OK,
10         id: " + id;
11     }
12     /**
13      * 可能出现访问异常
14      * @param id
15      * @return
16      */
17     @HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler",
18     commandProperties = {
19         @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "5000")
20     })
21     public String paymentInfo_TimeOut(Integer id) {
22         int timeNumber = 3;

```

```

21         // 故意写的一行，为了让该方法执行时抛出异常
22         // int age = 10 / 0;
23         try {
24             TimeUnit.SECONDS.sleep(timeNumber);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28         return "线程池: " + Thread.currentThread().getName() + "
29 paymentInfo_TimeOut, id: " + id + " 耗时(秒)" + timeNumber;
30     }
31     public String paymentInfo_TimeOutHandler(Integer id) {
32         return "线程池: " + Thread.currentThread().getName() + "
33 paymentInfo_TimeOutHandler, id: " + id + " 服务降级处理方法";
34     }
35
36     // ===== 服务熔断 =====
37     @HystrixCommand(fallbackMethod =
38         "paymentCircuitBreaker_fallback", commandProperties = {
39             @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),
40             // 是否开启断路器
41             @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value
42             = "10"), // 请求次数
43             @HystrixProperty(name =
44             "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), // 时间窗口期，即断
45             // 路器开启的时间，在这个时间内不允许访问任何微服务
46             @HystrixProperty(name =
47             "circuitBreaker.errorThresholdPercentage", value = "60"), // 失败率达到多少后熔断。
48             // (10次请求有60%的请求，即6次失败，那就熔断)
49         })
50         public String paymentCircuitBreaker(@PathVariable("id") Integer id)
51     {
52         if(id < 0)
53         {
54             throw new RuntimeException("*****id 不能负数");
55         }
56         String serialNumber = IdUtil.simpleUUID(); // 这里类似
57         UUID.randomUUID().toString(), 不用细究
58
59         return Thread.currentThread().getName()+"\t"+调用成功，流水号：" +
60         serialNumber;
61     }
62     public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id)
63     {
64         return "id 不能负数，请稍后再试, /(ㄒoㄒ)/~~ id: " +id;
65     }
66 }

```

主要是添加了如下内容：

```

// ===== 服务熔断 =====
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
    @HystrixProperty(name = "circuitBreaker.enabled", value = "true"), // 是否开启断路器
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"), // 请求次数
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), // 时间窗口期
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60") // 失败率达到多少后熔断. (10次请求有60%的请求, 即6次失败, 那就熔断)
})
public String paymentCircuitBreaker(@PathVariable("id") Integer id)
{
    if(id < 0)
    {
        throw new RuntimeException("*****id 不能负数");
    }
    String serialNumber = IdUtil.simpleUUID(); // 这里类似 UUID.randomUUID().toString(), 不用细究

    return Thread.currentThread().getName()+"\t"+serialNumber;
}
public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id)
{
    return "id 不能负数, 请稍后再试, /(\u2019)\u2019 id: " + id;
}

```

上述主要配置的参数：

- `circuitBreaker.enabled`：是否开启断路器
- `circuitBreaker.requestVolumeThreshold`：指定请求次数
- `circuitBreaker.sleepWindowInMilliseconds`：指定休眠窗口, 即断路器开启的时间。在这个时间内不允许访问任何微服务。单位毫秒
- `circuitBreaker.errorThresholdPercentage`：指定失败率阈值。即，失败率达到多少后熔断。(在本例中10次请求有60%的请求失败, 即6次失败, 那就熔断)

官网上的流程和步骤：

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 exceeds the threshold error percentage
`(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 single request is let through (this is the `HALF-OPEN` state). If the request fails, the circuit-breaker returns to the `OPEN` state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to `CLOSED` and the logic in 1. takes over again.

2. 修改controller

由于我们在service中新增了方法，所以我们需要在controller中也增加一个方法，来调用service的方法。

主要是新增了下面这个方法：

```
    ...
    @RestController
    @Slf4j
    public class PaymentController {
        @Resource
        private PaymentService paymentService;
        @Value("${server.port}")
        private String serverPort;

        // ===== 服务熔断 =====
        @GetMapping("/payment/circuit/{id}")
        public String paymentCircuitBreaker(@PathVariable("id") Integer id) {
            String result = paymentService.paymentCircuitBreaker(id);
            log.info("****result: " + result);
            return result;
        }
    }
```

测试

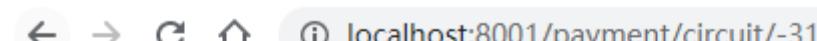
1. 启动eureka7001单机版
2. 启动 `cloud-provider-hystrix-payment8001` 模块
3. 访问: <http://localhost:8001/payment/circuit/31>

正常访问，没有异常



4. 访问: <http://localhost:8001/payment/circuit/-31>

由于传入的id是负数，所以会调用了服务降级方法进行兜底



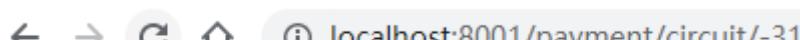
id 不能负数, 请稍后再试, /(ㄒoㄒ)/~~ id: -31

以上只是测试了服务降级，还没有测试服务熔断。

下面进行测试服务熔断！

1. 访问: <http://localhost:8001/payment/circuit/-31>

这里需要多次且快速访问！最好在10秒内访问6次以上



id 不能负数, 请稍后再试, /(ㄒoㄒ)/~~ id: -31

2. 然后立即访问: <http://localhost:8001/payment/circuit/31>

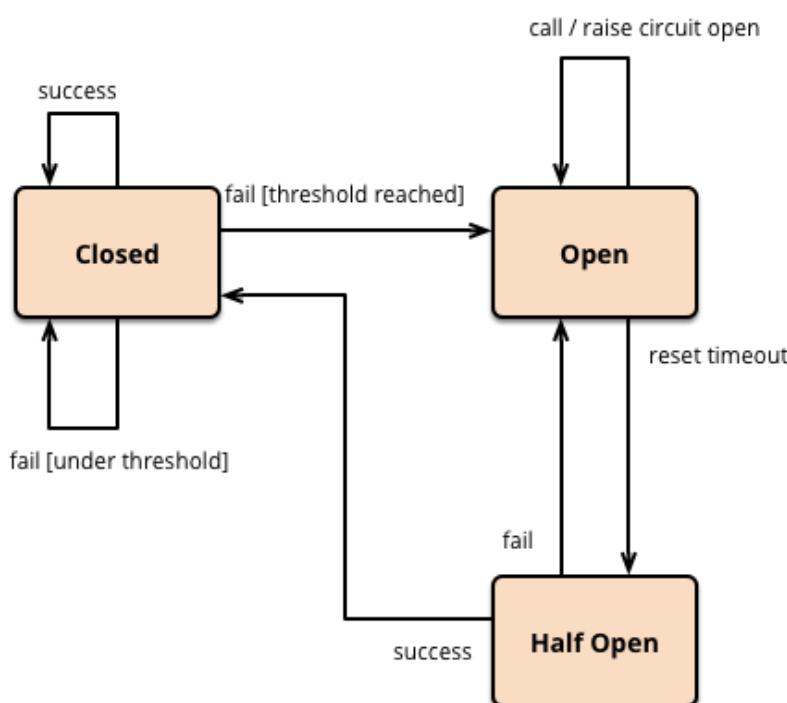
可以发现，明明id是正数，但是还是调用服务降级方法了！而没有调用正常的方法！这是因为步骤1的访问失败率大于60%，因此进行了服务熔断，导致在窗口期内只能访问服务降级方法，不能访问正常方法。

localhost:8001/payment/circuit/31

id 不能负数，请稍后再试，/(ㄒoㄒ)/~~ id: 31

小总结

This simple circuit breaker avoids making the protected call when the circuit is open, but would need an external intervention to reset it when things are well again. This is a reasonable approach with electrical circuit breakers in buildings, but for software circuit breakers we can have the breaker itself detect if the underlying calls are working again. We can implement this self-resetting behavior by trying the protected call again after a suitable interval, and resetting the breaker should it succeed.



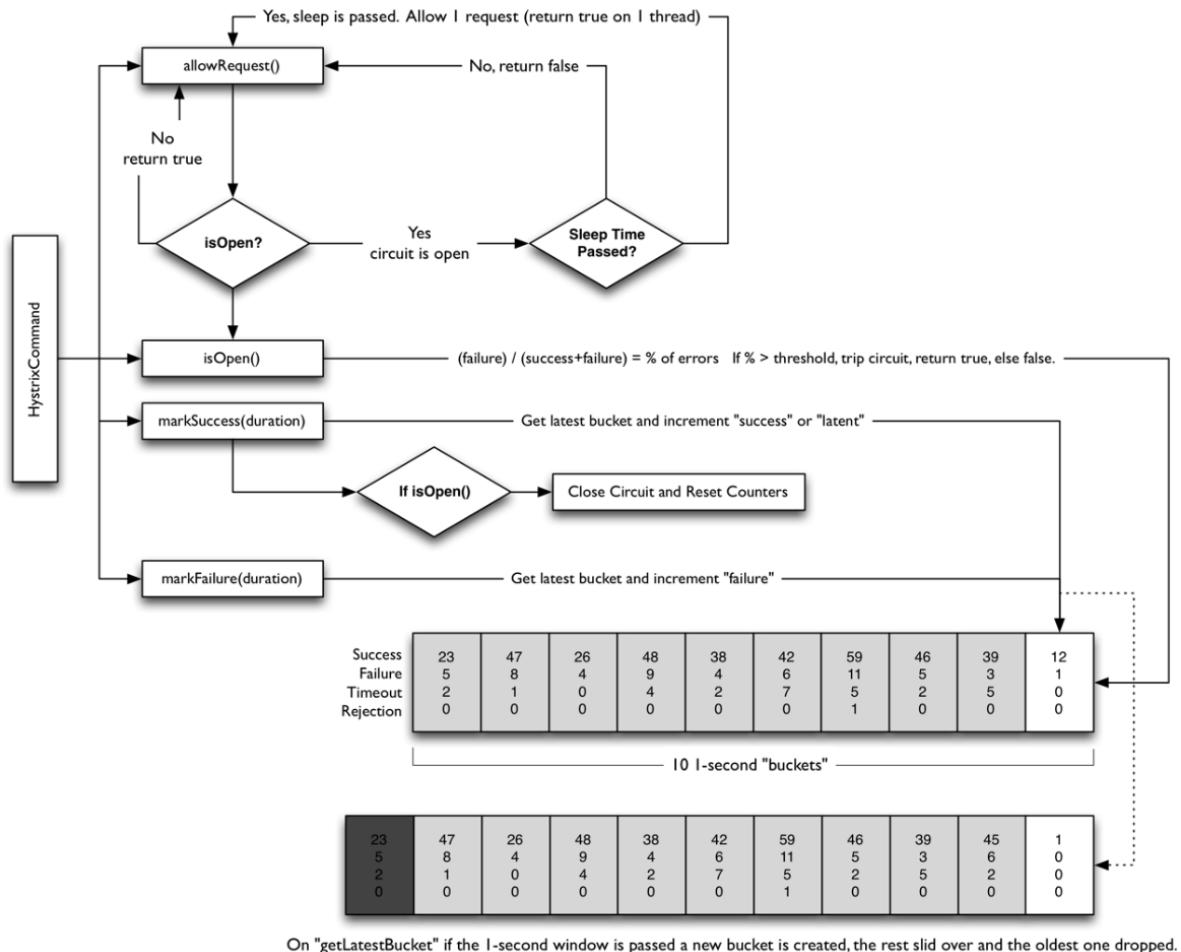
上述显示出来 熔断类型！

熔断类型有三种：

- **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，继而关闭熔断。

官网断路器流程图

官网：<https://github.com/Netflix/Hystrix/wiki/How-it-Works>



图中显示了 HystrixCommand 或 HystrixObservableCommand 如何与 HystrixCircuitBreaker 交互及其逻辑和决策流程，包括计数器在断路器中的行为方式。

断路器在什么情况下开始起作用

参数具体内容参考官网：<https://github.com/Netflix/Hystrix/wiki/Configuration#CommandCircuitBreaker>

```
// ===== 服务熔断 =====
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
    @HystrixProperty(name = "circuitBreaker.enabled", value = "true"), // 是否开启断路器
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"), // 请求次数
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), // 时间窗口期, 即断路器打开后等待时间
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60") // 失败率达到多少后熔断.
})
public String paymentCircuitBreaker(@PathVariable("id") Integer id)
```

涉及到断路器的三个重要参数：休眠时间窗口、请求总数阈值、错误百分比阈值

- 休眠时间窗口 (sleepWindowInMilliseconds)**：断路器跳闸后，在此值的时间内，hystrix会拒绝新的请求。休眠窗口结束之后，会将断路器设置为“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“打开”状态，如果成功就设置为“关闭”状态。。默认为5秒。
- 请求总数阈值 (requestVolumeThreshold)**：在滚动时间窗口（默认10秒）内，必须满足请求数量阈值才有资格熔断。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，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。
- 错误百分比阈值 (errorThresholdPercentage)**：当请求数量在滚动时间窗口内超过了阈值，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。

注：

上述说的 滚动时间窗口 所代表的参数是： `metrics.rollingStats.timeInMilliseconds`

此属性设置统计滚动窗口的持续时间，以毫秒为单位。这是 Hystrix 为断路器使用和发布的指标保留多长时间。默认1000ms

断路器开启或关闭的条件

1. 当满足一定的阈值的时候（默认10秒内超过20个请求次数）
2. 当失败率达到一定的时候（默认10秒内超过50%的请求失败）
3. 达到以上阈值，那么断路器将会开启
4. 当开启的时候，所有的请求都不会进行转发。
5. 在休眠时间窗口这段时间之后（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，然后重复4和5

断路器打开之后

当断路器开启之后：

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。

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。

所有的配置

Hystrix中大部分的配置如下：

```
1 //=====
2 @HystrixCommand(fallbackMethod = "strFallbackMethod",
3     groupKey = "strGroupCommand",
4     commandKey = "strCommand",
5     threadPoolKey = "strThreadPool",
6
7     commandProperties = {
8         // 设置隔离策略, THREAD 表示线程池 SEMAPHORE: 信号池隔离
9         @HystrixProperty(name = "execution.isolation.strategy", 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 (最大并发数)
11        @HystrixProperty(name =
"execution.isolation.semaphore.maxConcurrentRequests", value = "10"),
```

```
12         // 配置命令执行的超时时间
13         @HystrixProperty(name =
14             "execution.isolation.thread.timeoutinMilliseconds", value = "10"),
15             // 是否启用超时时间
16             @HystrixProperty(name = "execution.timeout.enabled", value =
17                 "true"),
18                 // 执行超时的时候是否中断
19                 @HystrixProperty(name =
20                     "execution.isolation.thread.interruptOnTimeout", value = "true"),
21                     // 执行被取消的时候是否中断
22                     @HystrixProperty(name =
23                         "execution.isolation.thread.interruptOnCancel", value = "true"),
24                         // 允许回调方法执行的最大并发数
25                         @HystrixProperty(name =
26                             "fallback.isolation.semaphore.maxConcurrentRequests", value = "10"),
27                             // 服务降级是否启用, 是否执行回调函数
28                             @HystrixProperty(name = "fallback.enabled", value = "true"),
29                             // 是否启用断路器
30                             @HystrixProperty(name = "circuitBreaker.enabled", value =
31                                 "true"),
32                                 // 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, 默认该值为 20
33         的时候,
34             // 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, 断
35         路器也不会打开。
36             @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",
37                 value = "20"),
38                 // 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
39                 // circuitBreaker.requestVolumeThreshold 的情况下, 如果错误请求数的百
40         分比超过50,
41             // 就把断路器设置为 "打开" 状态, 否则就设置为 "关闭" 状态。
42             @HystrixProperty(name =
43                 "circuitBreaker.errorThresholdPercentage", value = "50"),
44                 // 该属性用来设置当断路器打开之后的休眠时间窗。休眠时间窗结束之后,
45                 // 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
46         设置为 "打开" 状态,
47             // 如果成功就设置为 "关闭" 状态。
48             @HystrixProperty(name =
49                 "circuitBreaker.sleepWindowinMilliseconds", value = "5000"),
50                 // 断路器强制打开
51                 @HystrixProperty(name = "circuitBreaker.forceOpen", value =
52                     "false"),
53                     // 断路器强制关闭
54                     @HystrixProperty(name = "circuitBreaker.forceClosed", value =
55                         "false"),
56                         // 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间
57                         @HystrixProperty(name =
58                             "metrics.rollingStats.timeinMilliseconds", value = "10000"),
59                             // 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, 断路器在收集指标信息
60         的时候会根据
61             // 设置的时间窗长度拆分成多个 "桶" 来累计各度量值, 每个"桶"记录了一段时间内
62         的采集指标。
63             // 比如 10 秒内拆分成 10 个"桶"收集这样, 所以 timeinMilliseconds 必须能
64         被 numBuckets 整除。否则会抛异常
65             @HystrixProperty(name = "metrics.rollingStats.numBuckets", value
66                 = "10"),
67                     // 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。如果设置为
68                     false, 那么所有的概要统计都将返回 -1。
```

```

48             @HystrixProperty(name = "metrics.rollingPercentile.enabled",
49             value = "false"),
50                 // 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。
51             @HystrixProperty(name =
52             "metrics.rollingPercentile.timeInMilliseconds", value = "60000"),
53                 // 该属性用来设置百分位统计滚动窗口中使用 “桶”的数量。
54             @HystrixProperty(name = "metrics.rollingPercentile.numBuckets",
55             value = "60000"),
56                 // 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。如果在滚动时间窗
57                 // 内发生超过该设定值的执行次数，
58                 // 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秒，若在10秒内
59                 // 一个“桶”中发生了500次执行，
60                 // 那么该“桶”中只保留 最后的100次执行的统计。另外，增加该值的大小将会增加
61                 // 内存量的消耗，并增加排序百分位数所需的计算时间。
62             @HystrixProperty(name = "metrics.rollingPercentile.bucketSize",
63             value = "100"),
64                 // 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、 错误百分比）的间
65                 // 隔等待时间。
66             @HystrixProperty(name =
67             "metrics.healthSnapshot.intervalInMilliseconds", value = "500"),
68                 // 是否开启请求缓存
69             @HystrixProperty(name = "requestCache.enabled", value = "true"),
70                 // HystrixCommand的执行和事件是否打印日志到 HystrixRequestLog 中
71             @HystrixProperty(name = "requestLog.enabled", value = "true"),
72         },
73         threadPoolProperties = {
74             // 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大并发量
75             @HystrixProperty(name = "coreSize", value = "10"),
76                 // 该参数用来设置线程池的最大队列大小。当设置为 -1 时，线程池将使用
77             SynchronousQueue 实现的队列，
78                 // 否则将使用 LinkedBlockingQueue 实现的队列。
79             @HystrixProperty(name = "maxQueueSize", value = "-1"),
80                 // 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
81                 // 请求。
82             // 该参数主要是对 LinkedBlockingQueue 队列的补充，因为
83             LinkedBlockingQueue
84                 // 队列不能动态修改它的对象大小，而通过该属性就可以调整拒绝请求的队列大小了。
85             @HystrixProperty(name = "queueSizeRejectionThreshold", value =
86             "5"),
87         }
88     )
89     public String strConsumer() {
90         return "hello 2020";
91     }
92     public String str_fallbackMethod()
93     {
94         return "*****fall back str_fallbackMethod";
95     }

```

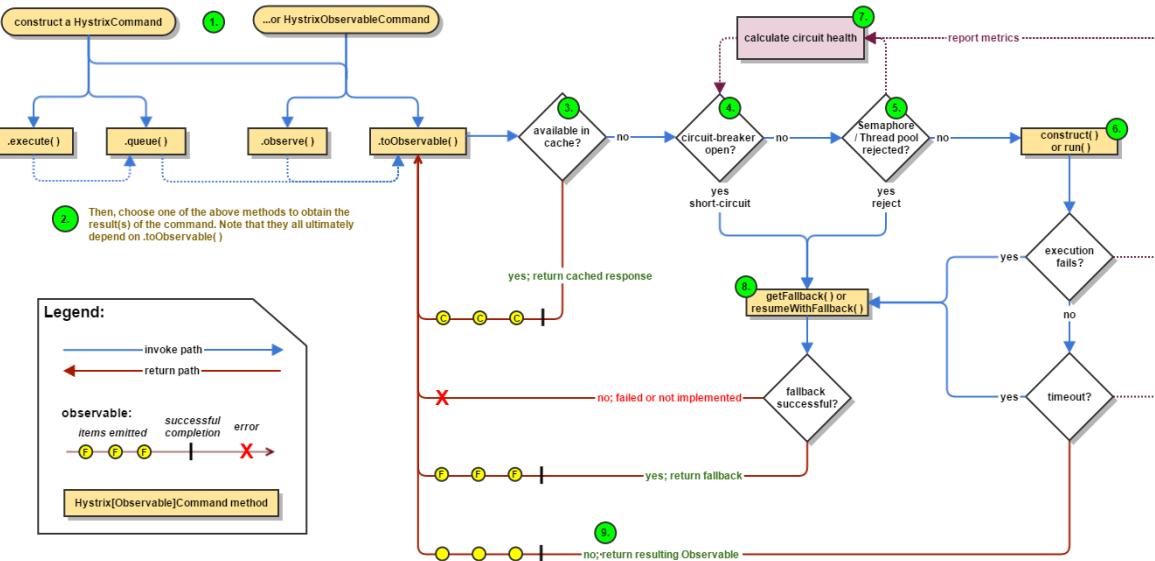
服务限流

后面讲解alibaba的Sentinel说明

Hystrix 工作流程

官网: <https://github.com/Netflix/Hystrix/wiki/How-it-Works>

工作流程图



一共九个步骤。

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。其中 HystrixComand 实现了下面前两种执行方式；而 HystrixObservableCommand 实现了后两种执行方式：
 1. 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。
 2. 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。
 3. 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)。
 4. 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)。
3. 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。
4. 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)。
5. 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 (不使用线程池的时候) 已经被占满，那么 Hystrix 也不会执行命令，而是转接到 fallback 处理逻辑 (第8步)。

6. 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。
7. 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"。
8. 当命令执行失败的时候，Hystrix 会进入 fallback 尝试回退处理，我们通常也称该操作为 "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：当前命令的线程池、请求队列或者信号量被占满的时候。第6步：HystrixObservableCommand.construct() 或 HystrixCommand.run() 抛出异常的时候。
9. 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 Observable 对象，但是它不会发射任何结果数据，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

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。

构建HystrixDashboard

1. 新建模块

新建 cloud-consumer-hystrix-dashboard9001

2. 改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. 写YML

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

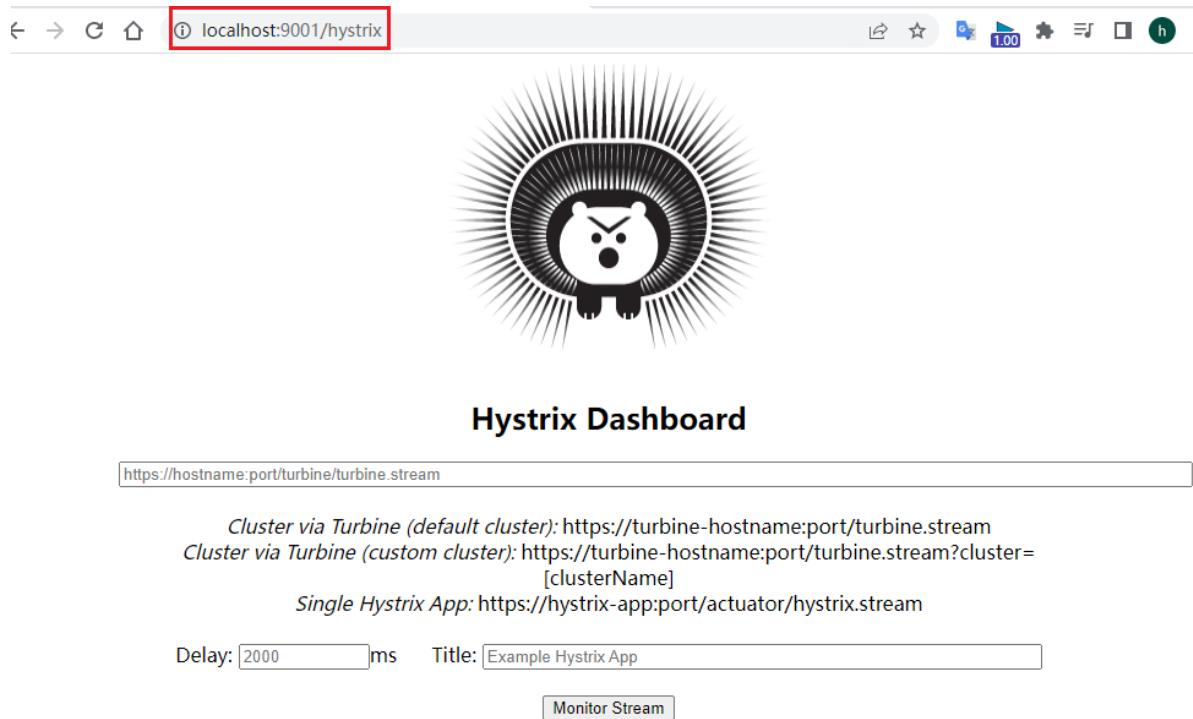
4. 主启动

```
1 @SpringBootApplication
2 @EnableHystrixDashboard
3 public class HystrixDashboardMain9001 {
4     public static void main(String[] args) {
5         SpringApplication.run(HystrixDashboardMain9001.class, args);
6     }
7 }
```

注意：`@EnableHystrixDashboard`注解表示启用Hystrix Dashboard功能。

启动 `cloud-consumer-hystrix-dashboard9001`，后续该微服务将监控微服务8001

启动9001后，访问地址：<http://localhost:9001/hystrix>



断路器演示（服务监控HystrixDashboard）

1. 修改cloud-provider-hystrix-payment8001

一、首先，在`cloud-provider-hystrix-payment8001`模块的pom.xml中一定要引入以下依赖：

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

二、在8001的主启动类中添加如下代码，用于指定监控路径

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableHystrix
4 public class PaymentHystrixMain8001 {
5     public static void main(String[] args) {
6         SpringApplication.run(PaymentHystrixMain8001.class, args);
7     }
8     /**
9      *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
10     *ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
11     *只要在自己的项目里配置上下面的servlet就可以了
12     */
13     @Bean
14     public ServletRegistrationBean getServlet() {
15         HystrixMetricsStreamServlet streamServlet = new
HystrixMetricsStreamServlet();
```

```
16     ServletRegistrationBean registrationBean = new
17     ServletRegistrationBean(streamServlet);
18     registrationBean.setLoadOnStartup(1);
19     registrationBean.addUrlMappings("/hystrix.stream");
20     registrationBean.setName("HystrixMetricsStreamServlet");
21     return registrationBean;
22 }
```

不建议配置bean，官网说明添加：management.endpoints.web.exposure.include: hystrix.stream 即可

2. 监控测试

1. 启动一个eureka7001单机版
2. 启动8001
3. 9001监控8001。填写监控地址：<http://localhost:8001/hystrix.stream>

The screenshot shows a web browser displaying the Hystrix Dashboard at <http://localhost:9001/hystrix>. The page features a large, stylized Hystrix logo with a bear face in the center. Below the logo, the text "Hystrix Dashboard" is displayed. A red box highlights the URL in the address bar. Further down, there are configuration fields: "Delay: [2000] ms" and "Title: T3", both with red boxes around them. A prominent red box also surrounds the "Monitor Stream" button.

点击 Monitor Stream 按钮后，出现如下界面

Hystrix Stream: T3

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

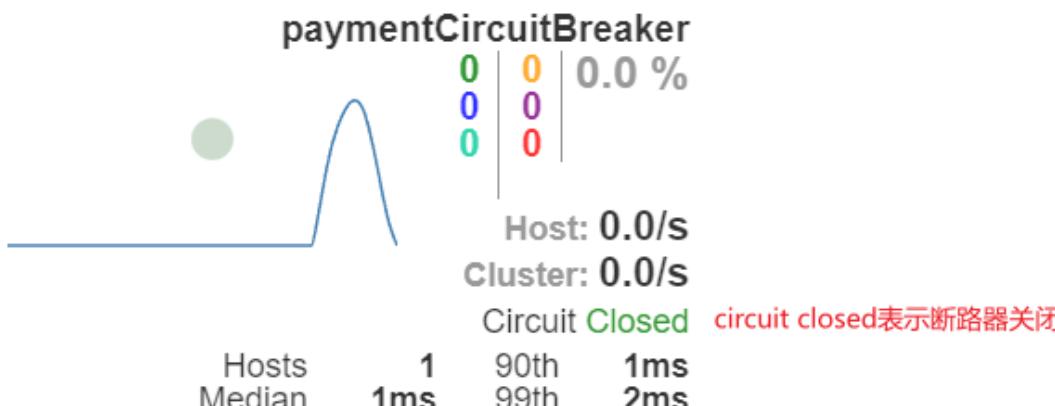


4. 多次访问8001中的方法: <http://localhost:8001/payment/circuit/31>

可以发现监控界面会发生如下变化:

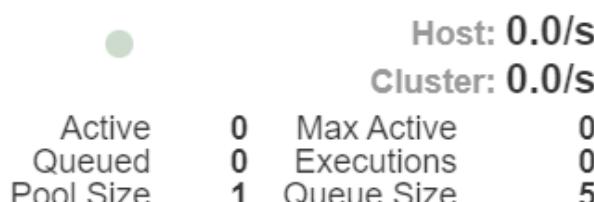
Hystrix Stream: T3

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#)



Thread Pools Sort: [Alphabetical](#) | [Volume](#) |

PaymentService

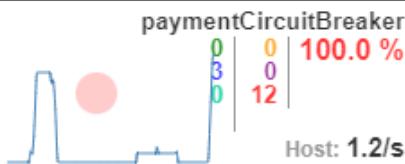


其中: Host表示峰值, Circuit表示断路器的状态

5. 多次访问地址: <http://localhost:8001/payment/circuit/-31>。让它造成微服务熔断。再次观察监控界面

Hystrix Stream: T3

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)
[Success](#) | [Short-Circuited](#) |



Host: 1.2/s
Cluster: 1.2/s

Circuit **Open**

Hosts	1	90th	1ms
Median	0ms	99th	1ms
Mean	0ms	99.5th	1ms

Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |

PaymentService

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	1	Queue Size	5

这里可以看到断路器处于Open状态，即造成了服务熔断

如何看监控界面

HystrixDashboard界面一般如下图所示。

Hystrix Stream: T3



Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)
[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)



Host: 0.0/s
Cluster: 0.0/s

Circuit **Closed**

Hosts	1	90th	0ms
Median	0ms	99th	0ms
Mean	0ms	99.5th	0ms

Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |

PaymentService

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	1	Queue Size	5

那么我们如何看懂监控界面呢？

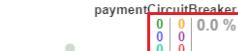
主要是看：

1. 7色。界面右侧有7种颜色，对应7种请求处理的状态

Hystrix Stream: T3



Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)
[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)



1. 7色

对应这里

Hosts	1	90th	0ms
Median	0ms	99th	0ms
Mean	0ms	99.5th	0ms

Thread Pools

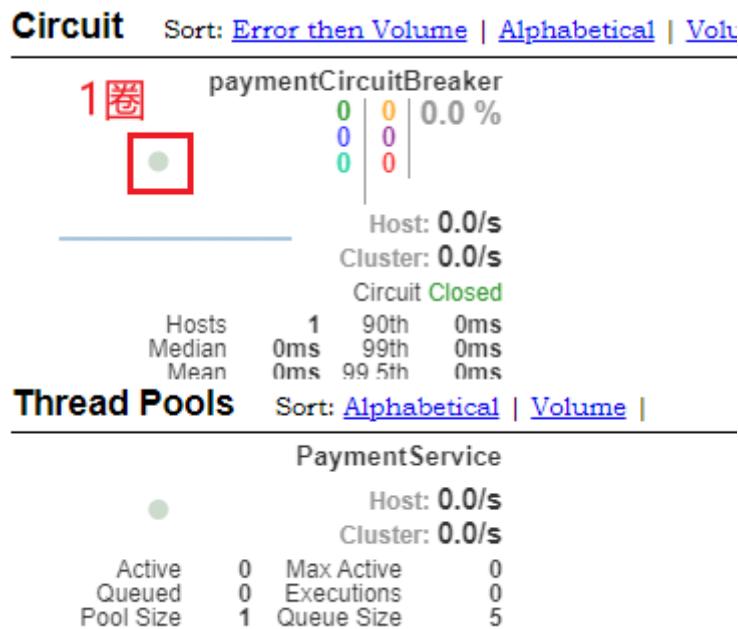
Sort: [Alphabetical](#) | [Volume](#) |

PaymentService

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	1	Queue Size	5

2. 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。

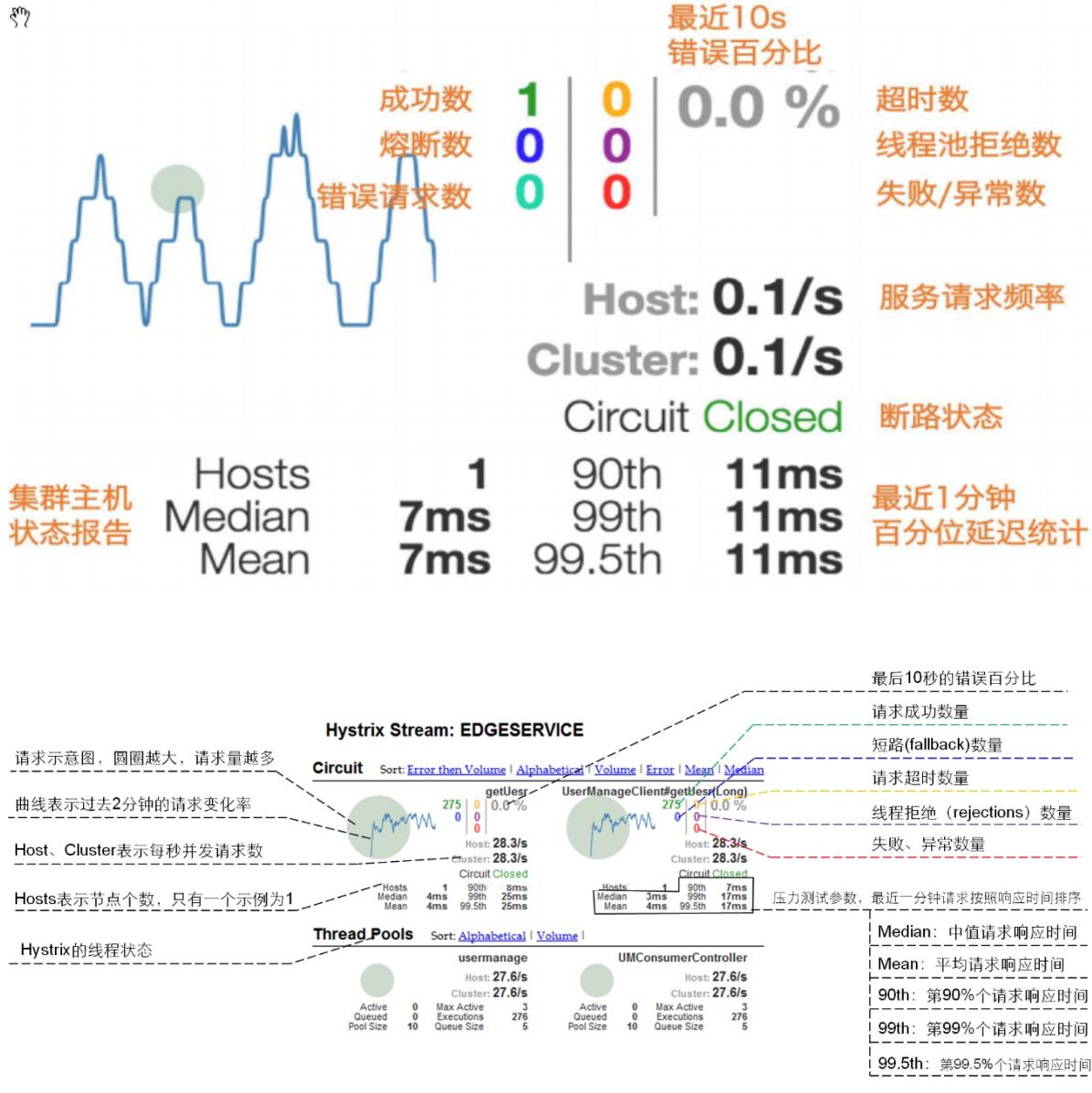
该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，就可以在大量的实例中快速的发现故障实例和高压力实例。



3. 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：



小结

可以看到Hystrix的服务监控HystrixDashboard需要自己去搭建一个平台（例如我们搭建的9001），借用该平台才能监控到其他微服务状态。

后续的Alibaba Sentinel会更加方便。

十、服务网关

服务网关一般有 **zuul** 和 **Gateway** 两种技术，这里只学Gateway就行！

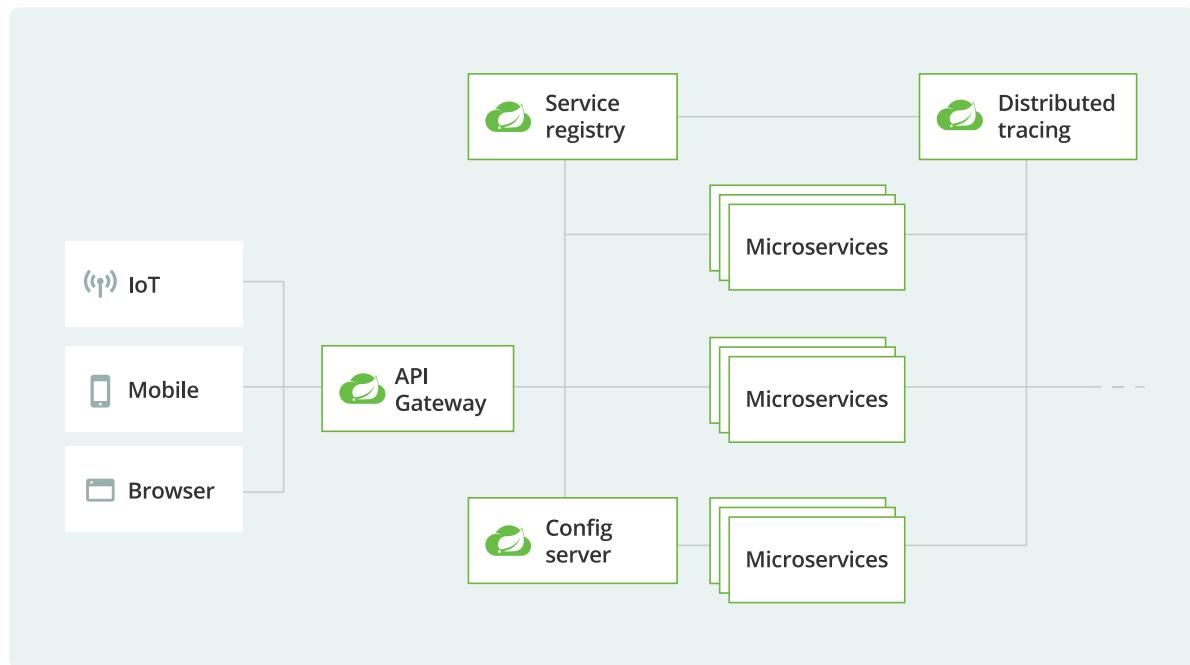
zuul 的官网：<https://github.com/Netflix/zuul>

Gateway 的官网：<https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.1.RELEASE/reference/html/>

Gateway简介

Gateway概述

Spring Cloud的架构如下：



上面有个模块 **API Gateway** 表示的就是网关这个组件

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。
总结SpringCloud Gateway 就是一句话：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

The screenshot shows the official Spring Cloud Gateway documentation page. At the top, there's a navigation bar with links to 'How to Include Spring Cloud Gateway', 'Glossary', 'How It Works', 'Route Predicate Factories', 'GatewayFilter Factories', 'Global Filters', 'TLS and SSL', 'Configuration', and 'Route Metadata Configuration'. Below the navigation, the main content area has a title 'Spring Cloud Gateway' and a sub-section '2.2.1.RELEASE'. A red-bordered box contains the following text: 'This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. 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.'

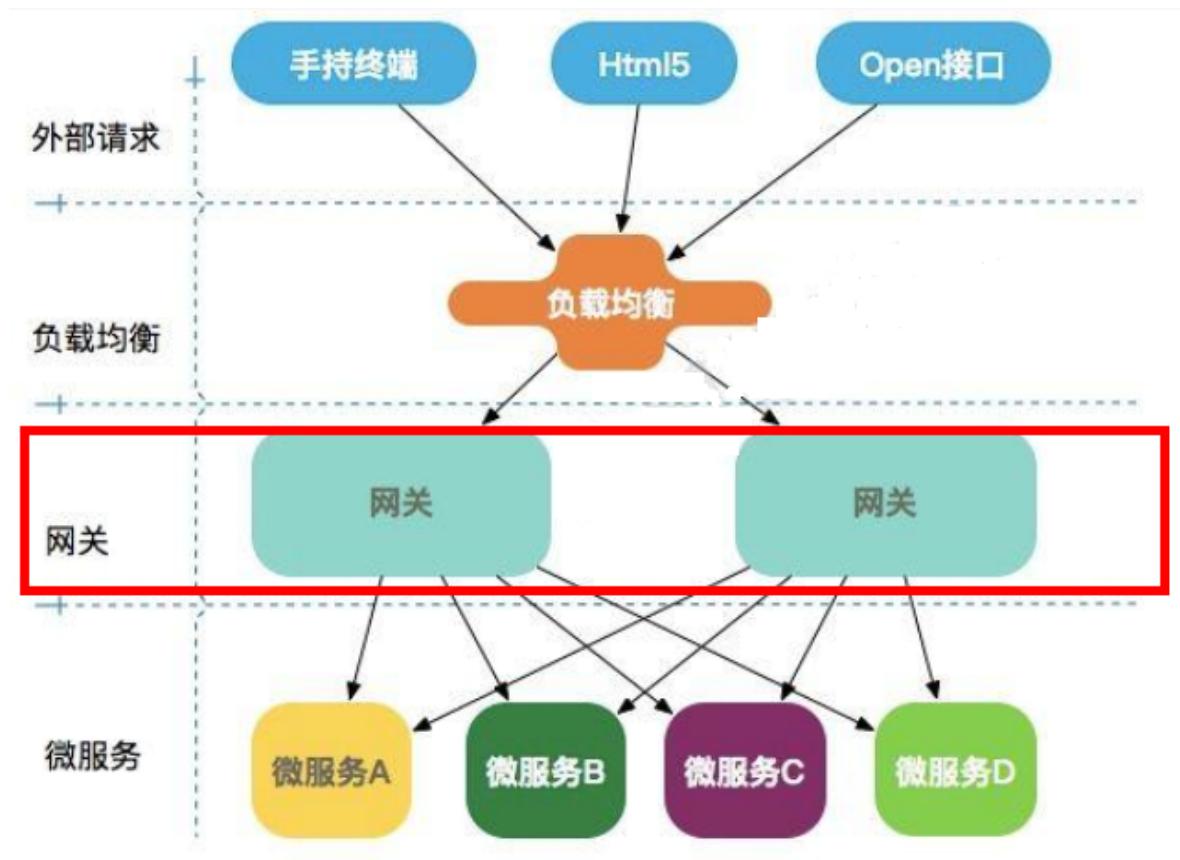
SpringCloud Gateway 是 Spring Cloud 的一个全新项目，基于 Spring 5.0+Spring Boot 2.0 和 Project Reactor 等技术开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

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。

SpringCloud Gateway 使用的Webflux中的reactor-netty响应式编程组件，底层使用了Netty通讯框架。

Gateway能干什么



Spring Cloud Gateway能干什么

- 反向代理
- 鉴权
- 流量控制
- 熔断
- 日志监控
-

有了Zuul，为什么还要选择Gateway？

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。

SpringCloud Gateway具有如下特性：

1. 基于Spring Framework 5, Project Reactor 和 Spring Boot 2.0 进行构建；
2. 动态路由：能够匹配任何请求属性；
3. 可以对路由指定 Predicate（断言）和 Filter（过滤器）；
4. 集成Hystrix的断路器功能；
5. 集成 Spring Cloud 服务发现功能；
6. 易于编写的 Predicate（断言）和 Filter（过滤器）；
7. 请求限流功能；
8. 支持路径重写。

SpringCloud Gateway 与 Zuul 的区别：

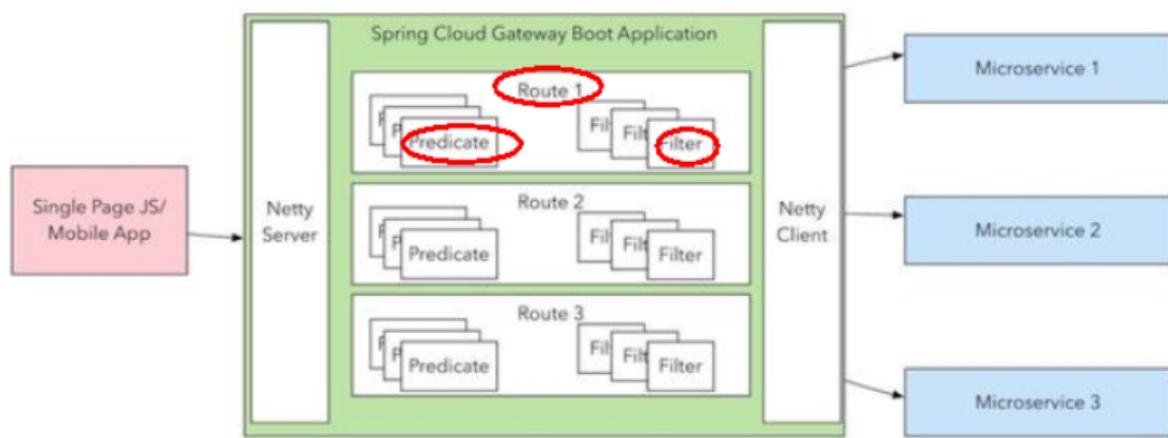
1. Zuul 1.x，是一个基于阻塞 I/O 的 API Gateway
2. Zuul 1.x **基于Servlet 2.5，使用阻塞架构**，它不支持任何长连接(如 WebSocket)。Zuul 的设计模式和Nginx较像，每次 I/O 操作都是从工作线程中选择一个执行，请求线程被阻塞到工作线程完成，但是差别是Nginx 用C++ 实现，Zuul 用 Java 实现，而 JVM 本身会有第一次加载较慢的情况，使得Zuul 的性能相对较差。
3. Zuul 2.x理念更先进，想基于Netty非阻塞和支持长连接，但SpringCloud目前还没有整合。Zuul 2.x的性能较 Zuul 1.x 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。
4. Spring Cloud Gateway 建立在 Spring Framework 5、Project Reactor 和 Spring Boot 2 之上，使用非阻塞 API。
5. Spring Cloud Gateway 还支持 WebSocket，并且与Spring紧密集成拥有更好的开发体验

三大核心概念

三大核心概念为：

- 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。如果断言为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；

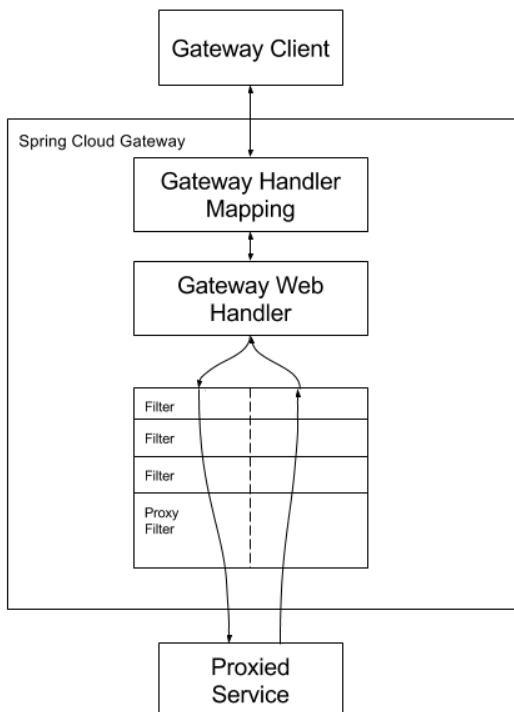
filter，就可以理解为一个无所不能的拦截器。有了这两个元素，再加上目标uri，就可以实现一个具体的路由了；

Gateway工作流程

[官网](#)描述的工作流程：

3. How It Works

The following diagram provides a high-level overview of how Spring Cloud Gateway works:



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.

1. 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 Gateway Web Handler。
2. 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。

Gateway的核心逻辑：路由转发+执行过滤器链

入门配置

新建Module

新建一个module，名为 `cloud-gateway-gateway9527`

改pom

```
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>com.atguigu.springcloud</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>
```

注：主要是引入这两个包。这里注意，**网关也要注册进注册中心**。

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

写YML

```
1 server:
2     port: 9527
3 spring:
4     application:
5         name: cloud-gateway
6 eureka:
7     instance:
8         hostname: cloud-gateway-service
9     client:
10        service-url:
11            register-with-eureka: true
12            fetch-registry: true
13            defaultZone: http://eureka7001.com:7001/eureka
```

主启动

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class GatewayMain9527 {
4     public static void main(String[] args) {
5         SpringApplication.run(GatewayMain9527.class, args);
6     }
7 }
```

由于这个模块是用于做网关的，所以不需要业务类

制造一个需求

假设，我们目前以 `cloud-provider-payment8001` 模块为例，它的controller中有以下两个方法

```
@GetMapping("/payment/get/{id}")
public CommonResult getPaymentById(@PathVariable("id") Long id) {
    Payment paymentById = paymentService.getPaymentById(id);
    log.info("*****查询结果: " + paymentById);
    if (paymentById != null) {
        return new CommonResult(code: 200, message: "查询成功, serverPort: " + serverPort, paymentById);
    } else {
        return new CommonResult(code: 444, message: "没有对应记录, 查询失败");
    }
}

@GetMapping("/payment/lb")
public String getPaymentLB() { return serverPort; }
```

我们在这里制造一个需求：假设，我们目前不想暴露8001端口，希望在8001外面套一层9527，以此来访问上述两个方法。

下面我们进行网关配置。

网关配置

在 `cloud-gateway-gateway9527` 模块的 `application.yml` 中，添加如下配置：

```
1  spring:
2    cloud:
3      gateway:
4        routes:
5          - 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
6            uri: http://localhost:8001      # 匹配后提供服务的路由地址
7            predicates:
8              - Path=/payment/get/**     # 断言，路径相匹配的进行路由。这里 ** 表示通配符
9          - 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
10         uri: http://localhost:8001
11         predicates:
12           - Path=/payment/lb/**    # 断言，路径相匹配的进行路由。这里 ** 表示通配符
```

测试

1. 启动7001
2. 启动 `cloud-provider-payment8001`
3. 启动9527网关
4. 访问：<http://eureka7001.com:7001/>

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-GATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-7CK6BAS:cloud-gateway:9527
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - payment8001

5. 在没有网关时，我们访问8001的方法都是访问如下地址：

1. <http://localhost:8001/payment/get/31>
2. <http://localhost:8001/payment/lb>

6. 在有网关后，我们通过如下地址来访问8001的方法：

1. <http://localhost:9527/payment/get/31>
2. <http://localhost:9527/payment/lb>

```
{  
    code: 200,  
    message: "查询成功, serverPort: 8001",  
    - data: {  
        id: 31,  
        serial: "尚硅谷001"  
    }  
}  
  
8001
```

访问说明：

能这么访问，是因为我们在9527中进行了相应设置。

```
application.yml  
server:  
  port: 9527  
spring:  
  application:  
    name: cloud-gateway  
  cloud:  
    gateway:  
      routes:  
        - id: payment_routh # payment_route # 路由的ID  
          uri: http://localhost:8001 # 匹配后提供服务的地址  
          predicates:  
            - Path=/payment/get/** # 断言, 路径相匹配的进行  
            - id: payment_routh2 # payment_route # 路由的ID  
              uri: http://localhost:8001  
              predicates:  
                - Path=/payment/lb/** # 断言, 路径相匹配的进行  
  
PaymentController.java  
8001  
@GetMapping("/payment/get/{id}")  
public CommonResult getPaymentById(@PathVariable("id") Long id) {  
    Payment paymentById = paymentService.getPaymentById(id);  
    log.info("*****查询结果: " + paymentById);  
    if (paymentById != null) {  
        return new CommonResult( code: 200, message: "查询成功, serverPort: 8001" )  
    } else {  
        return new CommonResult( code: 404, message: "没有对应记录, 查找失败" )  
    }  
}  
  
@GetMapping("/payment/lb")  
public String getPaymentLB() {  
    return serverPort;  
}
```

Gateway 网关路由的两种配置方式

在上述这个案例中，Gateway在yml配置文件中进行配置。

其实，还有一种配置方式，即在代码中注入RouteLocator的Bean。

官网案例：

Using Java config:

GatewayConfig.java

```
RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...
.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1"))
.route("proxied-route",
    r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2"))
)
```

接下来，我们自己用这种配置方式写一个案例！

业务需求：通过9527网关访问到外网的百度新闻网址（<https://news.baidu.com/guonei>）

步骤如下：

编写配置类

在 `cloud-gateway-gateway9527` 中，创建一个配置类

```
1  @Configuration
2  public class GateWayConfig {
3      @Bean
4      public RouteLocator customRouteLocator(RouteLocatorBuilder
5          routeLocatorBuilder) {
6          // https://news.baidu.com/guonei
7          return routeLocatorBuilder.routes()
8              .route("path_route_atguigu", r -> r.path("/guonei")
9                  .uri("https://news.baidu.com")).build();
10     }
11 }
```

测试

重新启动9527微服务，然后访问地址：<http://localhost:9527/guonei>



通过微服务名实现动态路由

默认情况下，Gateway需要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。

演示案例

为了制作演示案例，我们使用一个eureka7001作为注册中心，两个服务提供者 `cloud-provider-payment8001`、`cloud-provider-payment8002`

修改yml

对 `cloud-gateway-gateway9527` 中的yaml配置文件修改配置：

```
1  spring:
2    cloud:
3      gateway:
4        discovery:
5          locator:
6            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
7          routes:
8            - id: payment_routh # payment_route    # 路由的ID，没有固定规则但要求唯一，建议
9              配合服务名
10             # uri: http://localhost:8001      # 匹配后提供服务的路由地址
11             uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
12             predicates:
13               - Path=/payment/get/**      # 断言，路径相匹配的进行路由。这里 ** 表示通配符
```

```
13         - 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
14             # uri: http://localhost:8001
15             uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
16             predicates:
17                 - Path=/payment/lb/** # 断言, 路径相匹配的进行路由。这里 ** 表示通配符
```

主要是对 `spring.cloud.gateway` 下的配置进行如上修改！

注：

- 对 `uri: lb://cloud-payment-service` 进行解释
 - 其中的 `lb` 表示的是路由的一种通信协议，它实现了负载均衡（load balance）通信能力
 - 其中的 `cloud-payment-service` 表示的是注册中心的一个服务名

测试

启动上述的各个微服务。

1. 在浏览器上多次访问：<http://localhost:9527/payment/lb>

可以发现是使用的轮询策略访问的两个微服务。

Predicate的使用

Predicate是什么

在我们每次启动9527这个网关时，会发现每次都有如下日志信息：

```
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [After]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Before]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Between]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Cookie]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Header]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Host]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Method]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Path]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Query]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [ReadBodyPredicateFactory]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [RemoteAddr]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [Weight]
.s.c.g.r.RouteDefinitionRouteLocator : Loaded RoutePredicateFactory [CloudFoundryRouteService]
.s.b.d.a.OptionalLiveReloadServer   : Unable to start LiveReload server
```

Route Predicate Factories是什么？

4. Configuring Route Predicate Factories and
Gateway Filter Factories

5. Route Predicate Factories

5.1. The After Route Predicate Factory

- 5.2. The Before Route Predicate Factory
- 5.3. The Between Route Predicate Factory
- 5.4. The Cookie Route Predicate Factory
- 5.5. The Header Route Predicate Factory
- 5.6. The Host Route Predicate Factory
- 5.7. The Method Route Predicate Factory
- 5.8. The Path Route Predicate Factory
- 5.9. The Query Route Predicate Factory
- 5.10. The RemoteAddr Route Predicate Factory
- 5.11. The Weight Route Predicate Factory

6. GatewayFilter Factories

7. Global Filters

8. HttpHeadersFilters

9. TLS and SSL

10. Configuration

11. Route Metadata Configuration

5. Route Predicate Factories

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.

5.1. The After Route Predicate Factory

The `After` route predicate factory takes one parameter, a `datetime` (which is a java `ZonedDateTime`). This predicate matches requests that happen after the specified datetime. The following example configures an after route predicate:

Example 1. application.yml

```
spring:  
  cloud:  
    gateway:  
      routes:  
        - id: after_route  
          uri: https://example.org  
          predicates:  
            - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

YAML

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver).

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Route Predicate工厂可以进行组合

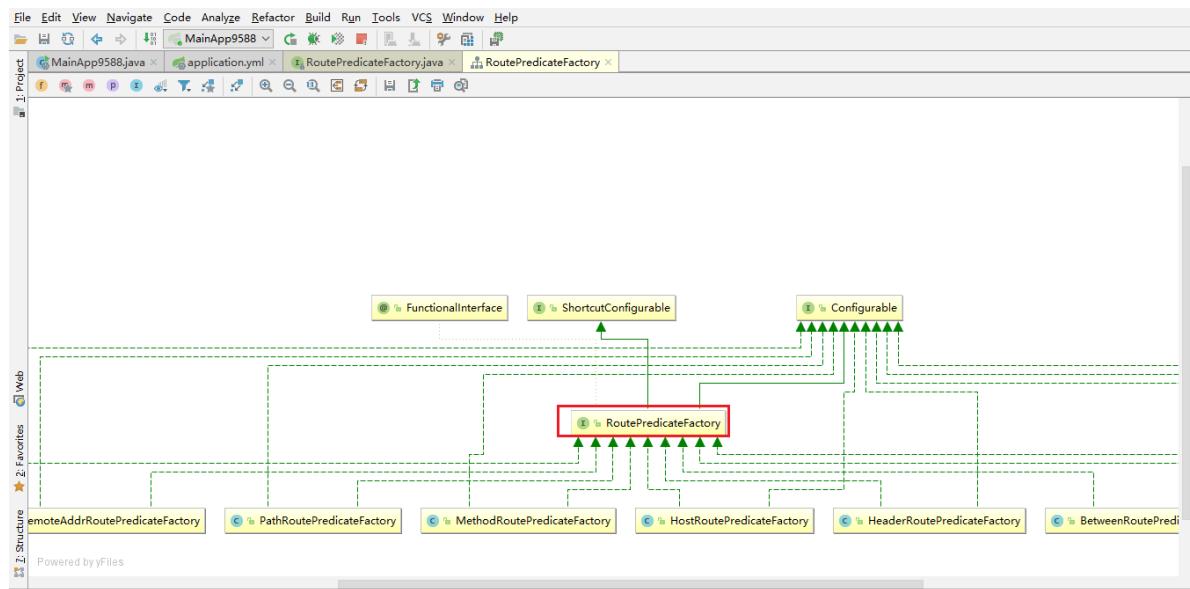
Spring Cloud Gateway 创建 Route 对象时，使用 `RoutePredicateFactory` 创建 Predicate 对象，
Predicate 对象可以赋值给 Route。Spring Cloud Gateway 包含许多内置的Route Predicate Factories。

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。

常用的Route Predicate

1. After Route Predicate
2. Before Route Predicate
3. Between Route Predicate
4. Cookie Route Predicate
5. Header Route Predicate
6. Host Route Predicate
7. Method Route Predicate
8. Path Route Predicate
9. Query Route Predicate

上述常用的Route Predicate，都实现了 `RoutePredicateFactory` 这个接口



演示After Route Predicate

首先，从官网给的教程来看，我们需要先构造出相应格式的时间

4.1. The After Route Predicate Factory

The after route predicate factory takes one parameter, a datetime. This predicate matches requests that happen after the specified datetime. The following example configures an after route predicate:

Example 1. application.yml

```

spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            - After=2017-01-20T17:42:47.789-07:00[America/Denver]
  
```

This route matches any request made after Jan 20, 2017 17:42 Mountain Time (Denver).

获取相应格式的时间

- 在 `cloud-gateway-gateway9527` 中编写一个测试类

```

1  public class T2 {
2      public static void main(String[] args) {
3          ZonedDateTime zbj = ZonedDateTime.now();      // 默认时区
4          System.out.println(zbj);
5      }
6  }
  
```

运行后，就可以得到相应格式的时间：

```
1  2022-05-03T16:12:18.765+08:00[Asia/Shanghai]
```

修改YML

在获取相应格式的时间后，接下来就需要修改 `cloud-gateway-gateway9527` 的yml配置文件：

```
spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      discovery:
        locator:
          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
      routes:
        - 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
          # uri: http://localhost:8001 # 匹配后提供服务的路由地址
          uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
          predicates:
            - Path=/payment/get/** # 断言，路径相匹配的进行路由。这里 ** 表示通配符
        - 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
          # uri: http://localhost:8001
          uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
          predicates:
            - Path=/payment/lb/** # 断言，路径相匹配的进行路由。这里 ** 表示通配符
            - After=2022-05-03T16:12:18.765+08:00[Asia/Shanghai]
```

只需模仿官方教程，修改这里，就可以让访问 `/payment/lb` 这个方法时 只允许在这个时间之后的请求才能访问。

演示Cookie Route Predicate

官网给的教程如下

4.4. The Cookie Route Predicate Factory

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. The following example configures a cookie route predicate factory:

Example 4. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: cookie_route
          uri: https://example.org
          predicates:
            - Cookie=chocolate, ch.p
```

YAML

This route matches requests that have a cookie named `chocolate` whose value matches the `ch.p` regular expression.

Cookie Route Predicate 需要两个参数：一个是Cookie name，一个是正则表达式。

路由规则会通过获取对应的 Cookie name 值和正则表达式去匹配，如果匹配上就会执行路由，如果没有匹配上则不执行。

修改YML

同样，为了演示 Cookie Route Predicate 的效果，对yml进行修改：

```
cloud:
  gateway:
    discovery:
    locator:
      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
    routes:
      - 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
        # uri: http://localhost:8001 # 匹配后提供服务的路由地址
        uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
        predicates:
          - Path=/payment/get/** # 断言，路径相匹配的进行路由。这里 ** 表示通配符
      - 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
        # uri: http://localhost:8001
        uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
        predicates:
          - Path=/payment/lb/** # 断言，路径相匹配的进行路由。这里 ** 表示通配符
          - After=2022-05-03T16:12:18.765+08:00[Asia/Shanghai]
          - Cookie=username,zzyy
```

上述Cookie的配置的意思是：只有当请求中的Cookie含有username=zzyy这个键值对的才允许访问。

测试：

这里测试是直接使用curl进行测试的。具体curl是什么，可以自行百度。

1. 测试不带cookie进行访问：

在cmd命令行中，执行以下命令：

```
1 curl http://localhost:9527/payment/lb
```

会显示如下报错信息。

```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.19041.867]
(c) 2020 Microsoft Corporation. 保留所有权利。

C:\Users\H>curl http://localhost:9527/payment/lb
{"timestamp": "2022-05-03T08:28:24.797+0000", "path": "/payment/lb", "status": 404, "error": "Not Found", "message": null, "requestId": "7a67eb0e", "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\tat org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]\n\t\t\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\t\t\tat reactor.core.publisher.MonoDefer.subscribe(MonoDefer.java:44)"}
```

2. 测试带cookie进行访问：

在cmd命令行中，执行以下命令：

```
1 curl http://localhost:9527/payment/lb --cookie "username=zzyy"
```

可以发现访问成功

```
C:\Users\H>curl http://localhost:9527/payment/lb --cookie "username=zzyy"
8001
C:\Users\H>
```

演示Header Route Predicate

官网教程

4.5. The Header Route Predicate Factory

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. The following example configures a header route predicate:

Example 5. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: header_route
          uri: https://example.org
          predicates:
            - Header=X-Request-Id, \d+
```

YAML

This route matches if the request has a header named `X-Request-Id` whose value matches the `\d+` regular expression (that is, it has a value of one or more digits).

上述配置的意思是：如果请求具有名为 `X-Request-Id` 且其值与 `\d+` 正则表达式（即，具有一位或多位数字的值）匹配的请求头，则此路由匹配。

修改YML

同样，为了演示 Header Route Predicate 的效果，对yml进行修改：

```
gateway:
  discovery:
    locator:
      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
  routes:
    - 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
      # uri: http://localhost:8001      # 匹配后提供服务的路由地址
      uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
      predicates:
        - Path=/payment/get/**    # 断言，路径相匹配的进行路由。这里 ** 表示通配符
    - 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
      # uri: http://localhost:8001
      uri: lb://cloud-payment-service # 匹配后提供服务的路由地址
      predicates:
        - Path=/payment/lb/**    # 断言，路径相匹配的进行路由。这里 ** 表示通配符
        - After=2022-05-03T16:12:18.765+08:00[Asia/Shanghai]
        - Header=X-Request-Id, \d+
          - Cookie=username,zzyy
```

测试：

在cmd中执行如下命令：

```
1 curl http://localhost:9527/payment/lb -H "X-Request-Id:123"
```

可以看到成功访问

```
8002  
C:\Users\H>curl http://localhost:9527/payment/lb -H "X-Request-Id:123"  
8002  
C:\Users\H>  
C:\Users\H>curl http://localhost:9527/payment/lb -H "X-Request-Id:123"  
8001  
C:\Users\H>
```

Filter的使用

官网：

<https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.1.RELEASE/reference/html/#gateway-factories>

<https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.1.RELEASE/reference/html/#global-filters>

5. GatewayFilter Factories

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.



For more detailed examples of how to use any of the following filters, take a look at the [unit tests](#).

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。

Filter简介

Spring Cloud Gateway的Filter的生命周期有两个：

- pre (在业务逻辑之前)
- post (在业务逻辑之后)

这里有点像Spring的前置通知和后置通知。

Filter的种类有两个：

- GatewayFilter：单一的。 (官网上描述有31种)
- GlobalFilter：全局的。 (官网上描述有10种)

常用的GatewayFilter

GatewayFilter有31种配置，具体需要看官网。这里举个 `AddRequestParameter GatewayFilter` 的例子。

5.2. The `AddRequestParameter GatewayFilter` Factory

The `AddRequestParameter GatewayFilter` Factory takes a name and value parameter. The following example configures an `AddRequestParameter GatewayFilter`:

Example 15. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: https://example.org
          filters:
            - AddRequestParameter=red, blue
```

YAML

This will add `red=blue` to the downstream request's query string for all matching requests.

`AddRequestParameter` is aware of the URI variables used to match a path or host. URI variables may be used in the value and are expanded at runtime. The following example configures an `AddRequestParameter GatewayFilter` that uses a variable:

Example 16. application.yml

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: https://example.org
          predicates:
            - Host: {segment}.myhost.org
          filters:
            - AddRequestParameter=foo, bar-{segment}
```

YAML

在上图的Example 15中，这个Filter的配置，会将 `red=blue` 添加到所有匹配请求的请求查询参数中。

自定义过滤器

自定义全局GlobalFilter 需要实现 `GlobalFilter` 和 `Ordered` 两个接口

自定义全局GlobalFilter能够：

- 全局日志记录
- 统一网关鉴权
-

案例代码

在 `cloud-gateway-gateway9527` 中创建一个类

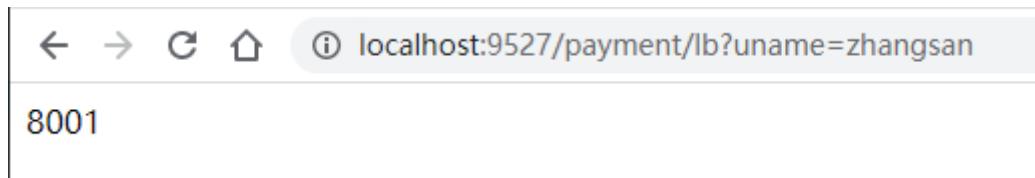
```
1  @Component
2  @Slf4j
3  public class MyLogGatewayFilter implements GlobalFilter, Ordered {
4      @Override
```

```
5     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
6         chain) {
7             log.info("*****come in MyLogGatewayFilter: " + new Date());
8             // 获取请求参数中key为uname的value值
9             String uname = exchange.getRequest().getQueryParams().getFirst("uname");
10            if (uname == null) {
11                log.info("*****用户名为null, 非法用户。o(╥﹏╥)o");
12                exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
13                return exchange.getResponse().setComplete();
14            }
15            return chain.filter(exchange);
16        }
17    /**
18     * 获取加载过滤器的顺序。数字越小，优先级越高
19     * @return
20     */
21    @Override
22    public int getOrder() {
23        return 0;
24    }
25 }
```

测试

- 用浏览器访问: <http://localhost:9527/payment/lb?uname=zhangsan>

可以正常访问



- 用浏览器访问: <http://localhost:9527/payment/lb>

发现无法访问



该网页无法正常运作

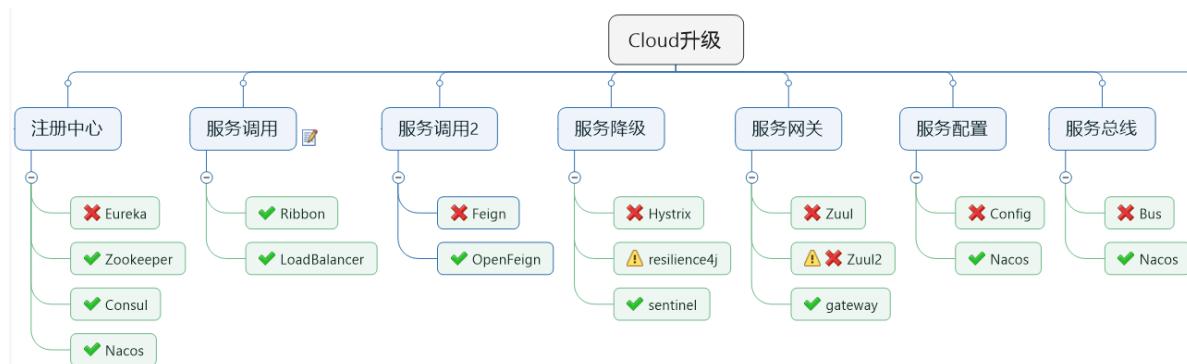
如果问题仍然存在，请与网站所有者联系。

HTTP ERROR 406

重新加载

十一、SpringCloud Config 分布式配置中心

讲了这么多，终于讲到服务配置了.....目前 **Config** 和 **Bus** 虽然没有停更维护，但是却被Alibaba的技术所替代了。



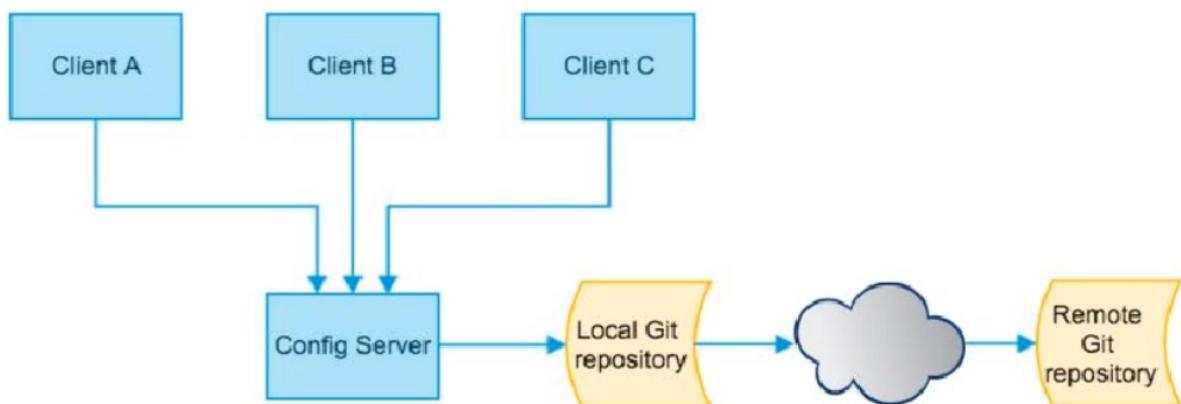
简介

分布式系统面临的问题——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自服务，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。

Config 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**。



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。

Config 的作用

SpringCloud Config的作用有：

- 集中管理配置文件
- 不同环境不同配置，动态的配置更新，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、curl访问刷新均可

与GitHub整合配置

由于 Spring Cloud Config 默认使用 Git 来存储配置文件（也有其他方式，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/>

**Quick Start**

- [Client Side Usage](#)
- [Spring Cloud Config Server](#)
- [Serving Alternative Formats](#)
- [Serving Plain Text](#)
- [Embedding the Config Server](#)
- [Push Notifications and Spring Cloud Bus](#)
- [Spring Cloud Config Client](#)

Spring Cloud Config

2.2.1.RELEASE

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for applications across all environments. The concepts on both client and server map identically to the `Environment` and `PropertySource` abstractions, so they fit very well with Spring applications but can be used with any application running in any language. As an application moves through the deployment pipeline from dev to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate. The default implementation of the server storage backend uses git, so it easily supports labelled versions of configuration environments as well as being accessible to a wide range of tooling for managing the content. It is easy to add alternative implementations and plug them in with Spring configuration.

Config服务端配置与测试

在GitHub上创建一个仓库

登录自己的GitHub账号，新建一个名为 `springcloud-config` 的新仓库

下面的yml配置文件，请参考周阳老师的github：<https://github.com/zzyybs/springcloud-config> 或者直接从其中拷贝出来：<https://github.com/zzyybs/springcloud-config.git>

The screenshot shows a GitHub repository page for 'springcloud-config'. At the top, there are buttons for 'master' (with 1 branch), '0 tags', 'Go to file', 'Add file', and 'Code'. Below is a commit history table:

hugh first commit	b4ac3cf 1 minute ago	1 commit
README.md	first commit	1 minute ago
config-dev.yml	first commit	1 minute ago
config-prod.yml	first commit	1 minute ago
config-test.yml	first commit	1 minute ago

Below the table is the 'README.md' file content:

```
springcloud-config
```

springcloud-config的配置中心

springcloud-config的配置中心

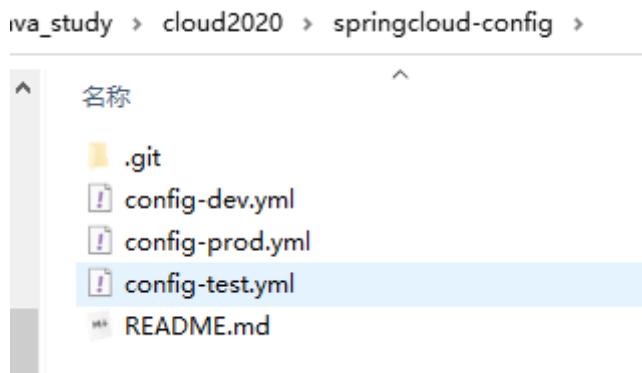
注：由于在国内github比较难访问，建议换成gitee的仓库！！！

在本地新建git仓库并clone

```
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020
$ pwd
/d/java_study/cloud2020

H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020
$ git clone git@github.com:hi_123/springcloud-config.git
Cloning into 'springcloud-config'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 0), reused 6 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.

H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020
$ |
```



可以看到此时，在本地硬盘目录下有多个配置文件。这些配置文件表示多个环境的配置文件。

- 开发环境：config-dev.yml
- 生产环境：config-prod.yml
- 测试环境：config-test.yml

新建模块

在本地新建Module模块：[cloud-config-center-3344](#)，用它作为Cloud的配置中心模块

改POM

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.cloud</groupId>
4         <artifactId>spring-cloud-config-server</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.boot</groupId>
12        <artifactId>spring-boot-starter-web</artifactId>
13    </dependency>
14
15    <dependency>
```

```

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

```

从上可以看到主要又新加了一个包

```

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

```

YML

```

1  server:
2      port: 3344
3  spring:
4      application:
5          name: cloud-config-center      # 注册进eureka服务器的微服务名
6  cloud:
7      config:
8          server:
9              git:
10                 uri: git@github.com:hugh-98/springcloud-config.git      # GitHub 上面的
11                   git 仓库名字
12                   # 搜索目录
13                   search-paths:
14                     - springcloud-config

```

```
14      # 读取分支
15      label: master
16
17  # 服务注册进eureka
18 eureka:
19   client:
20     service-url:
21       defaultZone: http://localhost:7001/eureka
```

在之前的模块中，我们将服务注册进eureka还需要如下配置。

```
eureka:
  client:
    # 表示是否将自己注册进EurekaServer, 默认为true
    register-with-eureka: true
    # 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true才能配合ribbon使用负载均衡
    fetch-registry: true
  service-url:
    #      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
    defaultZone: http://eureka7001.com:7001/eureka
```

但其实，上面的两行配置是默认为 `true` 的，所以我们在这个模块没有写。

```
# 服务注册进eureka
eureka:
  client:
    regist|
```

The screenshot shows a code editor with the Java code above. A red box highlights the configuration block under the `eureka:client:` key. Below the code, a tooltip provides detailed information about the configuration keys:

- `eureka.client.register-with-eureka=true` (Indicates whether to register with EurekaServer)
- `eureka.client.fetch-registry=true` (Indicates whether to fetch registry information from EurekaServer)
- `eureka.client.backup-registry-impl` (Gets the backup registry implementation)
- `eureka.client.fetch-remote-regions-registry` (Gets the remote regions registry)
- `eureka.client.registry-fetch-interval-seconds` (Gets the interval between registry fetches)
- `eureka.client.registry-refresh-single-vip-address` (Gets the single vip address for refresh)
- `eureka.client.should-enforce-registration-at-init` (Determines if registration should be enforced at init)
- `eureka.client.should-unregister-on-shutdown` (Determines if the client should unregister on shutdown)

At the bottom of the tooltip, there is a note: "Press Ctrl+逗号 to see all configuration keys with different prefix Next Tip".

主启动类

```
1  @SpringBootApplication
2  @EnableConfigServer
3  public class ConfigCenterMain3344 {
4      public static void main(String[] args) {
5          SpringApplication.run(ConfigCenterMain3344.class, args);
6      }
7  }
```

注意需要加注解 `@EnableConfigServer`

测试

在测试前，我们在windows下修改 `C:\Windows\System32\drivers\etc\hosts` 文件，增加映射。

```
127.0.0.1      activate.navicat.com
127.0.0.1      eureka7001.com
127.0.0.1      eureka7002.com
127.0.0.1      config-3344.com
```

然后测试，通过Config微服务是否可以从GitHub上获取配置内容。

1. 先启动eureka7001
2. 再启动微服务3344

注：在配置中心连接gitee的时候，有可能报错！

我的解决方案如下：

```
application:
  name: cloud-config-center  # 注册进eureka服务器的微服务名
cloud:
  config:
    server:
      git:
        uri: https://gitee.com/.../springcloud-config.git      # GitHub 上面的 git 仓库名字
        # 搜索目录
        search-paths:
          - springcloud-config
        username: [REDACTED]
        password: [REDACTED]
      # 读取分支
      label: master
```

如上图所示。我将uri换成了https格式的链接，并加上了gitee账号的用户名username和密码password。这样能连接成功。

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

可以读取到远程gitee仓库中的配置文件！

 A screenshot of a web browser window. The address bar shows the URL `http://config-3344.com:3344/master/config-dev.yml`. The page content is visible below the address bar.

```
config:
  info: master branch, springcloud-config/config-dev.yml version=1
```

以上测试，说明了 Config Server 能够成功连接到远程的Git仓库

配置读取规则

从官网中可以看到，可以通过以下五种形式的HTTP服务来访问远程Git的配置文件！

The HTTP service has resources in the following form:

```
{application}/{profile}[/{label}]\n{application}-{profile}.yml\n{label}/{application}-{profile}.yml\n{application}-{profile}.properties\n{label}/{application}-{profile}.properties
```

where `application` is injected as the `spring.config.name` in the `SpringApplication` (what is normally `application` in a regular Spring Boot app), `profile` is an active profile (or comma-separated list of properties), and `label` is an optional git label (defaults to `master`.)

我们主要掌握三种即可：

```
1  /{label}/{application}-{profile}.yml\n2  /{application}-{profile}.yml\n3  /{application}/{profile}[/{label}]
```

这里的 `label` 是Git中的分支名！

`{label}/{application}-{profile}.yml`

读取 `/ {label}/{application}-{profile}.yml` 这种格式的配置文件

1. master分支

如果配置文件在master分支上，则可以通过如下形式访问到远程git仓库中的配置文件：

- `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`

2. dev分支

如果配置文件在master分支上，则可以通过如下形式访问到远程git仓库中的配置文件：

- `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`

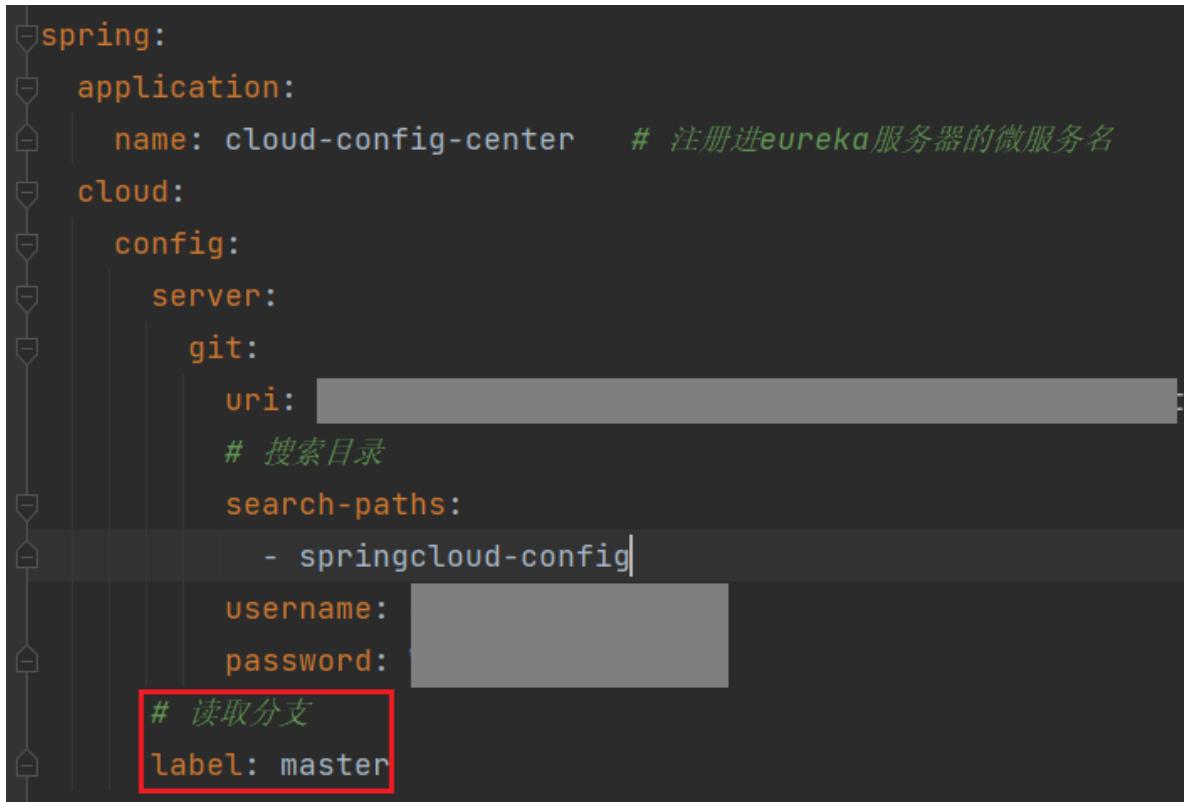
读取 `/ {application}-{profile}.yml` 这种格式的配置文件

如果读取以下形式的配置文件：

- `http://config-3344.com:3344/config-dev.yml`
- `http://config-3344.com:3344/config-test.yml`
- `http://config-3344.com:3344/config-prod.yml`

那么默认会读取 `application.yml` 中配置的分支名下的文件！

目前 Config Server 中默认配置是读取master分支!



```
spring:
  application:
    name: cloud-config-center # 注册进eureka服务器的微服务名
  cloud:
    config:
      server:
        git:
          uri: [REDACTED]
          # 搜索目录
          search-paths:
            - springcloud-config
          username: [REDACTED]
          password: [REDACTED]
          # 读取分支
        label: master
```

{application}/{profile}[/{label}]

可以通过以下形式访问配置文件:

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

可以发现访问后是json字符串的形式!



config-3344.com:3344/config/dev/master

```
{
  "name": "config",
  - profiles: [
    "dev"
  ],
  "label": "master",
  "version": "a7ac659fb6a2f25fbe",
  "state": null,
  - propertySources: [
    {
      "name": "https://gitee.com/springcloud-config.git/config-dev.yml",
      - source: {
        "config.info": "master branch, springcloud-config/config-dev.yml version=1"
      }
    }
  ]
}
```

小结

读取配置文件的规则时：

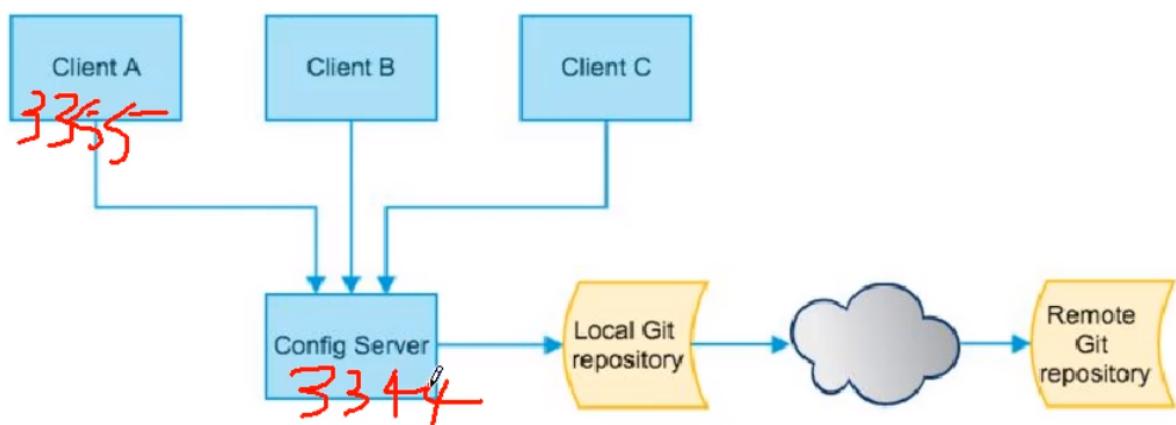
label：表示分支（branch）

application：表示服务名

profile：表示环境（dev/test/prod）

Config客户端配置与测试

在这一节，我们需要创建一个3355模块作为客户端，用客户端去访问Config Server，获取服务端配置。



新建模块

新建 `cloud-config-client-3355` 作为Config的客户端

改POM

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.cloud</groupId>
4         <artifactId>spring-cloud-starter-config</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.boot</groupId>
12        <artifactId>spring-boot-starter-web</artifactId>
13    </dependency>
14    <dependency>
15        <groupId>org.springframework.boot</groupId>
16        <artifactId>spring-boot-starter-actuator</artifactId>
17    </dependency>
```

```

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

```

可以看到它导入的config包，和服务端略有不同！

```

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>

```

bootstrap.yml

注意：这里不是新建 `application.yml`，而是需要学习新建一个新的配置文件 `bootstrap.yml`！

`bootstrap.yml` 是什么：

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

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，`Bootstrap Context` 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`。

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`

创建一个 `bootstrap.yml` 文件，内容如下：

```
1 server:
2   port: 3355
3 spring:
4   application:
5     name: config-client
6   cloud:
7     # Config 客户端配置
8     config:
9       label: master # 分支名称
10      name: config # 配置文件名称
11      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
12      uri: http://localhost:3344 # 配置中心地址
13 eureka:
14   client:
15     service-url:
16       defaultZone: http://localhost:7001/eureka
17
```

主启动类

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class ConfigClientMain3355 {
4   public static void main(String[] args) {
5     SpringApplication.run(ConfigClientMain3355.class, args);
6   }
7 }
```

业务类

```
1 @RestController
2 public class ConfigClientController {
3   @Value("${config.info}")
4   private String configInfo;
5
6   @GetMapping("/configInfo")
7   public String getConfigInfo() {
8     return configInfo;
9   }
10 }
```

这里的 `config.info` 是远程Git配置文件中的配置！

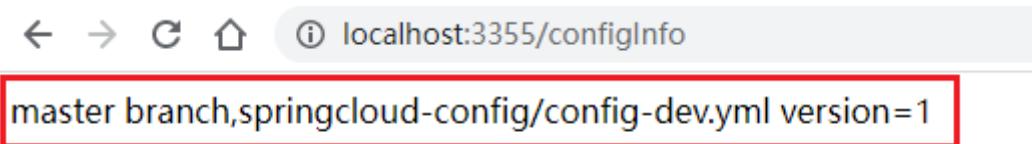
) config-dev.yml 76 Bytes
jrh 提交于 2小时前 . init commit

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

这里的业务类是用于读取远程配置文件中的 config.info 的值!

测试

1. 启动7001eureka
2. 启动Config配置中心3344微服务并自测
3. 启动3355作为Client, 准备访问: <http://localhost:3355/configInfo>



可以访问远程仓库的配置文件!

分布式配置的动态刷新问题

虽然上述测试成功了，但是目前分布式配置还有以下问题。下面我们进行演示。

进行如下操作:

1. 现在假设运维修改了Gitee上的配置文件

比如：我们将远程仓库上的config-dev.yml配置文件修改！

```
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)  
$ cat config-dev.yml  
config:  
  info: "master branch,springcloud-config/config-dev.yml version=1"  
  
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)  
$ vim config-dev.yml  
  
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)  
$ cat config-dev.yml  
config:  
  info: "master branch,springcloud-config/config-dev.yml version=2"  
  
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)  
$ |
```

可以看到Gitee上的文件产生变动

config-dev.yml 76 Bytes

hugh98 提交于 刚刚 . second commit

```

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

```

按照道理，gitee上的文件发生了变动，那么Config Server配置中心和客户端都应该产生相应的变动

下面我们看一下实际情况。

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

发现Config Server配置中心立刻响应

```

config:
info: master branch, springcloud-config/config-dev.yml version=2

```

3. 访问：<http://localhost:3355/configInfo>

发现Config Client客户端没有做出相应调整！！！

```

master branch,springcloud-config/config-dev.yml version=1

```

4. 重启3355微服务，再次访问：<http://localhost:3355/configInfo>

此时发现，当Config Client 客户端重启后，才能产生正确的变化。

```

master branch,springcloud-config/config-dev.yml version=2

```

以上就是问题所在！！！

可以看到3355没有变化，除非自己重启或者重新加载。

但是，难道每次修改配置文件后，客户端都需要重启？？？

解决访问请看下一节！

Config客户端之动态刷新

为了避免每次更新配置都要重启 Config Client 客户端微服务，所以这里需要讲解解决方法——动态刷新。

以下是实现步骤。主要是针对Config Client客户端3355模块进行修改。

pom引入actuator监控

首先需要引入 `actuator` 包

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

修改YML，暴露监控端口

在 `bootstrap.yml` 中加入如下内容：

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

修改业务类Controller

在 `ConfigClientController` 类中，加上 `@RefreshScope` 注解。

```
1 @RestController
2 @RefreshScope
3 public class ConfigClientController {
4   @Value("${config.info}")
5   private String configInfo;
6
7   @GetMapping("/configInfo")
8   public String getConfigInfo() {
9     return configInfo;
10  }
11 }
```

SpringCloud 使用 `@RefreshScope` 注解，实现配置文件的动态加载！

`@RefreshScope` 注解仅需加在所有引入外部配置文件的类上！

比如当前这个controller类中就使用了@Value引入外部配置文件中的内容

测试1

此时再次进行测试。

1. 启动7001
2. 启动3344配置中心
3. 启动3355 Config客户端

4. 对远程仓库中的config-dev.yml进行如下修改

```
H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)
$ cat config-dev.yml
config:
  info: "master branch,springcloud-config/config-dev.yml version=2"

H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)
$ vim config-dev.yml

H@DESKTOP-7CK6BAS MINGW64 /d/java_study/cloud2020/springcloud-config (master)
$ cat config-dev.yml
config:
  info: "master branch,springcloud-config/config-dev.yml version=3"
```

master ▾ springcloud-config / config-dev.yml

config-dev.yml 76 Bytes

hugh98 提交于刚刚 · third commit

```
1 config:
2   info: "master branch,springcloud-config/config-dev.yml version=3"
```

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

config:

```
info: master branch,springcloud-config/config-dev.yml version=3
```

6. 重启3355后, 访问: <http://localhost:3355/configInfo>

localhost:3355/configInfo

```
master branch,springcloud-config/config-dev.yml version=3
```

7. 再次修改该配置文件, 让version=4

8. 再访问: <http://config-3344.com:3344/master/config-dev.yml>

config:

```
info: master branch,springcloud-config/config-dev.yml version=4
```

9. 再访问: <http://localhost:3355/configInfo>

localhost:3355/configInfo

```
master branch,springcloud-config/config-dev.yml version=3
```

发现还是3! 说明配置没有生效? ! !

那么如何让配置生效?

让配置生效

从上述测试可以发现，我们的配置并没有生效。

这是因为，**需要运维人员发送Post请求刷新Config Client**。即，我们这里需要发送post请求刷新3355微服务。

可以利用curl发送如下命令：

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

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

测试2

发送POST请求后，我们进行下面测试。

现在三个微服务（7001、3344、3355）都在运行中，不要重启服务。

1. 再次访问：<http://localhost:3355/configInfo>

可以获取远程仓库中的最新配置



总结：

1. 当远程仓库中的配置文件改动后，需要人发送POST请求刷新Config Client，才能让Config Client获取远程仓库中的最新配置文件。

存在的问题

目前还可能存在的问题：

- 假如有多个微服务客户端，那么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？想大范围的自动刷新，有什么方法？这就是消息总线的作用！请看下一章！

十二、 SpringCloud Bus消息总线

官网最新版本：<https://cloud.spring.io/spring-cloud-bus/reference/html/>

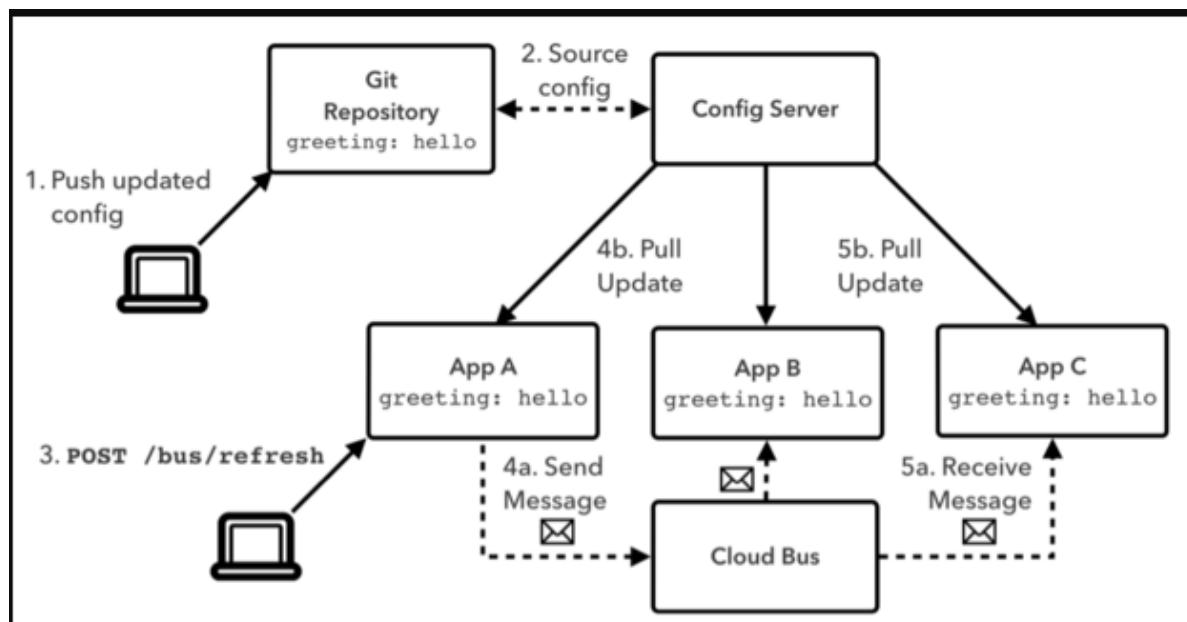
概述

针对前面SpringCloud Config分布式配置中心的缺点，利用消息总线就可以实现分布式自动刷新配置功能。

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

什么是Bus

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。

- 1. Quick Start
- 2. Bus Endpoints
- 3. Addressing an Instance
- 4. Addressing All Instances of a Service**
- 5. Service ID Must Be Unique
- 6. Customizing the Message Broker
- 7. Tracing Bus Events
- 8. Broadcasting Your Own Events
- 9. Configuration properties

Spring Cloud Bus

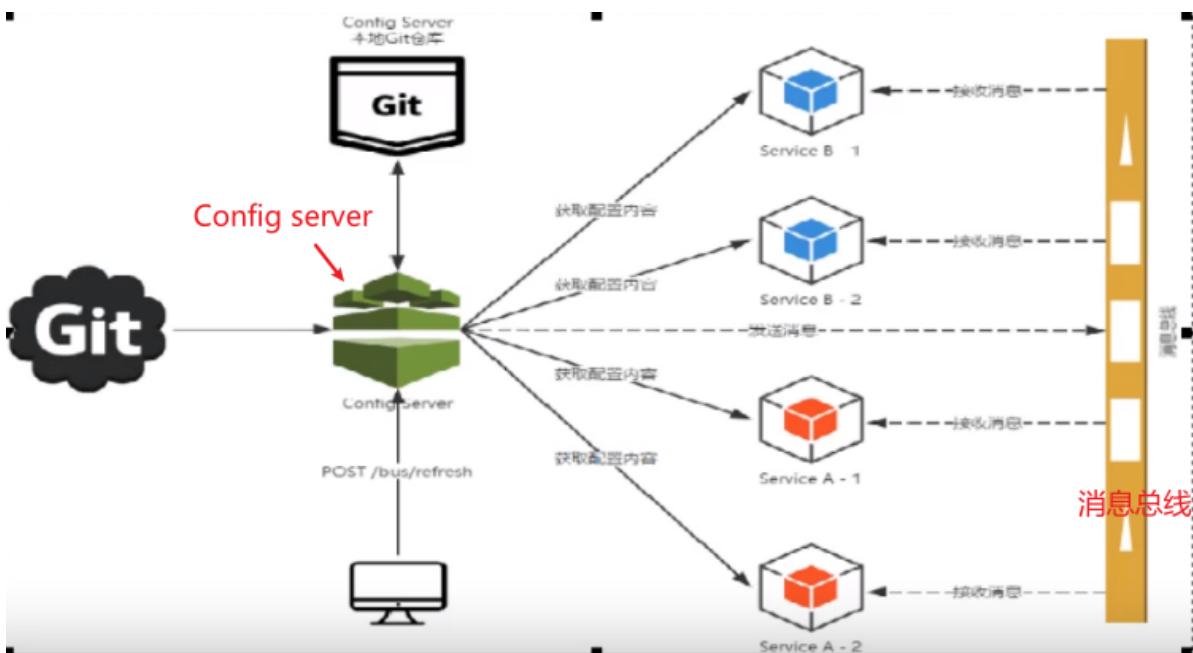
Spring Cloud Bus links the nodes of a distributed system with a lightweight message broker. This broker can then be used to broadcast state changes (such as configuration changes) or other management instructions. A key idea is that the bus is like a distributed actuator for a Spring Boot application that is scaled out. However, it can also be used as a communication channel between apps. This project provides starters for either an AMQP broker or Kafka as the transport.



Spring Cloud is released under the non-restrictive Apache 2.0 license. If you would like to contribute to this section of the documentation or if you find an error, please find the source code and issue trackers in the project at [docslink](#)[github].

Bus能干什么

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。



为什么被称为总线

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springCloudBus)。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。

RabbitMQ环境配置

这里需要学习RabbitMQ，并且安装一个。

请自行参考RabbitMQ安装教程。

这里我在虚拟机（IP: 192.168.179.130）上已安装好了。

并且在 `rabbitmq_server-3.8.8/sbin` 目录下启动管理功能：

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

```
[root@node1 sbin]# rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@node1:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_delayed_message_exchange
  rabbitmq_federation
  rabbitmq_federation_management
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@node1...
Plugin configuration unchanged.
[root@node1 sbin]#
```

安装完成后，可以从浏览器中登录：<http://192.168.179.130:15672>

 不安全 | 192.168.179.130:15672



Username: *

Password: *

Login

输入自己设置的账号密码登录

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@node1	36 32768 available	0 29401 available	444 1048576 available	89 MiB 728 MiB high watermark & 4 MiB low watermark	14 GiB	1d 11h	basic disc 4 rss	This node All nodes	

Churn statistics

Ports and contexts

Export definitions

Import definitions

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

SpringCloud Bus动态刷新全局广播

首先，必须具备RabbitMQ.

其次，为了演示广播效果，我们以3355模块为模板 再制造一个3366模块，增加复杂度。

制造3366模块

新建一个模块

新建一个Module，名为 `cloud-config-client3366`

改POM

```

1  <dependencies>
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-config</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.boot</groupId>
12         <artifactId>spring-boot-starter-web</artifactId>
13     </dependency>
14     <dependency>
15         <groupId>org.springframework.boot</groupId>
16         <artifactId>spring-boot-starter-actuator</artifactId>
17     </dependency>
18 
```

```
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>
```

写YML

注意：这里配置文件名为 `bootstrap.yml`

```
1  server:
2      port: 3366
3  spring:
4      application:
5          name: config-client
6      cloud:
7          # Config 客户端配置
8          config:
9              label: master # 分支名称
10             name: config # 配置文件名称
11             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
12             uri: http://localhost:3344 # 配置中心地址
13  eureka:
14      client:
15          service-url:
16              defaultZone: http://localhost:7001/eureka
17  # 暴露监控端点
18  management:
19      endpoints:
20          web:
21              exposure:
22                  include: "*"
```

主启动类

```
1  @SpringBootApplication
2  @EnableEurekaClient
3  public class ConfigClientMain3366 {
4      public static void main(String[] args) {
5          SpringApplication.run(ConfigClientMain3366.class, args);
6      }
7  }
```

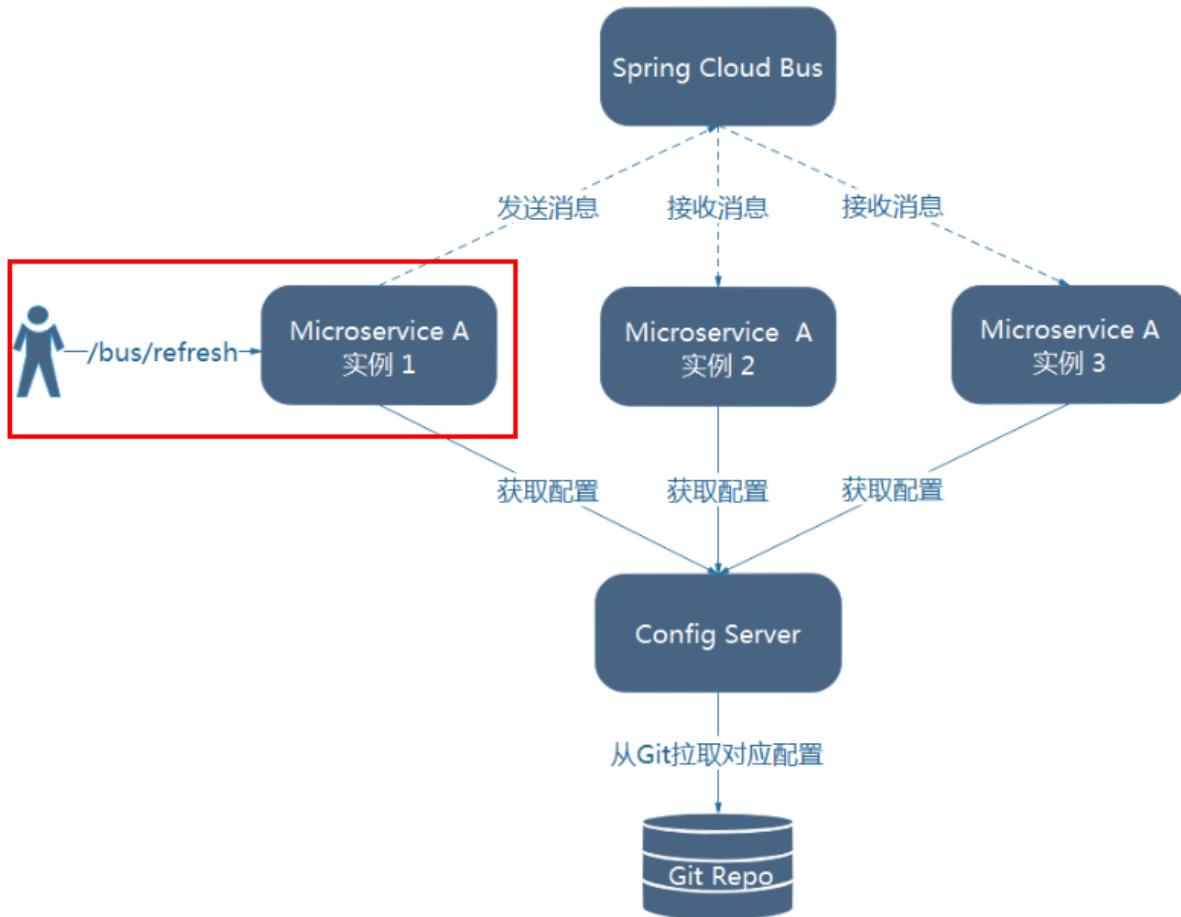
Controller

```
1  @RestController
2  @RefreshScope
3  public class ConfigClientController {
4      @Value("${config.info}")
5      private String configInfo;
6
7      @GetMapping("/configInfo")
8      public String getConfigInfo() {
9          return configInfo;
10     }
11 }
```

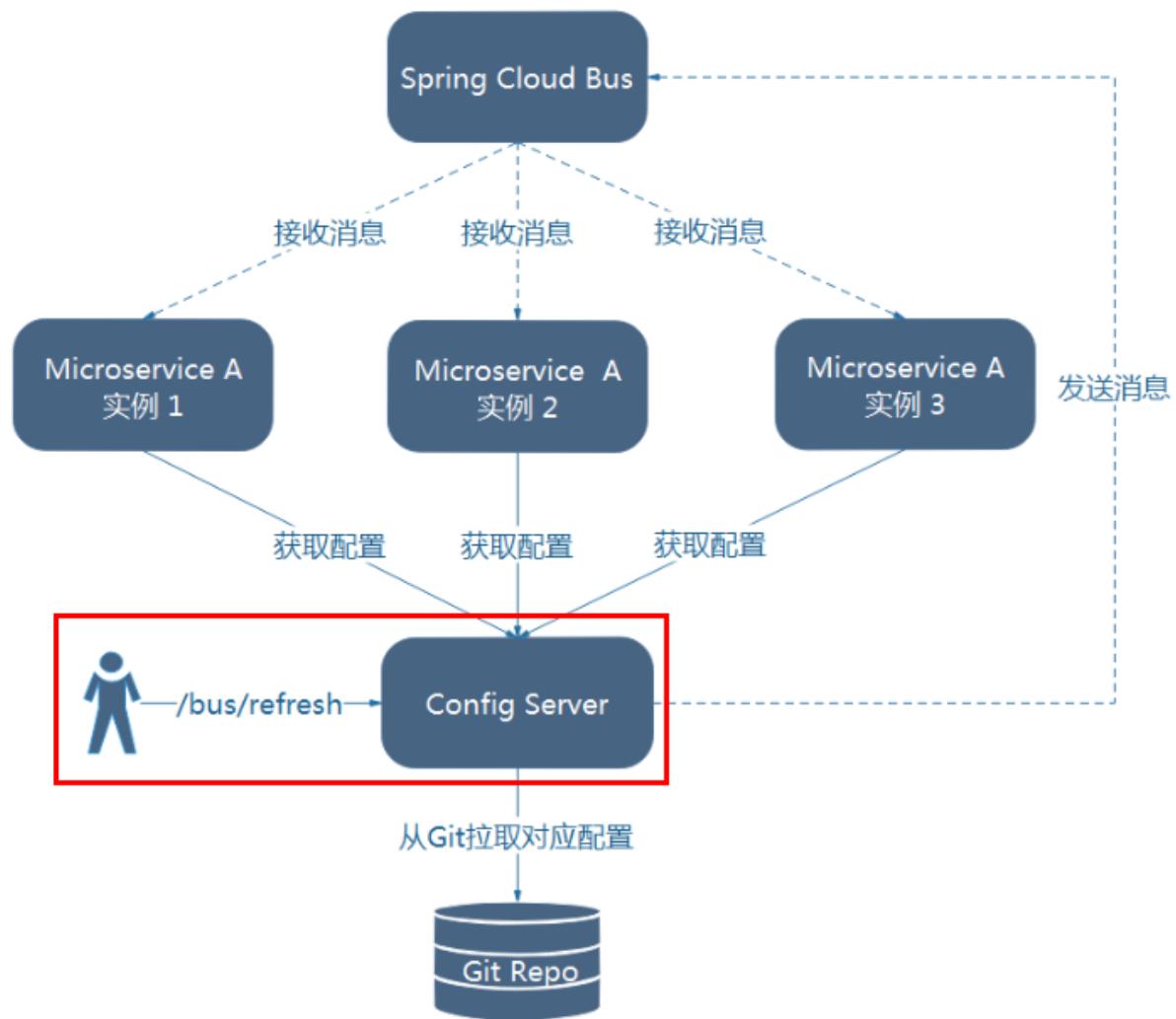
设计思想

广播消息通知主要有两种设计思想：

- 利用消息总线触发一个客户端/bus/refresh，然后刷新所有客户端的配置。



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



第二个设计思想的架构显然更加合适，第一个设计思想不合适的原因如下：

1. **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。
2. **破坏了微服务各节点的对等性。**
3. **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

因此，后面我们采用第二个设计思想！

给服务端添加消息总线支持

给我们的Config Server服务端 `cloud-config-center-3344` 添加消息总线支持

修改POM

在pom.xml中引入新包：

```
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>
```

修改YML

在原来的基础上，给 `application.yml` 中添加以下配置：

```
1 spring:
2   # rabbitmq相关配置
3   rabbitmq:
4     host: 192.168.179.130
5     port: 5672
6     username: admin
7     password: 123
8
9   # rabbitmq相关配置，暴露bus刷新配置的端点
10  management:
11    endpoints:
12      web:
13        exposure:
14          include: 'bus-refresh'
```

给客户端添加消息总线支持

给Config Client客户端（3355、3366）添加消息总线支持

这里以3355为例。其实3366步骤也差不多

修改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>
```

修改YML

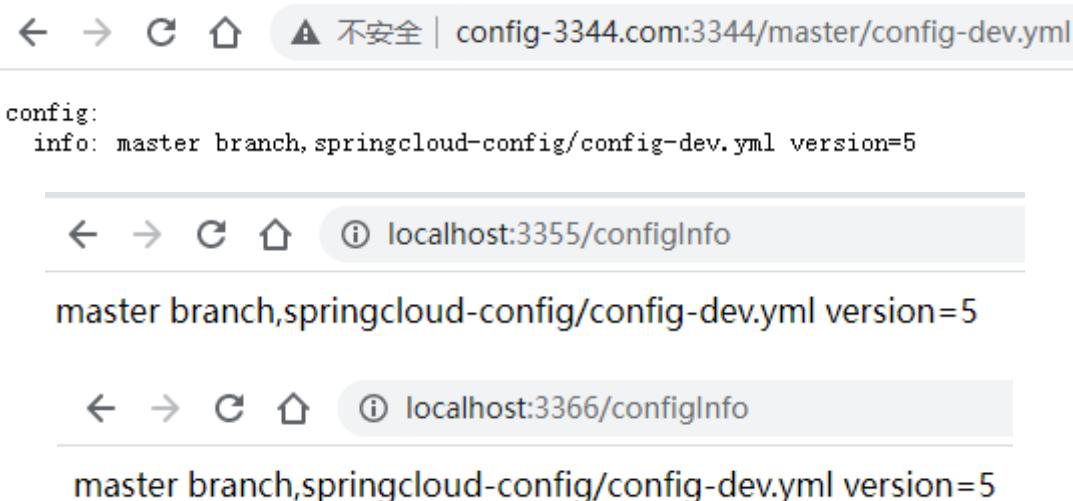
在原来的基础上，给 `bootstrap.yml` 中添加以下配置：

```
1  spring:
2      # rabbit相关配置，15672是web管理界面的端口，5672是MQ的访问端口
3      rabbitmq:
4          host: 192.168.179.130
5          port: 5672
6          username: admin
7          password: 123
8      # 暴露监控端点
9      management:
10         endpoints:
11             web:
12                 exposure:
13                     include: "*"      # 'refresh'
```

测试

1. 启动7001eureka注册中心
2. 启动3344 Config Server注册中心服务端
3. 启动3355和3366 Config Client
4. 分别访问：<http://config-3344.com:3344/master/config-dev.yml>，
<http://localhost:3355/configInfo>，<http://localhost:3366/configInfo>。这三个网址

发现其中的版本号都是version=5

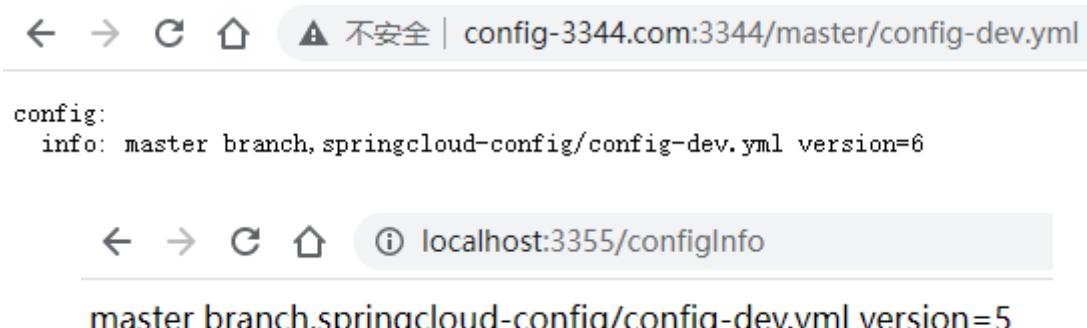


The image contains three screenshots of a browser window. Each screenshot shows a different endpoint's configuration response:

- The top screenshot shows the response for <http://config-3344.com:3344/master/config-dev.yml>. The content is: config: info: master branch, springcloud-config/config-dev.yml version=5
- The middle screenshot shows the response for <http://localhost:3355/configInfo>. The content is: master branch,springcloud-config/config-dev.yml version=5
- The bottom screenshot shows the response for <http://localhost:3366/configInfo>. The content is: master branch,springcloud-config/config-dev.yml version=5

5. 修改Gitee远程仓库中的该文件，让其version=6
6. 再次访问：<http://config-3344.com:3344/master/config-dev.yml>，
<http://localhost:3355/configInfo>，<http://localhost:3366/configInfo>

可以发现，只有3344注册中心的能获取到最新文件。这是因为我们还没利用POST请求来刷新配置



The image contains two screenshots of a browser window. Both screenshots show the same configuration response for the 3344 endpoint:

- The top screenshot shows the response for <http://config-3344.com:3344/master/config-dev.yml>. The content is: config: info: master branch, springcloud-config/config-dev.yml version=6
- The bottom screenshot shows the response for <http://localhost:3355/configInfo>. The content is: master branch,springcloud-config/config-dev.yml version=5

master branch,springcloud-config/config-dev.yml version=5

7. 给Config Server发送post请求刷新配置

```
1 curl -X POST "http://localhost:3344/actuator/bus-refresh"
```

- 再次访问: <http://config-3344.com:3344/master/config-dev.yml> ,
<http://localhost:3355/configInfo> , <http://localhost:3366/configInfo>

可以发现，三个微服务都可以访问到最新文件version=6

```
config:  
info: master branch, springcloud-config/config-dev.yml version=6
```

master branch,springcloud-config/config-dev.yml version=6

master branch,springcloud-config/config-dev.yml version=6

- 还可以通过浏览器观察RabbitMQ，可以发现MQ中新增了一个名为 `springCloudBus` 的交换机和三个Queues



Exchanges

▼ All exchanges (8)

Pagination

Page of 1 - Filter: Regex ?

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	

Queues

▼ All queues (3)

Pagination

Page of 1 - Filter: Regex ?

Displaying 3 items , page size

Name	Type	Features	State	Messages			Message rates		
				Ready	Unacked	Total	incoming	deliver / get	ack
springCloudBus.anonymous.Moz9F2QhRgCYP_izVqC-6w	classic	AD Excl ML	idle	0	0	0	0.00/s	0.00/s	0.00/s
springCloudBus.anonymous.OUCG8ZVXQDSNUKFErSxRhw	classic	AD Excl ML	idle	0	0	0	0.00/s	0.00/s	0.00/s
springCloudBus.anonymous.kh9okQfDTWeZwmHrv04CSQ	classic	AD Excl ML	idle	0	0	0	0.00/s	0.00/s	0.00/s

可以看到，一次发送，处处生效

SpringCloud Bus动态刷新定点通知

如果我们不想通过广播通知所有的Config Client，而是想顶点通知。

比如，我们只想通知3355，不通知3366。

那么就需要了解 SpringCloud Bus动态刷新定点通知

定点通知做法

注意： 实现定点通知的配置 和 全局广播的配置相同，只是发送POST请求那里不同！

全局广播时，发送的POST请求是：

```
1 curl -X POST "http://配置中心的IP+端口号/actuator/bus-refresh"
```

而，定点通知时，发送POST请求是：

```
1 curl -X POST "http://配置中心的IP+端口号/actuator/bus-refresh/{destination}"
```

这样，/bus/refresh请求不再发送到具体的服务实例上，而是发给config server 并通过 `destination` 参数类指定需要更新配置的服务或实例。

这里：`destination` 参数 填的是 指定微服务的服务名+端口号。 (即，`spring.application.name + server.port`)

测试案例

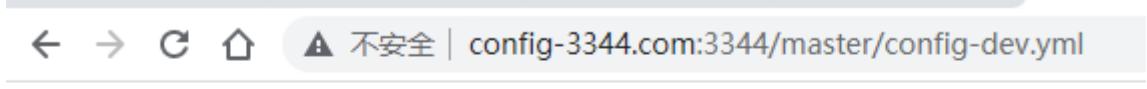
这里以刷新运行在3355端口上的config-client为例，只通知3355，不通知3366。

1. 保持7001、3344、3355、3366处于运行状态。目前都能获取到的配置文件version=6
2. 修改Gitee仓库文件version=7
3. 发送以下POST请求

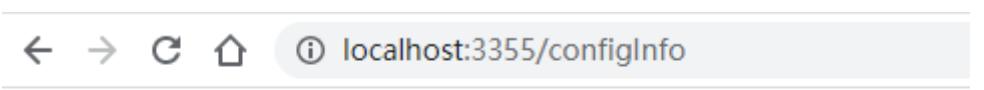
```
1 curl -X POST "http://localhost:3344/actuator/bus-refresh/config-client:3355"
```

4. 分别访问：<http://config-3344.com:3344/master/config-dev.yml>，
<http://localhost:3355/configInfo>，<http://localhost:3366/configInfo>

可以发现，只有3344和3355获得了最新的配置文件，3366还是旧文件。



```
config:  
  info: master branch, springcloud-config/config-dev.yml version=7
```



```
master branch,springcloud-config/config-dev.yml version=7
```

master branch,springcloud-config/config-dev.yml version=6

十三、SpringCloud Stream消息驱动

为什么引入cloud stream?

说到消息，首先会想到 MQ 消息中间件。常见的 MQ 只有四种：[ActiveMQ](#)、[RabbitMQ](#)、[RocketMQ](#)、[Kafka](#)。如果说，我们要掌握四种 MQ，那所花费的时间肯定很长。而如果我们只学了某种 MQ，但是在企业中用的是另一种 MQ，那么怎么办？这是第一个问题。

第二个问题，现在企业的系统，可能存在一个系统中含有多个不同的 MQ 消息中间件，这样就会存在以下困难：1. 切换。2. 维护。3. 开发

那么就会想：有没有一种新技术的诞生，让我们不再关注具体 MQ 的细节，让我们只需用一种适配绑定的方式，自动的给我们在各种 MQ 内切换？

这就是引入 cloud stream 的原因！

它解决的痛点是什么？

cloud stream 让我们不再关注具体 MQ 的细节，让我们只需用一种适配绑定的方式，自动的给我们 在各种 MQ 内切换。

即，可能在一个系统中四种 MQ 都有，但是利用 cloud stream 就可以屏蔽底层的细节插件，这样操作一个 cloud stream 就可以操作底层各种各样不同的 MQ

消息驱动概述

什么是SpringCloud 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。

一句话：SpringCloud Stream 能够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。

官网：<https://spring.io/projects/spring-cloud-stream>

<https://docs.spring.io/spring-cloud-stream/docs/current/reference/html/>

Spring Cloud Stream是用于构建与共享消息传递系统连接的高度可伸缩的事件驱动微服务框架，该框架提供了一个灵活的编程模型，它建立在已经建立和熟悉的Spring术语和最佳实践上，包括支持持久化的发布/订阅、消费组以及消息分区这三个核心概念。

Spring Cloud Stream

3.2.3

OVERVIEW LEARN SUPPORT SAMPLES

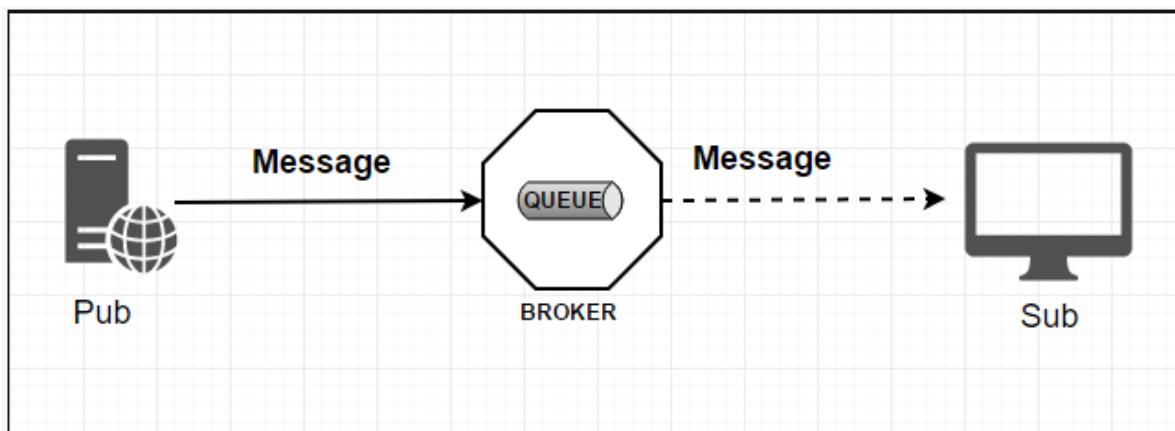
Spring Cloud Stream is a framework for building highly scalable event-driven microservices connected with shared messaging systems.

The framework provides a flexible programming model built on already established and familiar Spring idioms and best practices, including support for persistent pub/sub semantics, consumer groups, and stateful partitions.

设计思想

标准MQ

首先，标准MQ的架构如下：

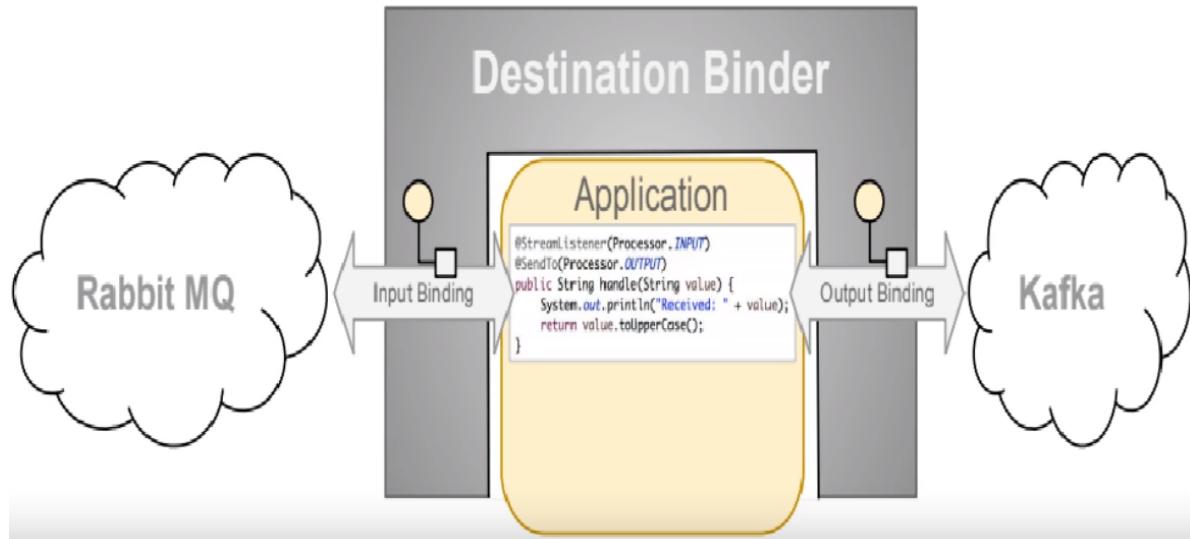


在没有引入SpringCloud Stream的时候：

- 生产者和消费者之间靠**消息（Message）**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？谁负责收，谁负责发？
 - 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springcloud Stream给我们提供了一种解耦合的方式**。

stream为什么可以实现统一底层差异？

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。

即，**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**。

[Back to index](#)

[Reference Guide](#)

[1. Usage](#)

[2. RabbitMQ Binder Overview](#)

[3. Configuration Options](#)

[4. Using Existing Queues/Exchanges](#)

[5. Retry With the RabbitMQ Binder](#)

[6. Error Channels](#)

[7. Dead-Letter Queue Processing](#)

[8. Partitioning with the RabbitMQ Binder](#)

[Appendices](#)

2. RabbitMQ Binder Overview

The following simplified diagram shows how the RabbitMQ binder operates:



Figure 1. RabbitMQ Binder

By default, the RabbitMQ Binder implementation maps each destination to a `TopicExchange`. For each consumer group, a `Queue` is bound to that `TopicExchange`. Each consumer instance has a corresponding RabbitMQ `Consumer` instance for its group's `Queue`. For partitioned producers and consumers, the queues are suffixed with the partition index and use the partition index as the routing key. For anonymous consumers (those with no `group` property), an auto-delete queue (with a randomized unique name) is used.

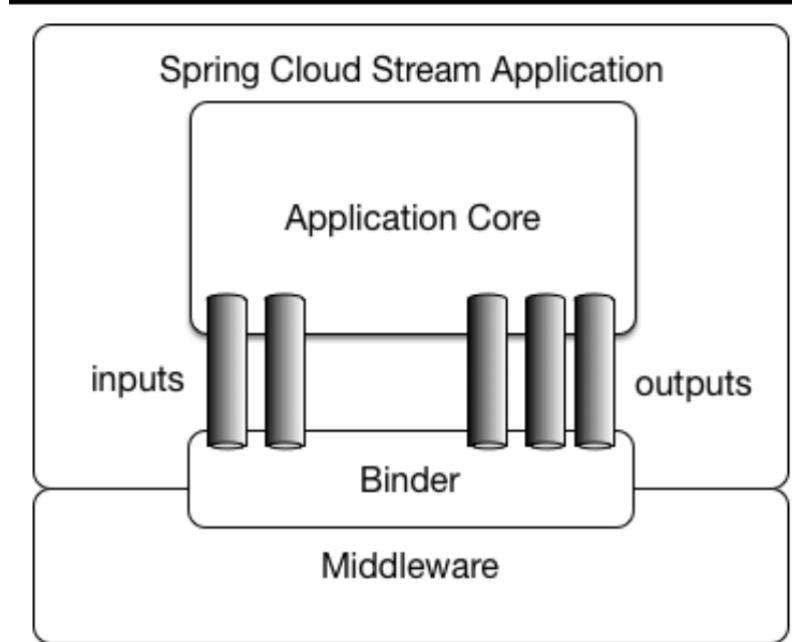
By using the optional `autoBindDlq` option, you can configure the binder to create and configure dead-letter queues (DLQs) (and a dead-letter exchange `DLX`, as well as routing infrastructure). By default, the dead letter queue has the name of the destination, appended with `.dlq`. If retry is enabled (`maxAttempts > 1`), failed messages are delivered to the DLQ after retries are exhausted. If retry is disabled (`maxAttempts = 1`), you should set `requeueRejected` to `false` (the default) so that failed messages are routed to the DLQ, instead of being re-queued. In addition, `republishToDlq` causes the binder to publish a failed message to the DLQ (instead of rejecting it). This feature lets additional information (such as the stack trace in the `x-exception-stacktrace` header) be added to the message in headers. See the `frameMaxHeadroom` property for information about truncated stack traces. This option does not need retry enabled. You can republish a failed message after just one attempt. Starting with version 1.2, you can configure the delivery mode of republished messages. See the `republishDeliveryMode` property.

If the stream listener throws an `ImmediateAcknowledgeAmqpException`, the DLQ is bypassed and the message simply discarded. Starting with version 2.1, this is true regardless of the setting of `republishToDlq`; previously it was only the case when `republishToDlq` was `false`.

什么是Binder?

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（Binder）**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。Stream对消息中间件的进一步封装，可以做到代码层面对中间件的无感知，甚至于动态的切换中间件(rabbitmq切换为kafka)，使得微服务开发的高度解耦，服务可以关注更多自己的业务流程



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可以生成Binding，Binding用来绑定消息容器的生产者和消费者，它有两种类型，INPUT和OUTPUT。

这个 input 与 output 是相对于应用程序来说的，input对于应用程序就是读（从外面到应用程序），output对于应用程序就是写（从应用程序写到外面）

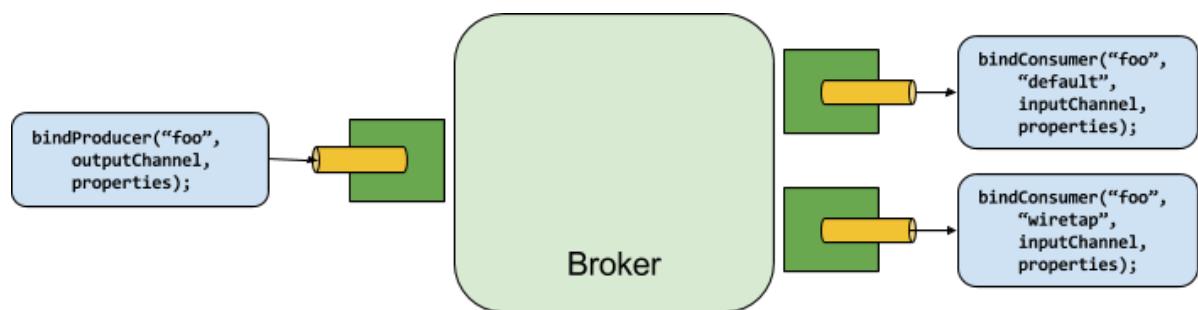
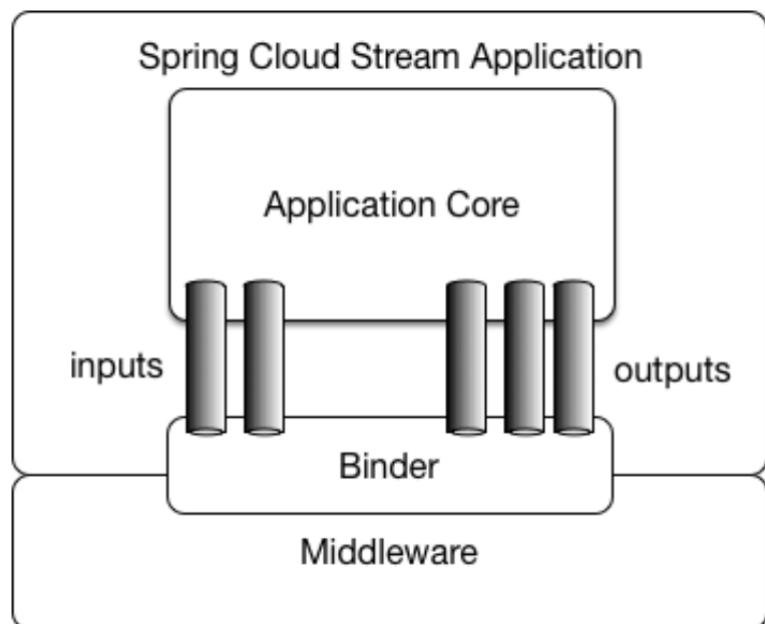
Stream中的消息通信方式遵循了发布-订阅模式

Stream中的消息通信方式遵循了发布-订阅模式，主要就是利用 Topic 主题进行广播。

这个 Topic 在 RabbitMQ 中就是 Exchange，在 Kafka 中就是 Topic。

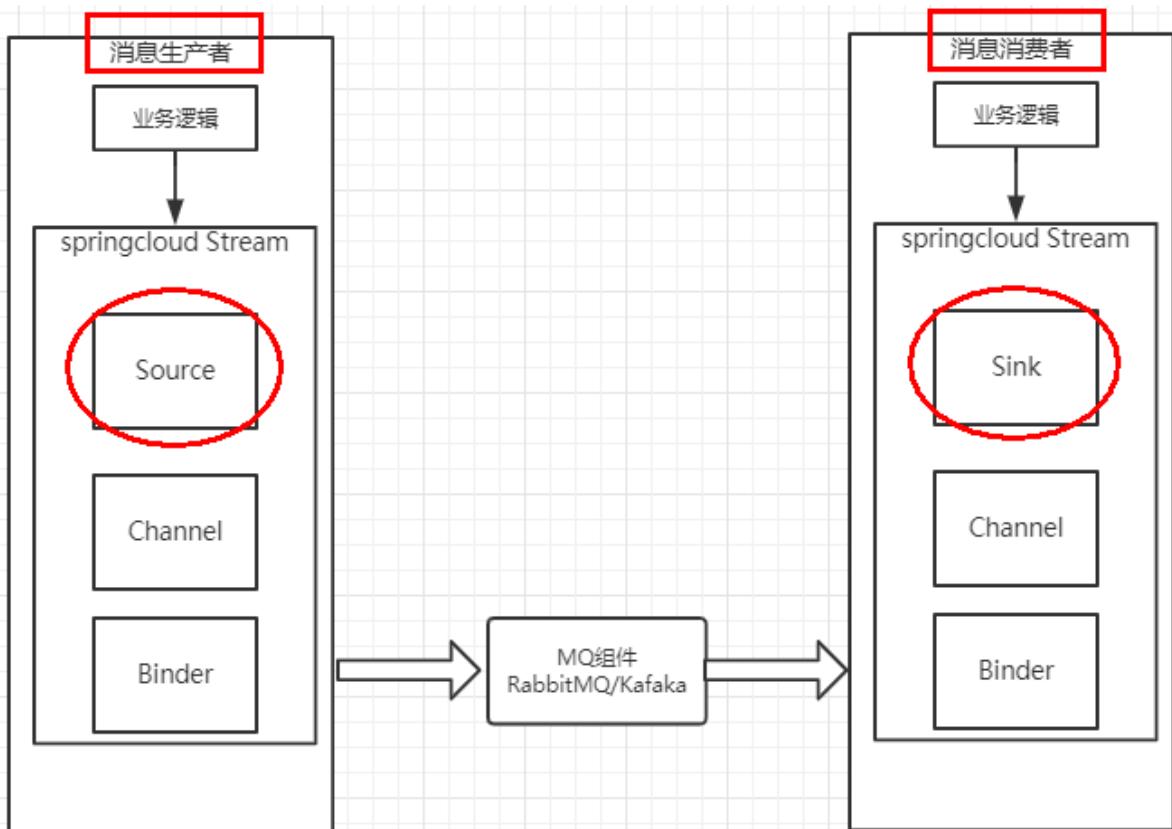
SpringCloud Stream标准流程套路

SpringCloud Stream官方架构图：



从上图其实可以看到，消息生产者其实跟 `outputChannel` (输出管道) 连接在一起，消息消费者其实跟 `inputChannel` (输入管道) 连接在一起。

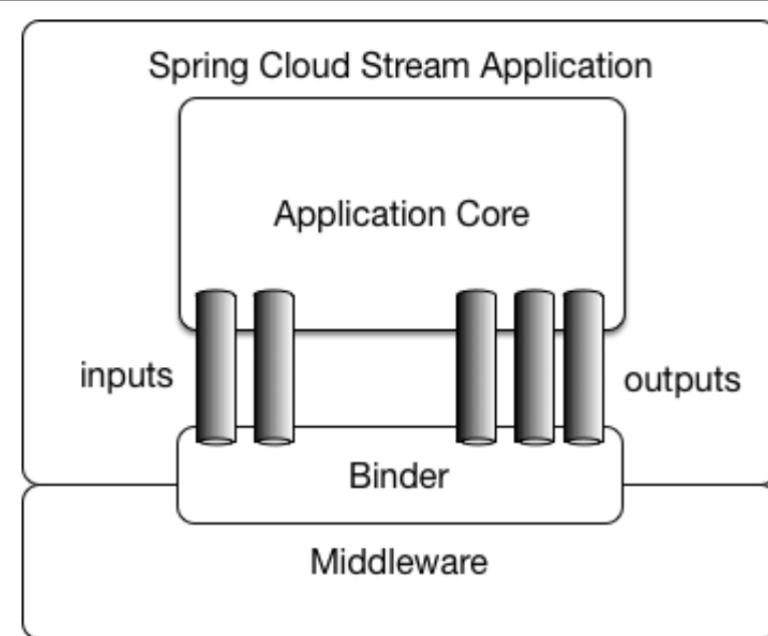
SpringCloud Stream大致模型如下：



从中可以看到SpringCloud Stream的三个重要的东西：

- **Binder**：它能方便的连接中间件，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，接收消息就是输入。**

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（对应于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，通过该输入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

首先，需要安装RabbitMQ。

然后，在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

新建一个模块，名为 `cloud-stream-rabbitmq-provider8801`

改POM

```

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

引入的包中，需要了解的新东西有：

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

表示用rabbit做中间件。如果用kafka做中间件，则用 kafka 替换 rabbit

写YML

```
1 server:
2   port: 8801
3
4 spring:
5   application:
6     name: cloud-stream-provider
7   rabbitmq:
8     host: 192.168.179.130
9     port: 5672
10    username: admin
11    password: 123
12  cloud:
13    stream:
14      binders: # 在此处配置要绑定的rabbitmq的服务信息
15        defaultRabbit: # 表示定义的名称，用于binding整合，自定义
16          type: rabbit # 消息组件类型
17        bindings: # 服务的整合处理
18          output: # 这个名字是一个通道的名称
19            destination: studyExchange # 表示要使用的Exchange名称，自定义
20            content-type: application/json # 设置消息类型，本次为json；文本可以设置
为"text/plain"
21          binder: defaultRabbit # 设置要绑定的消息服务的具体设置
22
23 eureka:
24   client:
25     service-url:
26       defaultZone: http://localhost:7001/eureka
27   instance:
28     lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔(默认30秒)
29     lease-expiration-duration-in-seconds: 5
30     instance-id: send-8801.com # 信息列表显示的主机名称
31     prefer-ip-address: true # 访问的路径变为IP地址
```

主启动类

```
1 @SpringBootApplication
2 public class StreamMQMain8801 {
3   public static void main(String[] args) {
4     SpringApplication.run(StreamMQMain8801.class, args);
5   }
6 }
```

业务类

1. 新建一个接口，作为发送消息接口

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

2. 新建一个类，作为发送消息接口的实现类

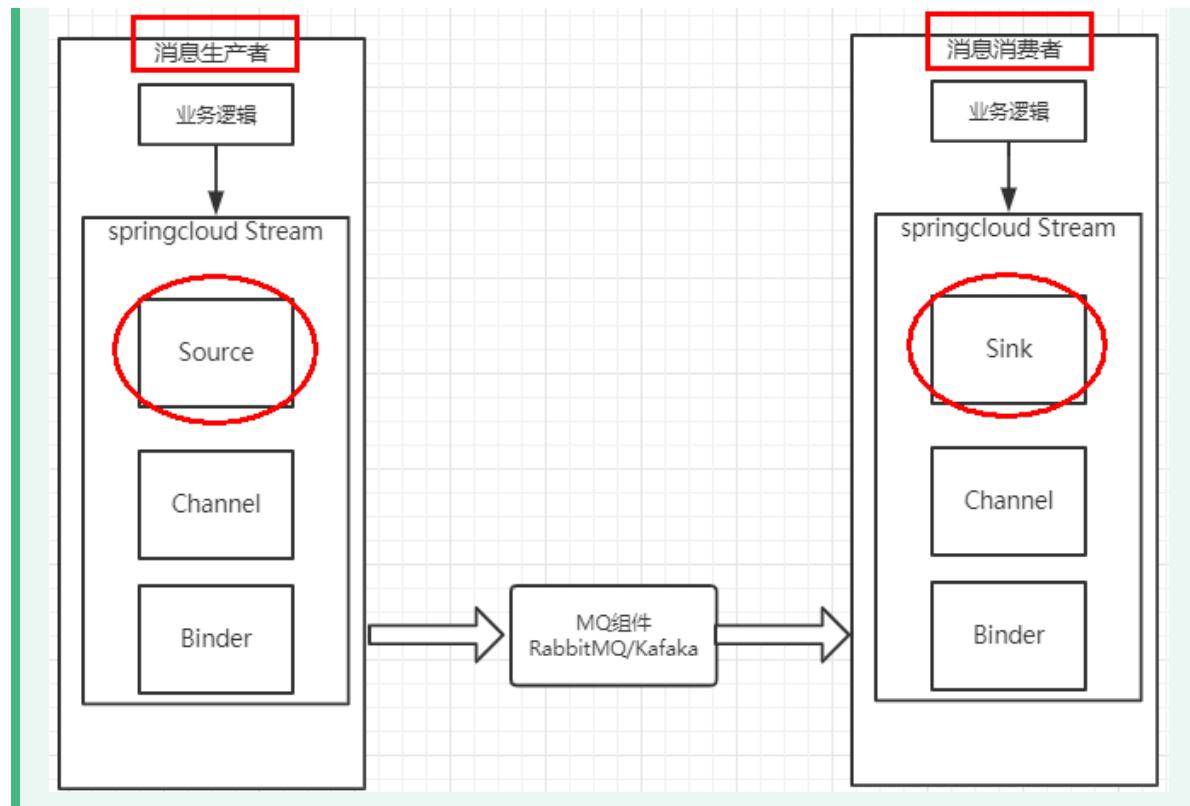
```
1 // 注意: import org.springframework.cloud.stream.messaging.Source  
2 @EnableBinding(Source.class)      // 定义消息的推送管道  
3 public class MessageProviderImpl implements IMessageProvider {  
4  
5     @Resource  
6     private MessageChannel output; // 消息发送管道  
7  
8     @Override  
9     public String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

注意：这个类上的注解不是 @Service，而是 @EnableBinding；并且service中不用DAO，因为这个微服务是用于跟消息中间件打交道的服务！

@EnableBinding：作用是让信道channel和exchange绑定在一起。

里面的 Source.class 对应下图中的 Source，因为这个服务是消息生产者。

(后面依次类推可知，在消息消费者中，需要注解 @EnableBinding(Sink.class))



在上述的注解 `@EnableBinding` 的源码中可以看到包含了 `@Configuration`。因此该类会生成对象并注入到IOC容器中。

```

@Target({ ElementType.TYPE, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@Import({ BindingBeansRegistrar.class, BinderFactoryAutoConfiguration.class })
@EnableIntegration
public @interface EnableBinding {

    /**
     * 一个接口列表，其中包含用输入和/或输出注释的方法，以指示绑定目标。
     * 返回：
     * 接口列表
     */
    Class<?>[] value() default {};
}

```

3. 新建一个controller类

```

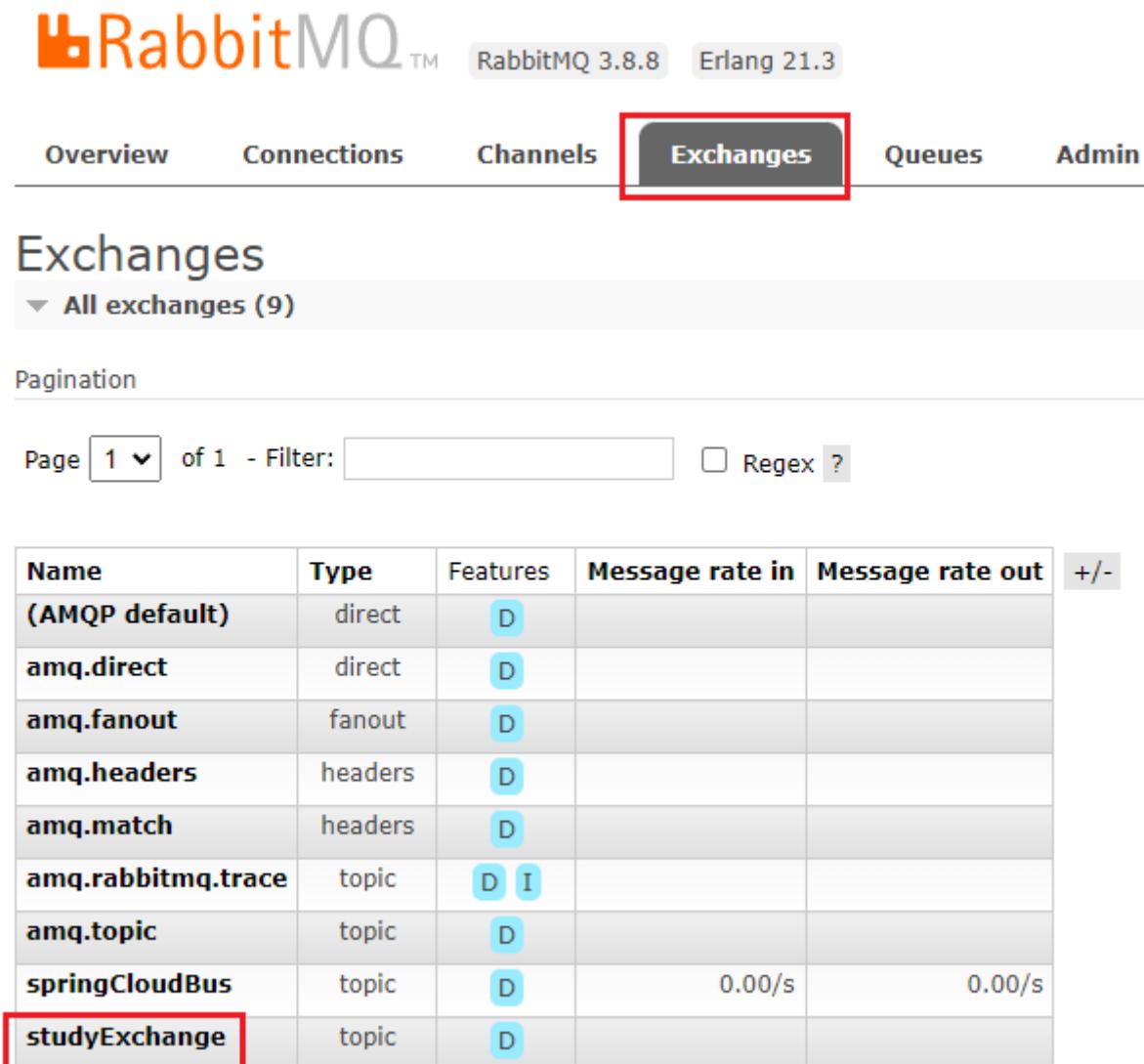
1  @RestController
2  public class SendMessageController {
3
4      @Resource
5      private IMessageProvider messageProvider;
6
7      @GetMapping(value = "/sendMessage")
8      public String sendMessage() {
9          return messageProvider.send();
10     }
11 }

```

测试

1. 启用7001eureka
2. 启动rabbitmq
3. 启动8801
4. 登录rabbitmq的管理页面: <http://192.168.179.130:15672>

可以看到，出现了一个 studyExchange 交换机。这是在8001微服务的application.yml中配置的。



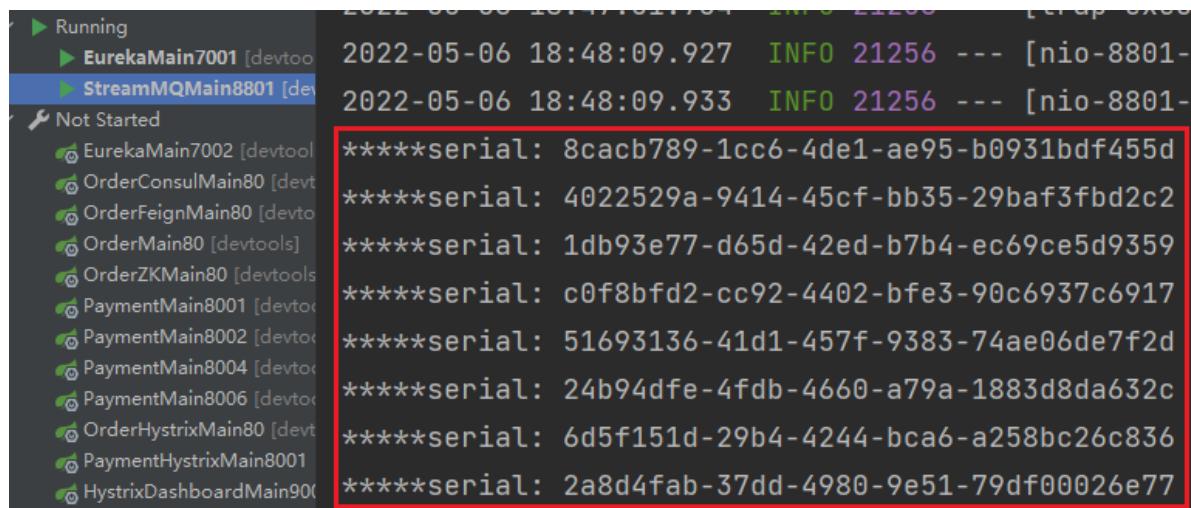
The screenshot shows the RabbitMQ Management Console interface. At the top, it displays the RabbitMQ logo and version information: RabbitMQ 3.8.8 Erlang 21.3. Below the header, there is a navigation bar with tabs: Overview, Connections, Channels, Exchanges (which is highlighted with a red box), Queues, and Admin. The main content area is titled "Exchanges" and shows a list of exchanges. A dropdown menu "All exchanges (9)" is open. Below the list, there are pagination controls ("Page 1 of 1") and a filter input field. The table lists the following exchanges:

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			

5. 多次访问地址: <http://localhost:8801/sendMessage>

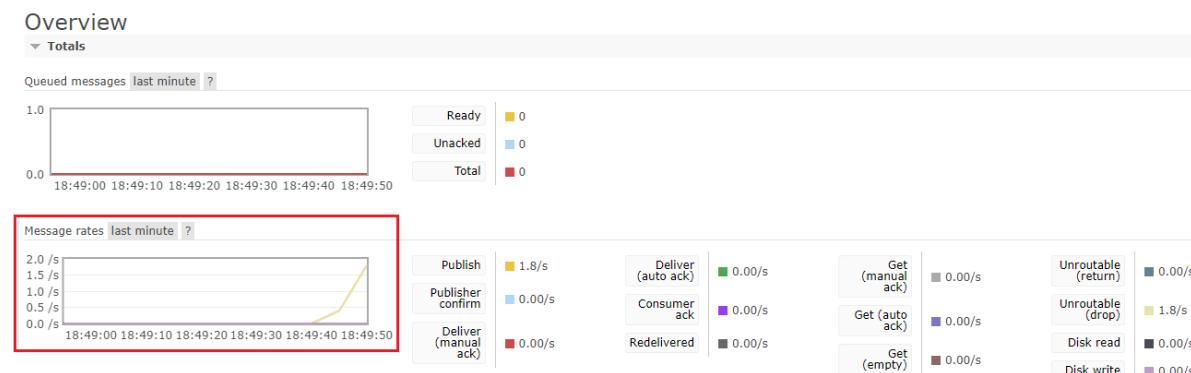
2a8d4fab-37dd-4980-9e51-79df00026e77

后台会打印生成的UUID



6. 在多次发送消息后，观察mq

可以看到mq中有消息送到。



消息驱动之消费者

新建Module

新建一个模块，名为 `cloud-stream-rabbitmq-consumer8802`

改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>
```

写YML

```

1  server:
2      port: 8802
3
4  spring:
5      application:
6          name: cloud-stream-consumer
7
8  rabbitmq:
9      host: 192.168.179.130
10     port: 5672
11     username: admin
12     password: 123
13
14  cloud:
15      stream:
16          binders:      # 在此处配置要绑定的rabbitmq的服务信息
17              defaultRabbit:    # 表示定义的名称，用于binding整合，自定义
18                  type: rabbit    # 消息组件类型
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，自定义
22                      content-type: application/json  # 设置消息类型，本次为json；文本可以设置
23                      为"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:
28          service-url:
29              defaultZone: http://localhost:7001/eureka
30
31  instance:
32      lease-renewal-interval-in-seconds: 2  # 设置心跳的时间间隔(默认30秒)
```

```
27     lease-expiration-duration-in-seconds: 5
28     instance-id: receive-8802.com      # 信息列表显示的主机名称
29     prefer-ip-address: true        # 访问的路径变为IP地址
```

注意：和生产者不同的地方主要是：

```
cloud:
  stream:
    binders:      # 在此处配置要绑定的rabbitmq的服务信息
    defaultRabbit:   # 表示定义的名称，用于binding整合，自定义
      type: rabbit    # 消息组件类型
    bindings:      # 服务的整合处理
      input:       # 这个名字是一个通道的名称
        destination: studyExchange # 表示要使用的Exchange名称，自定义
        content-type: application/json # 设置消息类型，本次为json；文本可以设置为"text/plain"
        binder: defaultRabbit      # 设置要绑定的消息服务的具体设置
eureka:
```

因为，生产者对应的是output通道，消费者对应的是input通道。

output通道的默认名就是 **output**，input通道的默认名就是 **input**。

主启动类

```
1  @SpringBootApplication
2  public class StreamMQMain8802 {
3      public static void main(String[] args) {
4          SpringApplication.run(StreamMQMain8802.class, args);
5      }
6  }
7
```

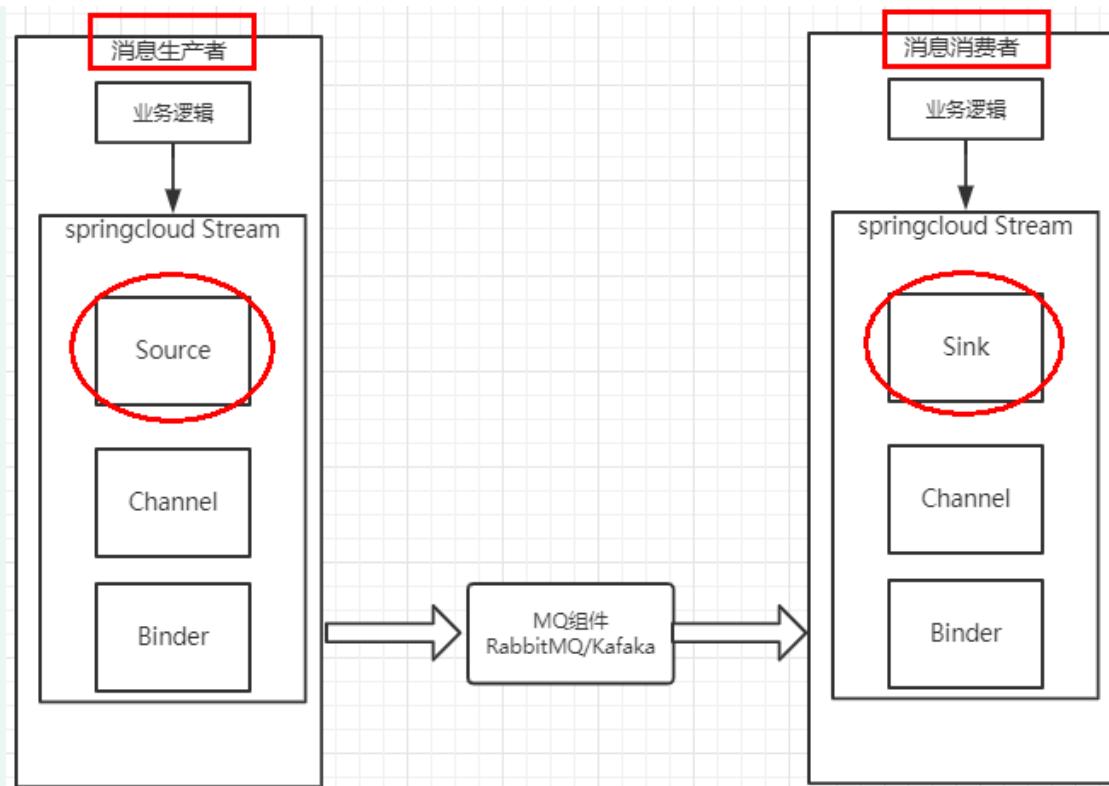
业务类

由于它是一个消费者，所以业务类中只要controller类就行了。

```
1  @EnableBinding(Sink.class)
2  public class ReceiveMessageListenerController {
3      @Value("${server.port}")
4      private String serverPort;
5
6      @StreamListener(Sink.INPUT)
7      public void input(Message<String> message) {
8          System.out.println("消费者1号， ----> 接收到的消息：" + message.getPayload()
9          + "\t port: " + serverPort);
10     }
11 }
```

注意：这里没有添加 **RestController** 注解

关于注解 **@EnableBinding** 和 **@StreamListener**，**Sink.class** 和 **Sink.INPUT**，可以看下图。

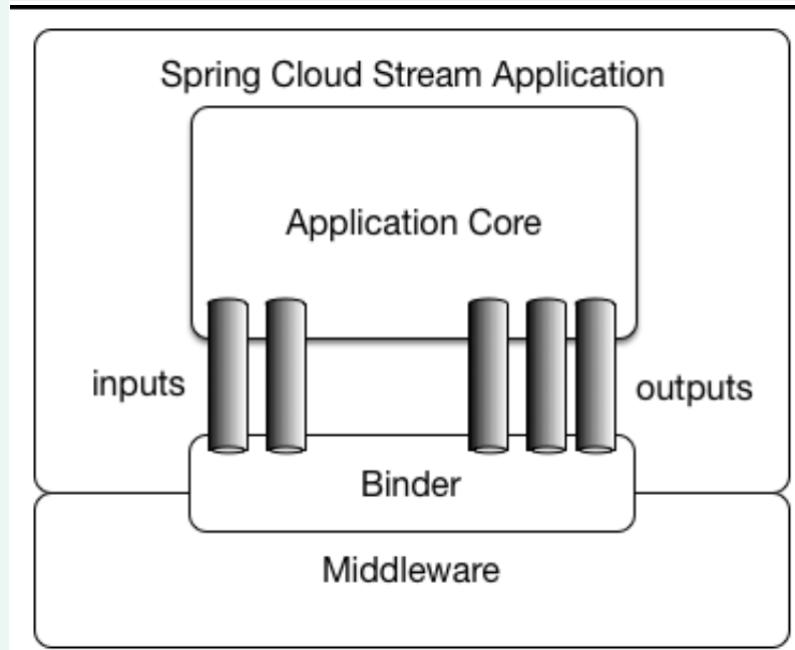


`@EnableBinding`：可以将信道channel 和 exchange 绑定在一起

`@StreamListener`：监听队列，用于消费者的队列的消息接收

`Sink.class`：就是图中消费者要连接的Stream中的Sink

`Sink.INPUT`：就是图中消费者要连接的Stream中的Channel。消费者对应output通道，如下图。



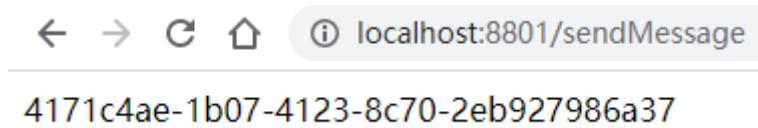
测试

1. 启动eureka7001
2. 启动rabbitmq
3. 启动生产者8801
4. 启动消费者8802

下面测试：8801发送消息，8802接收消息

1. 访问地址：<http://localhost:8801/sendMessage>

页面会返回：

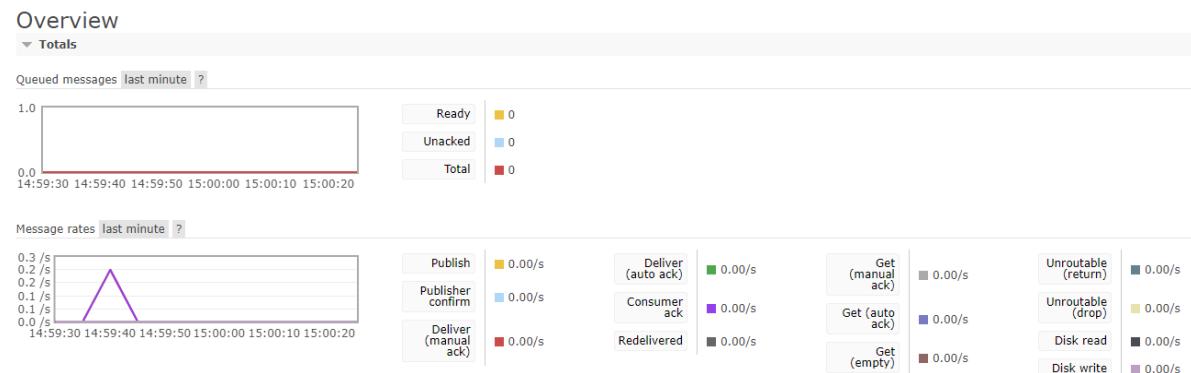


即，生产者发送了一个消息。

观察8802的控制台，发现会打印出该消息，说明接收到消息。

```
2022-05-07 14:57:31.086 INFO 1920 --- [(7)-10.1.125.73] o.s.web.servlet.DispatcherServlet: 消费者1号, ----> 接收到的消息: 4171c4ae-1b07-4123-8c70-2eb927986a37 port: 8802
```

观察rabbitmq管理页面：



可以发现有一个队列。

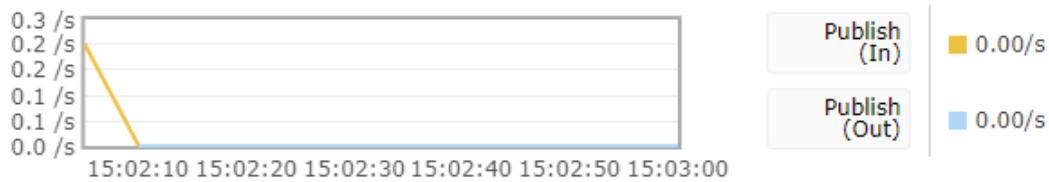
The screenshot shows the RabbitMQ Management Queues page. It lists one queue: 'studyExchange.anonymous.sJz_KfDRRpq3QnEjpwzyw'.

Overview				Messages			Message rates	
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver , get
studyExchange.anonymous.sJz_KfDRRpq3QnEjpwzyw	classic	AD Excl ML	idle	0	0	0	0.00/s	0.00/

Exchange: studyExchange

Overview

Message rates last minute ?



Details

Type	topic
Features	durable: true
Policy	

Bindings

This exchange



To	Routing key	Arguments	
studyExchange.anonymous.sJz_KfDRRpq3QnEjpgwzyw	#		Unbind

演示会出现的问题

为了演示操作，这里将8802复制一份，变成8803， [cloud-stream-rabbitmq-consumer8803](#)

配置基本一致，主要修改端口号之类的。

将8802作为消费者1号，将8803作为消费者2号。

- 启动eureak7001, rabbitmq
- 启动8801, 8802, 8803

目前有一个消息生产者，两个消息消费者。这时运行就会出现两个问题：**有重复消费问题、消息持久化问题**

- 重复消费问题

我们先来演示重复消费问题。

- 两次访问： <http://localhost:8801/sendMessage>

查看控制台，可以看到8802和8803同时都接收到了两条消息

```
消费者1号, ----> 接收到的消息: 33b00774-01bf-4955-b3d0-70dfe2505403 port: 8802
消费者1号, ----> 接收到的消息: 612489dc-6708-4b21-8e68-90d689764ed3 port: 8802
```

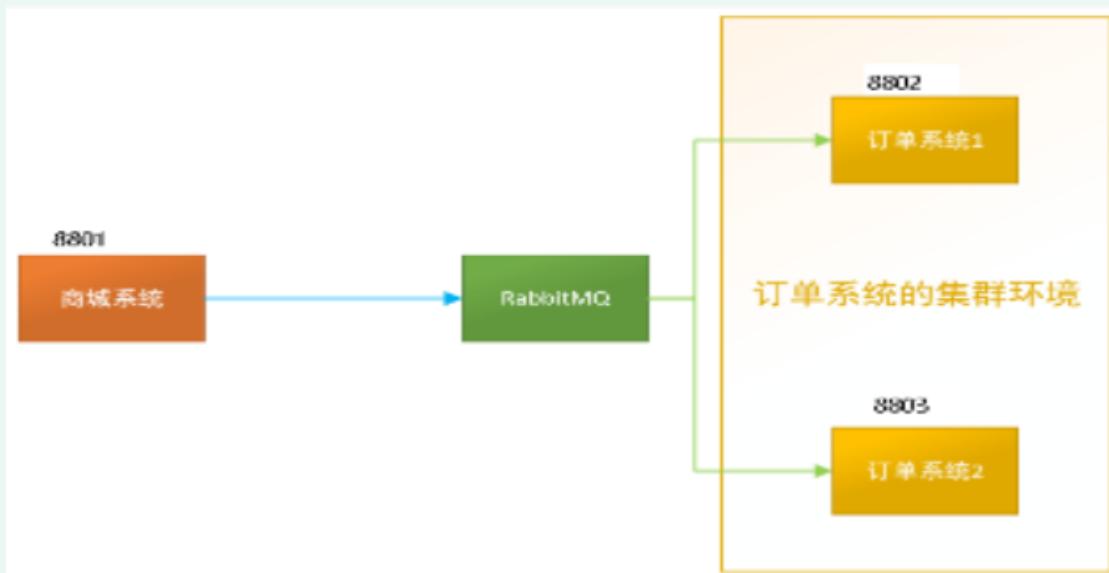
```
消费者2号, ----> 接收到的消息: 33b00774-01bf-4955-b3d0-70dfe2505403 port: 8803
消费者2号, ----> 接收到的消息: 612489dc-6708-4b21-8e68-90d689764ed3 port: 8803
```

那么如何解决?

利用分组和持久化属性group

为什么要解决重复消费问题?

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, 只有其中一个可以消费。

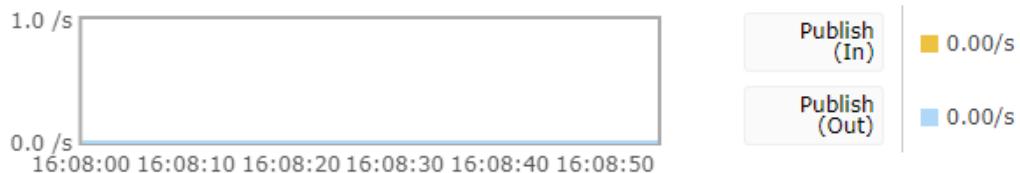
那么, 如何知道这两个消费者是否是同一组?

可以查看rabbitmq管理页面, 服务对应的交换机为 `studyExchange`, 可以看到它绑定了两个队列, 且 `Routing key` 都是`#`, 说明交换机收到生产者的消息会广播到两个队列中, 而两个消费者分别绑定了一个队列, 则它们就属于不同组。

Exchange: studyExchange

Overview

Message rates last minute ?



Details

Type	topic
Features	durable: true
Policy	

Bindings

This exchange



To	Routing key	Arguments	
studyExchange.anonymous.CLoIgn_HQFmsdVmbLixlKg	#		<button>Bind</button>
studyExchange.anonymous.o70moeOxTpeh7Iziou9ulw	#		<button>Bind</button>

持久化问题在下面一节中演示

分组消费与持久化

出现重复消息的原因是：默认分组group是不同的。

将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。

下面我们这样做：

- 先自定义配置分组
- 再自定义配置分为同一个组，解决重复消费问题

自定义分组

我们先演示如何自定义分组。

下面我们准备将8802分到一个自定义为 `atguiguA` 的组，将8803分到一个自定义为 `atguiguB` 的组。

操作如下：

1. 修改8802的yml文件。只要添加如下一行的代码。

```
cloud:
  stream:
    binders: # 在此处配置要绑定的rabbitmq的服务信息
      defaultRabbit: # 表示定义的名称, 用于binding整合, 自定义
        type: rabbit # 消息组件类型
    bindings: # 服务的整合处理
      input: # 这个名字是一个通道的名称
        destination: studyExchange # 表示要使用的Exchange名称, 自定义
        content-type: application/json # 设置消息类型, 本次为json; 文本可以设置为"text/plain"
        binder: defaultRabbit # 设置要绑定的消息服务的具体设置
      group: atguiguA
```

2. 同样, 修改8803的yml文件。只要添加如下一行的代码。

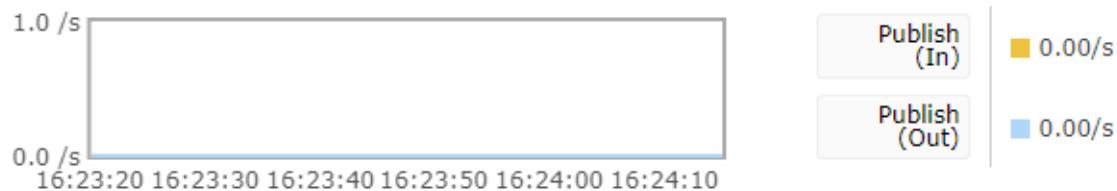
```
cloud:
  stream:
    binders: # 在此处配置要绑定的rabbitmq的服务信息
      defaultRabbit: # 表示定义的名称, 用于binding整合, 自定义
        type: rabbit # 消息组件类型
    bindings: # 服务的整合处理
      input: # 这个名字是一个通道的名称
        destination: studyExchange # 表示要使用的Exchange名称, 自定义
        content-type: application/json # 设置消息类型, 本次为json; 文本可以设置为"text/plain"
        binder: defaultRabbit # 设置要绑定的消息服务的具体设置
      group: atguiguB
```

上述操作就能自定义分组。下面我们查看rabbitmq管理页面, 看下配置是否生效。

Exchange: studyExchange

Overview

Message rates last minute ?



Details

Type	topic
Features	durable: true
Policy	

Bindings

This exchange



To	Routing key	Arguments	
studyExchange.atguiguA	#		<button>Bind</button> <button>Unbind</button>
studyExchange.atguiguB	#		<button>Bind</button> <button>Unbind</button>

可以看到，已经有了自定义分组。

当然，现在虽然自定义分组了，但是两个消费者还是在不同分组中，所以还是存在重复消费问题。

解决重复消费问题

既然知道如何自定义分组了，那么我们将两个消费者分到相同组就可以避免重复消费了。

操作：

- 修改8802的yml文件。将配置中的 `group` 的值写成 `atguiguA`
- 修改8803的yml文件。将配置中的 `group` 的值写成 `atguiguA`

即将两个服务的group改成相同的，就是分到相同组。

此时再测试，就可以发现没有重复消费问题了。

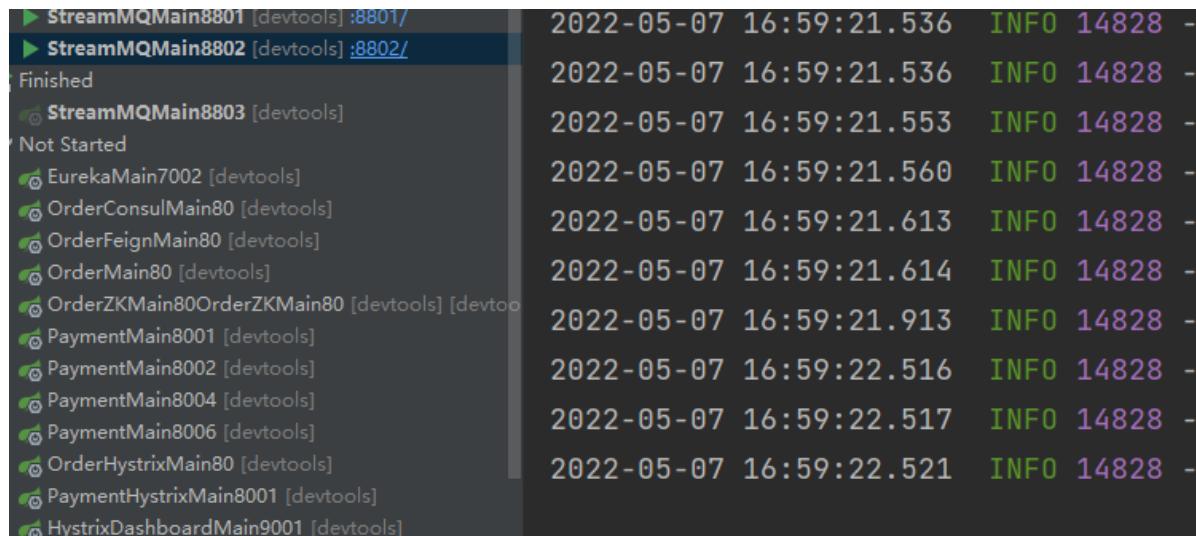
持久化问题演示

通过上述分组，我们解决了重复消费问题，其实也解决了持久化的问题。

我们可以看一下，没有分组为什么就会出现持久化问题？

演示操作：

1. 停掉8802和8803服务
2. 在8802中的 `application.yml` 中去掉group配置。 (注意：8803的分组还是atguiguA)
3. 四次访问：<http://localhost:8801/sendMessage>。发送消息到mq
4. 先启动8802，它是无分组属性配置，观察控制台可以发现没有接收到消息



Time	Level	Component	Message
2022-05-07 16:59:21.536	INFO	14828	-
2022-05-07 16:59:21.536	INFO	14828	-
2022-05-07 16:59:21.553	INFO	14828	-
2022-05-07 16:59:21.560	INFO	14828	-
2022-05-07 16:59:21.613	INFO	14828	-
2022-05-07 16:59:21.614	INFO	14828	-
2022-05-07 16:59:21.913	INFO	14828	-
2022-05-07 16:59:22.516	INFO	14828	-
2022-05-07 16:59:22.517	INFO	14828	-
2022-05-07 16:59:22.521	INFO	14828	-

5. 再启动8803，它是有分组属性配置，观察控制台可以发现它收到了消息。



Time	Level	Component	Message
2022-05-07 17:01:24.903	INFO	23200	[restartedMain] o.s.i.monitor.Inte
消费者2号, ----> 接收到的消息: d5df4933-4691-4115-9222-e13966d4af27 port: 8803			
消费者2号, ----> 接收到的消息: 5397a31e-5356-4b86-abb3-20a2c9761611 port: 8803			
消费者2号, ----> 接收到的消息: 09ccd0e1-acf5-408e-9610-8987259759c8 port: 8803			
消费者2号, ----> 接收到的消息: 11dcab26-82fe-4466-8ecd-3b39b6ea9fc9 port: 8803			

小结

对于分组消费与持久化的原理探究，我觉得主要还是跟RabbitMQ的特性有关。想了解的可以去看 RabbitMQ。

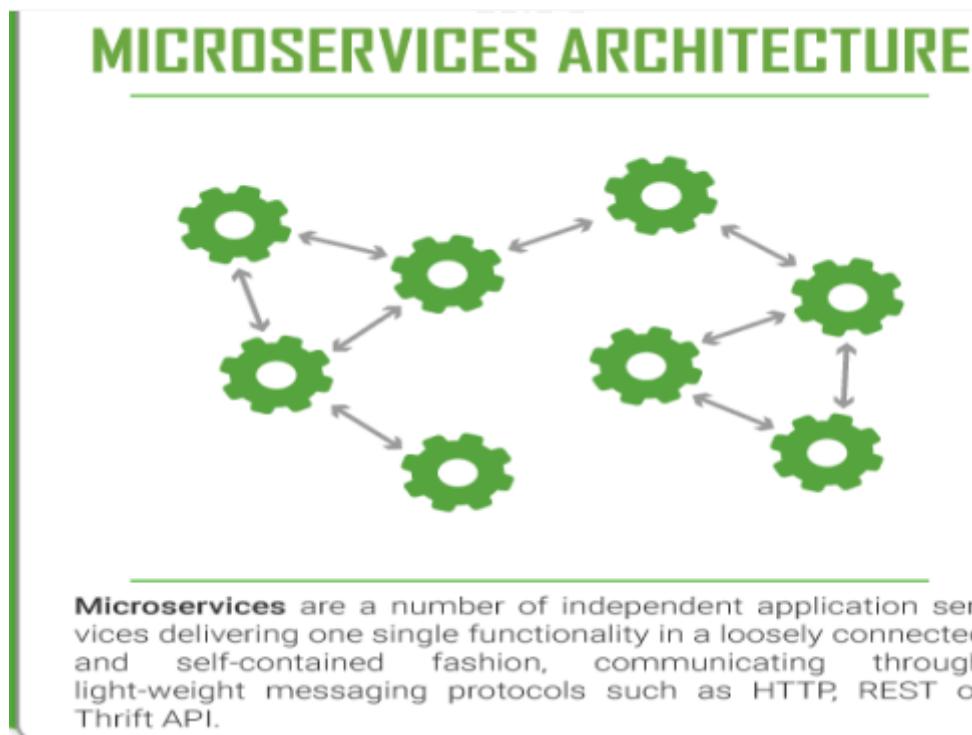
十四、SpringCloud Sleuth 分布式请求链路跟踪

概述

为什么会出现这个技术&解决哪些问题

为什么会出现这个技术？需要解决哪些问题？

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。



因此，我们有必要知道在调用一个服务A后，从而会调用哪些其他服务，并且调用每个服务所花费的时间是多少

SpringCloud Sleuth是什么

官网：<https://cloud.spring.io/spring-cloud-sleuth/1.0.x/>

github：<https://github.com/spring-cloud/spring-cloud-sleuth>

<https://spring.io/projects/spring-cloud-sleuth>

Spring Cloud Sleuth 提供了一套完整的服务跟踪解决方案。

在分布式系统中提供追踪解决方案并且兼容支持了zipkin

Spring Cloud Sleuth

Adrian Cole, Spencer Gibb, Marcin Grzejszczak, Dave Syer

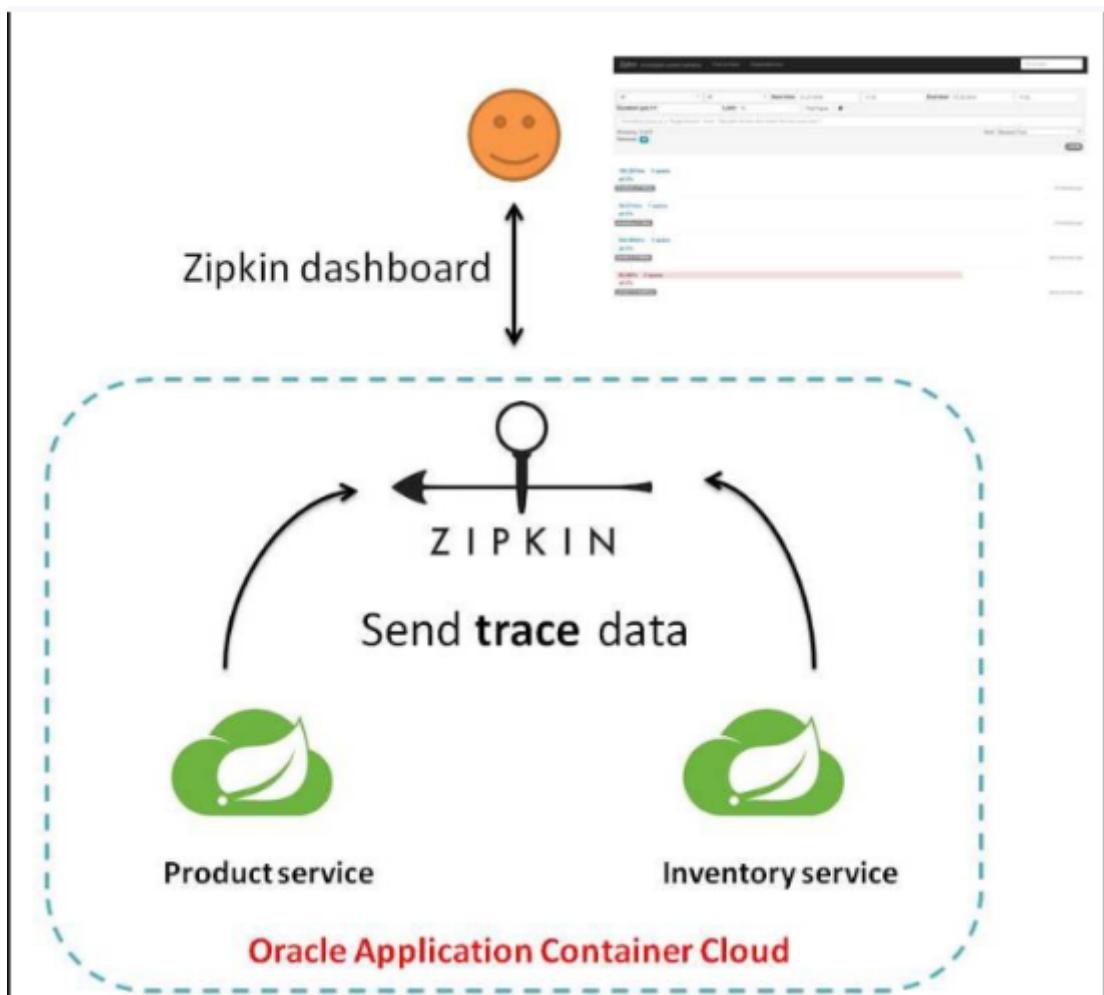
1.0.13.BUILD-SNAPSHOT

Spring Cloud Sleuth implements a distributed tracing solution for [Spring Cloud](#).

Table of Contents

- Terminology
- Purpose
 - Distributed tracing with Zipkin
 - Visualizing errors
 - Live examples
 - Log correlation
 - JSON Logback with Logstash
- Adding to the project
 - Only Sleuth (log correlation)
 - Sleuth with Zipkin via HTTP

解决的架构图：



搭建链路监控步骤

zipkin

下载

注意：SpringCloud从F版起，已不需要自己构建Zipkin Server了，只需调用jar包即可。

下载该jar包的地址：<https://repo1.maven.org/maven2/io/zipkin/java/zipkin-server/>

里面版本很多，我们下载2.12.9版本的。

io/zipkin/java/zipkin-server/2.12.9

./			
zipkin-server-2.12.9-exec.jar	2019-04-12 10:45	48245638	
zipkin-server-2.12.9-exec.jar.asc	2019-04-12 10:45	821	
zipkin-server-2.12.9-exec.jar.md5	2019-04-12 10:45	32	
zipkin-server-2.12.9-exec.jar.md5.asc	2019-04-12 10:45	821	
zipkin-server-2.12.9-exec.jar.sha1	2019-04-12 10:45	40	

运行jar

后续，我们直接用以下这个命令运行jar

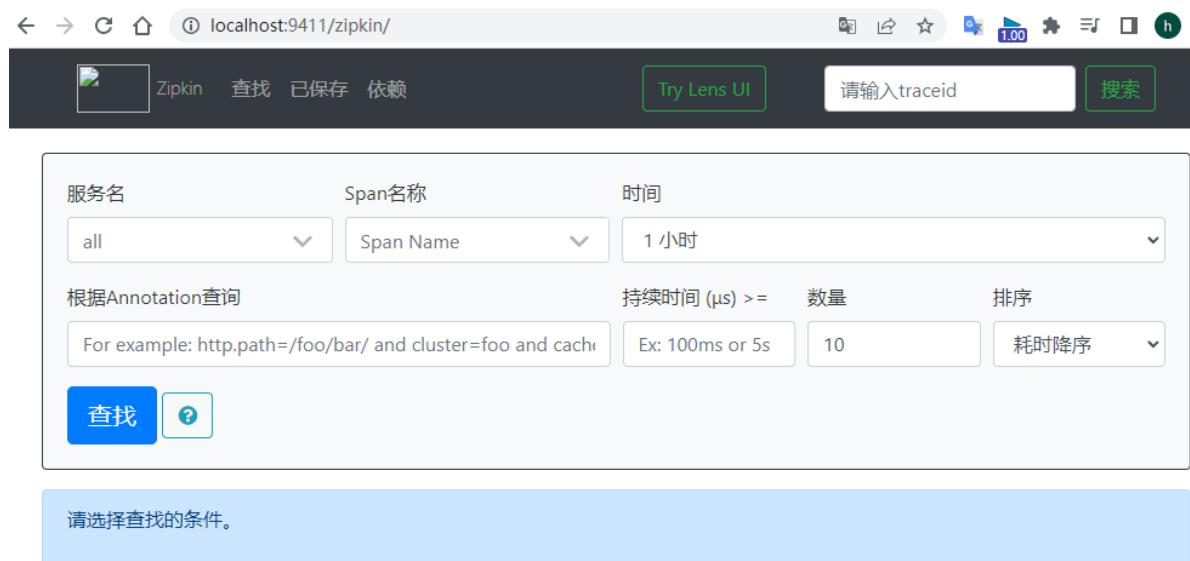
```
1     java -jar zipkin-server-2.12.9-exec.jar
```

在cmd中运行后会出现如下图案。

```
D:\java_study\cloud2020>java -jar zipkin-server-2.12.9-exec.jar
*****
  **      **
   *      *
  **      **
  **      **
  **      **
  *****
  ****
  ****
 ****      ****
 ****      ****
 *****
 *****
 *****
 *****
 *****
 ****
  ****
   **
   **
 ****      **      ****      **  **
   **      **      **  *
   **      **      ****
   **      **      **  **
 ****      **      **  **
 :: Powered by Spring Boot ::      (v2.1.4.RELEASE)
```

运行控制台

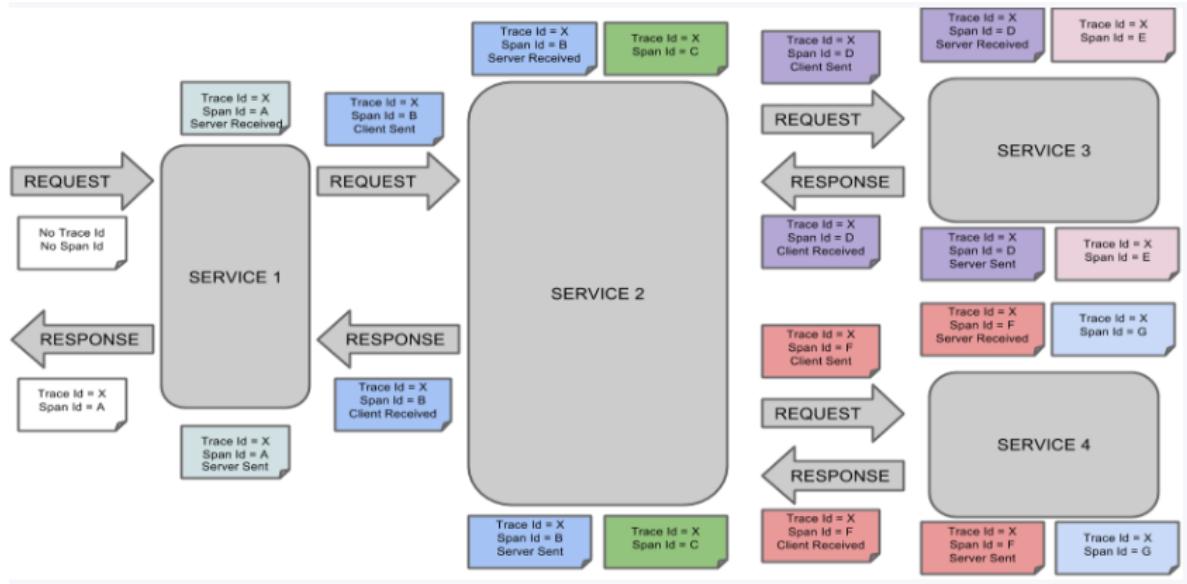
上述cmd中显示的是后台，还有个前台，可以访问地址：<http://localhost:9411/zipkin/>



术语

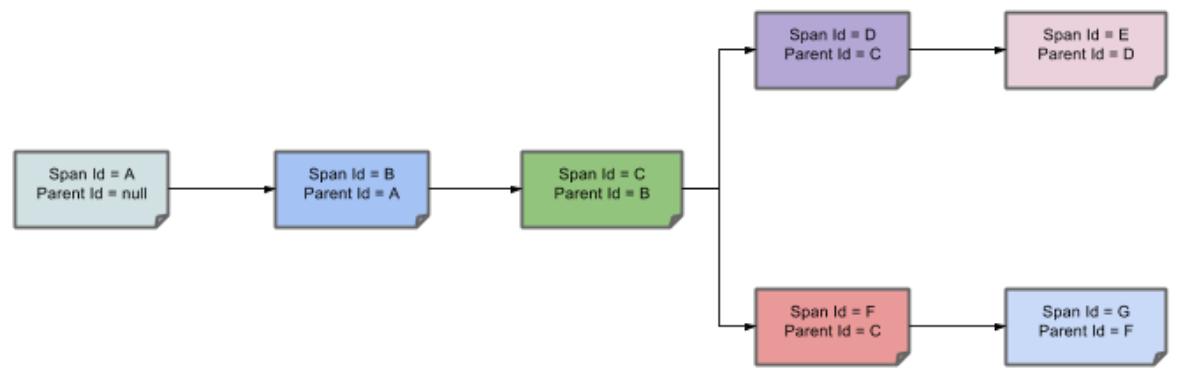
1. 完整的调用链路

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

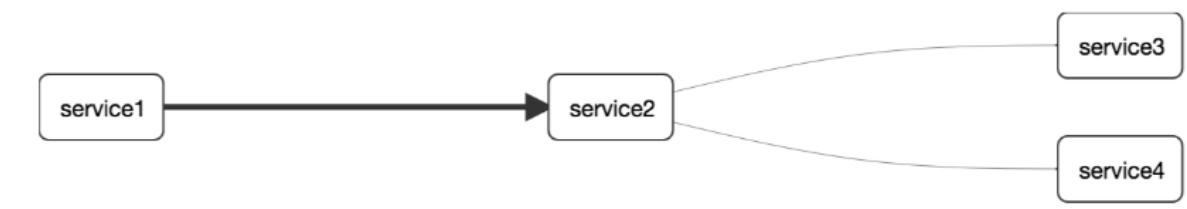


2. 简单解释上图

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：



3. 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：标识调用链路来源，通俗的理解 span就是一次请求信息。

服务提供者

由于之前新建了很多服务，所以在这里就在之前服务的基础上做修改，完成演示。

利用 `cloud-provider-payment8001` 作为服务提供者

改POM

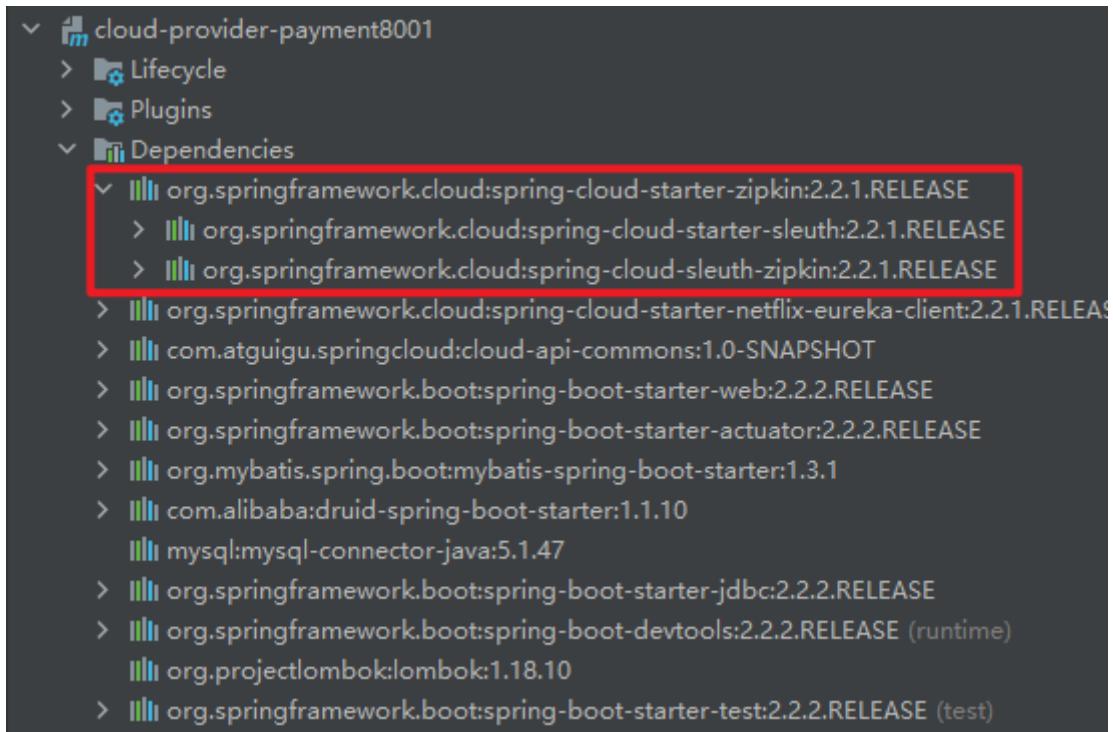
```
1 <dependencies>
2     <!--包含了sleuth+zipkin-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-zipkin</artifactId>
6     </dependency>
7     <!-- 导入 eureka-client 包 -->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
11    </dependency>
12    <!-- 引用自己定义的 api 通用包 -->
13    <dependency>
14        <groupId>com.atguigu.springcloud</groupId>
15        <artifactId>cloud-api-commons</artifactId>
16        <version>${project.version}</version>
17    </dependency>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-starter-web</artifactId>
21    </dependency>
22    <dependency>
23        <groupId>org.springframework.boot</groupId>
24        <artifactId>spring-boot-starter-actuator</artifactId>
25    </dependency>
26    <dependency>
27        <groupId>org.mybatis.spring.boot</groupId>
28        <artifactId>mybatis-spring-boot-starter</artifactId>
29    </dependency>
30    <dependency>
31        <groupId>com.alibaba</groupId>
32        <artifactId>druid-spring-boot-starter</artifactId>
33        <version>1.1.10</version>
34    </dependency>
35    <!--mysql-connector-java-->
36    <dependency>
37        <groupId>mysql</groupId>
38        <artifactId>mysql-connector-java</artifactId>
39    </dependency>
40    <!--jdbc-->
41    <dependency>
42        <groupId>org.springframework.boot</groupId>
43        <artifactId>spring-boot-starter-jdbc</artifactId>
44    </dependency>
45    <dependency>
46        <groupId>org.springframework.boot</groupId>
47        <artifactId>spring-boot-devtools</artifactId>
48        <scope>runtime</scope>
```

```
49         <optional>true</optional>
50     </dependency>
51     <dependency>
52         <groupId>org.projectlombok</groupId>
53         <artifactId>lombok</artifactId>
54         <optional>true</optional>
55     </dependency>
56     <dependency>
57         <groupId>org.springframework.boot</groupId>
58         <artifactId>spring-boot-starter-test</artifactId>
59         <scope>test</scope>
60     </dependency>
61 </dependencies>
```

其实就是在原来基础上，添加了一个包：

```
<dependencies>
    <!-- 包含了sleuth+zipkin-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
```

可以看到这个包下包含了sleuth和zipkin



YML

```
1 server:
2   port: 8001
3 spring:
4   application:
5     name: cloud-payment-service
```

```

6     zipkin:
7         base-url: http://localhost:9411
8     sleuth:
9         sampler:
10            # 采样率值介于 0 到 1 之间. 1 则表示全部采集
11            probability: 1
12     datasource:
13         type: com.alibaba.druid.pool.DruidDataSource
14         driver-class-name: com.mysql.jdbc.Driver
15         url: jdbc:mysql://localhost:3306/db2019?
16         useUnicode=true&characterEncoding=utf-8&useSSL=false
17         username: root
18         password: 123456
19     mybatis:
20         mapper-locations: classpath:mapper/*.xml
21         type-aliases-package: com.atguigu.springcloud.entities
22
23 eureka:
24     client:
25         # 表示是否将自己注册进EurekaServer, 默认为true
26         register-with-eureka: true
27         # 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true才能
28         # 配合ribbon使用负载均衡
29         fetch-registry: true
30         service-url:
31             #      defaultZone:
32             http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
33             defaultZone: http://eureka7001.com:7001/eureka
34         instance:
35             instance-id: payment8001
36             prefer-ip-address: true
37             # Eureka 客户端向服务端发送心跳的时间间隔, 单位是秒 (默认30s)
38             lease-renewal-interval-in-seconds: 1
39             # Eureka 服务端在收到最后一次心跳后等待时间上限, 单位为秒 (默认90s), 超时将剔除服务
40             lease-expiration-duration-in-seconds: 2

```

主要添加了以下配置：

```

spring:
  application:
    name: cloud-payment-service
    zipkin:
      base-url: http://localhost:9411
    sleuth:
      sampler:
        # 采样率值介于 0 到 1 之间. 1 则表示全部采集
        probability: 1

```

业务类controller

```
1  @RestController
2  @Slf4j
3  public class PaymentController {
4
5      @GetMapping("/payment/zipkin")
6      public String paymentZipkin() {
7          return "hi, i am paymentzipkin fall back, welcome to atguigu!!!";
8      }
9
10 }
```

服务消费者

以 `cloud-consumer-order80` 模块为基础

改POM

```
1 <dependencies>
2     <!--包含了sleuth+zipkin-->
3     <dependency>
4         <groupId>org.springframework.cloud</groupId>
5         <artifactId>spring-cloud-starter-zipkin</artifactId>
6     </dependency>
7     <!-- 导入 eureka-client 包 -->
8     <dependency>
9         <groupId>org.springframework.cloud</groupId>
10        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
11    </dependency>
12    <!-- 引用自己定义的 api 通用包 -->
13    <dependency>
14        <groupId>com.atguigu.springcloud</groupId>
15        <artifactId>cloud-api-commons</artifactId>
16        <version>${project.version}</version>
17    </dependency>
18    <dependency>
19        <groupId>org.springframework.boot</groupId>
20        <artifactId>spring-boot-starter-web</artifactId>
21    </dependency>
22    <dependency>
23        <groupId>org.springframework.boot</groupId>
24        <artifactId>spring-boot-starter-actuator</artifactId>
25    </dependency>
26
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>
```

主要新加的包：

```
<dependencies>
    <!-- 包含了sleuth+zipkin-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zipkin</artifactId>
    </dependency>
    <!-- 导入 eureka-client 包 -->
```

写YML

```
1  server:
2      port: 80
3
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true才能
9          # 配合ribbon使用负载均衡
10         fetch-registry: true
11         service-url:
12             defaultZone: http://eureka7001.com:7001/eureka
13
14  spring:
15      application:
16          name: cloud-order-service
17      zipkin:
18          base-url: http://localhost:9411
19      sleuth:
20          sampler:
21              probability: 1
```

```
spring:
  application:
    name: cloud-order-service
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      probability: 1
```

写业务类

```
1  @RestController
2  @Slf4j
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/zipkin")
10     public String paymentZipkin() {
11         String result = restTemplate.getForObject(PAYMENT_URL +
12             "/payment/zipkin/", String.class);
13         return result;
14     }
15 }
```

测试

- 依次启动eureka7001、8001、80
- 打开浏览器多次访问：<http://localhost/consumer/payment/zipkin>
- 访问：<http://localhost:9411/zipkin/>

就能检测到链路中的服务

The screenshot shows the Zipkin UI interface. At the top, there's a navigation bar with links for 'Zipkin', '查找' (Search), '已保存' (Saved), and '依赖' (Dependencies). On the right side of the header, there are buttons for 'Try Lens UI', a search input field '请输入.traceid' (Enter traceid), and a '搜索' (Search) button.

The main area is a table with three columns: '服务名' (Service Name), 'Span名称' (Span Name), and '时间' (Time). The '服务名' column has dropdown menus for 'all' and 'cloud-order-service'. The 'Span名称' column has dropdown menus for 'all' and 'bar/ and cluster=foo and cach'. The '时间' column has a dropdown menu set to '1 小时' (1 hour).

Below the table, there are filters for '持续时间 (μs) >=' (Duration (μs) >=) and '数量' (Count), both set to 'Ex: 100ms or 5s' and '10'. There's also a '排序' (Sort) dropdown set to '耗时降序' (Descending by Duration).

A red box highlights the '服务名' dropdown menu, which lists 'all' (selected), 'cloud-order-service', and 'cloud-payment-service'.

服务名 **选择要查找的服务名**
cloud-order-service

Span名称 **选择要查找的方法**
all

时间 **选择多久前调用的**
1 小时

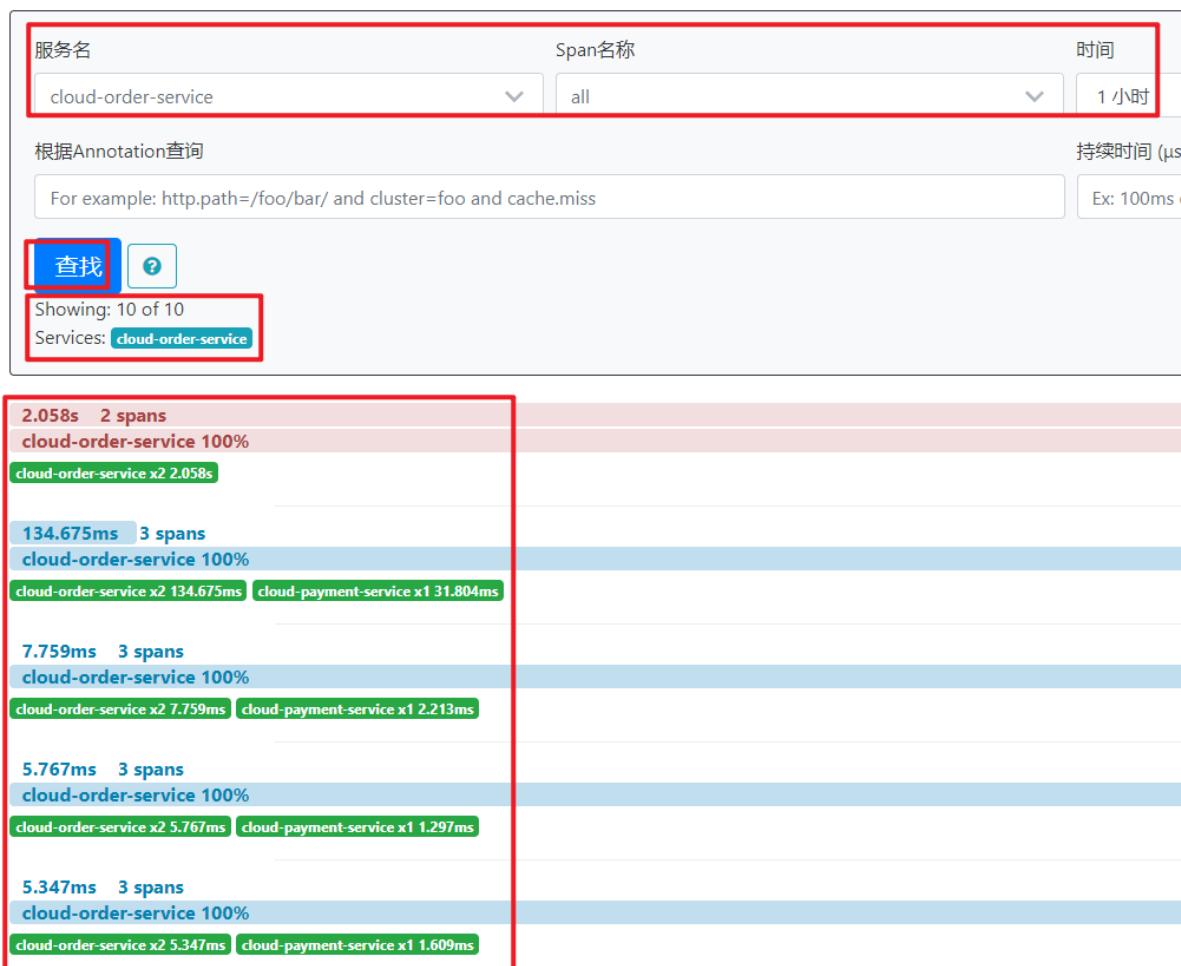
根据Annotation查询
For example: http.path=/foo/ba

查找 **?**

请选择要查找的属性

选择上述要查找的东西后，点击查找，会出现以下界面。

我们可以清晰的看到调用的服务，以及请求所走过的所有服务链路和所耗费的时间。



十五、SpringCloud Alibaba 入门简介

为什么会出现SpringCloud Alibaba

主要因为Spring Cloud Netflix项目进入维护模式了。

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

可以看到：

Notable Changes in the Greenwich Release Train

This milestone is compatible with Spring Boot 2.1.1.RELEASE. Updates were made across the projects for Java 11 compatibility.

Go to the [Greenwich.RC1 github project](#) to see all issues assigned to this release.

Spring Cloud Netflix Projects Entering Maintenance Mode

Recently, Netflix [announced](#) that Hystrix is entering maintenance mode. Ribbon has been in a [similar state](#) since 2016. Although Hystrix and Ribbon are now in maintenance mode, they are still deployed at scale at Netflix.

The Hystrix Dashboard and Turbine have been superseded by Atlas. The last commits to these project are 2 years and 4 years ago respectively. Zuul 1 and Archaius 1 have both been superseded by later versions that are not backward compatible.

The following Spring Cloud Netflix modules and corresponding starters will be placed into maintenance mode:

1. spring-cloud-netflix-archaius
2. spring-cloud-netflix-hystrix-contract
3. spring-cloud-netflix-hystrix-dashboard
4. spring-cloud-netflix-hystrix-stream
5. spring-cloud-netflix-hystrix
6. spring-cloud-netflix-ribbon
7. spring-cloud-netflix-turbine-stream
8. spring-cloud-netflix-turbine
9. spring-cloud-netflix-zuul

This does not include the Eureka or concurrency-limits modules.

SpringCloud Alibaba带来了什么？

SpringCloud Alibaba是什么

SpringCloud Alibaba GitHub官网：<https://github.com/alibaba/spring-cloud-alibaba>

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。

SpringCloud Alibaba能干什么

服务限流降级：默认支持 Servlet、Feign、RestTemplate、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。

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。

等等

其实关于SpringCloud Alibaba的介绍可以在GitHub上了解：<https://github.com/alibaba/spring-cloud-alibaba/blob/2.2.x/README-zh.md>

学之前建议一定要看下官网！

如何引入SpringCloud Alibaba依赖？

如何使用

如何引入依赖

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

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

然后在 `dependencies` 中添加自己所需使用的依赖即可使用。

```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>com.alibaba.cloud</groupId>
5       <artifactId>spring-cloud-alibaba-dependencies</artifactId>
6       <version>2.2.7.RELEASE</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10    </dependencies>
11 </dependencyManagement>
```

下面我们会使用 **spring cloud alibaba 2.1.0.RELEASE**

SpringCloud Alibaba主要组件

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。

SpringCloud Alibaba学习资料获取

官网：<https://spring.io/projects/spring-cloud-alibaba>

Spring Cloud Alibaba provides a one-stop solution for distributed application development. It contains all the components required to develop distributed applications, making it easy for you to develop your applications using Spring Cloud.

With Spring Cloud Alibaba, you only need to add some annotations and a small amount of configurations to connect Spring Cloud applications to the distributed solutions of Alibaba, and build a distributed application system with Alibaba middleware.

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。

GitHub地址：<https://github.com/alibaba/spring-cloud-alibaba>

官方参考文档：<https://github.com/alibaba/spring-cloud-alibaba/wiki>

十六、SpringCloud Alibaba Nacos服务注册和配置中心

之前我们讲过服务注册和配置中心。之前我们学过：

- 服务注册：eureka、zookeeper、consul
- 配置中心：springcloud config、springcloud bus

而将要学习的nacos 可以兼顾两者。

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。

Nacos是什么

Nacos官方文档：<https://nacos.io/zh-cn/docs/what-is-nacos.html>

Nacos是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

Nacos是什么?

Nacos简介

概念

架构

功能和需求列表

Nacos 2.0

兼容性及使用

升级文档

鉴权插件

配置加密插件

快速开始

Nacos

Nacos Spring

什么是 Nacos

概览

欢迎来到 Nacos 的世界！

Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。

Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos 是构建以“服务”为中心的现代应用架构（例如微服务范式、云原生范式）的服务基础设施。

什么是 Nacos？

服务（Service）是 Nacos 世界的一等公民。Nacos 支持几乎所有主流类型的“服务”的发现、配置和管理：

[Kubernetes Service](#)

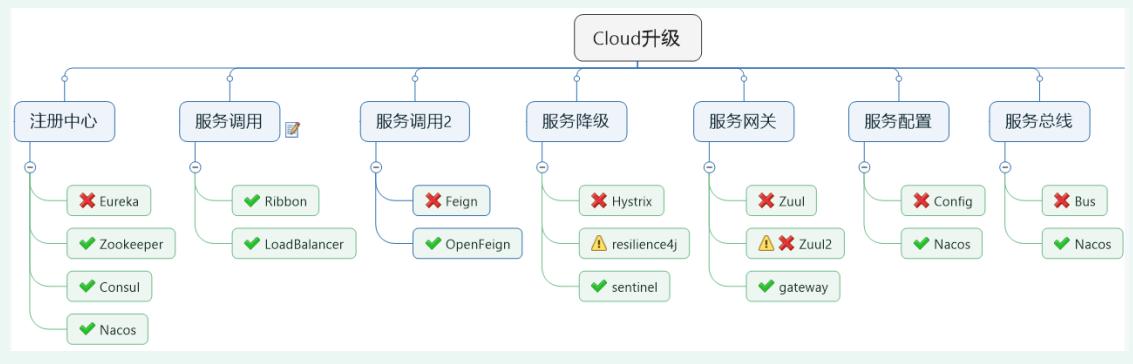
[gRPC & Dubbo RPC Service](#)

[Spring Cloud RESTful Service](#)

Nacos就是 注册中心+配置中心 的组合。

Nacos = Eureka + Config + Bus

就如下图所示。注册中心、服务配置以及服务总线中都用Nacos



Nacos能干什么

Nacos可以替代Eureka做服务注册中心，也可以替代Config做服务配置中心。

Nacos去哪下

<https://github.com/alibaba/Nacos>

官方文档：<https://nacos.io/zh-cn/index.html>

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

各种注册中心比较

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

Nacos安装并运行

安装和启动Nacos都可以参考：<https://nacos.io/zh-cn/docs/quick-start.html>

下载Nacos

首先，本地需要有Java8+Maven

然后从官网下载Nacos。

下载步骤参考官网：<https://nacos.io/zh-cn/docs/quick-start.html>

本次我们选择1.1.4的版本

运行Nacos

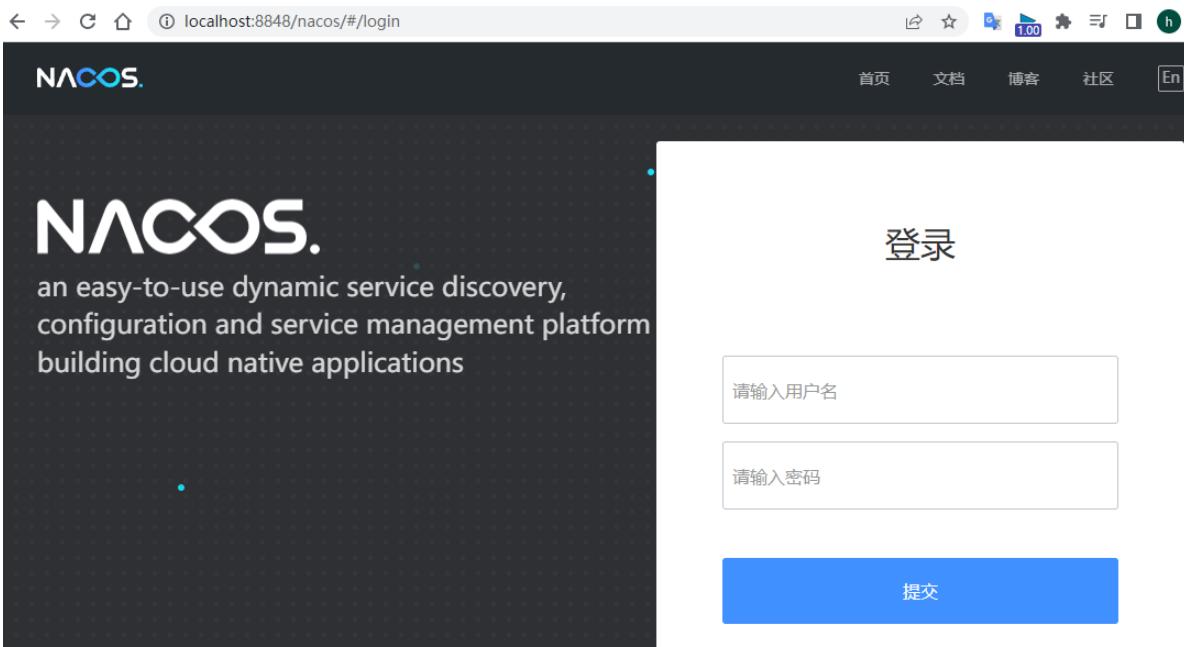
下载后，解压安装包。直接运行bin目录下的startup.cmd

```
D:\java_study\cloud2020\nacos\bin\startup.cmd
Nacos 1.1.4
Running in stand alone mode, All function modules
Port: 8848
Pid: 924
Console: http://192.168.179.1:8848/nacos/index.html
https://nacos.io

2022-05-08 12:54:50,259 INFO Bean 'org.springframework.security.config.annotation.configuration.ObjectPostProcessorConfiguration' of type [org.springframework.security.config.annotation.configuration.ObjectPostProcessorConfiguration$$EnhancerBySpringCGLIB$$f525183a] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
```

运行成功后，直接访问：<http://localhost:8848/nacos>

就会出现如下页面



默认用户名和密码都是 nacos

登录后：

A screenshot of the Nacos configuration management interface. The header is identical to the login page. The left sidebar shows a navigation tree with 'NACOS 1.1.4' at the top, followed by '配置管理' (Configuration Management) with '配置列表' (List) selected, '历史版本' (History Versions), and '监听查询' (Monitoring Query); '服务管理' (Service Management) with '服务列表' (List) and '订阅者列表' (Subscriber List); and '命名空间' (Namespace) with '命名空间' (Namespace). The main content area is titled 'public' and shows a search bar with '配置管理 | public' and a message '查询结果: 共查询到 0 条满足要求的配置' (Query results: 0 items found). Below the search bar are two input fields: 'Data ID:' and 'Group:', each with a placeholder '模糊查询请输入Data ID' or 'Group'. There are buttons for '查询' (Search), '高级查询' (Advanced Search), and '导出查询结果' (Export Query Results). A table below the search bar has columns for '操作' (Operation), '归属应用' (Owner Application), 'Group', and 'Data Id'. The table body is empty with the text '没有数据' (No data).

Nacos作为服务注册中心

官方文档地址：<https://spring.io/projects/spring-cloud-alibaba#learn>

Spring Cloud Alibaba

2021.0.1.0



OVERVIEW LEARN SAMPLES

Documentation

Each Spring project has its own; it explains in great details how you can use **project features** and what you can achieve with them.

2021.0.1.0	CURRENT	GA	Reference Doc.	API Doc.
2.2.7.RELEASE	GA		Reference Doc.	API Doc.
2.1.2.RELEASE	GA		Reference Doc.	API Doc.
2.0.2.RELEASE	GA		Reference Doc.	API Doc.
1.5.1.RELEASE	GA		Reference Doc.	API Doc.

<https://spring-cloud-alibaba-group.github.io/github-pages/greenwich/spring-cloud-alibaba.html>
spring-cloud-alibaba-group.github.io/github-pages/greenwich/spring-cloud-alibaba.html

Spring Cloud Alibaba Reference Documentation

Jim Fang, Jing Xiao, Mercy Ma, Xiaolong Zuo, Bingting Peng, Yuxin Wang

Table of Contents

1. Introduction

2. Dependency Management

3. Spring Cloud Alibaba Nacos Discovery

3.1. Service Registration/Discovery: Nacos Discovery

3.2. How to Introduce Nacos Discovery for service registration/discovery

3.3. An example of using Nacos Discovery for service registration/discovery and call

3.3.1. Nacos Server Startup

3.3.2. Start a Provider Application

3.3.3. Start a Consumer Application

3.4. Nacos Discovery Endpoint

3.5. More Information about Nacos Discovery Starter Configurations

4. Spring Cloud Alibaba Nacos Config

4.1. How to Introduce Nacos Config for configuration

4.2. Quickstart

4.2.1. Initialize Nacos Server

Usage on the Client

其实下面的配置都可以参考上述文档!

下面我们演示 Nacos如何作为服务注册中心。

基于Nacos的服务提供者

新建Module

新建一个项目子工程名为 `cloudalibaba-provider-payment9001`

改POM

首先父工程的POM中一定要有：

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

然后子工程的POM：

```
1 <dependencies>
2   <dependency>
3     <groupId>com.alibaba.cloud</groupId>
4     <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5   </dependency>
6   <!-- SpringBoot整合Web组件 -->
7   <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-web</artifactId>
10    </dependency>
11    <dependency>
12      <groupId>org.springframework.boot</groupId>
13      <artifactId>spring-boot-starter-actuator</artifactId>
14    </dependency>
15    <!--日常通用jar包配置-->
16    <dependency>
17      <groupId>org.springframework.boot</groupId>
18      <artifactId>spring-boot-devtools</artifactId>
19      <scope>runtime</scope>
20      <optional>true</optional>
21    </dependency>
22    <dependency>
23      <groupId>org.projectlombok</groupId>
24      <artifactId>lombok</artifactId>
25      <optional>true</optional>
26    </dependency>
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter-test</artifactId>
30      <scope>test</scope>
31    </dependency>
32 </dependencies>
```

写YML

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

主启动

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class PaymentMain9001 {
4   public static void main(String[] args) {
5     SpringApplication.run(PaymentMain9001.class, args);
6   }
7 }
```

业务类

```
1 @RestController
2 public class PaymentController {
3   @Value("${server.port}")
4   private String serverPort;
5
6   @GetMapping("/payment/nacos/{id}")
7   public String getPayment(@PathVariable("id") Integer id) {
8     return "nacos registry, serverPort: " + serverPort + "\t id" + id;
9   }
10 }
```

测试

1. 先启动nacos
2. 再启动9001服务
3. 登录nacos管理页面: <http://localhost:8848/nacos>

可以在服务列表中看到已注册的服务!

NACOS 1.1.4

公共空间

服务列表 | 公共空间

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-payment-provider	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除

配置管理
配置列表
历史版本
监听查询
服务管理
服务列表
订阅者列表
命名空间
集群管理
节点列表

4. 访问: <http://localhost:9001/payment/nacos/1>

← → ⏪ ⏹ ⓘ localhost:9001/payment/nacos/1

nacos registry, serverPort: 9001 id1

为演示nacos负载均衡作准备

为了下一章节演示Nacos的负载均衡，参照9001新建9002

基于Nacos的服务消费者

新建Module

新建子工程 `cloudalibaba-consumer-nacos-order83`

改POM

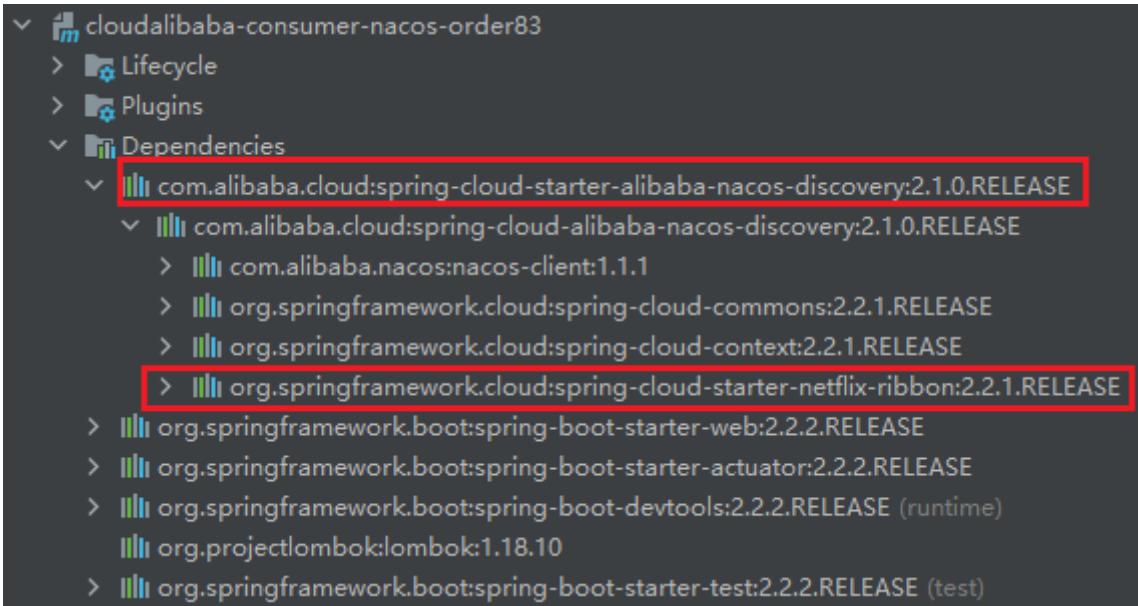
```

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

```

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

从引入的包中可以看到Nacos支持负载均衡，因为里面包含了ribbon



写YML

```
1 server:
2   port: 83
3 spring:
4   application:
5     name: nacos-order-consumer
6   cloud:
7     nacos:
8       discovery:
9         server-addr: 127.0.0.1:8848
10      # 消费者将要去访问的微服务名称(已注册到nacos的微服务提供者) 可选操作
11      service-url:
12        nacos-user-service: http://nacos-payment-provider
```

主启动

```
1  @SpringBootApplication
2  @EnableDiscoveryClient
3  public class OrderNacosMain83 {
4      public static void main(String[] args) {
5          SpringApplication.run(OrderNacosMain83.class, args);
6      }
7 }
```

业务类

1. 配置类

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

2. controller

```
1  @RestController
2  public class OrderNacosController {
3      @Resource
4      private RestTemplate restTemplate;
5
6      @Value("${service-url.nacos-user-service}")
7      private String serverURL;
8
9      @GetMapping("/consumer/payment/nacos/{id}")
10     public String paymentInfo(@PathVariable("id") Long id) {
11         return restTemplate.getForObject(serverURL + "/payment/nacos/" + id,
12             String.class);
13     }
14 }
```

例子中我们使用较原始的一种方式，即显示的使用RestTemolate的方式来访问。其实也可以通过带有负载均衡的RestTemplate 和 FeignClient 也是可以访问的，或者通过OpenFeign来访问！

测试

1. 启动nacos
2. 启动9001和9002
3. 启动83
4. 观察nacos管理页面

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

NACOS.

首页 文档 博客 社区 En nacos

NACOS 1.1.4

public

服务列表 | public

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-payment-provider	DEFAULT_GROUP	1	2	2	false	详情 示例代码 删除
nacos-order-consumer	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除

5. 多次访问: <http://localhost:83/consumer/payment/nacos/13>

可以发现能调用服务提供者的方法，并且具备负载均衡功能！

localhost:83/consumer/payment/nacos/13

nacos registry, serverPort: 9001 id13

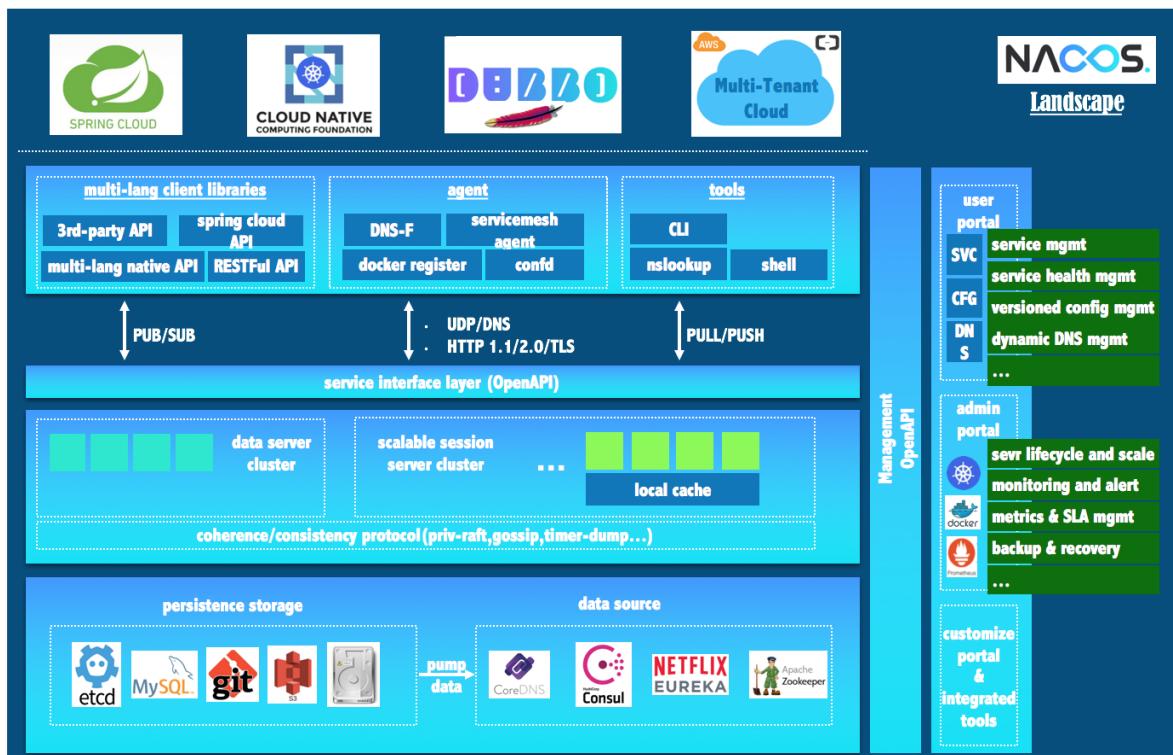
localhost:83/consumer/payment/nacos/13

nacos registry, serverPort: 9002 id13

服务注册中心对比

Nacos既支持AP，也支持CP，可以随时切换！

Nacos全景图：





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

切换

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（一致性）；而A的定义是所有的请求都会收到响应（可用性）。

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模式为了服务的可用性而减弱了一致性，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册服务，如果服务不存在，则会返回错误。

如何切换模式？

使用以下命令即可：

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

Nacos作为服务配置中心

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

在SpringCloud Config中，我们需要先将配置文件写到远程Gitee或GitHub，然后用Config结合Bus实现动态刷新和动态更新。

现在，有了Nacos，我们可以将配置文件写入Nacos，然后直接利用Nacos做Config类似的功能。

下面演示Nacos代替Config作服务配置中心。

在Nacos管理页面中有一个配置列表，我们可以将配置文件放到其中。

The screenshot shows the Nacos 1.1.4 configuration management interface. On the left, there's a sidebar with 'NACOS 1.1.4' at the top, followed by three sections: '配置管理' (Configuration Management) which has '配置列表' (Configuration List) selected and highlighted with a red box; '服务管理' (Service Management) with '服务列表' (Service List) and '订阅者列表' (Subscriber List); and '集群管理' (Cluster Management) with '节点列表' (Node List). The main panel has a header 'public' and a search bar with 'Data ID:' and 'Group:'. Below the search bar are buttons for '查询' (Query), '高级查询' (Advanced Query), '导出查询结果' (Export Query Results), '导入配置' (Import Configuration), and a red-boxed '+' button for adding new configurations. The table below the search bar is currently empty, showing '没有数据' (No Data).

点击上述界面中的 ，可以新建配置

The screenshot shows the Nacos configuration management interface. On the left, there is a sidebar with a tree structure:

- 配置管理
 - 配置列表
 - 历史版本
 - 监听查询
- 服务管理
 - 服务列表
 - 订阅者列表
- 命名空间
- 集群管理
 - 节点列表

The main area is titled "新建配置" (New Configuration). It contains the following fields:

- Data ID:
- Group: (X)
- 更多高级选项 (More Advanced Options)
- Description:
- 配置格式 (Configuration Format): TEXT (selected), JSON, XML, YAML, HTML, Properties
- 配置内容 (Configuration Content):
1
⑦ :

At the bottom right are two buttons: "发布" (Publish) and "返回" (Back).

Nacos作为配置中心-基础配置

新建Module

新建一个模块 `cloudalibaba-config-nacos-client3377`

改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>

```

```

<dependencies>
    <!--nacos-config-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
    </dependency>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--web + actuator-->
    <dependency>

```

写YML

这里写两个配置文件 `bootstrap.yml` 和 `application.yml`

为什么需要两个配置文件?

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

因此，全局配置需要放在bootstrap，自己本地配置放在application

bootstrap.yml

```

1  # nacos配置
2  server:
3      port: 3377
4  spring:
5      application:
6          name: nacos-config-client
7      cloud:

```

```
8     nacos:
9         discovery:
10            server-addr: localhost:8848 # nacos服务注册中心地址
11            config:
12                server-addr: localhost:8848 # nacos作为配置中心地址
13                file-extension: yaml # 指定配置的扩展名为yaml
14
15    #
16    ${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
```

application.yml

```
1  spring:
2    profiles:
3      active: dev # 表示开发环境
```

主启动

```
1  @SpringBootApplication
2  @EnableDiscoveryClient
3  public class NacosConfigClientMain3377 {
4      public static void main(String[] args) {
5          SpringApplication.run(NacosConfigClientMain3377.class, args);
6      }
7  }
```

业务类

```
1  @RestController
2  @RefreshScope // 在控制器类上加入@RefreshScope注解，使当前类下的配置支持Nacos的动态刷新
3  public class ConfigClientController {
4
5      @Value("${config.info}")
6      private String configInfo;
7
8      @GetMapping("/config/info")
9      public String getConfigInfo() {
10          return configInfo;
11      }
12  }
```

注意： `@RefreshScope` 注解。这是Spring Cloud原生注解，在SpringCloud Config中也是用它做动态刷新

SpringCloud 使用 `@RefreshScope` 注解，实现配置文件的动态加载！

`@RefreshScope` 注解仅需加在所有引入外部配置文件的类上！

比如当前这个controller类中就使用了@Value引入外部配置文件中的内容

在Nacos中添加配置信息

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

1. 理论

首先，我们可以看到在Nacos中新建配置需要一个Data ID

The screenshot shows the 'Create Configuration' page in the Nacos UI. Several input fields are highlighted with red boxes:

- * Data ID: A text input field.
- * Group: A dropdown menu set to 'DEFAULT_GROUP'.
- Configuration Format: A radio button group where 'TEXT' is selected.
- * Configuration Content: A large text area containing the configuration data.

The 'Data ID' field is mandatory, indicated by a red border and a red asterisk (*). The 'Group' dropdown is also highlighted with a red border. The 'Configuration Format' radio buttons are shown below, with 'TEXT' being the selected option. The 'Configuration Content' area is also highlighted with a red border.

从官网中可以看到，`dataId` 的完整格式如下：

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

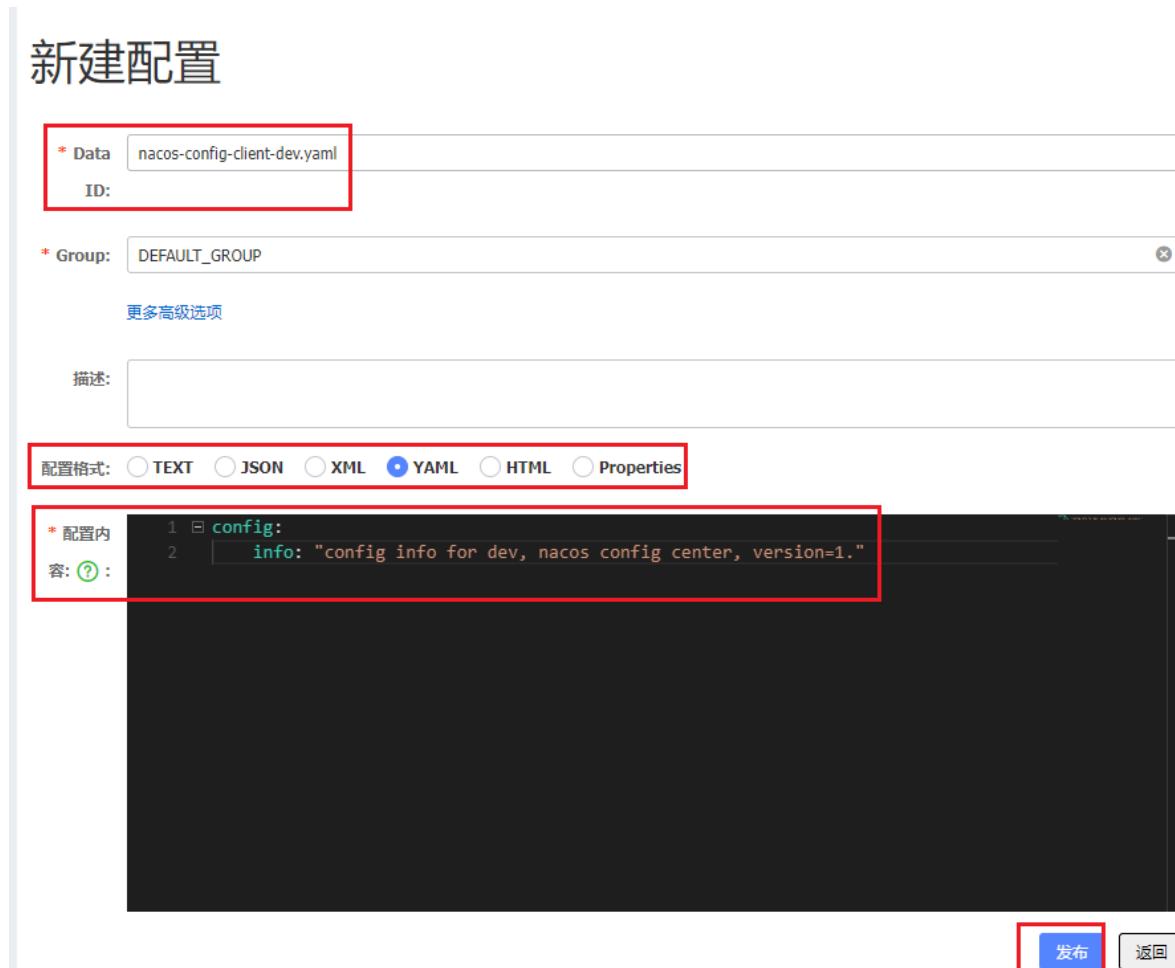
```
 ${prefix}-${spring.profiles.active}.${file-extension}
```

- `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.profiles.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.profiles.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。

2. 实操

根据上述的理论知识，我们可以知道我们需要创建的配置名为 `nacos-config-client-dev.yaml`

在nacos中新建配置，如下：



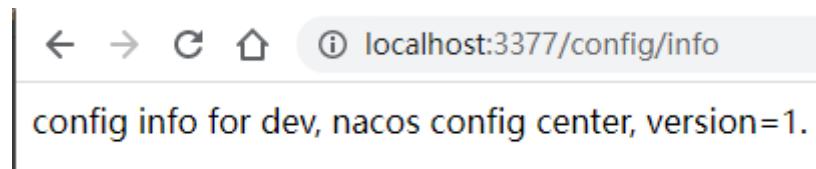
在发布后，查看配置列表就可以发现：

The screenshot shows the Nacos 1.1.4 configuration management interface. On the left sidebar, there are sections for Configuration Management, History Version, Listen Query, Service Management, and Service List. The main area is titled 'Configuration Management | public' and displays a table of configurations. A red box highlights the first row of the table, which contains the Data ID 'nacos-config-client-dev.yaml' and Group 'DEFAULT_GROUP'. To the right of the table are buttons for 'Delete', 'Export Selected Configuration', and 'Clone'. At the bottom, there are pagination controls for '每页显示: 10' (Page Size: 10), navigation arrows, and a page number '1'.

测试

- 启动前要保证nacos客户端-配置管理-配置列表中有对应的yaml配置文件
- 运行3377
- 调用接口: <http://localhost:3377/config/info>, 查看配置信息

可以读取到nacos配置文件中的配置!



自带动态刷新

- 修改Nacos中的yaml配置文件

我这里将version=1改成version=2

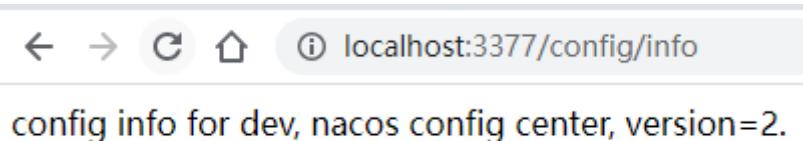
The screenshot shows the Nacos 1.1.4 configuration management interface. The configuration entry for 'nacos-config-client-dev.yaml' in 'DEFAULT_GROUP' has its 'Edit' button highlighted with a red box. The rest of the interface is identical to the previous screenshot.

修改后的内容如下:

```
1 config:
2   info: "config info for dev, nacos config center, version=2."
```

- 再次调用接口: <http://localhost:3377/config/info>

可以看到能立刻访问到最新配置。



Nacos作为配置中心-分类配置

问题

分布式开发中的多环境多项目管理

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？

以上两个问题，运用Nacos的分类管理功能即可解决！

Nacos的图形化管理界面

1. 配置管理

NACOS 1.1.4

配置管理

配置列表

历史版本

监听查询

服务管理

服务列表

订阅者列表

命名空间

集群管理

节点列表

public

配置管理 | public

查询结果：共查询到 1 条满足要求的配置。

Data ID: 模糊查询请输入Data ID

Group: 模糊查询请输入Group

查询

	Data Id	Group
	nacos-config-client-dev.yaml	DEFAULT_GROUP

删除 导出选中的配置 克隆

2. 命名空间

The screenshot shows the Nacos 1.1.4 web interface. The top navigation bar includes links for 首页 (Home), 文档 (Documentation), 博客 (Blog), 社区 (Community), En (English), and nacos. On the left, a sidebar menu is open under '配置管理' (Configuration Management) with options: 配置列表 (Configuration List), 历史版本 (History Versions), 监听查询 (Monitoring Query), 服务管理 (Service Management), 服务列表 (Service List), 订阅者列表 (Subscriber List), 命名空间 (Namespace) (which is highlighted with a red box), and 节点列表 (Node List). The main content area is titled '命名空间' (Namespace) and displays a table with one row. The table columns are: 命名空间名称 (Namespace Name), 命名空间ID (Namespace ID), 配置数 (Config Count), and 操作 (Operations). The single row contains 'public(保留空间)' (public(reserved space)) in the first column, an empty string in the second, '1 / 200' in the third, and '详情 (Detail) | 删除 (Delete) | 编辑 (Edit)' in the fourth. A blue button at the top right of the content area says '新建命名空间' (Create Namespace).

Namespace+Group+Data ID 三者关系? 为什么这么设计?

1. namespace、group、data id是什么?

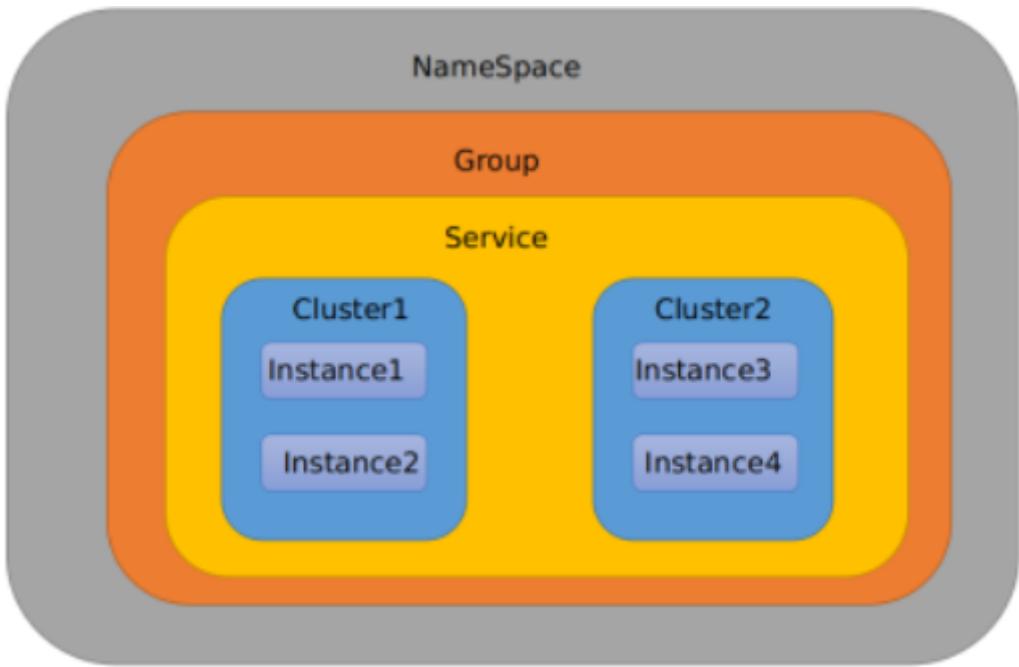
它们类似Java里面的package名和类名

最外层的namespace是可以用于区分部署环境的，group和data id 逻辑上区分两个目标对象。

官网 给的解释：

- **namespace**：用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。
- **group**：Nacos 中的一组配置集，是组织配置的维度之一。通过一个有意义的字符串（如 Buy 或 Trade）对配置集进行分组，从而区分 Data ID 相同的配置集。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP。配置分组的常见场景：不同的应用或组件使用了相同的配置类型，如 database_url 配置和 MQ_topic 配置。
- **Data ID**：Nacos 中的某个配置集的 ID。配置集 ID 是组织划分配置的维度之一。Data ID 通常用于组织划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 通常采用类 Java 包（如 com.taobao.tc.refund.log.level）的命名规则保证全局唯一性。此命名规则非强制。

2. 三者关系如下：



默认情况：

Namespace=public, Group=DEFAULT_GROUP, 默认Cluster是DEFAULT

Nacos默认的命名空间是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。

下面我们学习下三种方案加载配置。即：

- Data ID方案
- Group方案
- Namespace方案

Data ID 方案

演示内容：

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

1. 新建dev配置Data ID

这里我们就使用之前的配置文件

即：

The screenshot shows the Nacos Configuration Management interface. On the left sidebar, there are links for '配置管理' (Configuration Management), '配置列表' (Configuration List), '历史版本' (History Versions), '监听查询' (Monitoring Query), '服务管理' (Service Management), and '服务列表' (Service List). The main area has tabs for '配置管理' and 'public'. A search bar at the top says '模糊查询请输入Data ID'. Below it, a table lists configurations. One row is selected, with its Data ID 'nacos-config-client-dev.yaml' highlighted by a red box. The table columns are 'Data Id' and 'Group'. At the bottom of the table are buttons for '删除' (Delete), '导出选中的配置' (Export Selected Configuration), and '克隆' (Clone). To the right of the table, there's a '每页显示' (Items per page) dropdown set to 10.

	Data Id	Group
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP

2. 新建test配置Data ID

配置中主要就是Data ID和配置内容和dev中的不同

<

配置详情

配置管理

配置列表

历史版本

监听查询

服务管理

服务列表

订阅者列表

命名空间

集群管理

节点列表

* Data: nacos-config-client-test.yaml

ID:

* Group: DEFAULT_GROUP

更多高级选项

描述:

* MD5: be7fe09f0c5174d618921882a4e534da

* 配置
内容:
config:
info: "config info for test, nacos config center, version=2."

返回

3. 通过spring.profile.active属性就能进行多环境下配置文件的读取

```
main
└── java
    └── com
        └── atguigu
            └── springcloud
                └── controller
                    ├── ConfigClientController.java
                    └── NacosConfigClientMain3377.java
resources
└── application.yml
```

```
1   spring:
2   profiles:
3       active: test
4       # active: dev      # 表示开发环境
5
6
7
```

4. 测试

重启3377

然后访问: <http://localhost:3377/config/info>

可以显示

config info for test, nacos config center, version=2.

Group方案

这一节演示内容：

通过Group实现环境区分

1. 新建两个Group

新建 TEST_GROUP

新建配置

* Data: nacos-config-client-info.yaml
ID:

* Group: TEST_GROUP ×

[更多高级选项](#)

描述:

配置格式: TEXT JSON XML YAML HTML Properties

式:

* 配置:
内容:
② :

```
1 config:
  2   info: "nacos-config-client-info.yaml, TEST_GROUP, version=1."
```

发布

返回

新建 DEV_GROUP

新建配置

* Data: nacos-config-client-info.yaml
ID:

* Group: DEV_GROUP ×

[更多高级选项](#)

描述:

配置格: TEXT JSON XML YAML HTML Properties

式:

```
* 配置
内容:
① config:
② info: "nacos-config-client-info.yaml, DEV_GROUP, version=1."
```

发布 返回

可以看到这两个配置的Data ID一样，但是Group不一样

NACOS 1.1.4

配置管理 | public 查询结果: 共查询到 4 条满足要求的配置。

配置列表 Data ID: 模糊查询请输入Data ID Group: 模糊查

<input type="checkbox"/>	Data Id	Group
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP
<input type="checkbox"/>	nacos-config-client-test.yaml	DEFAULT_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	TEST_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	DEV_GROUP

删除 导出选中的配置 克隆 每页显示:

2. 在3377模块中修改 `bootstrap.yml` 和 `application.yml`

- 在 `bootstrap.yml` 中的 `config` 下增加一条group的配置即可。
- 在 `application.yml` 中将 `active` 的值改成 `info`

```

bootstrap.yml
4   spring:
5     application:
6       name: nacos-config-client
7     cloud:
8       nacos:
9         discovery:
10          server-addr: localhost:8848 # nacos
11        config:
12          server-addr: localhost:8848 # nacos
13          file-extension: yaml # 指定配置的扩展名
14        group: TEST_GROUP

application.yml
1   spring:
2     profiles:
3       active: info
4       # active: dev # 表示开发环境

```

3. 测试: 重启3377后, 访问: <http://localhost:3377/config/info>

← → C ⌂ ⓘ localhost:3377/config/info

nacos-config-client-info.yaml, TEST_GROUP, version=1.

4. 测试: 在 `bootstrap.yml` 中切换GROUP为`DEV_GROUP`, 重启3377后, 访问: <http://localhost:3377/config/info>

localhost:3377/config/info

nacos-config-client-info.yaml, DEV_GROUP, version=1.

Namespace方案

1. 新建dev/test的Namespace

NACOS 1.1.4

命名空间

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		4 / 200	详情 删除 编辑
test	706f5f53-127a-4fb6-b920-029321782f9e	0 / 200	详情 删除 编辑
dev	b5d448be-8418-4a97-8c1c-e89e4facb7be	0 / 200	详情 删除 编辑

2. 有了多个命名空间后，在服务列表和配置列表中都会出现多个命名空间

NACOS 1.1.4

public | test | dev

服务列表 | public

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
nacos-config-client	DEFAULT_GROUP	1	1	1	false

3. 按照域名配置填写

在默认命名空间中有三个配置文件

The screenshot shows the Nacos Configuration Management interface. At the top, there are tabs for 'public', 'test', and 'dev'. The 'public' tab is selected. Below the tabs, it says '配置管理 | public' and '查询结果: 共查询到 3 条满足要求的配置项'. There are search fields for 'Data ID' and 'Group'. A table below lists three configuration items:

	Data Id	Group
<input type="checkbox"/>	nacos-config-client-info.yaml	TEST_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	DEV_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	DEFAULT_GROUP

将三个配置文件克隆到dev中，并编辑配置文件以和上述文件区分。

4. 修改3377中的 `bootstrap.yml`

注意：namespace中的值是dev这个命名空间的ID

The screenshot shows a code editor with a file named 'bootstrap.yml'. The file content is as follows:

```
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 # nacos服务注册中心地址
      config:
        server-addr: localhost:8848 # nacos作为配置中心地址
        file-extension: yaml # 指定配置的扩展名为yaml
        group: DEV_GROUP
        namespace: b5d448be-8418-4a97-8c1c-e89e4facb7be
```

5. 访问：<http://localhost:3377/config/info>

可以看到访问到的是dev命名空间下的DEV_GROUP下的 `nacos-config-client-info.yaml` 配置文件

nacos-config-client-info.yaml, b5d448be-8418-4a97-8c1c-e89e4facb7be, DEV_GROUP, version=1.

Nacos集群和持久化配置(重要)

官网说明

官网 集群部署: <https://nacos.io/zh-cn/docs/cluster-mode-quick-start.html>

集群部署架构图:

(旧版)

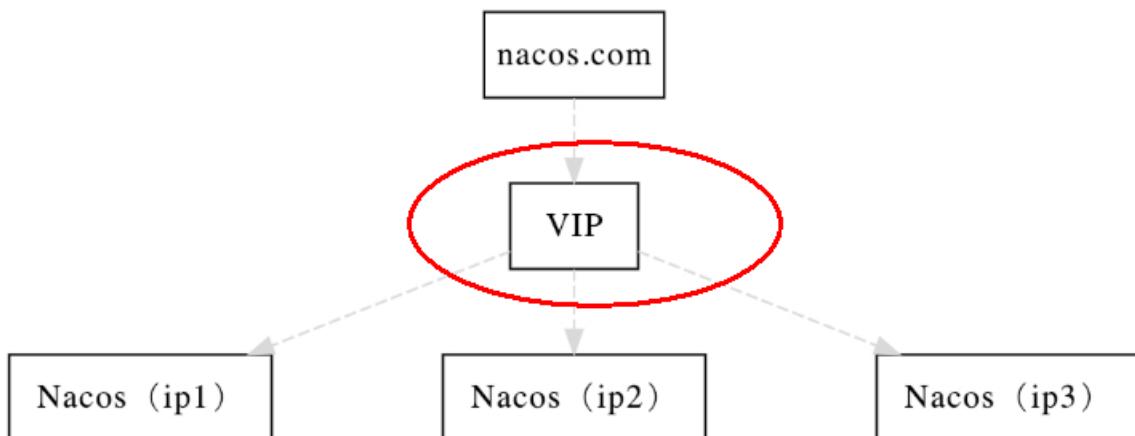
集群部署架构图

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



(新版)

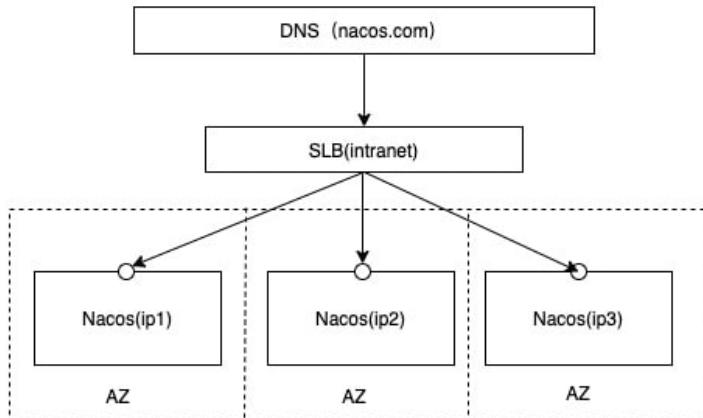
集群部署架构图

因此开源的时候推荐用户把所有服务放到一个vip下面，然后挂到一个域名下面

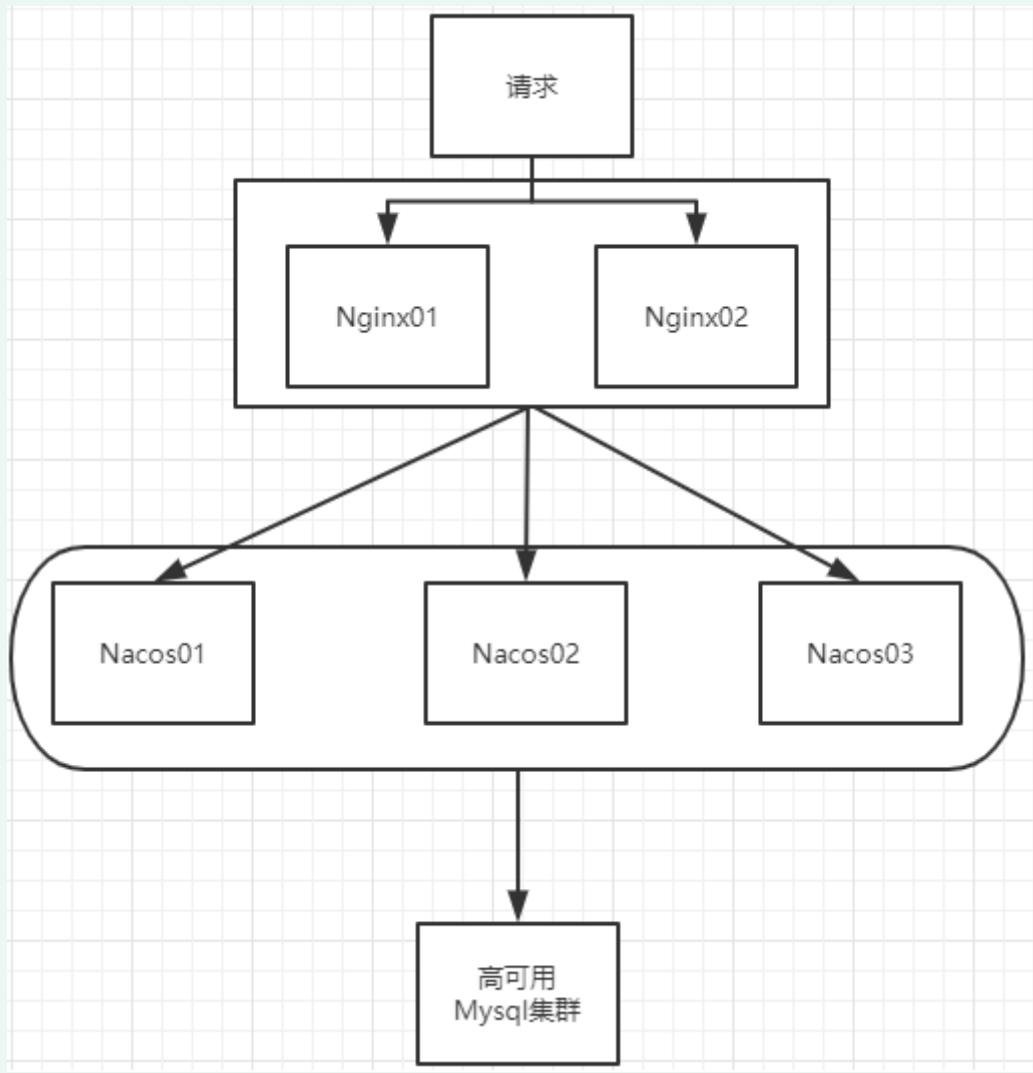
<http://ip1:port/openAPI> 直连ip模式，机器挂则需要修改ip才可以使用。

<http://SLB:port/openAPI> 挂载SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，直连SLB即可，下面挂server真实ip，可读性不好。

<http://nacos.com:port/openAPI> 域名 + SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，可读性好，而且换ip方便，推荐模式



真实部署架构大致如下图：



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。

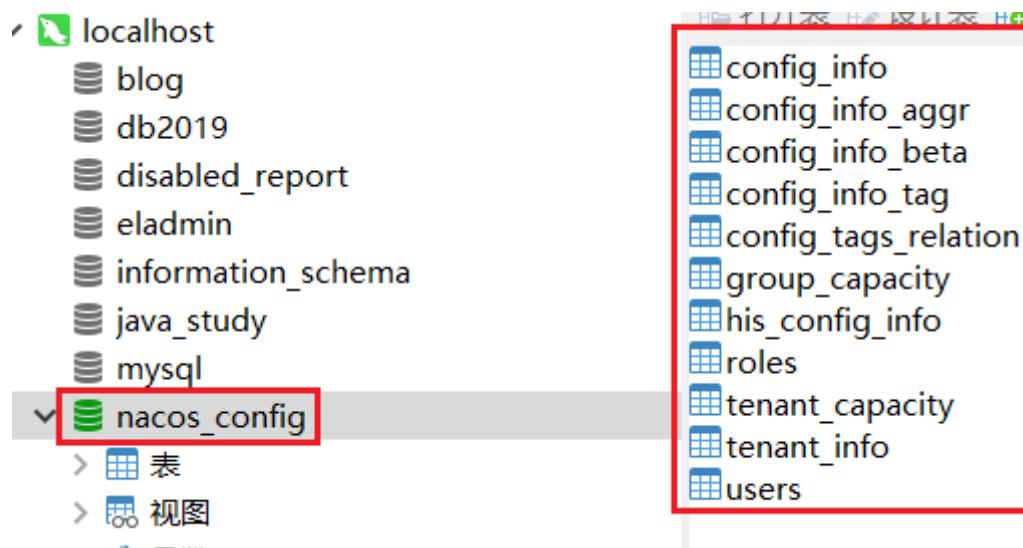
Nacos持久化配置解释

- Nacos默认自带的是嵌入式数据库derby

derby到mysql切换配置步骤

- 在 nacos-server-1.1.4\nacos\conf 目录下找到sql脚本，名为 nacos-mysql.sql
- 用mysql先创建一个名为 nacos_config 的数据库，然后执行该脚本

然后，在MySQL中就能看见配置信息表



- 在 nacos-server-1.1.4\nacos\conf 目录下找到 application.properties
- 参考[官网](#)修改配置，让其连接MySQL

```
***** Config Module Related Configurations *****  
### If use MySQL as datasource:  
# spring.datasource.platform=mysql  
  
### Count of DB:  
# db.num=1  
  
### Connect URL of DB:  
# db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?characterEncoding=utf8&connectTimeout=1000&socketT  
# db.user.0=nacos  
# db.password.0=nacos
```

配置如下：

```
1 spring.datasource.platform=mysql
2 db.num=1
3 db.url.0=jdbc:mysql://127.0.0.1:3306/nacos_config?
4   characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useUnicode=true&useSSL=false&serverTimezone=UTC
5 db.user=root
6 db.password=123456
```

5. 重启nacos，登录客户端

可以发现之前的配置文件不见了。因为我们用新的数据库MySQL

The screenshot shows the Nacos 1.1.4 web interface. The top navigation bar has 'NACOS' and '1.1.4'. The main menu on the left includes '配置管理', '服务管理', and '集群管理'. Under '配置管理', there are tabs for 'public' and 'private'. The 'public' tab shows a search bar for 'Data ID' and 'Group', both currently empty. Below the search bars is a table with columns 'Data Id', 'Group', and '归属应用'. A message '没有数据' (No data) is displayed. The 'private' tab also shows a search bar and a message '没有数据'.

Linux版Nacos+MySQL+Nginx生产环境配置

!!!务必看下官网教程：<https://nacos.io/zh-cn/docs/cluster-mode-quick-start.html>

根据架构设计，预计需要 1个Nginx+3个nacos注册中心+1个mysql

这里我用3台Linux虚拟机：

- 192.168.179.130：安装一台Nginx，1个nacos，1个mysql
- 192.168.179.131：安装1个nacos
- 192.168.179.132：安装1个nacos

假设安装的nacos放在 `/opt` 目录。

nginx默认安装路径是 `/usr/local/nginx`

mysql默认安装路径是 `/usr/bin/mysql`

```
1 # 如果想找出安装目录可以执行如下命令:
2 find / -name nginx
```

这里我下的各个软件版本如下：

- nginx版本为 1.20.1
- mysql版本为 5.7.38 , 用户为 root , 密码为 12345678

Nacos安装步骤

注意：在安装Nacos前一定要保证linux中安装了jdk8及以上

安装教程参考官网：<https://nacos.io/zh-cn/docs/cluster-mode-quick-start.html>

安装步骤如下：

1. 下载nacos到 /opt 目录

这里Nacos安装版本为 1.1.4

```
1 wget https://github.com/alibaba/nacos/releases/download/1.1.4/nacos-server-1.1.4.tar.gz
```



```
[root@node1 opt]# ll
total 53932
-rw-r--r--. 1 root root 25548 Apr  7 2017 mysql57-community-release-el7-10.noarch.rpm
-rw-r--r--. 1 root root 52115827 May 11 10:57 nacos-server-1.1.4.tar.gz
drwxr-xr-x. 9 1001 1001 186 May  9 16:07 nginx-1.20.1
-rw-r--r--. 1 root root 1061461 May 25 2021 nginx-1.20.1.tar.gz
drwxr-xr-x. 9 1169 1169 12288 May  9 16:06 pcre-8.35
-rw-r--r--. 1 root root 1996552 Apr  9 2014 pcre-8.35.tar.gz
```

2. 解压

```
1 tar -xvf nacos-server-1.1.4.tar.gz
```

3. 将nacos-1.1.4复制到 /mynacos 目录下。 (可选)

```
1 cp -r nacos-1.1.4 /mynacos/
```

MySQL数据库配置

Linux服务器上MySQL数据库配置。

在MySQL上需要使用nacos自带的sql脚本，用于创建nacos对应的数据库和表。

脚本位置在 `nacos/conf` 目录下，脚本名称为 `nacos-mysql.sql`

脚本部分内容如下：

```
/*
 *  数据库全名 = nacos_config
 *  表名称 = config_info
 */
CREATE TABLE `config_info` (
    `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'id',
    `data_id` varchar(255) NOT NULL COMMENT 'data_id',
    `group_id` varchar(255) DEFAULT NULL,
    `content` longtext NOT NULL COMMENT 'content',
    `md5` varchar(32) DEFAULT NULL COMMENT 'md5',
    `gmt_create` datetime NOT NULL DEFAULT '2010-05-05 00:00:00' COMMENT '创建时间',
    `gmt_modified` datetime NOT NULL DEFAULT '2010-05-05 00:00:00' COMMENT '修改时间',
    `src_user` text COMMENT 'source user',
    `src_ip` varchar(20) DEFAULT NULL COMMENT 'source ip',
    `app_name` varchar(128) DEFAULT NULL,
    `tenant_id` varchar(128) DEFAULT '' COMMENT '租户字段',
    `c_desc` varchar(256) DEFAULT NULL,
    `c_use` varchar(64) DEFAULT NULL,
    `effect` varchar(64) DEFAULT NULL,
    `type` varchar(64) DEFAULT NULL,
    `c_schema` text,
    PRIMARY KEY (`id`),
    UNIQUE KEY `uk_configinfo_datagrouptenant` (`data_id`, `group_id`, `tenant_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin COMMENT='config_info';
```

为了执行脚本，进行如下操作：

1. 进入mysql

```
1 mysql -u root -p
2 [password]
```

2. 创建数据库 `nacos_config`

```
1 create database nacos_config;
```

3. 使用创建的数据库 `nacos_config`

```
1 use nacos_config;
```

4. 执行sql脚本（注意：保证sql脚本在当前目录下）

```
1 source nacos-mysql.sql;
```

执行完毕后，`nacos_config` 数据库中就有如下表：

```
mysql> show tables;
+-----+
| Tables_in_nacos_config |
+-----+
| config_info
| config_info_aggr
| config_info_beta
| config_info_tag
| config_tags_relation
| group_capacity
| his_config_info
| roles
| tenant_capacity
| tenant_info
| users
+-----+
11 rows in set (0.00 sec)
```

application.properties配置文件修改

在 nacos/conf 目录下找到 `application.properties`，使用vim进行编辑：

在文件末尾加上如下内容即可

```
1  ### use MySQL as datasource:
2  spring.datasource.platform=mysql
3  db.num=1
4  db.url.0=jdbc:mysql://192.168.179.130:3306/nacos_config?
5    characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&
6    seUnicode=true&useSSL=false&serverTimezone=UTC
7  db.user=root
8  db.password=12345678
```

集群配置

在 nacos/conf 目录下，有文件 `cluster.conf.example`，将文件复制一份，重命名为 `cluster.conf`，这就是集群配置文件。

利用vim写入如下内容（节点ip:port）：

```
1  192.168.179.130:8848
2  192.168.179.131:8848
3  192.168.179.132:8848
```

启动nacos

在三台Linux都配置好上述文件后，将它们的nacos启动。

```
1  sh startup.sh
```

启动后，在浏览器上登录nacos客户端：<http://192.168.179.130:8848/nacos>

使用默认用户名密码登录后可以看到集群的节点列表：

节点Ip	节点状态	集群任期	Leader上时(ms)	心跳上时(ms)
192.168.179.130:8848	LEADER	1	11630	1000
192.168.179.131:8848	FOLLOWER	1	14198	1000
192.168.179.132:8848	FOLLOWER	1	15664	500

Nginx配置

配置Nginx，由它作为负载均衡器

- 修改Nginx配置文件 `/usr/local/nginx/conf/nginx.conf`

部分内容修改成如下形式：

```
#gzip on;

upstream cluster{
    server 192.168.179.130:8848;
    server 192.168.179.131:8848;
    server 192.168.179.132:8848;
}

server {
    listen 1111;
    server_name localhost;

    #charset koi8-r;

    #access_log logs/host.access.log main;

    location / {
        #root html;
        #index index.html index.htm;
        proxy_pass http://cluster;
    }
}
```

- 启动nginx

```
1 ./nginx
```

测试

按照上述配置配置好mysql、nacos、nginx后并启动它们。

在浏览器上访问：<http://192.168.179.130:1111/nacos>

节点Ip	节点状态	集群任期	Leader上时(ms)	心跳上时(ms)
192.168.179.130:8848	LEADER	1	15868	4000
192.168.179.131:8848	FOLLOWER	1	14426	4000
192.168.179.132:8848	FOLLOWER	1	16020	3500

测试nacos作为服务注册中心

- 利用子工程9002，修改其中的 `application.yml`

```
1 server:
2   port: 9002
3 spring:
4   application:
5     name: nacos-payment-provider
6   cloud:
7     nacos:
8       discovery:
9         server-addr: 192.168.179.130:1111
10  management:
11    endpoints:
12      web:
13        exposure:
14          include: '*'
```

主要注意如下配置是Nginx的地址，因为Nginx作为代理了，监听了1111端口

```
1 server-addr: 192.168.179.130:1111
2
3启动9002
4访问nacos的客户端：http://192.168.179.130:1111/nacos
```

成功注册

Nacos 1.1.4

public

服务列表 | public

服务名称: 请输入服务名称 分组名称: 请输入分组名称 隐藏空服务: 查询

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
nacos-payment-provider	DEFAULT_GROUP	1	1	1	false

配置管理
配置列表
历史版本
监听查询
服务管理
服务列表
订阅者列表

十七、SpringCloud Alibaba Sentinel实现熔断与限流

为什么用Sentinel?

由于Hystrix有以下缺点：

1. 需要人工搭建监控平台
2. 没有一套web界面可以给我们进行更加细粒度化的配置（如，流量控制、速率控制、服务熔断、服务降级、……）

Sentinel就将以上缺点解决了！Sentinel拥有的功能如下：

1. Sentinel是单独一个组件，可以独立出来
2. Sentinel有直接界面化的细粒度统一配置

Sentinel 简介

必看！！！官网：<https://github.com/alibaba/Sentinel>

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

A powerful flow control component enabling reliability, resilience and monitoring for microservices.
(面向云原生微服务的高可用流控防护组件)

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**。

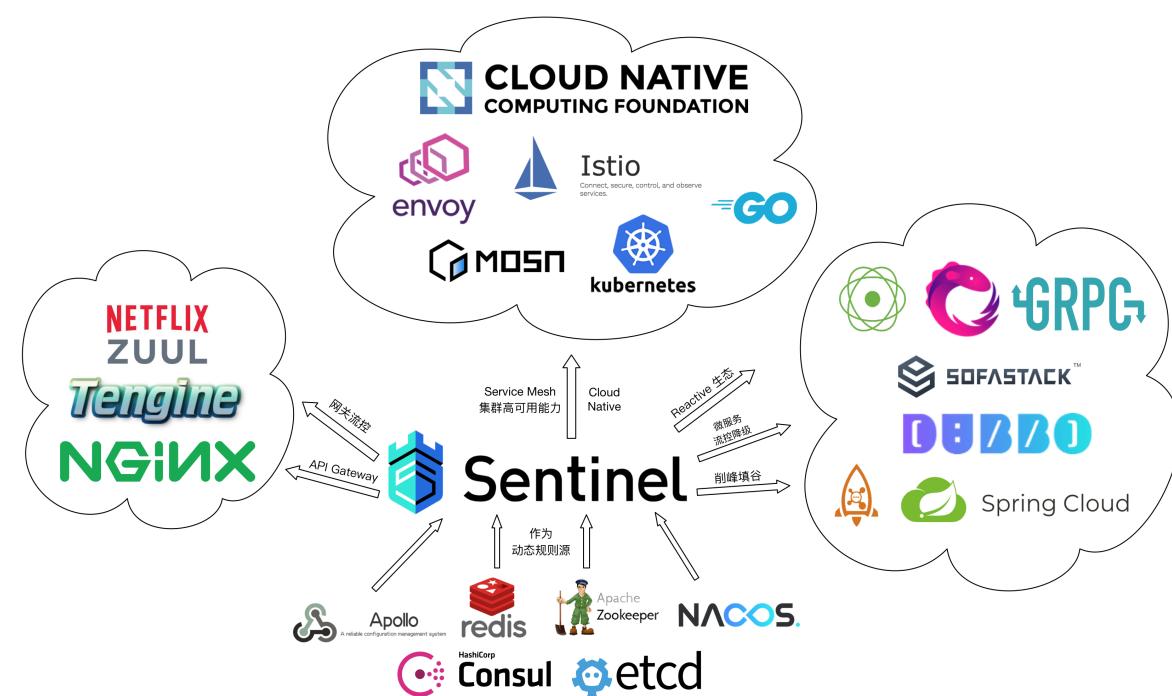
Sentinel 具有以下特征：

- **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、Apache Dubbo、gRPC、Quarkus 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。同时 Sentinel 提供 Java/Go/C++ 等多语言的原生实现。
 - **完善的 SPI 扩展机制**: 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:



Sentinel 的开源生态:

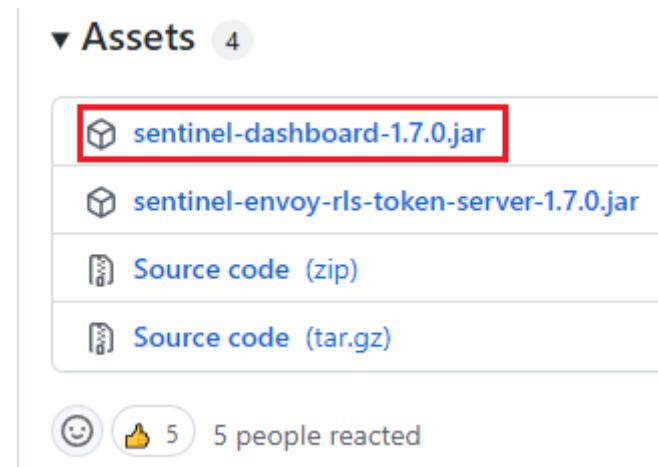


Sentinel 下载安装运行

下载地址: <https://github.com/alibaba/Sentinel/releases>

选择合适的版本下载即可。

这里我下载1.7.0版本的



Sentinel使用

Sentinel官方文档: <https://github.com/alibaba/spring-cloud-alibaba/wiki/Sentinel-en>

Sentinel示例: <https://github.com/alibaba/spring-cloud-alibaba/blob/2.2.x/spring-cloud-alibaba-examples/sentinel-example/sentinel-core-example/readme-zh.md>

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

安装Sentinel控制台

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。

Sentinel可以分为前台8080 和 后台。

上一小节下载的jar包就是后台。

运行后台的前提: 搭建了Java8环境、8080端口不能被占用。

使用以下命令即可启动

```
1 java -jar sentinel-dashboard-1.7.0.jar
```

启动后, 访问Sentinel管理界面: <http://localhost:8080/>

登录账号密码均为 **sentinel**



初始化演示工程

我们要演示的内容：

1. 微服务能够注册进Nacos
2. Sentinel可以监控微服务，并对微服务进行流控、降级等操作

启动Nacos

在本地启动Nacos。保证访问：<http://localhost:8848/nacos>，可以成功。

注意：Nacos的数据源使用MySQL

新建微服务8401

新建Module

新建 `cloudalibaba-sentinel-service8401`

改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     <!--SpringCloud alibaba sentinel-datasource-nacos 后续做持久化用到-->
8     <dependency>
9         <groupId>com.alibaba.csp</groupId>
10        <artifactId>sentinel-datasource-nacos</artifactId>
11    </dependency>
12    <!--SpringCloud alibaba sentinel -->
13    <dependency>
14        <groupId>com.alibaba.cloud</groupId>
```

```

15         <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
16     </dependency>
17     <!--openfeign-->
18     <dependency>
19         <groupId>org.springframework.cloud</groupId>
20         <artifactId>spring-cloud-starter-openfeign</artifactId>
21     </dependency>
22     <!-- SpringBoot整合Web组件+actuator -->
23     <dependency>
24         <groupId>org.springframework.boot</groupId>
25         <artifactId>spring-boot-starter-web</artifactId>
26     </dependency>
27     <dependency>
28         <groupId>org.springframework.boot</groupId>
29         <artifactId>spring-boot-starter-actuator</artifactId>
30     </dependency>
31     <!--日常通用jar包配置-->
32     <dependency>
33         <groupId>org.springframework.boot</groupId>
34         <artifactId>spring-boot-devtools</artifactId>
35         <scope>runtime</scope>
36         <optional>true</optional>
37     </dependency>
38     <dependency>
39         <groupId>cn.hutool</groupId>
40         <artifactId>hutool-all</artifactId>
41         <version>4.6.3</version>
42     </dependency>
43     <dependency>
44         <groupId>org.projectlombok</groupId>
45         <artifactId>lombok</artifactId>
46         <optional>true</optional>
47     </dependency>
48     <dependency>
49         <groupId>org.springframework.boot</groupId>
50         <artifactId>spring-boot-starter-test</artifactId>
51         <scope>test</scope>
52     </dependency>
53 </dependencies>

```

写YML

```

1 server:
2   port: 8401
3 spring:
4   application:
5     name: cloudalibaba-sentinel-service
6   cloud:
7     nacos:
8       discovery:
9         server-addr: localhost:8848
10    sentinel:
11      transport:
12        dashboard: localhost:8080
13        # 服务和sentinel发生交互的port，默认是8719。如果被其他应用占用，则会自动从8719开始
           依次+1扫描，直到找到未被占用的端口

```

```
14         port: 8719
15
16     management:
17         endpoints:
18             web:
19                 exposure:
20                     include: '*'
```

主启动

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class MainApp8401 {
4     public static void main(String[] args) {
5         SpringApplication.run(MainApp8401.class, args);
6     }
7 }
```

业务类

```
1 @RestController
2 public class FlowLimitController {
3
4     @GetMapping("/testA")
5     public String testA() {
6         return "-----testA";
7     }
8
9     @GetMapping("/testB")
10    public String testB() {
11        return "-----testB";
12    }
13 }
```

测试

1. 启动Sentinel8080

```
1 java -jar sentinel-dashboard-1.7.0.jar
```

2. 启动微服务8401
3. 查看sentinel控制台 <http://localhost:8080>

发现并没有发现8401服务

Sentinel 控制台 1.7.0

应用名

搜索

① 首页

欢迎使用 Sentinel 控制台

这是因为 Sentinel 采用的是懒加载机制

因此，需要我们执行一次8401服务上的接口，之后，Sentinel就可以监控到了。

4. 依次访问：<http://localhost:8401/testA> , <http://localhost:8401/testB>
5. 再次观察Sentinel控制台。可以发现就有了对该服务的监控

Sentinel 控制台 1.7.0

注销

应用名 搜索

① 首页

cloudalibaba-sentinel-service (1/1) ▼

实时监控

链点链路

流控规则

降级规则

热点规则

系统规则

授权规则

集群流控

机器列表

cloudalibaba-sentinel-service

实时监控

/testB

通过 QPS 拒绝 QPS

时间	通过 QPS	拒绝 QPS	响应时间 (ms)
13:42:48	1.0	0.0	3.0
13:41:27	1.0	0.0	4.0
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

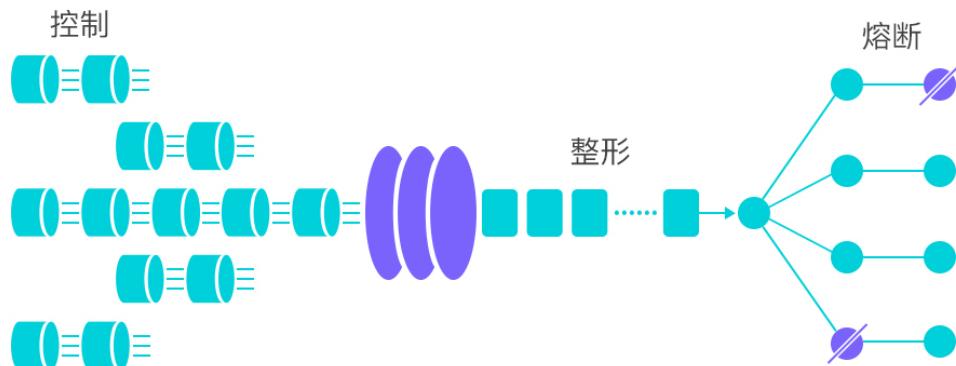
可以看到，目前sentinel8080正在监控微服务8401

流控规则

官方文档： <https://github.com/alibaba/Sentinel/wiki/%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6>

基本介绍

流量控制在网络传输中是一个常用的概念，它用于调整网络包的发送数据。然而，从系统稳定性角度考虑，在处理请求的速度上，也有非常多的讲究。任意时间到来的请求往往是随机不可控的，而系统的处理能力是有限的。我们需要根据系统的处理能力对流量进行控制。Sentinel 作为一个调配器，可以根据需要把随机的请求调整成合适的形状，如下图所示：



流量控制有以下几个角度：

- 资源的调用关系，例如资源的调用链路，资源和资源之间的关系；
- 运行指标，例如 QPS、线程池、系统负载等；
- 控制的效果，例如直接限流、冷启动、排队等。

Sentinel 的设计理念是让您自由选择控制的角度，并进行灵活组合，从而达到想要的效果。

我们可以在Sentinel控制台添加流控规则。如下图所示。

新增流控规则

资源名	资源名
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
	单机阈值
是否集群	<input checked="" 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 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

新增 取消

解释上图流程规则中的内容（也可以参考官网 <https://sentinelguard.io/zh-cn/docs/basic-api-resource-rule.html>）：

- 资源名 (resource) : 唯一名称, 默认请求路径
- 针对来源 (limitApp) : Sentinel可以针对调用者进行限流, 填写微服务名。默认是 default (不区分来源)
- 阈值类型/单机阈值:
 - QPS (每秒的请求数量) : 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
- 是否需要集群: 默认不需要集群
- 流控模式 (strategy) :
 - 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 (controlBehavior) :
 - 快速失败: 直接失败, 抛异常
 - Warm Up: 根据codeFactor (冷加载因子, 默认3) 的值, 从阈值/codeFactor, 经过预热时长, 才达到设定的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。

流控模式

流控模式有三种:

- 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]

下面我们来演示这三种模式。

直接

1. 在sentinel8080控制台中给 `/testA` 接口新增流控规则

新增流控规则

资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 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 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	
新增并继续添加 新增 取消	

2. 新增流控规则后，可以查看到目前已有的流控规则

Sentinel 控制台 1.7.0

应用名	搜索
<input type="radio"/> 首页	
cloudalibaba-sentinel-service	(1/1)▼
<input type="radio"/> 实时监控	
<input type="radio"/> 簇点链路	
<input checked="" type="radio"/> 流控规则	
<input type="radio"/> 降级规则	
<input type="radio"/> 热点规则	
<input type="radio"/> 系统规则	
<input type="radio"/> 授权规则	
<input type="radio"/> 集群流控	
<input type="radio"/> 机器列表	

cloudalibaba-sentinel-service

[+ 新增流控规则](#)

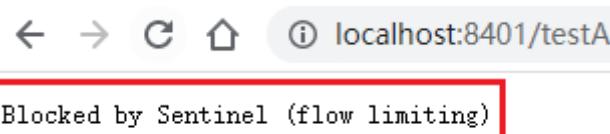
流控规则	10.1.125.73:8720	关键字	刷新				
资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
/testA	default	直接	QPS	1	单机	快速失败	编辑 删除

共 1 条记录, 每页 10 条记录

3. 访问 <http://localhost:8401/testA>

当QPS没有达到阈值，即1s中最多访问一次，则可以正常访问。如果QPS达到阈值，则会显示如下页面：

报错 (Blocked by Sentinel (flow limiting))



思考：上述达到阈值后的报错信息是系统自带的，那么是否可以自定义报错信息？即达到阈值后的后续处理由自己定义，而不是交给系统

演示 阈值类型变成 线程数。

这个线程数，相当于并行线程数

并发数控制用于保护业务线程池不被慢调用耗尽。

Sentinel 并发控制 不负责创建和管理线程池，而是简单统计当前请求上下文的线程数目（正在执行的调用数目）；

如果超出阈值，新的请求会被立即拒绝，效果类似于信号量隔离；



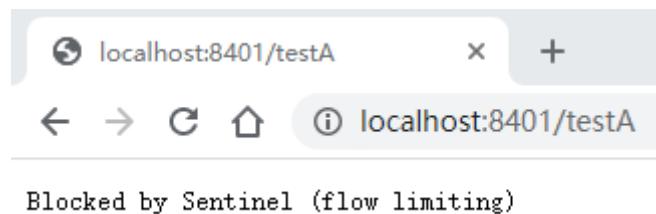
这说明微服务8401只允许有1个线程接收请求。并行线程数最多1个。

为了演示效果，我们把 /testA 接口的代码修改成如下：

```
@RestController
public class FlowLimitController {

    @GetMapping("/testA")
    public String testA() {
        // 暂定毫秒
        try {
            Thread.sleep( millis: 800);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "-----testA";
    }
}
```

重启8401微服务。用多次访问：<http://localhost:8401/testA>



关联

关联：当关联的资源达到阈值时，就限流自己

如下代码。目前 /testA 方法和 /testB 方法在同一个controller中。那么两个方法相关联。那么比如，当访问B的资源达到阈值后，就会限流A。

```
@RestController  
public class FlowLimitController {  
  
    @GetMapping("/testA")  
    public String testA() {  
        return "-----testA";  
    }  
  
    @GetMapping("/testB")  
    public String testB() {  
        return "-----testB";  
    }  
}
```

下面演示：

下面流控规则表示，当访问 /testB 方法请求达到1这个阈值后，/testA 方法会被限流。

编辑流控规则

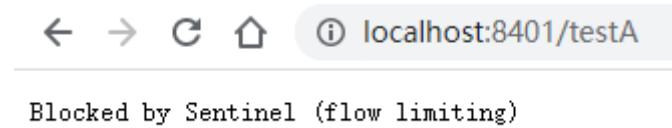
资源名	/testA
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	1
是否集群	<input type="checkbox"/>
流控模式	<input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路
关联资源	/testB
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm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

然后用jmeter模拟并发密集的访问：<http://localhost:8401/testB>

这里设置每秒2个线程，循环20次。

在启动jmeter之前，先访问A：<http://localhost:8401/testA>。可以正常访问。

启动jmeter后，再访问A： <http://localhost:8401/testA>。可以看到，拒绝访问。A被限流。



链路

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）

流控效果

可以看官方文档：<https://github.com/alibaba/Sentinel/wiki/%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6>

流控效果（controlBehavior）：

- 快速失败：直接失败，抛异常
- Warm Up（预热）：根据codeFactor（冷加载因子，默认3）的值，从阈值/codeFactor，经过预热时长，才达到设定的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

快速失败

在流控模式中，我们已经了解了快速失败的效果，就是直接抛出异常（Blocked by Sentinel (flow limiting)）

它的源码来自于：com.alibaba.csp.sentinel.slots.block.flow.controller.DefaultController

下面我们来讲预热和排队等待的效果。

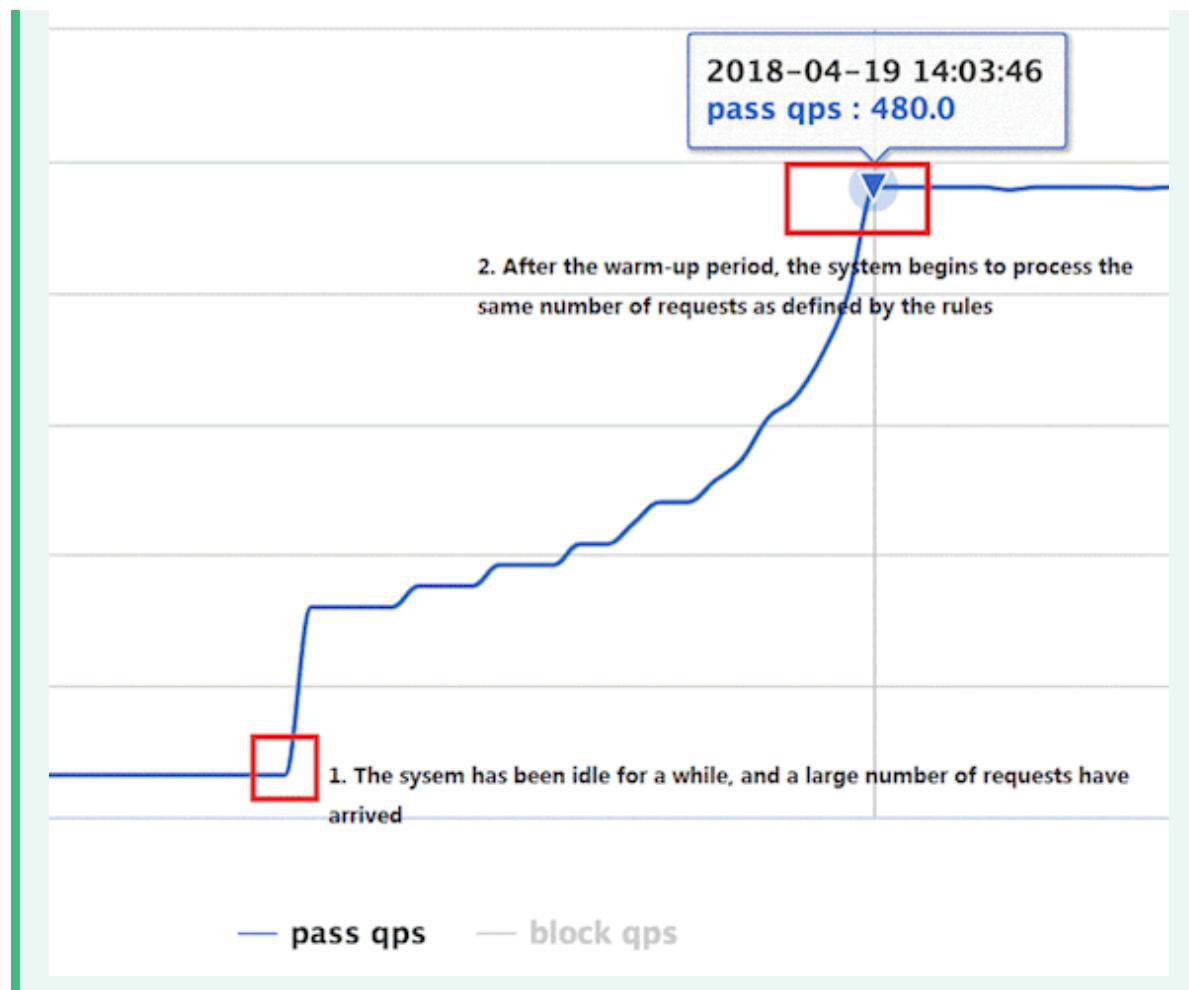
预热

Warm Up（预热）：根据codeFactor（冷加载因子，默认3）的值，从阈值/codeFactor，经过预热时长，才达到设定的QPS阈值。

即，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。

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：



源码：

```
com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController
```

从源码中可以看到默认 coldFactor 的值为3

```
↓ public class WarmUpController implements TrafficShapingController {

    protected double count;
    private int coldFactor;
    protected int warningToken = 0;
    private int maxToken;
    protected double slope;

    protected AtomicLong storedTokens = new AtomicLong(initialValue: 0);
    protected AtomicLong lastFilledTime = new AtomicLong(initialValue: 0);

    public WarmUpController(double count, int warmUpPeriodInSec, int coldFactor) {
        construct(count, warmUpPeriodInSec, coldFactor);
    }

    public WarmUpController(double count, int warmUpPeriodInSec) {
        construct(count, warmUpPeriodInSec, coldFactor: 3);
    }
}
```

演示：

1. 新增流控规则

新增流控规则

资源名	/testB
针对来源	default
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数
单机阈值	10
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 checked="" type="radio"/> Warm Up <input type="radio"/> 排队等待
预热时长	5

[关闭高级选项](#)

[新增](#) [取消](#)

2. 连续多次访问 <http://localhost:8401/testB>

从实时监控中可以看到，当请求数突然增大，在QPS达到3左右时就有部分请求被拒绝，而到预热完成后，即使QPS达到6，也没有拒绝。

这是因为设置的单机阈值 **threshold** 是10，预热时长为5。这样 QPS 的真实阈值会从 **threshold / 3** 开始，经过与预热时长5s后，真实阈值才达到10.



预热方式的应用场景：

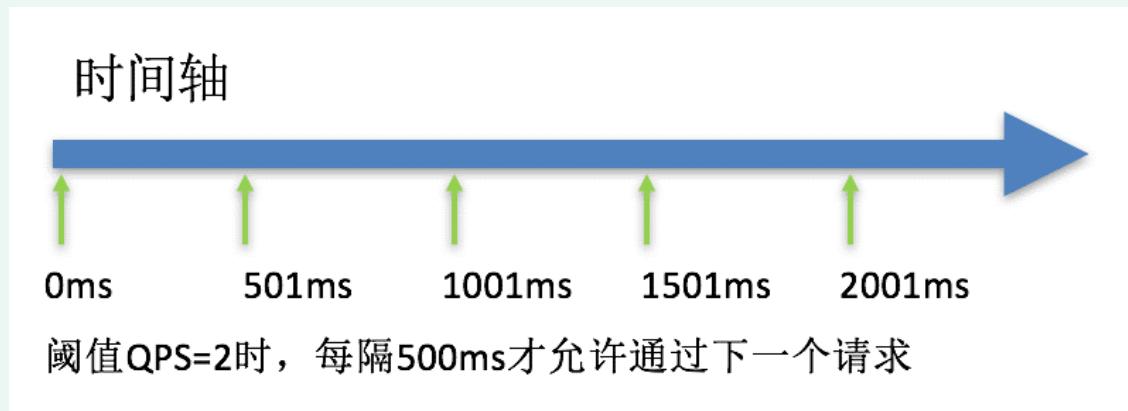
秒杀系统在开启的瞬间，会有很多流量上来，很有可能把系统打死，预热方式就是为了保护系统，可慢慢的把流量放进来，慢慢的把阀值增长到设置的阀值。

排队等待

排队等待，也叫匀速排队。

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请求。

注意：

- 阈值必须设置为QPS
- 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：

1. 设置一条流控规则。规则表示1秒匀速的通过10个请求，也就是每个请求平均间隔恒定为 `1000 / 10 = 100 ms`，每个请求的最长等待时间设置为 `20s`

编辑流控规则

资源名 /testB

针对来源 default

阈值类型 QPS 线程数 单机阈值 10

是否集群

流控模式 直接 关联 链路

流控效果 快速失败 Warm Up 排队等待

超时时间 20000

[关闭高级选项](#)

[保存](#) [取消](#)

2. 使用Jmeter测试，设置1秒发送200个请求，循环10次。

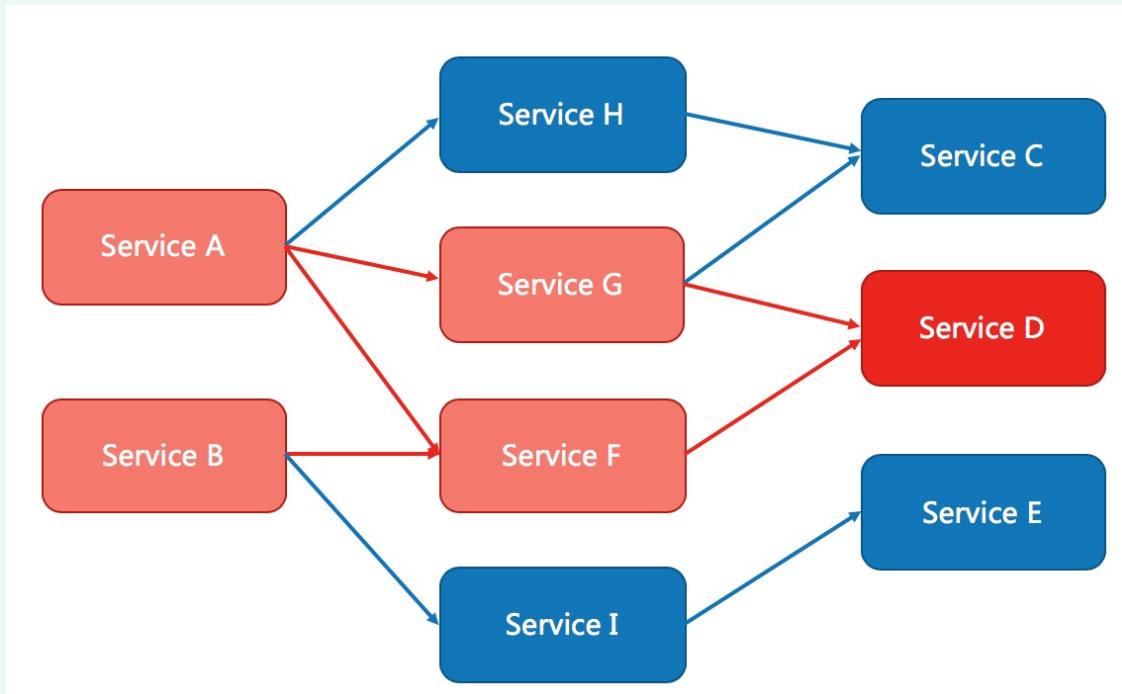
就能在sentinel控制台中看到如下实时监控。可以看到每秒匀速通过QPS=10。



降级规则

官方文档：<https://github.com/alibaba/Sentinel/wiki/%E7%86%94%E6%96%AD%E9%99%8D%E7%BA%A7>

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。**

注意：本文档针对 Sentinel 1.8.0 及以上版本。1.8.0 版本对熔断降级特性进行了全新的改进升级，请使用最新版本以更好地利用熔断降级的能力。

基本介绍

在sentinel控制台中新增降级规则的步骤如下：

cloudalibaba-sentinel-service

+ 新增降级规则

实时监控

簇点链路

流控规则

降级规则

热点规则

系统规则

授权规则

集群流控

机器列表

(s) 操作

新增 1 / 1 页 取消

上述的降级策略（新版本改为熔断策略）如下：

- **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，默认1s) 内请求数目大于设置的最小请求数目（默认为5），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），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

注：在我目前安装的Sentinel 1.7.0中，这个慢调用比例策略叫 RT 策略。

- **异常比例 (ERROR_RATIO)**：当单位统计时长 (statIntervalMs，默认1s) 内请求数目大于设置的最小请求数目（默认是5），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%。
- **异常数 (ERROR_COUNT)**：当单位统计时长（默认1s）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。

熔断降级规则 (DegradeRule) 包含下面几个重要的属性：

Field	说明	默认值
resource	资源名，即规则的作用对象	
grade	熔断策略，支持慢调用比例/异常比例/异常数策略	慢调用比例
count	慢调用比例模式下为慢调用临界 RT（超出该值计为慢调用）；异常比例/异常数模式下为对应的阈值	
timeWindow	熔断时长，单位为 s	
minRequestAmount	熔断触发的最小请求数，请求数小于该值时即使异常比率超出阈值也不会熔断（1.7.0 引入）	5
statIntervalMs	统计时长（单位为 ms），如 $60 * 1000$ 代表分钟级（1.8.0 引入）	1000 ms
slowRatioThreshold	慢调用比例阈值，仅慢调用比例模式有效（1.8.0 引入）	

注意：在sentinel1.7.0中断路器没有半开（HALF-OPEN）状态，但是在sentinel1.8.0及以上版本中断路器是有半开状态的！！！

演示熔断降级策略

注：由于本次使用的Sentinel1.7.0，所以没办法测试慢调用比例这个策略

所以下面演示的策略有：

- RT
- 异常比例
- 异常数

RT

1. 为了演示RT策略，我们在controller中新增一个接口

```
public class FlowLimitController {  
    @GetMapping("testA")  
    public String testA() { return "-----testA"; }  
    @GetMapping("testB")  
    public String testB() {...}  
    @GetMapping("testD")  
    public String testD() {  
        // 暂停几秒钟  
        try {  
            Thread.sleep( millis: 1000 );  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        log.info("testD 测试RT");  
        return "-----testD";  
    }  
}
```

2. 配置一个降级规则

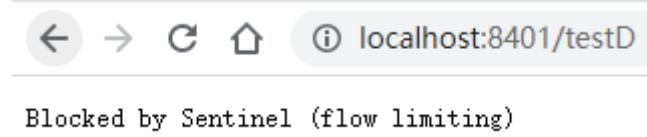


3. 利用jmeter进行压测

jmeter设置为每秒发送10个请求给 <http://localhost:8401/testD>



4. 在浏览器中访问: <http://localhost:8401/testD>



5. 观察sentinel控制台的实时监控



结论:

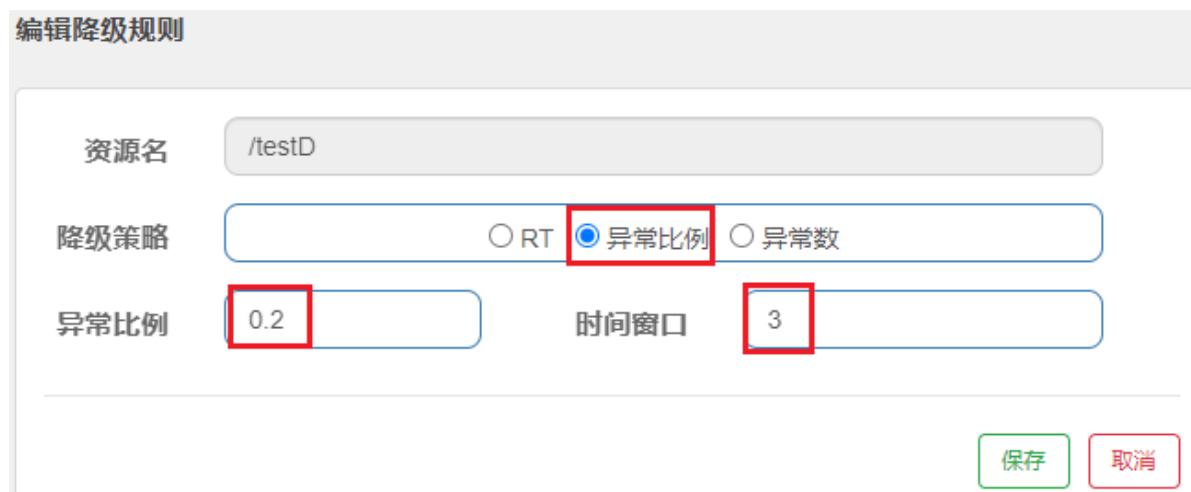
按照上述配置, 永远1秒钟进来10个请求调用testD, 而在熔断规则中我们希望200毫秒能处理完本次任务, 如果超过200毫秒还没处理完, 那么在未来1秒钟的时间窗口内, 断路器打开, 微服务不可用。

异常比例

- 修改演示的controller对应接口

```
@GetMapping("/testD")
public String testD() {
    log.info("testD 测试异常比例");
    int age = 10 / 0;
    return "-----testD";
}
```

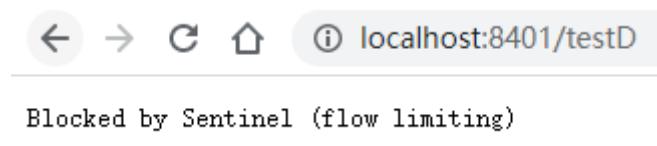
- 修改降级规则



- 利用jmeter进行测试。设置为每秒10个请求 <http://localhost:8401/testD>，并永远循环。



- 访问：<http://localhost:8401/testD>



- 查看sentinel实时监控



注意：

异常比例 (ERROR_RATIO): 当单位统计时长 (`statIntervalMs`, 默认1s) 内请求数目大于设置的最小请求数目 (默认是5)，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%。

上述说明在异常比例策略中，要使熔断器熔断，必须同时满足两个条件：

1. 单位统计时长内的请求数需要大于设置的最小请求数 (默认是5)
2. 异常的比例大于阈值

注意，一定要两个条件都满足！！！

异常数

异常数 (ERROR_COUNT): 当单位统计时长 (默认1s) 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。

演示：

1. 新增演示的controller对应接口

```

@GetMapping("/testE")
public String testE() {
    log.info("testE 测试异常数");
    int age = 10 / 0;
    return "-----testE";
}

```

2. 配置一个降级规则

新增降级规则

资源名: /testE

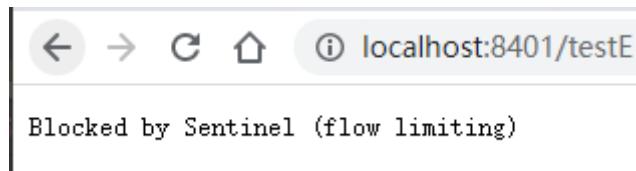
降级策略: RT 异常比例 异萂数

异常数: 5

时间窗口: 61

新增 取消

3. 直接在浏览器中访问五次以上: <http://localhost:8401/testE>



热点key限流

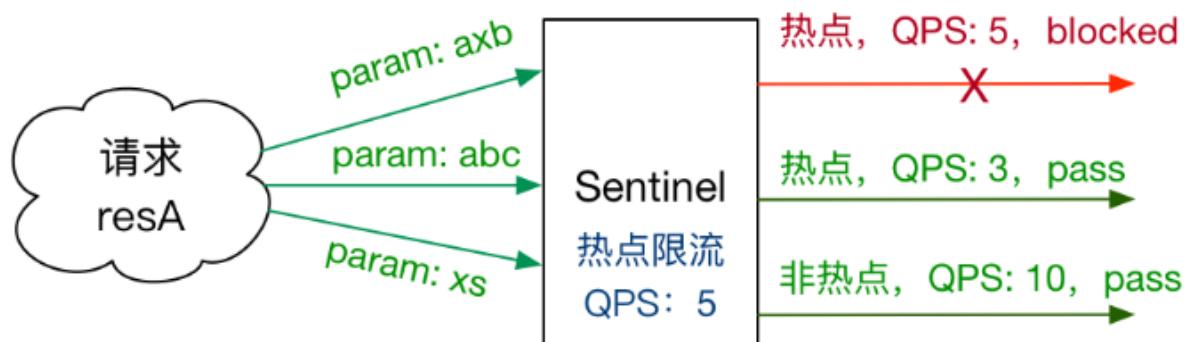
官方文档: <https://github.com/alibaba/Sentinel/wiki/%E7%83%AD%E7%82%B9%E5%8F%82%E6%95%BF%E9%99%90%E6%B5%81>

基本介绍

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。

注：热点限流的源码可以看下：[com.alibaba.csp.sentinel.slots.block.BlockException](#)

回顾

之前我们在 [流控规则](#) 里思考过一个问题：服务熔断后的兜底方法是直接报错（Blocked by Sentinel (flow limiting)），那么这个错误信息是否可以自定义？类似Hystrix，某个方法出问题了，就找到对应的兜底降级方法？

结论：

是可以的。在Hystrix中，我们在客户端会用到注解 `@HystrixCommand`，而现在在Sentinel中，需要用到注解 `@SentinelResource`

参考官网文档：<https://github.com/alibaba/Sentinel/wiki/%E5%A6%82%E4%BD%95%E4%BD%BF%E7%94%A8>

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

```
1 // 原本的业务方法.
2 @SentinelResource(blockHandler = "blockHandlerForGetUser")
3 public User getUserId(String id) {
4     throw new RuntimeException("getUserById command failed");
5 }
6
7 // blockHandler 函数，原方法调用被限流/降级/系统保护的时候调用
8 public User blockHandlerForGetUser(String id, BlockException ex) {
9     return new User("admin");
10 }
```

注意 `blockHandler` 函数会在原方法被限流/降级/系统保护的时候调用，而 `fallback` 函数会针对所有类型的异常。请注意 `blockHandler` 和 `fallback` 函数的形式要求，更多指引可以参见 [Sentinel注解支持文档](#)。

关于 `@SentinelResource` 注解的具体讲解请见后续。

演示代码

1. 在8401模块的Controller类中，添加了如下代码：

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

public String deal_testHotKey(String p1, String p2, BlockException ex) {
    return "-----deal_testHotKey~~~";
}
```

注：这里其实就是参考了官方文档：<https://github.com/alibaba/Sentinel/wiki/%E5%A6%82%E4%BD%95%E4%BD%BF%E7%94%A8#%E7%83%AD%E7%82%B9%E8%A7%84%E5%88%99-paramflowrule>

方式四：注解方式定义资源

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

```
// 原本的业务方法。  
@SentinelResource(blockHandler = "blockHandlerFor GetUser")  
public User getUserById(String id) {  
    throw new RuntimeException("getUserById command failed");  
}  
  
// blockHandler 函数，原方法调用被限流/降级/系统保护的时候调用  
public User blockHandlerFor GetUser(String id, BlockException ex) {  
    return new User("admin");  
}
```

注意 `blockHandler` 函数会在原方法被限流/降级/系统保护的时候调用，而 `fallback` 函数会针对所有类型的异常。请注意 `blockHandler` 和 `fallback` 函数的形式要求，更多指引可以参见 [Sentinel 注解支持文档](#)。

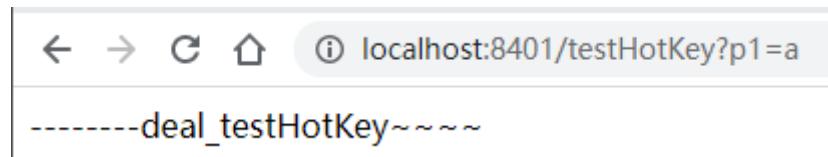
2. 修改完后，启动8401
3. 访问：<http://localhost:8401/testHotKey> 和 <http://localhost:8401/testHotKey?p1=a&p2=b> 都能访问成功
4. 在sentinel控制台中配置热点规则。配置规则如下：



上述配置表示：对testHotKey中的第0个参数设置热点规则，当请求中含有第0个参数达到阈值时，进行熔断。注意：这里第0个参数是指对应controller方法中的参数 `p1`。

5. 访问：<http://localhost:8401/testHotKey?p1=a> (多次访问，保证一秒内最少访问2次)

会显示如下报错信息。即进行熔断了，并使用了 `deal_testHotKey` 方法进行兜底。



6. 对 controller 中的代码进行修改，在注解 `SentinelResource` 中不配置 `blockHandler` 会怎么样？

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

public String deal_testHotKey(String p1, String p2, BlockException ex) {
    return "-----deal_testHotKey~~~~";
}
```

7. 再次访问：<http://localhost:8401/testHotKey?p1=a>（多次访问，保证一秒内最少访问2次）

则会出现以下报错信息。

A screenshot of a browser window. The address bar shows 'localhost:8401/testHotKey?p1=a'. Below the address bar, the main content area displays the text 'Whitelabel Error Page'.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun May 15 16:04:07 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

No message available

java.lang.reflect.UndeclaredThrowableException

所以，在配置`@SentinelResource`注解时，一定要配置兜底方法！

参数例外项

在配置热点规则时，除了可以根据参数索引配置外，还有一个参数例外项的配置。

新增热点规则

资源名	资源名		
限流模式	QPS 模式		
参数索引	请填入传入的热点参数的索引 (从 0 开始)		
单机阈值	0	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="▼"/>		
参数值	例外项参数值	限流阈值	<input type="button" value="+ 添加"/>
参数值	参数类型	限流阈值	操作
关闭高级选项			
		<input type="button" value="新增"/>	<input type="button" value="取消"/>

介绍

在之前的案例中，演示了第一个参数p1，当QPS超过1秒1次点击后马上就被限流。

但是，在某些情况下，我们希望能实现以下要求：

- 在参数p1的值不是特殊值的时候，它的限流阈值QPS=1。
- 在参数p1的值是某个特殊值的时候，它的限流阈值和平时不一样。

比如，当p1的值等于5时，它的阈值QPS=200；当p1的值不等于5时，它的阈值QPS=1.

这就是参数例外项要发挥的作用！

演示配置

1. 编辑之前的热点规则，添加参数例外项：

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	java.lang.String		
参数值	5	限流阈值	200
+添加			
参数值	参数类型	限流阈值	操作
关闭高级选项			
		保存	取消

注意记得点击添加按钮。添加成功后如下：

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	java.lang.String		
参数值	例外项参数值	限流阈值	+ 添加
参数值	参数类型	限流阈值	操作
5	java.lang.String	200	删除

[关闭高级选项](#)

[保存](#) [取消](#)

注意：参数类型只支持7种类型。

参数类型	java.lang.String
参数值	int double java.lang.String long float char byte
参数值	

上述配置表示，当第0个参数的值等于5时，限流阈值为200；当第0个参数的值不等于5时，限流阈值为1。

2. 访问：<http://localhost:8401/testHotKey?p1=1> (多次访问，保证一秒内最少访问2次)

很快就限流了

← → ⌂ ⌂ i localhost:8401/testHotKey?p1=1

-----deal_testHotKey~~~

3. 浏览器多次频繁访问: <http://localhost:8401/testHotKey?p1=5>

不会限流

← → ⌂ ⌂ i localhost:8401/testHotKey?p1=5

-----testHotKey

其他说明

之前我们的限流的方法代码如下:

```
@GetMapping("testHotKey")
@SentinelResource(value = "testHotKey", blockHandler = "deal_testHotKey")
public String testHotKey(@RequestParam(value = "p1", required = false) String p1,
                        @RequestParam(value = "p2", required = false) String p2) {

    return "-----testHotKey";
}

public String deal_testHotKey(String p1, String p2, BlockException ex) {
    return "-----deal_testHotKey~~~";
}
```

如果我们将其修改成如下代码:

```
@GetMapping("testHotKey")
@SentinelResource(value = "testHotKey", blockHandler = "deal_testHotKey")
public String testHotKey(@RequestParam(value = "p1", required = false) String p1,
                        @RequestParam(value = "p2", required = false) String p2) {
    int age = 10 / 0;
    return "-----testHotKey";
}

public String deal_testHotKey(String p1, String p2, BlockException ex) {
    return "-----deal_testHotKey~~~";
}
```

那么请问：我们是否还会使用deal_testHotKey方法进行兜底？

测试：

1. 重启8401
2. 访问: <http://localhost:8401/testHotKey?p1=1>

可以发现直接报错：

```
← → ⌂ ⌄ ⓘ localhost:8401/testHotKey?p1=1

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun May 15 16:38:42 CST 2022
There was an unexpected error (type=Internal Server Error, status=500).
/by zero
java.lang.ArithmetricException: / by zero
    at com.atguigu.springcloud.controller.FlowLimitController.testHotKey(FlowLimitController.java:34)
    at com.atguigu.springcloud.controller.FlowLimitController$$FastClassBySpringCGLIB$$c...
```

其实原因在[官方文档](#)中已经解释了！

④ 方式四：注解方式定义资源

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

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

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

注意 `blockHandler` 函数会在原方法被限流/降级/系统保护的时候调用，而 `fallback` 函数会针对所有类型的异常。请注意 `blockHandler` 和 `fallback` 函数的形式要求，更多指引可以参见[Sentinel 注解支持文档](#)。

主要是 `blockHandler` 函数是在方法被限流/降级/系统保护的时候调用

需要注意 `blockHandler` 和 `fallback` 函数的区别！

`fallback` 函数后面再讲

系统规则

系统规则介绍

系统规则是什么？

官方文档：<https://github.com/alibaba/Sentinel/wiki/%E7%B3%BB%E7%BB%9F%E8%87%AA%E9%80%82%E5%BA%94%E9%99%90%E6%B5%81>

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。

各项配置参数说明

在sentinel控制台中我们可以看到 系统规则 这个列表

The screenshot shows the Sentinel Control Panel version 1.7.0. On the left, there's a sidebar with navigation items: 实时监控, 旗点链路, 流控规则, 降级规则, 热点规则, 系统规则 (which is selected and highlighted with a red border), 授权规则, 集群流控, and 机器列表. The main area displays a service named "cloudalibaba-sentinel-service" with one rule listed. A modal window titled "新增系统保护规则" (Add System Protection Rule) is open. Inside the modal, there are two tabs: "阈值类型" (Threshold Type) and "降级策略" (Degradation Strategy). The "Threshold Type" tab is active, showing five options: LOAD (radio button selected), RT, 线程数 (Threads), 入口 QPS (Entrance QPS), and CPU 使用率 (CPU Usage Rate). Below these options is a threshold input field labeled "[0, ~)的正整数" (Positive integer from [0, ~)). At the bottom of the modal are two buttons: "新增" (Add) and "取消" (Cancel).

根据官方文档可知上述参数的说明：

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。

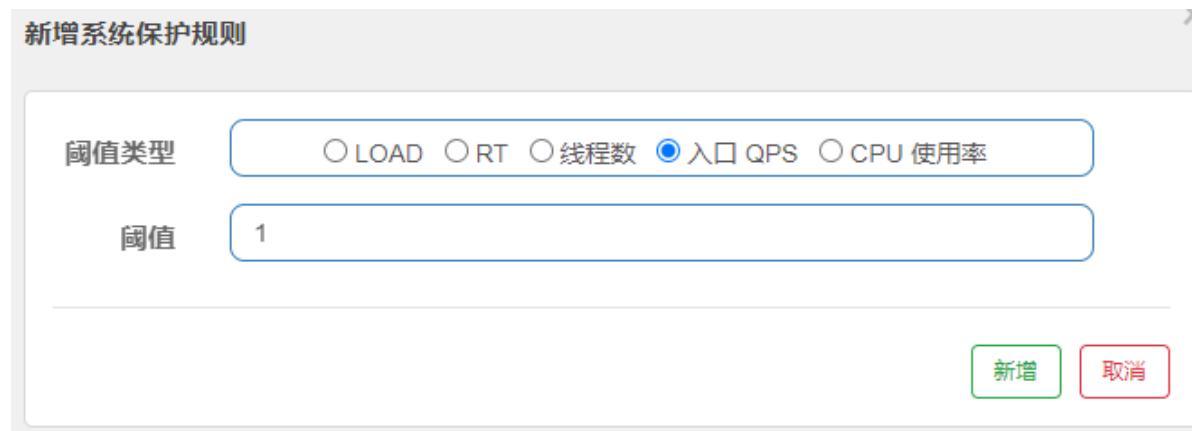
系统规则支持以下的模式：

- **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。

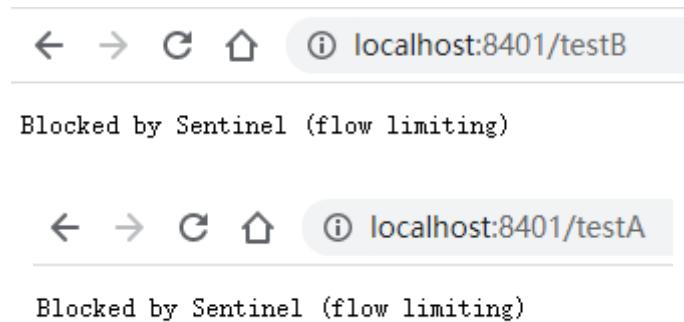
配置全局QPS

下面我们来演示配置入口QPS这个模式。

1. 新增系统规则



2. 分别频繁访问：<http://localhost:8401/testB> 和 <http://localhost:8401/testA>



@SentinelResource

之前在 热点key限流 这一节中，我们使用过 `@SentinelResource`。现在我们来具体讲解 `@SentinelResource`

可以参考 [官方文档](#)

@SentinelResource注解介绍

必看！

根据 [官方文档](#) 来介绍

`@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。

1.8.0 版本开始，`defaultFallback` 支持在类级别进行配置。

注：1.6.0之前的版本 fallback 函数只针对降级异常（`DegradeException`）进行处理，不能针对业务异常进行处理。

特别地，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。若未配置 `blockHandler`、`fallback` 和 `defaultFallback`，则被限流降级时会将 `BlockException` 直接抛出（若方法本身未定义 `throws BlockException` 则会被 JVM 包装一层 `UndeclaredThrowableException`）。

下面我们结合限流演示@SentinelResource的使用

按资源名称限流+后续处理

1. 启动Nacos
2. 启动sentinel

3. 修改8401模块

一、在pom中引入下面这个自定义包

```
1 <!--日常通用jar包配置-->
2 <dependency>
3   <groupId>com.atguigu.springcloud</groupId>
4   <artifactId>cloud-api-commons</artifactId>
5   <version>${project.version}</version>
6 </dependency>
```

二、新创建一个业务类RateLimitController

```
1 @RestController
2 public class RateLimitController {
3     @GetMapping("/byResource")
4     @SentinelResource(value = "byResource", blockHandler = "handleException")
5     public CommonResult byResource() {
6         return new CommonResult(200, "按资源名称限流测试OK", new Payment(2022L,
7             "serial001"));
8     }
9     public CommonResult handleException(BlockException e) {
10        return new CommonResult(444, e.getClass().getCanonicalName() + "\t 服务不
11        可用");
12    }
13 }
```

4. 配置流控规则

这一节我们需要演示的是按资源名称限流

因此，我们下面在sentinel控制台中新增一条流控规则：

The screenshot shows the CloudAlibaba-Sentinel Service control panel. On the left sidebar, under the 'cloudalibaba-sentinel-service' section, the 'Flow Control Rules' option is selected. In the main area, a modal window titled 'Add Flow Control Rule' is open. The 'Resource Name' field contains 'byResource'. The 'Target Source' field is set to 'default'. Under 'Threshold Type', the 'QPS' radio button is selected. The 'Single Machine Threshold' input field is set to '1'. At the bottom right of the modal, there are 'Add' and 'Cancel' buttons, with the 'Add' button highlighted by a red box.

注：上图中的资源名一定是对应 `@SentinelResource` 注解的 value 值。

5. 测试

访问一次：<http://localhost:8401/byResource>

```
{  
  code: 200,  
  message: "按资源名称限流测试OK",  
  data: {  
    id: 2022,  
    serial: "serial001"  
  }  
}
```

频繁访问：<http://localhost:8401/byResource>

```
{  
  code: 444,  
  message: "com.alibaba.csp.sentinel.slots.block.flow.FlowException",  
  data: null  
}
```

服务不可用

6. 思考存在的问题

当8401服务关闭（或宕机）后，可以发现sentinel控制台中的流控规则消失了！这就是目前存在的问题。

The screenshot shows the Sentinel Control Panel. The left sidebar lists applications, with 'cloudalibaba-sentinel-service' selected. Under this application, the 'Flow Control Rules' option is selected. The main area displays a table titled 'Flow Control Rules'. The table has columns for 'Resource Name', 'Source Application', 'Flow Control Mode', 'Threshold Type', 'Threshold Value', 'Threshold Mode', 'Flow Control Effect', and 'Operation'. A red box highlights the entire body of the table, indicating that no rules are listed.

可以发现目前的规则是临时的，并没有做到持久化。

我们后续再讲如何将规则持久化

按照 Url 地址限流 + 后续处理

下面我们演示如何通过访问的 Url 地址来限流

1. 修改业务类

在业务类RateLimitController中添加一个新方法，用于演示

```
1  @GetMapping("/rateLimit/byUrl")
2  @SentinelResource(value = "byUrl")
3  public CommonResult byUrl() {
4      return new CommonResult(200, "按url限流测试OK", new Payment(2022L,
5          "serial002"));
6  }
```

可以看到上述我们没在 `@SentinelResource` 注解中写兜底方法，那么它会用系统默认的。

2. 配置流控规则

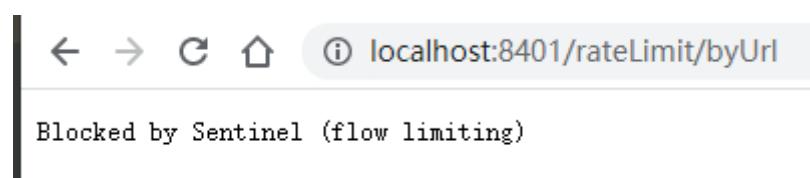
在sentinel控制台中配置如下流控规则：



注意：在上述配置中，资源名中填写的是url，对应的是 `@GetMapping` 中的值！

3. 测试

频繁访问：<http://localhost:8401/rateLimit/byUrl>



上面兜底方案面临的问题

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

其实和刚学Hystrix时说的问题类似，当然后续也有解决方法

客户自定义限流处理逻辑

这里我们统一解决一下上一小节的4个问题。

自定义限流处理类

创建一个 `CustomerBlockHandler` 类用于自定义限流处理逻辑

```
1  public class CustomerBlockHandler {  
2      // 注意：这里一定要用 static 修饰，  
3      // 同时返回值类型和RateLimitController中的customerBlockHandler方法的返回值类型相同  
4      public static CommonResult handlerException(BlockException e) {  
5          return new CommonResult(444, "按客户自定义, global handlerException---1");  
6      }  
7  
8      public static CommonResult handlerException2(BlockException e) {  
9          return new CommonResult(444, "按客户自定义, global handlerException---2");  
10     }  
11 }
```

这里为什么需要使用 `static` 修饰？

参考 [@SentinelResource注解介绍](#) 中的 `blockHandler` / `blockHandlerClass` 属性。

修改RateLimitController业务类

在 `RateLimitController` 类中新增一个方法即可：

```
1  @GetMapping("/rateLimit/customerBlockHandler")  
2  @SentinelResource(value = "customerBlockHandler",  
3                      blockHandlerClass = CustomerBlockHandler.class,  
4                      blockHandler = "handlerException2")  
5  public CommonResult customerBlockHandler() {  
6      return new CommonResult(200, "按客户自定义", new Payment(2022L, "serial003"));  
7  }
```

这里对于 `blockHandlerClass` 和 `blockHandler` 的作用请参考 [@SentinelResource注解介绍](#) 中的 `blockHandler` / `blockHandlerClass` 属性。

Sentinel 控制台配置

在sentinel控制台新增流控规则：



测试

频繁访问: <http://localhost:8401/rateLimit/customerBlockHandler>

可以看到访问的是 `handlerException2` 这个兜底方法

```
← → ⌂ ⌂ localhost:8401/rateLimit/customerBlockHandler
{
  code: 444,
  message: "按客户自定义, global handlerException---2",
  data: null
}
```

如果我们再次修改 `customerBlockHandler` 方法上的 `blockHandler`，让其使用 `handlerException` 方法兜底：

```
@GetMapping("/rateLimit/customerBlockHandler")
@SentinelResource(value = "customerBlockHandler",
    blockHandlerClass = CustomerBlockHandler.class,
    blockHandler = "handlerException")
public CommonResult customerBlockHandler() {
    return new CommonResult(code: 200, message: "按客户自定义", new Payment(id: 2022L, serial: "serial"))
}
```

用热部署重启8401后，再次频繁访问：<http://localhost:8401/rateLimit/customerBlockHandler>

可以看到访问的是 `handlerException` 这个兜底方法

```
← → C ⌂ ⓘ localhost:8401/rateLimit/customerBlockHandler
{
  code: 444,
  message: "按客户自定义, global handlerException---1",
  data: null
}
```

更多说明

Sentinel的API介绍 参考文档: <https://github.com/alibaba/Sentinel/wiki/%E5%A6%82%E4%BD%95%E4%BD%BF%E7%94%A8#%E5%85%B6%E5%AE%83-api>

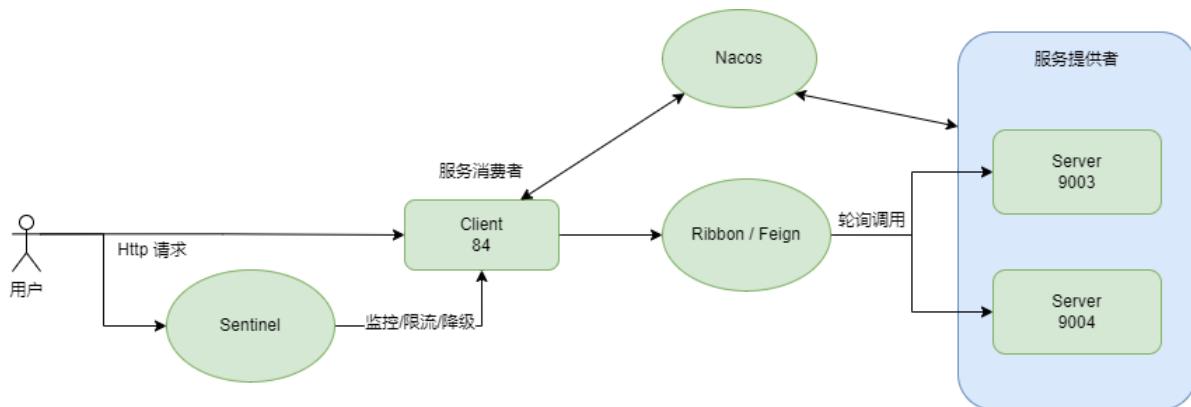
Sentinel主要有三个核心API:

- `Tracer` : 用于记录业务异常
- `SphU` : 定义资源
- `ContextUtil` : 上下文工具类

服务熔断功能

sentinel 整合 ribbon / openFeign

我们下面要搭建的项目整体框架如下:



整合Ribbon系列

先启动nacos和sentinel

再创建服务提供者9003和9004，然后再创建服务消费者84

创建服务提供者

这里以创建9003为例，9004类似步骤。

1. 创建一个子工程 `cloudalibaba-provider-payment9003`
2. 改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>com.atguigu.springcloud</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>
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>
```

3. 写YML

```
1 server:
2     port: 9003
3     spring:
4         application:
5             name: nacos-payment-provider
6         cloud:
7             nacos:
8                 discovery:
9                     server-addr: localhost:8848 # 配置 Nacos 地址
10        management:
```

```
11     endpoints:  
12         web:  
13             exposure:  
14                 include: '*'
```

4. 主启动

```
1  @SpringBootApplication  
2  @EnableDiscoveryClient  
3  public class PaymentMain9003 {  
4      public static void main(String[] args) {  
5          SpringApplication.run(PaymentMain9003.class, args);  
6      }  
7  }
```

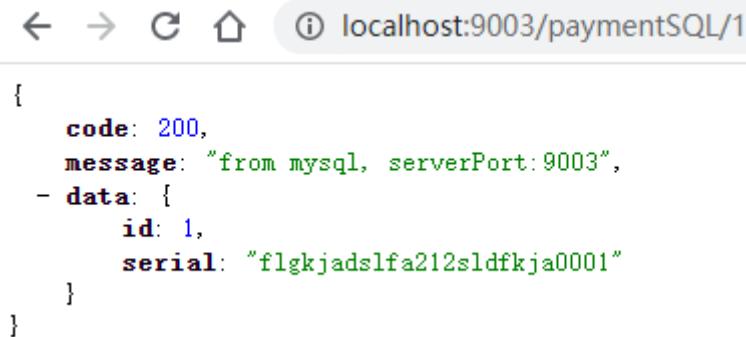
5. 业务类

```
1  @RestController  
2  public class PaymentController {  
3      @Value("${server.port}")  
4      private String serverPort;  
5  
6      public static HashMap<Long, Payment> hashMap = new HashMap<>();  
7  
8      static {  
9          hashMap.put(1L, new Payment(1L, "flgkjadsdfa212sldfkja0001"));  
10         hashMap.put(2L, new Payment(2L, "flgkjadsdfa212sldfkja0002"));  
11         hashMap.put(3L, new Payment(3L, "flgkjadsdfa212sldfkja0003"));  
12     }  
13  
14     @GetMapping("/paymentSQL/{id}")  
15     public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id) {  
16         Payment payment = hashMap.get(id);  
17         CommonResult<Payment> result = new CommonResult<>(200,  
18                         "from mysql, serverPort:" + serverPort, payment);  
19         return result;  
20     }  
21 }
```

6. 测试

访问: <http://localhost:9003/paymentSQL/1>

会出现



创建服务消费者

1. 新建子工程 `cloudalibaba-consumer-nacos-order84`
2. 改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     <!--SpringCloud alibaba sentinel -->
8     <dependency>
9         <groupId>com.alibaba.cloud</groupId>
10        <artifactId>spring-cloud-starter-alibaba-sentinel</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>com.atguigu.springcloud</groupId>
15        <artifactId>cloud-api-commons</artifactId>
16        <version>${project.version}</version>
17    </dependency>
18    <!-- SpringBoot整合Web组件 -->
19    <dependency>
20        <groupId>org.springframework.boot</groupId>
21        <artifactId>spring-boot-starter-web</artifactId>
22    </dependency>
23    <dependency>
24        <groupId>org.springframework.boot</groupId>
25        <artifactId>spring-boot-starter-actuator</artifactId>
26    </dependency>
27    <!--日常通用jar包配置-->
28    <dependency>
29        <groupId>org.springframework.boot</groupId>
30        <artifactId>spring-boot-devtools</artifactId>
31        <scope>runtime</scope>
32        <optional>true</optional>
33    </dependency>
34    <dependency>
35        <groupId>org.projectlombok</groupId>
36        <artifactId>lombok</artifactId>
37        <optional>true</optional>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework.boot</groupId>
41        <artifactId>spring-boot-starter-test</artifactId>
42        <scope>test</scope>
43    </dependency>
44 </dependencies>
```

3. 写YML

```
1 server:
2   port: 84
3 spring:
4   application:
5     name: nacos-order-consumer
6   cloud:
```

```

7     nacos:
8         discovery:
9             server-addr: localhost:8848
10        sentinel:
11            transport:
12                dashboard: localhost:8080
13                port: 8719
14
15        service-url:
16            nacos-user-service: http://nacos-payment-provider

```

4. 主启动

```

1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class OrderNacosMain84 {
4     public static void main(String[] args) {
5         SpringApplication.run(OrderNacosMain84.class, args);
6     }
7 }

```

5. 业务类

config配置

```

1 @Configuration
2 public class ApplicationContextConfig {
3
4     @Bean
5     @LoadBalanced
6     public RestTemplate getRestTemplate() {
7         return new RestTemplate();
8     }
9 }

```

controller类

```

1 @RestController
2 @Slf4j
3 public class CircleBreakerController {
4
5     @Value("${service-url.nacos-user-service}")
6     private String SERVICE_URL;
7
8     @Resource
9     private RestTemplate restTemplate;
10
11    @RequestMapping("/consumer/fallback/{id}")
12    @SentinelResource(value = "fallback") // 没有配置
13    public CommonResult<Payment> fallback(@PathVariable("id") Long id) {
14        CommonResult<Payment> result =
15            restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id,
16            CommonResult.class, id);
16        if (id == 4) {
17            throw new IllegalArgumentException("IllegalArgumentException, 非法参数
18            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     return result;
22 }
23 }
```

从上述controller类可以看到，当前的 `@SentinelResource` 注解没有配置 `fallback` 或 `blockHandler` 属性，当限流时只会出现默认报错页面。

没有配置blockHandler和fallback

下面我们演示没有配置blockHandler和fallback时的情况。

1. 首先启动nacos和sentinel
2. 启动9003和9004服务提供者
3. 启动84服务消费者
4. 多次访问地址: <http://localhost:84/consumer/fallback/1>

多次访问时发现是轮询访问两个服务提供者



5. 访问地址: <http://localhost:84/consumer/fallback/4>

```
← → C ⌂ ⓘ localhost:84/consumer/fallback/4
```

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue May 17 10:22:26 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

IllegalArgumentException, 非法参数异常

6. 访问地址: <http://localhost:84/consumer/fallback/5>



This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue May 17 10:22:57 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

NullPointerException, 该ID没有对应记录, 空指针异常

从上述测试来看, 如果我们没有配置blockHandler和fallback, 可以看到, 报错信息会直接显示到页面中, 对用户不友好。

只配置fallback

- 在服务消费者84的 `CircleBreakerController` 类中修改 `fallback` 方法, 并新增一个 `handlerFallback` 方法

```
1  @RequestMapping("/consumer/fallback/{id}")
2  // @SentinelResource(value = "fallback")    // 没有配置
3  @SentinelResource(value = "fallback", fallback = "handlerFallback")    // 只配置
   fallback
4  public CommonResult<Payment> fallback(@PathVariable("id") Long id) {
5      CommonResult<Payment> result =
6          restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id,
7          CommonResult.class, id);
8      if (id == 4) {
9          throw new IllegalArgumentException("IllegalArgumentException, 非法参数异
常");
10     } else if (result.getData() == null) {
11         throw new NullPointerException("NullPointerException, 该ID没有对应记录, 空指
针异常");
12     }
13     return result;
14 }
15 public CommonResult handlerFallback(@PathVariable("id") Long id, Throwable e) {
16     Payment payment = new Payment(id, "null");
17     return new CommonResult(444, "兜底异常handlerFallback, exception内容 " +
18     e.getMessage(), payment);
19 }
```

测试:

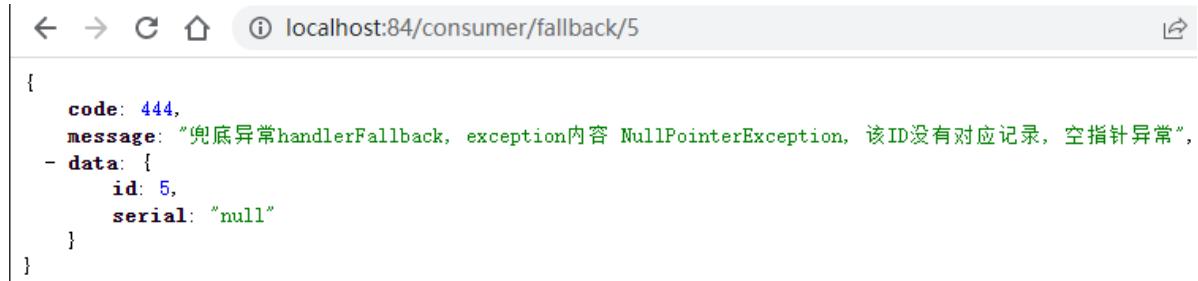
- 重启84服务消费者
- 访问: <http://localhost:84/consumer/fallback/1>. 依旧正常访问, 轮询访问两个服务提供者
- 访问: <http://localhost:84/consumer/fallback/4>

可以看到, 现在显示的报错信息就是我们自定义的。

```
{  
    code: 444,  
    message: "兜底异常handlerFallback, exception内容 InvalidArgumentException, 非法参数异常",  
    - data: {  
        id: 4,  
        serial: "null"  
    }  
}
```

4. 访问: <http://localhost:84/consumer/fallback/5>

可以看到, 现在显示的报错信息也是我们自定义的。



只配置blockHandler

这里我们演示只配置blockHandler的效果

1. 在服务消费者84的 `CircleBreakerController` 类中修改 `fallback` 方法, 并新增一个 `blockHandler` 方法

```
1  @RequestMapping("/consumer/fallback/{id}")  
2  // @SentinelResource(value = "fallback") // 没有配置  
3  // @SentinelResource(value = "fallback", fallback = "handlerFallback") // 只配置  
4  fallback  
5  @SentinelResource(value = "fallback", blockHandler = "blockHandler") // 只配置  
6  blockHandler. 注意: blockHandler只负责sentinel控制台配置违规  
7  public CommonResult<Payment> fallback(@PathVariable("id") Long id) {  
8      CommonResult<Payment> result =  
9          restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id,  
10         CommonResult.class, id);  
11         if (id == 4) {  
12             throw new IllegalArgumentException("IllegalArgumentException, 非法参数异  
13             常");  
14         } else if (result.getData() == null) {  
15             throw new NullPointerException("NullPointerException, 该ID没有对应记录, 空指  
16             针异常");  
17         }  
18         return result;  
19     }  
20     public CommonResult blockHandler(@PathVariable("id") Long id, BlockException  
21     blockException) {  
22         Payment payment = new Payment(id, "null");  
23         return new CommonResult(445, "blockHandler-sentinel限流, 无此流水:  
24         blockException " + blockException.getMessage(), payment);  
25     }
```

测试:

1. 重启服务消费者84

2. 由于我们配置了blockHandler，所以我们在配置一下sentinel
在sentinel控制台中新增降级规则

The screenshot shows the 'Add Degradation Rule' dialog box. It has several input fields and buttons:

- 资源名:** fallback (highlighted with a red box)
- 降级策略:** RT 异常比例 异常数 (highlighted with a red box)
- 异常数:** 2 (highlighted with a red box)
- 时间窗口:** 2 (highlighted with a red box)
- Buttons:** 新增并继续添加 (gray), 新增 (green), 取消 (red)

3. 访问: <http://localhost:84/consumer/fallback/1> 能正常访问
4. 访问: <http://localhost:84/consumer/fallback/4>

1秒内只访问一次，则报错页面是系统默认的：（这是因为未达到降级规则，所以抛出的是非法参数异常）



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue May 17 10:54:27 CST 2022
There was an unexpected error (type=Internal Server Error, status=500).
IllegalArgumentException, 非法参数异常

1秒内访问多次，则报错页面是自定义的：（这是因为达到了降级规则，所以抛出的是BlockException）



5. 访问: <http://localhost:84/consumer/fallback/5>

1秒内只访问一次，则报错页面是系统默认的：（这是因为未达到降级规则，所以抛出的是空指针异常）

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue May 17 10:58:02 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

NullPointerException, 该ID没有对应记录, 空指针异常

1秒内访问多次，则报错页面是自定义的：（这是因为达到了降级规则，所以抛出的是BlockException）

```
{  
    code: 445,  
    message: "blockHandler-sentinel限流, 无此流水: blockException null",  
    - data: {  
        id: 5,  
        serial: "null"  
    }  
}
```

fallback和blockHandler都配置

- 在服务消费者84的 `CircleBreakerController` 类中修改 `fallback` 方法，并新增一个 `blockHandler` 方法和 `handlerFallback` 方法

```
1  @RequestMapping("/consumer/fallback/{id}")  
2  // @SentinelResource(value = "fallback") // 没有配置  
3  // @SentinelResource(value = "fallback", fallback = "handlerFallback") // 只配置  
4  // @SentinelResource(value = "fallback", blockHandler = "blockHandler") // 只配  
5  // @SentinelResource(value = "fallback", blockHandler = "blockHandler", fallback =  
6  // "handlerFallback") // blockHandler和fallback都配置  
7  public CommonResult<Payment> fallback(@PathVariable("id") Long id) {  
8      CommonResult<Payment> result =  
9          restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id,  
CommonResult.class, id);  
10     if (id == 4) {  
11         throw new IllegalArgumentException("IllegalArgumentException, 非法参数异  
常");  
12     } else if (result.getData() == null) {  
13         throw new NullPointerException("NullPointerException, 该ID没有对应记录, 空指  
针异常");  
14     }  
15     return result;  
16 }  
17 public CommonResult handlerFallback(@PathVariable("id") Long id, Throwable e) {  
18     Payment payment = new Payment(id, "null");  
19     return new CommonResult(444, "兜底异常handlerFallback, exception内容 " +  
e.getMessage(), payment);  
20 }
```

```
21
22     public CommonResult blockHandler(@PathVariable("id") Long id, BlockException
23             blockException) {
24         Payment payment = new Payment(id, "null");
25         return new CommonResult(445, "blockHandler-sentinel限流, 无此流水:
blockException " + blockException.getMessage(), payment);
26     }
```

测试：

1. 重启服务消费者84
2. 在sentinel控制台中删除之前的降级/流控规则，换成下面的流控规则：



3. 访问：<http://localhost:84/consumer/fallback/1>

1秒内最多访问1次，则访问正常：

```
{  
    code: 200,  
    message: "from mysql, serverPort:9004",  
    data: {  
        id: 1,  
        serial: "flgkjadsdfa212sldfkja0001"  
    }  
}
```

1秒内访问多次，则报错：（可以看到这里满足流控规则，所以抛出的异常是BlockException，由blockHandler指定的方法处理）

```
← → C ⌂ ⓘ localhost:84/consumer/fallback/1

{
  code: 445,
  message: "blockHandler-sentinel限流，无此流水: blockException null",
  - data: {
    id: 1,
    serial: "null"
  }
}
```

4. 访问: <http://localhost:84/consumer/fallback/4>

1秒内最多访问1次，则出现以下报错信息（因为这里不满足流控规则，但是由于会抛出异常，所以会走 `fallback` 属性指定的方法）

```
← → C ⌂ ⓘ localhost:84/consumer/fallback/4

{
  code: 444,
  message: "兜底异常handlerFallback, exception内容 IllegalArgumentException, 非法参数异常",
  - data: {
    id: 4,
    serial: "null"
  }
}
```

1秒内访问多次，则出现以下报错信息（因为这里满足流控规则，所以抛出 `BlockException` 异常，因此会走 `blockHandler` 属性指定的方法）

```
← → C ⌂ ⓘ localhost:84/consumer/fallback/4

{
  code: 445,
  message: "blockHandler-sentinel限流，无此流水: blockException null",
  - data: {
    id: 4,
    serial: "null"
  }
}
```

小结

- 没有配置 `blockHandler` 和 `fallback`
- 只配置 `fallback`
- 只配置 `blockHandler`
- `blockHandler` 和 `fallback` 都配置

从上述演示这四个的效果来看：

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。若未配置 `blockHandler`、`fallback` 和 `defaultFallback`，则被限流降级时会将 `BlockException` 直接抛出（若方法本身未定义 `throws BlockException` 则会被 JVM 包装一层 `UndeclaredThrowableException`）。

异常忽略属性

在 `@SentinelResource` 注解中还有一个属性：`exceptionsToIgnore`

`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。

下面我们演示以下异常忽略属性的使用。

1. 修改服务消费者84的 `CircleBreakerController` 类中的 `fallback` 方法。在 `@SentinelResource` 注解中添加 `exceptionsToIgnore` 属性

```
1  @RequestMapping("/consumer/fallback/{id}")
2  @SentinelResource(value = "fallback", blockHandler = "blockHandler", fallback =
3      "handlerFallback",
4          exceptionsToIgnore = {IllegalArgumentException.class})
5  public CommonResult<Payment> fallback(@PathVariable("id") Long id) {
6      CommonResult<Payment> result =
7          restTemplate.getForObject(SERVICE_URL + "/paymentSQL/" + id,
8              CommonResult.class, id);
9      if (id == 4) {
10          throw new IllegalArgumentException("IllegalArgumentException, 非法参数异常");
11      } else if (result.getData() == null) {
12          throw new NullPointerException("NullPointerException, 该ID没有对应记录, 空指针异常");
13      }
14      return result;
15 }
```

测试：

1. 重启84服务
2. 访问：<http://localhost:84/consumer/fallback/4>

1秒最多访问一次，则页面显示默认错误。（因为忽略了非法参数异常，所以fallback不起作用）



← → ⌂ ⌂ ⓘ localhost:84/consumer/fallback/4

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue May 17 12:55:37 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

IllegalArgumentException, 非法参数异常

整合Feign系列

在整合Ribbon系列工程代码的基础上修改一下代码，让其变成整合Feign系列的代码。

修改消费者84的代码

由于Feign组件一般用于消费侧，所以这里修改消费者84的代码。

1. 在pom中引入新包

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

2. 在yml中激活sentinel对feign的支持

```
1 # 激活Sentinel对Feign的支持
2 feign:
3   sentinel:
4     enabled: true
```

3. 新增一个业务接口

```
1 @FeignClient(value = "nacos-payment-provider", fallback =
2 PaymentFallbackService.class)
3 public interface PaymentService {
4   @GetMapping("/paymentSQL/{id}")
5   CommonResult<Payment> paymentSQL(@PathVariable("id") Long id);
6 }
```

注：

- `@FeignClient` 注解中的 `fallback` 属性：定义容错的处理类，也就是回退逻辑。`fallback`类必须实现 `FeignClient` 标注的接口。

4. 实现指定fallback类

```
1 @Component
2 public class PaymentFallbackService implements PaymentService {
3   @Override
4   public CommonResult<Payment> paymentSQL(Long id) {
5     return new CommonResult<>(444, "服务降级返回, -----PaymentFallbackService",
6     new Payment(id, "errorSerial....."));
7 }
```

5. 修改 `CircleBreakerController` 类

```
1 @RestController
```

```

2   @Slf4j
3   public class CircleBreakerController {
4
5       @Resource
6       private PaymentService paymentService;
7
8       @GetMapping("/consumer/openfeign/{id}")
9       public CommonResult<Payment> paymentSQL(@PathVariable("id") Long id) {
10          if (id == 4) {
11              throw new RuntimeException("没有该id");
12          }
13          return paymentService.paymentSQL(id);
14      }
15  }

```

6. 修改主启动类。在类上添加 `@EnableFeignClients` 注解

```

1   @SpringBootApplication
2   @EnableDiscoveryClient
3   @EnableFeignClients
4   public class OrderNacosMain84 {
5       public static void main(String[] args) {
6           SpringApplication.run(OrderNacosMain84.class, args);
7       }
8   }

```

测试

代码修改完成。

下面测试一下是否能正常启动。

访问地址: <http://localhost:84/consumer/openfeign/1>

可以正常显示。



测试84调用9003，此时故意关闭9003微服务提供者，看84消费侧自动降级，不会被耗死。

操作如下：

1. 关闭微服务提供者9003和9004
2. 访问地址: <http://localhost:84/consumer/openfeign/1>

可以看到会调用指定的fallback方法

```
{  
    code: 444,  
    message: "服务降级返回， -----PaymentFallbackService",  
    - data: {  
        id: 1,  
        serial: "errorSerial....."  
    }  
}
```

小结

在上述代码中，我们主要使用到 `@FeignClient` 注解

参考 [官方文档](#) 可知：

#6.1.5. Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

```
@FeignClient(name = "hello", fallback = HystrixClientFallback.class)  
protected interface HystrixClient {  
    @RequestMapping(method = RequestMethod.GET, value = "/hello")  
    Hello iFailSometimes();  
  
    static class HystrixClientFallback implements HystrixClient {  
        @Override  
        public Hello iFailSometimes() {  
            return new Hello("fallback");  
        }  
    }  
}
```

JAVA

Hystrix 支持回退 (fallback) 的概念：当电路开路或出现错误时执行的默认代码路径。要为给定 `@FeignClient` 启用回退，请将 `fallback` 属性设置为实现回退的类名。您还需要将实现声明为 Springbean。

注：OpenFeign 中包含了 Hystrix

熔断框架比较

参考：<https://github.com/alibaba/Sentinel/wiki/%E5%9C%A8%E7%94%9F%E4%BA%A7%E7%8E%AF%E5%A2%83%E4%B8%AD%E4%BD%BF%E7%94%A8-Sentinel#%E5%90%8C%E7%B1%BB%E7%BB%84%E4%BB%B6%E5%8A%9F%E8%83%BD%E5%AF%B9%E6%AF%94>

	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离（并发控制）	线程池隔离/信号量隔离	信号量隔离
熔断降级策略	基于慢调用比例、异常比例、异常数	基于异常比例	基于异常比例、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	支持近十种动态数据源	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
单机限流	基于 QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
集群流控	支持	不支持	不支持
流量整形	支持预热模式与匀速排队控制效果	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
热点识别/防护	支持	不支持	不支持
多语言支持	Java/Go/C++	Java	Java
Service Mesh 支持	支持 Envoy/Istio	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、实时监控、机器发现等	简单的监控查看	不提供控制台，可对接其它监控系统

规则持久化

在 `@SentinelResource` 的一个小节中，我们可以看到目前我们学习的Sentinel还是存在问题的。

规则持久化是什么

一旦我们重启应用，sentinel规则就会消失。因此，在生产环境中需要将配置规则进行持久化。

参考：<https://github.com/alibaba/Sentinel/wiki/%E5%9C%A8%E7%94%9F%E4%BA%A7%E7%8E%AF%E5%A2%83%E4%B8%AD%E4%BD%BF%E7%94%A8-Sentinel>

从官方文档中我们可以知道，规则的推送由下面三种模式：

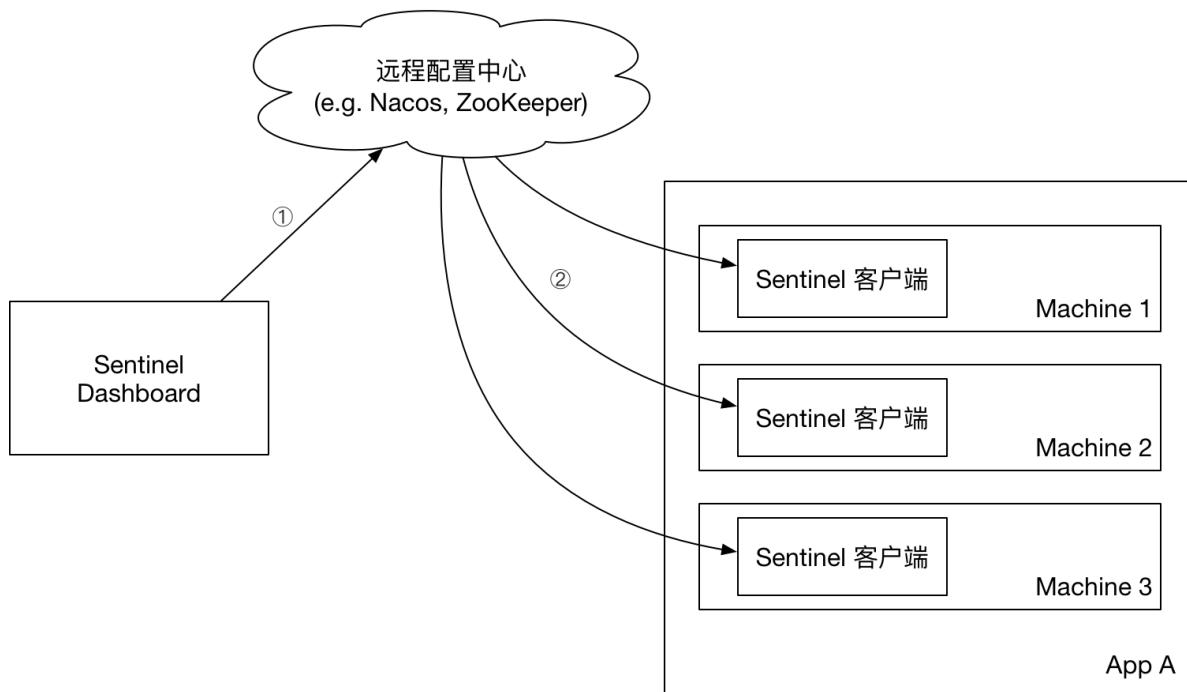
推送模式	说明	优点	缺点
原始模式	API 将规则推送至客户端并直接更新到内存中，扩展写数据源（ <code>WritableDataSource</code> ）	简单，无任何依赖	不保证一致性；规则保存在内存中，重启即消失。严重不建议用于生产环境
Pull模式	扩展写数据源（ <code>WritableDataSource</code> ），客户端主动向某个规则管理中心定期轮询拉取规则，这个规则中心可以是 RDBMS、文件等	简单，无任何依赖；规则持久化	不保证一致性；实时性不保证，拉取过于频繁也可能会有性能问题。
Push模式	扩展读数据源（ <code>ReadableDataSource</code> ），规则中心统一推送，客户端通过注册监听器的方式时刻监听变化，比如使用 Nacos、Zookeeper 等配置中心。这种方式有更好的实时性和一致性保证。 生产环境下一般采用 push 模式的数据源。	规则持久化；一致性；快速	引入第三方依赖

官方推荐 push 模式！

如何使用

因为官方推荐push模式，所以我们这里只讲如何配置push模式。

可以参考官方文档：<https://github.com/alibaba/Sentinel/wiki/%E5%9C%A8%E7%94%9F%E4%BA%A7%E7%8E%AF%E5%A2%83%E4%B8%AD%E4%BD%BF%E7%94%A8-Sentinel#Push%E6%A8%A1%E5%BC%8F>



下面我们演示将限流配置规则持久化进 nacos 保存，只要刷新sentinel监控的服务的某个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，针对该服务上sentinel上的流控规则持续有效。

演示步骤

下面我们以 cloudalibaba-sentinel-service8401 来演示

改POM

修改 cloudalibaba-sentinel-service8401 的pom.xml文件

```
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>
```

改YML

修改 cloudalibaba-sentinel-service8401 的application.yml文件

```
1 server:
2   port: 8401
3   spring:
4     application:
5       name: cloudalibaba-sentinel-service
6     cloud:
7       nacos:
8         discovery:
9           server-addr: localhost:8848
10    sentinel:
11      transport:
12        dashboard: localhost:8080
13        # 服务和sentinel发生交互的port，默认是8719。如果被其他应用占用，则会自动从8719开始
14        # 依次+1扫描，直到找到未被占用的端口
15        port: 8719
16      datasource:
17        dsl:
18          nacos:
19            server-addr: localhost:8848
20            dataId: cloudalibaba-sentinel-service
21            groupId: DEFAULT_GROUP
22            data-type: json
23            rule-type: flow
24
25      management:
26        endpoints:
27          web:
28            exposure:
29              include: '*'
```

注：主要添加的是Nacos数据源配置：

```

1  spring:
2    application:
3      name: cloudalibaba-sentinel-service
4    cloud:
5      sentinel:
6        datasource:
7          dsl:
8            nacos:
9              server-addr: localhost:8848
10             dataId: ${spring.application.name}
11             groupId: DEFAULT_GROUP
12             data-type: json
13             rule-type: flow

```

测试是否能正常启动

1. 启动8401
2. 访问：<http://localhost:8401/rateLimit/byUrl>



添加Nacos业务规则配置

登录nacos：<http://localhost:8848/nacos>

在nacos中新建配置：

新建配置

* Data ID: cloudalibaba-sentinel-service

* Group: DEFAULT_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 {  
3     "resource": "/rateLimit/byUrl",  
4     "limitApp": "default",  
5     "grade": 1,  
6     "count": 1,  
7     "strategy": 0,  
8     "controlBehavior": 0,  
9     "clusterMode": false  
10 }  
11 ]
```

上述配置是什么？

```
1 [  
2     {  
3         "resource": "/rateLimit/byUrl",  
4         "limitApp": "default",  
5         "grade": 1,  
6         "count": 1,  
7         "strategy": 0,  
8         "controlBehavior": 0,  
9         "clusterMode": false  
10     }  
11 ]
```

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。

测试

- 启动8401, 访问一次: <http://localhost:8401/rateLimit/byUrl>
- 检查sentinel控制台中是否出现在nacos中配置的业务规则

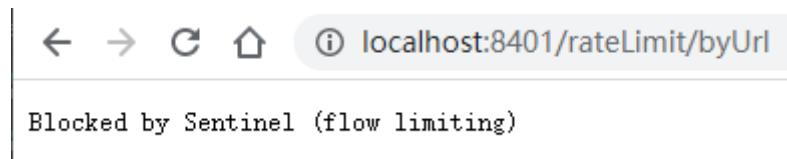
可以看到, nacos中配置的规则应用到sentinel中了

Sentinel 控制台 1.7.0

The screenshot shows the Sentinel Control Panel interface. On the left, there's a sidebar with navigation links: 首页, 实时监控, 簇点链路, 流控规则 (highlighted with a red box), 降级规则, and 热点规则. The main content area is titled 'cloudalibaba-sentinel-service'. It has a search bar and a button '+ 新增流控规则'. Below that is a table with columns: 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, and 操作. A single row is shown: '/rateLimit/byUrl', 'default', '直接', 'QPS', '1', '单机', '快速失败', with '编辑' and '删除' buttons. At the bottom, it says '共 1 条记录, 每页 10 条记录'.

- 快速访问多次: <http://localhost:8401/rateLimit/byUrl>

可以看到, 满足流控规则, 所以报错。



- 停止8401服务后, 再查看sentinel

发现sentinel中的流控规则也消失了

Sentinel 控制台 1.7.0

The screenshot shows the Sentinel Control Panel interface, similar to the previous one but with a different result. The main content area is titled 'cloudalibaba-sentinel-service'. The table below shows zero results: '共 0 条记录, 每页 10 条记录'.

- 再重启8401, 访问: <http://localhost:8401/rateLimit/byUrl> 后, 再查看sentinel

可以看到流控规则又出现了。

The screenshot shows the Sentinel Control Console interface. On the left, there's a sidebar with navigation links: 首页 (Home), 实时监控 (Real-time Monitoring), 旗点链路 (Flag Point Chain), 流控规则 (Flow Control Rules) (which is highlighted with a red box), and 降级规则 (Degradation Rules). The main area displays the configuration for the 'cloudalibaba-sentinel-service' application. At the top right, there's a button for '新增流控规则' (Add Flow Control Rule). Below it, a search bar has '10.1.125.73:8720' selected. A table lists the flow control rules:

资源名	来源应用	流控模式	阈值类型	阈值	流控模式	操作
/rateLimit/byUrl	default	直接	QPS	1	单机	快速失败

On the far right of the table, there are '编辑' (Edit) and '删除' (Delete) buttons.

十八、SpringCloud Alibaba Seata处理分布式事务

分布式事务问题

在引入分布式之前，我们用的是单机单库，不会出现分布式事务问题。

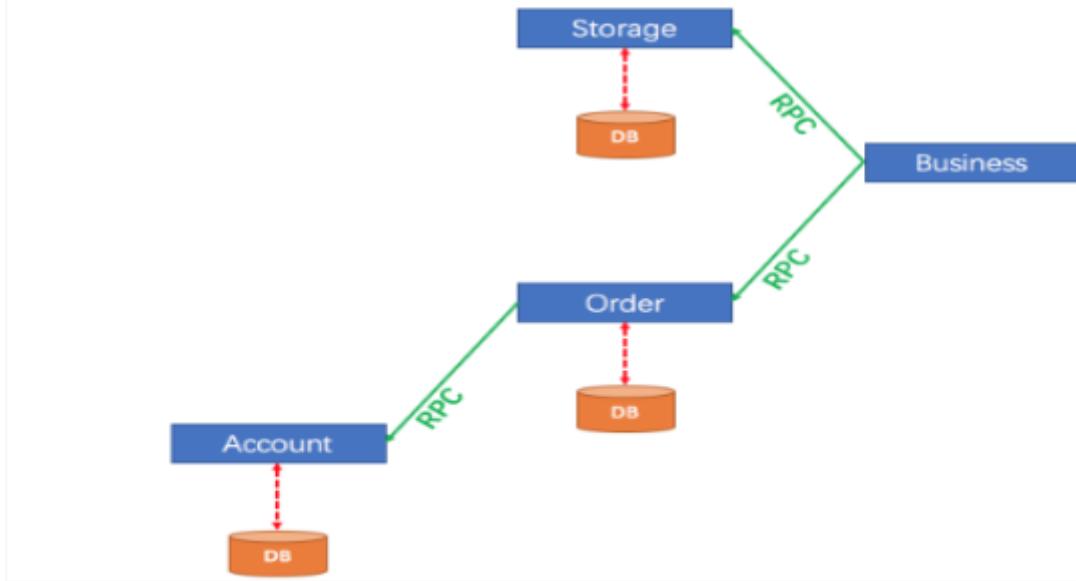
在引入分布式之后，

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



总结：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。

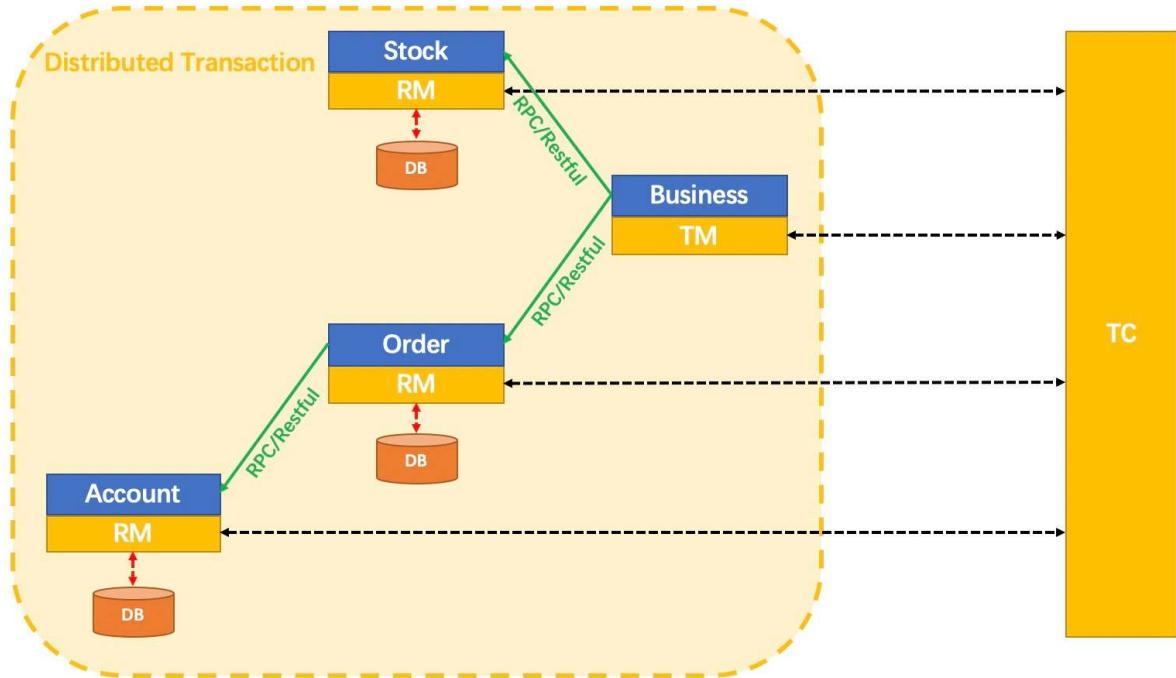
Seata简介

Seata是什么

官网：<http://seata.io/zh-cn/>

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。

Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。



什么是 AT、TCC、SAGA 和 XA 事务模式？

参考官网：<https://seata.io/zh-cn/docs/overview/what-is-seata.html>

Seata默认用的是 AT 模式

Seata术语

一个典型的分布式事务过程：

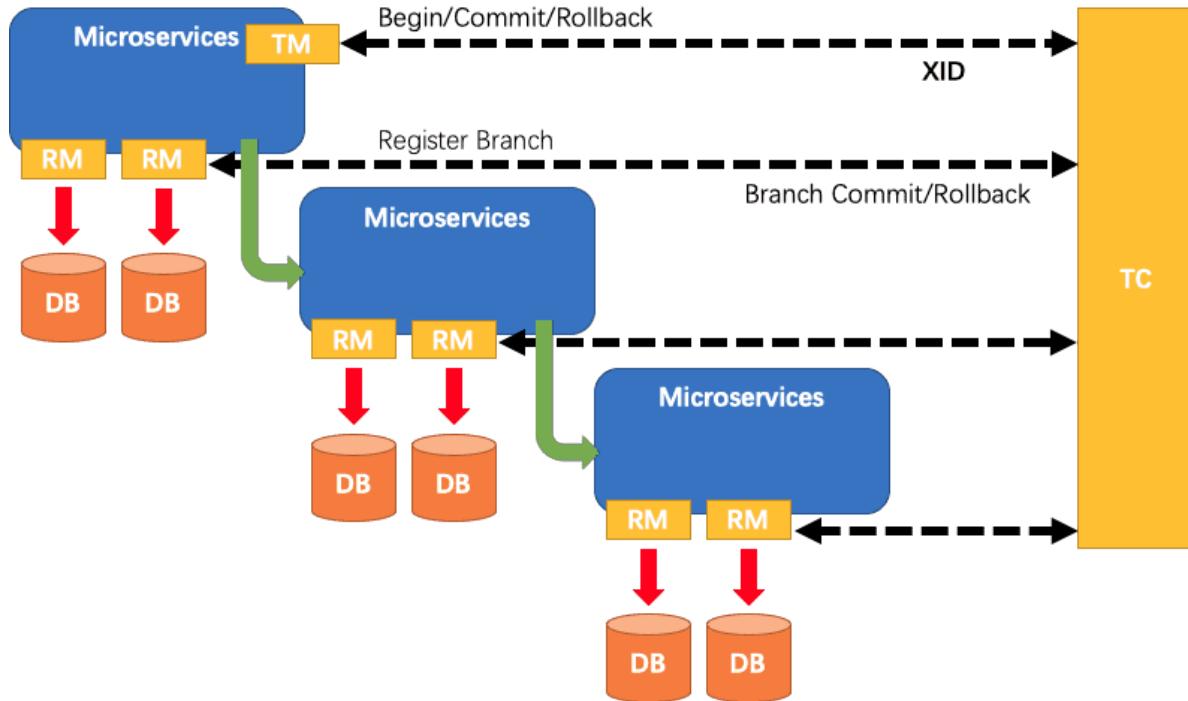
分布式事务的一些概念：

分布式事务处理过程 (一个ID+三组件模型)

- Transaction ID (XID)：全局唯一的事务ID
- 三组件概念：
 - Transaction Coordinator (TC) : 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
 - Transaction Manager (TM) : 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。
 - Resource Manager (RM) : 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。

分布式事务处理过程：

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 下管辖的全局分支事务完成提交或回滚请求



Seata 的下载地址

本次下载安装Seata0.9.0

下载地址: <https://github.com/seata/seata/releases/tag/v0.9.0>

▼ Assets 4

seata-server-0.9.0.tar.gz	linux用这个
seata-server-0.9.0.zip	windows用这个
Source code (zip)	
Source code (tar.gz)	

Seata的简单使用

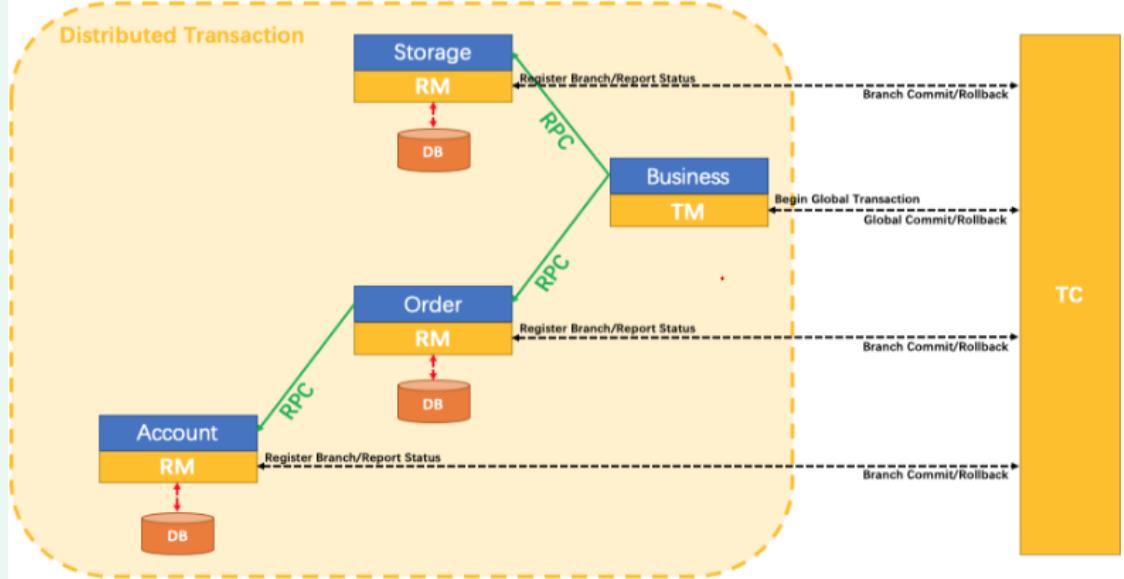
注意: Seata的搭建和编码要学懂比较难, 但是学会Seata的使用比较简单。

我们之前学习Spring框架时, 使用本地事务所需要的注解是 `@Transactional`

现在要想使用Seata就需要记住全局事务的注解 `@GlobalTransactional`

举例:

一个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. 下载安装Seata

1. 下载Seata0.9.0版本
2. 解压到指定目录

2. 修改seata的file.conf配置文件

修改conf目录下的file.conf配置文件（修改前先备份一下）

主要修改：自定义事务组名称+事务日志存储模型为db+数据库连接信息

在打开file.conf后，主要修改其中的 **service模块** 和 **store模块**

service模块：

```

} service {
    #vgroup->rgroup
    vgroup_mapping.my_test_tx_group = "default"
    #only support single node
    default.grouplist = "127.0.0.1:8091"
    #degrade current not support
    enableDegrade = false
    #disable
    disable = false
    #unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days, default permanent
    max.commit.retry.timeout = "-1"
    max.rollback.retry.timeout = "-1"
}

```

store模块：

```

## transaction log store
store {
    ## store mode: file, db
    mode = "file"

    ## file store
    file {
        dir = "sessionStore"

        # branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions
        max-branch-session-size = 16384
        # globe session size , if exceeded throws exceptions
        max-global-session-size = 512
        # file buffer size , if exceeded allocate new buffer
        file-write-buffer-cache-size = 16384
        # when recover batch read size
        session.reload.read_size = 100
        # async, sync
        flush-disk-mode = async
    }

    ## database store
    db {
        ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.
        datasource = "dbcp"
        ## mysql/oracle/h2/oceanbase etc.
        db-type = "mysql"
        driver-class-name = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://127.0.0.1:3306/seata"
        user = "mysql"
        password = "mysql"
        min-conn = 1
        max-conn = 3
        global.table = "global_table"
        branch.table = "branch_table"
        lock-table = "lock_table"
        query-limit = 100
    }
}

```

修改后：

service模块：自定义事务组名称

```

service {
    #vgroup->rgroup
    vgroup_mapping.my_test_tx_group = "fsp_tx_group"      指定一个自定义分组
    #only support single node
    default.grouplist = "127.0.0.1:8091"
    #degrade current not support
    enableDegrade = false
    #disable
    disable = false
    #unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days, default permanent
    max.commit.retry.timeout = "-1"
    max.rollback.retry.timeout = "-1"
}

```

store模块：事务日志存储模型为db+数据库连接信息

```
## transaction log store
store {
    ## store mode: file、db
    mode = "db" 事务日志存储模型为db

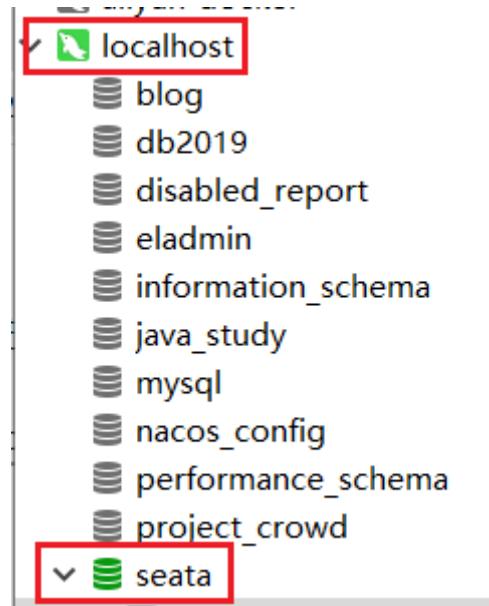
    ## file store
    file {
        dir = "sessionStore"

        # branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions
        max-branch-session-size = 16384
        # globe session size , if exceeded throws exceptions
        max-global-session-size = 512
        # file buffer size , if exceeded allocate new buffer
        file-write-buffer-cache-size = 16384
        # when recover batch read size
        session.reload.read_size = 100
        # async, sync
        flush-disk-mode = async
    }

    ## database store
}
db {
    ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.
    datasource = "dbcp"
    ## mysql/oracle/h2/oceanbase etc.
    db-type = "mysql"
    driver-class-name = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://127.0.0.1:3306/seata"          数据库连接信息
    user = "root"
    password = "123456"
    min-conn = 1
    max-conn = 3
    global.table = "global_table"
    branch.table = "branch_table"
    lock-table = "lock_table"
    query-limit = 100
}
}
```

3. 在mysql中新建seata库

从上述配置文件中可以了解到，我们还需要配置MySQL，在MySQL下新建名为 **seata** 的库。



4. 在seata库里建表

建表的sql文件 db_store.sql 在 seata/conf 目录下



5. 修改registry配置文件

在 seata/conf 目录下有个配置文件 registry.conf , 我们需要修改这个配置文件。

原配置文件:

```
1  registry {
2      # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
3      type = "file"
4
5      nacos {
6          serverAddr = "localhost"
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     addressWaitTime = "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 }

```

修改后：（主要是指明注册中心为nacos，以及修改nacos连接信息）

```

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

  nacos {
    serverAddr = "localhost:8848"
    namespace = ""
    cluster = "default"
  }

  eureka {
    serviceUrl = "http://localhost:8761/eureka"
    application = "default"
    weight = "1"
  }

  redis {
  }
}

```

6. 启动测试

1. 在启动seata之前，先启动nacos
2. 启动seata：点击 **seata/bin** 目录下的seata-server.bat

启动成功

```
2022-05-20 12:53:09.365 INFO [main]io.seata.common.loader EnhancedServiceLoader.loadFile:236 -load DataSourceGenerator[dbcp] extension by class[io.seata.server.store.db.DbcpDataSourceGenerator]
2022-05-20 12:53:09.616 INFO [main]io.seata.common.loader EnhancedServiceLoader.loadFile:236 -load LogStore[DB] extension by class[io.seata.core.store.db.LogStoreDataBaseDAO]
2022-05-20 12:53:09.616 INFO [main]io.seata.common.loader EnhancedServiceLoader.loadFile:236 -load TransactionStoreManager[DB] extension by class[io.seata.server.store.db.DatabaseTransactionStoreManager]
2022-05-20 12:53:09.617 INFO [main]io.seata.common.loader EnhancedServiceLoader.loadFile:236 -load SessionManager[DB] extension by class[io.seata.server.session.db.DataBaseSessionManager]
2022-05-20 12:53:10.189 INFO [main]io.seata.core.rpc.netty.AbstractRpcRemotingServer.start:156 -Server started ...
2022-05-20 12:53:10.197 INFO [main]io.seata.common.loader EnhancedServiceLoader.loadFile:236 -load RegistryProvider[Nacos] extension by class[io.seata.discovery.registry.nacos.NacosRegistryProvider]
```

订单/库存/账户业务数据库准备

说明

1. 以下演示都需要先启动 Nacos 后 启动 Seata。保证两个都成功启动。
2. 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。

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;
```

创建业务表

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
3     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
4
5     `user_id` BIGINT(11) DEFAULT NULL COMMENT '用户id',
6
7     `product_id` BIGINT(11) DEFAULT NULL COMMENT '产品id',
8
9     `count` INT(11) DEFAULT NULL COMMENT '数量',
10
11    `money` DECIMAL(11,0) DEFAULT NULL COMMENT '金额',
12
13    `status` INT(1) DEFAULT NULL COMMENT '订单状态: 0: 创建中; 1: 已完结'
14
15 ) ENGINE=INNODB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
16
```

- seata_storage库下建t_storage表

```
1 CREATE TABLE t_storage (
2
3     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
4
5     `product_id` BIGINT(11) DEFAULT NULL COMMENT '产品id',
6
7     `total` INT(11) DEFAULT NULL COMMENT '总库存',
8
9     `used` INT(11) DEFAULT NULL COMMENT '已用库存',
10
11    `residue` INT(11) DEFAULT NULL COMMENT '剩余库存'
12
13 ) ENGINE=INNODB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
14
15 INSERT INTO seata_storage.t_storage(`id`, `product_id`, `total`, `used`,
`residue`) VALUES ('1', '1', '100', '0', '100');
```

- seata_account库下建t_account表

```
1 CREATE TABLE t_account (
2
3     `id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY COMMENT 'id',
4
5     `user_id` BIGINT(11) DEFAULT NULL COMMENT '用户id',
6
7     `total` DECIMAL(10,0) DEFAULT NULL COMMENT '总额度',
8
9     `used` DECIMAL(10,0) DEFAULT NULL COMMENT '已用余额',
10
11    `residue` DECIMAL(10,0) DEFAULT '0' COMMENT '剩余可用额度'
12
13 ) ENGINE=INNODB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
14
```

```
15  INSERT INTO seata_account.t_account(`id`, `user_id`, `total`, `used`, `residue`)
16    VALUES ('1', '1', '1000', '0', '1000');
17  SELECT * FROM t_account;
```

创建回滚表

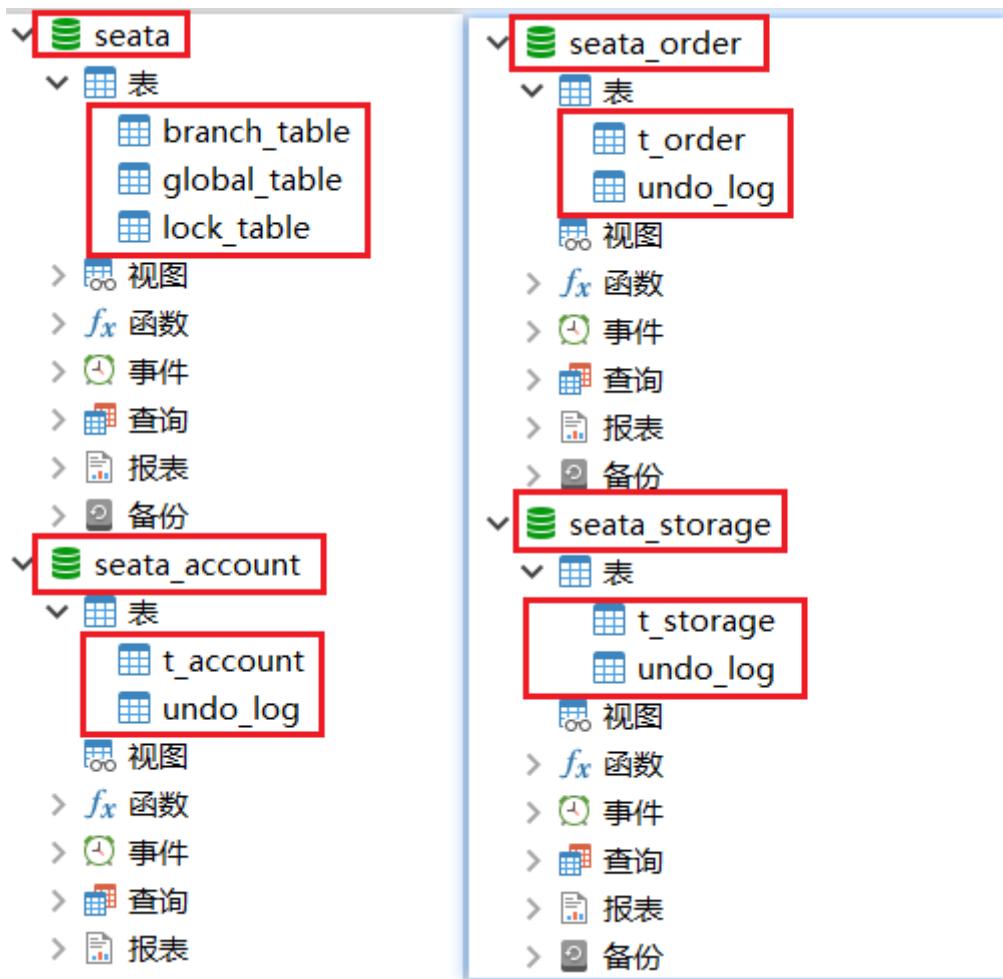
订单-库存-账户3个库下都需要建各自的回滚日志表

回滚日志表的建表sql在 `seata/conf` 目录下，文件名为 `db_undo_log.sql`

直接复制其中的sql语句创建对应的表即可。

最终效果

最终数据库有：



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

新建订单Order-Module

新建模块

新建一个模块 `seata-order-service2001`

改POM

```
1 <dependencies>
2     <!--nacos-->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5             <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
6     </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>${mysql.version}</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>${mybatis.spring.boot.version}</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>

```

写YML

```

1  server:
2      port: 2001
3  spring:
4      application:
5          name: seata-order-service
6      cloud:
7          alibaba:
8              seata:
9                  # 自定义事务组名称需要与 seata-server 中对应
10                 tx-service-group: fsp_tx_group
11             nacos:
12                 discovery:
13                     server-addr: localhost:8848
14             datasource:
15                 url: jdbc:mysql://localhost:3306/seata_order?useSSL=false
16                 driver-class-name: com.mysql.jdbc.Driver
17                 username: root
18                 password: 123456
19             feign:
20                 hystrix:
21                     enabled: false
22             logging:
23                 level:
24                     io:
25                         seata: info
26         mybatis:
27             mapperLocations: classpath:mapper/*.xml

```

file.conf

在本项目的 resources 目录下新建一个 file.conf 文件:

(注意: 这个文件的内容就是seata安装目录下的 conf 目录下的 file.conf 中的内容, 二者完全相同)

```
1  transport {
2      # tcp udt unix-domain-socket
3      type = "TCP"
4
5      #NIO NATIVE
6      server = "NIO"
7
8      #enable heartbeat
9      heartbeat = true
10
11     #thread factory for netty
12     thread-factory {
13         boss-thread-prefix = "NettyBoss"
14         worker-thread-prefix = "NettyServerNIOWorker"
15         server-executor-thread-prefix = "NettyServerBizHandler"
16         share-boss-worker = false
17         client-selector-thread-prefix = "NettyClientSelector"
18         client-selector-thread-size = 1
19         client-worker-thread-prefix = "NettyClientWorkerThread"
20
21         # netty boss thread size,will not be used for UDT
22         boss-thread-size = 1
23
24         #auto default pin or 8
25         worker-thread-size = 8
26     }
27     shutdown {
28         # when destroy server, wait seconds
29         wait = 3
30     }
31     serialization = "seata"
32     compressor = "none"
33 }
34 service {
35     vgroup_mapping.fsp_tx_group = "default" #修改自定义事务组名称
36     default.grouplist = "127.0.0.1:8091"
37     enableDegrade = false
38     disable = false
39     max.commit.retry.timeout = "-1"
40     max.rollback.retry.timeout = "-1"
41     disableGlobalTransaction = false
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     ## transaction log store
54     store {
55         ## store mode: file、db
56         mode = "db"
57
58         ## file store
59         file {
60             dir = "sessionStore"
61
62             # branch session size , if exceeded first try compress lockkey, still
63             # exceeded throws exceptions
64             max-branch-session-size = 16384
65
66             # globe session size , if exceeded throws exceptions
67             max-global-session-size = 512
68
69             # file buffer size , if exceeded allocate new buffer
70             file-write-buffer-cache-size = 16384
71
72             # when recover batch read size
73             session.reload.read_size = 100
74
75             # async, sync
76             flush-disk-mode = async
77         }
78         ## database store
79         db {
80             ## the implement of javax.sql.DataSource, such as
81             #DruidDataSource(druid)/BasicDataSource(dbcp) etc.
82             datasource = "dbcp"
83
84             ## mysql/oracle/h2/oceanbase etc.
85             db-type = "mysql"
86             driver-class-name = "com.mysql.jdbc.Driver"
87             url = "jdbc:mysql://127.0.0.1:3306/seata"
88             user = "root"
89             password = "123456"
90             min-conn = 1
91             max-conn = 3
92             global.table = "global_table"
93             branch.table = "branch_table"
94             lock-table = "lock_table"
95             query-limit = 100
96         }
97     }
98     lock {
99         ## the lock store mode: local、remote
100        mode = "remote"
101        local {
102            ## store locks in user's database
103        }
104        remote {
105            ## store locks in the seata's server
106        }
107    }
108    recovery {
109        #schedule committing retry period in milliseconds
110        committing-retry-period = 1000
```

```

109     #schedule asyn committing retry period in milliseconds
110     asyn-committing-retry-period = 1000
112
113     #schedule rollbacks retry period in milliseconds
114     rollbacks-retry-period = 1000
115
116     #schedule timeout retry period in milliseconds
117     timeout-retry-period = 1000
118 }
119 transaction {
120     undo.data.validation = true
121     undo.log.serialization = "jackson"
122     undo.log.save.days = 7
123     #schedule delete expired undo_log in milliseconds
124     undo.log.delete.period = 86400000
125     undo.log.table = "undo_log"
126 }
127 ## metrics settings
128 metrics {
129     enabled = false
130     registry-type = "compact"
131     # multi exporters use comma divided
132     exporter-list = "prometheus"
133     exporter-prometheus-port = 9898
134 }
135 support {
136     ## spring
137     spring {
138         # auto proxy the DataSource bean
139         datasource.autoproxy = false
140     }
141 }
```

registry.conf

在本项目的 resources 目录下新建一个 registry.conf 文件:

(注意: 这个文件的内容就是seata安装目录下的 conf 目录下的 registry.conf 中的内容, 二者完全相同)

```

1  registry {
2      # file , nacos , eureka, redis、zk、consul、etcd3、sofa
3      type = "nacos"
4      nacos {
5          serverAddr = "localhost:8848"
6          namespace = ""
7          cluster = "default"
8      }
9      eureka {
10         serviceUrl = "http://localhost:8761/eureka"
11         application = "default"
12         weight = "1"
13     }
14     redis {
15         serverAddr = "localhost:6379"
```

```
16     db = "0"
17 }
18 zk {
19     cluster = "default"
20     serverAddr = "127.0.0.1:2181"
21     session.timeout = 6000
22     connect.timeout = 2000
23 }
24 consul {
25     cluster = "default"
26     serverAddr = "127.0.0.1:8500"
27 }
28 etcd3 {
29     cluster = "default"
30     serverAddr = "http://localhost:2379"
31 }
32 sofa {
33     serverAddr = "127.0.0.1:9603"
34     application = "default"
35     region = "DEFAULT_ZONE"
36     datacenter = "DefaultDataCenter"
37     cluster = "default"
38     group = "SEATA_GROUP"
39     addressWaitTime = "3000"
40 }
41 file {
42     name = "file.conf"
43 }
44 }
45 config {
46     # file、nacos、apollo、zk、consul、etcd3
47     type = "file"
48     nacos {
49         serverAddr = "localhost"
50         namespace = ""
51     }
52     consul {
53         serverAddr = "127.0.0.1:8500"
54     }
55     apollo {
56         app.id = "seata-server"
57         apollo.meta = "http://192.168.1.204:8801"
58     }
59     zk {
60         serverAddr = "127.0.0.1:2181"
61         session.timeout = 6000
62         connect.timeout = 2000
63     }
64     etcd3 {
65         serverAddr = "http://localhost:2379"
66     }
67     file {
68         name = "file.conf"
69     }
70 }
```

domain

创建一个 `domain` 目录

- 在该目录下创建 `CommonResult` 类

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
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 }
```

- 在该目录下创建 `Order` 类

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class Order {
5
6      private Long id;
7      private Long userId;
8      private Long productId;
9      private Integer count;
10     private BigDecimal money;
11
12     /**
13      * 订单状态:
14      * 0: 创建中
15      * 1: 已完结
16      */
17     private Integer status;
18 }
```

Dao接口及实现

- 创建 `OrderDao` 接口

```
1  @Mapper
2  public interface OrderDao {
3
4      /**
5       * 创建订单
6       * @param order
7       */
8      void create(Order order);
9
10     /**
11      * 修改订单状态
12      * @param userId
13      */
14 }
```

```
13     * @param status
14     */
15     void update(@Param("userId") Long userId, @Param("status") Integer status);
16 }
```

2. 在 resources 目录下新建 mapper 目录，然后在其中添加xml文件

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="com.atguigu.springcloud.alibaba.dao.OrderDao">
4      <resultMap id="BaseResultMap"
5          type="com.atguigu.springcloud.alibaba.domain.Order">
6          <id column="id" property="id" jdbcType="BIGINT"/>
7          <result column="user_id" property="userId" jdbcType="BIGINT"/>
8          <result column="product_id" property="productId" jdbcType="BIGINT"/>
9          <result column="count" property="count" jdbcType="INTEGER"/>
10         <result column="money" property="money" jdbcType="DECIMAL"/>
11         <result column="status" property="status" jdbcType="INTEGER"/>
12     </resultMap>
13
14     <insert id="create">
15         insert into t_order(id, user_id, product_id, count, money, status)
16         values(null, #{userId}, #{productId}, #{count}, #{money}, 0)
17     </insert>
18
19     <update id="update">
20         update t_order set status=1 where user_id=#{userId} and status=#{status}
21     </update>
22 </mapper>
```

Service接口及实现

1. 新建 OrderService 接口

```
1  public interface OrderService {
2
3      /**
4      * 创建订单
5      * @param order
6      */
7      void create(Order order);
8  }
```

2. 新建 StorageService 接口（利用openfeign调用远程服务需要该接口）

```

1  @FeignClient(value = "seata-storage-service")
2  public interface StorageService {
3
4      /**
5       * 扣减库存
6       * @param productId
7       * @param count
8       * @return
9       */
10     @PostMapping(value = "/storage/decrease")
11     CommonResult decrease(@RequestParam("productId") Long productId,
12                           @RequestParam("count") Integer count);
13 }

```

3. 新建 AccountService 接口 (利用openfeign调用远程服务需要该接口)

```

1  @FeignClient(value = "seata-account-service")
2  public interface AccountService {
3
4      /**
5       * 扣减账户余额
6       * @param userId
7       * @param money
8       * @return
9       */
10     @PostMapping("/account/decrease")
11     CommonResult decrease(@RequestParam("userId") Long userId,
12                           @RequestParam("money") BigDecimal money);
13 }

```

4. 新建 OrderServiceImpl 作为 OrderService 的实现类

```

1  @Service
2  @Slf4j
3  public class OrderServiceImpl implements OrderService {
4
5      @Resource
6      private OrderDao orderDao;
7
8      @Resource
9      private StorageService storageService;
10
11     @Resource
12     private AccountService accountService;
13
14     /**
15      * 流程: 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
16      * @param order
17      */
18     @Override
19     public void create(Order order) {
20         log.info("----->下单开始");
21         // 本应用创建订单
22         orderDao.create(order);
23         // 远程调用库存服务扣减库存
24         log.info("----->order-service中扣减库存开始");
25         storageService.decrease(order.getProductId(), order.getCount());
26         log.info("----->order-service中扣减库存结束");

```

```
27         // 远程调用账户服务扣减余额
28         log.info("----->order-service中扣减余额开始");
29         accountService.decrease(order.getUserId(), order.getMoney());
30         log.info("----->order-service中扣减余额结束");
31         // 修改订单状态为已完成
32         log.info("----->order-service中修改订单状态开始");
33         orderDao.update(order.getUserId(), 0);
34         log.info("----->order-service中修改订单状态结束");
35         log.info("----->下单结束");
36     }
37 }
```

Controller

创建一个 `OrderController` 类

```
1  @RestController
2  public class OrderController {
3      @Resource
4      private OrderService orderService;
5      /**
6       * 创建订单
7       * 注意：这里为什么用GetMapping注解？
8       * 因为这是面向消费侧的接口，即面向用户的接口，浏览器地址栏中只能使用get方式请求，这样方便
9       * 在地址栏访问
10      * 实际在底层用的还是PostMapping
11      * @param order
12      * @return
13      */
14      @GetMapping("/order/create")
15      public CommonResult create(Order order) {
16          orderService.create(order);
17          return new CommonResult(200, "订单创建成功");
18      }
19 }
```

Config配置

1. 创建一个 `MyBatisConfig` 配置类

```
1  @Configuration
2  @MapperScan("com.atguigu.springcloud.alibaba.dao")
3  public class MyBatisConfig {
4 }
```

2. 创建一个 `DataSourceProxyConfig`

```
1  /**
2   * 使用seata进行数据源代理
3   */
4  @Configuration
5  public class DataSourceProxyConfig {
6      @Value("${mybatis.mapperLocations}")
7      private String mapperLocations;
8 }
```

```

9     @Bean
10    @ConfigurationProperties(prefix = "spring.datasource")
11    public DataSource druidDataSource() {
12        return new DruidDataSource();
13    }
14
15    @Bean
16    public DataSourceProxy dataSourceProxy(DataSource dataSource) {
17        return new DataSourceProxy(dataSource);
18    }
19
20    @Bean
21    public SqlSessionFactory sqlSessionFactoryBean(DataSourceProxy
dataSourceProxy) throws Exception {
22        SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
23        sqlSessionFactoryBean.setDataSource(dataSourceProxy);
24        sqlSessionFactoryBean.setMapperLocations(new
PathMatchingResourcePatternResolver().getResources(mapperLocations));
25        sqlSessionFactoryBean.setTransactionFactory(new
SpringManagedTransactionFactory());
26        return sqlSessionFactoryBean.getObject();
27    }
28}

```

主启动

```

1 // 取消数据源的自动创建(需要使用我们自定义的DataSourceProxyConfig对数据源进行管理)
2 @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
3 @EnableFeignClients
4 @EnableDiscoveryClient
5 public class SeataOrderMainApp2001 {
6     public static void main(String[] args) {
7         SpringApplication.run(SeataOrderMainApp2001.class, args);
8     }
9 }

```

测试

注意要先启动nacos，然后启动seata

1. 启动2001模块，如果没报错，说明搭建成功。

新建库存Storage-Module

新建模块

新建模块 seata-storage-service2002

改POM

和上一个模块相同

```
1 <dependencies>
2     <!--nacos-->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
6     </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>${mysql.version}</version>
42    </dependency>
43    <dependency>
44        <groupId>com.alibaba</groupId>
45        <artifactId>druid-spring-boot-starter</artifactId>
46        <version>1.1.10</version>
47    </dependency>
48    <dependency>
49        <groupId>org.mybatis.spring.boot</groupId>
50        <artifactId>mybatis-spring-boot-starter</artifactId>
51        <version>${mybatis.spring.boot.version}</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>
```

写YML

```
1 server:
2   port: 2002
3 spring:
4   application:
5     name: seata-storage-service
6   cloud:
7     alibaba:
8       seata:
9         tx-service-group: fsp_tx_group
10    nacos:
11      discovery:
12        server-addr: localhost:8848
13    datasource:
14      url: jdbc:mysql://localhost:3306/seata_storage?useSSL=false
15      driver-class-name: com.mysql.jdbc.Driver
16      username: root
17      password: 123456
18  logging:
19    level:
20      io:
21        seata: info
22  mybatis:
23    mapperLocations: classpath:mapper/*.xml
```

file.conf & registry.conf

和上一个模块一样，完全相同。

这里就不写了

domain

1. 创建 CommonResult 类

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

2. 创建 Storage 类

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class Storage {
5
6      private Long id;
7      /**
8       * 产品id
9       */
10     private Long productId;
11     /**
12      * 总库存
13      */
14     private Integer total;
15
16     /**
17      * 已用库存
18      */
19     private Integer used;
20
21     /**
22      * 剩余库存
23      */
24     private Integer residue;
25 }
```

Dao接口及实现

1. StorageDao

```
1  @Mapper
2  public interface StorageDao {
3
4      /**
5       * 扣减库存
6       * @param productId
7       * @param count
8       */
9      void decrease(@Param("productId") Long productId, @Param("count") Integer
10      count);
11 }
```

2. StorageMapper.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="com.atguigu.springcloud.alibaba.dao.StorageDao">
4
5      <resultMap id="BaseResultMap"
6          type="com.atguigu.springcloud.alibaba.domain.Storage">
7          <id column="id" property="id" jdbcType="BIGINT"/>
8          <result column="product_id" property="productId" jdbcType="BIGINT"/>
9          <result column="total" property="total" jdbcType="INTEGER"/>
10         <result column="used" property="used" jdbcType="INTEGER"/>
11         <result column="residue" property="residue" jdbcType="INTEGER"/>
12     </resultMap>
13     <update id="decrease">
14         update t_storage
15         set used=used+#{count}, residue=residue-#{count}
16         where product_id=#{productId}
17     </update>
18 </mapper>
```

Service接口及实现

1. 创建 StorageService 接口

```
1  public interface StorageService {
2      /**
3      * 扣减库存
4      * @param productId
5      * @param count
6      */
7      void decrease(Long productId, Integer count);
8  }
```

2. 创建 StorageServiceImpl

```
1  @Service
2  public class StorageServiceImpl implements StorageService {
3
4      private static final Logger LOGGER =
5      LoggerFactory.getLogger(StorageServiceImpl.class);
6
7      @Resource
8      private StorageDao storageDao;
9
10     @Override
11     public void decrease(Long productId, Integer count) {
12         LOGGER.info("----->storage-service中扣减库存开始");
13         storageDao.decrease(productId, count);
14         LOGGER.info("----->storage-service中扣减库存结束");
15     }
16 }
```

Controller

```
1  @RestController
2  public class StorageController {
3      @Resource
4      private StorageService storageService;
5
6      /**
7       * 扣减库存
8       * @param productId
9       * @param count
10      * @return
11     */
12     @RequestMapping("/storage/decrease")
13     public CommonResult decrease(Long productId, Integer count) {
14         storageService.decrease(productId, count);
15         return new CommonResult(200, "扣减库存成功!");
16     }
17 }
```

config配置

和上一个模块相同。

这里不再重复描述

主启动

```
1  @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
2  @EnableDiscoveryClient
3  @EnableFeignClients
4  public class SeataStorageServiceApplication2002 {
5      public static void main(String[] args) {
6          SpringApplication.run(SeataStorageServiceApplication2002.class, args);
7      }
8 }
```

测试

注意：在启动前需要先启动nacos，然后启动seata

1. 启动2002模块。启动成功，无报错，则说明搭建成功

新建账户Account-Module

新建模块

新建 seata-account-service2003

改POM

和上一个模块相同

```
1 <dependencies>
2     <!--nacos-->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
6     </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>${mysql.version}</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>${mybatis.spring.boot.version}</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>
```

写YML

```
1  server:
2    port: 2003
3  spring:
4    application:
5      name: seata-account-service
6    cloud:
7      alibaba:
8        seata:
9          tx-service-group: fsp_tx_group
10   nacos:
11     discovery:
12       server-addr: localhost:8848
13   datasource:
14     url: jdbc:mysql://localhost:3306/seata_account?useSSL=false
15     driver-class-name: com.mysql.jdbc.Driver
16     username: root
17     password: 123456
18   feign:
19     hystrix:
20       enabled: false
21   logging:
22     level:
23       io:
24         seata: info
25   mybatis:
26     mapperLocations: classpath:mapper/*.xml
```

file.conf & registry.conf

和上述模块的相同。

不重复描述了

domain

1. CommonResult

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
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 }
```

2. Account

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class Account {
5      private Long id;
6      /**
7       * 用户id
8       */
9      private Long userId;
10     /**
11      * 总额度
12      */
13     private BigDecimal total;
14     /**
15      * 已用额度
16      */
17     private BigDecimal used;
18     /**
19      * 剩余额度
20      */
21     private BigDecimal residue;
22 }
```

Dao接口及实现

1. AccountDao 接口

```
1  @Mapper
2  public interface AccountDao {
3      /**
4       * 扣减账户余额
5       * @param userId
6       * @param money
7       */
8      void decrease(@Param("userId") Long userId, @Param("money") BigDecimal money);
9 }
```

2. AccountMapper.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="com.atguigu.springcloud.alibaba.dao.AccountDao">
4
5      <resultMap id="BaseResultMap"
6          type="com.atguigu.springcloud.alibaba.domain.Account">
7          <id column="id" property="id" jdbcType="BIGINT"/>
8          <result column="user_id" property="userId" jdbcType="BIGINT"/>
9          <result column="total" property="total" jdbcType="DECIMAL"/>
10         <result column="used" property="used" jdbcType="DECIMAL"/>
11         <result column="residue" property="residue" jdbcType="DECIMAL"/>
12     </resultMap>
13
14     <update id="decrease">
15         update t_account
16         set residue=residue-#{money}, used=used+#{money}
17         where user_id=#{userId}
18     </update>
19 </mapper>
```

Service接口及实现

1. AccountService 接口

```
1  public interface AccountService {
2      /**
3      * 扣减账户余额
4      * @param userId
5      * @param money
6      */
7      void decrease(@RequestParam("userId") Long userId, @RequestParam("money")
8          BigDecimal money);
9  }
```

2. AccountServiceImpl 实现类

```
1  @Service
2  public class AccountServiceImpl implements AccountService {
3      private static final Logger LOGGER =
4          LoggerFactory.getLogger(AccountServiceImpl.class);
5
6      @Resource
7      private AccountDao accountDao;
8
9      @Override
10     public void decrease(Long userId, BigDecimal money) {
11         LOGGER.info("----->account-service中扣减账户余额开始");
12         accountDao.decrease(userId, money);
13         LOGGER.info("----->account-service中扣减账户余额结束");
14     }
15 }
```

controller

```
1  @RestController
2  public class AccountController {
3      @Resource
4      AccountService accountService;
5
6      /**
7       * 扣减账户余额
8       * @param userId
9       * @param money
10      * @return
11     */
12     @RequestMapping("/account/decrease")
13     public CommonResult decrease(@RequestParam("userId") Long userId,
14         @RequestParam("money") BigDecimal money) {
15         accountService.decrease(userId, money);
16         return new CommonResult(200, "扣减账户余额成功!");
17     }
18 }
```

Config配置

和上个模块配置相同，不再重复描述

主启动

```
1  @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
2  @EnableFeignClients
3  @EnableDiscoveryClient
4  public class SeataAccountMainApp2003 {
5      public static void main(String[] args) {
6          SpringApplication.run(SeataAccountMainApp2003.class, args);
7      }
8  }
```

测试

注意：在启动前需要先启动nacos，然后启动seata

1. 启动2003模块。启动成功，无报错，则说明搭建成功

演示seata处理分布式事务

数据库初始情况

有三个数据库

1. `seata_storage` 数据库：里面有个 `t_storage` 表

The screenshot shows the MySQL Workbench interface with the connection set to 'localhost' and the schema to 'seata_storage'. A query window contains the command: 'select * from t_storage;'. Below the query window, there are tabs for '信息', '结果 1', '剖析', and '状态'. The '结果 1' tab displays a table with the following data:

id	product_id	total	used	residue
1	1	100	0	100

2. `seata_order` 数据库：里面有个 `t_order` 表

The screenshot shows the MySQL Workbench interface with the connection set to 'localhost' and the schema to 'seata_order'. A query window contains the command: 'select * from t_order;'. Below the query window, there are tabs for '信息', '结果 1', '剖析', and '状态'. The '结果 1' tab displays a table with the following data:

id	user_id	product_id	count	money	status
(N/A)	(N/A)	(N/A)	(N/A)	(N/A)	(N/A)

3. `seata_account` 数据库：里面有个 `t_account` 表

The screenshot shows the MySQL Workbench interface with the connection set to 'localhost' and the schema to 'seata_account'. A query window contains the command: 'select * from t_account;'. Below the query window, there are tabs for '信息', '结果 1', '剖析', and '状态'. The '结果 1' tab displays a table with the following data:

id	user_id	total	used	residue
1	1	1000	0	1000

正常下单

注意：目前我们没有使用 `@GlobalTransactional` 注解！

现在我们演示正常下单的情况：

1. 访问：<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>

表示创建一个订单，`userId=1`, `productId=1`, `count=10`, `money=100`

可以看到创建订单成功。

```
← → C ⌂ ⓘ localhost:2001/order/create?userId=1&productId=1&count=10&money=100
{
  code: 200,
  message: "订单创建成功",
  data: null
}
```

2. 查看 `seata_order` 数据库中的 `t_order` 表

可以看到里面已经有订单信息

t_order @seata_order (local)						
开始事务		文本		筛选	排序	导入
id	user_id	product_id	count	money	status	
1	1	1	10	100	1	

3. 查看 `seata_account` 数据库中 `t_account` 表

t_account @seata_account (...)				
开始事务		文本		筛选
id	user_id	total	used	residue
1	1	1000	100	900

4. 查看 `seata_storage` 数据库中的 `t_storage` 表

t_storage @seata_storage (l...)				
开始事务		文本		筛选
id	product_id	total	used	residue
1	1	100	10	90

5. 查看2001服务的日志打印情况：

可以看到

```
: ----->下单开始
: ----->order-service中扣减库存开始
: Flipping property: seata-storage-service.r...
: Client: seata-storage-service instantiated
: Using serverListUpdater PollingServerListU...
: Flipping property: seata-storage-service.r...
: DynamicServerListLoadBalancer for client s...
ection failure:0;      Total blackout seconds:0
: ----->order-service中扣减库存结束
: ----->order-service中扣减余额开始
: Flipping property: seata-account-service.r...
: Client: seata-account-service instantiated
: Using serverListUpdater PollingServerListU...
: Flipping property: seata-account-service.r...
: DynamicServerListLoadBalancer for client s...
ection failure:0;      Total blackout seconds:0
: ----->order-service中扣减余额结束
: ----->order-service中修改订单状态开始
: ----->order-service中修改订单状态结束
: ----->下单结束
```

以上就是正常下单的情况。可以看到目前没什么问题。

下面演示出现异常的情况。

超时异常, 没加@GlobalTransactional

我们假设2001服务调用账户扣减服务（即调用2003服务）超时。

为了模拟这个超时，我们在2003服务的 `AccountServiceImp1` 的 `decrease` 方法中加入以下代码：

```

@Override
public void decrease(Long userId, BigDecimal money) {
    LOGGER.info("----->account-service中扣减账户余额开始 userId: " + userId + ", money: " + money);
    // 模拟超时异常
    // 暂停几秒钟线程
    try {
        Thread.sleep( millis: 20000);      // 暂停20s
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    accountDao.decrease(userId, money);
    LOGGER.info("----->account-service中扣减账户余额结束");
}

```

测试：

- 再次访问：<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>

从报错信息可以看到：我们在访问 `seata-account-service` (即2003) 服务时出现超时异常。

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun May 22 16:52:59 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

Read timed out executing POST http://seata-account-service/account/decrease?userId=1&money=100

- 检查数据库是否回滚

发现并没有：

在 `seata_order` 库的 `t_order` 表中依旧多了一条信息。 (从status字段知该信息表明订单未支付)

id	user_id	product_id	count	money	status
1	1	1	10	100	1
2	1	1	10	100	0

在 `seata_storage` 库的 `t_storage` 表中却发现库存被扣减了。

id	product_id	total	used	residue
1	1	100	20	80

在 `seata_account` 库的 `t_account` 表中发现被扣钱了

对象 t_account @seata_account (...

开始事务 文本 筛选 排序 导入 导出

	id	user_id	total	used	residue
▶	1	1	1000	200	800

从以上测试可知：

当库存和账户金额扣减后，订单状态(status)并没有设置为已经完成，没有从0改为1

注：由于feign具有重试机制，账户余额还有可能被多次扣减。

超时异常，添加@GlobalTransactional

下面我们为2001服务添加上 `@GlobalTransactional` 注解

找到2001服务中的 `OrderServiceImpl` 实现类，在 `create` 方法上添加该注解：

```

@Override
@GlobalTransactional(name = "fsp-create-order", rollbackFor = Exception.class)
public void create(Order order) {
    log.info("-----> order = " + order);
    log.info("-----> 下单开始");
    // 本应用创建订单
    orderDao.create(order);
    // 远程调用库存服务扣减库存
    log.info("----->order-service中扣减库存开始");
    storageService.decrease(order.getProductId(), order.getCount());
    log.info("----->order-service中扣减库存结束");
    // 远程调用账户服务扣减余额
    log.info("----->order-service中扣减余额开始");
    accountService.decrease(order.getUserId(), order.getMoney());
    log.info("----->order-service中扣减余额结束");
    // 修改订单状态为已完成
    log.info("----->order-service中修改订单状态开始");
    orderDao.update(order.getUserId(), status: 0);
    log.info("----->order-service中修改订单状态结束");
    log.info("----->下单结束");
}

```

测试：

1. 再次访问：<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>

依旧是超时异常

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun May 22 17:14:15 CST 2022

There was an unexpected error (type=Internal Server Error, status=500).

Read timed out executing POST http://seata-account-service/account/decrease?userId=1&money=100

2. 检查数据库

检查 seata_order 库中的 t_order 表，发现没有生成记录，还是原来的记录

对象 t_order @seata_order (local)						
开始事务		文本		筛选	排序	导入
id	user_id	product_id	count	money	status	
1	1	1	10	100	1	
2	1	1	10	100	0	

检查 seata_account 库中的 t_account 表，发现没有修改账户余额，还是原来的账户余额

对象 t_account @seata_account (local)				
开始事务		文本		筛选
id	user_id	total	used	residue
1	1	1000	200	800

检查 seata_storage 库中的 t_storage 表，发现记录没有被修改。

对象 t_storage @seata_storage (local)				
开始事务		文本		筛选
id	product_id	total	used	residue
1	1	100	20	80

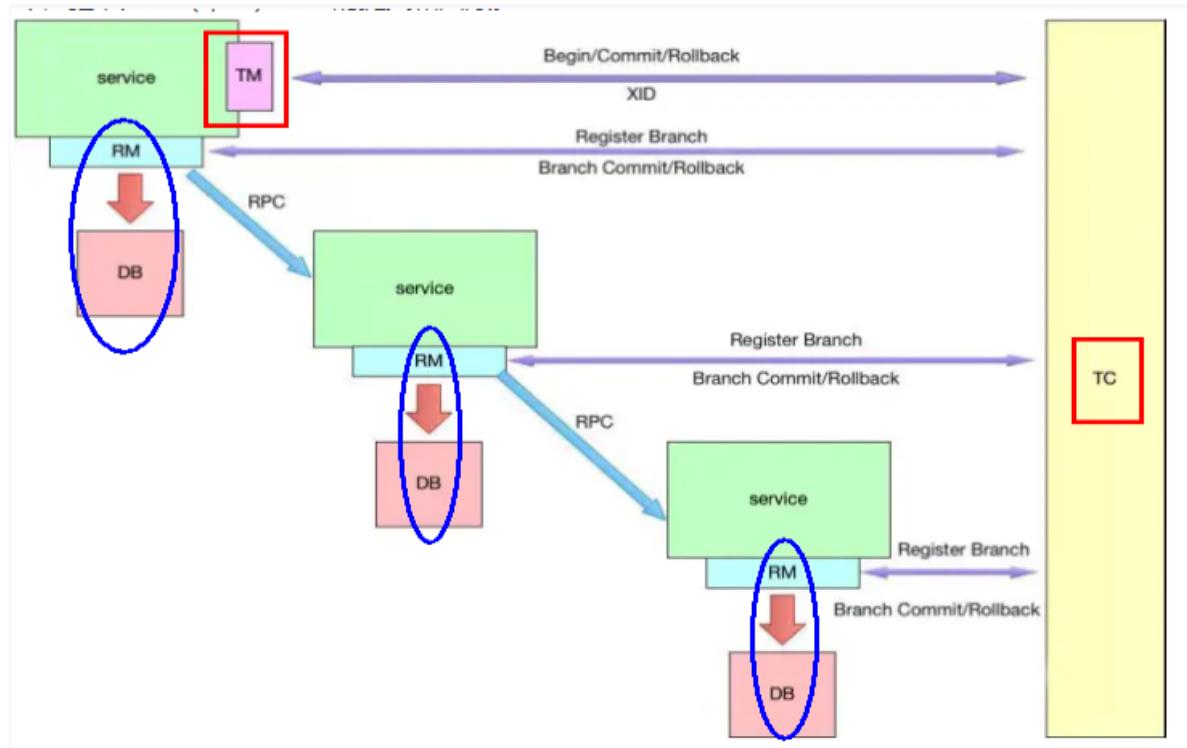
可以看到：添加 `@GlobalTransactional` 注解后实现了分布式事务的管理

补充

Seata

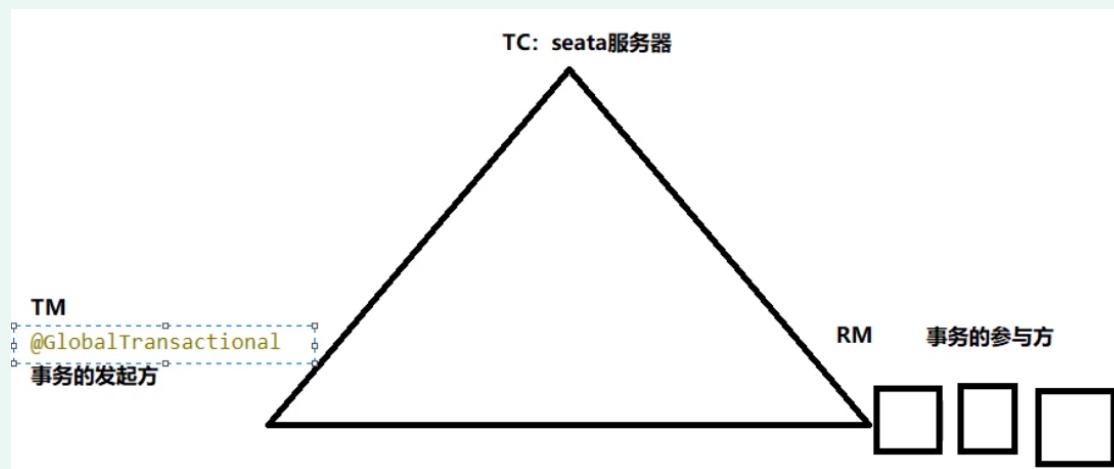
seata 全称: Simple Extensible Autonomous Transaction Architecture, 简单可扩展自治事务框架

再看 TC/TM/RM 三大组件



- **Transaction Coordinator (TC)** : 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
- **Transaction Manager (TM)** : 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。
- **Resource Manager (RM)** : 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。

TC、TM和RM三者：



分布式事务的流程：

1. TM 开启分布式事务 (TM 向 TC 注册全局事务记录)
2. 按业务场景，编排数据库、服务等事务内部资源 (RM 向 TC 汇报资源准备状态)
3. 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)
4. TC 汇总事务消息，决定分布式事务是提交还是回滚
5. TC 通知所有 RM 提交/回滚 资源，事务二阶段结束

AT 模式

AT 模式如何做到对业务的无侵入？

AT 模式是什么

<https://seata.io/zh-cn/docs/overview/what-is-seata.html>

AT 模式

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：

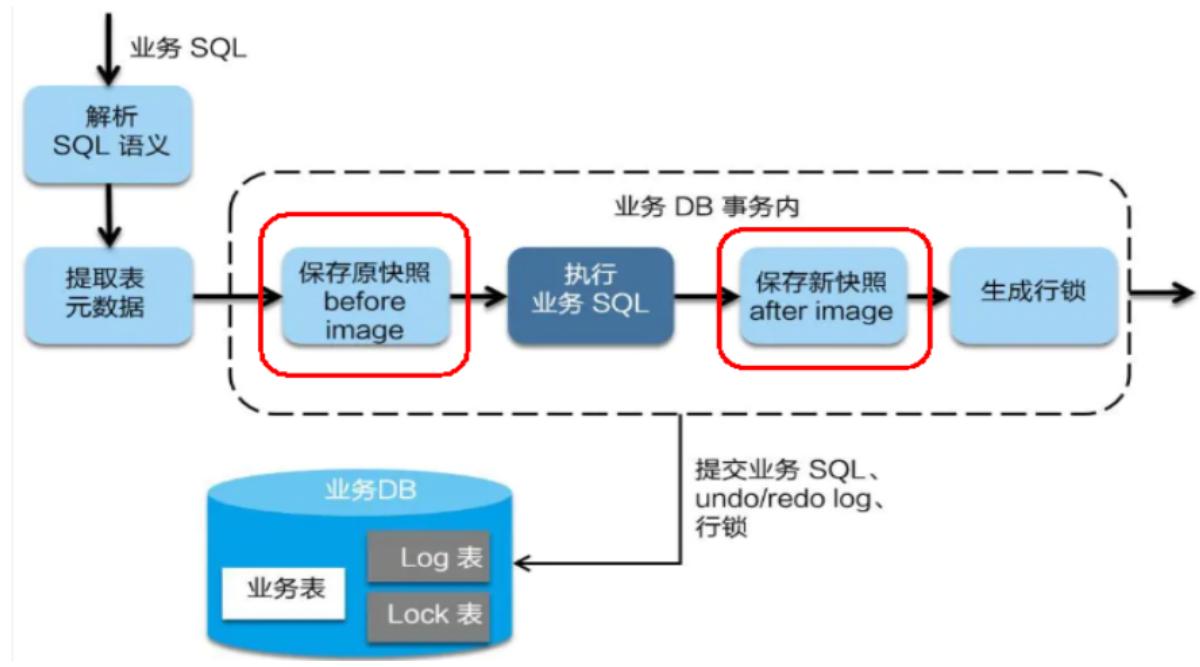
- 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。
- 二阶段：
 - 提交异步化，非常快速地完成。
 - 回滚通过一阶段的回滚日志进行反向补偿。

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

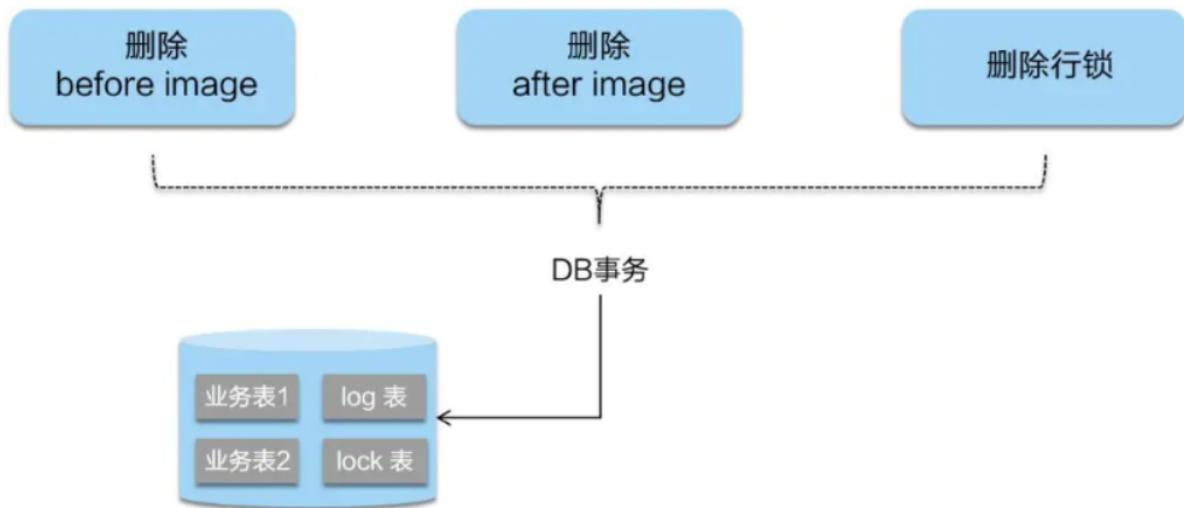
以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



二阶段提交

二阶段如果是顺利提交的话，

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。



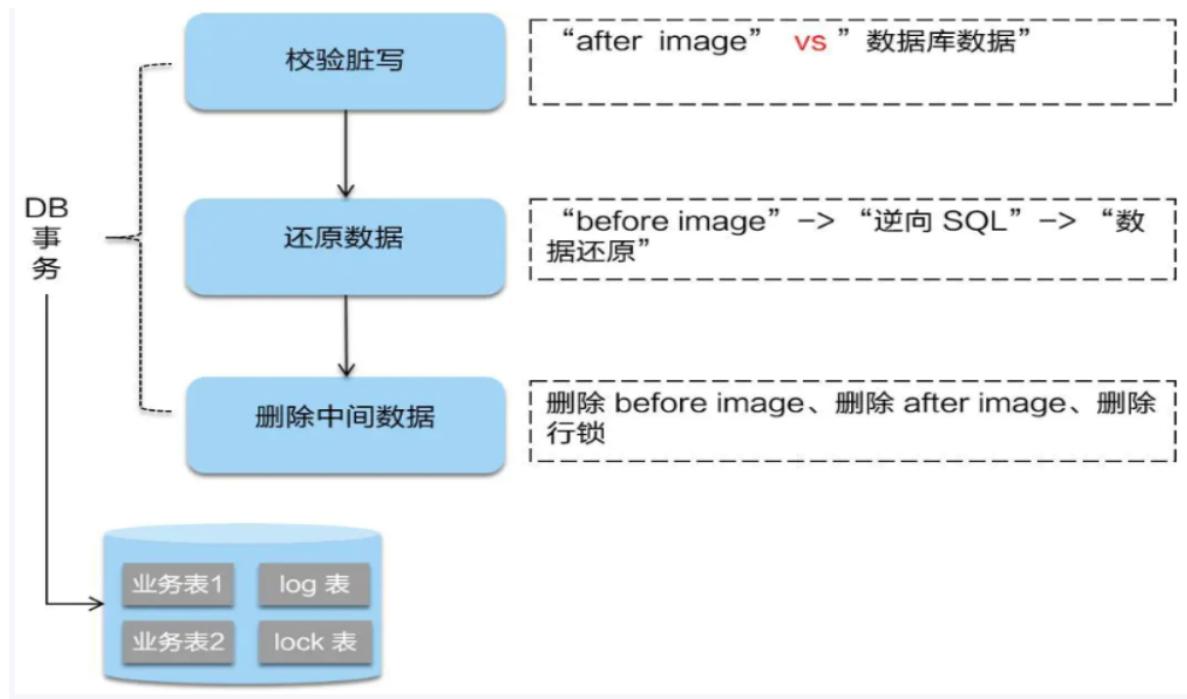
二阶段回滚

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。



debug

下面我们以debug模式来看一下。

- 首先，给2003服务的 `AccountServiceImpl` 打上断点

```

@Override
public void decrease(Long userId, BigDecimal money) {
    LOGGER.info("----->account-service中扣减账户余额开始 userId:" + userId);
    // 模拟超时异常
    // 暂停几秒钟线程
    try {
        Thread.sleep(20000);      // 暂停20s
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    accountDao.decrease(userId, money);
    LOGGER.info("----->account-service中扣减账户余额结束");
}

```

- 以debug模式运行三个服务
- 访问：<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>
- 查看 seata 库中的 `branch_table` 表

对象	branch_table @seata (localh... ip:port:transaction_id)	全局事务id	资源id	AT模式	装了seata服务器的ip
开始事务	文本	筛选	排序	导入	导出
branch_id	xid	transaction_id	resource_group_id	resource_id	
2106927690	10.1.125.73:8091:2106927688	2106927688	(null)	jdbcmysql://localhost:3306/seata_order	lock_key branch_type status client_id
2106927694	10.1.125.73:8091:2106927688	2106927688	(null)	jdbcmysql://localhost:3306/seata_storage	AT AT 2 seata-storage-service:10.1.125.73:55779
2106927698	10.1.125.73:8091:2106927688	2106927688	(null)	jdbcmysql://localhost:3306/seata_account	lock_order:4 t_storage: t_account: AT 2 seata-account-service:10.1.125.73:55751

- 查看 `seata_account` 库、`seata_order` 库以及 `seata_storage` 库中各自的 `undo_log` 表，可以发现里面都有数据。

补充

