

# SpringCloud

## 简介

### Boot与Cloud版本选型

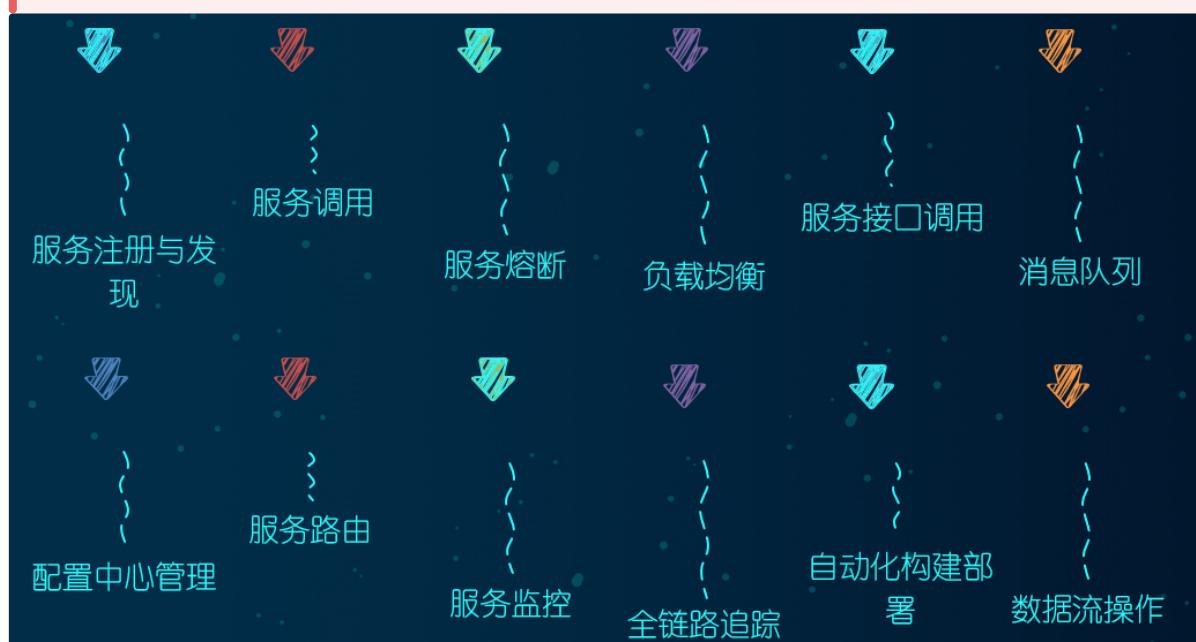
- Spring Boot
  - git源码地址: <https://github.com/spring-projects/spring-boot/releases/>
  - 官网地址: <https://spring.io/projects/spring-boot#learn>
- Spring Cloud
  - git源码地址: <https://github.com/spring-cloud/spring-cloud>
  - 官网地址: <https://spring.io/projects/spring-cloud#overview>
- Spring Cloud Alibaba
  - git源码地址: <https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E>
  - 官网地址: <https://spring-cloud-alibaba-group.github.io/github-pages/2022/zh-cn/2022.0.0.0-RC2.html>
- 版本适配匹配度,依照官网进行适配
  - SpringCloud VS SpringBoot VS SpringCloud Alibaba版本三者制约对应关系
  - 同时使用 SpringCloud 和 SpringBoot 时,由cloud决定boot版本

### 版本/组件变化

SpringCloud是基于SpringBoot之上的用来快速构建微服务系统的工具集，拥有功能完善的轻量级微服务组件，例如服务治理(Eureka),声明式REST调用(Feign),客户端负载均衡(Ribbon),服务容错(Hystrix),服务网关(Zuul)以及服务配置(Spring Cloud Config),服务跟踪(Sleuth)等等。

官网链接:<http://projects.spring.io/spring-cloud/>

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。



# SpringCloud Netflix

由于微服务的快速发展，Netflix公司的微服务又相对比较成熟，于是在技术上毫无保留的把一整套微服务架构核心技术栈开源了出来，叫做 **Netflix OSS**，也正是如此，在技术上依靠开源社区的力量不断的壮大。Pivotal在Netflix开源的一整套核心技术产品线的同时，做了一系列的封装，就变成了Spring Cloud；Spring Cloud只是集成Netflix的各个组件，实现了各种starter，例如 **Eureka-starter**、**Zuul-starter**，可以理解成简化Netflix微服务的使用，但是底层仍然是引用的Netflix，封装过后同时又可以和其他Spring Cloud产品进行无缝集成。

Netflix的各个组件在2018年是最火的一年，而后面相对来说他的很多产品已经基本上不再使用，原因就是Netflix的部分组件已经停更。再加上由于他的停更，Alibaba就采取了他的思想，研究出了 **Spring Cloud Alibaba**，而对于现在而言 **Spring Cloud Alibaba**基本上完全可以取代Netflix组件。

针对多种 Netflix 组件提供的开发工具包，其中包括 Eureka、Ribbon、Feign、Hystrix、Zuul、Archaius 等。

**Netflix Eureka**：一个基于 Rest 服务的服务治理组件，包括服务注册中心、服务注册与服务发现机制的实现，实现了云端负载均衡和中间层服务器的故障转移。

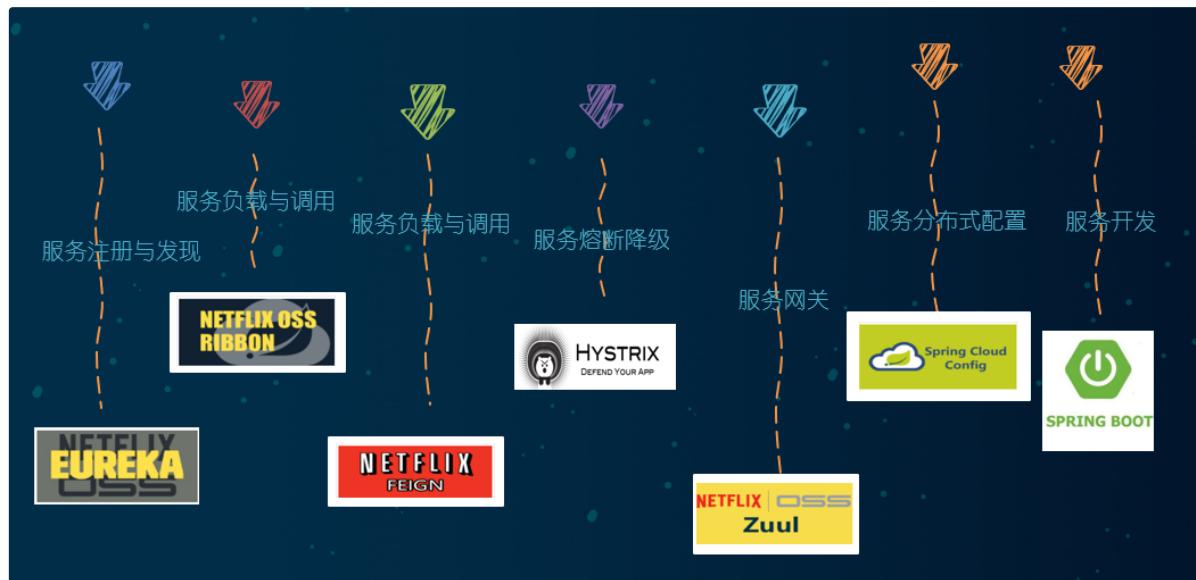
**Netflix Ribbon**：客户端负载均衡的服务调用组件。

**Netflix Hystrix**：容错管理工具，实现断路器模式，通过控制服务的节点，从而对延迟和故障提供更强大的容错能力。

**Netflix Feign**：基于 Ribbon 和 Hystrix 的声明式服务调用组件。

**Netflix zuul**：微服务网关，提供动态路由，访问过滤等服务。

**Netflix Archaius**：配置管理 API，包含一系列配置管理 API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。



## Netflix OSS被移除的原因

2020-12-22日Spring 官方博客宣布，Spring Cloud 2020.0.正式发布。2020.0.0是第一个使用新的版本号命名方案的Spring Cloud 发行版本。

本次更新却正式开启了Spring Cloud Netflix 体系的终结进程。Netflix 公司是目前微服务落地中最成功的公司。它开源了诸如Eureka、Hystrix、Zuul、Feign、Ribbon 等等广大开发者所知微服务套件，统称为 Netflix OSS。在当时Netflix OSS 成为微服务组件上事实的标准。但是在2018年Netflix 公司宣布其核心组件Hystrix、Ribbon、Zuul、Eureka 等进入维护状态，不再进行新特性开发，只修BUG。这直接影响了 Spring Cloud 项目的发展路线，Spring 官方不得不采取了应对措施，在2019年的在 SpringOne 2019 大会中，Spring Cloud 宣布 Spring Cloud Netflix项目进入维护模式，并在2020年移除相关的Netflix OSS 组件。

被移除的组件

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](#)

## Spring Cloud Alibaba

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。

### 主要功能：

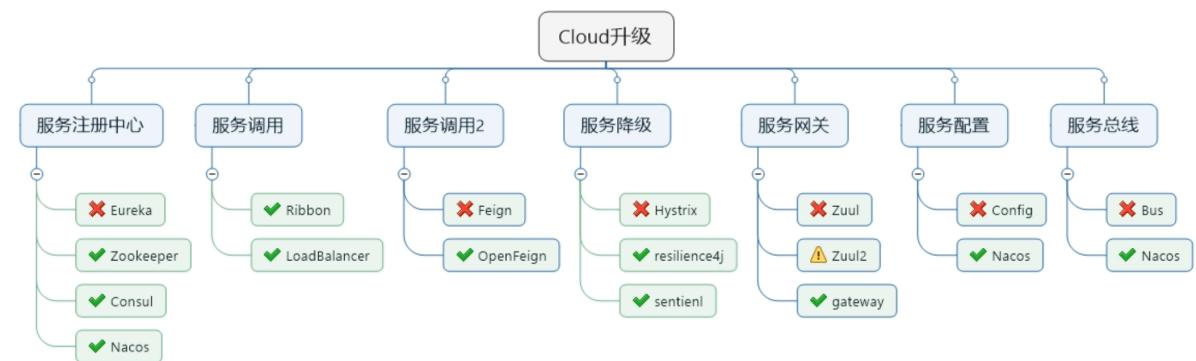
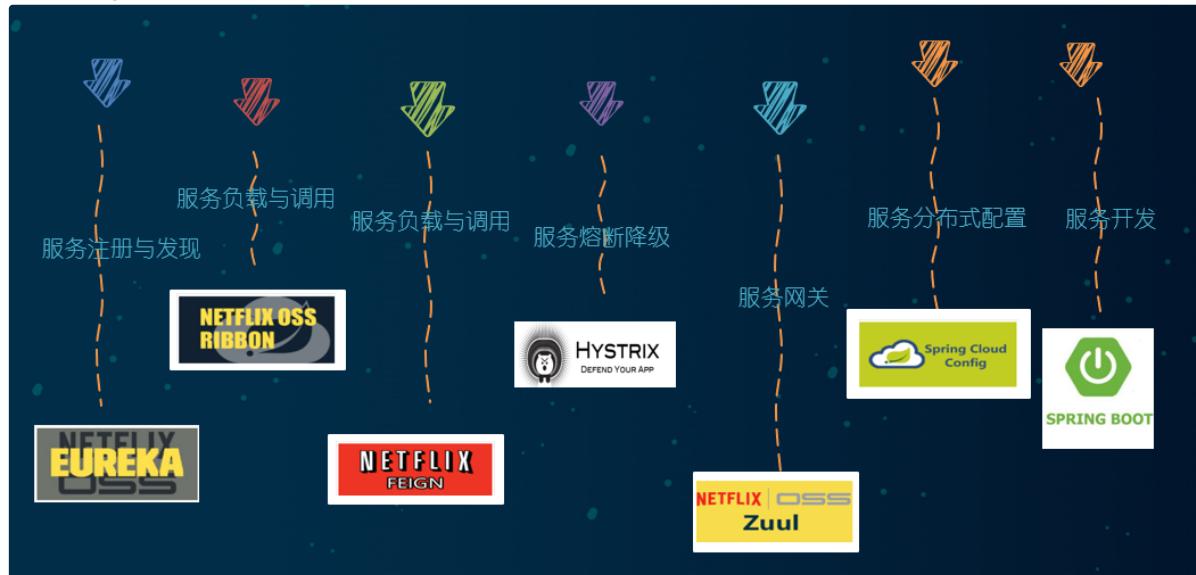
- **服务限流降级**：默认支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Spring Cloud Gateway, Zuul, Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
- **服务注册与发现**：适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持。
- **分布式配置管理**：支持分布式系统中的外部化配置，配置更改时自动刷新。
- **消息驱动能力**：基于 Spring Cloud Stream 为微服务应用构建消息驱动能力。
- **分布式事务**：使用 @GlobalTransactional 注解，高效并且对业务零侵入地解决分布式事务问题。
- **阿里云对象存储**：阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **分布式任务调度**：提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker (schedulerx-client) 上执行。
- **阿里云短信服务**：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

### 组件：

- **[Sentinel]**：阿里巴巴源产品，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。

# 组件变化

随着springCloud的不断更新呢，从一开始的Netflix的组件逐渐更新为以下组件



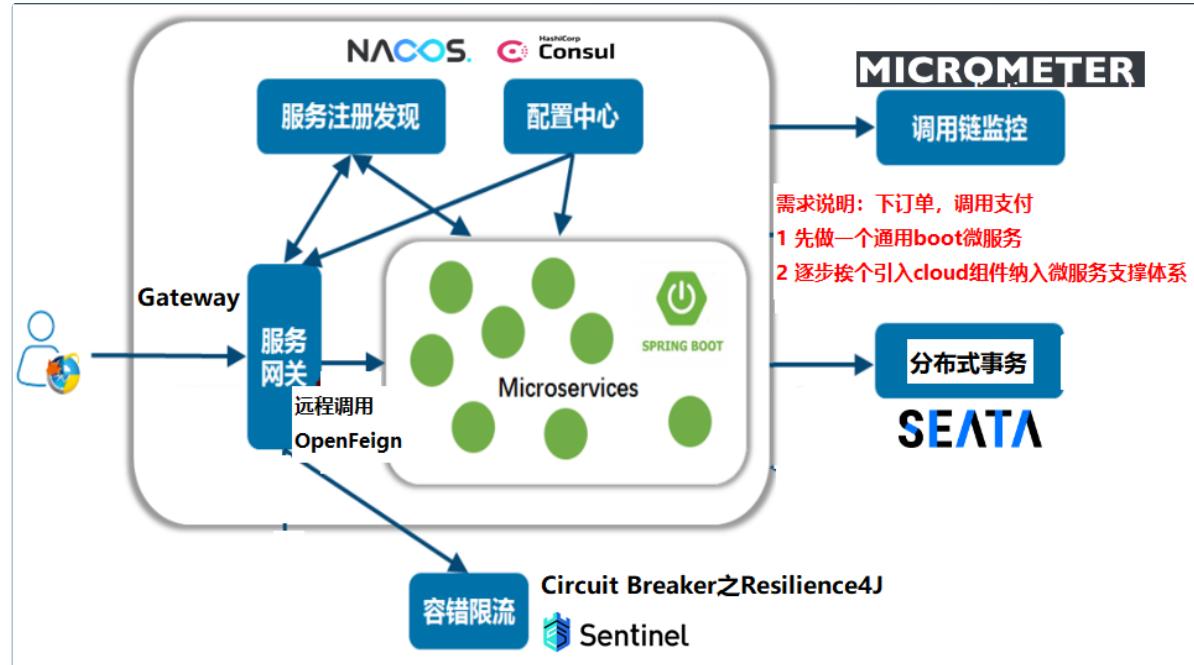
## Breaking changes

- As announced, the following modules have been removed from spring-cloud-netflix:
  - spring-cloud-netflix-archaius
  - spring-cloud-netflix-concurrency-limits
  - spring-cloud-netflix-core
  - spring-cloud-netflix-dependencies
  - spring-cloud-netflix-hystrix
  - spring-cloud-netflix-hystrix-contract
  - spring-cloud-netflix-hystrix-dashboard
  - spring-cloud-netflix-hystrix-stream
  - spring-cloud-netflix-ribbon
  - spring-cloud-netflix-sidecar
  - spring-cloud-netflix-turbine
  - spring-cloud-netflix-turbine-stream
  - spring-cloud-netflix-zuul
  - spring-cloud-starter-netflix-archaius
  - spring-cloud-starter-netflix-hystrix
  - spring-cloud-starter-netflix-hystrix-dashboard
  - spring-cloud-starter-netflix-ribbon
  - spring-cloud-starter-netflix-turbine
  - spring-cloud-starter-netflix-turbine-stream
  - spring-cloud-starter-netflix-zuul

Support for ribbon, hystrix and zuul was removed across the release train projects.

# 微服务架构编码Base工程搭建

订单->支付，业务需求说明



约定 > 配置 > 编码

## 新建Maven父工程

新建一个Project,Maven父工程

父工程pom文件内容

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

    <groupId>com.xxxxxx.cloud</groupId>
```

```
<artifactId>mscloudv5</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>

<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <hutool.version>5.8.22</hutool.version>
    <lombok.version>1.18.26</lombok.version>
    <druid.version>1.1.20</druid.version>
    <mybatis.springboot.version>3.0.2</mybatis.springboot.version>
    <mysql.version>8.0.11</mysql.version>
    <swagger3.version>2.2.0</swagger3.version>
    <mapper.version>4.2.3</mapper.version>
    <fastjson2.version>2.0.40</fastjson2.version>
    <persistence-api.version>1.0.2</persistence-api.version>
    <spring.boot.test.version>3.1.5</spring.boot.test.version>
    <spring.boot.version>3.2.0</spring.boot.version>
    <spring.cloud.version>2023.0.0</spring.cloud.version>
    <spring.cloud.alibaba.version>2022.0.0.0-
RC2</spring.cloud.alibaba.version>
</properties>

<dependencyManagement>
    <dependencies>
        <!--springboot 3.2.0-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>${spring.boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--springcloud 2023.0.0-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring.cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--springcloud alibaba 2022.0.0.0-RC2-->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring.cloud.alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--SpringBoot集成mybatis-->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis.springboot.version}</version>
        </dependency>
        <!--Mysql数据库驱动8 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
</dependency>
<!--SpringBoot集成druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>${druid.version}</version>
</dependency>
<!--通用Mapper4之tk.mybatis-->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
    <version>${mapper.version}</version>
</dependency>
<!--persistence-->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
    <version>${persistence-api.version}</version>
</dependency>
<!-- fastjson2 -->
<dependency>
    <groupId>com.alibaba.fastjson2</groupId>
    <artifactId>fastjson2</artifactId>
    <version>${fastjson2.version}</version>
</dependency>
<!-- swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html
-->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>${swagger3.version}</version>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>${hutool.version}</version>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>${lombok.version}</version>
    <optional>true</optional>
</dependency>
<!-- spring-boot-starter-test -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>${spring.boot.test.version}</version>
    <scope>test</scope>
</dependency>
</dependencies>
</dependencyManagement>
</project>
```

mysql驱动选择  
mysql5

```
# mysql5.7---JDBC四件套
jdbc.driverClass = com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/db2024?
useUnicode=true&characterEncoding=UTF-8&useSSL=false
jdbc.user = root
jdbc.password =123456
```

```
# Maven的POM文件处理
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
```

mysql8

```
# mysql8.0---JDBC四件套
jdbc.driverClass = com.mysql.cj.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/db2024?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
jdbc.user = root
jdbc.password =123456
```

```
# Maven的POM
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
```

## Mapper4一键生成

mybatis-generator  
<https://mybatis.org/generator/>  
MyBatis通用Mapper4官网  
<https://github.com/abel533/Mapper>  
mapper5官网  
<https://github.com/mybatis-mapper/mapper>

## SQL配置

db2024 库 **t\_pay** 支付信息表SQL

```
DROP TABLE IF EXISTS `t_pay`;
CREATE TABLE `t_pay` (
    `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    `pay_no` VARCHAR(50) NOT NULL COMMENT '支付流水号',
    `order_no` VARCHAR(50) NOT NULL COMMENT '订单流水号',
    `user_id` INT(10) DEFAULT '1' COMMENT '用户账号ID',
    `amount` DECIMAL(8,2) NOT NULL DEFAULT '9.9' COMMENT '交易金额',
    `deleted` TINYINT(4) UNSIGNED NOT NULL DEFAULT '0' COMMENT '删除标志, 默认0不删除, 1删除',
    `create_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    `update_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4 COMMENT='支付交易表';
```

```
INSERT INTO t_pay(pay_no,order_no) VALUES('pay17203699','6544bafb424a');
SELECT * FROM t_pay;
```

## mybatis\_generator

创建一个普通Maven工程

### POM文件

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

    <parent>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>mscloudv5</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!--我自己独一份，只是一个普通Maven工程，与boot和cloud无关-->
    <artifactId>mybatis_generator2024</artifactId>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!--Mybatis 通用mapper tk单独使用，自己独有+自带版本号-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.13</version>
        </dependency>
        <!-- Mybatis Generator 自己独有+自带版本号-->
        <dependency>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-core</artifactId>
            <version>1.4.2</version>
        </dependency>
        <!--通用Mapper-->
        <dependency>
            <groupId>tk.mybatis</groupId>
            <artifactId>mapper</artifactId>
        </dependency>
        <!--mysql8.0-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <!--persistence-->
        <dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
        </dependency>
    </dependencies>
```

```
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
</dependencies>

<build>
    <resources>
        <resource>
            <directory>${basedir}/src/main/java</directory>
            <includes>
                <include>**/*.xml</include>
            </includes>
        </resource>
        <resource>
            <directory>${basedir}/src/main/resources</directory>
        </resource>
    </resources>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.4.2</version>
            <configuration>

                <configurationFile>${basedir}/src/main/resources/generatorConfig.xml</configurationFile>
                    <overwrite>true</overwrite>
                    <verbose>true</verbose>
                </configuration>
            <dependencies>
```

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
    <version>4.2.3</version>
</dependency>
</dependencies>
</plugin>
</plugins>
</build>
</project>
```

## 配置

`src\main\resources` 路径下新建配置文件

### config.properties

- mysql5配置

```
#User表包名
package.name=com.xxx.cloud
# mysql5.7
jdbc.driverClass = com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/db2024?
useUnicode=true&characterEncoding=UTF-8&useSSL=false
jdbc.user = root
jdbc.password =123456
```

- mysql8配置

```
#t_pay表包名
package.name=com.xxx.cloud
# mysql8.0
jdbc.driverClass = com.mysql.cj.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/db2024?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
jdbc.user = root
jdbc.password =123456
```

### generatorConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
    <properties resource="config.properties"/>
    <context id="Mysql" targetRuntime="MyBatis3Simple" defaultModelType="flat">
        <property name="beginningDelimiter" value="`"/>
        <property name="endingDelimiter" value="`"/>
        <plugin type="tk.mybatis.mapper.generator.MapperPlugin">
            <property name="mappers" value="tk.mybatis.mapper.common.Mapper"/>
            <property name="caseSensitive" value="true"/>
        </plugin>
    </context>
</generatorConfiguration>
```

```

<jdbcConnection driverClass="${jdbc.driverClass}"
    connectionURL="${jdbc.url}"
    userId="${jdbc.user}"
    password="${jdbc.password}">
</jdbcConnection>
<javaModelGenerator targetPackage="${package.name}.entities"
targetProject="src/main/java"/>
<sqlMapGenerator targetPackage="${package.name}.mapper"
targetProject="src/main/java"/>
<javaClientGenerator targetPackage="${package.name}.mapper"
targetProject="src/main/java" type="XMLMAPPER"/>
<table tableName="t_pay" domainObjectName="Pay">
    <generatedKey column="id" sqlStatement="JDBC"/>
</table>
</context>
</generatorConfiguration>

```

完成以上配置后，在maven中plugins中已经有插件mybatis-generator:generator插件，点击一键生成生成entity+mapper接口 + xml实现SQL

## Rest通用Base工程构建

### 微服务提供支付Module模块(8001)

cloud-provider-payment8001  
新建module，普通Maven模块 **cloud-provider-payment8001**

#### POM文件

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>mscloudv5</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <artifactId>cloud-provider-payment8001</artifactId>
    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <!--SpringBoot通用依赖模块-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>

```

```
<groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--SpringBoot集成druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
</dependency>
<!-- Swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
<!--mybatis和springboot整合-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
</dependency>
<!--Mysql数据库驱动8 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!--persistence-->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
</dependency>
<!--通用Mapper4-->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!-- fastjson2 -->
<dependency>
    <groupId>com.alibaba.fastjson2</groupId>
    <artifactId>fastjson2</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
```

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

## YML配置

```
server:
  port: 8001

# =====applicationName + druid-mysql8 driver=====
spring:
  application:
    name: cloud-payment-service

  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db2024?
    characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
    username: root
    password: 123456

# =====mybatis=====
mybatis:
  mapper-locations: classpath:mapper/*.xml
  type-aliases-package: com.xxx.cloud.entities
  configuration:
    map-underscore-to-camel-case: true
```

## 业务类

### entities 实体类

主实体类 Pay

```
/**
 * 表名: t_pay
 * 表注释: 支付交易表
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "t_pay")
public class Pay {
    @Id
    @GeneratedValue(generator = "JDBC")
    private Integer id;
    // 支付流水号
    @Column(name = "pay_no")
    private String payNo;
    // 订单流水号
    @Column(name = "order_no")
    private String orderNo;
    // 用户账号ID
}
```

```
@Column(name = "user_id")
private Integer userId;
//交易金额
private BigDecimal amount;
//删除标志， 默认0不删除， 1删除
private Byte deleted;
//创建时间
@Column(name = "create_time")
private Date createTime;
//更新时间
@Column(name = "update_time")
private Date updateTime;
}
```

传递数值 PayDTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PayDTO implements Serializable
{
    private Integer id;
    //支付流水号
    private String payNo;
    //订单流水号
    private String orderNo;
    //用户账号ID
    private Integer userId;
    //交易金额
    private BigDecimal amount;
}
```

## mapper

将之前在mybatis-generator中的mapper和entities都复制到这里即可  
**PayMapper 接口**

```
public interface PayMapper extends Mapper<Pay> { }
```

## 映射文件PayMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxx.cloud.mapper.PayMapper">
    <resultMap id="BaseResultMap" type="com.xxx.cloud.entities.Pay">
        <!--
            WARNING - @mbg.generated
        -->
        <id column="id" jdbcType="INTEGER" property="id" />
        <result column="pay_no" jdbcType="VARCHAR" property="payNo" />
        <result column="order_no" jdbcType="VARCHAR" property="orderNo" />
        <result column="user_id" jdbcType="INTEGER" property="userId" />
        <result column="amount" jdbcType="DECIMAL" property="amount" />
        <result column="deleted" jdbcType="TINYINT" property="deleted" />
        <result column="create_time" jdbcType="TIMESTAMP" property="createTime" />
        <result column="update_time" jdbcType="TIMESTAMP" property="updateTime" />
    </resultMap>
</mapper>
```

## service

### PayService 服务接口

```
public interface PayService {  
    int add(Pay pay);  
    int delete(Integer id);  
    int update(Pay pay);  
    Pay getById(Integer id);  
    List<Pay> getPay();  
}
```

### PayServiceImpl 实现类

```
public class PayServiceImpl implements PayService {  
    @Resource  
    private PayMapper payMapper;  
    @Override  
    public int add(Pay pay) {  
        return payMapper.insert(pay);  
    }  
    @Override  
    public int delete(Integer id) {  
        return payMapper.deleteByPrimaryKey(id);  
    }  
    @Override  
    public int update(Pay pay) {  
        return payMapper.updateByPrimaryKeySelective(pay);  
    }  
    @Override  
    public Pay getById(Integer id) {  
        return payMapper.selectByPrimaryKey(id);  
    }  
    @Override  
    public List<Pay> getPay() {  
        return payMapper.selectAll();  
    }  
}
```

## controller

### PayController

```
@Slf4j  
@RestController  
public class PayController {  
    @Resource  
    private PayService payService;  
    @PostMapping("/pay/add")  
    public String addPay(@RequestBody Pay pay) {  
        System.out.println(pay.toString());  
        int result = payService.add(pay);  
        return "成功插入记录,返回值:" + result;  
    }  
    @DeleteMapping("/pay/del/{id}")  
    public Integer deletePay(@PathVariable("id") Integer id) {  
        return payService.delete(id);  
    }  
    @PutMapping("/pay/update")  
    public String update(@RequestBody PayDTO payDTO) {
```

```

        Pay pay = new Pay();
        BeanUtil.copyProperties(payDTO, pay);
        int result = payService.update(pay);
        return "成功修改记录: " + result;
    }
    @GetMapping("/pay/get/{id}")
    public Pay getById(@PathVariable("id") Integer id) {
        return payService.getById(id);
    }
    @GetMapping("/pay/getall")
    public List<Pay> getPayList() {
        return payService.getAll();
    }
}

```

## 改进

### 时间格式问题

使用@JsonFormat注解来格式化数据库中返回的时间格式

```

@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", timezone = "GMT+8")
private Date updateTime;

```

### 统一返回值

定义返回值标准格式,三大标配

- code: 状态值,由后端统一定义各种返回结果的状态码
- message: 描述,本次接口调用的结果描述
- data: 数据,本次返回的数据
- 扩展: timestamp,接口调用时间

#### 1. 创建枚举类 ReturnCodeEnum

分类	分类描述
1**	信息, 服务器收到请求, 需要请求者继续执行操作
2**	成功, 操作被成功接收并处理
3**	重定向, 需要进一步的操作以完成请求
4**	客户端错误, 请求包含语法错误或无法完成请求
5**	服务器错误, 服务器在处理请求的过程中发生了错误

```

@Getters
public enum ReturnCodeEnum {
    // 操作失败
    RC999("999", "操作xxx失败"),
    // 操作成功
    RC200("200", "success"),
    // 服务降级
    RC201("201", "服务开启降级保护,请稍后再试"),
    // 热点参数限流
    RC202("202", "热点参数限流,请稍后再试"),
}

```

```

// 系统规则不满足
RC203("203", "系统规则不满足要求,请稍后再试"),
// 授权规则不通过
RC204("204", "授权规则不通过,请稍后再试"),
// access_denied
RC403("403", "无访问权限,请联系管理员授予权限"),
// access_denied
RC401("401", "匿名用户访问无权限资源时的异常"),
RC404("404", "404页面找不到的异常"),
// 服务异常
RC500("500", "系统异常,请稍后再试"),
RC375("375", "数学运算异常,请稍后再试"),

INVALID_TOKEN("2001", "访问令牌不合法"),
ACCESS_DENIED("2003", "没有权限访问该资源"),
CLIENT_AUTHENTICATION_FAILED("1001", "客户端认证失败"),
USERNAME_OR_PASSWORD_ERROR("1002", "用户名或密码错误"),
BUSINESS_ERROR("1004", "业务逻辑异常"),
UNSUPPORTED_GRANT_TYPE("1003", "不支持的认证模式");

// 自定义状态码
private final String code;
// 自定义描述
private final String message;

ReturnCodeEnum(String code, String message) {
    this.code = code;
    this.message = message;
}

// 遍历枚举
public static ReturnCodeEnum getReturnCodeEnum(String code) {
    for (ReturnCodeEnum element : ReturnCodeEnum.values()) {
        if (element.getCode()
            .equalsIgnoreCase(code)) {
            return element;
        }
    }
    return null;
}

// 遍历枚举,流的方式
public static ReturnCodeEnum getReturnCodeEnumV1(String code) {
    return Arrays.stream(ReturnCodeEnum.values())
        .filter(ele -> ele.getCode()
            .equalsIgnoreCase(code))
        .findFirst()
        .orElse(null);
}
}

```

## 2. 创建统一返回对象 ResultData

```

@Data
@Accessors(chain = true)
public class ResultData<T> {
    /**
     * 结果状态 ,具体状态码参见枚举类ReturnCodeEnum.java
     */
    private String code;
}

```

```

private String message;
private T data;
private long timestamp;

public ResultData() {
    this.timestamp = System.currentTimeMillis();
}

public static <T> ResultData<T> success(T data) {
    ResultData<T> resultData = new ResultData<>();
    resultData.setCode(ReturnCodeEnum.RC200.getCode());
    resultData.setMessage(ReturnCodeEnum.RC200.getMessage());
    resultData.setData(data);
    return resultData;
}

public static <T> ResultData<T> fail(String code, String message) {
    ResultData<T> resultData = new ResultData<>();
    resultData.setCode(code);
    resultData.setMessage(message);
    return resultData;
}
}

```

Controller返回数据格式

```

@PostMapping(value = "/pay/add")
public ResultData<String> addPay(@RequestBody Pay pay) {
    int i = payService.add(pay);
    return ResultData.success("成功插入记录，返回值：" + i);
}

```

## 全局异常接入返回的标准格式

```

@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResultData<String> exception(RuntimeException e) {
        System.out.println("----come in GlobalExceptionHandler");
        log.error(e.getMessage(), e);
        return ResultData.fail(ReturnCodeEnum.RC500.getCode(), e.getMessage());
    }
}

```

## 引入微服务理念

### 微服务调用者订单Module模块(80)

cloud-consumer-order80

新建module，普通Maven模块 **Cloud-consumer-order80**

## POM文件

```
<dependencies>
    <!--web + actuator-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <!--hutool-all-->
    <dependency>
        <groupId>cn.hutool</groupId>
        <artifactId>hutool-all</artifactId>
    </dependency>
    <!--fastjson2-->
    <dependency>
        <groupId>com.alibaba.fastjson2</groupId>
        <artifactId>fastjson2</artifactId>
    </dependency>
    <!-- swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html -->
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    </dependency>
</dependencies>

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

## YML配置

```
server:
  port: 80
```

## 业务类

### entities

PayDTO,传递数值

```
/**
 * 一般而言，调用者不应该获悉服务提供者的entity资源并知道表结构关系，所以服务提供方给出的接口
 * 文档都应成为DTO
```

```

*/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PayDTO implements Serializable
{
    private Integer id;
    //支付流水号
    private String payNo;
    //订单流水号
    private String orderNo;
    //用户账号ID
    private Integer userId;
    //交易金额
    private BigDecimal amount;
}

```

`ResultData`统一返回值和`ReturnCodeEnum`从8001支付模块复制过来

## RestTemplate

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  
<https://docs.spring.io/spring-framework/docs/6.0.11/javadoc-api/org/springframework/web/client/RestTemplate.html>

(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。

### GET请求方法

```

/**
 * 使用GET方法请求指定URL，并将响应数据转换为指定类型的对象返回。
 * @param url 请求的URL
 * @param responseType 响应数据应该转换成的类型
 * @param uriVariables 用于替换URL中占位符的变量
 * @return 响应数据转换后的对象
 */
<T> T getForObject(String url, Class<T> responseType, Object... uriVariables);

/**
 * 使用GET方法请求指定URL，并将响应数据转换为指定类型的对象返回。
 * @param url 请求的URL
 * @param responseType 响应数据应该转换成的类型
 * @param uriVariables 包含用于替换URL中占位符的变量的映射
 * @return 响应数据转换后的对象
 */
<T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables);

/**
 * 使用GET方法请求指定URI，并将响应数据转换为指定类型的对象返回。
 * @param url 请求的URI
 * @param responseType 响应数据应该转换成的类型
 * @return 响应数据转换后的对象
 */
<T> T getForObject(URI url, Class<T> responseType);

/**
 * 使用GET方法请求指定URL，并将响应数据转换为指定类型的对象返回，同时保留响应状态码和头信息。
 * @param url 请求的URL

```

```
* @param responseType 响应数据应该转换成的类型
* @param urivariables 用于替换URL中占位符的变量
* @return 响应数据转换后的对象以及响应状态码和头信息
*/
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object...
urivariables);

/**
* 使用GET方法请求指定URL，并将响应数据转换为指定类型的对象返回，同时保留响应状态码和头信息。
* @param url 请求的URL
* @param responseType 响应数据应该转换成的类型
* @param urivariables 包含用于替换URL中占位符的变量的映射
* @return 响应数据转换后的对象以及响应状态码和头信息
*/
<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
Map<String, ?> urivariables);

/**
* 使用GET方法请求指定URI，并将响应数据转换为指定类型的对象返回，同时保留响应状态码和头信息。
* @param url 请求的URI
* @param responseType 响应数据应该转换成的类型
* @return 响应数据转换后的对象以及响应状态码和头信息
*/
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType);
```

## POST请求方法

```
/**
* 使用POST方法向指定URL发送请求，并将请求数据转换为指定类型的对象返回。
* @param url 请求的URL
* @param request 请求数据
* @param responseType 响应数据应该转换成的类型
* @param urivariables 用于替换URL中占位符的变量
* @return 响应数据转换后的对象
*/
<T> T postForObject(String url, @Nullable Object request, Class<T>
responseType, Object... urivariables);

/**
* 使用POST方法向指定URL发送请求，并将请求数据转换为指定类型的对象返回。
* @param url 请求的URL
* @param request 请求数据
* @param responseType 响应数据应该转换成的类型
* @param urivariables 包含用于替换URL中占位符的变量的映射
* @return 响应数据转换后的对象
*/
<T> T postForObject(String url, @Nullable Object request, Class<T>
responseType, Map<String, ?> urivariables);

/**
* 使用POST方法向指定URI发送请求，并将请求数据转换为指定类型的对象返回。
* @param url 请求的URI
* @param request 请求数据
* @param responseType 响应数据应该转换成的类型
* @return 响应数据转换后的对象
*/
<T> T postForObject(URI url, @Nullable Object request, Class<T> responseType);

/**
* 使用POST方法向指定URL发送请求，并将请求数据转换为指定类型的对象返回，同时保留响应状态码和头
* 信息。
*/
```

```

    * @param url 请求的URL
    * @param request 请求数据
    * @param responseType 响应数据应该转换成的类型
    * @param uriVariables 用于替换URL中占位符的变量
    * @return 响应数据转换后的对象以及响应状态码和头信息
   */
<T> ResponseEntity<T> postForEntity(String url, @Nullable Object request,
Class<T> responseType, Object... uriVariables);

/**
 * 使用POST方法向指定URL发送请求，并将请求数据转换为指定类型的对象返回，同时保留响应状态码和头
信息。
* @param url 请求的URL
* @param request 请求数据
* @param responseType 响应数据应该转换成的类型
* @param uriVariables 包含用于替换URL中占位符的变量的映射
* @return 响应数据转换后的对象以及响应状态码和头信息
*/
<T> ResponseEntity<T> postForEntity(String url, @Nullable Object request,
Class<T> responseType, Map<String, ?> uriVariables);

/**
 * 使用POST方法向指定URI发送请求，并将请求数据转换为指定类型的对象返回，同时保留响应状态码和头
信息。
* @param url 请求的URI
* @param request 请求数据
* @param responseType 响应数据应该转换成的类型
* @return 响应数据转换后的对象以及响应状态码和头信息
*/
<T> ResponseEntity<T> postForEntity(URI url, @Nullable Object request, Class<T>
responseType);

```

RestTemplate配置类

```

@Configuration
public class RestTemplateConfig
{
    @Bean
    public RestTemplate restTemplate()
    {
        return new RestTemplate();
    }
}

```

## Controller

```

@RestController
public class OrderController{
    public static final String PaymentSrv_URL = "http://localhost:8001";//先写
死，硬编码
    @Autowired
    private RestTemplate restTemplate;

    /**
     * 一般情况下，通过浏览器的地址栏输入url，发送的只能是get请求
     * 我们底层调用的是post方法，模拟消费者发送get请求，客户端消费者
     * 参数可以不添加@RequestBody
     * @param payDTO
     * @return
     */
}

```

```

    // 添加订单方法
    @GetMapping("/consumer/pay/add")
    public ResultData addOrder(PayDTO payDTO){
        return restTemplate.postForObject(PaymentSrv_URL +
        "/pay/add", payDTO, ResultData.class);
    }
    // 根据订单号获取订单
    @GetMapping("/consumer/pay/get/{id}")
    public ResultData getPayInfo(@PathVariable("id") Integer id){
        return restTemplate.getForObject(PaymentSrv_URL + "/pay/get/" + id,
        ResultData.class, id);
    }
    // 获取全部订单
    @GetMapping("/consumer/pay/getall")
    public ResultData getPayAll() {
        return restTemplate.getForObject(PaymentSrv_URL + "/pay/getall",
        ResultData.class);
    }
    // 删除订单
    @DeleteMapping("/consumer/pay/del/{id}")
    public ResultData delOrder(@PathVariable("id") String id) {
        ResponseEntity<ResultData> response = restTemplate.exchange(
            PaymentSrv_URL + "/pay/del/" + id,
            HttpMethod.DELETE,
            null,
            ResultData.class
        );
        ResultData body = response.getBody();
        return body;
    }
    // 修改订单
    @PutMapping("/consumer/pay/update")
    public ResultData updateOrder(@RequestBody PayDTO payDTO) {
        HttpEntity<PayDTO> httpEntity = new HttpEntity<>(payDTO);
        ResponseEntity<ResultData> response = restTemplate.exchange(
            PaymentSrv_URL + "/pay/update",
            HttpMethod.PUT,
            httpEntity,
            ResultData.class
        );
        return response.getBody();
    }
}

```

## 工程重构(cloud-api-commons)

引入工具模块, cloud-api-commons  
将对外暴露通用的组件/api/接口/工具类等

新建一个Module **cloud-api-commons**, 用来放置其他Module中重复的组件,接口和工具类

## POM文件

```

<dependencies>
    <!--SpringBoot通用依赖模块-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>

```

```

<groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
</dependencies>

```

## Entities/ResultData/Exception

Entities

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class PayDTO implements Serializable {
    private Integer id;
    //支付流水号
    private String payNo;
    //订单流水号
    private String orderNo;
    //用户账号ID
    private Integer userId;
    //交易金额
    private BigDecimal amount;
}

```

ResultData / ReturnCodeEnum

```

@Data
@Accessors(chain = true)
public class ResultData<T> {
    /**
     * 结果状态 ,具体状态码参见枚举类ReturnCodeEnum.java
     */
    private String code;
    private String message;
    private T data;
    private long timestamp;
    public ResultData() {
        this.timestamp = System.currentTimeMillis();
    }
    public static <T> ResultData<T> success(T data) {
        ResultData<T> resultData = new ResultData<>();
        resultData.setCode(ReturnCodeEnum.RC200.getCode());
        resultData.setMessage(ReturnCodeEnum.RC200.getMessage());
        resultData.setData(data);
        return resultData;
    }
    public static <T> ResultData<T> fail(String code, String message) {
        ResultData<T> resultData = new ResultData<>();
        resultData.setCode(code);
        resultData.setMessage(message);
    }
}

```

```
        return resultData;
    }
}
```

```
@Getter
public enum ReturnCodeEnum {
    // 操作失败
    RC999("999", "操作xxx失败"),
    // 操作成功
    RC200("200", "success"),
    // 服务降级
    RC201("201", "服务开启降级保护,请稍后再试"),
    // 热点参数限流
    RC202("202", "热点参数限流,请稍后再试"),
    // 系统规则不满足
    RC203("203", "系统规则不满足要求,请稍后再试"),
    // 授权规则不通过
    RC204("204", "授权规则不通过,请稍后再试"),
    // access_denied
    RC403("403", "无访问权限,请联系管理员授予权限"),
    // access_denied
    RC401("401", "匿名用户访问无权限资源时的异常"),
    RC404("404", "404页面找不到的异常"),
    // 服务异常
    RC500("500", "系统异常,请稍后再试"),
    RC375("375", "数学运算异常,请稍后再试"),
    INVALID_TOKEN("2001", "访问令牌不合法"),
    ACCESS_DENIED("2003", "没有权限访问该资源"),
    CLIENT_AUTHENTICATION_FAILED("1001", "客户端认证失败"),
    USERNAME_OR_PASSWORD_ERROR("1002", "用户名或密码错误"),
    BUSINESS_ERROR("1004", "业务逻辑异常"),
    UNSUPPORTED_GRANT_TYPE("1003", "不支持的认证模式");
    // 自定义状态码
    private final String code;
    // 自定义描述
    private final String message;
    ReturnCodeEnum(String code, String message) {
        this.code = code;
        this.message = message;
    }
    // 遍历枚举
    public static ReturnCodeEnum getReturnCodeEnum(String code) {
        for (ReturnCodeEnum element : ReturnCodeEnum.values()) {
            if (element.getCode()
                .equalsIgnoreCase(code)) {
                return element;
            }
        }
        return null;
    }
    // 遍历枚举,流的方式
    public static ReturnCodeEnum getReturnCodeEnumV1(String code) {
        return Arrays.stream(ReturnCodeEnum.values())
            .filter(ele -> ele.getCode()
                .equalsIgnoreCase(code))
            .findFirst()
            .orElse(null);
    }
}
```

Exception

```
@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResultData<String> exception(RuntimeException e) {
        System.out.println("----come in GlobalExceptionHandler");
        log.error(e.getMessage(), e);
        return ResultData.fail(ReturnCodeEnum.RC500.getCode(), e.getMessage());
    }
}
```

## 改造模块

使用maven将 `cloud-api-commons` 打包,然后将模块坐标复制到80和8001模块的pom文件中

```
<!-- 引入自己定义的api通用包 -->
<dependency>
    <groupId>com.xxx.cloud</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

## Consul(服务注册与发现)

Eureka停更不进行维护,逐渐放弃对Eureka的使用

为什么需要引入服务注册中心

为什么引入: 微服务所在的IP地址和端口号硬编码到订单微服务中, 会存在非常多的问题

1) 如果订单微服务和支付微服务的IP地址或者端口号发生了变化, 则支付微服务将变得不可用, 需要同步修改订单微服务中调用支付微服务的IP地址和端口号。

2) 如果系统中提供了多个订单微服务和支付微服务, 则无法实现微服务的负载均衡功能。

3) 如果系统需要支持更高的并发, 需要部署更多的订单微服务和支付微服务, 硬编码订单微服务则后续的维护会变得异常复杂。

所以, 在微服务开发的过程中, 需要引入服务治理功能, 实现微服务之间的动态注册与发现

## 简介

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

什么是Consul

<https://developer.hashicorp.com/consul/docs/intro>

spring consul

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

1. 可以干什么

服务发现: 提供HTTPS和DNS两种发现方式

健康检测: 支持多种方式,HTTP,TCP,Docker,Shell脚本定制化监控

KV存储: Key,Value的存储方式

多数据中心: Consul支持多数据中心

可视化Web界面

2. 下载地址

<https://developer.hashicorp.com/consul/install>

# 安装并运行Consul

<https://developer.hashicorp.com/consul/install>

进行下载对应版本的consul

下载完毕之后, 在控制台进入安装包路径进行运行查看版本信息 `consul --version`, 如果有版本信息则代表适配平台

- 启动

在控制台进入安装包路径运行, 使用开发模式启动

```
consul agent -dev
```

启动完毕之后, 在浏览器使用 `localhost:8500` 进行访问

## 服务注册与发现

在各个Module模块进行配置Consul

### 配置Module模块(80)

给服务提供者8001模块和服务消费者80模块配置Consul, 进行服务注册

#### POM

<https://docs.spring.io/spring-cloud-consul/reference/quickstart.html>

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

#### YML

```
spring:
  ####Spring Cloud Consul for Service Discovery
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
```

## 主启动类

需要在主启动类添加 `@EnableDiscoveryClient`, 开启服务发现

```
@SpringBootApplication
@EnableDiscoveryClient
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main80.class, args);
    }
}
```

## Controller

需要将消费者模块的URL的硬编码，给修改成Consul服务，需要访问Consul进行查看服务注册中心上的微服务名称

```
//public static final String PaymentSrv_URL = "http://localhost:8001"; //先写死，  
硬编码  
public static final String PaymentSrv_URL = "http://cloud-payment-service"; //服  
务注册中心上的微服务名称
```

## RestTemplate

RestTemplate需要添加`@LoadBalanced`注解，支持负载均衡才可以访问Consul注册的其他服务模块

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

## 三个注册中心异同点

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

CAP

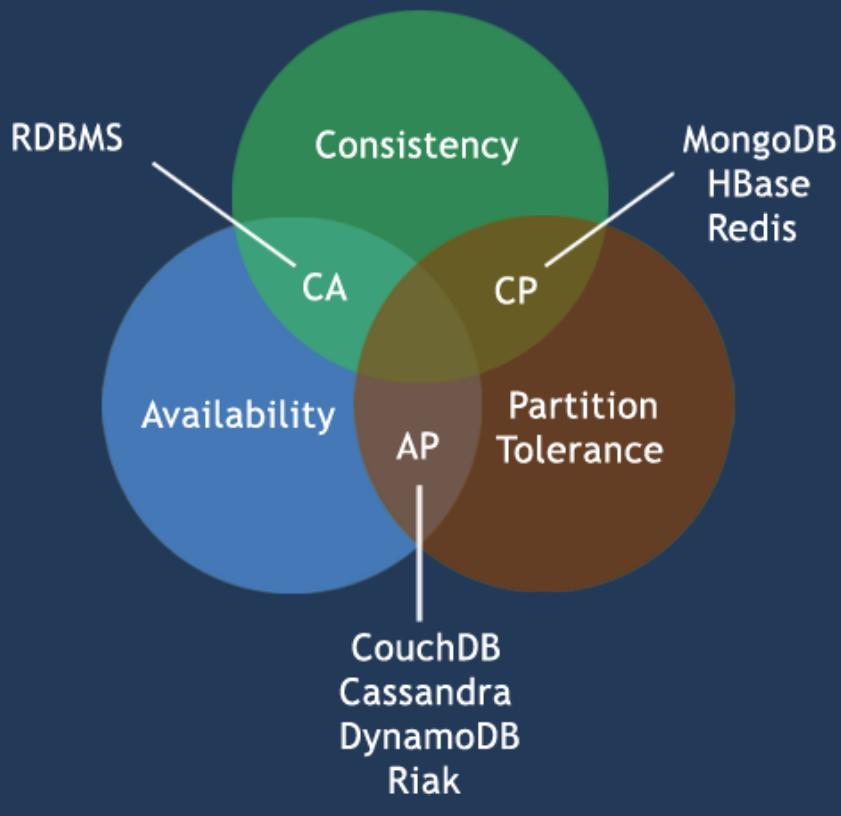
C: Consistency (强一致性.)

A: Availability (可用性)

P: Partition tolerance (分区容错性)

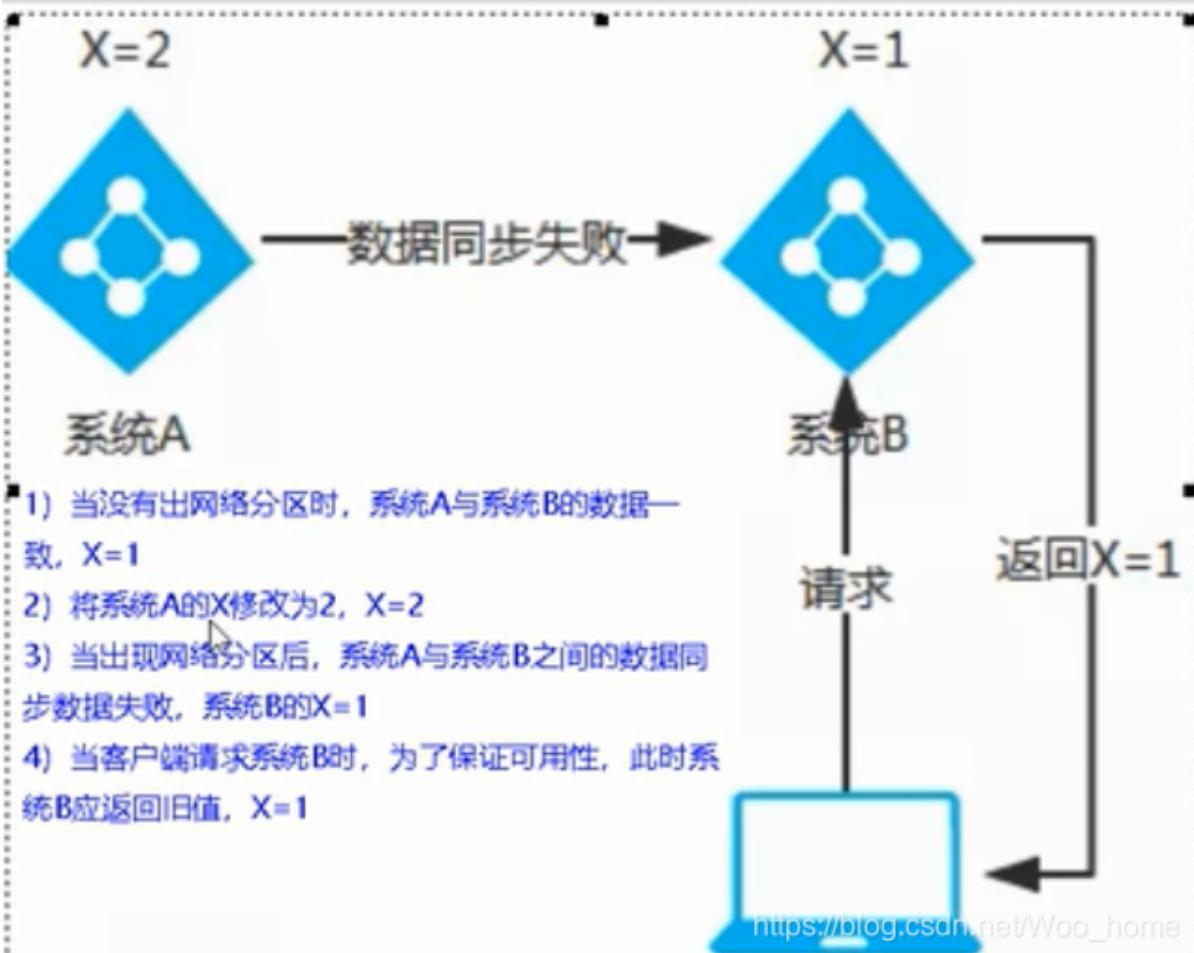
- CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三大类：
- CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。
- CP - 满足一致性，分区容忍必的系统，通常性能不是特别高。
- AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。

# CAP Theorem



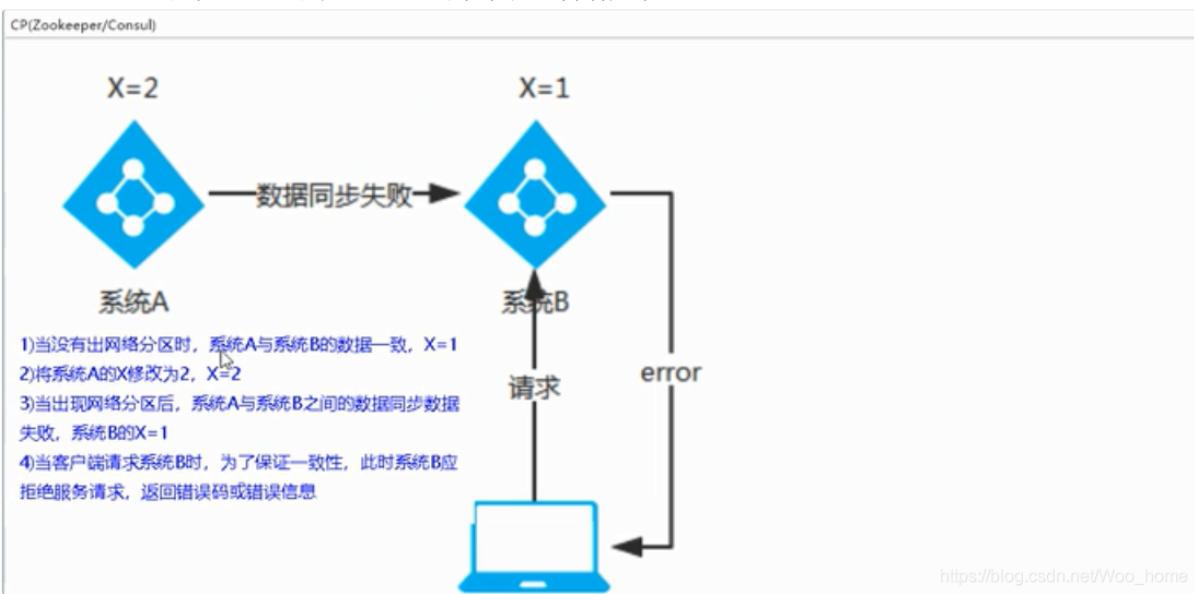
- **AP架构**

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



#### • 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 AP



## 服务配置与刷新

微服务意味着要将单体应用中的业务拆分成一个个子服务, 每个服务的粒度相对较小, 因此系统中会出现大量的服务。由于每个服务都需要必要的配置信息才能运行, 所以一套集中式的、动态的配置管理设施是必不可少的。比如某些配置文件中的内容大部分都是相同的, 只有个别的配置项不同。就拿数据库配置来说吧, 如果每个微服务使用的技术栈都是相同的, 则每个微服务中关于数据库的配置几乎都是相同的, 有时候主机迁移了, 我希望一次修改, 处处生效。

## 服务配置案例(8001)

### 需求

通用全局配置信息,直接注册进Consul服务器,从Consul获取  
修改微服务模块 `cloud-provider-payment8001`

### POM依赖

```
<!--SpringCloud consul config-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
```

### YML

- 新增配置文件bootstrap.yml文件
- application.yml是用户级的资源配置项
- bootstrap.yml是系统级的，优先级更加高
- Spring Cloud会创建一个“Bootstrap Context”，作为Spring应用的 `Application Context` 的父上下文。初始化的时候，`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`。
- `Bootstrap` 属性有高优先级，默认情况下，它们不会被本地配置覆盖。`Bootstrap context` 和 `Application Context` 有着不同的约定，所以新增了一个 `bootstrap.yml` 文件，保证 `Bootstrap Context` 和 `Application Context` 配置的分离。
- application.yml文件改为bootstrap.yml,这是很关键的或者两者共存
- 因为bootstrap.yml是比application.yml先加载的。bootstrap.yml优先级高于application.yml

#### bootstrap.yml

```
spring:
  application:
    name: cloud-payment-service
    #####Spring Cloud Consul for Service Discovery
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
      config:
        profile-separator: '-' # default value is ',', we update '-'
        format: YAML

# config/cloud-payment-service/data
#       /cloud-payment-service-dev/data
#       /cloud-payment-service-prod/data
```

#### application.yml

```

server:
  port: 8001

# =====applicationName + druid-mysql8 driver=====
spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db2024?
    characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
    username: root
    password: 123456
  profiles:
    active: dev # 多环境配置加载内容dev/prod,不写就是默认default配置

# =====mybatis=====
mybatis:
  mapper-locations:classpath:mapper/*.xml
  type-aliases-package:com.xxx.cloud.entities
  configuration:
    map-underscore-to-camel-case: true

```

## consul服务器Key/Value配置

### 1. 参考规则

```

# config/cloud-payment-service/data
#       /cloud-payment-service-dev/data
#       /cloud-payment-service-prod/data

```

### 2. 在consul服务器上的Key/Value选项中创建config/文件夹

The screenshot shows the Consul UI interface. On the left, there is a sidebar with the following navigation items: Overview, Services, Nodes, Key/Value (which is highlighted with a red border), Intentions, ACCESS CONTROLS (with a red dot), Tokens, Policies, and Roles. The main area is titled 'New Key / Value'. It has a 'Key or folder' input field containing 'config/' and a note below it: 'To create a folder, end a key with /'. At the bottom are two buttons: 'Save' (blue) and 'Cancel'.

### 3. 在config文件夹下分别创建其他三个文件夹

```

cloud-payment-service/
cloud-payment-service-dev/
cloud-payment-service-prod/

```

### 4. 在三个文件夹下分别创建data文件,data不再是文件夹

内容为:

```

project:
  info: weclome we project

```

## controller

在controller中,进行配置获取consul配置的信息,然后通过访问请求查看配置信息

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

@GetMapping(value = "/pay/get/info")
private String getInfoByConsul(@Value("${project.info}") String projectInfo)
{
    return "projectInfo: "+projectInfo+"\t"+port;
}
```

## 动态刷新

1. 在主启动类中添加注解 `@RefreshScope`,进行动态刷新
2. 在bootstrap.yml中修改 `spring.cloud.consul.config.watch.wait-time`

等待(或阻止)监视查询的秒数,默认为55。我们修改成1,则服务器和本地的更新时间为1s

```
spring.cloud.consul.config.watch.wait-time: 1
```

## consul持久化配置

### 命令行启动模式

创建一个存放consul配置的数据的目录

```
consul agent -bind=127.0.0.1 -server -bootstrap-expect 1 -ui -data-dir=consul数据目录
```

- `-dev`: 开发模式,关闭所有持久化选项,用于快速开启consul代理
- `-data-dir`: 设置指定目录,用于保存data数据
- `-ui`: 使用ui界面
- `-server`: 设置为服务器模式
- `-bootstrap-expect`: 设置服务节点,配合`-server`使用,consul会等待服务器数量达到指定值,开启集群服务,自动选举leader

## 文件启动

将一下文件配置写入bat文件,然后右键管理员进行运行,这样就会进入后台运行模式,开机也会重启consul

```
@echo. 服务启动.....
@echo off
@sc create Consul binpath= "consul安装包程序地址 agent -server -ui -bind=127.0.0.1 -client=0.0.0.0 -bootstrap-expect 1 -data-dir consul数据存储地址"
@net start Consul
@sc config Consul start= AUTO
@echo.Consul start is OK.....success
@pause
```

# LoadBalancer(负载均衡)

Ribbon目前也进入维护模式

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

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。

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

ribbon替代: spring-cloud-loadblancer

## 概述

<https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html>

### 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

### spring-cloud-starter-loadbalancer组件是什么

Spring Cloud LoadBalancer是由SpringCloud官方提供的一个开源的、简单易用的客户端负载均衡器，它包含在SpringCloud-commons中用它来替换了以前的Ribbon组件。相比较于Ribbon，SpringCloud LoadBalancer不仅能够支持RestTemplate，还支持WebClient（WebClient是Spring Web Flux中提供的功能，可以实现响应式异步请求）

- 客户端负载 VS 服务器端负载区别

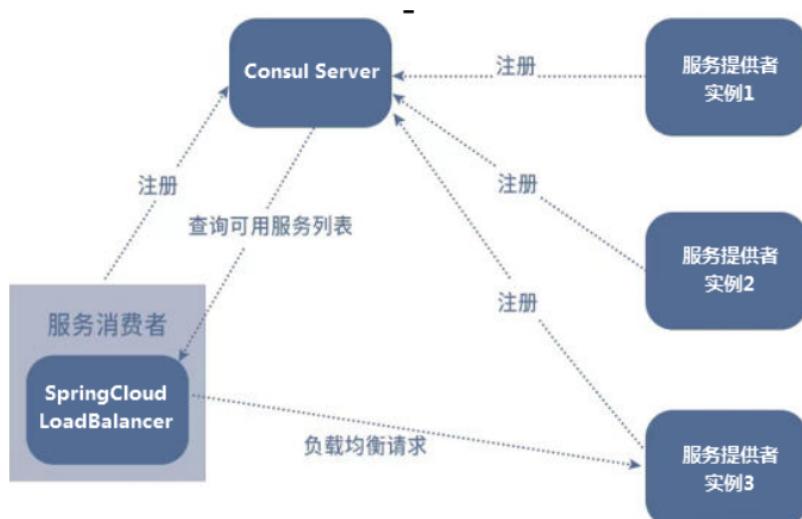
loadbalancer本地负载均衡客户端 VS Nginx服务端负载均衡区别

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

loadbalancer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。

## 负载均衡解析

### 概述



LoadBalancer 在工作时分成两步：

**第一步**，先选择ConsulServer从服务端查询并拉取服务列表，知道了它有多个服务（上图3个服务），这3个实现是完全一样的，

**第二步**，按照指定的负载均衡策略从server取到的服务注册列表中由客户端自己选择一个地址，所以LoadBalancer是一个**客户端的**负载均衡器。

# 负载均衡案例

## 使用LoadBalancer

<https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html>

### Spring Cloud LoadBalancer integrations

To make it easy to use Spring Cloud LoadBalancer, we provide `ReactorLoadBalancerExchangeFilterFunction` (which can be used with `WebClient`) and `BlockingLoadBalancerClient` (which works with `RestTemplate` and `RestClient`). You can see more information and examples of usage in the following sections:

- `Spring RestTemplate` as a LoadBalancer Client
- `Spring RestClient` as a LoadBalancer Client
- `Spring WebClient` as a LoadBalancer Client
- `Spring WebFlux WebClient` with `ReactorLoadBalancerExchangeFilterFunction`

参考: <https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html#spring-cloud-loadbalancer-integrations>

使用`@LoadBalanced`注解进行使用负载均衡

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

## 步骤 (轮询)

1. 按照8001微服务模块拷贝8002微服务模块，并修改各自的配置
2. 启动Consul，将8001、8002微服务模块启动后注册进Consul微服务
3. 将消费者订单80模块修改POM，并注册进入Consul，新增LoadBalancer组件

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

4. 在订单80模块修改Controller文件，并启动80

在controller中创建一个方法，进行访问8001或8002的controller，查看是否轮询成功

```
@GetMapping(value = "/consumer/pay/get/info")
private String getInfoByConsul(){
    return restTemplate.getForObject(PaymentSrv_URL + "/pay/get/info",
String.class);
}
```

## 总结

编码使用DiscoverClient动态获取所有上线的服务列表

```
@GetMapping("/consumer/discovery")
public String discovery() {
    // 获取所有注册在Eureka Server上的服务名称
    List<String> services = discoveryClient.getServices();
    for (String element : services) {
        System.out.println(element);
    }
    System.out.println("=====");
    // 获取指定服务名的所有实例信息
    List<ServiceInstance> instances = discoveryClient.getInstances("cloud-
payment-service");
    for (ServiceInstance element : instances) {
        System.out.println(element.getServiceId() + "\t"
            + element.getHost() + "\t"
            + element.getPort() + "\t"
            + element.getUri());
    }
    // 返回第一个实例的服务名和端口号
    return instances.get(0)
        .getServiceId() + ":"
        + instances.get(0)
            .getPort();
}
```

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<ServiceInstance> 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.....

## 负载均衡算法原理

<https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html#switching-between-the-load-balancing-algorithms>

默认为两种算法

1. 轮询

```
public class RoundRobinLoadBalancer implements
ReactorServiceInstanceLoadBalancer
```

2. 随机

```
public class RandomLoadBalancer implements ReactorServiceInstanceLoadBalancer
```

接口 `ReactiveLoadBalancer` 源码: <https://github.com/spring-cloud/spring-cloud-commons/blob/main/spring-cloud-commons/src/main/java/org/springframework/cloud/client/loadbalancer/reactive/ReactiveLoadBalancer.java>

- 算法切换

从默认的轮询，切换为随机算法，修改`RestTemplateConfig`

```
@Configuration
@LoadBalancerClient(value = "cloud-payment-service", configuration =
RestTemplateConfig.class)
public class RestTemplateConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(Environment
environment, LoadBalancerClientFactory loadBalancerClientFactory) {
        String name =
environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new
RandomLoadBalancer(loadBalancerClientFactory.getLazyProvider(name,
ServiceInstanceListSupplier.class), name);
    }
}
```

## OpenFeign(服务接口调用)

### 简介

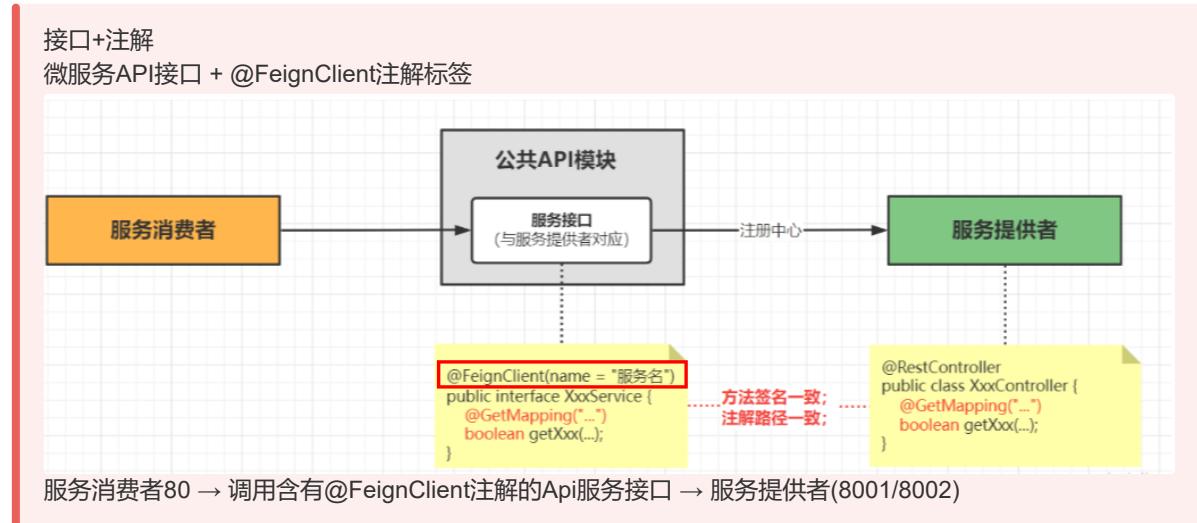
官网: <https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/#spring-cloud-feign>

GitHub: <https://github.com/spring-cloud/spring-cloud-openfeign>

#### 1. OpenFeign能干什么

- 前面在使用**SpringCloud LoadBalancer+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。
  - OpenFeign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OpenFeign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(在一个微服务接口上面标注一个`@FeignClient`注解即可)，即可完成对服务提供方的接口绑定，统一对外暴露可以被调用的接口方法，大大简化了调用客户端的开发量，也即由服务提供者给出调用接口清单，消费者直接通过OpenFeign调用即可，OpenFeign同时还集成SpringCloud LoadBalancer可以在使用OpenFeign时提供Http客户端的负载均衡，也可以集成阿里巴巴Sentinel来提供熔断、降级等功能。而与SpringCloud LoadBalancer不同的是，通过OpenFeign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。
- 可插拔的注解支持，包括Feign注解和JAX-RS注解
  - 支持可插拔的HTTP编码器和解码器
  - 支持Sentinel和它的Fallback
  - 支持SpringCloud LoadBalancer的负载均衡
  - 支持HTTP请求和响应的压缩

# 通用步骤



## 配置Module(feign80)

新建一个Module `cloud-consumer-feign-order80`,用来测试openFeign的使用

### POM

参考80POM文件,另加openfeign依赖([跳转80POM](#))

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

### YML

```
server:
  port: 80

spring:
  application:
    name: cloud-consumer-openfeign-order
  ####Spring Cloud Consul for Service Discovery
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        prefer-ip-address: true #优先使用服务ip进行注册
        service-name: ${spring.application.name}
```

### 主启动类

主启动类需要添加 `@EnableFeignClients` 注解,开启OpenFeign功能并激活

```

@SpringBootApplication
@EnableFeignClients
public class MainOpenFeign80 {
    public static void main(String[] args) {
        SpringApplication.run(MainOpenFeign80.class, args);
    }
}

```

## 业务类

按照架构图说明,并进行编码

### 修改通用模块(common)

1. 引入openfeign依赖

```

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

```

2. 新建支付服务接口 PayFeignApi,配置 @FeignClient

参考8001和8002的Controller层,请求地址需要和8001和8002Controller中请求地址一致

```

//value是Consul中的服务模块名称
@FeignClient(value = "cloud-payment-service")
public interface PayFeignApi {
    // 新增支付流水方法
    @PostMapping("/pay/add")
    ResultData addPay(PayDTO payDTO);
    //按照ID查询支付流水
    @GetMapping("/pay/get/{id}")
    ResultData getPayInfo(@PathVariable("id") Integer id);
    // 删除支付流水方法
    @DeleteMapping("/pay/del/{id}")
    ResultData deletePay(@PathVariable("id") Integer id);
    // 修改支付流水方法
    @PutMapping("/pay/update")
    ResultData updatePay(PayDTO payPTO);
    // 查询全部支付流水
    @GetMapping("/pay/getall")
    ResultData getPayInfo();
    //openfeign天然支持负载均衡演示
    @GetMapping(value = "/pay/get/info")
    String mylb();
}

```

### 修改feign80模块

仿照之前80工程Controller到 [cloud-consumer-feign-order80](#) 中

```

@RestController
@Slf4j
public class OrderController {
    @Resource
    private PayFeignApi payFeignApi;
    @PostMapping("/feign/pay/add")

```

```

public ResultData addOrder(@RequestBody PayDTO payDTO) {
    System.out.println("第一步：模拟本地addOrder新增订单成功(省略sql操作)，第二步：再开启addPay支付微服务远程调用");
    ResultData resultData = payFeignApi.addPay(payDTO);
    return resultData;
}
@GetMapping("/feign/pay/get/{id}")
public ResultData getOrderInfo(@PathVariable("id") Integer id) {
    System.out.println("-----支付微服务远程调用，按照id查询订单支付流水信息");
    ResultData payInfo = payFeignApi.getPayInfo(id);
    return payInfo;
}
@DeleteMapping("/feign/pay/del/{id}")
public ResultData deleteOrder(@PathVariable("id") Integer id) {
    System.out.println("-----支付微服务远程调用，按照id删除订单支付流水信息");
    ResultData resultData = payFeignApi.deletePay(id);
    return resultData;
}
@PutMapping("/feign/pay/update")
public ResultData updateOrder(@RequestBody PayDTO payDTO) {
    System.out.println("-----支付微服务远程调用，更新订单支付流水信息");
    ResultData resultData = payFeignApi.updatePay(payDTO);
    return resultData;
}
@GetMapping("/feign/pay/get")
public ResultData getPayInfo() {
    System.out.println("-----支付微服务远程调用，获取所有订单支付流水信息");
    ResultData payInfo = payFeignApi.getPayInfo();
    return payInfo;
}
@GetMapping(value = "/feign/pay/get/info")
private String getInfoByConsul() {
    return payFeignApi.mylb();
}
}

```

## 高级特性

### 超时控制

在Spring Cloud微服务架构中，大部分公司都是利用OpenFeign进行服务间的调用，而比较简单的业务使用默认配置是不会有多大问题的，但是如果是业务比较复杂，服务要进行比较繁杂的业务计算，那后台很有可能会出现Read Timeout这个异常，因此定制化配置超时时间就有必要了。

### 超时测试

自己测试超时设置，默认超时时间为60s

再服务提供方 **cloud-provider-payment8001** 中controller里某个方法中故意写暂停62秒程序

```

// 暂停62秒钟线程，故意写bug，测试出feign的默认调用超时时间
try {
    TimeUnit.SECONDS.sleep(62);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

再服务调用方 **cloud-consumer-feign-order80** 中controller写调用异常

```

@GetMapping("/feign/pay/get/{id}")
public ResultData getPayInfo(@PathVariable("id") Integer id)
{
    System.out.println("-----支付微服务远程调用，按照id查询订单支付流水信息");
    ResultData resultData = null;
    try
    {
        System.out.println("调用开始----:"+Dateutil.now());
        resultData = payFeignApi.getPayInfo(id);
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("调用结束----:"+Dateutil.now());
        ResultData.fail(ReturnCodeEnum.RC500.getCode(),e.getMessage());
    }
    return resultData;
}

```

## 超时配置

默认OpenFeign客户端等待60秒钟，但是服务端处理超过规定时间会导致Feign客户端返回报错。

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，默认60秒太长或者业务时间太短都不好  
yml文件中开启配置：

**connectTimeout**: 连接超时时间

**readTimeout**: 请求处理超时时间

在服务调用端中application.yml文件中进行配置

```

spring:
  cloud:
    openfeign:
      client:
        config:
          # 默认全局配置
          default:
            #连接超时时间
            connectTimeout: 3000
            #读取超时时间
            readTimeout: 3000
          # 指定服务配置
          cloud-payment-service:
            #连接超时时间
            connectTimeout: 5000
            #读取超时时间
            readTimeout: 5000

```

## 重试机制

<https://docs.spring.io/spring-cloud-openfeign/reference/spring-cloud-openfeign.html#spring-cloud-feign-overriding-defaults>

默认是关闭的，给了默认值

- 开启Retryer功能

在消费者 **cloud-consumer-feign-order80** 中新增配置类FeignConfig并修改Retryer配置

```

@Configuration
public class FeignConfig
{
    @Bean
    public Retryer myRetryer()
    {
        //return Retryer.NEVER_RETRY; //Feign默认配置是不走重试策略的

        //最大请求次数为3(1+2), 初始间隔时间为100ms, 重试间最大间隔时间为1s
        return new Retryer.Default(100,1,3);
    }
}

```

结果为总共调用3次,  $3 = 1(\text{default}) + 2$

## 默认HttpClient修改

<https://docs.spring.io/spring-cloud-openfeign/reference/spring-cloud-openfeign.html#spring-cloud-feign-overriding-defaults>

OpenFeign中http client

如果不做特殊配置, OpenFeign默认使用JDK自带的HttpURLConnection发送HTTP请求, 由于默认HttpURLConnection没有连接池、性能和效率比较低, 如果采用默认, 性能上不是最厉害的, 所以加到最大。

Apache HttpClient 5 替换 OpenFeign 默认的 HttpURLConnection

1. 修改微服务模块消费者端 `cloud-consumer-openfeign-order`
2. POM修改

```

<!-- httpclient5-->
<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
    <version>5.3</version>
</dependency>
<!-- feign-hc5-->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-hc5</artifactId>
    <version>13.1</version>
</dependency>

```

3. Apache HttpClient 5配置开启,在yaml配置文件中进行配置开启

```

# Apache HttpClient5 配置开启
spring:
  cloud:
    openfeign:
      httpclient:
        hc5:
          enabled: true

```

## 请求响应压缩

<https://docs.spring.io/spring-cloud-openfeign/reference/spring-cloud-openfeign.html#feign-and-primary>

### 1. 对请求和响应进行GZIP压缩

Spring Cloud OpenFeign支持对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。  
通过下面的两个参数设置，就能开启请求与相应的压缩功能：

```
spring.cloud.openfeign.compression.request.enabled=true  
spring.cloud.openfeign.compression.response.enabled=true
```

### 2. 细粒度化设置

对请求压缩做一些更细致的设置，比如下面的配置内容指定压缩的请求数据类型并设置了请求压缩的大小下限，只有超过这个大小的请求才会进行压缩：

```
spring.cloud.openfeign.compression.request.enabled=true  
spring.cloud.openfeign.compression.request.mime-  
types=text/xml,application/xml,application/json #触发压缩数据类型  
spring.cloud.openfeign.compression.request.min-request-size=2048 #最小触发压缩的大  
小
```

## 日志打印功能

<https://docs.spring.io/spring-cloud-openfeign/reference/spring-cloud-openfeign.html#feign-logging>

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，就是对 Feign 接口的调用情况进行监控和输出

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

在消费者的配置文件中进行配置日志 Bean

```
@Configuration  
public class FeignConfig {  
    @Bean  
    Logger.Level feignLoggerLevel() {  
        return Logger.Level.FULL;  
    }  
}
```

### • yaml 日志开启

```
# 公式(三段): logging.level + 含有@FeignClient注解的完整带包名的接口名 + debug  
# feign 日志以什么级别监控哪个接口  
logging.level.com.xxx.cloud.apis.PayFeignApi=debug
```

## CircuitBreaker(断路器)

Hystrix进入维护模式，官宣停更

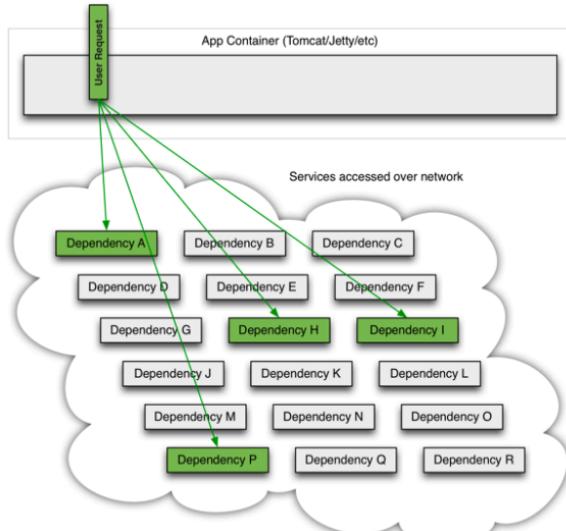
Hystrix是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

# 概述

## 简介

### 1. 分布式系统面临的问题

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



服务雪崩

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？



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。

### 2. 诉求

- 问题的节点，快速熔断（快速返回失败处理或者返回默认兜底数据【服务降级】）。
- “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。

### 3. 如何避免上述问题

- 服务熔断：当达到最大服务访问后，直接拒绝访问跳闸限电（OPEN），此时调用方会接受服务降级的处理并返回友好兜底提示，CLOSE状态提供服务，OPEN拒绝服务
- 服务降级：服务器忙，请稍后再试，不让客户端等待并立刻返回一个友好提示，fallback
- 服务限流：秒杀高并发等操作，严禁一窝蜂的过来拥挤，一秒钟N个，有序进行
- 服务限时
- 服务预热
- 接近实时的监控
- 兜底的处理动作

# Circuit Breaker是什么

<https://spring.io/projects/spring-cloud-circuitbreaker#overview>

实现原理

CircuitBreaker的目的是保护分布式系统免受故障和异常，提高系统的可用性和健壮性。

当一个组件或服务出现故障时，CircuitBreaker会迅速切换到开放OPEN状态(保险丝跳闸断电)，阻止请求发送到该组件或服务从而避免更多的请求发送到该组件或服务。这可以减少对该组件或服务的负载，防止该组件或服务进一步崩溃，并使整个系统能够继续正常运行。同时，CircuitBreaker还可以提高系统的可用性和健壮性，因为它可以在分布式系统的各个组件之间自动切换，从而避免单点故障的问题。

## Resilience4J

1. 是什么

<https://github.com/resilience4j/resilience4j?tab=readme-ov-file#1-introduction>

2. 可以干什么

<https://github.com/resilience4j/resilience4j?tab=readme-ov-file#3-overview>

3. 官网

<https://resilience4j.readme.io/docs/circuitbreaker>

4. 中文手册

<https://github.com/lmhmhl/Resilience4j-Guides-Chinese/blob/main/index.md>

## 案例实战

### 熔断(CircuitBreaker 服务熔断+服务降级)

#### 熔断器3大状态

断路器通过有限状态机实现，有三个普通状态：关闭(CLOSE)、开启(OPEN)、半开(HALF\_OPEN)，还有两个特殊状态：禁用、强制开启。

- 当熔断器关闭时，所有的请求都会通过熔断器
  - 如果失败率超过设定的阈值，熔断器就会从关闭状态转换到打开状态，这时所有的请求都会被拒绝。
  - 当经过一段时间后，熔断器会从打开状态转换到半开状态，这时仅有一定数量的请求会被放入，并重新计算失败率
  - 如果失败率超过阈值，则变成打开状态，如果失败率低于阈值，则变为关闭状态
- 熔断器使用滑动窗口来存储和统计调用的结果，你可以选择基于调用数量的滑动窗口或者基于时间的滑动窗口
  - 基于访问数量的滑动窗口统计了最近N次调用的返回结果，基于事件的滑动窗口统计了最近N秒的调用返回结果
- 初次以外，熔断器还会有两种特殊状态：DISABLED（始终允许访问）和FORCED\_OPEN（始终拒绝访问）。
  - 这两个状态不会生成熔断器事件（除状态转换外），并且不会记录事件的成功或失败
  - 退出这两个状态的唯一方法是触发状态转换或者充值熔断器

## 配置参数参考

- 官网: <https://resilience4j.readme.io/docs/circuitbreaker#create-and-configure-a-circuitbreaker>
- 中文手册: <https://github.com/lmhml/Resilience4j-Guides-Chinese/blob/main/core-modules/CircuitBreaker.md#%E5%88%9B%E5%BB%BA%E5%92%8C%E9%85%8D%E7%BD%AEcircuitbreaker>

failure-rate-threshold	以百分比配置失败率峰值
sliding-window-type	断路器的滑动窗口期类型 可以基于“次数” (COUNT_BASED) 或者“时间” (TIME_BASED) 进行熔断， 默认是COUNT_BASED。
sliding-window-size	若COUNT_BASED，则10次调用中有50%失败（即5次）打开熔断断路器；若为TIME_BASED则，此时还有额外的两个设置属性，含义为：在N秒内 (sliding-window-size) 100% (slow-call-rate-threshold) 的请求超过N秒 (slow-call-duration-threshold) 打开断路器。
slowCallRateThreshold	以百分比的方式配置，断路器把调用时间大于 slowCallDurationThreshold的调用视为慢调用，当慢调用比例大于等于峰值时，断路器开启，并进入服务降级。
slowCallDurationThreshold	配置调用时间的峰值，高于该峰值的视为慢调用。
permitted-number-of-calls-in-half-open-state	运行断路器在HALF_OPEN状态下时进行N次调用，如果故障或慢速调用仍然高于阈值，断路器再次进入打开状态。
minimum-number-of-calls	在每个滑动窗口期样本数，配置断路器计算错误率或者慢调用率的最小调用数。比如设置为5意味着，在计算故障率之前，必须至少调用5次。如果只记录了4次，即使4次都失败了，断路器也不会进入到打开状态。
wait-duration-in-open-state	从OPEN到HALF_OPEN状态需要等待的时间

## 熔断+降级案例需求说明

N次访问中当执行方法的失败率达到50%时CircuitBreaker将进入开启OPEN状态(保险丝跳闸断电)拒绝所有请求。等待N秒后，CircuitBreaker将自动从开启OPEN状态过渡到半开HALF\_OPEN状态，允许一些请求通过以测试服务是否恢复正常。

如还是异常CircuitBreaker将重新进入开启OPEN状态；如正常将进入关闭CLOSE闭合状态恢复正常处理请求。



## COUNT\_BASED(计数滑动)

1. 基于访问数量的滑动窗口是通过一个有N个元素的循环数组实现。
2. 如果滑动窗口的大小等于10，那么循环数组总是有10个统计值。滑动窗口增量更新总的统计值，随着新的调用结果被记录在环形数组中，总的统计值也随之进行更新。当环形数组满了，时间最久的元素将被驱逐，将从总的统计值中减去该元素的统计值，并该元素所在的桶进行重置。
3. 检索快照（总的统计值）的时间复杂度为O(1)，因为快照已经预先统计好了，并且和滑动窗口大小无关。

关于此方法实现的空间需求（内存消耗）为O(n)。

4. <https://github.com/lmhmhl/Resilience4j-Guides-Chinese/blob/main/core-modules/CircuitBreaker.md>

### 1. 修改 `cloud-provider-payment8001` 提供者端

新建PayCircuitController文件，用于测试

```
@RestController
public class PaycircuitController {
    //=====Resilience4j CircuitBreaker 的例子
    @GetMapping(value = "/pay/circuit/{id}")
    public String mycircuit(@PathVariable("id") Integer id) {
        if (id == -4) throw new RuntimeException("----circuit id 不能负数");
        if (id == 9999) {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return "Hello, circuit! inputId: " + id + "\t" + Idutil.simpleUUID();
    }
}
```

### 2. 修改 `cloud-api-commons` 通用模块

修改api接口PayFeignApi文件

```
/**
 * Resilience4j CircuitBreaker 的例子
 * @param id
 * @return
 */
@GetMapping(value = "/pay/circuit/{id}")
String mycircuit(@PathVariable("id") Integer id);
```

### 3. 修改 `cloud-consumer-feign-order80` 消费者端

修改POM

```

<!--resilience4j-circuitbreaker-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
<!-- 由于断路保护等需要AOP实现，所以必须导入AOP包 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>

```

修改全局配置

```

# 开启circuitbreaker和分组激活
# spring.cloud.openfeign.circuitbreaker.enabled
spring.cloud.openfeign.circuitbreaker.enabled=true
spring.cloud.openfeign.circuitbreaker.group.enabled=true

```

```

# Resilience4j CircuitBreaker 按照次数: COUNT_BASED 的例子
# 6次访问中当执行方法的失败率达到50%时CircuitBreaker将进入开启OPEN状态(保险丝跳闸断电)拒绝所有请求。
# 等待5秒后, CircuitBreaker 将自动从开启OPEN状态过渡到半开HALF_OPEN状态, 允许一些请求通过以测试服务是否恢复正常。
# 如还是异常CircuitBreaker 将重新进入开启OPEN状态; 如正常将进入关闭CLOSE闭合状态恢复正常处理请求。
resilience4j:
    circuitbreaker:
        configs:
            default:
                #设置50%的调用失败时打开断路器, 超过失败请求百分比CircuitBreaker变为OPEN状态。
                failureRateThreshold: 50
                # 滑动窗口的类型
                slidingwindowType: COUNT_BASED
                #滑动窗口的大小配置COUNT_BASED表示6个请求, 配置TIME_BASED表示6秒
                slidingWindowSize: 6
                #断路器计算失败率或慢调用率之前所需的最小样本(每个滑动窗口周期)。如果
                minimumNumberOfCalls为10, 则必须最少记录10个样本, 然后才能计算失败率。如果只记录了9次调用,
                即使所有9次调用都失败, 断路器也不会开启。
                minimumNumberOfCalls: 6
                #是否启用自动从开启状态过渡到半开状态, 默认值为true。如果启用, CircuitBreaker将自动从开启状态过渡到半开状态, 并允许一些请求通过以测试服务是否恢复正常
                automaticTransitionFromOpenToHalfOpenEnabled: true
                #从OPEN到HALF_OPEN状态需要等待的时间
                waitDurationInOpenState: 5s
                #半开状态允许的最大请求数, 默认值为10。在半开状态下, CircuitBreaker将允许最多
                permittedNumberOfCallsInHalfOpenState个请求通过, 如果其中有任何一个请求失败,
                CircuitBreaker将重新进入开启状态。
                permittedNumberOfCallsInHalfOpenState: 2
                recordExceptions:
                    - java.lang.Exception
            instances:
                cloud-payment-service:
                    baseConfig: default

```

新建OrderCircuitController

```

@RestController
public class OrderCircuitController
{
    @Resource

```

```

private PayFeignApi payFeignApi;

@GetMapping(value = "/feign/pay/circuit/{id}")
@CircuitBreaker(name = "cloud-payment-service", fallbackMethod =
"myCircuitFallback")
public String myCircuitBreaker(@PathVariable("id") Integer id)
{
    return payFeignApi.myCircuit(id);
}
//myCircuitFallback就是服务降级后的兜底处理方法
public String myCircuitFallback(Integer id, Throwable t) {
    // 这里是容错处理逻辑，返回备用结果
    return "myCircuitFallback, 系统繁忙，请稍后再试----/(ㄒoㄒ)/~~";
}
}

```

配置完成之后进行访问消费者端的请求路径进行测试完成

## TIME\_BASED(时间滑动)

1. 基于时间的滑动窗口是通过有N个桶的环形数组实现。
2. 如果滑动窗口的大小为10秒，这个环形数组总是有10个桶，每个桶统计了在这一秒发生的所有调用的结果（部分统计结果），数组中的第一个桶存储了当前这一秒内的所有调用的结果，其他的桶存储了之前每秒调用的结果。
3. 滑动窗口不会单独存储所有的调用结果，而是对每个桶内的统计结果和总的统计值进行增量的更新，当新的调用结果被记录时，总的统计值会进行增量更新。
4. 检索快照（总的统计值）的时间复杂度为O(1)，因为快照已经预先统计好了，并且和滑动窗口大小无关。
5. 关于此方法实现的空间需求（内存消耗）约等于O(n)。由于每次调用结果（元组）不会被单独存储，只是对N个桶进行单独统计和一次总分的统计。
6. 每个桶在进行部分统计时存在三个整型，为了计算，失败调用数，慢调用数，总调用数。还有一个long类型变量，存储所有调用的响应时间。
7. <https://github.com/lmhml/Resilience4j-Guides-Chinese/blob/main/core-modules/CircuitBreaker.md>

- 全局配置

```

# 开启circuitbreaker和分组激活
# spring.cloud.openfeign.circuitbreaker.enabled
spring.cloud.openfeign.circuitbreaker.enabled=true
spring.cloud.openfeign.circuitbreaker.group.enabled=true

```

```

# Resilience4j circuitBreaker 按照时间: TIME_BASED 的例子
resilience4j:
  timelimiter:
    configs:
      default:
        #timelimiter 默认限制远程1s，超于1s就超时异常，配置了降级，就走降级逻辑
        timeout-duration: 10s
  circuitbreaker:
    configs:
      default:
        # 设置50%的调用失败时打开断路器，超过失败请求百分比circuitBreaker变为OPEN状态。
        failureRateThreshold: 50
        # 慢调用时间阈值，高于这个阈值的视为慢调用并增加慢调用比例。

```

```

slowCallDurationThreshold: 2s
# 慢调用百分比峰值，断路器把调用时间大于slowCallDurationThreshold，视为慢调用，当慢调用比例高于阈值，断路器打开，并开启服务降级
slowCallRateThreshold: 30
# 滑动窗口的类型
slidingwindowType: TIME_BASED
# 滑动窗口的大小配置，配置TIME_BASED表示2秒
slidingWindowSize: 2
# 断路器计算失败率或慢调用率之前所需的最小样本(每个滑动窗口周期)。
minimumNumberOfCalls: 2
# 半开状态允许的最大请求数，默认值为10。
permittedNumberOfCallsInHalfOpenState: 2
# 从OPEN到HALF_OPEN状态需要等待的时间
waitDurationInOpenState: 5s
recordExceptions:
- java.lang.Exception
instances:
cloud-payment-service:
baseConfig: default

```

当满足一定的峰值和失败率达到一定条件后，断路器将会进入OPEN状态，服务熔断

当OPEN的时候，所有的请求都不会调用主营业务逻辑方法，而是直接fallbackMethod兜底背锅方法，服务降级  
一段时间后，断路器从OPEN进入HALF\_OPEN半开状态，会放几个请求过去探测链路是否开通，如果失败，继续开启，重复上述过程

## 隔离(BulkHead)

<https://resilience4j.readme.io/docs/bulkhead>  
<https://github.com/lmhmhl/Resilience4j-Guides-Chinese/blob/main/core-modules/bulkhead.md>

依赖隔离&负载保护：用来限制对于下游服务的最大并发数量的限制（限制并发）

Resilience4j提供了两种隔离的实现方式，可以限制并发执行数量

1. **SemaphoreBulkhead** 使用了信号量
2. **FixedThreadPoolBulkhead** 使用了有界队列和固定大小线程池

## SemaphoreBulkhead (信号量舱壁)

基本上就是我们JUC信号灯内容的同样思想 Semaphore

信号量舱壁 (SemaphoreBulkhead) 原理

1. 当信号量有空闲时，进入系统的请求会直接获取信号量并开始业务处理。
2. 当信号量全被占用时，接下来的请求将会进入阻塞状态，SemaphoreBulkhead提供了一个阻塞计时器，
3. 如果阻塞状态的请求在阻塞计时内无法获取到信号量则系统会拒绝这些请求。
4. 若请求在阻塞计时内获取到了信号量，那将直接获取信号量并执行相应的业务处理。

1. 修改 **cloud-provider-payment8001** 服务提供者

修改PayCircuitController

```

//=====Resilience4j bulkhead 的例子
@GetMapping(value = "/pay/bulkhead/{id}")
public String myBulkhead(@PathVariable("id") Integer id)
{
    if(id == -4) throw new RuntimeException("----bulkhead id 不能-4");
}

```

```

if(id == 9999)
{
    try { TimeUnit.SECONDS.sleep(5); }
    catch (InterruptedException e) { e.printStackTrace(); }
}

return "Hello, bulkhead! inputId: "+id+ "\t " + IdUtil.simpleUUID();
}

```

## 2. 修改 **cloud-common-api** 公共模块

修改PayFeignApi接口

```

/**
 * Resilience4j Bulkhead 的例子
 * @param id
 * @return
 */
@GetMapping(value = "/pay/bulkhead/{id}")
public String myBulkhead(@PathVariable("id") Integer id);

```

## 3. 修改 **cloud-consumer-feign-order80**

POM文件

```

<!--resilience4j-bulkhead-->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-bulkhead</artifactId>
</dependency>

```

全局配置

```

# 开启circuitbreaker和分组激活
# spring.cloud.openfeign.circuitbreaker.enabled
spring.cloud.openfeign.circuitbreaker.enabled=true
spring.cloud.openfeign.circuitbreaker.group.enabled=true

####resilience4j bulkhead 的例子
resilience4j:
    bulkhead:
        configs:
            default:
                # 隔离允许并发线程执行的最大数量
                max-concurrent-calls: 2
                # 当达到并发调用数量时，新的线程的阻塞时间，我只愿意等待1秒，过时不候进舱壁兜底
    fallback
        max-wait-duration: 1s
    instances:
        cloud-payment-service:
            base-config: default
#timelimiter 默认限制远程1s，超于1s就超时异常，配置了降级，就走降级逻辑
timelimiter:
    configs:
        default:
            timeout-duration: 20s

```

## 4. 业务类

修改 **cloud-consumer-feign-order80 OrderCircuitController**

```

@GetMapping(value = "/feign/pay/bulkhead/{id}")
@Bulkhead(name = "cloud-payment-service", fallbackMethod =
"myBulkheadFallback", type = Bulkhead.Type.SEMAPHORE)
public String myBulkhead(@PathVariable("id") Integer id) {
    return payFeignApi.myBulkhead(id);
}
public String myBulkheadFallback(Throwable t) {
    return "myBulkheadFallback, 隔板超出最大数量限制, 系统繁忙, 请稍后再试----/(ㄒoㄒ)/~~";
}

```

## 测试步骤

浏览器新打开2个窗口，各点一次，分别点击<http://localhost/feign/pay/bulkhead/9999>

每个请求调用需要耗时5秒，2个线程瞬间达到配置过的最大并发数2

此时第3个请求正常的请求访问，<http://localhost/feign/pay/bulkhead/3>

直接被舱壁限制隔离了，碰不到8001

等其中一个窗口停止了，再去正常访问，并发数小于2 了，可以OK

## FixedThreadPoolBulkhead (固定线程池舱壁)

基本上就是我们JUC-线程池内容的同样思想 ThreadPoolExecutor

### 固定线程池舱壁 (FixedThreadPoolBulkhead)

1. FixedThreadPoolBulkhead的功能与SemaphoreBulkhead一样也是**用于限制并发执行的次数的**，但是二者的实现原理存在差别而且表现效果也存在细微的差别。FixedThreadPoolBulkhead使用一个固定线程池和一个等待队列来实现舱壁。
2. 当线程池中存在空闲时，则此时进入系统的请求将直接进入线程池开启新线程或使用空闲线程来处理请求。
3. 当线程池中无空闲时时，接下来的请求将进入等待队列，
4. 若等待队列仍然无剩余空间时接下来的请求将直接被拒绝，
5. 在队列中的请求等待线程池出现空闲时，将进入线程池进行业务处理。
6. 另外：ThreadPoolBulkhead只对CompletableFuture方法有效，所以我们必创建返回CompletableFuture类型的方法

io.github.resilience4j.bulkhead.internal.FixedThreadPoolBulkhead  
ThreadPoolExecutor  
submit进线程池返回CompletableFuture

1. 修改 [cloud-consumer-feign-order80](#)

### POM文件

```

<!--resilience4j-bulkhead-->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-bulkhead</artifactId>
</dependency>

```

### 全局配置

```
# 开启circuitbreaker和分组激活
# spring.cloud.openfeign.circuitbreaker.enabled
spring.cloud.openfeign.circuitbreaker.enabled=true
# spring.cloud.openfeign.circuitbreaker.group.enabled 请设置为false 新启线程和原来
主线程脱离
spring.cloud.openfeign.circuitbreaker.group.enabled=false
```

```
resilience4j:
  timelimiter:
    configs:
      default:
        #timelimiter默认限制远程1s，超过报错不好演示效果所以加上10秒
        timeout-duration: 10s
  thread-pool-bulkhead:
    configs:
      default:
        # 配置核心线程池大小
        core-thread-pool-size: 1
        # 配置最大线程池大小
        max-thread-pool-size: 1
        # 配置队列的容量
        queue-capacity: 1
    instances:
      cloud-payment-service:
        baseConfig: default
```

修改 OrderCircuitController文件

```
@GetMapping(value = "/feign/pay/bulkhead/{id}")
@Bulkhead(name = "cloud-payment-service", fallbackMethod =
"myBulkheadPoolFallback", type = Bulkhead.Type.THREADPOOL)
public CompletableFuture<String> myBulkheadTHREADPOOL(@PathVariable("id")
Integer id) {
    System.out.println(Thread.currentThread().getName() + "\t" + "-----开始
进入");
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + "\t" + "-----开始
离开");
    return CompletableFuture.supplyAsync(() -> payFeignApi.myBulkhead(id) +
"Bulkhead.Type.THREADPOOL");
}

public CompletableFuture<String> myBulkheadPoolFallback(Integer id, Throwable
t) {
    return CompletableFuture.supplyAsync(() -> "Bulkhead.Type.THREADPOOL, 系统繁
忙，请稍后再试----/(ㄒoㄒ)/~~");
}
```

2. 关于服务提供者端和公共模块的API配置和以上SemaphoreBulkhead一样

# 限流(RateLimiter)

限流 (频率控制)

官网: <https://resilience4j.readme.io/docs/ratelimiter>

中文手册: <https://github.com/lmhml/Resilience4j-Guides-Chinese/blob/main/core-modules/ratelimiter.md>

## 常见限流算法

### 1. 漏斗算法 (Leaky Bucket)

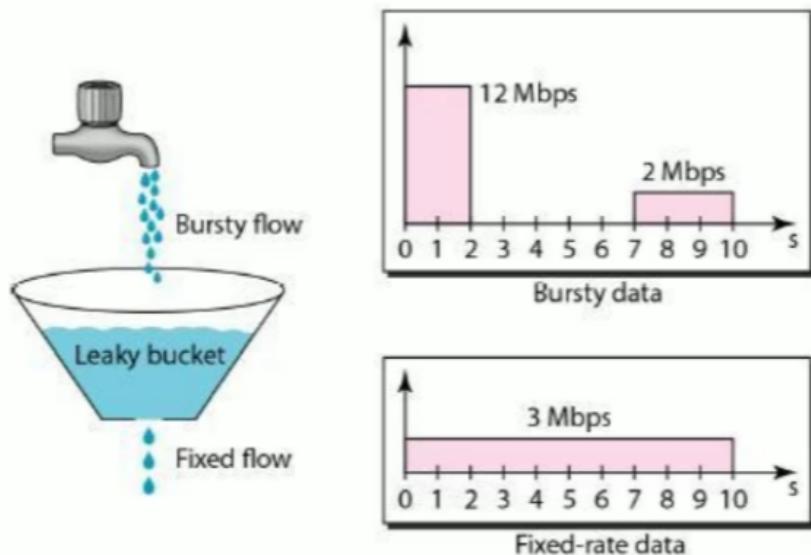
一个固定容量的漏桶，按照设定常量固定速率流出水滴，类似医院打吊针，不管你源头流量多大，我设定匀速流出。

如果流入水滴超出了桶的容量，则流入的水滴将会溢出了(被丢弃)，而漏桶容量是不变的。



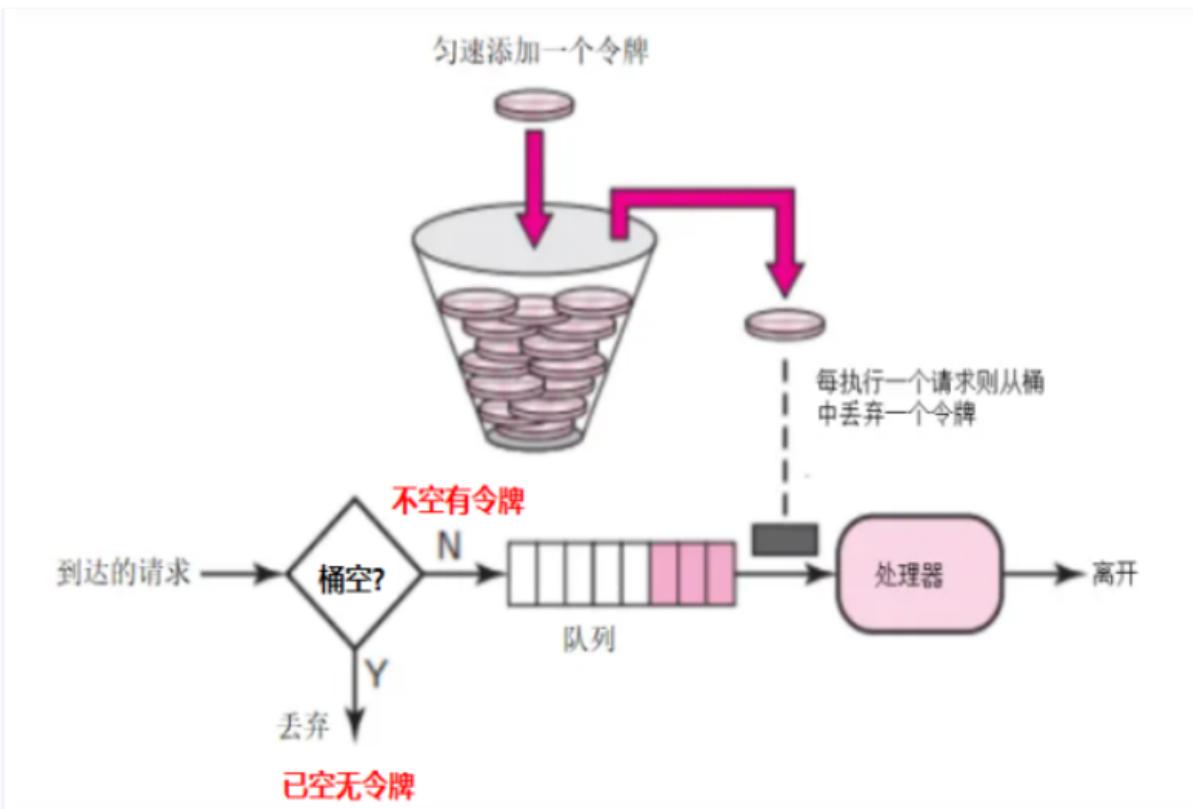
缺点：

这里有两个变量，一个是桶的大小，支持流量突发增多时可以存多少的水 (burst) ，另一个是水桶漏洞的大小 (rate) 。因为漏桶的漏出速率是固定的参数，所以，即使网络中不存在资源冲突 (没有发生拥塞)，漏桶算法也不能使流突发 (burst) 到端口速率。因此，漏桶算法对于存在突发特性的流量来说缺乏效率。



为了更好的控制流量，漏桶算法需要通过两个变量进行控制：一个是桶的大小，支持流量突发增多时可以存多少的水 (burst) ，另一个是水桶漏洞的大小 (rate) 。

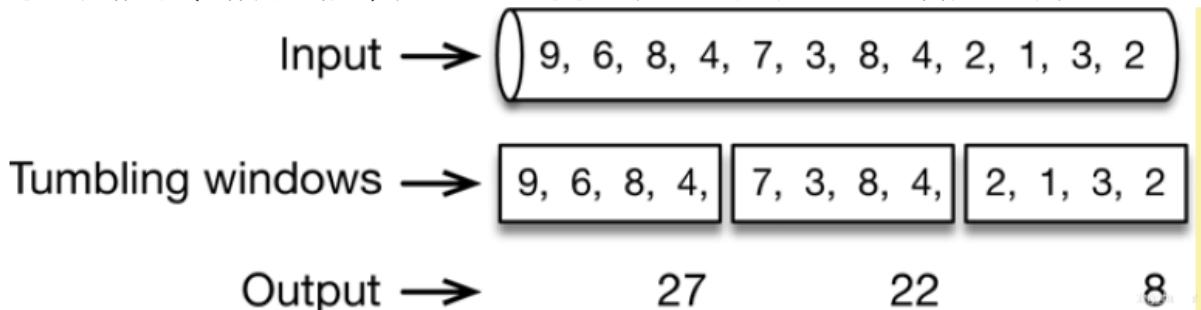
### 2. 令牌桶算法 (Token Bucket)



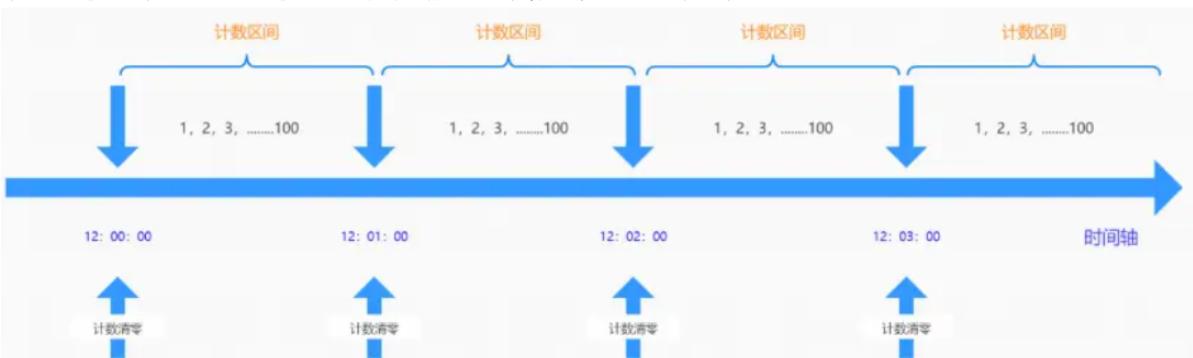
### 3. 滚动时间窗 (tumbling time window)

允许固定数量的请求进入(比如1秒取4个数据相加, 超过25值就over)超过数量就拒绝或者排队, 等下一个时间段进入。

由于是在一个时间间隔内进行限制, 如果用户在上个时间间隔结束前请求 (但没有超过限制), 同时在当前时间间隔刚开始请求 (同样没超过限制), 在各自的时间间隔内, 这些请求都是正常的。下图统计了3次, but.....



缺点: 间隔临界的一段时间内的请求就会超过系统限制, 可能导致系统被压垮



假如设定1分钟最多可以请求100次某个接口, 如12:00:00-12:00:59时间段内没有数据请求但12:00:59-12:01:00时间段内突然并发100次请求, 紧接着瞬间跨入下一个计数周期计数器清零; 在12:01:00-12:01:01内又有100次请求。那么也就是说在时间临界点左右可能同时有2倍的峰值进行请求, 从而造成后台处理请求**加倍过载**的bug, 导致系统运营能力不足, 甚至导致系统崩溃

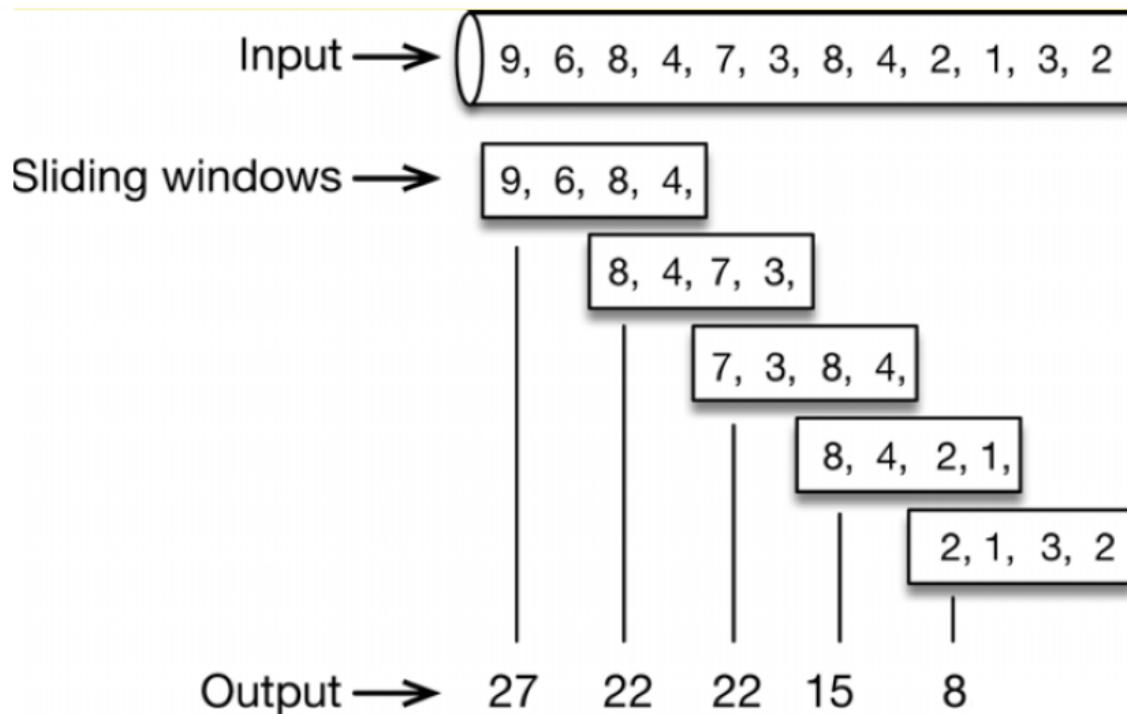
### 4. 滑动时间窗口 (sliding time window)

顾名思义, 该时间窗口是滑动的。所以, 从概念上讲, 这里有两个方面的概念需要理解:

窗口: 需要定义窗口的大小

滑动: 需要定义在窗口中滑动的大小, 但理论上讲滑动的大小不能超过窗口大小

滑动窗口算法是把固定时间片进行划分并且随着时间移动，移动方式为开始时间点变为时间列表中的第2个时间点，结束时间点增加一个时间点，不断重复，通过这种方式可以巧妙的避开计数器的临界点的问题。下图统计了5次



## 配置

1. 修改 `cloud-provider-payment8001` 服务提供模块

修改PayCircuitController新增myRateLimit方法

```
//=====Resilience4j ratelimit 的例子
@GetMapping(value = "/pay/ratelimit/{id}")
public String myRateLimit(@PathVariable("id") Integer id)
{
    return "Hello, myRateLimit欢迎到来 inputId: "+id+"\n" +
    idUtil.randomUUID();
}
```

2. 修改 `cloud-common-api` 公共模块

PayFeignApi接口新增api方法

```
@GetMapping(value = "/pay/ratelimit/{id}")
public String myRateLimit(@PathVariable("id") Integer id);
```

3. 修改 `cloud-consumer-feign-order80` 消费者端

POM

```
<!--resilience4j-ratelimiter-->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-ratelimiter</artifactId>
</dependency>
```

全局配置

```
# 开启circuitbreaker和分组激活
# spring.cloud.openfeign.circuitbreaker.enabled
spring.cloud.openfeign.circuitbreaker.enabled=true
# spring.cloud.openfeign.circuitbreaker.group.enabled=true
```

```
####resilience4j ratelimiter 限流的例子
resilience4j:
  ratelimiter:
    configs:
      default:
        limitForPeriod: 2 #在一次刷新周期内，允许执行的最大请求数
        limitRefreshPeriod: 1s # 限流器每隔limitRefreshPeriod刷新一次，将允许处理的最大请求数量重置为limitForPeriod
        timeout-duration: 1 # 线程等待权限的默认等待时间
    instances:
      cloud-payment-service:
        baseConfig: default
```

修改OrderCircuitController

```
@GetMapping(value = "/feign/pay/ratelimit/{id}")
@RateLimiter(name = "cloud-payment-service", fallbackMethod =
"myRateLimitFallback")
public String myBulkhead(@PathVariable("id") Integer id){
    return payFeignApi.myRateLimit(id);
}
public String myRateLimitFallback(Integer id, Throwable t){
    return "你被限流了，禁止访问/(ㄒoㄒ)/~~";
}
```

## Sleuth(Micrometer) + ZipKin分布式链路追踪

Sleuth目前进入维护模式

Sleuth替换方案：Micrometer Tracing

Sleuth官网：

<https://spring.io/projects/spring-cloud-sleuth>

<https://github.com/spring-cloud/spring-cloud-sleuth>

Spring Cloud Sleuth将不适用于Spring Boot 3.x以后。Sleuth将支持的Spring Boot的最后一个主要版本是2.x。

## 概述

- Micrometer

<https://micrometer.io/docs/>

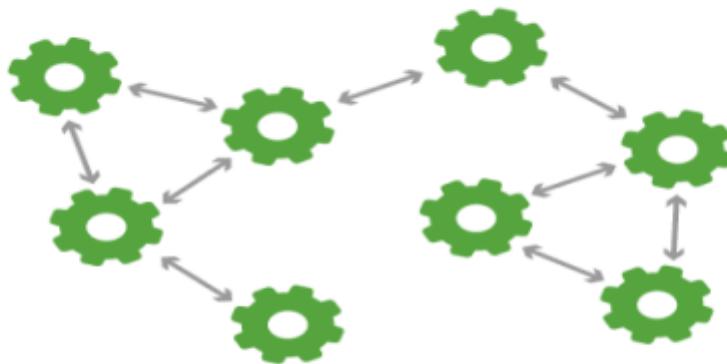
<https://github.com/micrometer-metrics/micrometer>

Spring Cloud Sleuth(micrometer)提供了一套完整的分布式链路追踪(Distributed Tracing)解决方案且兼容支持了zipkin展现，zipkin对micrometer收集的数据进行展现

- 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。

# MICROSERVICES ARCHITECTURE



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

随着问题的复杂化 + 微服务的增多 + 调用链条的变长，服务与服务之间的链路调用会非常复杂

- 在分布式与微服务场景下需要解决的问题，在大规模分布式与微服务集群下

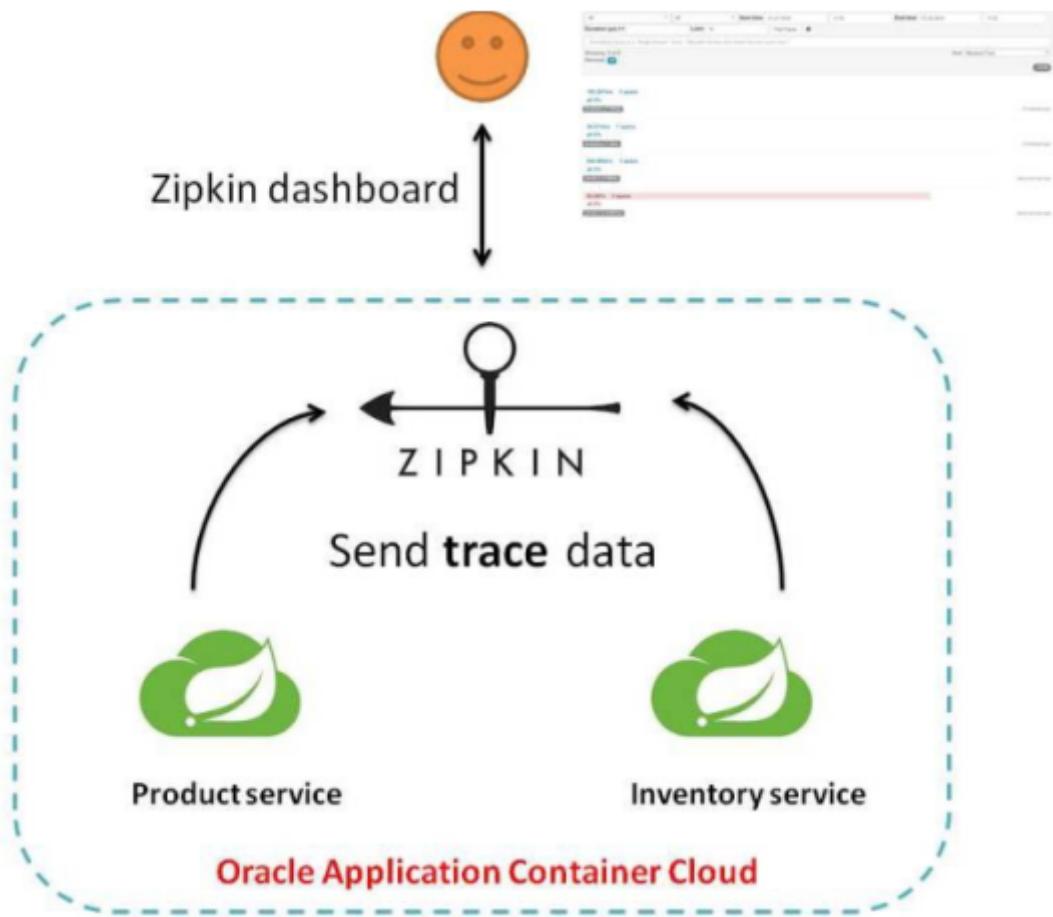
- 如何实时观测系统的整体调用链路情况。
- 如何快速发现并定位到问题。
- 如何尽可能精确的判断故障对系统的影响范围与影响程度。
- 如何尽可能精确的梳理出服务之间的依赖关系，并判断出服务之间的依赖关系是否合理。
- 如何尽可能精确的分析整个系统调用链路的性能与瓶颈点。
- 如何尽可能精确的分析系统的存储瓶颈与容量规划。

分布式链路追踪技术要解决的问题，分布式链路追踪（Distributed Tracing），就是将一次分布式请求还原成调用链路，进行日志记录，性能监控并将一次分布式请求的调用情况集中展示。比如各个服务节点上的耗时、请求具体到达哪台机器上、每个服务节点的请求状态等等。

- ZipKin

<https://zipkin.io/>

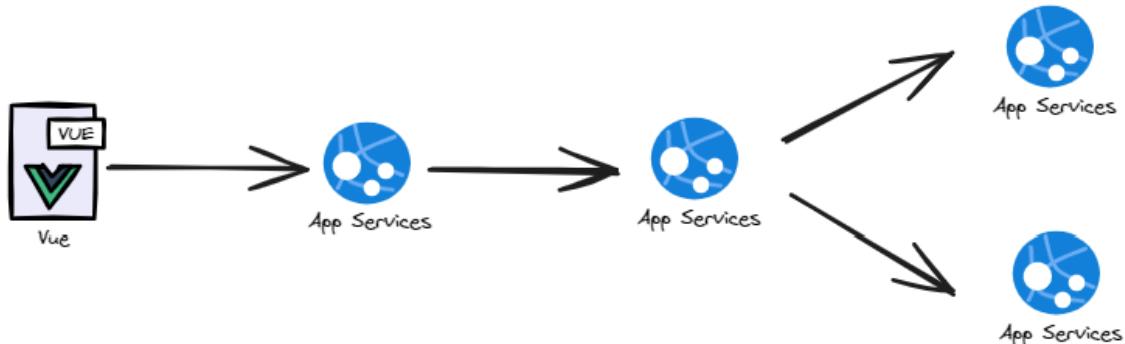
Zipkin是一种分布式链路跟踪系统图形化的工具，Zipkin是Twitter开源的分布式跟踪系统，能够收集微服务运行过程中的实时调用链路信息，并能够将这些调用链路信息展示到Web图形化界面上供开发人员分析，开发人员能够从ZipKin中分析出调用链路中的性能瓶颈，识别出存在问题的应用程序，进而定位问题和解决问题。



- 其他分布式链路追踪技术解决方案
  - Cat: 有大众点评开源，基于Java开发的实时应用监控平台，包括实时应用监控，业务监控。集成方案是通过代码埋点的方式来实现监控，比如：拦截器，过滤器等。对代码的侵入性很大，集成成本较高。风险较大
  - ZipKin: 由Twitter公司开源，开放源代码分布式的跟踪系统，用于收集服务的定时数据，以解决微服务架构中的延迟问题，包括：数据的收集、存储、查找和展现。结合spring-cloud-sleuth使用较为简单，集成方便，但是功能较为简单
  - Pinpoint: Pinpoint是一款开源的基于字节码注入的调用链分析，以及应用监控分析工具，特点是支持多种插件，UI功能强大，接入端无代码侵入
  - SkyWalking是国人开源的基于字节码注入的调用链分析，以及应用监控分析工具，特点是支持多种插件，UI功能较强，接入端五代码侵入

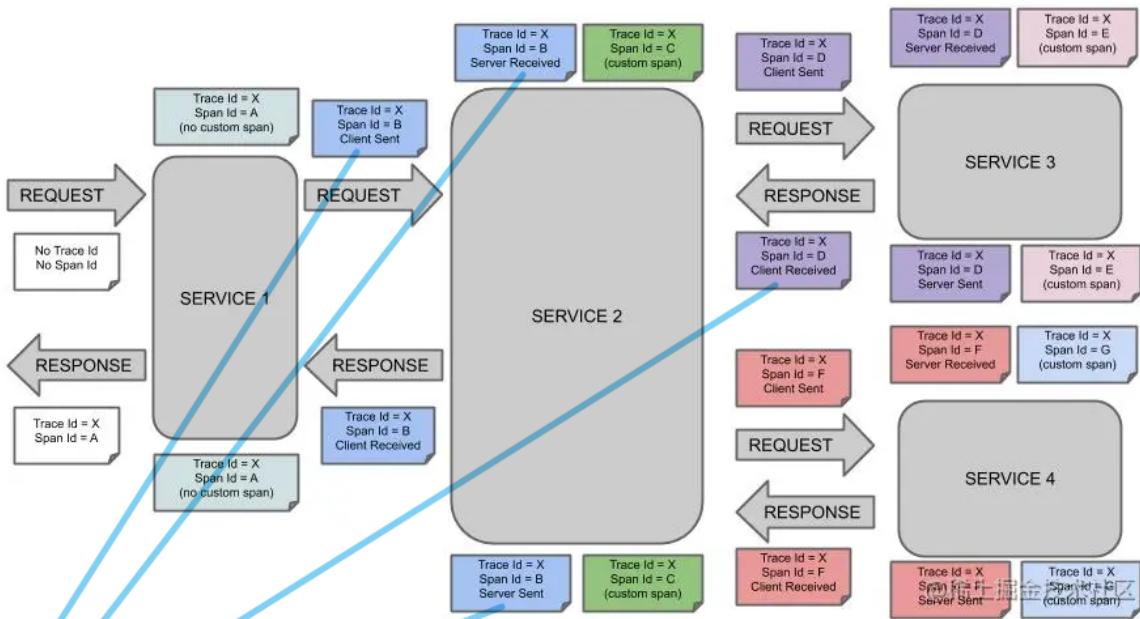
## 分布式链路追踪原理

假定3个微服务调用的链路：Service1调用Service2， Service2调用Service3和Service4



每一条链路追踪会在每个服务调用的时候加上Trace ID和Span ID

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（Span：表示调用链路来源，就是一次请求信息）



CS: Client Sent : 客户端发送这个请求的时间

SR: Server Received: 服务端接受这个请求的时间

CR: Client Received: 客户端接收到数据的时间

SS: Server Sent: 服务端发送响应的时间

A-B

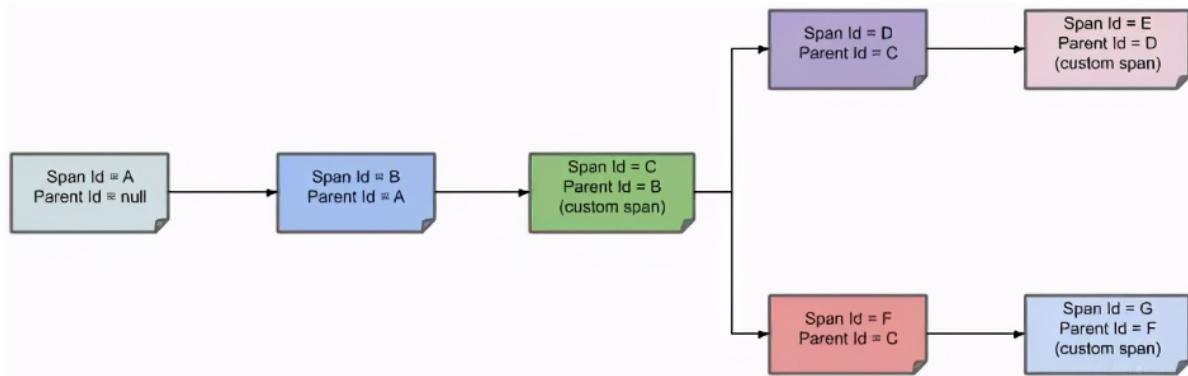
SR-CS: 网络传输时间 (请求过去)

SS-SR: 业务处理时间

CR-CS: 远程调用耗时

CS-SS: 网络传输时间 (响应过来)

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



1	<b>第一个节点: Span ID = A, Parent ID = null, Service 1 接收到请求。</b>
2	第二个节点: Span ID = B, Parent ID= A, Service 1 发送请求到 Service 2 返回响应给Service 1 的过程。
3	第三个节点: Span ID = C, Parent ID= B, Service 2 的中间解决过程。
4	第四个节点: Span ID = D, Parent ID= C, Service 2 发送请求到 Service 3 返回响应给Service 2 的过程。
5	第五个节点: Span ID = E, Parent ID= D, Service 3 的中间解决过程。
6	第六个节点: Span ID = F, Parent ID= C, Service 2 发送请求到 Service 4 返回响应给 Service 2 的过程。
7	第七个节点: Span ID = G, Parent ID= F, Service 4 的中间解决过程。
8	通过 Parent ID 就可找到父节点，整个链路即可以进行跟踪追溯了。

# 搭建监控案例步骤

Micrometer: 数据采样  
ZipKin: 图形展示

1. 下载ZipKin，并对其进行运行

```
java -jar zipkin-server-3.0.0-rc0-exec.jar
```

2. 修改父工程POM

```
<properties>
    <micrometer-tracing.version>1.2.0</micrometer-tracing.version>
    <micrometer-observation.version>1.12.0</micrometer-observation.version>
    <feign-micrometer.version>12.5</feign-micrometer.version>
    <zipkin-reporter-brave.version>2.17.0</zipkin-reporter-brave.version>
</properties>
```

由于Micrometer Tracing是一个门面工具自身并没有实现完整的链路追踪系统，具体的链路追踪另外需要引入的是第三方链路追踪系统的依赖：

```
<!--micrometer-tracing-bom导入链路追踪版本中心，体系化说明-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bom</artifactId>
    <version>${micrometer-tracing.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
<!--tracing指标追踪-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing</artifactId>
    <version>${micrometer-tracing.version}</version>
</dependency>
<!--micrometer-tracing-bridge-brave适配zipkin的桥接包-->
<!--一个Micrometer模块，用于与分布式跟踪工具 Brave 集成，以收集应用程序的分布式跟踪数据。Brave是一个开源的分布式跟踪工具，它可以帮助用户在分布式系统中跟踪请求的流转，它使用一种称为"跟踪上下文"的机制，将请求的跟踪信息存储在请求的头部，然后将请求传递给下一个服务。在整个请求链中，Brave会将每个服务处理请求的时间和其他信息存储到跟踪数据中，以便用户可以了解整个请求的路径和性能。-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
    <version>${micrometer-tracing.version}</version>
</dependency>
<!--
micrometer-observation
一个基于度量库 Micrometer的观测模块，用于收集应用程序的度量数据
-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
    <version>${micrometer-observation.version}</version>
</dependency>
<!--
feign-micrometer
一个Feign HTTP客户端的Micrometer模块，用于收集客户端请求的度量数据
-->
<dependency>
    <groupId>io.github.openfeign</groupId>
```

```

<artifactId>feign-micrometer</artifactId>
<version>${feign-micrometer.version}</version>
</dependency>
<!--
zipkin-reporter-brave
一个用于将 Brave 跟踪数据报告到zipkin 跟踪系统的库
-->
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
    <version>${zipkin-reporter-brave.version}</version>
</dependency>

```

补充包: `spring-boot-starter-actuator` SpringBoot框架的一个模块用于监视和管理应用程序

### 3. 修改服务提供者8001模块 `cloud-provider-payment8001`

POM文件

```

<!--micrometer-tracing指标追踪 1-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing</artifactId>
</dependency>
<!--micrometer-tracing-bridge-brave适配zipkin的桥接包 2-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<!--micrometer-observation 3-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
</dependency>
<!--feign-micrometer 4-->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-micrometer</artifactId>
</dependency>
<!--zipkin-reporter-brave 5-->
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>

```

全局文件配置

```

# =====zipkin=====
management:
  zipkin:
    tracing:
      endpoint: http://localhost:9411/api/v2/spans
  tracing:
    sampling:
      #采样率默认为0.1(0.1就是10次只能有一次被记录下来), 值越大收集越及时。
      probability: 1.0

```

新建业务类 PayMicrometerController

```

@RestController
public class PayMicrometerController
{
    // Micrometer(sleuth)进行链路监控的例子
    @GetMapping(value = "/pay/micrometer/{id}")
    public String myMicrometer(@PathVariable("id") Integer id)
    {
        return "Hello, 欢迎来到myMicrometer inputId: " + id + "\t 服务返回:" + IdUtil.simpleUUID();
    }
}

```

#### 4. 修改公共模块Api接口 [cloud-payment-api](#)

```

// Micrometer(sleuth)进行链路监控的例子
@GetMapping(value = "/pay/micrometer/{id}")
public String myMicrometer(@PathVariable("id") Integer id);

```

#### 5. 修改服务调用者80模块 [cloud-consumer-feign-order80](#)

POM

```

<!--micrometer-tracing指标追踪 1-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing</artifactId>
</dependency>
<!--micrometer-tracing-bridge-brave适配zipkin的桥接包 2-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<!--micrometer-observation 3-->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
</dependency>
<!--feign-micrometer 4-->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-micrometer</artifactId>
</dependency>
<!--zipkin-reporter-brave 5-->
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>

```

全局配置文件

```

# zipkin图形展现地址和采样率设置
management:
  zipkin:
    tracing:
      endpoint: http://localhost:9411/api/v2/spans
    tracing:
      sampling:
        probability: 1.0 #采样率默认为0.1(0.1就是10次只能有一次被记录下来), 值越大收集越及时。

```

新建业务类 OrderMicrometerController

```

@RestController
@Slf4j
public class OrderMicrometerController
{
    @Resource
    private PayFeignApi payFeignApi;

    @GetMapping(value = "/feign/micrometer/{id}")
    public String myMicrometer(@PathVariable("id") Integer id)
    {
        return payFeignApi.myMicrometer(id);
    }
}

```

## 6. 进行测试

访问请求地址 `localhost:80/feign/micrometer/{id}`

进入 `localhost:9411` 地址查看链路追踪效果

# Gateway(网关)

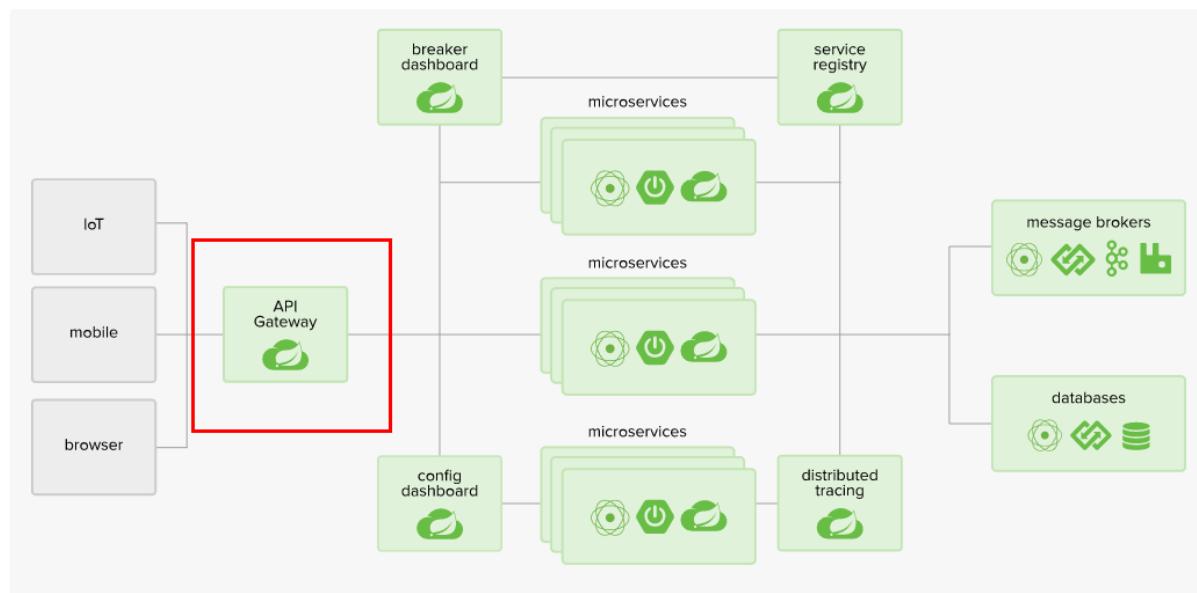
## 概述

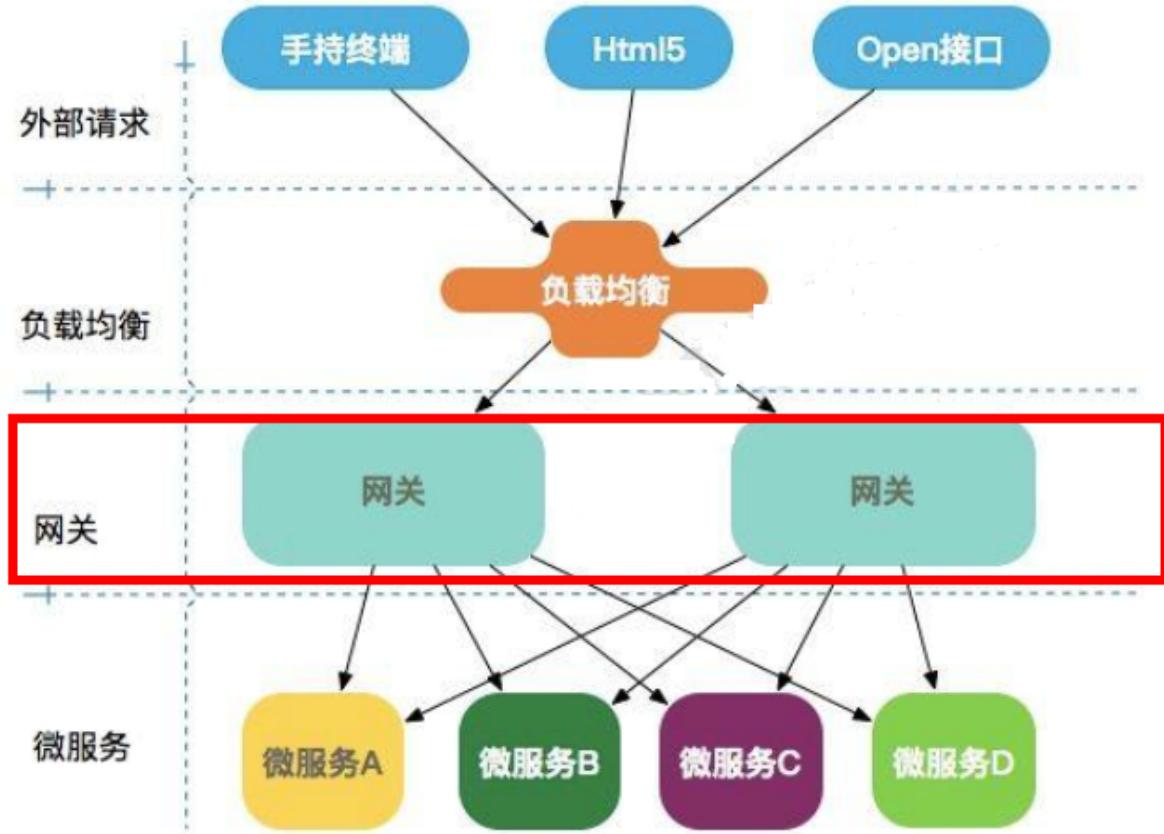
Cloud全家桶中有个很重要的组件就是网关，在1.x版本中都是采用的Zuul网关；但在2.x版本中，zuul的升级一直跳票，SpringCloud最后自己研发了一个网关SpringCloud Gateway替代Zuul，那就是SpringCloud Gateway一句话：gateway是原zuul1.x版的替代

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#glossary>

Spring Cloud Gateway组件的核心是一系列的过滤器，通过这些过滤器可以将客户端发送的请求转发(路由)到对应的微服务。Spring Cloud Gateway是加在整个微服务最前沿的防火墙和代理器，隐藏微服务结点IP端口信息，从而加强安全保护。Spring Cloud Gateway本身也是一个微服务，需要注册进服务注册中心。

功能：反向代理，鉴权，流量控制，熔断，日志监控

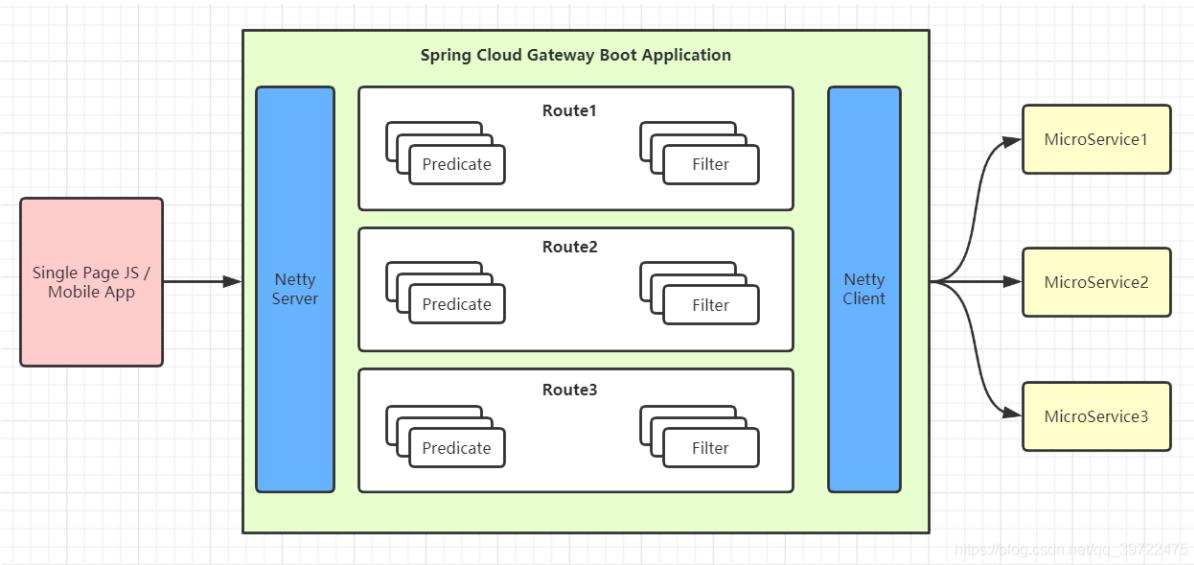




## Gateway三大核心

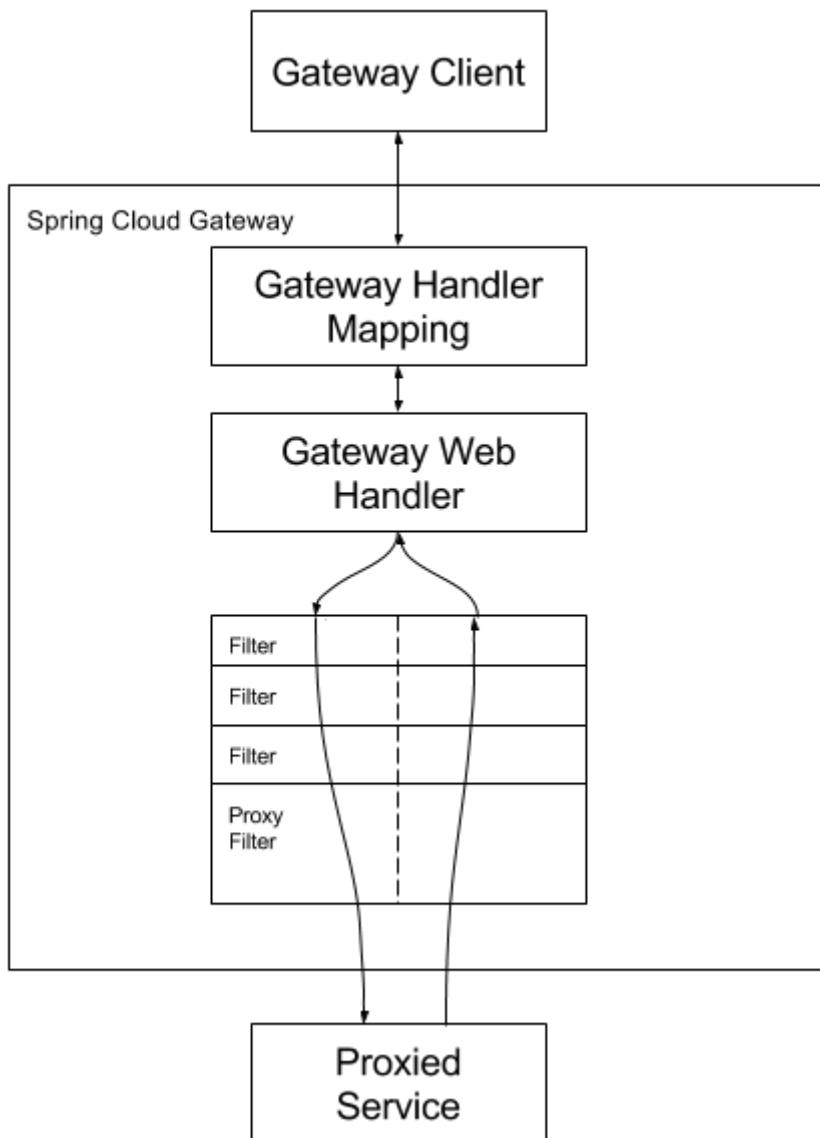
### 2. Glossary

- **Route**: The basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates, and a collection of filters. A route is matched if the aggregate predicate is true.
- **Predicate**: This is a [Java 8 Function Predicate](#). The input type is a [Spring Framework ServerWebExchange](#). This lets you match on anything from the HTTP request, such as headers or parameters.
- **Filter**: These are instances of [GatewayFilter](#) that have been constructed with a specific factory. Here, you can modify requests and responses before or after sending the downstream request.
- **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](#)，匹配请求中所有内容，若请求内容与断言相匹配则进行路由
- **Filter (过滤)**：Spring框架中GatewayFilter的实例，使用过滤器，可以在请求被路由前后对请求进行修改



## Gateway工作流程

路由转发 + 断言判断 + 执行过滤器



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。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。
2. 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。
3. 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；
4. 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。

## 入门配置

### 配置网关9527模块

1. 新建网关模块Module **cloud-gateway9527**
2. POM依赖文件

```
<dependencies>
    <!--gateway-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <!--服务注册发现consul discovery，网关也要注册进服务注册中心统一管控-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-consul-discovery</artifactId>
    </dependency>
    <!-- 指标监控健康检查的actuator，网关是响应式编程删除掉spring-boot-starter-web
dependency-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>

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

3. 全局配置文件

```
server:
  port: 9527
spring:
  application:
    # 以微服务注册进consul或nacos服务列表内
    name: cloud-gateway
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        prefer-agent-address: true
        service-name: ${spring.application.name}
```

```

gateway:
  routes:
    # 路由的ID，没有固定规则，要求唯一，配合服务名
    - id: pay_route1
      # 匹配后提供服务的路由地址
      uri: http://localhost:8001
      # 断言：路径相匹配的进行路由
      predicates:
        - Path=/pay/gateway/get/**
    - id: pay_route2
      uri: http://localhost:8001
      predicates:
        - Path=/pay/gateway/info/**

```

4. 启动consul，启动9257模块，查看是否注册进consul

## 配置8001服务提供模块

1. 新建PayGateWayController

```

@RestController
public class PayGatewayController
{
    @Resource
    PayService payService;

    @GetMapping(value = "/pay/gateway/get/{id}")
    public ResultData<Pay> getById(@PathVariable("id") Integer id)
    {
        Pay pay = payService.getById(id);
        return ResultData.success(pay);
    }

    @GetMapping(value = "/pay/gateway/info")
    public ResultData<String> getGatewayInfo()
    {
        return ResultData.success("gateway info test: "+ IdUtil.simpleUUID());
    }
}

```

## 配置公共&消费者模块

1. 修改 `cloud-api-commons` 公共模块

同时并修改 `@FeignClient` 注解，修改为网关模块的服务名称 `cloud-gateway`

```

@FeignClient(value = "cloud-gateway")
public interface PayFeignApi {
    //Gateway进行网关测试案例01
    @GetMapping(value = "/pay/gateway/get/{id}")
    public ResultData<Pay> getById(@PathVariable("id") Integer id);
    // Gateway进行网关测试案例02
    @GetMapping(value = "/pay/gateway/info")
    public ResultData<String> getGatewayInfo();
}

```

2. 修改 `cloud-consumer-feign-order80` 消费者模块

新建 `OrderGatewayController`，调用接口进行访问

```

@RestController
public class OrderGatewayController
{
    @Resource
    private PayFeignApi payFeignApi;

    @GetMapping(value = "/feign/pay/gateway/get/{id}")
    public ResultData getById(@PathVariable("id") Integer id)
    {
        return payFeignApi.getById(id);
    }

    @GetMapping(value = "/feign/pay/gateway/info")
    public ResultData<String> getGatewayInfo()
    {
        return payFeignApi.getGatewayInfo();
    }
}

```

## 总结

1. 进行测试，请求以下地址，查看是否成功
2. 断开网关模块，在请求以下地址查看是否生效  
<http://localhost/feign/pay/gateway/info>  
<http://localhost/feign/pay/gateway/get/3>
3. 在系统内的模块互相访问之间，不需要进行网关的链接
4. 在系统外的请求需要访问系统内的内容，则需要进行访问网关，网关进行访问系统内容

## GateWay高级特性

### Route

1. 动态获取服务URI

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#reactive-loadbalancer-client-filter>  
将网关的全局配置文件中的URI更改成服务名称

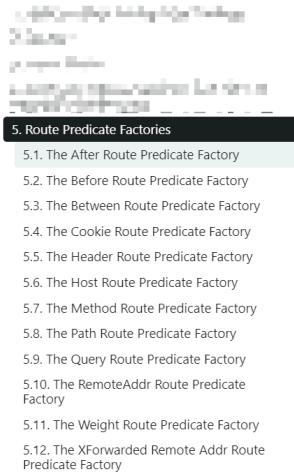
```

spring:
  cloud:
    gateway:
      routes:
        # 路由的ID，没有固定规则，要求唯一，配合服务名
        - id: pay_route1
          # 匹配后提供服务的路由地址
          uri: lb://cloud-payment-service
          # 断言：路径相匹配的进行路由
          predicates:
            - Path=/pay/gateway/get/**

```

### Predicate(谓词)

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#gateway-request-predicates-factories>  
是什么？



## 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:

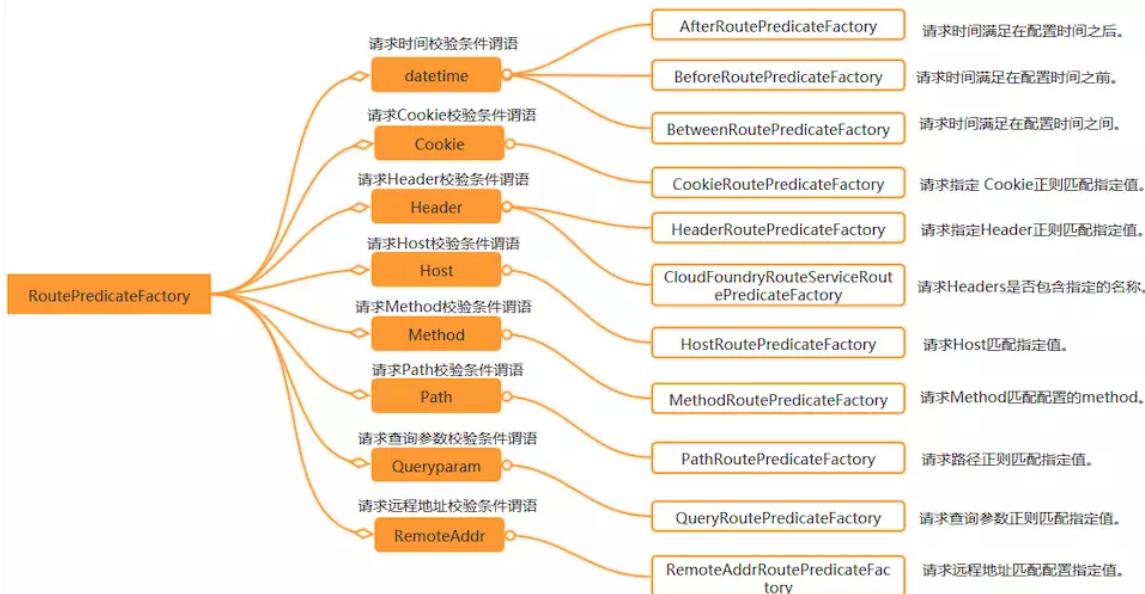
*Example 1. application.yml*

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

整体架构描述：

`ShortcutConfigurable -> RoutePredicateFactory`

`Configurable -> RoutePredicateFactory`



## 配置语法

配置语法总体概述：两种配置

### 1. Shortcut Configuration

### 2. Fully Expanded Arguments

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#configuring-route-predicate-factories-and-gateway-filter-factories>

#### 1. Shortcut Configuration

快捷方式配置由过滤器名称识别，后跟等号（`=`），后跟用逗号（`,`）分隔的参数值。

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            # Shortuct Configuration
            - Cookie=mycookie,mycookievalue
```

## 2. Fully Expanded Arguments

完全展开的参数看起来更像带有名称/值对的标准yaml配置。通常，会有一个 `name` 键和一个 `args`，`args` 是用于配置谓词或过滤器的键值对映射。

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: https://example.org
          predicates:
            # Fully Expanded Arguments
            - name: Cookie
              args:
                name: mycookie
                regexp: mycookievalue
```

## 内置Route Predicate

测试地址: [localhost:9527/pay/gateway/info](http://localhost:9527/pay/gateway/info)

### 1. After Route Predicate

匹配在指定日期时间之后发生的请求

```
predicates:
  # 时间通过这个函数获取: ZonedDateTime.now(); // 默认时区
  - After=2024-05-10T19:52:30.336041400+08:00[Asia/Shanghai]
```

### 2. Before Route Predicate

匹配在指定日期时间之前发生的请求

```
predicates:
  # 时间通过这个函数获取: ZonedDateTime.now(); // 默认时区
  - Before=2024-05-10T19:52:30.336041400+08:00[Asia/Shanghai]
```

### 3. Between Route Predicate

匹配发生在 `datetime1` 之后和 `datetime2` 之前的请求

```
predicates:
  # 时间通过这个函数获取: ZonedDateTime.now(); // 默认时区
  - Before=2024-05-10T19:52:30.336041400+08:00[Asia/Shanghai],2024-05-10T19:55:30.336041400+08:00[Asia/Shanghai]
```

### 4. Cookie Route Predicate

匹配具有给定名称且其值与正则表达式匹配的请求

```
predicates:
# 请求需要携带Cookie, 键值对为 username:zhangsan
- Cookie=username, zhangsan
```

#### 5. Header Route Predicate

匹配具有给定名称的标头的请求

```
predicates:
- Header=X-Request-ID, \d+
```

#### 6. Host Route Predicate

匹配具有给定 Host 标头的请求

```
predicates:
- Host=**.xxx.com
```

#### 7. Path Route Predicate

匹配给定路径的请求

```
predicates:
- Path=/xxxx/get/**
```

#### 8. Query Route Predicate

匹配给定请求参数的请求

```
predicates:
# 要有参数名username并且值还要是整数才能路由
- Query=username, \d+
```

#### 9. RemoteAddr Route Predicate

匹配请求远程地址是Addr的

```
predicates:
- RemoteAddr=192.168.202.1/32 # 外部访问我的IP限制, 最大跨度不超过32, 目前是1~24它们是
CIDR 表示法。
```

#### 10. Method Route Predicate

匹配的HTTP请求方法的请求

```
predicates:
- Method=GET, POST
```

## 自定义断言

1. 继承AbstractRoutePredicateFactory抽象类
2. 实现RoutePredicateFactory接口
3. 自定义断言: 文件名称为以 **RoutePredicateFactory** 后缀为结尾

参考自带的RoutePredicateFactory文件

```
@Component
public class MyRoutePredicateFactory extends
AbstractRoutePredicateFactory<MyRoutePredicateFactory.Config> {
    public MyRoutePredicateFactory() {
        super(MyRoutePredicateFactory.Config.class);
```

```

    }

    public List<String> shortcutFieldOrder() {
        return collections.singletonList("userType");
    }

    // 重写apply方法
    @Override
    public Predicate<ServerWebExchange> apply(MyRoutePredicateFactory.Config config) {
        return new Predicate<ServerWebExchange>() {
            @Override
            public boolean test(ServerWebExchange serverWebExchange) {
                // 检查request的参数里面，userType是否为指定的值，符合配置就通过
                String userType =
serverWebExchange.getRequest().getQueryParams().getFirst("userType");
                if (userType == null) return false;
                // 如果说参数存在，就和config的数据进行比较
                if (userType.equals(config.getUserType())) {
                    return true;
                }
                return false;
            }
        };
    }

    @Validated
    public static class Config {
        @NotEmpty
        private String userType; // 钻、金、银等用户等级

        public void setUserType(String userType) {
            this.userType = userType;
        }

        public String getUserType() {
            return userType;
        }
    }
}

```

测试

```

predicates:
- My=diamond

```

## Filter(过滤)

### 概述

与SpringMVC里面的拦截器Interceptor, Servlet的过滤器  
pre和post分别在请求被执行前调用和被执行后调用

#### 功能需求:

1. 请求鉴权
2. 异常处理
3. 记录接口调用时长

4. ....

#### Filter类型:

##### 1. 全局默认过滤器 Global Filters

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#global-filters>

gateway出厂默认，作用域全局所有路由，实现GlobalFilter接口即可

##### 2. 单一内置过滤器 GatewayFilter

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#gatewayfilter-factories>

网关过滤器，主要作用于单一路由或者某个路由分组

##### 3. 自定义过滤器

## Gateway内置过滤器

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#gatewayfilter-factories>

常用的内置过滤器

### RequestHeader相关

前置操作

修改服务提供的 8001 PayGatewayController

```
@GetMapping(value = "/pay/gateway/filter")
public ResultData<String> getGatewayFilter(HttpServletRequest request)
{
    String result = "";
    Enumeration<String> headers = request.getHeaderNames();
    while(headers.hasMoreElements())
    {
        String headName = headers.nextElement();
        String headValue = request.getHeader(headName);
        System.out.println("请求头名: " + headName +"\t\t\t" +"请求头值: " +
headValue);
        if(headName.equalsIgnoreCase("X-Request-val1")
           || headName.equalsIgnoreCase("X-Request-val2")) {
            result = result+headName + "\t " + headValue + " ";
        }
    }
    return ResultData.success("getGatewayFilter 过滤器 test: "+result+" \t "+
DateUtil.now());
}
```

1. AddRequestHeader GatewayFilter Factory: 给所有请求添加请求头

2. RemoveRequestHeader GatewayFilter Factory: 给所有请求删除匹配 X-Request-val 的标头

3. SetRequestHeader GatewayFilter Factory: 给所有请求替换所有标头

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_header_route
          uri: lb://cloud-payment-service
          predicates:
            - Path=/pay/gateway/filter/**
          filters:
            - AddRequestHeader=X-Request-val, val1
            - RemoveRequestHeader=X-Request-val
            - SetRequestHeader=X-Request-val, val-new
```

## RequestParam相关

前置操作

修改服务提供的 8001 PayGatewayController

```
@GetMapping(value = "/pay/gateway/filter")
public ResultData<String> getGatewayFilter(HttpServletRequest request) {
    System.out.println("=====");
    String customerId = request.getParameter("customerId");
    System.out.println("request Parameter customerId: "+customerId);

    String customerName = request.getParameter("customerName");
    System.out.println("request Parameter customerName: "+customerName);
    System.out.println("=====");
}
```

1. **AddRequestParameter GatewayFilter Factory**: 给所有请求添加请求参数 color
2. **RemoveRequestParameter GatewayFilter Factory**: 删除所有请求的请求参数 color

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: lb://cloud-payment-service
          predicates:
            - Path=/pay/gateway/filter/**
          filters:
            - AddRequestParameter=color, blue
            - RemoveRequestParameter=color, blu
```

## ResponseHeader相关

通过chrome开发者工具查看响应头

1. **AddResponseHeader GatewayFilter Factory**: 给所有请求添加响应头
2. **SetResponseHeader GatewayFilter Factory**: 给所有请求修改响应头
3. **RemoveResponseHeader GatewayFilter Factory**: 给所有请求删除响应头

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: lb://cloud-payment-service
          predicates:
            - Path=/pay/gateway/filter/**
          filters:
            - AddResponseHeader=X-Response-val, val
            - SetResponseHeader=Date,2099-1-1
            - RemoveResponseHeader=Transfer-Encoding
```

## 前缀和路径相关

1. **PrefixPath GatewayFilter Factory**: 给请求自动添加路径前缀
2. **SetPath GatewayFilter Factory**: 访问路径修改
3. **RedirectTo GatewayFilter Factory**: 重定向某个页面

```
spring:
  cloud:
    gateway:
      routes:
        - id: add_request_parameter_route
          uri: lb://cloud-payment-service
          predicates:
            - Path=/xxxx/{segment}
          filters:
            #自动添加前缀/pay
            - PrefixPath=/pay
            # 将/pay/yyyy修改成/pay/gateway, segment为动态参数
            - SetPath=SetPath=pay/gateway/{segment}
            # 重定向到http://www.baidu.com/
            - RedirectTo=302, http://www.baidu.com/
```

## 其他

**Default Filters**: 相当于全局通用, 自定义配置Global

```
spring:
  cloud:
    gateway:
      default-filters:
        - AddResponseHeader=X-Response-Default-Red, Default-Blue
        - PrefixPath=/httpbin
```

## 自定义过滤器

### 自定义全局Filter

<https://docs.spring.io/spring-cloud-gateway/docs/4.0.9/reference/html/#global-filters>

- 修改 `cloud-gateway9527` 模块新建 `MyGlobalFilter` 类并实现 `GlobalFilter`, `Orderd` 两个接口
- 定义请求各个方法的所需时间

```
@Component
@Slf4j
public class MyGlobalFilter implements GlobalFilter, Ordered
{
    // 数字越小优先级越高
    @Override
    public int getOrder()
    {
        return 0;
    }

    private static final String BEGIN_VISIT_TIME = "begin_visit_time";//开始访问
    //时间

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //先记录下访问接口的开始时间
        exchange.getAttributes().put(BEGIN_VISIT_TIME,
System.currentTimeMillis());

        return chain.filter(exchange).then(Mono.fromRunnable(()->{
            Long beginvisitTime = exchange.getAttribute(BEGIN_VISIT_TIME);
```

```

        if (beginVisitTime != null){
            log.info("访问接口主机: " +
exchange.getRequest().getURI().getHost());
            log.info("访问接口端口: " +
exchange.getRequest().getURI().getPort());
            log.info("访问接口URL: " +
exchange.getRequest().getURI().getPath());
            log.info("访问接口URL参数: " +
exchange.getRequest().getURI().getRawQuery());
            log.info("访问接口时长: " + (System.currentTimeMillis() -
beginVisitTime) + "ms");
            log.info("我是美丽分割线:
#####
");
            System.out.println();
        }
    });
}
}

```

## 自定义条件Filter

参考GateWay内置默认的Filter

- 修改 `cloud-gateway9527` 模块，新建 `MyGatewayFilterFactory` 类
- 条件：查看请求是否有请求参数 `myparams`

```

@Component
public class MyGatewayFilterFactory extends
AbstractGatewayFilterFactory<MyGatewayFilterFactory.Config> {
    public MyGatewayFilterFactory() {
        super(MyGatewayFilterFactory.Config.class);
    }

    @Override
    public GatewayFilter apply(MyGatewayFilterFactory.Config config) {
        return new GatewayFilter() {
            @Override
            public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
                ServerHttpRequest request = exchange.getRequest();
                System.out.println("进入了自定义网关过滤器MyGatewayFilterFactory,
status: " + config.status);
                if (request.getQueryParams().containsKey("myparams")) {
                    return chain.filter(exchange);
                } else {

exchange.getResponse().setStatus(HttpStatus.BAD_GATEWAY);
                    return exchange.getResponse().setComplete();
                }
            }
        };
    }

    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("status");
    }

    public static class Config {
        //设定一个状态值/标志位，它等于多少，匹配和才可以访问
    }
}

```

```
    private String status;  
}  
}
```

yaml配置

```
filters:  
- My=mysparams
```

## Spring Cloud Alibaba

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。

官网定义: <https://github.com/alibaba/spring-cloud-alibaba/wiki>

<https://github.com/alibaba/spring-cloud-alibaba/blob/2022.x/README-zh.md>

英文文档: <https://spring-cloud-alibaba-group.github.io/github-pages/2022/en-us/2022.0.0.0-RC2.html>

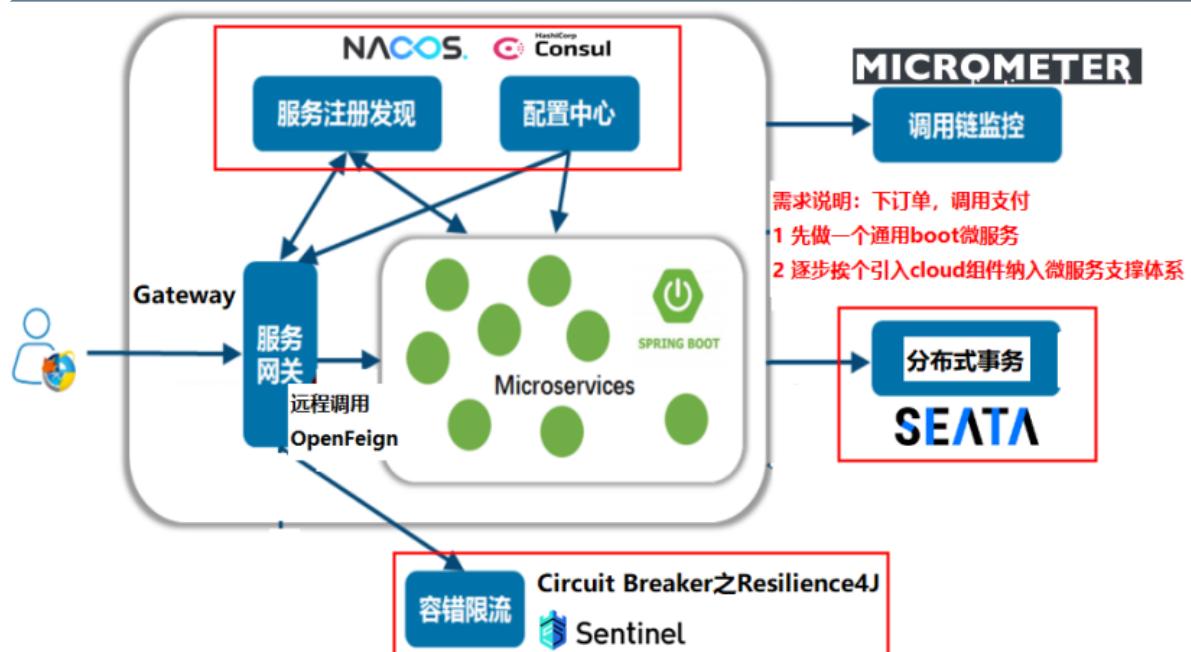
中文文档: <https://spring-cloud-alibaba-group.github.io/github-pages/2022/zh-cn/2022.0.0.0-RC2.html>

## Nacos(服务注册和配置中心)

### 介绍

<https://nacos.io/docs/v2/quickstart/quick-start/>

Nacos: Dynamic Naming and Configuration Service



<https://nacos.io/>

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 = Eureka + Config Bus

Nacos = Spring Cloud Consul

CAP介绍

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

## Discovery 服务注册中心

概述: <https://nacos.io/docs/v2/ecology/use-nacos-with-spring-cloud/>

- 通过 Nacos Server 和 spring-cloud-starter-alibaba-nacos-discovery 实现服务的注册与发现。

参考文档: <https://spring-cloud-alibaba-group.github.io/github-pages/2022/zh-cn/2022.0.0.0-RC2.html>

- nacos启动: 在bin目录下启动, `startup.cmd -m standalone`

## 配置服务提供者(9001)

负载均衡: 参考9001, 拷贝虚拟端口映射, 在idea配置中进行拷贝, 端口9002, 在9001和9002之间进行轮询

- 新建Module `cloudalibaba-provider-payment9001`
- POM文件

```
<dependencies>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!-- 引入自己定义的api通用包 -->
    <dependency>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--SpringBoot通用依赖模块-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <!--hutool-->
    <dependency>
        <groupId>cn.hutool</groupId>
        <artifactId>hutool-all</artifactId>
    </dependency>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.28</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

```

</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```

### 3. 全局配置文件

```

server:
  port: 9001
spring:
  application:
    name: nacos-payment-provider
cloud:
  nacos:
    discovery:
      # 配置nacos地址
      server-addr: localhost:8848

```

### 4. 主启动类加上注解 `@EnableDiscoveryClient`

### 5. 业务类, 新增 `PayAlibabaController` 类

```

@RestController
public class PayAlibabaController {
    @Value("${server.port}")
    private String serverPort;

    @GetMapping(value = "/pay/nacos/{id}")
    public String getPayInfo(@PathVariable("id") Integer id) {
        return "nacos registry, serverPort: " + serverPort + ", id: " + id;
    }
}

```

测试地址:

<http://localhost:9001/pay/nacos/12>

<http://localhost:8848/nacos/>

## 配置服务消费者(83)

Consumer 应用可能还没像启动一个 Provider 应用那么简单。因为在 Consumer 端需要去调用 Provider 端提供的REST 服务。例子中我们使用最原始的一种方式, 即显示的使用 LoadBalanceClient 和 RestTemplate 结合的方式来访问。

### 1. 新建Module `cloudalibaba-consumer-nacos-order83`

### 2. POM文件

```

<dependencies>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--loadbalancer-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
    <!--web + actuator-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>

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

```

### 3. 全局配置

```

server:
  port: 83
spring:
  application:
    name: nacos-order-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  service-url:
    nacos-user-service: http://nacos-payment-provider

```

### 4. 主启动类加上 `@EnabledDiscoveryClient` 注解

### 5. 业务类配置

新建 `RestTemplateConfig` 类

```

@Configuration
public class RestTemplateConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

新建 OrderNacosController 类

```

@RestController
public class OrderNacosController {
    @Resource
    private RestTemplate restTemplate;

    @Value("${service-url.nacos-user-service}")
    private String serverUrl;

    @GetMapping("/consumer/pay/nacos/{id}")
    public String paymentInfo(@PathVariable("id") String id) {
        String result = restTemplate.getForObject(serverUrl + "/pay/nacos/" + id, String.class);
        return result + "\t" + "我是OrderNacosController83调用者。。。。。";
    }
}

```

## Config 服务配置中心(3377)

之前案例Consul8500服务配置动态变更功能可以被Nacos取代  
通过 Nacos Server 和 spring-cloud-starter-alibaba-nacos-config 实现中心话全局配置的动态变更

- <https://nacos.io/docs/v2/ecology/use-nacos-with-spring-cloud/>
- 参考中文文档: [https://spring-cloud-alibaba-group.github.io/github-pages/2022/zh-cn/2022.0.0.0-RC2.html#\\_spring\\_cloud\\_alibaba\\_nacos\\_config](https://spring-cloud-alibaba-group.github.io/github-pages/2022/zh-cn/2022.0.0.0-RC2.html#_spring_cloud_alibaba_nacos_config)

1. 新建配置中心模块 `cloudalibaba-config-nacos-client3377`

2. POM文件

```

<dependencies>
    <!--bootstrap-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bootstrap</artifactId>
    </dependency>
    <!--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>
        <groupId>org.springframework.boot</groupId>

```

```

<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
</dependencies>

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

```

### 3. 全局配置文件

1. Nacos同Consul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，为了满足动态刷新和全局广播通知
2. springboot中配置文件的加载是存在优先级顺序的，bootstrap优先级高于application

bootstrap.yml

```

# nacos配置
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        # nacos服务注册中心地址
        server-addr: localhost:8848
      config:
        file-extension: yaml
        # nacos配置中心地址
        server-addr: localhost:8848

```

application.yml

```

server:
  port: 3377
spring:
  profiles:
    active: dev #开发环境
#    active: prod # 生产环境
#    active: test # 测试环境

```

4. 主启动类加 `@EnableDiscoveryClient` 注解，进行服务注册发现
5. 业务类，新建 `NacosConfigClientController` 类，添加 `@RefreshScope` 进行动态刷新

```

@RestController
//在控制器类加入@RefreshScope注解使当前类下的配置支持Nacos的动态刷新功能。
@RefreshScope

```

```

public class NacosConfigClientController
{
    // 读取Nacos中的配置信息
    @Value("${config.info}")
    private String configInfo;

    @GetMapping("/config/info")
    public String getConfigInfo() {
        return configInfo;
    }
}

```

## 6. 在Nacos中添加配置信息

### 2. 在 `bootstrap.properties` 中配置 Nacos server 的地址和应用名

```

spring.cloud.nacos.config.server-addr=127.0.0.1:8848

spring.application.name=example

```

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

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

```

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

```

- `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。

```

# nacos注册中心
server:
  port: 3377

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yaml格式的配置

```

```

# Nacos注册配置, application.yml
spring:
  profiles:
    active: dev

```

本案例的DataID是：`nacos-config-client-dev.yaml`

进行创建配置：

NACOS 2.2.3

当前集群没有开启鉴权, 请参考[文档](#)开启鉴权~

## 配置管理

public

**创建配置** Data ID 已开启默认模糊查询 Group 已开启默认模糊查询 模拟模糊匹配  查询 高级查询 导入配置 +

查询到 0 条满足要求的配置。

### 新建配置

\* 命名空间

\* Data ID  \* Group

更多高级选项

描述

配置格式  TEXT  JSON  XML  YAML  HTML  Properties

\* 配置内容  config:  
info: nacos-config-client-dev.yaml, come from nacosconfig, version=1

测试: 调用 <localhost:3377/config/info> 查看是否成功  
自带动态刷新, 同时nacos支持回滚, 可以回滚到之前某个版本

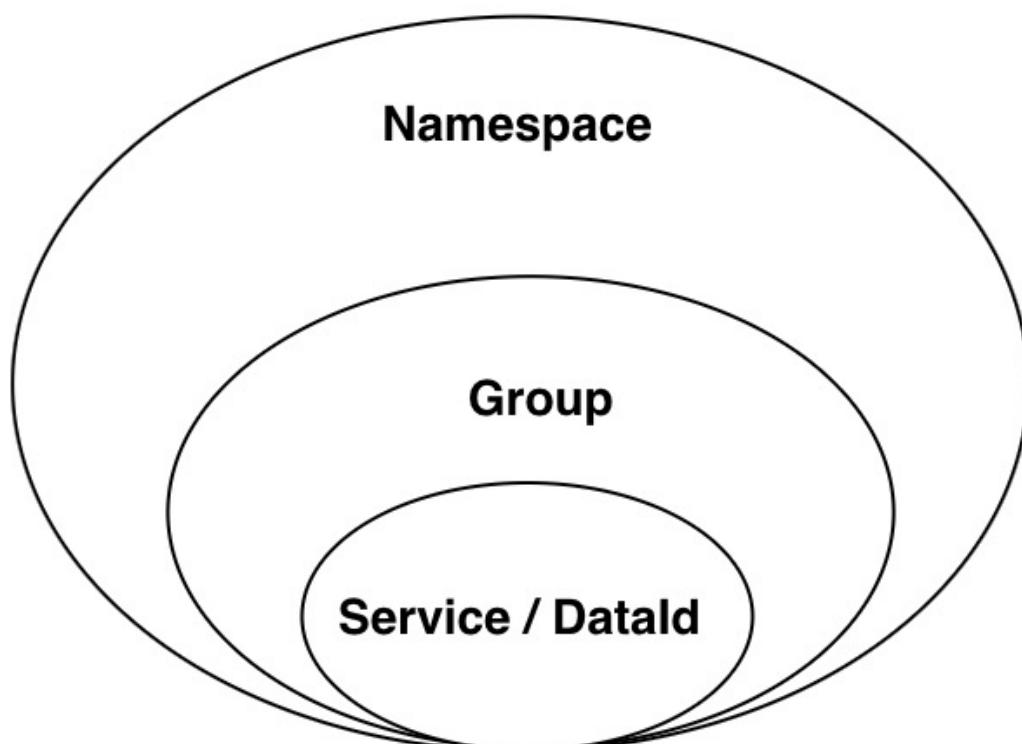
## 数据模型-Namespace-Group-DataId

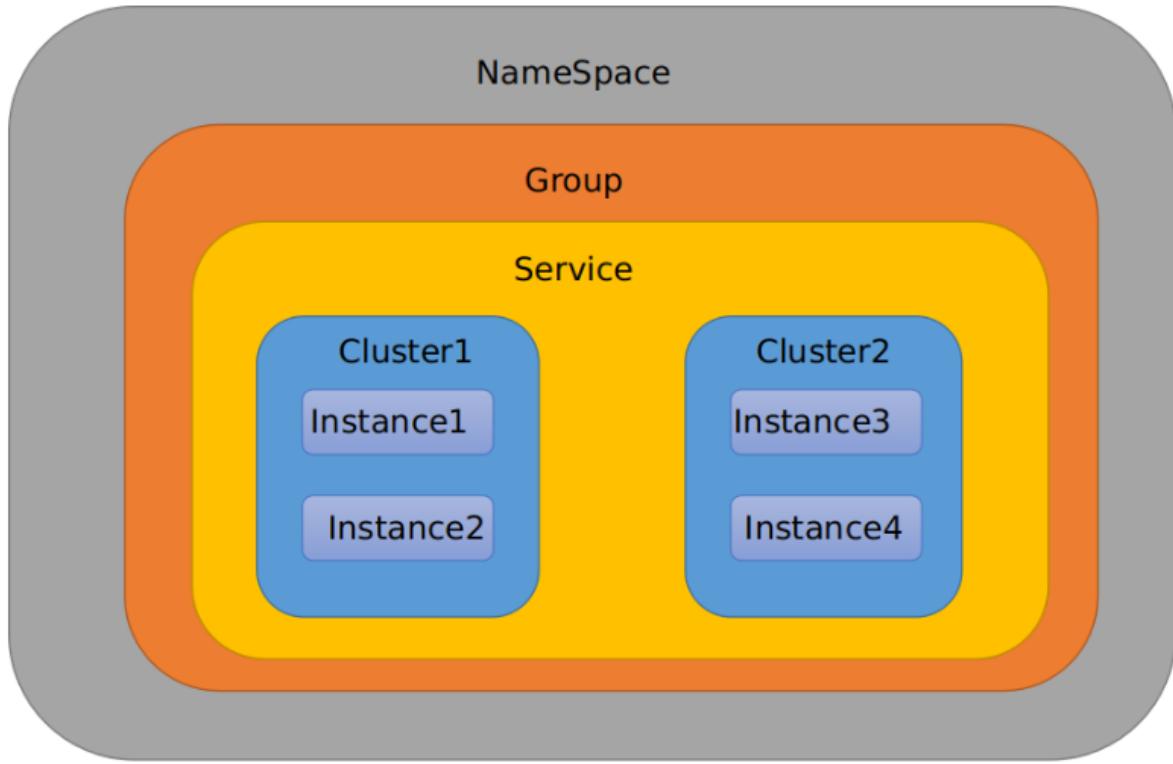
### 简介

<https://nacos.io/docs/v2/architecture/>

Nacos 数据模型 Key 由三元组唯一确定, Namespace默认是空串, 公共命名空间 (public) , 分组默认是 DEFAULT\_GROUP。

## Nacos data model





是什么	类似Java里面的package名和类名，最外层的Namespace是可以用于区分部署环境的，Group和DataID逻辑上区分两个目标对象
默认值	默认情况：Namespace=public、Group=DEFAULT_GROUP 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可以包含一个或者多个Cluster（集群），Nacos默认Cluster是DEFAULT，Cluster是对指定微服务的一个虚拟划分。见下一节：服务领域模型-补充说明



# 加载配置

## DataID方案

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

- 默认命名空间+默认分组DEFAULT\_GROUP+新建DataID

新建配置的DataID **nacos-config-client-test.yaml**

The screenshot shows the Nacos Configuration Center interface. At the top, there are several search and filter options: 'public' (selected), 'Data ID' (set to '已开启默认模糊查询'), 'Group' (set to '已开启默认模糊查询'), '默认模糊匹配' (switched on), '查询' (button), '高级查询' (dropdown), '导入配置' (button), and a '+' button. Below these, a message says '查询到 2 条满足要求的配置。'. A table lists two configuration items:

	Data ID	Group	归属应用	操作
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">⋮</a>
<input type="checkbox"/>	nacos-config-client-test.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">⋮</a>

- 修改配置中心模块3377

修改全局配置

application.yaml

```
server:
  port: 3377
spring:
  profiles:
    #      active: dev #开发环境
    #      active: prod # 生产环境
    active: test # 测试环境
```

配置完成进行测试：<http://localhost:3377/config/info>

## Group方案

通过Group实现环境区分

- 默认空间public + 新建PROD\_GROUP + 新建DataID

新建GROUP: **PROD\_GROUP**

新建prod配置DataID: **nacos-config-client-prod.yaml**

The screenshot shows the Nacos Configuration Center interface for creating a new group. At the top, there are fields for '命名空间' (Namespace) and 'Data ID' (set to 'nacos-config-client-prod.yaml'). Below these, there is a 'Group' field set to 'PROD\_GROUP', which is also highlighted with a red box. There is a '描述' (Description) input field and a '更多高级选项' (More Advanced Options) link. At the bottom, there is a '配置内容' (Content) section with a code editor containing YAML configuration. The configuration content is:

```
1 | config:
2 |   info: nacos-config-client-prod.yaml,PROD_GROUP,come from nacosconfig,version=1
```

### 配置管理

The screenshot shows the Nacos Configuration Center interface for managing configurations. At the top, there are search and filter options: 'public' (selected), 'Data ID' (set to '已开启默认模糊查询'), 'Group' (set to '已开启默认模糊查询'), '默认模糊匹配' (switched on), '查询' (button), '高级查询' (dropdown), '导入配置' (button), and a '+' button. Below these, a message says '查询到 3 条满足要求的配置。'. A table lists three configuration items:

	Data ID	Group	归属应用	操作
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">⋮</a>
<input type="checkbox"/>	nacos-config-client-prod.yaml	PROD_GROUP		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">⋮</a>

## 2. 修改配置中心模块3377

修改全局配置

application.yaml

```
server:
  port: 3377
spring:
  profiles:
    # active: dev #开发环境
    active: prod # 生产环境
    # active: test # 测试环境
```

bootstrap.yaml

新增 group: PROD\_GROUP

```
# nacos配置
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        # nacos服务注册中心地址
        server-addr: localhost:8848
      config:
        file-extension: yaml
        # nacos配置中心地址
        server-addr: localhost:8848
        group: PROD_GROUP
```

配置完成进行测试：<http://localhost:3377/config/info>

## Namespace方案

通过Namespace实现命名空间环境区分

1. 新建命名空间 **Prod\_Namespace** + group: **PROD\_GROUP** + DataID: **nacos-config-client-prod.yaml**

The screenshot shows the Nacos 2.2.3 interface. In the left sidebar, under '命名空间' (Namespace), the 'Prod\_Namespace' entry is highlighted. The main panel displays a table of namespaces, where 'Prod\_Namespace' is listed with a DataID of 'nacos-config-client-prod.yaml' and a Group of 'PROD\_GROUP'. Below the table, a configuration management section shows the selected namespace 'Prod\_Namespace' and a search bar with filters for 'Data ID' (nacos-config-client-prod.yaml) and 'Group' (PROD\_GROUP). A message at the top indicates that the cluster has not yet been secured.

## 2. 修改全局配置

bootstrap.yaml

```
# nacos配置
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        # nacos服务注册中心地址
        server-addr: localhost:8848
      config:
        file-extension: yaml
        # nacos配置中心地址
        server-addr: localhost:8848
        group: PROD_GROUP
        namespace: Prod_Namespace
```

application.yaml

```
server:
  port: 3377
spring:
  profiles:
    # active: dev #开发环境
    active: prod # 生产环境
    # active: test # 测试环境
```

配置完成进行测试：<http://localhost:3377/config/info>

## Sentinel(熔断和限流)

官网：<https://sentinelguard.io/zh-cn/index.html>  
对标：Spring Cloud CircuitBreaker  
文档：<https://github.com/alibaba/Sentinel/wiki/>主页  
下载：<https://github.com/alibaba/Sentinel/releases>  
运行：java -jar sentinel-dashboard.-1.8.6.jar  
后台端口号为：8719  
前台端口号为：8080  
用户名，密码：sentinel, sentinel

## 简介

主要以流量为切入点，从流量路由、流量控制、流量整形、熔断降级、系统自适应过载保护、热点流量防护等多个维度来帮助开发者保障微服务的稳定性。

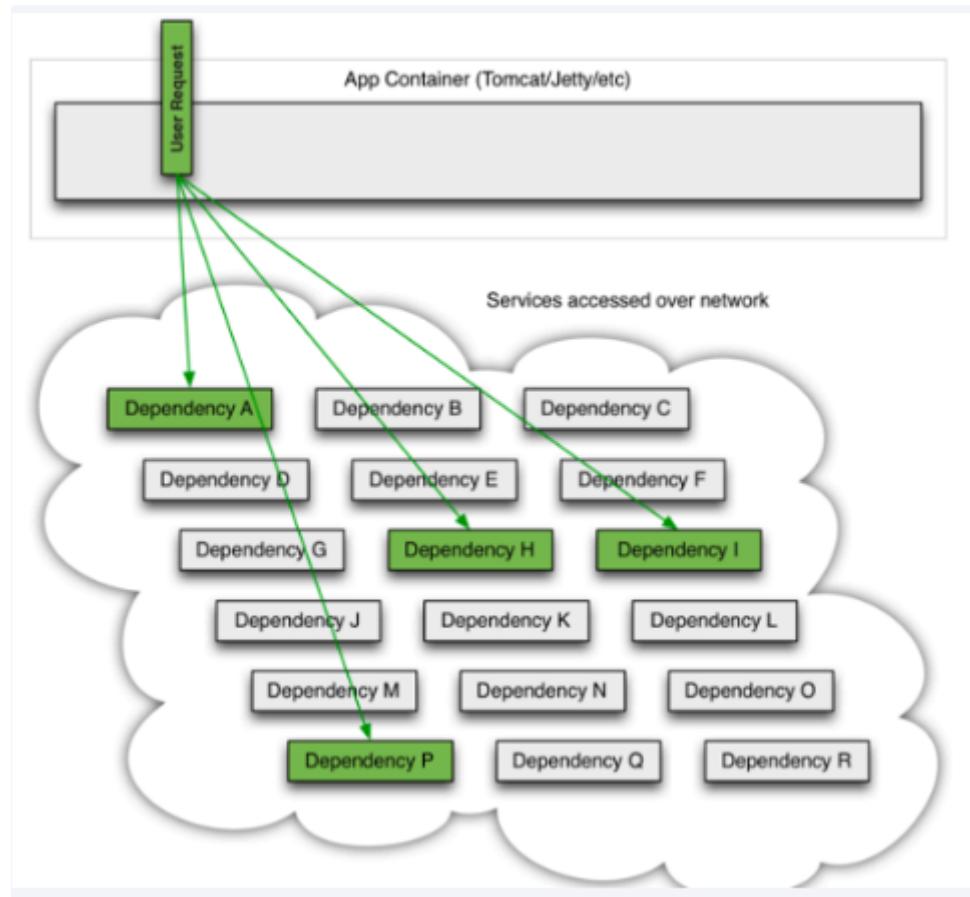
- Sentinel的特征
  - 丰富的应用场景：Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。
  - 完备的实时监控：Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
  - 广泛的开源生态：Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Apache Dubbo、gRPC、Quarkus 的整合。您只需要引入相应的依赖并进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：



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



#### • 服务降级

- 服务降级，说白了就是一种服务托底方案，如果服务无法完成正常的调用流程，就使用默认的托底方案来返回数据

#### • 服务熔断

- 在分布式与微服务系统中，如果下游服务因为访问压力过大导致响应很慢或者一直调用失败时，上游服务为了保证系统的整体可用性，会暂时断开与下游服务的调用连接。这种方式就是熔断。类比保险丝达到最大服务访问后，直接拒绝访问，拉闸限电，然后调用服务降级的方法并返回友好提示
- 服务熔断一般情况下会有三种状态：闭合、开启和半熔断；
- 闭合状态(保险丝闭合通电OK)：服务一切正常，没有故障时，上游服务调用下游服务时，不会有任何限制
- 开启状态(保险丝断开通电Error)：上游服务不再调用下游服务的接口，会直接返回上游服务中预定的方法
- 半熔断状态：处于开启状态时，上游服务会根据一定的规则，尝试恢复对下游服务的调用。此时，上游服务会以有限的流量来调用下游服务，同时，会监控调用的成功率。如果成功率达到预期，则进入关闭状态。如果未达到预期，会重新进入开启状态

#### • 服务限流

- 服务限流就是限制进入系统的流量，以防止进入系统的流量过大而压垮系统。其主要的作用就是保护服务节点或者集群后面的数据节点，防止瞬时流量过大使服务和数据崩溃（如前端缓存大量失效），造成不可用；还可用于平滑请求，类似秒杀高并发等操作，严禁一窝蜂的过来拥挤，大家排队，一秒钟N个，有序进行
- 限流算法有两种，一种就是简单的请求总量计数，一种就是时间窗口限流（一般为1s），如令牌桶算法和漏桶算法就是时间窗口的限流算法

#### • 服务隔离

- 有点类似于系统的垂直拆分，就按照一定的规则将系统划分成多个服务模块，并且每个服务模块之间是互相独立的，不会存在强依赖的关系。如果某个拆分后的服务发生故障后，能够将故障产生的影响限制在某个具体的服务内，不会向其他服务扩散，自然也就不会对整体服务产生致命的影响
- 线程池隔离和信号量隔离

#### • 服务超时

- 整个系统采用分布式和微服务架构后，系统被拆分成一个个小服务，就会存在服务与服务之间互相调用的现象，从而形成一个个调用链。
- 形成调用链关系的两个服务中，主动调用其他服务接口的服务处于调用链的上游，提供接口供其他服务调用的服务处于调用链的下游。服务超时就是在上游服务调用下游服务时，设置一个最大响应时间，如果超过这个最大响应时间下游服务还未返回结果，则断开上游服务与下游服务之间的请求连接，释放资源。

## 整合Sentinel入门案例(8401)

1. 打开nacos(8848)和sentinel(8080)服务，进入前台页面
2. 新建微服务8401模块，[cloudalibaba-sentinel-service8401](#)

POM文件

```
<dependencies>
    <!--SpringCloud alibaba sentinel -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
    </dependency>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!-- 引入自己定义的api通用包 -->
    <dependency>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
```

```

</dependency>
<!--SpringBoot通用依赖模块-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```

## 全局配置

application.yaml

```

server:
  port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
    nacos:
      discovery:
        #Nacos服务注册中心地址
        server-addr: localhost:8848
    sentinel:
      transport:
        #配置Sentinel dashboard控制台服务地址
        dashboard: localhost:8080
        #默认8719端口，假如被占用会自动从8719开始依次+1扫描，直至找到未被占用的端口
        port: 8719

```

主启动类加上注解 `@EnabledDiscoveryClient`

## 新建业务类 FlowLimitController

```
@RestController
public class FlowLimitController
{
    @GetMapping("/testA")
    public String testA()
    {
        return "-----testA";
    }

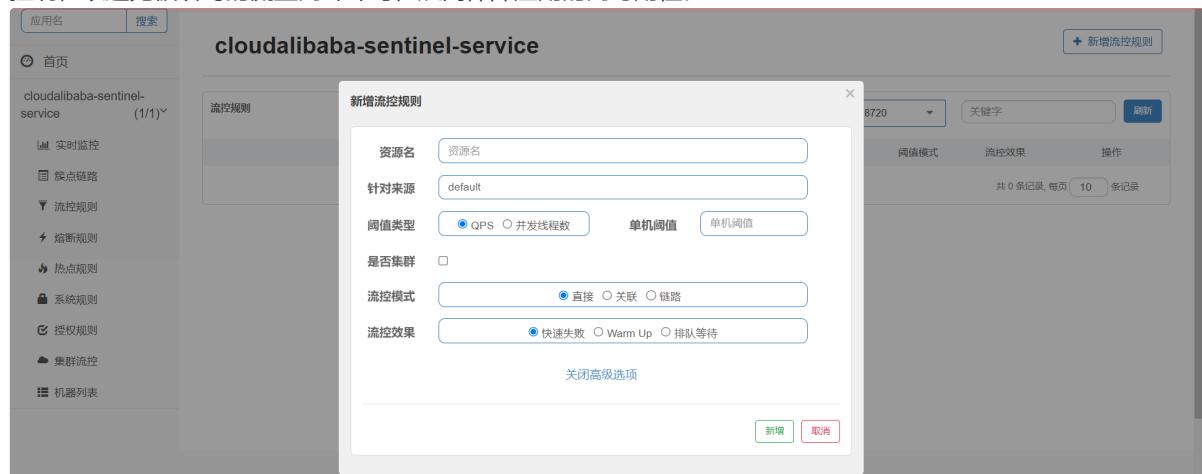
    @GetMapping("/testB")
    public String testB()
    {
        return "-----testB";
    }
}
```

然后访问业务类接口，使其进入sentinel中进行记录

## 流控规则

### 介绍

**流量控制** (flow control)，其原理是监控应用流量的 QPS 或并发线程数等指标，当达到指定的阈值时对流量进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。



资源名	资源的唯一名称，默认就是请求的接口路径，可以自行修改，但是要保证唯一。
针对来源	具体针对某个微服务进行限流，默认值为default，表示不区分来源，全部限流。
阈值类型	QPS表示通过QPS进行限流，并发线程数表示通过并发线程数限流。
单机阈值	与阈值类型组合使用。如果阈值类型选择的是QPS，表示当调用接口的QPS达到阈值时，进行限流操作。 如果阈值类型选择的是并发线程数，则表示当调用接口的并发线程数达到阈值时，进行限流操作。
是否集群	选中则表示集群环境，不选中则表示非集群环境。

# 流控模式

流控模式

直接  关联  链路

## 直接

默认的流控模式，当接口达到限流条件时，直接开启限流功能

表示1秒钟内查询1次就是OK，若超过次数1，就直接-快速失败，报默认错误

The screenshot shows the 'cloudalibaba-sentinel-service' dashboard. On the left sidebar, under 'flow control rules', there is a single entry '(1/1)'. In the center, a modal window titled '新增流控规则' (Add Flow Control Rule) is open. Inside, the 'flow control mode' section has the '直接' (Direct) radio button selected. Other settings include resource name '/testA', default source, QPS threshold of 1, and fast failure effect. The background shows a table of existing flow control rules.

测试地址：<http://localhost:8401/testA>，快速点击，报错 Blocked by Sentinel (flow limiting)

## 关联

当与A关联的资源B达到阈值后，就限流A自己

当关联资源/testB的qps阈值超过1时，就限流/testA的Rest访问地址，当关联资源到阈值后限制配置好的资源名，B惹事，A挂了

The screenshot shows the same 'cloudalibaba-sentinel-service' dashboard. The 'flow control rules' sidebar still shows '(1/1)'. The modal window now has the '关联' (Associated) radio button selected under 'flow control mode'. It also includes '关联资源' (/testB) and other standard settings like resource name and QPS threshold. The background table shows one rule where the 'flow control mode' is '关联' and the 'flow control effect' is '快速失败' (Fast Failure).

测试地址：<http://localhost:8401/testB>，访问多次testB，则限流testA

## 链路

来自不同链路的请求对同一个目标访问时，实施针对性的不同限流措施，比如C请求就限流，D请求就OK

1. 修改微服务 [cloudalibaba-sentinel-service8401](#)

新增FlowLimitService

```

@Service
public class FlowLimitService {
    @SentinelResource(value = "common")
    public void common() {
        System.out.println("-----FlowLimitService come in");
    }
}

```

修改FFlowLimitController

```

@RestController
public class FlowLimitController{
    /**流控-链路演示demo
     * C和D两个请求都访问flowLimitService.common()方法，阈值到达后对C限流，对D不管
     */
    @Resource private FlowLimitService flowLimitService;
    @GetMapping("/testC")
    public String testC(){
        flowLimitService.common();
        return "-----testC";
    }
    @GetMapping("/testD")
    public String testD(){
        flowLimitService.common();
        return "-----testD";
    }
}

```

修改全局配置

```

spring:
  cloud:
    sentinel:
      # controller层的方法对service层调用不认为是同一个根链路
      web-context-unify: false

```

## 2. sentinel配置



测试地址: <http://localhost:8401/testC>, <http://localhost:8401/testD>, /testC 500, /testD OK

# 流控效果

流控效果

快速失败  Warm Up  排队等待

## 直接 -> 快速失败(默认)

直接抛出异常 locked by Sentinel (flow limiting)

## 预热WarmUp

限流冷启动: <https://github.com/alibaba/Sentinel/wiki/限流---冷启动>

当流量突然增大的时候，我们常常会希望系统从空闲状态到繁忙状态的切换的时间长一些。即如果系统在此之前长期处于空闲的状态，我们希望处理请求的数量是缓步的增多，经过预期的时间以后，到达系统处理请求个数的最大值。Warm Up (冷启动，预热) 模式就是为了实现这个目的。

- 说明：阈值除以冷却因子coldFactor(默认值为3)，经过预热时长后达到阈值
- 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。
- 源码：<com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController>

案例，单机阈值为10，预热时长设置5秒。

系统初始化的阈值为 $10 / 3$  约等于3，即单机阈值刚开始为3(我们人工设定单机阈值是10，sentinel计算后QPS判定为3开始)；

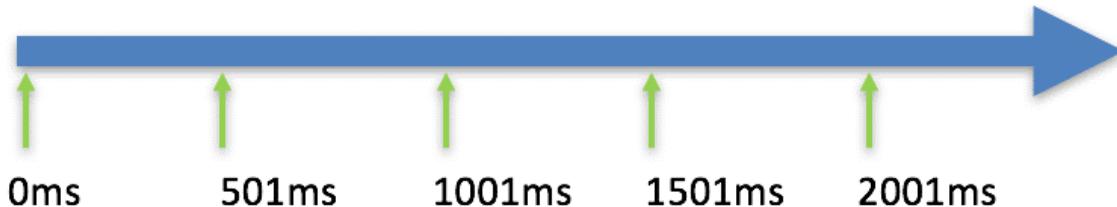
然后过了5秒后阈值才慢慢升高恢复到设置的单机阈值10，也就是说5秒钟内QPS为3，过了保护期5秒后QPS为10



测试地址：localhost:8401/testB

## 排队等待

### 时间轴



阈值QPS=2时，每隔500ms才允许通过下一个请求

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

注意：匀速排队模式下不支持 OPS > 1000 的场景

### 1. 修改模块FlowLimitController

```
@GetMapping("/testE")
public String testE()
{
    System.out.println(System.currentTimeMillis()+"\ttestE,排队等待");
    return "-----testE";
}
```

### 2. sentinel配置

按照单机阈值，一秒钟通过一个请求，10秒后的请求作为超时处理，放弃



测试：<http://localhost:8401/testE>

通过jmeter进行压力测色hi

## 流控效果(并发线程数)



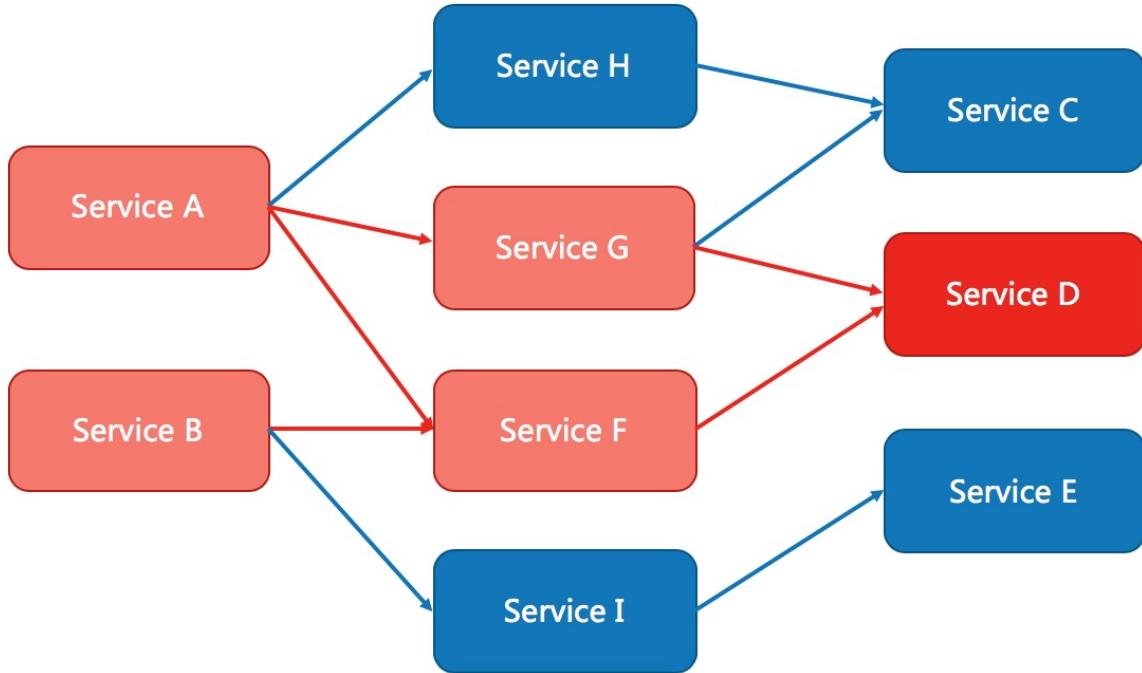
jmeter模拟多个线程+循环请求：<http://localhost:8401/testB>

## 熔断规则

<https://github.com/alibaba/Sentinel/wiki/熔断降级>

## 介绍

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。

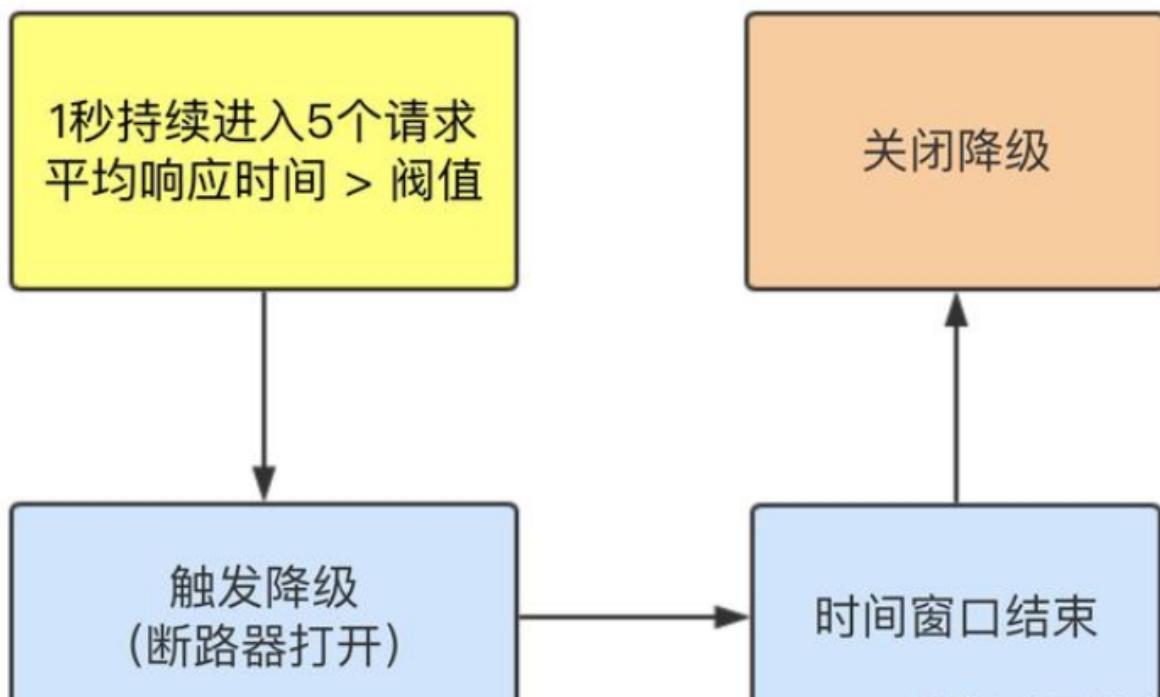


现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。因此我们需要对不稳定的弱依赖服务调用进行熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。熔断降级作为保护自身的手段，通常在客户端（调用端）进行配置。

## 熔断策略

### 慢调用比例

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



- 进入熔断状态判断依据：在统计时长内，实际请求数目 > 设定的最小请求数目且实际慢调用比例 > 比例阈值，进入熔断状态。



1. 调用：一个请求发送到服务器，服务器给与响应，一个响应就是一个调用。
  2. 最大RT：即最大的响应时间，指系统对请求作出响应的业务处理时间。
  3. 慢调用：处理业务逻辑的实际时间>设置的最大RT时间，这个调用叫做慢调用。
  4. 慢调用比例：在所有调用中，慢调用占有实际的比例 = 慢调用次数 / 总调用次数
  5. 比例阈值：自己设定的，比例阈值 = 慢调用次数 / 调用次数
  6. 统计时长：时间的判断依据
  7. 最小请求数：设置的调用最小请求数，上图比如1秒钟打进来10个线程（大于我们配置的5个了）调用被触发
- 触发条件 + 熔断状态
    - 熔断状态(保险丝跳闸断电，不可访问)：在接下来的熔断时长内请求会自动被熔断
    - 探测恢复状态(探路先锋)：熔断时长结束后进入探测恢复状态
    - 结束熔断(保险丝闭合恢复，可以访问)：在探测恢复状态，如果接下来的一个请求数响应时间小于设置的慢调用 RT，则结束熔断，否则继续熔断。

### 1. 修改 8401 模块 FlowLimitController

```
// 新增熔断规则-慢调用比例
@GetMapping("/testF")
public String testF() {
    //暂停几秒钟线程
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("----测试:新增熔断规则-慢调用比例 ");
    return "-----testF 新增熔断规则-慢调用比例";
}
```

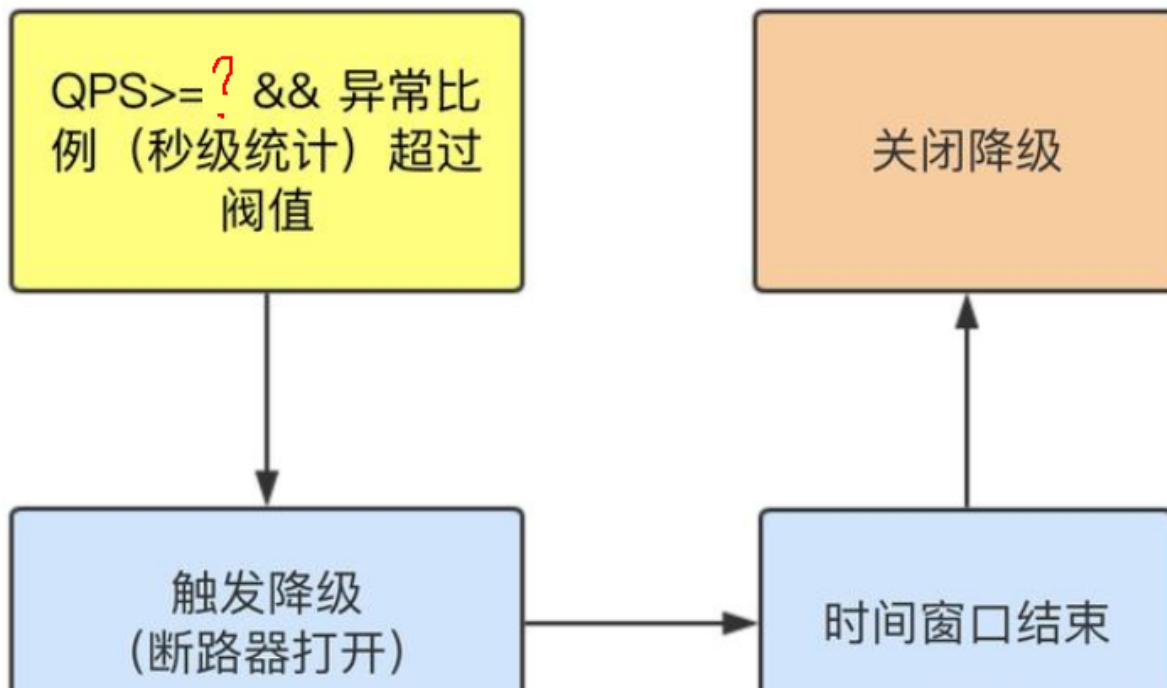
### 2. 增加熔断慢调用配置



3. 使用 **jmeter** 进行慢调用测试，测试地址：<http://localhost:8401/testF>

## 异常比例

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



### 1. 修改 8401 模块 FlowLimitController

```
@GetMapping("/testG")
public String testG()
{
    System.out.println("----测试:新增熔断规则-异常比例 ");
    int age = 10/0;
    return "-----testG,新增熔断规则-异常比例 ";
}
```

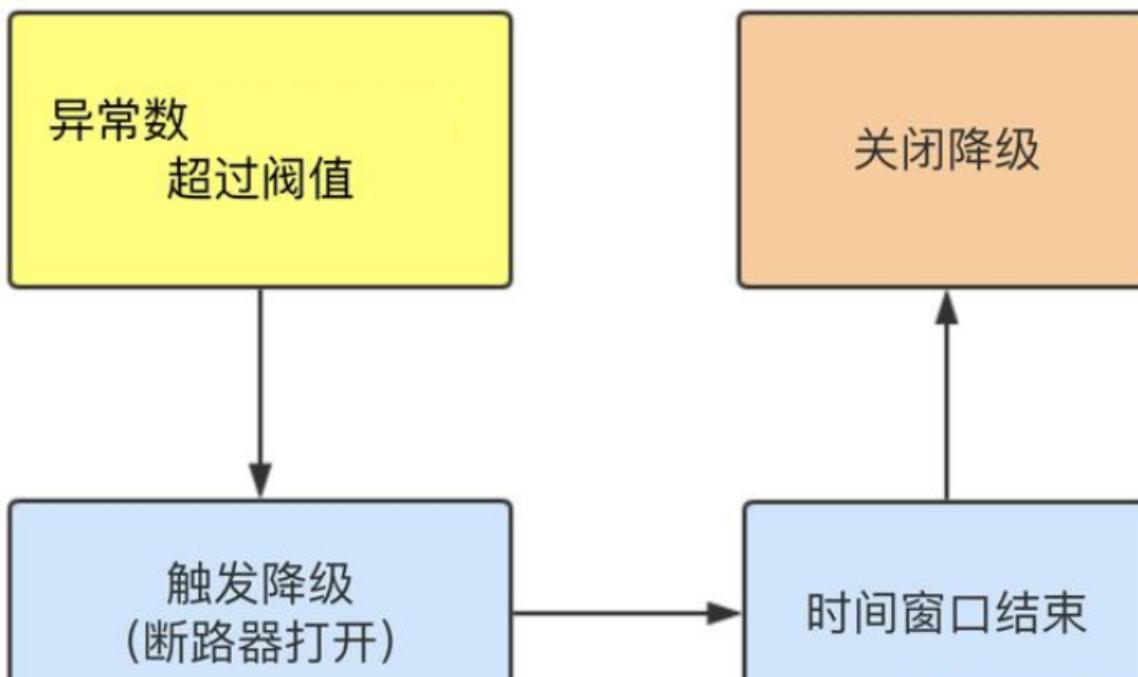
### 2. 增加熔断异常比例配置



### 3. 使用 jmeter 进行异常比例测试，测试地址: <http://localhost:8401/testG>

## 异常数

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

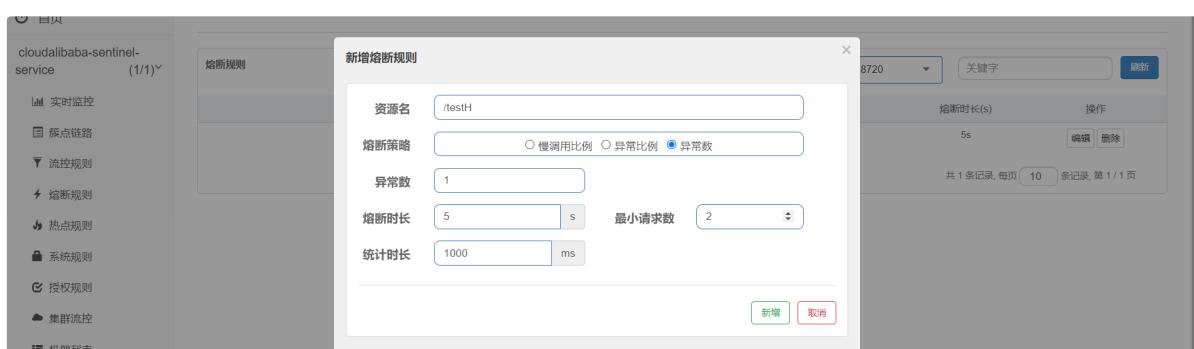


### 1. 修改 8401 模块 FlowLimitController

```

@GetMapping("/testH")
public String testH()
{
    System.out.println("----测试:新增熔断规则-异常数 ");
    int age = 10/0;
    return "-----testH,新增熔断规则-异常数 ";
}
  
```

### 2. 增加熔断异常比例配置



3. 使用 **jmeter** 进行异常数测试, 测试地址: <http://localhost:8401/testH>

## @SentinelResource注解

SentinelResource是一个流量防卫防护组件注解，用于指定防护资源，对配置的资源进行流量控制、熔断降级等功能

源码说明：

```

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface SentinelResource {

    //资源名称
    String value() default "";
  
```

```

//entry类型，标记流量的方向，取值IN/OUT，默认是OUT
EntryType entryType() default EntryType.OUT;
//资源分类
int resourceType() default 0;

//处理BlockException的函数名称，函数要求：
//1. 必须是 public
//2. 返回类型 参数与原方法一致
//3. 默认需和原方法在同一个类中。若希望使用其他类的函数，可配置blockHandlerClass，并指定blockHandlerClass里面的方法。
String blockHandler() default "";

//存放blockHandler的类，对应的处理函数必须static修饰。
Class<?>[] blockHandlerClass() default {};

//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进行处理。若同时配置了 fallback 和 defaultFallback，以fallback为准。函数要求：
//1. 返回类型与原方法一致
//2. 参数类型需要和原方法相匹配
//3. 默认需和原方法在同一个类中。若希望使用其他类的函数，可配置fallbackClass，并指定fallbackClass里面的方法。
String fallback() default "";

//存放fallback的类。对应的处理函数必须static修饰。
String defaultFallback() default "";

//用于通用的 fallback 逻辑。默认fallback函数可以针对所有类型的异常进行处理。若同时配置了 fallback 和 defaultFallback，以fallback为准。函数要求：
//1. 返回类型与原方法一致
//2. 方法参数列表为空，或者有一个 Throwable 类型的参数。
//3. 默认需和原方法在同一个类中。若希望使用其他类的函数，可配置fallbackClass，并指定fallbackClass里面的方法。
Class<?>[] fallbackClass() default {};

//需要trace的异常
Class<? extends Throwable>[] exceptionsToTrace() default {Throwable.class};

//指定排除忽略掉哪些异常。排除的异常不会计入异常统计，也不会进入fallback逻辑，而是原样抛出。
Class<? extends Throwable>[] exceptionsToIgnore() default {};
}

```

## 按照rest地址限流 + 默认限流返回

通过访问的rest地址来限流，会返回Sentinel自带默认的限流处理消息

### 1. 添加业务类 RateLimitController

```

@RestController
@Slf4j
public class RateLimitController
{
    @GetMapping("/rateLimit/byUrl")
    public String byUrl()
    {
        return "按rest地址限流测试OK";
    }
}

```

## 2. 配置sentinel



测试地址：<http://localhost:8401/rateLimit/byUrl>, 会返回自带默认限流处理结果

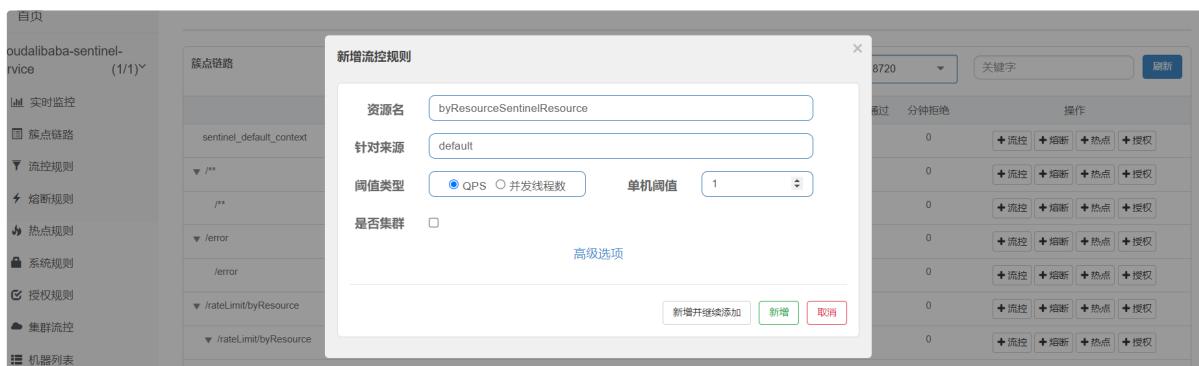
## 按SentinelResource资源名称限流+自定义返回

不想用默认的返回提示，返回自定义限流提示

### 1. 修改业务类 RateLimitController

```
@RestController
@Slf4j
public class RateLimitController {
    @GetMapping("/rateLimit/byResource")
    // 使用value作为配置流控规则资源名
    @SentinelResource(value = "byResourceSentinelResource",blockHandler =
    "handleException")
    public String byResource() {
        return "按资源名称SentinelResource限流测试OK";
    }
    public String handleException(BlockException exception) {
        return "服务不可用@SentinelResource启动"+"\t"+o(╥_╥)o";
    }
}
```

### 2. 配置流控规则



## 按SentinelResource资源名称限流+自定义限流返回+服务降级处理

按SentinelResource配置，点击超过限流配置返回自定义限流提示+程序异常返回fallback服务降级

### 1. 修改业务类 RateLimitController

```
@RestController
@Slf4j
public class RateLimitController{
    @GetMapping("/rateLimit/doAction/{id}")
```

```

    @SentinelResource(value = "doActionSentinelResource",
                      blockHandler = "doActionBlockHandler", fallback =
"doActionFallback")
    public String doAction(@PathVariable("id") Integer id) {
        if (id == 0) {
            throw new RuntimeException("id等于0直接异常");
        }
        return "doAction";
    }
    public String doActionBlockHandler(@PathVariable("id") Integer id,
                                       BlockException blockException) {
        log.error("sentinel配置自定义限流了:{}" , blockException);
        return "sentinel配置自定义限流了";
    }
    public String doActionFallback(@PathVariable("id") Integer id, Throwable e)
{
    log.error("程序逻辑异常了:{}" , e);
    return "程序逻辑异常了" + "\t" + e.getMessage();
}
}

```

## 2. 配置流控规则



- blockHandler: 主要针对sentinel配置后出现的违规情况处理
- fallback: 程序异常JVM抛出的异常服务降级

## 热点规则

热点即经常访问的数据，很多时候我们希望统计或者限制某个热点数据中访问频次最高的TopN数据，并对其进行限流或者其它操作  
<https://github.com/alibaba/Sentinel/wiki/热点参数限流>

### 1. 修改业务类 RateLimitController

```

@RestController
@Slf4j
public class RateLimitController{
    @GetMapping("/testHotKey")
    @SentinelResource(value = "testHotKey",blockHandler =
"dealHandler_testHotKey")
    public String testHotKey(@RequestParam(value = "p1",required = false)
String p1,
                           @RequestParam(value = "p2",required = false)
String p2){
        return "-----testHotKey";
    }
    public String dealHandler_testHotKey(String p1,String p2,BlockException
exception)
    {

```

```

        return "----dealHandler_testHotKey";
    }
}

```

## 2. 热点规则配置

The screenshot shows the 'Sentinel 控制台 1.8.6' interface. On the left sidebar, under the 'cloudalibaba-sentinel-service' section, the '热点规则' (Hotpoint Rules) item is selected and highlighted with a red box. In the center, a modal window titled '编辑热点规则' (Edit Hotpoint Rule) is open. Inside, the '资源名' (Resource Name) is set to 'testHotKey', '限流模式' (Flow Control Mode) is 'QPS 模式' (QPS Mode), '参数索引' (Parameter Index) is '0', '单机阈值' (Single Machine Threshold) is '1', and '统计窗口时长' (Statistical Window Duration) is '1 秒' (1 second). A checkbox for '是否集群' (Is Clustered) is unchecked. At the bottom right of the modal are '保存' (Save) and '取消' (Cancel) buttons.

## 3. 测试：

1. <http://localhost:8401/testHotKey?p1=123>
2. <http://localhost:8401/testHotKey?p1=123&p2=123>
3. <http://localhost:8401/testHotKey?p2=123>

## 4. 参数例外项配置

1. 当参数为某个特殊值时，到达某个约定值时【普通正常限流】规则失效
2. 参数必须是基本类型或者是String

This screenshot shows the same control panel as before, but the '参数例外项' (Parameter Exception Item) tab is active in the modal. It displays a table with one row of data. The '参数类型' (Parameter Type) is 'java.lang.String', '参数值' (Parameter Value) is '5', '限流阈值' (Flow Control Threshold) is '200', and there is a green '+' add button highlighted with a red box. Below the table, there is a '关闭高级选项' (Close Advanced Options) button. At the bottom right of the modal are '保存' (Save) and '取消' (Cancel) buttons.

# 授权规则

1. 在某些场景下，需要根据调用接口的来源判断是否允许执行本次请求。此时就可以使用Sentinel提供的授权规则来实现，Sentinel的授权规则能够根据请求的来源判断是否允许本次请求通过
2. 在Sentinel的授权规则中，提供了 白名单与黑名单 两种授权类型。白放行、黑禁止
3. 调用方信息通过 `ContextUtil.enter(resourceName, origin)` 方法中的 `origin` 参数传入

<https://github.com/alibaba/Sentinel/wiki/黑白名单控制>

## 1. 修改8401模块

新增EmpowerController

```

@RestController
@Slf4j
public class EmpowerController //Empower授权规则, 用来处理请求的来源
{
    @GetMapping(value = "/empower")
    public String requestSentinel4(){
        log.info("测试Sentinel授权规则empower");
        return "Sentinel授权规则";
    }
}

```

新增MyRequestOriginParser

```

@Component
public class MyRequestOriginParser implements RequestOriginParser
{
    @Override
    public String parseOrigin(HttpServletRequest httpServletRequest) {
        // 请求中需带参数 serverName
        return httpServletRequest.getParameter("serverName");
    }
}

```

## 2. 配置sentinel



此时, 如果请求中带参数 `serverName=test1` 或 `serverName=test2` 就会被黑名单屏蔽

- 测试: <http://localhost:8401/empower>, <http://localhost:8401/empower?serverName=test1>, <http://localhost:8401/empower?serverName=test2>

## 规则持久化

将限流配置规则持久化进Nacos保存, 只要刷新8401某个rest地址, sentinel控制台上的流控规则就能看到, 只要Nacos里面的配置不能删除, 针对8401上sentinel上的流控规则持续有效

- 修改 8401 模块
- POM文件

```

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

```

- 全局配置

```
spring:
cloud:
sentinel:
datasource:
# 自定义key
ds1:
nacos:
server-addr: localhost:8848
dataId: ${spring.application.name}
groupId: DEFAULT_GROUP
data-type: json
# com.alibaba.cloud.sentinel.datasource.RuleType
rule-type: flow
```

```
流量控制规则 FlowRule
FLOW( name: "flow", FlowRule.class),
| degrade.
熔断降级规则 DegradeRule
DEGRADE( name: "degrade", DegradeRule.class),
| param flow.
热点规则 ParamFlowRule
PARAM_FLOW( name: "param-flow", ParamFlowRule.class),
| system.
系统保护规则 SystemRule
SYSTEM( name: "system", SystemRule.class),
| authority.
访问控制规则 AuthorityRule
AUTHORITY( name: "authority", AuthorityRule.class),
| gateway flow.
```

#### 4. 添加Nacos业务规则配置

```
[{
  {
    // 给这个请求进行限流，资源名称
    "resource": "/rateLimit/byurl",
    // 来源应用
    "limitApp": "default",
    // 阈值类型：0表示线程数，1表示QPS
    "grade": 1,
    // 单机阈值
    "count": 1,
    // strategy: 流控模式，0表示直接，1表示关联，2表示链路
    "strategy": 0,
    // 流控效果：0表示快速失败，1表示warm up，2表示排队等待
    "controlBehavior": 0,
    // 是否集群
    "clusterMode": false
  }
]
```

在Nacos配置管理中新建配置

当前集群没有开启鉴权,请参考[文档](#)开启鉴权~

## 编辑配置

The screenshot shows the CloudAlibaba Sentinel configuration interface. At the top, there are fields for '命名空间' (Namespace) set to 'spring.application.name', 'Data ID' set to 'cloudalibaba-sentinel-service', and 'Group' set to 'DEFAULT\_GROUP'. Below these are '更多高级选项' (More Advanced Options) and a '描述' (Description) input field. A note below says 'Beta发布' (Beta Release) with a checkbox. Configuration formats are listed: TEXT (radio button), JSON (radio button, selected), XML, YAML, HTML, Properties. The main area shows a code editor with the following JSON configuration:

```
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  ]
```

4. 测试: <http://localhost:8401/rateLimit/byUrl>

同时, 关闭8401, 并重新打开查看是否显示持久化配置

# OpenFeign和Sentinel集成实现fallback服务降级

## 需求说明

1. `cloudalibaba-consumer-nacos-order83` 通过OpenFeign调用 `cloudalibaba-provider-payment9001`
2. 83 通过OpenFeign调用 9001微服务, 正常访问 OK
3. 83 通过OpenFeign调用 9001微服务, 异常访问 error, 访问者要有fallback服务降级的情况, 不要持续访问9001加大微服务负担, 但是通过feign接口调用的又方法各自不同, 如果每个不同方法都加一个fallback配对方法, 会导致代码膨胀不好管理, 工程埋雷...../(ㄒoㄒ)/~~
4. `public @interface FeignClient`, 通过fallback属性进行统一配置, feign接口里面定义的全部方法都走统一的服务降级, 一个搞定即可。
5. 9001微服务自身还带着sentinel内部配置的流控规则, 如果满足也会被触发, 也即本例有2个Case
  1. OpenFeign接口的统一fallback服务降级处理
  2. Sentinel访问触发了自定义的限流配置, 在注解 `@SentinelResource` 里面配置的blockHandler方法。

## 编程步骤

1. 启动nacos服务器8848

2. 启动sentinel

## 修改服务提供方

`cloudalibaba-provider-payment9001`

1. POM

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

```

<!--alibaba-sentinel-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
<!--nacos-discovery-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
<!-- 引入自己定义的api通用包 -->
<dependency>
    <groupId>com.xxx.cloud</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<!--SpringBoot通用依赖模块-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```

## 2. 全局配置

```

server:
  port: 9001

spring:
  application:

```

```

name: nacos-payment-provider
cloud:
nacos:
discovery:
server-addr: localhost:8848 #配置Nacos地址
sentinel:
transport:
#配置Sentinel dashboard控制台服务地址
dashboard: localhost:8080
#默认8719端口，假如被占用会自动从8719开始依次+1扫描，直至找到未被占用的端口
port: 8719

```

3. 主启动类：添加`@EnableDiscoveryClient` 注解

4. 新建 `PayAlibabaController`

```

@RestController
public class PayAlibabaController {
    @Value("${server.port}")
    private String serverPort;

    @GetMapping(value = "/pay/nacos/{id}")
    public String getPayInfo(@PathVariable("id") Integer id) {
        return "nacos registry, serverPort: " + serverPort + "\t id" + id;
    }

    @GetMapping("/pay/nacos/get/{orderNo}")
    @SentinelResource(value = "getPayByOrderNo", blockHandler =
"handlerBlockHandler")
    public ResultData getPayByOrderNo(@PathVariable("orderNo") String orderNo) {
        //模拟从数据库查询出数据并赋值给DTO
        PayDTO payDTO = new PayDTO();
        payDTO.setId(1024);
        payDTO.setOrderNo(orderNo);
        payDTO.setAmount(BigDecimal.valueOf(9.9));
        payDTO.setPayNo("pay:" + IdUtil.fastUUID());
        payDTO.setUserId(1);

        return ResultData.success("查询返回值: " + payDTO);
    }
    public ResultData handlerBlockHandler(@PathVariable("orderNo") String
orderNo,
                                         BlockException exception){
        return ResultData.fail(ReturnCodeEnum.RC500.getCode(),"getPayByOrderNo服
务不可用, " +
                "触发sentinel流控配置规则" + "\t" + "o(\u2014\u2014)o");
    }
    /**
     * fallback服务降级方法纳入到Feign接口统一处理，全局一个
     */
    public ResultData myFallback(@PathVariable("orderNo") String
orderNo, Throwable throwable){
        return ResultData.fail(ReturnCodeEnum.RC500.getCode(),"异常情
况: " + throwable.getMessage());
    }
}

```

5. 测试：<http://localhost:9001/pay/nacos/get/13>

## 修改公共API模块

cloud-api-commons

### 1. POM

```
<dependencies>
    <!--openfeign-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <!--alibaba-sentinel-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
    </dependency>
    <!--web + actuator-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <!--hutool-->
    <dependency>
        <groupId>cn.hutool</groupId>
        <artifactId>hutool-all</artifactId>
    </dependency>
</dependencies>

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

### 2. 新增 PayFeignSentinelApi 接口

```
@FeignClient(value = "nacos-payment-provider",
    // 服务降级处理，调用PayFeignSentinelFallback类
    fallback = PayFeignSentinelApiFallback.class)
public interface PayFeignSentinelApi{
    @GetMapping("/pay/nacos/get/{orderNo}")
    public ResultData getPayByOrderNo(@PathVariable("orderNo") String orderNo);
}
```

### 3. 新建全局统一服务降级类

```

@Component
// 实现PayFeignSentinelApi接口，进行实现每个方法的服务降级
public class PayFeignSentinelApiFallback implements PayFeignSentinelApi{
    @Override
    public ResultData getPayByOrderNo(String orderNo)
    {
        return ResultData.fail(ReturnCodeEnum.RC500.getCode(),
                "对方服务宕机或不可用，Fallback服务降级o(╥﹏╥)o");
    }
}

```

#### 4. gav坐标

```

<!-- 引入自己定义的api通用包 -->
<dependency>
    <groupId>com.xxx.cloud</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>

```

## 修改服务调用者

### cloudalibaba-consumer-nacos-order83

#### 1. POM

```

<dependencies>
    <!-- 引入自己定义的api通用包 -->
    <dependency>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--openfeign-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <!--alibaba-sentinel-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
    </dependency>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--loadbalancer-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
    <!--web + actuator-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>

```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
</dependencies>

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

```

## 2. 全局配置

```

server:
  port: 83

spring:
  application:
    name: nacos-order-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
#消费者将要去访问的微服务名称(nacos微服务提供者叫什么你写什么)
  service-url:
    nacos-user-service: http://nacos-payment-provider

# 激活Sentinel对Feign的支持
feign:
  sentinel:
    enabled: true

```

3. 主启动类：添加 `@EnableDiscoveryClient` 开启服务发现，添加 `@EnableFeignClient` 开启Feign功能

4. 修改业务类 `OrderNacosController`

```

@RestController
public class OrderNacosController
{
    @Resource
    private PayFeignSentinelApi payFeignSentinelApi;

    @GetMapping(value = "/consumer/pay/nacos/get/{orderNo}")
    public ResultData getPayByOrderNo(@PathVariable("orderNo") String orderNo){
        return payFeignSentinelApi.getPayByOrderNo(orderNo);
    }
}

```

5. 启动后会出现故障

```

16 public class OrderNacosController {
17
18     @Resource
19     private RestTemplate restTemplate;
20
21     @Resource
22     private PayFeignSentinelApi payFeignSentinelApi;

```

Services

Caused by: java.lang.IllegalStateException: Method PayFeignSentinelApi#getPayByOrderNo(String) not annotated with HTTP method type (ex. GET, POST)

Warnings:

- Class PayFeignSentinelApi has annotations [FeignClient] that are not used by contract Default
- Method getPayByOrderNo has an annotation GetMapping that is not used by contract Default

springboot+springcloud版本太高导致与阿里巴巴Sentinel不兼容

解决方案，要不将阿里Sentinel版本提高，要不将boot+cloud降低版本

```

<spring.boot.version>3.2.0</spring.boot.version>
<spring.cloud.version>2023.0.0</spring.cloud.version>

```

转换为

```

<spring.boot.version>3.0.9</spring.boot.version>
<spring.cloud.version>2022.0.2</spring.cloud.version>

```

## 6. 配置sentinel配置

cloud-gateway9527

cloud-provider-payment8001

cloud-provider-payment8002

cloudalibaba-config-nacos-client377

cloudalibaba-consumer-nacos-order83

cloudalibaba-provider-payment9001

src

main

java

com.atguigu.cloud

controller

PayAlibabaController

Main9001

resources

application.yml

target

pom.xml

cloudalibaba-sentinel-service8401

mybatis-generator [mybatis\_generator]

pom.xml

External Libraries

Scratches and Consoles

首页

cloudalibaba-sentinel-service (0/1)\*

nacos-payment-provider (1/2)

新增流控规则

资源名: getPayByOrderNo

针对来源: default

阈值类型: QPS

单机阈值: 1

是否集群: 不是

流控模式: 直接

流控效果: 快速失败

7. 测试: <http://localhost:83/consumer/pay/nacos/get/111>

## GateWay和Sentinel集成实现服务限流(9528)

<https://github.com/alibaba/Sentinel/wiki/网管限流#spring-cloud-gateway>  
**cloudalibaba-sentinel-gateway9528** 保护 **cloudalibaba-provider-payment9001**

1. 启动nacos服务 8848
2. 启动sentinel服务
3. 新建Module **cloudalibaba-sentinel-gateway9528**
4. POM

```

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>

```

```

<artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-transport-simple-http</artifactId>
    <version>1.8.6</version>
</dependency>
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>
    <version>1.8.6</version>
</dependency>
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
    <scope>compile</scope>
</dependency>
</dependencies>

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

```

## 5. 全局配置

```

server:
  port: 9528

spring:
  application:
    name: cloudalibaba-sentinel-gateway      # sentinel+gataway整合Case
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  gateway:
    routes:
      - id: pay_routh1 #pay_routh1          #路由的ID(类似mysql主键ID), 没有固定规则但要求唯一, 建议配合服务名
        uri: http://localhost:9001           #匹配后提供服务的路由地址
        predicates:
          - Path=/pay/**                  # 断言, 路径相匹配的进行路由

```

## 6. 主启动类: 加上 `@EnableDiscoveryClient` 注解

### 7. 根据官网进行改写, 配置config

<https://github.com/alibaba/Sentinel/wiki/网管限流#spring-cloud-gateway>

```

@Configuration
public class GatewayConfiguration {

    private final List<ViewResolver> viewResolvers;
    private final ServerCodecConfigurer serverCodecConfigurer;
}

```

```

public GatewayConfiguration(ObjectProvider<List<ViewResolver>>
viewResolversProvider, ServerCodecConfigurer serverCodecConfigurer) {
    this.viewResolvers =
    viewResolversProvider.getIfAvailable(Collections::emptyList);
    this.serverCodecConfigurer = serverCodecConfigurer;
}

@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public SentinelGatewayBlockExceptionHandler
sentinelGatewayBlockExceptionHandler() {
    // Register the block exception handler for Spring cloud Gateway.
    return new SentinelGatewayBlockExceptionHandler(viewResolvers,
serverCodecConfigurer);
}

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

@PostConstruct //javax.annotation.PostConstruct
public void doInit() {
    initBlockHandler();
}

//处理/自定义返回的例外信息
private void initBlockHandler() {
    Set<GatewayFlowRule> rules = new HashSet<>();
    rules.add(new GatewayFlowRule("pay_route1").setCount(2)
        .setIntervalSec(1));

    GatewayRuleManager.loadRules(rules);
    BlockRequestHandler handler = new BlockRequestHandler() {
        @Override
        public Mono<ServerResponse> handleRequest(ServerWebExchange
exchange, Throwable t) {
            Map<String, String> map = new HashMap<>();

            map.put("errorCode",
HttpStatus.TOO_MANY_REQUESTS.getReasonPhrase());
            map.put("errorMessage", "请求太过频繁，系统忙不过来，触发限流
(sentinel+gateway整合case)");

            return ServerResponse.status(HttpStatus.TOO_MANY_REQUESTS)
                .contentType(MediaType.APPLICATION_JSON)
                .body(BodyInserters.fromValue(map));
        }
    };
    GatewayCallbackManager.setBlockHandler(handler);
}
}

```

8. 测试: <http://localhost:9528/pay/nacos/123>, <http://localhost:9001/pay/nacos/123>

# Seata(处理分布式事务)

## 简介

Simple Extensible Autonomous Transaction Architecture (简单可扩展自治事务框架)

Apache Seata(incubating) 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。

<https://seata.apache.org/zh-cn/>

<https://github.com/apache/incubator-seata>

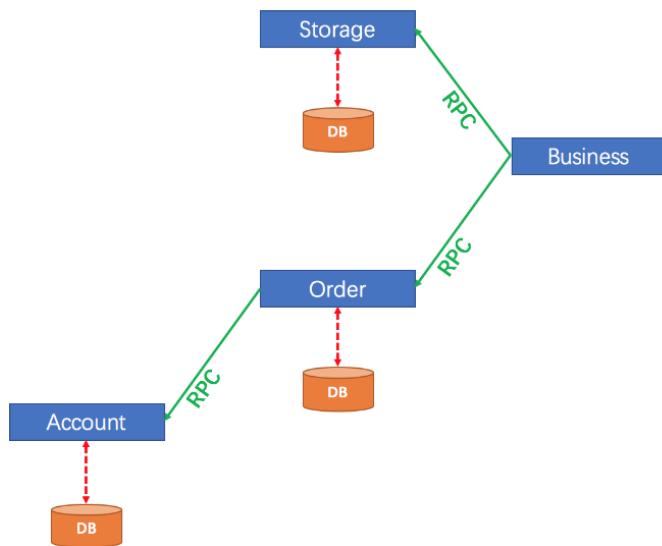
- 分布式事务问题如何产生

- 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，但是关系型数据库提供的能力是基于单机事务的，一旦遇到分布式事务场景，就需要通过更多其他技术手段来解决问题。
- 分布式事务before：单机单库没有问题，表结构1:1, 1:N, N:N
- 分布式事务after：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。此时每个服务自己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。

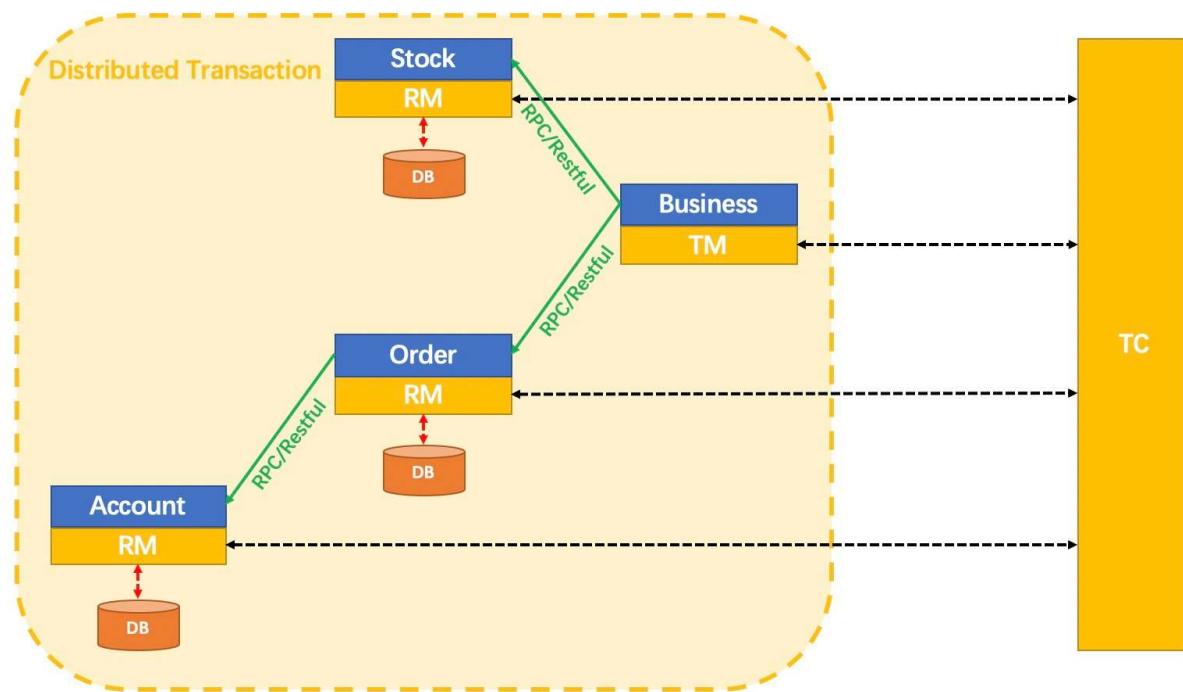
### 架构图



- 发展历程

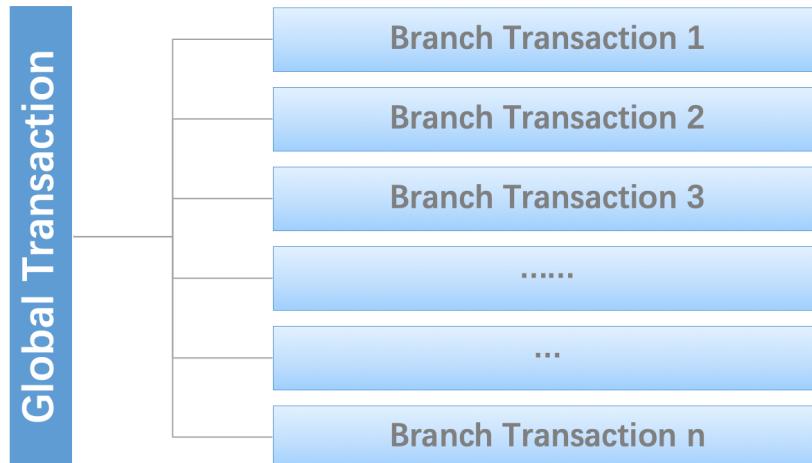
- 阿里巴巴作为国内最早一批进行应用分布式（微服务化）改造的企业，很早就遇到微服务架构下的分布式事务问题。
- 2019年1月份蚂蚁金服和阿里巴巴共同开源的分布式事务解决方案：
- 2014 年，阿里中间件团队发布 TXC (Taobao Transaction Constructor)，为集团内应用提供分布式事务服务。
- 2016 年，TXC 在经过产品化改造后，以 GTS (Global Transaction Service) 的身份登陆阿里云，成为当时业界唯一一款云上分布式事务产品。在阿云里的公有云、专有云解决方案中，开始服务于众多外部客户。
- 2019 年起，基于 TXC 和 GTS 的技术积累，阿里中间件团队发起了开源项目 Fescar (Fast & Easy Commit And Rollback, FESCAR)，和社区一起建设这个分布式事务解决方案。

- 2019年fescar(全称fast easy commit and rollback)被重命名为了seata (simple extensible autonomous transaction architecture)。TXC、GTS、Fescar以及seata一脉相承，为解决微服务架构下的分布式事务问题交出了一份与众不同的答卷。



## 工作流程简介

纵观整个分布式事务的管理，就是全局事务ID的传递和变更，需要让开发者无感知  
分布式事务是由批量分支事务组成的全局事务，通常分支事务只是本地事务。



Seata对分布式事务的协调和控制就是  $1 + 3$

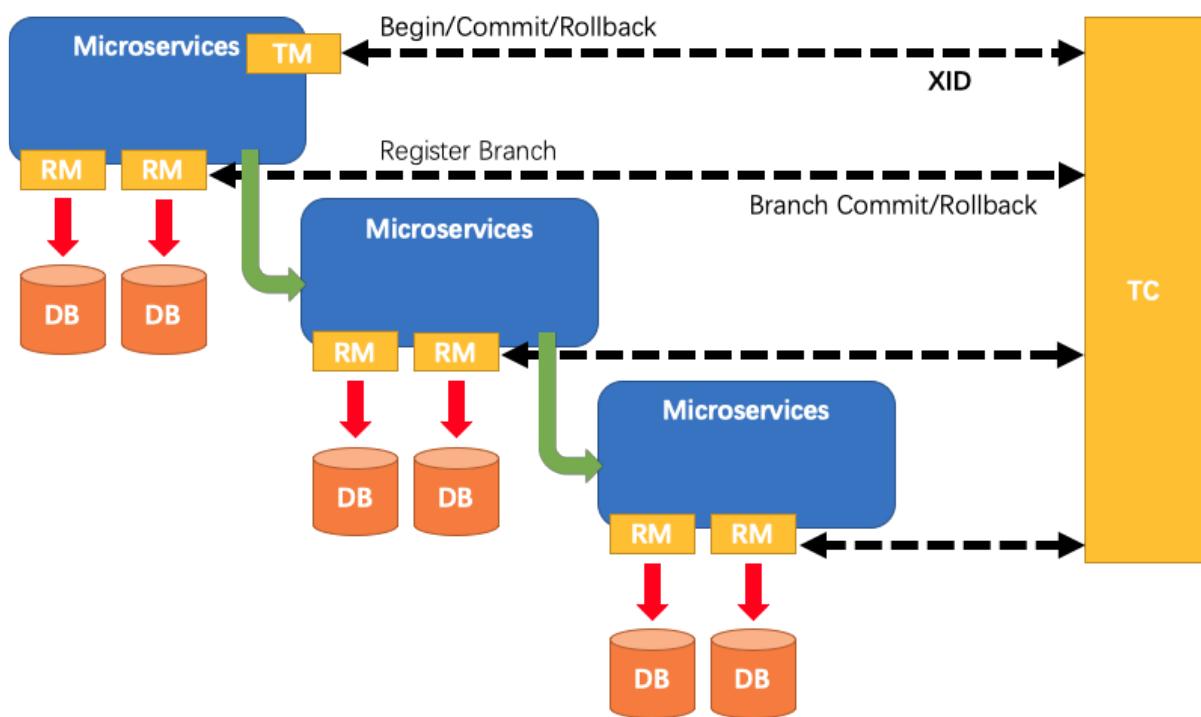
- 1个XID：XID是全局事务的唯一标识，它可以在服务的调用链路中传递，绑定到服务的事务上下文中
- TC：事务协调者，维护全局和分支事务的状态，驱动全局事务提交或回滚（就是Seata）
- TM：事务管理器，定义全局事务的范围：开始全局事务、提交或回滚全局事务（标注全局 `@GlobalTransactional` 启动入口动作的微服务模块，事务的发起者）
- RM：资源管理器，管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚（数据库本身，可以是多个RM）

三个组件相互协作，TC以Seata服务器(Server)形式独立部署，TM和RM则是以Seata Client的形式集成在微服务中运行

### Seata托管分布式事务的典型生命周期：

1. TM要求TC开始一个新的全局事务。TC生成一个代表全局事务的XID。
2. XID通过微服务的链调起。
3. RM将本地事务注册为XID到TC的相应全局事务的分支。

4. TM要求TC提交或回滚XID的相应全局事务。
5. TC驱动XID对应全局事务下的所有分支事务，完成分支提交或回滚。



各事务模式：[https://seata.apache.org/zh-cn/docs/user mode/at](https://seata.apache.org/zh-cn/docs/user	mode/at)

1. AT模式
2. TCC模式
3. Saga模式
4. XA模式

## 安装配置

1. 下载：<https://github.com/apache/incubator-seata/releases/tag/v2.0.0>

参数配置：<https://seata.apache.org/zh-cn/docs/user/configurations>

2. Setta需要进行数据的存储，在mysql数据库中进行建库 + 建表

<https://github.com/apache/incubator-seata/blob/2.x/script/server/db/mysql.sql>

SQL脚本内容：

```

CREATE DATABASE seata;
USE seata;
-- ----- The script used when storeMode is 'db' -----
-- the table to store GlobalSession data
CREATE TABLE IF NOT EXISTS `global_table`(
  `xid`          VARCHAR(128) NOT NULL,
  `transaction_id`    BIGINT,
  `status`        TINYINT      NOT NULL,
  `application_id`   VARCHAR(32),
  `transaction_service_group` VARCHAR(32),
  `transaction_name`  VARCHAR(128),
  `timeout`       INT,
  `begin_time`    BIGINT,
  `application_data` VARCHAR(2000),
  `gmt_create`    DATETIME,
  `gmt_modified`  DATETIME,
  PRIMARY KEY (`xid`),
  KEY `idx_status_gmt_modified` (`status`, `gmt_modified`),
);

```

```

        KEY `idx_transaction_id` (`transaction_id`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4;

-- the table to store BranchSession data
CREATE TABLE IF NOT EXISTS `branch_table`
(
    `branch_id`          BIGINT      NOT NULL,
    `xid`                VARCHAR(128) NOT NULL,
    `transaction_id`    BIGINT,
    `resource_group_id` VARCHAR(32),
    `resource_id`        VARCHAR(256),
    `branch_type`        VARCHAR(8),
    `status`              TINYINT,
    `client_id`          VARCHAR(64),
    `application_data`  VARCHAR(2000),
    `gmt_create`         DATETIME(6),
    `gmt_modified`       DATETIME(6),
    PRIMARY KEY (`branch_id`),
    KEY `idx_xid` (`xid`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4;

-- the table to store lock data
CREATE TABLE IF NOT EXISTS `lock_table`
(
    `row_key`            VARCHAR(128) NOT NULL,
    `xid`                VARCHAR(128),
    `transaction_id`    BIGINT,
    `branch_id`          BIGINT      NOT NULL,
    `resource_id`        VARCHAR(256),
    `table_name`         VARCHAR(32),
    `pk`                 VARCHAR(36),
    `status`              TINYINT      NOT NULL DEFAULT '0' COMMENT '0:locked
,1:rollbacking',
    `gmt_create`         DATETIME,
    `gmt_modified`       DATETIME,
    PRIMARY KEY (`row_key`),
    KEY `idx_status`   (`status`),
    KEY `idx_branch_id`(`branch_id`),
    KEY `idx_xid`   (`xid`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4;

CREATE TABLE IF NOT EXISTS `distributed_lock`
(
    `lock_key`          CHAR(20) NOT NULL,
    `lock_value`        VARCHAR(20) NOT NULL,
    `expire`             BIGINT,
    primary key (`lock_key`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4;

INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('AsyncCommitting', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('RetryCommitting', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('RetryRollbacking', ' ', 0);
INSERT INTO `distributed_lock` (lock_key, lock_value, expire) VALUES
('TxTimeoutCheck', ' ', 0);

```

3. 对seata的配置进行更改，`seata\conf\application.yml`，备份

对application.yml进行修改，参考application.example.yml进行修改，配置数据库和服务注册中心

```
# Copyright 1999-2019 Seata.io Group.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

server:
  port: 7091

spring:
  application:
    name: seata-server

logging:
  config: classpath:logback-spring.xml
  file:
    path: ${log.home}:${user.home}/logs/seata
  extend:
    logstash-appender:
      destination: 127.0.0.1:4560
    kafka-appender:
      bootstrap-servers: 127.0.0.1:9092
      topic: logback_to_logstash

console:
  user:
    username: seata
    password: seata

seata:
  config:
    # support: nacos, consul, apollo, zk, etcd3
    type: nacos
    nacos:
      server-addr: 127.0.0.1:8848
      namespace:
        #后续自己在nacos里面新建，不想新建SEATA_GROUP，就写DEFAULT_GROUP
        group: SEATA_GROUP
        username: nacos
        password: nacos
  registry:
    # support: nacos, eureka, redis, zk, consul, etcd3, sofa
    type: nacos
    nacos:
      application: seata-server
      server-addr: 127.0.0.1:8848
      group: SEATA_GROUP
      username: nacos
      password: nacos
```

```

store:
# support: file 、 db 、 redis 、 raft
mode: db
db:
  datasource: druid
  db-type: mysql
  driver-class-name: com.mysql.cj.jdbc.Driver
  url: mysql://localhost:3306/seata?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
  user: root
  password: 123456
  min-conn: 10
  max-conn: 100
  global-table: global_table
  branch-table: branch_table
  lock-table: local_table
  distributed-lock-table: distributed_lock
  query-limit: 1000
  max-wait: 5000
# server:
# service-port: 8091 #If not configured, the default is '${server.port} + 1000'
security:
  secretKey: SeataSecretKey0c382ef121d778043159209298fd40bf3850a017
  tokenValidityInMilliseconds: 1800000
  ignore:
    urls:
    //,/*/*.css,/*/*.js,/*/*.html,/*/*.map,/*/*.svg,/*/*.png,/*/*.jpeg,/*/*.ico,/api/v1/auth/login,/metadata/v1/*

```

4. 启动nacos 8848

5. 启动seata, 在bin目录下执行 **seata-server.bat**, 查看localhost:7091

## 案例

### 数据库准备

准备订单 + 库存 + 账户 三个业务数据库Mysql

1. 启动Nacos和Seata
2. 这里我们创建三个服务，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 → 减库存 → 扣余额 → 改(订单)状态

3. 创建3个对应的业务数据库

```
#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
```

```
CREATE DATABASE seata_order;  
CREATE DATABASE seata_storage;  
CREATE DATABASE seata_account;
```

#### 4. 分别为上述3个数据库建立对应的 `undo_log` 回滚日志表

订单、库存、账户信息都需要建各自的 `undo_log` 回滚日志表 (AT模式专用, 其他模式不需要)

<https://github.com/apache/incubator-seata/blob/2.x/script/client/at/db/mysql.sql>

```
CREATE TABLE IF NOT EXISTS `undo_log`  
(  
    `branch_id`      BIGINT      NOT NULL COMMENT 'branch transaction id',  
    `xid`            VARCHAR(128) NOT NULL COMMENT 'global transaction id',  
    `context`        VARCHAR(128) NOT NULL COMMENT 'undo_log context,such as  
serialization',  
    `rollback_info`  LONGBLOB    NOT NULL COMMENT 'rollback info',  
    `log_status`     INT(11)     NOT NULL COMMENT '0:normal status,1:defense  
status',  
    `log_created`    DATETIME(6)  NOT NULL COMMENT 'create datetime',  
    `log_modified`   DATETIME(6)  NOT NULL COMMENT 'modify datetime',  
    UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)  
) ENGINE = InnoDB AUTO_INCREMENT = 1 DEFAULT CHARSET = utf8mb4 COMMENT ='AT  
transaction mode undo table';  
ALTER TABLE `undo_log` ADD INDEX `ix_log_created`(`log_created`);
```

#### 5. 给上述3库分别创建对应业务表

t\_order

```
CREATE TABLE t_order(  
    `id`  BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    `user_id`  BIGINT(11) DEFAULT NULL COMMENT '用户id',  
    `product_id`  BIGINT(11) DEFAULT NULL COMMENT '产品id',  
    `count`  INT(11) DEFAULT NULL COMMENT '数量',  
    `money`  DECIMAL(11,0) DEFAULT NULL COMMENT '金额',  
    `status`  INT(1) DEFAULT NULL COMMENT '订单状态: 0:创建中; 1:已完结'  
)ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;  
SELECT * FROM t_order;
```

t\_account

```
CREATE TABLE t_account(  
    `id`  BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY COMMENT 'id',  
    `user_id`  BIGINT(11) DEFAULT NULL COMMENT '用户id',  
    `total`  DECIMAL(10,0) DEFAULT NULL COMMENT '总额度',  
    `used`  DECIMAL(10,0) DEFAULT NULL COMMENT '已用账户余额',  
    `residue`  DECIMAL(10,0) DEFAULT '0' COMMENT '剩余可用额度'  
)ENGINE=INNODB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;  
INSERT INTO  
t_account(`id`, `user_id`, `total`, `used`, `residue`)VALUES('1', '1', '1000', '0', '10  
00');  
SELECT * FROM t_account;
```

t\_storage

```

CREATE TABLE t_storage(
`id` BIGINT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
`product_id` BIGINT(11) DEFAULT NULL COMMENT '产品id',
`total` INT(11) DEFAULT NULL COMMENT '总库存',
`used` INT(11) DEFAULT NULL COMMENT '已用库存',
`residue` INT(11) DEFAULT NULL COMMENT '剩余库存'
)ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
INSERT INTO
t_storage(`id`, `product_id`, `total`, `used`, `residue`)VALUES('1', '1', '100', '0', '100');
SELECT * FROM t_storage;

```

## 微服务编码

### Mybatis一键生成

1. 修改模块 `mybatis-generator`
2. 修改 `config.properties`

通过修改连接配置，进行生成对应的数据库文件

```

#t_pay表包名
package.name=com.xxx.cloud

# mysql8.0
jdbc.driverClass = com.mysql.cj.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/db2024?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
jdbc.user = root
jdbc.password =123456

# seata_order
#jdbc.driverClass = com.mysql.cj.jdbc.Driver
#jdbc.url = jdbc:mysql://localhost:3306/seata_order?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
#jdbc.user = root
#jdbc.password =123456

# seata_storage
#jdbc.driverClass = com.mysql.cj.jdbc.Driver
#jdbc.url = jdbc:mysql://localhost:3306/seata_storage?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
#jdbc.user = root
#jdbc.password =123456

# seata_account
#jdbc.driverClass = com.mysql.cj.jdbc.Driver
#jdbc.url = jdbc:mysql://localhost:3306/seata_account?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
#jdbc.user = root
#jdbc.password =123456

```

3. 修改 `generatorConfig.xml`

通过修改 `generatorConfig` 来配置文件生成位置以及信息

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
    <properties resource="config.properties"/>
    <context id="Mysql" targetRuntime="MyBatis3Simple" defaultModelType="flat">
        <property name="beginningDelimiter" value="`"/>
        <property name="endingDelimiter" value="`"/>

        <plugin type="tk.mybatis.mapper.generator.MapperPlugin">
            <property name="mappers" value="tk.mybatis.mapper.common.Mapper"/>
            <property name="caseSensitive" value="true"/>
        </plugin>

        <jdbcConnection driverClass="${jdbc.driverClass}"
                        connectionURL="${jdbc.url}"
                        userId="${jdbc.user}"
                        password="${jdbc.password}"/>
    </jdbcConnection>

    <javaModelGenerator targetPackage="${package.name}.entities"
targetProject="src/main/java"/>

        <sqlMapGenerator targetPackage="${package.name}.mapper"
targetProject="src/main/java"/>

        <javaClientGenerator targetPackage="${package.name}.mapper"
                            targetProject="src/main/java" type="XMLMAPPER"/>

        <table tableName="t_pay" domainObjectName="Pay">
            <generatedKey column="id" sqlStatement="JDBC"/>
        </table>

        <!-- seata_order -->
        <!--<table tableName="t_order" domainObjectName="Order">
            <generatedKey column="id" sqlStatement="JDBC"/>
        </table>-->

        <!--seata_storage-->
        <!--<table tableName="t_storage" domainObjectName="Storage">
            <generatedKey column="id" sqlStatement="JDBC"/>
        </table>-->

        <!--seata_account-->
        <!--<table tableName="t_account" domainObjectName="Account">
            <generatedKey column="id" sqlStatement="JDBC"/>
        </table>-->
    </context>
</generatorConfiguration>

```

## 修改公共模块

1. 修改 **cloud-api-commons** 新增库存和账户两个Feign服务接口
2. **StorageFeignApi** 接口

```

@FeignClient("seata-storage-service")
public interface StorageFeignApi {
    // 扣减库存
    @PostMapping(value = "/storage/decrease")
    ResultData decrease(@RequestParam("productId") Long productId,
    @RequestParam("count") Integer count);
}

```

### 3. AccountFeignApi 接口

```

@FeignClient("seata-account-service")
public interface AccountFeignApi {
    // 扣减账户余额
    @PostMapping("/account/decrease")
    ResultData decrease(@RequestParam("userId") Long userId,
    @RequestParam("money") Long money);
}

```

## 新建订单Order模块(2001)

1. 新建Module模块 **seata-order-service2001**

2. POM

```

<dependencies>
    <!-- nacos -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--alibaba-seata-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    </dependency>
    <!--openfeign-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <!--loadbalancer-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
    <!--cloud-api-commons-->
    <dependency>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--web + actuator-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

```

```
<!--SpringBoot集成druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
</dependency>
<!-- Swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
<!--mybatis和springboot整合-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
</dependency>
<!--Mysql数据库驱动8 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!--persistence-->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
</dependency>
<!--通用Mapper4-->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!-- fastjson2 -->
<dependency>
    <groupId>com.alibaba.fastjson2</groupId>
    <artifactId>fastjson2</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```
</plugins>
</build>
```

### 3. 全局配置

```
server:
  port: 2001

spring:
  application:
    name: seata-order-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848          #Nacos服务注册中心地址
# =====applicationName + druid-mysql8 driver=====
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/seata_order?
    characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
    username: root
    password: 123456
# =====mybatis=====
  mybatis:
    mapper-locations: classpath:mapper/*.xml
    type-aliases-package: com.xxx.cloud.entities
    configuration:
      map-underscore-to-camel-case: true

# =====seata=====
  seata:
    registry:
      type: nacos
      nacos:
        server-addr: 127.0.0.1:8848
        namespace: ""
        group: SEATA_GROUP
        application: seata-server
    tx-service-group: default_tx_group # 事务组，由它获得TC服务的集群名称
    service:
      vgroup-mapping: # 点击源码分析
        default_tx_group: default # 事务组与TC服务集群的映射关系
      data-source-proxy-mode: AT

    logging:
      level:
        io:
          seata: info
```

### 4. 主启动类

```

@SpringBootApplication
//import tk.mybatis.spring.annotation.MapperScan;
@MapperScan("com.xxx.cloud.mapper")
@EnableDiscoveryClient //服务注册和发现
@EnableFeignClients
public class SeataOrderMainApp2001{
    public static void main(String[] args){
        SpringApplication.run(SeataOrderMainApp2001.class,args);
    }
}

```

## 5. 业务类

### entities

将刚才mybatis一键生成的entities粘贴过来

```

@Table(name = "t_order")
@ToString
public class Order implements Serializable{
    @Id
    @GeneratedValue(generator = "JDBC")
    private Long id;
    //用户id
    @Column(name = "user_id")
    private Long userId;
    // 产品id
    @Column(name = "product_id")
    private Long productId;
}

```

### OrderMapper接口

将刚才mybatis一键生成的mapper粘贴过来

```
public interface OrderMapper extends Mapper<Order> {}
```

在resource文件夹下新建mapper文件夹后添加OrderMapper.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxx.cloud.mapper.OrderMapper">
    <resultMap id="BaseResultMap" type="com.xxx.cloud.entities.Order">
        <!--
            WARNING - @mbg.generated
        -->
        <id column="id" jdbcType="BIGINT" property="id" />
        <result column="user_id" jdbcType="BIGINT" property="userId" />
        <result column="product_id" jdbcType="BIGINT" property="productId" />
        <result column="count" jdbcType="INTEGER" property="count" />
        <result column="money" jdbcType="DECIMAL" property="money" />
        <result column="status" jdbcType="INTEGER" property="status" />
    </resultMap>
</mapper>

```

### Service接口实现

OrderService

```
public interface OrderService {
    // 创建订单
    void create(Order order);
}
```

### OrderServiceImpl

```
@Slf4j
@Service
public class OrderServiceImpl implements OrderService{
    @Resource
    private OrderMapper orderMapper;
    @Resource//订单微服务通过OpenFeign去调用库存微服务
    private StorageFeignApi storageFeignApi;
    @Resource//订单微服务通过OpenFeign去调用账户微服务
    private AccountFeignApi accountFeignApi;

    @Override
    @GlobalTransactional(name = "zzyy-create-order",
        rollbackFor = Exception.class)
    // AT XA
    // @GlobalTransactional
    // @Transactional(rollbackFor = Exception.class)
    @GlobalTransactional(name="example-create-order", rollback=Exception.class)
    public void create(Order order) {
        //xid检查
        String xid = RootContext.getXID();
        //1. 新建订单
        log.info("=====开始新建订单"+ "\t" +"xid_order:" +xid);
        //订单状态status: 0: 创建中; 1: 已完结
        order.setStatus(0);
        int result = orderMapper.insertSelective(order);
        //插入订单成功后获得插入mysql的实体对象
        Order orderFromDB = null;
        if(result > 0){
            orderFromDB = orderMapper.selectOne(order);
            //orderFromDB = orderMapper.selectByPrimaryKey(order.getId());
            log.info("-----> 新建订单成功, orderFromDB info: "+orderFromDB);
            System.out.println();
            //2. 扣减库存
            log.info("-----> 订单微服务开始调用Storage库存, 做扣减count");
            storageFeignApi.decrease(orderFromDB.getProductId(),
                orderFromDB.getCount());
            log.info("-----> 订单微服务结束调用Storage库存, 做扣减完成");
            System.out.println();
            //3. 扣减账号余额
            log.info("-----> 订单微服务开始调用Account账号, 做扣减money");
            accountFeignApi.decrease(orderFromDB.getUserId(),
                orderFromDB.getMoney());
            log.info("-----> 订单微服务结束调用Account账号, 做扣减完成");
            System.out.println();
            //4. 修改订单状态
            //订单状态status: 0: 创建中; 1: 已完结
            log.info("-----> 修改订单状态");
            orderFromDB.setStatus(1);

            Example whereCondition=new Example(Order.class);
            Example.Criteria criteria=whereCondition.createCriteria();
            criteria.andEqualTo("userId",orderFromDB.getUserId());
            criteria.andEqualTo("status",0);
```

```

        int updateResult =
orderMapper.updateByExampleSelective(orderFromDB, whereCondition);

        log.info("-----> 修改订单状态完成"+"\t"+updateResult);
        log.info("-----> orderFromDB info: "+orderFromDB);
    }
    System.out.println();
    log.info("=====结束新建订单"+"\t"+"xid_order:" +xid);
}
}

```

## Controller

```

@RestController
public class OrderController{
    @Resource
    private OrderService orderService;
    // 创建订单
    @GetMapping("/order/create")
    public ResultData create(Order order){
        orderService.create(order);
        return ResultData.success(order);
    }
}

```

## 新建库存Storage模块(2002)

1. 新建库存Storage模块 **seata-storage-service2002**
2. POM

```

<dependencies>
    <!-- nacos -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--alibaba-seata-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    </dependency>
    <!--openfeign-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <!--loadbalancer-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
    <!--cloud_commons_utils-->
    <dependency>
        <groupId>com.xxx.cloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!--web + actuator-->

```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--SpringBoot集成druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
</dependency>
<!-- Swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
<!--mybatis和springboot整合-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
</dependency>
<!--Mysql数据库驱动8 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!--persistence-->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
</dependency>
<!--通用Mapper4-->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
</dependency>
<!-- fastjson2 -->
<dependency>
    <groupId>com.alibaba.fastjson2</groupId>
    <artifactId>fastjson2</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

```

</dependencies>

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

```

### 3. 全局配置

```

server:
    port: 2002

spring:
    application:
        name: seata-storage-service
    cloud:
        nacos:
            discovery:
                server-addr: localhost:8848          #Nacos服务注册中心地址
# =====applicationName + druid-mysql8 driver=====
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.cj.jdbc.Driver
        url: jdbc:mysql://localhost:3306/seata_storage?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
        username: root
        password: 123456
# =====mybatis=====
mybatis:
    mapper-locations: classpath:mapper/*.xml
    type-aliases-package: com.xxx.cloud.entities
    configuration:
        map-underscore-to-camel-case: true
# =====seata=====
seata:
    registry:
        type: nacos
        nacos:
            server-addr: 127.0.0.1:8848
            namespace: ""
            group: SEATA_GROUP
            application: seata-server
            tx-service-group: default_tx_group # 事务组，由它获得TC服务的集群名称
    service:
        vgroup-mapping:
            default_tx_group: default # 事务组与TC服务集群的映射关系
        data-source-proxy-mode: AT

    logging:
        level:
            io:
                seata: info

```

### 4. 主启动类

```

@SpringBootApplication
//import tk.mybatis.spring.annotation.MapperScan;
@MapperScan("com.xxx.cloud.mapper")
@EnableDiscoveryClient //服务注册和发现
@EnableFeignClients
public class SeataStorageMainApp2002{
    public static void main(String[] args){
        SpringApplication.run(SeataStorageMainApp2002.class,args);
    }
}

```

## 5. 业务类

### entities

使用mybatis\_generate模块插件一键生成，将一键生成的entities粘贴过来

```

@Table(name = "t_storage")
public class Storage implements Serializable{
    @Id
    @GeneratedValue(generator = "JDBC")
    private Long id;
    // 产品id
    @Column(name = "product_id")
    private Long productId;
    // 总库存
    private Integer total;
    // 已用库存
    private Integer used;
    // 剩余库存
    private Integer residue;
}

```

### StorageMapper

使用mybatis\_generate模块插件一键生成，将一键生成的mapper粘贴过来

**StorageMapper** 接口

```

public interface StorageMapper extends Mapper<Storage>{
    // 扣减库存
    void decrease(@Param("productId") Long productId, @Param("count") Integer count);
}

```

在resource文件夹下新建mapper文件夹后添加StorageMapper.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxx.cloud.mapper.StorageMapper">
    <resultMap id="BaseResultMap" type="com.xxx.cloud.entities.Storage">
        <!--
            WARNING - @mbg.generated
        -->
        <id column="id" jdbcType="BIGINT" property="id" />
        <result column="product_id" jdbcType="BIGINT" property="productId" />
        <result column="total" jdbcType="INTEGER" property="total" />
        <result column="used" jdbcType="INTEGER" property="used" />
        <result column="residue" jdbcType="INTEGER" property="residue" />
    </resultMap>
    <update id="decrease">
        UPDATE
        t_storage

```

```

        SET
        used = used + #{count},
        residue = residue - #{count}
    WHERE product_id = #{productId}
</update>
</mapper>

```

## Service接口及实现

### StorageService

```

public interface StorageService {
    // 扣减库存
    void decrease(Long productId, Integer count);
}

```

### StorageServiceImpl

```

@Service
@Slf4j
public class StorageServiceImpl implements StorageService{
    @Resource
    private StorageMapper storageMapper;
    // 扣减库存
    @Override
    public void decrease(Long productId, Integer count) {
        log.info("----->storage-service中扣减库存开始");
        storageMapper.decrease(productId, count);
        log.info("----->storage-service中扣减库存结束");
    }
}

```

### Controller

```

@RestController
public class StorageController{
    @Resource
    private StorageService storageService;
    // 扣减库存
    @RequestMapping("/storage/decrease")
    public ResultData decrease(Long productId, Integer count) {
        storageService.decrease(productId, count);
        return ResultData.success("扣减库存成功!");
    }
}

```

## 新建账户Account模块(2003)

1. 新建库存Storage模块 [seata-account-service2003](#)

2. POM

```

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

```

```
<artifactId>spring-cloud-starter-alibaba-seata</artifactId>
</dependency>
<!--openfeign-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<!--loadbalancer-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
<!--cloud_commons_utils-->
<dependency>
    <groupId>com.xxxx.cloud</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<!--web + actuator-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--SpringBoot集成druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
</dependency>
<!-- Swagger3 调用方式 http://你的主机IP地址:5555/swagger-ui/index.html -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
</dependency>
<!--mybatis和springboot整合-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
</dependency>
<!--Mysql数据库驱动8 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!--persistence-->
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
</dependency>
<!--通用Mapper4-->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
</dependency>
<!--hutool-->
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
```

```

</dependency>
<!-- fastjson2 -->
<dependency>
    <groupId>com.alibaba.fastjson2</groupId>
    <artifactId>fastjson2</artifactId>
</dependency>
<!--lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <scope>provided</scope>
</dependency>
<!--test-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```

### 3. 全局配置

```

server:
  port: 2003

spring:
  application:
    name: seata-account-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848          #Nacos服务注册中心地址
# =====applicationName + druid-mysql8 driver=====
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/seata_account?
    characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&rewriteBatchedStatements=true&allowPublicKeyRetrieval=true
    username: root
    password: 123456
# =====mybatis=====
  mybatis:
    mapper-locations: classpath:mapper/*.xml
    type-aliases-package: com.xxx.cloud.entities
    configuration:
      map-underscore-to-camel-case: true
# =====seata=====
  seata:
    registry:
      type: nacos

```

```
nacos:
  server-addr: 127.0.0.1:8848
  namespace: ""
  group: SEATA_GROUP
  application: seata-server
tx-service-group: default_tx_group # 事务组，由它获得TC服务的集群名称
service:
  vgroup-mapping:
    default_tx_group: default # 事务组与TC服务集群的映射关系
  data-source-proxy-mode: AT

logging:
  level:
    io:
      seata: info
```

#### 4. 主启动类

```
@EnableDiscoveryClient
@EnableFeignClients
//import tk.mybatis.spring.annotation.MapperScan;
@MapperScan("com.xxx.cloud.mapper")
@SpringBootApplication
public class SeataAccountMainApp2003{
    public static void main(String[] args){
        SpringApplication.run(SeataAccountMainApp2003.class,args);
    }
}
```

#### 5. 业务类

##### entities

使用mybatis\_generate模块插件一键生成，将一键生成的entities粘贴过来

```
@Table(name = "t_account")
public class Account implements Serializable{
    @Id
    @GeneratedValue(generator = "JDBC")
    private Long id;
    // 产品id
    @Column(name = "user_id")
    private Long userId;
    // 总额度
    private Integer total;
    // 已用余额
    private Integer used;
    // 剩余可用额度
    private Integer residue;
}
```

##### AccountMapper

使用mybatis\_generate模块插件一键生成，将一键生成的mapper粘贴过来

AccountMapper 接口

```
public interface AccountMapper extends Mapper<Account>{
    // 本次消费金额
    void decrease(@Param("userId") Long userId, @Param("money") Long money);
}
```

在resource文件夹下新建mapper文件夹后添加AccountMapper.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxx.cloud.mapper.AccountMapper">
    <resultMap id="BaseResultMap" type="com.xxx.cloud.entities.Account">
        <!--
            WARNING - @mbg.generated
        -->
        <id column="id" jdbcType="BIGINT" property="id" />
        <result column="user_id" jdbcType="BIGINT" property="userId" />
        <result column="total" jdbcType="DECIMAL" property="total" />
        <result column="used" jdbcType="DECIMAL" property="used" />
        <result column="residue" jdbcType="DECIMAL" property="residue" />
    </resultMap>
    <!--
        money 本次消费金额
        t_account数据库表
        total总额度 = 累计已消费金额(used) + 剩余可用额度(residue)
    -->
    <update id="decrease">
        UPDATE
            t_account
        SET
            residue = residue - #{money}, used = used + #{money}
        WHERE user_id = #{userId};
    </update>
</mapper>

```

## Service接口及实现

### StorageService

```

public interface AccountService {
    // 扣减账户余额
    void decrease(@Param("userId") Long userId, @Param("money") Long money);
}

```

### StorageServiceImpl

```

@Service
@Slf4j
public class AccountServiceImpl implements AccountService
{
    @Resource
    AccountMapper accountMapper;

    // 扣减账户余额
    @Override
    public void decrease(Long userId, Long money) {
        log.info("----->account-service中扣减账户余额开始");
        accountMapper.decrease(userId,money);
        //myTimeOut();
        //int age = 10/0;
        log.info("----->account-service中扣减账户余额结束");
    }
    // 模拟超时异常，全局事务回滚
    private static void myTimeOut() {
        try {
            TimeUnit.SECONDS.sleep(65);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```
        }
    }
}
```

## Controller

```
@RestController
public class AccountController {
    @Resource
    AccountService accountService;
    // 扣减账户余额
    @RequestMapping("/account/decrease")
    public ResultData decrease(@RequestParam("userId") Long userId,
                               @RequestParam("money") Long money){
        accountService.decrease(userId,money);
        return ResultData.success("扣减账户余额成功！");
    }
}
```

## 测试

1. 启动nacos、启动seata、启动2001、2002、2003服务
2. 测试地址：<http://localhost:2001/order/create?userId=1&productId=1&count=10&money=100>
3. 添加了注解 `@GlobalTransactional` 若超时或者失败会进行回滚，若没有添加注解 `@GlobalTransactional` 则会回滚失败
4. 执行事务时会在各个数据库中的 `undo_log` 表中记录上一步数据信息，执行成功则删除 `undo_log` 表中的数据信息，执行失败则根据 `undo_log` 表中的数据信息回滚还原上一步的数据信息

## 总结

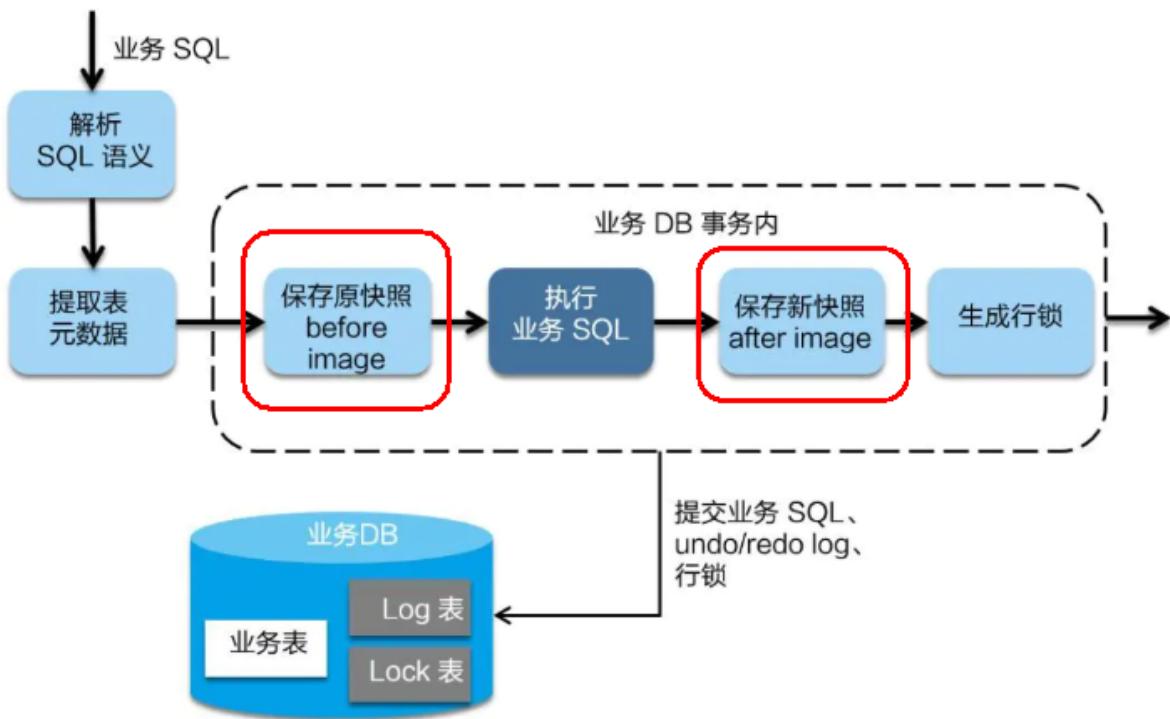
### AT模式如果做到对业务的无侵入

AT模式，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

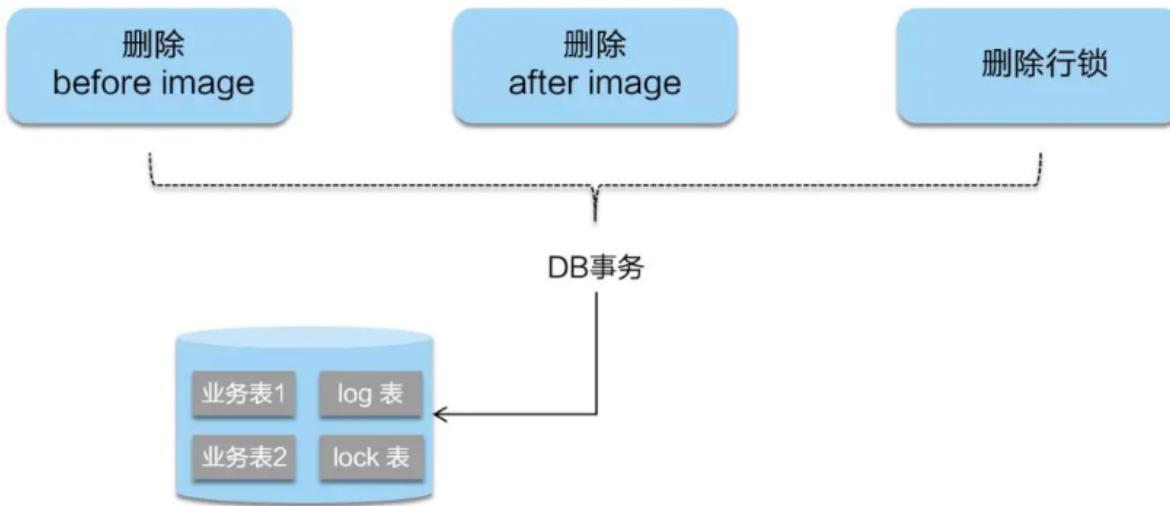
1. 在一阶段，Seata 会拦截“业务 SQL”，
2. 解析 SQL 语义，找到“业务 SQL”要更新的业务数据，在业务数据被更新前，将其保存成“before image”
3. 执行“业务 SQL”更新业务数据，在业务数据更新之后，其保存成“after image”，最后生成行锁
4. 以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



## 二阶段加载

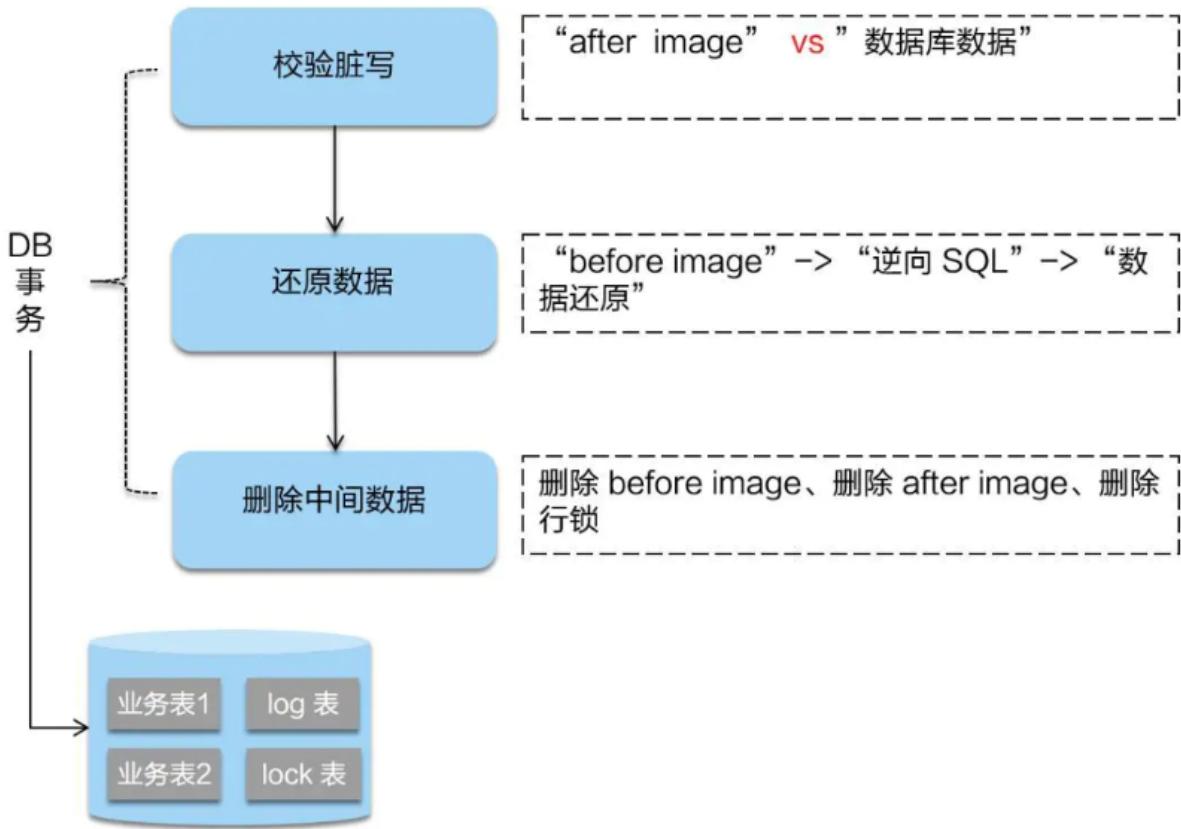
二阶段接分为两种情况：正常提交和异常回滚

1. 正常提交：二阶段如是顺利提交的话，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



2. 异常回滚：二阶段如果是回滚的话，Seata 就需要回滚一阶段已经执行的“业务 SQL”，还原业务数据。

回滚方式便是用“before image”还原业务数据；但在还原前要首先要校验脏写，对比“数据库当前业务数据”和“after image”，如果两份数据完全一致就说明没有脏写，可以还原业务数据，如果不一致就说明有脏写，出现脏写就需要转人工处理。



## 对于分布式事务问题，你知道的解决方案有哪些

1. 2PC: 两阶段提交
2. 3PC: 三阶段提交
3. TCC方案: (Try-Confirm-Cancel) 又被称为补偿方案, 类似2PC的柔性分布式解决方案, 2PC改良版
4. LocalMessage本地消息表
5. 独立消息微服务 + RabbitMQ/Kafka组件, 实现可靠消息最终一致方案
6. 最大努力通知方案