

框架高级课程系列之 SpringCloud

尚硅谷 JavaEE 教研组

版本：V2.5

课程计划

- 微服务、分布式概念、微服务架构
- 注册中心：Eureka
- 负载均衡：Ribbon
- 声明式调用远程方法：OpenFeign
- 熔断、降级、监控：Hystrix
- 网关：Gateway
- 链路跟踪：Sleuth
- 服务注册和配置中心：Spring Cloud Alibaba Nacos
- 熔断、降级、限流：Spring Cloud Alibaba Sentinel

1. 微服务理论

<https://www.martinfowler.com/articles/microservices.html> 微服务 microservices

<http://blog.cuicc.com/blog/2015/07/22/microservices/>

In short (简言之) , the microservice **architectural style** 【架构风格】[1] is an approach to developing **a single application as a suite of small services** 【独立应用变成一套小服务】 , each running in its own process and communicating with lightweight(轻量级沟通) mechanisms(每一个都运行在自己的进程内 (容器)), often an HTTP resource API(用 HTTP, 将功能写成能接受请求). These services are built around business capabilities (独立业务能力) and independently deployable by fully automated deployment machinery (应该自动化独立部署) . There is a bare minimum of centralized management of these services (应该有一个能管理这些服务的中心) , which may be written in different programming languages (独立开发语言)and use different data storage technologies (独立的数据存储) .

2. 分布式概念

2.1. 什么是分布式

《分布式系统原理与范型》定义：“分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像单个相关系统”。

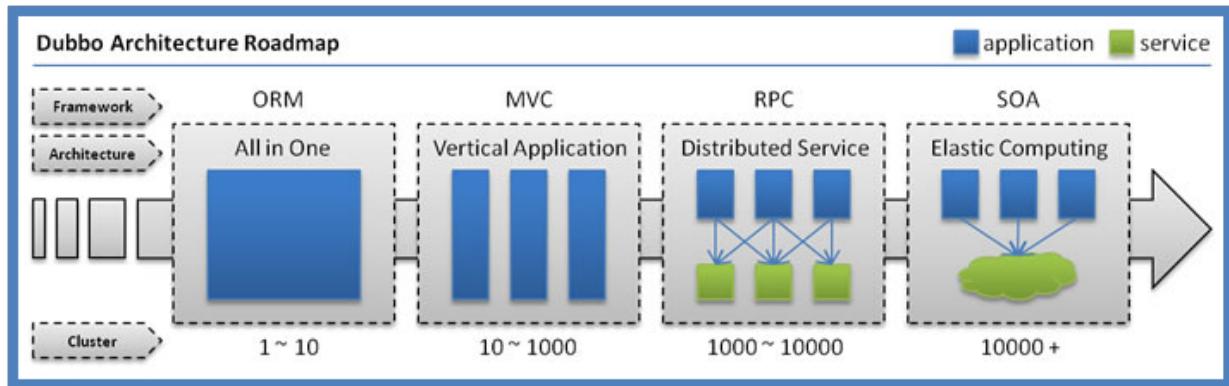
分布式系统 (distributed system) 是建立在网络之上的软件系统。

2.2. 分布式与集群的关系

集群指的是将几台服务器集中在一起，实现同一业务。

分布式中的每一个节点，都可以做集群。而集群并不一定就是分布式的。

2.3. 软件架构演变



单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的**数据访问框架(ORM)**是关键。

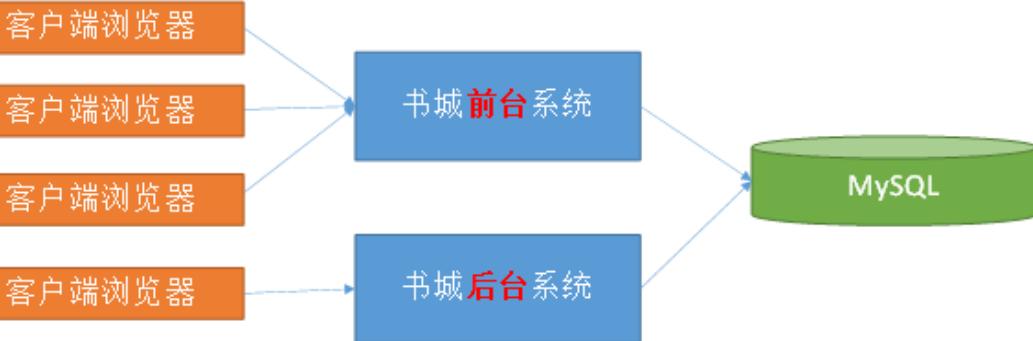
单一应用-All in one



垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的**Web 框架(MVC)**是关键。

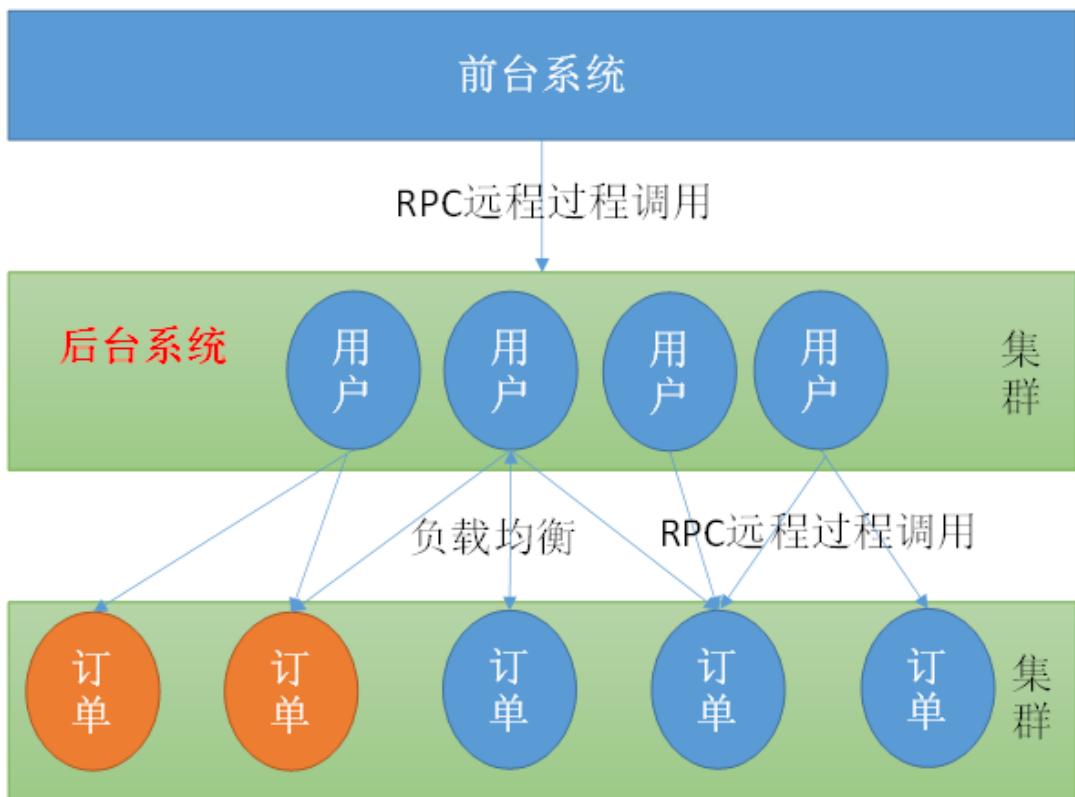
垂直应用架构（按照某种方式将系统拆开，独立运行）



分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，

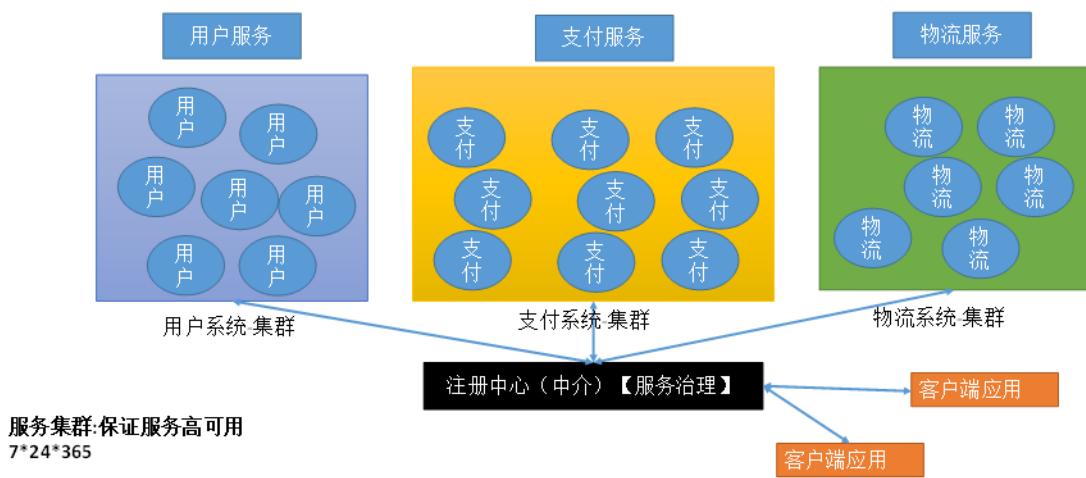
用于提高业务复用及整合的**分布式服务框架(RPC)**是关键。



流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加**一个调度中心**基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的**资源调度和治理中心(SOA)**是关键。

SOA架构（面向服务架构）（流动计算架构）



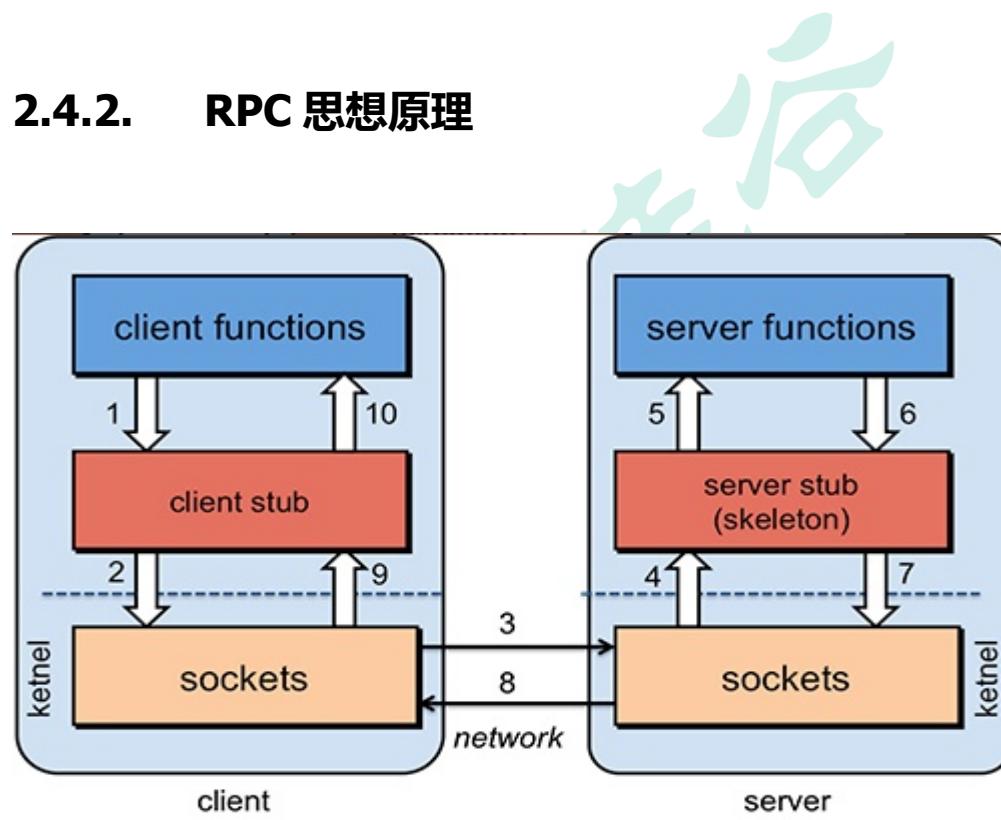
2.4. RPC 是什么

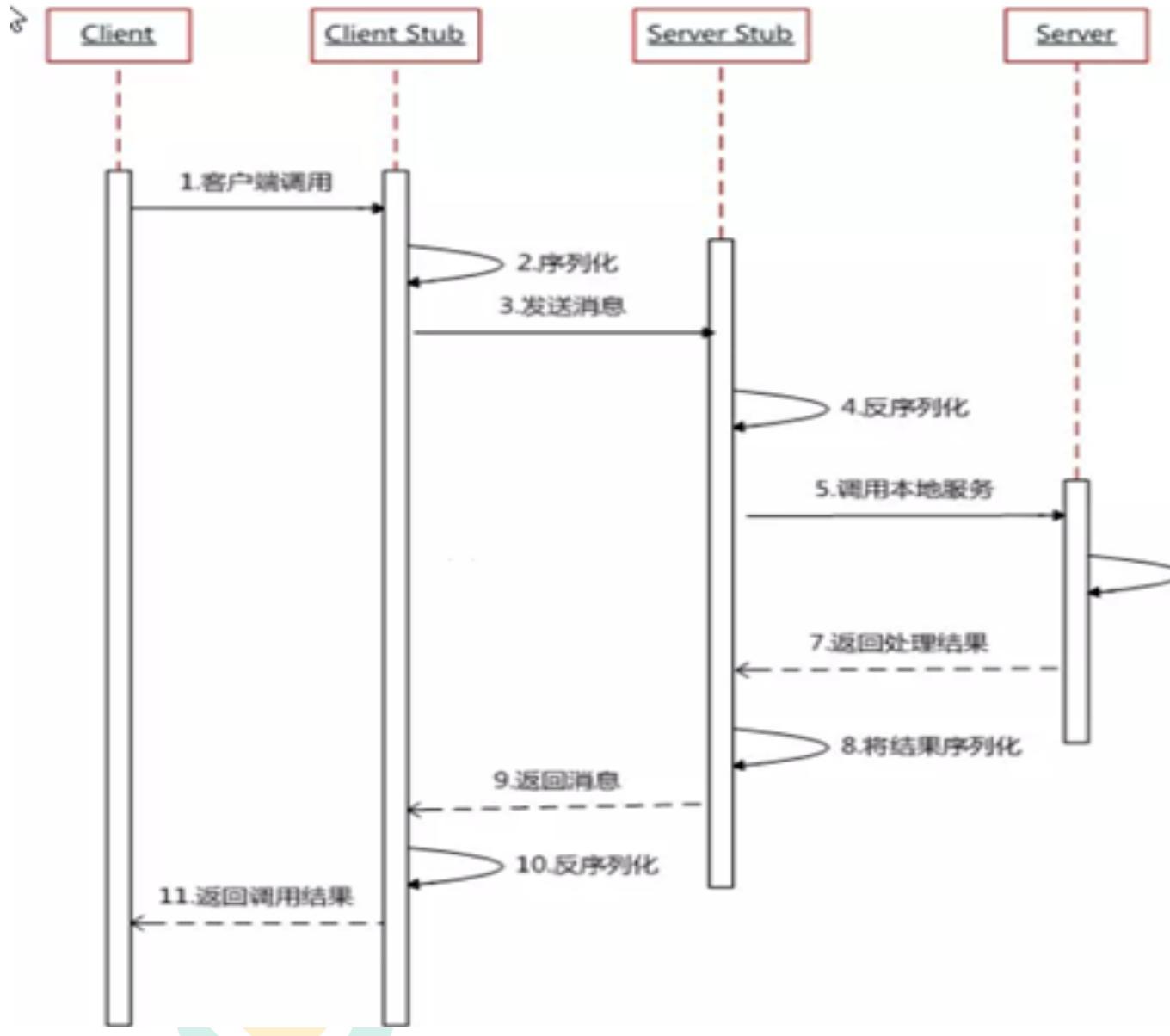
RPC 【Remote Procedure Call】是指远程过程调用，是一种进程间通信方式，他是一种技术的思想，而不是规范。

它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。

即程序员无论是调用本地的还是远程的函数，本质上编写的调用代码基本相同。

2.4.1. 解决分布式系统的各个服务之间互相交互问题





2.4.3. 服务之间的交互可以用两种方式

- RPC
 - ◆ Netty (Socket) + 自定义序列化
- RestAPI (严格来说, SpringCloud 是使用 Rest 方式进行服务之间交互的, 不属于 RPC)
 - ◆ HTTP+JSON

2.5. 分布式思想与基本概念

2.5.1. 高并发

1) 通过设计保证系统可以并行处理很多请求。应对大量流量与请求

- Tomcat 最多支持并发多少用户？

Tomcat 默认配置的最大请求数是 150，也就是说同时支持 150 个并发，当然了，也可以将其改大。

当某个应用拥有 250 个以上并发的时候，应考虑应用服务器的集群。

具体能承载多少并发，需要看硬件的配置，CPU 越多性能越高，分配给 JVM 的内存越多性能也就越高，但也会加重 GC 的负担。

- 操作系统对于进程中的线程数有一定的限制：

Windows 每个进程中的线程数不允许超过 2000

Linux 每个进程中的线程数不允许超过 1000

另外，在 Java 中每开启一个线程需要耗用 1MB 的 JVM 内存空间用于作为线程栈之用。

Tomcat 默认的 HTTP 实现是采用阻塞式的 Socket 通信，每个请求都需要创建一个线程处理。这种模式下的并发量受到线程数的限制，但对于 Tomcat 来说几乎没有 BUG 存在了。

Tomcat 还可以配置 NIO 方式的 Socket 通信，在性能上高于阻塞式的，每个请求也不需要创建一个线程进行处理，并发能力比前者高。但没有阻塞式的成熟。

这个并发能力还与应用的逻辑密切相关，如果逻辑很复杂需要大量的计算，那并发能力势必会下降。如果每个请求都含有很多的数据库操作，那么对于数据库的性能也是非常高的。

对于单台数据库服务器来说，允许客户端的连接数量是有限制的。

并发能力问题涉及整个系统架构和业务逻辑。

系统环境不同，Tomcat 版本不同、JDK 版本不同、以及修改的设定参数不同。并发量的差异还是挺大的。

- ✓ maxThreads="1000" 最大并发数，默认值为 200
- ✓ minSpareThreads="100"//初始化时创建的线程数，默认值为 10
- ✓ acceptCount="700"// 指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理，默认值为 100

<https://tomcat.apache.org/tomcat-8.0-doc/config/http.html>

2) 高并发衡量指标

- 响应时间(RT)
 - 请求做出响应的时间，即一个 http 请求返回所用的时间
- 吞吐量
 - 系统在单位时间内处理请求的数量
- QPS(Query/Request Per Second)、TPS (Transaction Per Second)
- 每秒查询（请求）数、每秒事务数
 - 专业的测试工具：Load Runner
 - Apache ab
 - Apache JMeter
- 并发用户数
 - 承载的正常使用系统功能的用户的数量

2.5.2. 高可用

服务集群部署

数据库主从+双机热备

- 主备方式 (Active-Standby 方式)

主-备方式即指的是一台服务器处于某种业务的激活状态（即 Active 状态），另一台服务器处于该业务的备用状态（即 Standby 状态）。

- 双主机方式 (Active-Active 方式)

双主机方式即指两种不同业务分别在两台服务器上互为主备状态（即 Active-Standby 和 Standby-Active 状态）

2.5.3. 注册中心

保存某个服务所在地址等信息，方便调用者实时获取其他服务信息

- 服务注册
 - 服务提供者
- 服务发现
 - 服务消费者

2.5.4. 负载均衡

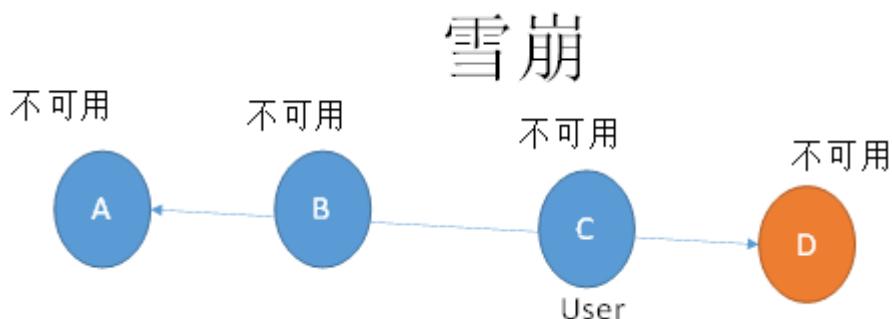
- 动态将请求派发给比较闲的服务器

策略：

- 轮询(Round Robin)
- 加权轮询(Weighted Round Robin)
- 随机 Random
- 哈希 Hash
- 最小连接数 LC
- 最短响应时间 LRT

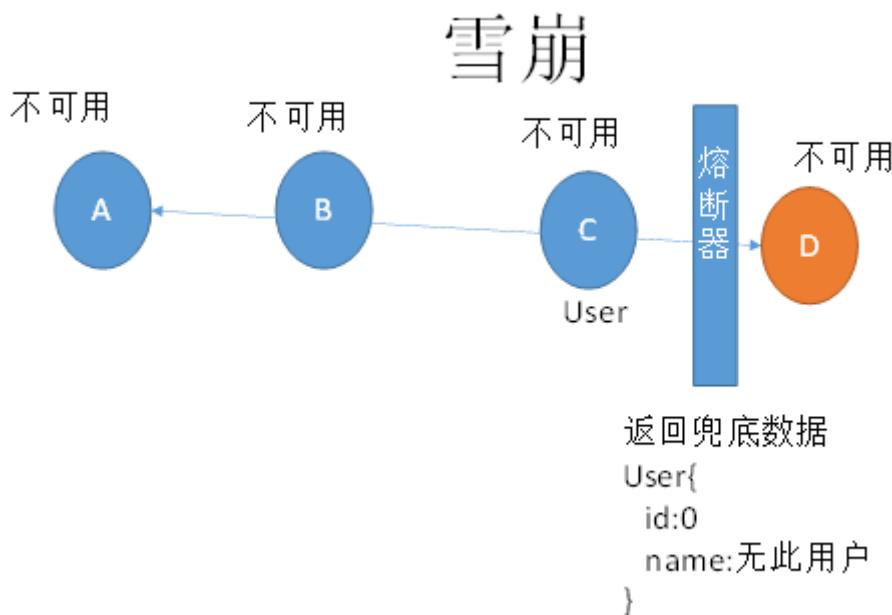
2.5.5. 服务雪崩

服务之间复杂调用，一个服务不可用，导致整个系统受影响不可用



2.5.6. 熔断

某个服务频繁超时，直接将其短路，快速返回 mock (模拟/虚拟) 值

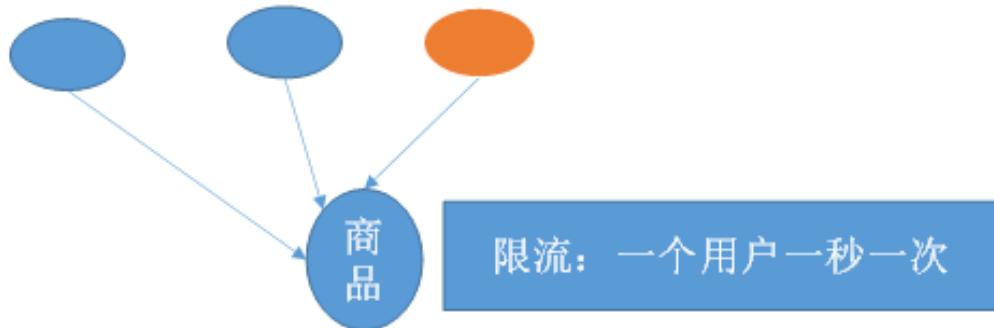


2.5.7. 限流

限制某个服务每秒的调用本服务的频率

防爬虫

Ddos 攻击；可能导致服务不可用（洪水攻击）



2.5.8. API 网关

API 网关要做很多工作，它作为一个系统的后端总入口，承载着所有服务的组合路由转换等工作，除此之外，我们一般也会把安全，限流，缓存，日志，监控，重试，熔断等放到 API 网关来做

2.5.9. 服务跟踪

追踪服务的调用链，记录整个系统执行请求过程。如：请求响应时间，判断链中的哪些服务属于慢服务（可能存在质量问题，需要改善）。

2.5.10. 弹性云

Elastic Compute Service (ECS) 弹性计算服务

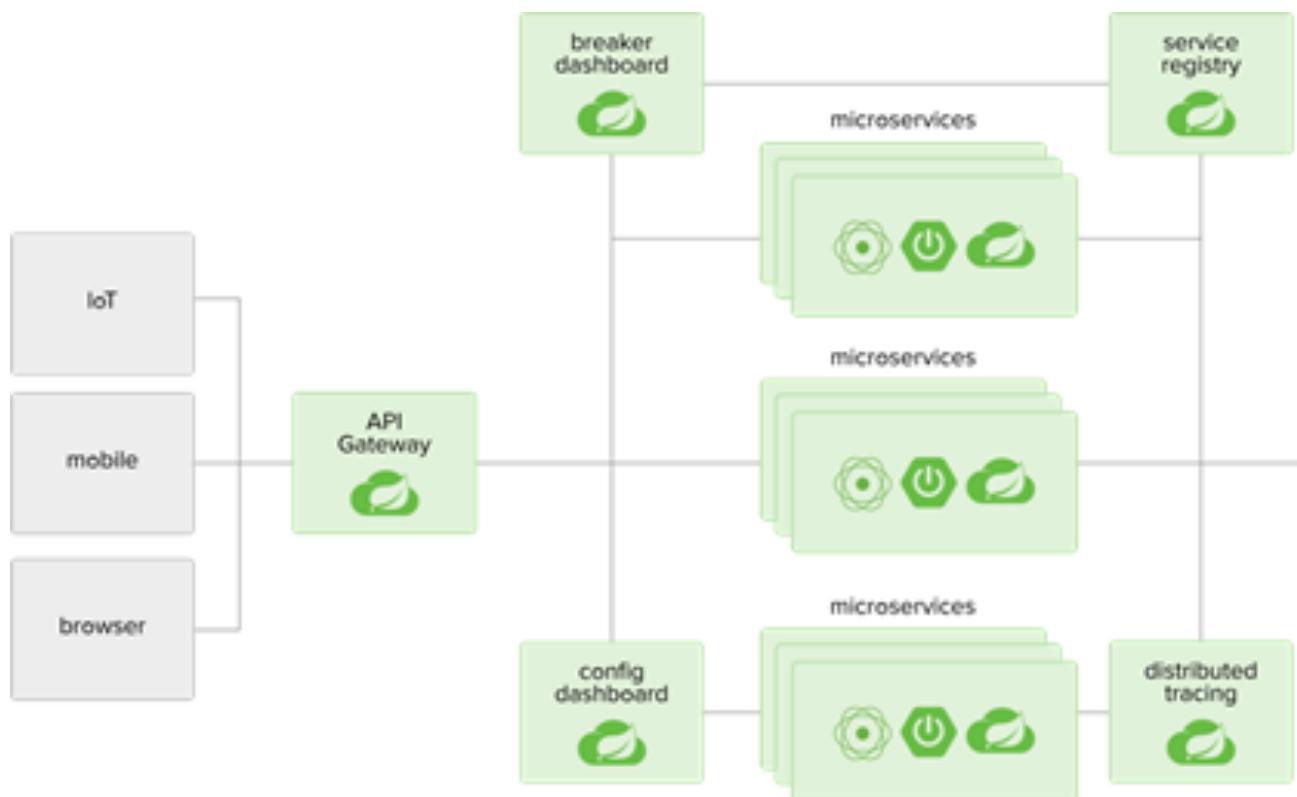
动态扩容，压榨服务器闲时能力

例如：双 11,618，高峰时多配置些服务器，平时减少多余的服务器配置（用于其他服务应用），避免资源浪费

3. SpringCloud 背景

3.1. 背景介绍

3.1.1. 微服务架构



物联网（ IoT , Internet of things ）即“万物相连的互联网”，是互联网基础上的延伸和扩展的网络，将各种信息传感设备与互联网结合起来而形成的一个巨大网络，实现在任何时间、任何地点，人、机、物的互联互通。

Breaker dashboard 断路器仪表板

Distributed Tracing 分布式跟踪（分布式处理程序链跟踪用于监视网络等待时间，并可视化通过微服务的请求流）

3.1.2. 微服务框架之 SpringBoot

<https://docs.spring.io/spring-boot/docs/2.3.6.RELEASE/reference/htmlsingle/>

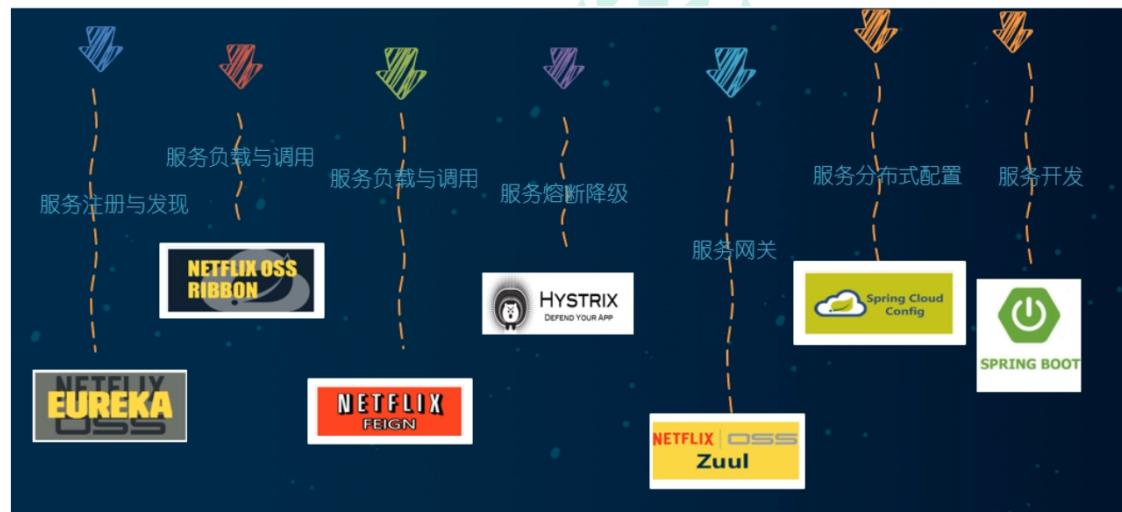
3.1.3. 分布式系统微服务架构之 SpringCloud

<https://docs.spring.io/spring-cloud/docs/Hoxton.SR9/reference/html/>

英文困难的同学，也不耽误学习的

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

3.1.4. 组件概述



3.2. 关于 SpringBoot 和 SpringCloud 版本

3.2.1. SpringCloud 版本选择

SpringBoot2.3.6 版和 SpringCloud Hoxton.SR9 版

SpringCloud Alibaba 2.2.6

3.2.2. Springboot 版本选择

git 源码地址：

<https://github.com/spring-projects/spring-boot/releases/>

SpringBoot2.0 新特性：

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0-Release-Notes>

通过上面官网发现，Boot 官方强烈建议你**升级到 2.X 以上版本**

Spring Boot 2.0 Release Notes

Phillip Webb edited this page on 10 Sep 2019 · 46 revisions

Upgrading from Spring Boot 1.5

Since this is a major release of Spring Boot, upgrading existing applications can be a little more involved than usual. We've put together a [dedicated migration guide](#) to help you upgrade your existing Spring Boot 1.5 applications.

If you're currently running with an earlier version of Spring Boot, we strongly recommend that you [upgrade to Spring Boot 1.5](#) before migrating to Spring Boot 2.0.

3.2.3. 官网看 Boot 版本

springboot(截至 2021.9.28)

Spring Boot

2.5.5



OVERVIEW

LEARN

SAMPLES

Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

2.5.5 CURRENT GA	Reference Doc.	API Doc.
2.6.0-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
2.6.0-M3 PRE	Reference Doc.	API Doc.
2.5.6-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
2.4.12-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
2.4.11 GA	Reference Doc.	API Doc.
2.3.12.RELEASE GA	Reference Doc.	API Doc.

3.2.4. SpringCloud 版本选择



- git 源码地址: <https://github.com/spring-projects/spring-cloud/wiki>
- 官网: <https://spring.io/projects/spring-cloud>

官网看 Cloud 版本

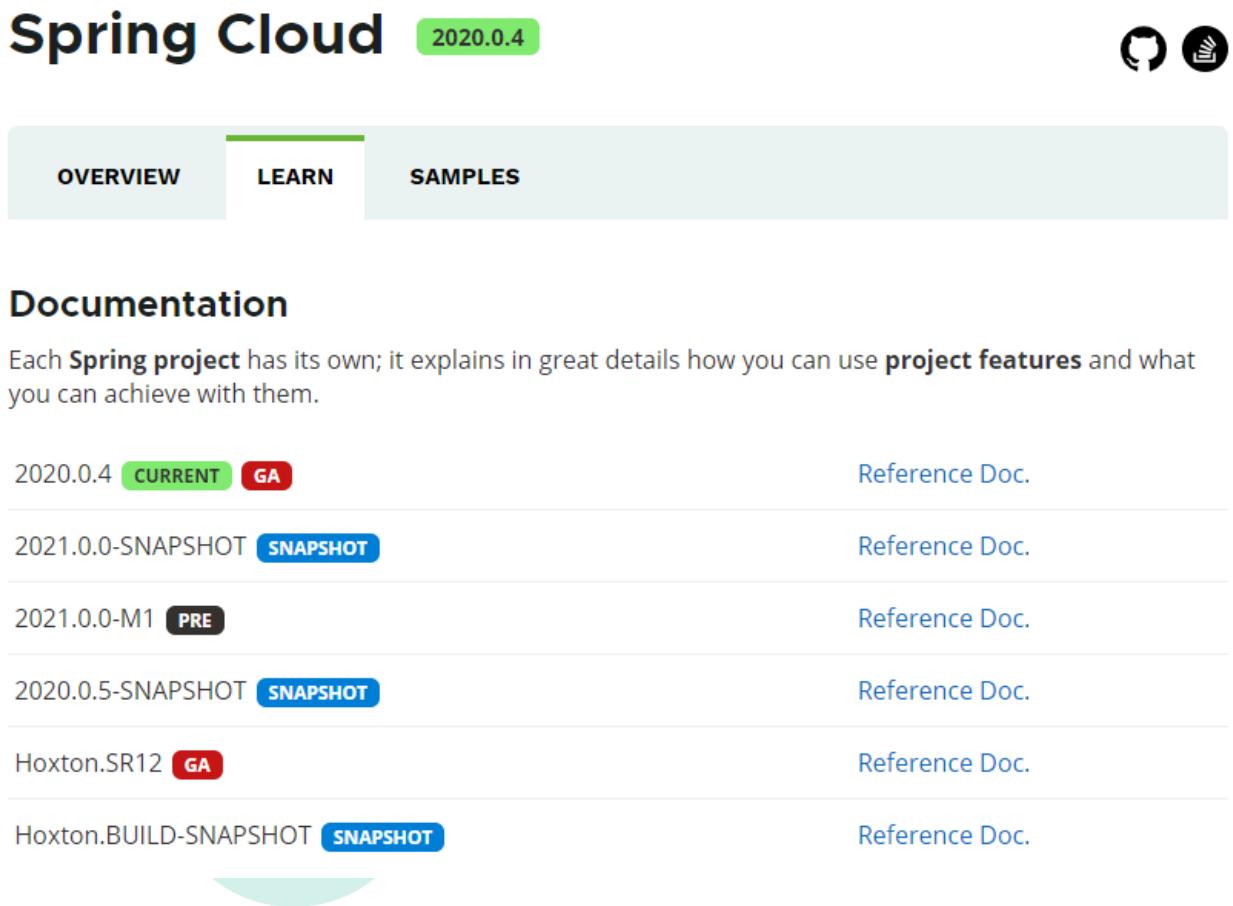
- Cloud 命名规则

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

Spring Cloud 是一个由许多子项目组成的综合项目，各子项目有不同的发布节奏。为了管理 SpringCloud 与各子项目的版本依赖关系，发布了一个清单，其中包括了某个 SpringCloud 版本对应的子项目版本。为了避免 SpringCloud 版本号与子项目

版本号混淆，SpringCloud 版本采用了名称而非版本号的命名，这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来应对版本时间顺序。例如 Angel 是第一个版本，Brixton 是第二个版本。当 SpringCloud 的发布内容积累到临界点或者一个重大 BUG 被解决后，会发布一个"service releases"版本，简称 SRX 版本，比如 Greenwich.SR2 就是 SpringCloud 发布的 Greenwich 版本的第二个 SRX 版本。

- SpringCloud(截至 2021.9.28)



The screenshot shows the official Spring Cloud documentation landing page. At the top, it displays "Spring Cloud" in large bold letters, followed by a green button labeled "2020.0.4". To the right are two icons: a GitHub logo and a PDF icon. Below this, there are three navigation tabs: "OVERVIEW" (disabled), "LEARN" (selected and highlighted in green), and "SAMPLES".

Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

Version	Status	Reference Doc.
2020.0.4	CURRENT GA	Reference Doc.
2021.0.0-SNAPSHOT	SNAPSHOT	Reference Doc.
2021.0.0-M1	PRE	Reference Doc.
2020.0.5-SNAPSHOT	SNAPSHOT	Reference Doc.
Hoxton.SR12	GA	Reference Doc.
Hoxton.BUILD-SNAPSHOT	SNAPSHOT	Reference Doc.



Hoxton.SR9

See all of the included issues and pull requests at the [GitHub project](#). Hoxton.SR10 is compatible with Spring Boot 2.3.x and 2.2.x.

2020-11-09

- Spring Cloud Starter Build [Hoxton.SR9](#)
- Spring Cloud Aws [2.2.5.RELEASE](#)
- Spring Cloud Vault [2.2.6.RELEASE](#)
- Spring Cloud Sleuth [2.2.6.RELEASE](#)
- Spring Cloud Contract [2.2.5.RELEASE](#)
- Spring Cloud Kubernetes [1.1.7.RELEASE](#)
- Spring Cloud Config [2.2.6.RELEASE](#)
- Spring Cloud Openfeign [2.2.6.RELEASE](#)
- Spring Cloud Commons [2.2.6.RELEASE](#)
- Spring Cloud Zookeeper [2.2.4.RELEASE](#)
- Spring Cloud Consul [2.2.5.RELEASE](#)
- Spring Cloud Gcp [1.2.6.RELEASE](#)
- Spring Cloud Netflix [2.2.6.RELEASE](#)
- Spring Cloud Gateway [2.2.6.RELEASE](#)
- Spring Cloud Cli [2.2.3.RELEASE](#)

3.2.5. SpringCloud 和 Springboot 之间的依赖关系

<https://docs.spring.io/spring-cloud/docs/Hoxton.SR9/reference/html/>

Spring Cloud

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Release Train Version: **Hoxton.SR9**

Supported Boot Version: **2.3.5.RELEASE**

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

→ C ⌂ 🔒 start.spring.io/actuator/info

```
▼ "spring-cloud": {  
    "Hoxton.SR12": "Spring Boot >=2.2.0.RELEASE and <2.4.0.M1",  
    "2020.0.4": "Spring Boot >=2.4.0.M1 and <2.5.6-SNAPSHOT",  
    "2020.0.5-SNAPSHOT": "Spring Boot >=2.5.6-SNAPSHOT and <2.6.0-M1",  
    "2021.0.0-M1": "Spring Boot >=2.6.0.M1 and <2.6.0-SNAPSHOT",  
    "2021.0.0-SNAPSHOT": "Spring Boot >=2.6.0-SNAPSHOT"  
},
```

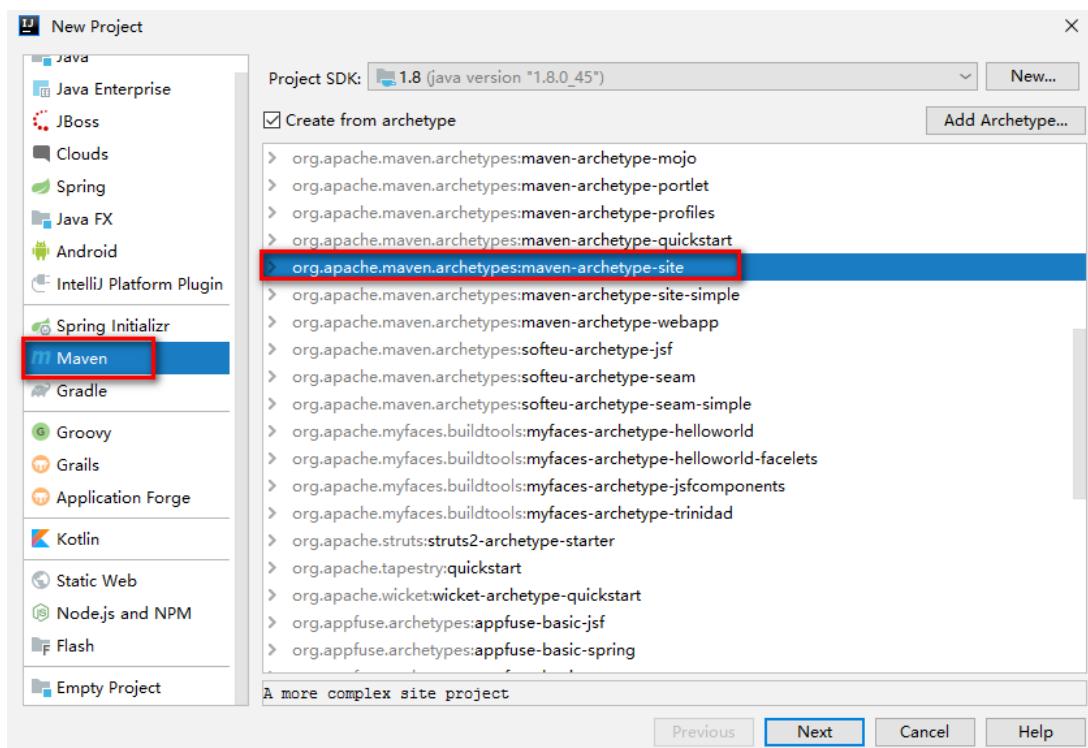
3.2.6. SpringCloud (授课选择版本)

- cloud
 - Hoxton.SR9
- boot
 - 2.3.6.RELEASE
- cloud Alibaba
 - 2.2.6.RELEASE
- java
 - JAVA8
- maven
 - 3.5 及以上
- mysql
 - 5.7 及以上

3.3. 微服务架构编码构建-IDEA 新建 project 工作空间

3.3.1. 微服务 cloud 整体聚合父工程 Project

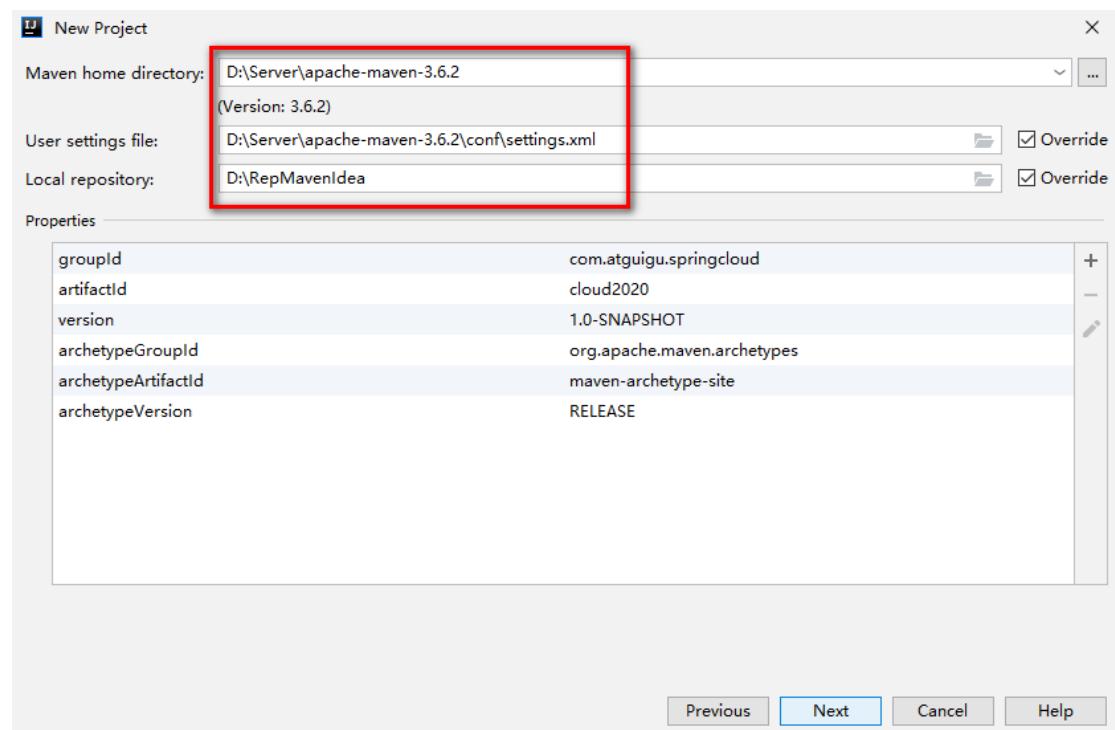
1. New Project



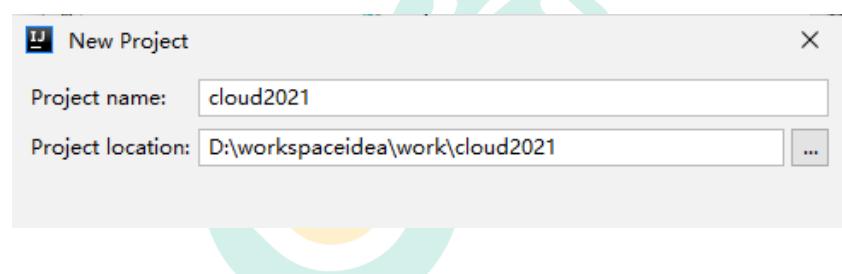
2. 聚合总工程名字



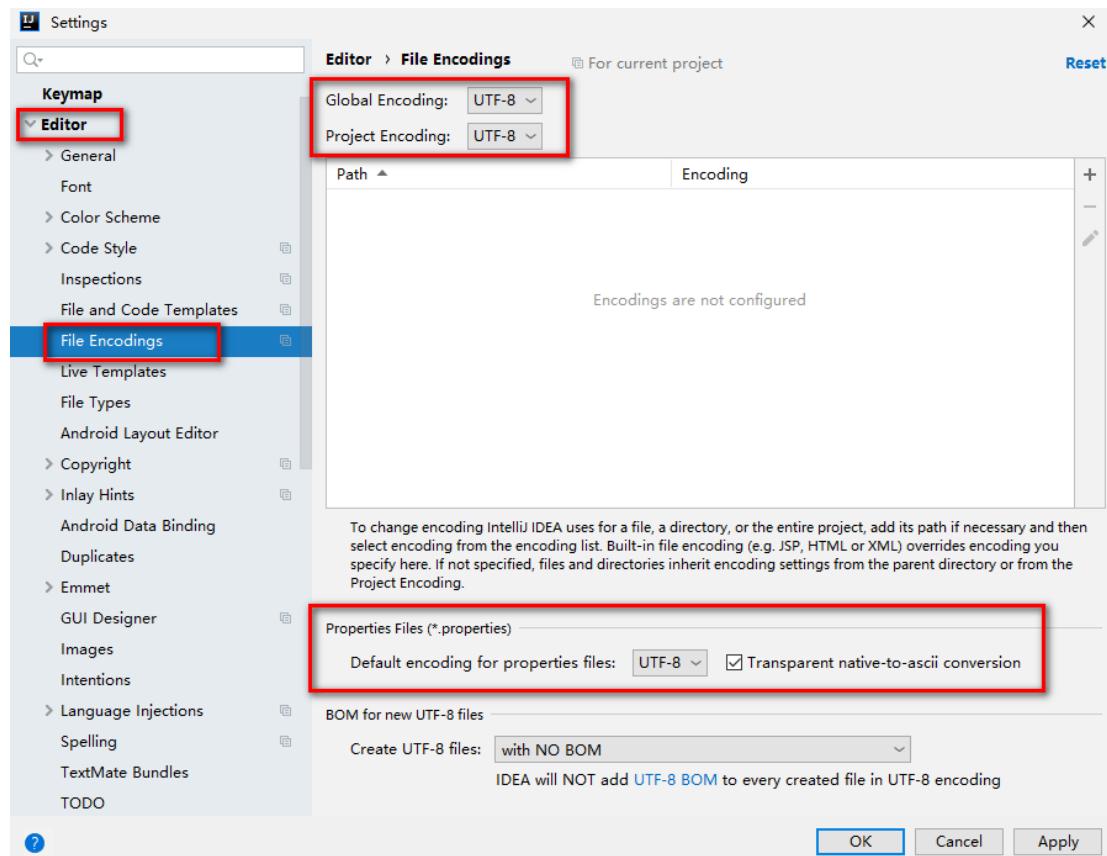
3. Maven 选版本



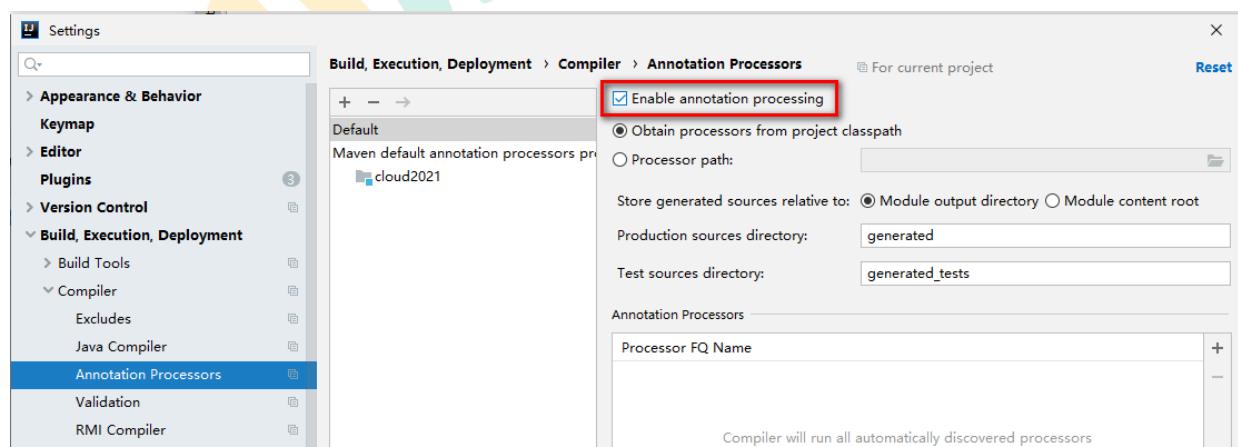
4. 工程名字



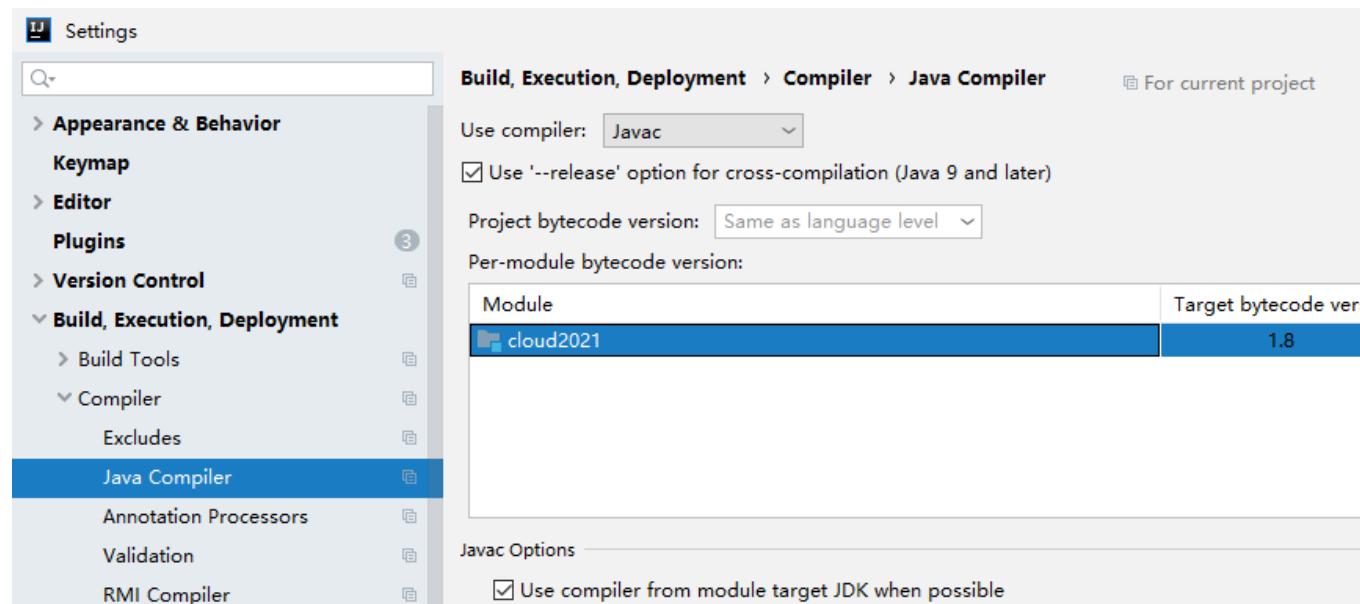
5. 字符编码



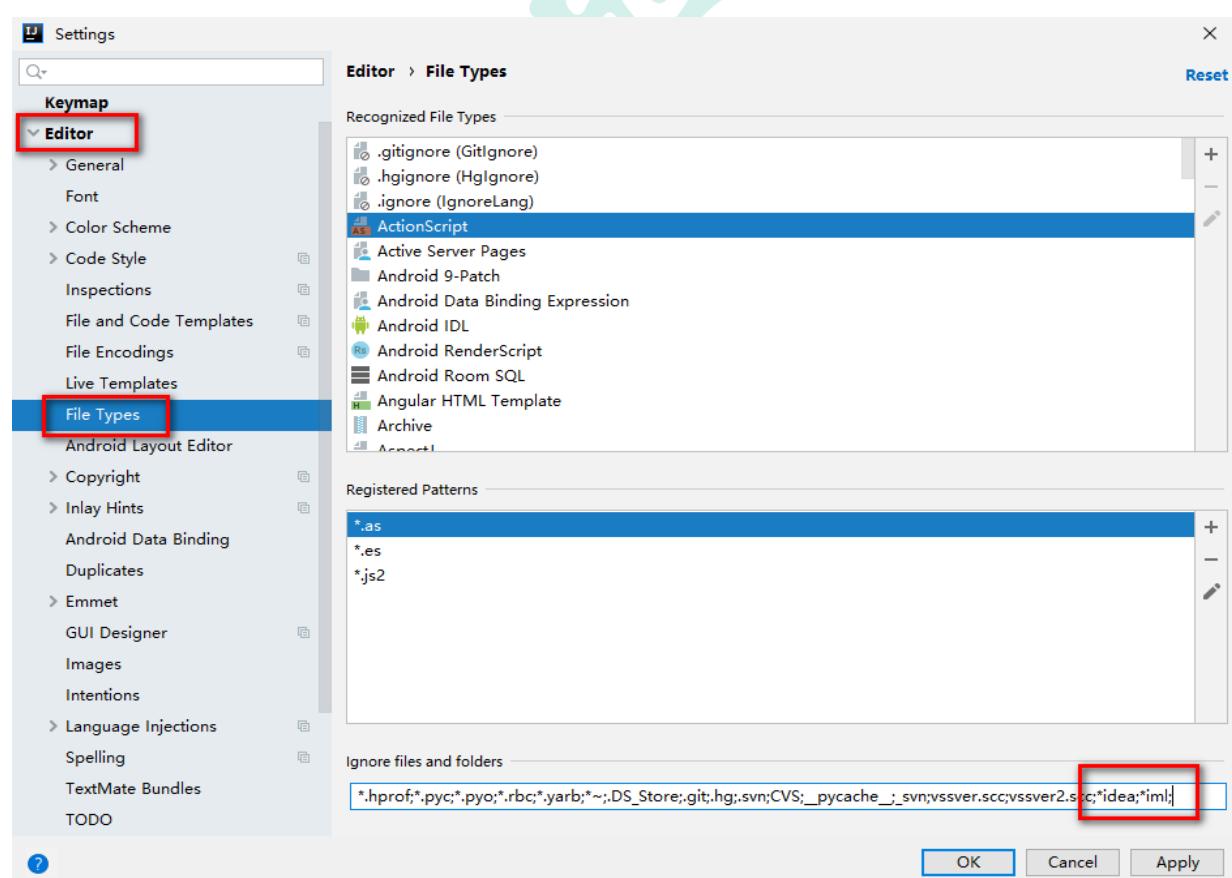
6. 注解生效激活



7. java 编译版本



8. File Type 过滤【可选】



3.3.2. 父工程 POM

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

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.atguigu</groupId>
  <artifactId>cloud2021</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <!-- 统一管理 jar 包版本 -->
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <junit.version>4.12</junit.version>
    <log4j.version>1.2.17</log4j.version>
    <lombok.version>1.16.18</lombok.version>
    <mysql.version>5.1.47</mysql.version>
    <druid.version>1.1.16</druid.version>
    <mybatis.spring.boot.version>1.3.0</mybatis.spring.boot.version>
  </properties>

  <!-- 子模块继承之后，提供作用：锁定版本+子 module 不用写 groupId 和 version -->
  <dependencyManagement>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.3.6.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Hoxton.SR9</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-alibaba-dependencies</artifactId>
        <version>2.2.6.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
```

```
<version>${druid.version}</version>
</dependency>

<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>${mybatis.spring.boot.version}</version>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j.version}</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>${lombok.version}</version>
    <optional>true</optional>
</dependency>

</dependencies>
</dependencyManagement>

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

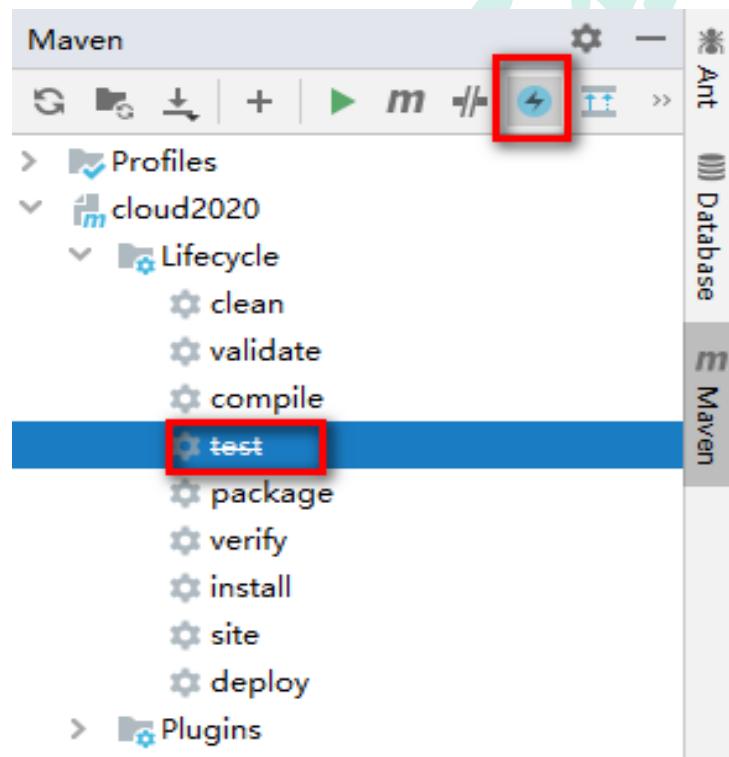
```
<configuration>
    <fork>true</fork>
    <addResources>true</addResources>
</configuration>
</plugin>
</plugins>
</build>

</project>
```

3.3.3. Maven 工程落地细节复习

Maven 中的 **dependencyManagement** 和 **dependencies** 区别

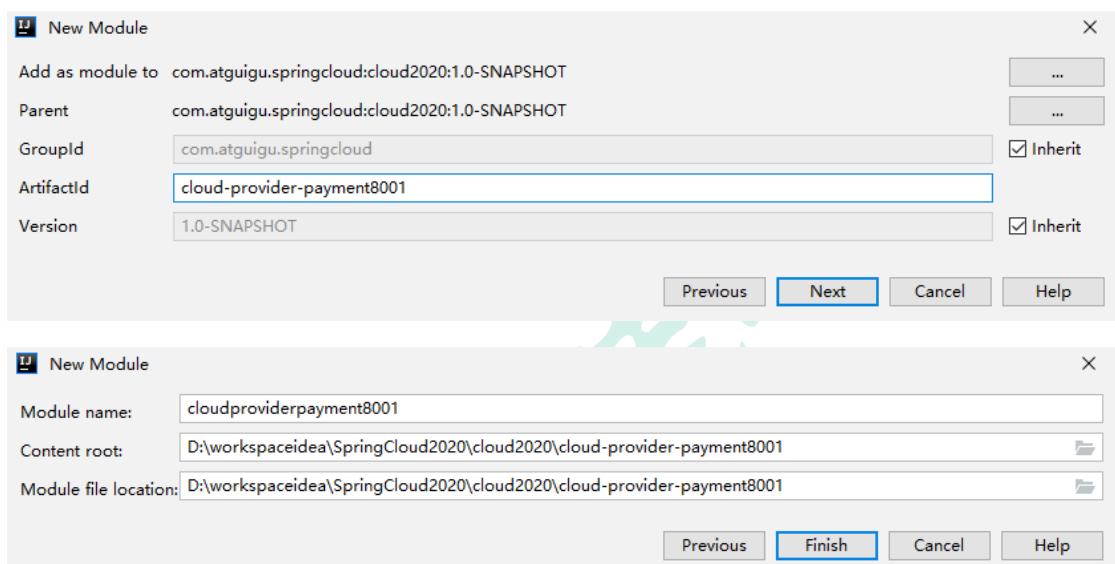
maven 中跳过单元测试【可选】



3.3.4. 父工程创建完成执行 mvn:install

3.4. 微服务架构编码构建-Rest 微服务-【服务提供者】

3.4.1. 建 cloud-provider-payment8001



创建完成后请回到父工程查看 pom 文件变化，增加了聚合模块

```
<modules>
  <module>cloud-provider-payment8001</module>
</modules>
```

3.4.2. 改 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">
  <parent>
    <artifactId>cloud2021</artifactId>
```

```
<groupId>com.atguigu.springcloud</groupId>
<version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>cloud-provider-payment8001</artifactId>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.10</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java </artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime </scope>
        <optional>true</optional>
    </dependency>
```

```
<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>
</dependency>

</dependencies>
</project>
```

3.4.3. 写 YML

```
server:
  port: 8001

spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/cloud2021?useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: root

  mybatis:
    mapperLocations: classpath:/mapper/*.xml
    type-aliases-package: com.atguigu.springcloud.entities
```

3.4.4. 主启动

```
package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PaymentMain8001 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain8001.class,args);
    }
}
```

3.4.5. 业务类

1. 建表 SQL

```
CREATE DATABASE IF NOT EXISTS cloud2021 DEFAULT CHARACTER SET utf8 ;

USE cloud2021 ;

DROP TABLE IF EXISTS payment ;

CREATE TABLE payment (
    id BIGINT (20) NOT NULL AUTO_INCREMENT COMMENT 'ID',
    SERIAL VARCHAR (300) DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE = INNODB AUTO_INCREMENT = 33 DEFAULT CHARSET = utf8 ;

INSERT INTO payment (id, SERIAL) VALUES(31, '尚硅谷 001'),(32, 'atguigu002') ;
```

2. Entities

1) 主实体 Payment

```
package com.atguigu.springcloud.entities;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.io.Serializable;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Payment implements Serializable {
    private Long id;
    private String serial;
}
```

2) Json 封装体 CommonResult

```
package com.atguigu.springcloud.entities;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class CommonResult <T> implements Serializable{

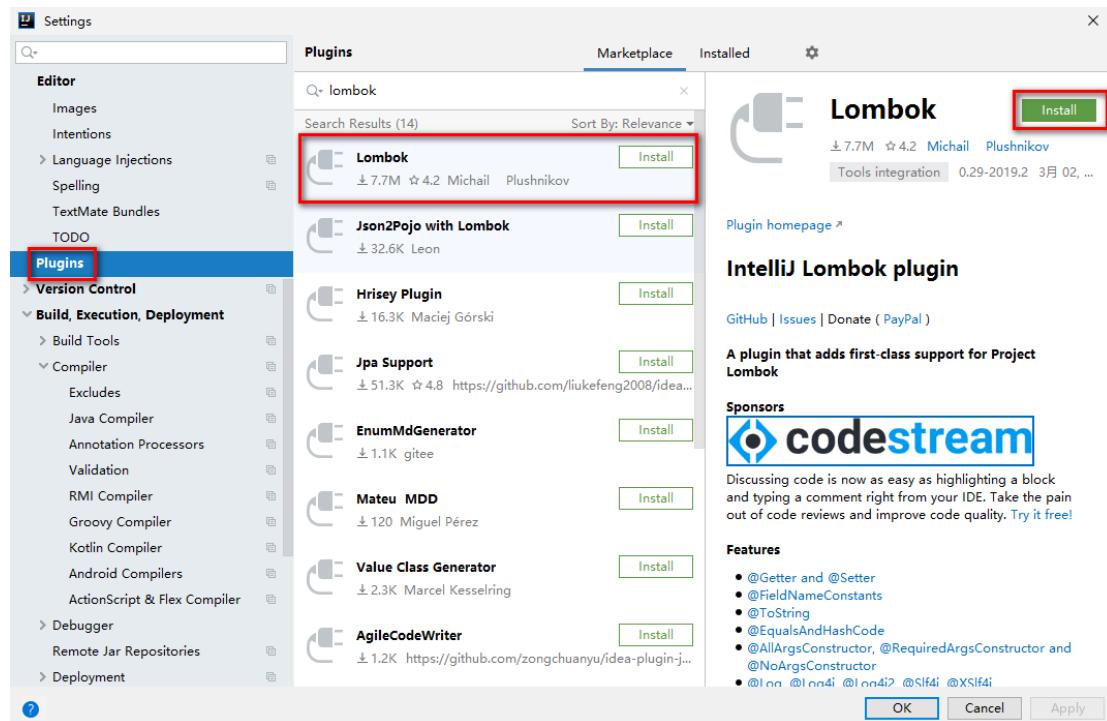
    private Integer code;
    private String message;
    private T data;

    public CommonResult(Integer code, String message){

```

```
this(code,message,null); //如果这行报错,请安装lombok插件  
}  
}
```

3) 安装 lombok 插件



<https://www.projectlombok.org/>

@Data：提供 getter/setter

@NoArgsConstructor, 无参构造器 @RequiredArgsConstructor @AllArgsConstructor
全参数构造器

@EqualsAndHashCode：提供 equals 和 hashCode 方法

@Getter/@Setter

@Slf4j 内置 log 对象，直接调用日志方法输出日志

3. Dao

1) 接口 PaymentDao

```
package com.atguigu.springcloud.dao;

import com.atguigu.springcloud.entities.Payment;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Param;
import org.springframework.stereotype.Repository;

@Component //代替@Repository 声明 bean
@Mapper //mybatis 提供的，等价： @MapperScan("com.atguigu.springcloud.dao")
//@Repository //spring 提供的。在此，只是为了声明 bean 对象
public interface PaymentDao {
    public int create(Payment payment);
    public Payment getPaymentById(@Param("id") Long id);
}
```

2) mybatis 的映射文件

src\main\resources\mapper\PaymentMapper.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.atguigu.springcloud.dao.PaymentDao">
    <insert id="create" useGeneratedKeys="true" keyProperty="id">
        insert into payment(serial) values(#{serial});
    </insert>

    <resultMap id="BaseResultMap"
type="com.atguigu.springcloud.entities.Payment">
        <id column="id" property="id" jdbcType="BIGINT"></id>
        <result column="serial" property="serial" jdbcType="VARCHAR"></result>
    </resultMap>
```

```
<select id="getPaymentById" parameterType="Long"
resultMap="BaseResultMap">
    select * from payment where id=#{id}
</select>

</mapper>
```

4. Service

1) 接口 PaymentService

```
package com.atguigu.springcloud.service;

import com.atguigu.springcloud.entities.Payment;
import org.apache.ibatis.annotations.Param;

public interface PaymentService {
    public int create(Payment payment); //写
    public Payment getPaymentById(Long id); //读取
}
```

2) 实现类 PaymentServiceImpl

```
package com.atguigu.springcloud.service.impl;

import com.atguigu.springcloud.dao.PaymentDao;
import com.atguigu.springcloud.entities.Payment;
import com.atguigu.springcloud.service.PaymentService;
import org.apache.ibatis.annotations.Param;
import org.springframework.stereotype.Service;
import javax.annotation.Resource;

@Service
public class PaymentServiceImpl implements PaymentService {
```

```
@Resource  
//@Autowired  
private PaymentDao paymentDao;  
  
public int create(Payment payment){  
    return paymentDao.create(payment);  
}  
  
public Payment getPaymentById( Long id){  
    return paymentDao.getPaymentById(id);  
}  
}
```

5. Controller



```
package com.atguigu.springcloud.controller;  
  
import com.atguigu.springcloud.entities.CommonResult;  
import com.atguigu.springcloud.entities.Payment;  
import com.atguigu.springcloud.service.PaymentService;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RestController;  
import javax.annotation.Resource;  
  
{@RestController  
@Slf4j  
public class PaymentController {  
  
    @Resource  
    private PaymentService paymentService;  
  
    @PostMapping(value = "/payment/create")  
    public CommonResult<Payment> create(Payment payment){ //埋雷  
        int result = paymentService.create(payment);  
        log.info("*****插入结果: " +result);  
        if (result>0){ //成功  
    }
```

```
return new CommonResult(200,"插入数据库成功",result);
}else {
    return new CommonResult(444,"插入数据库失败",null);
}
}

@GetMapping(value = "/payment/get/{id}")
public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id){
    Payment payment = paymentService.getPaymentById(id);
    log.info("*****查询结果: " + payment);
    if (payment!=null){ //说明有数据，能查询成功
        return new CommonResult(200,"查询成功",payment);
    }else {
        return new CommonResult(444,"没有对应记录，查询 ID: "+id,null);
    }
}
}
```

3.4.6. 测试

1. postman 测试 get 请求

The screenshot shows the Postman application interface. The request URL is `http://localhost:8001/payment/get/31`. The method is set to `GET`. The response status is `200 OK`, and the response body is a JSON object:

```
1: {
2:     "code": 200,
3:     "message": "查询成功",
4:     "data": {
5:         "id": 31,
6:         "serial": "尚硅谷001"
7:     }
8: }
```

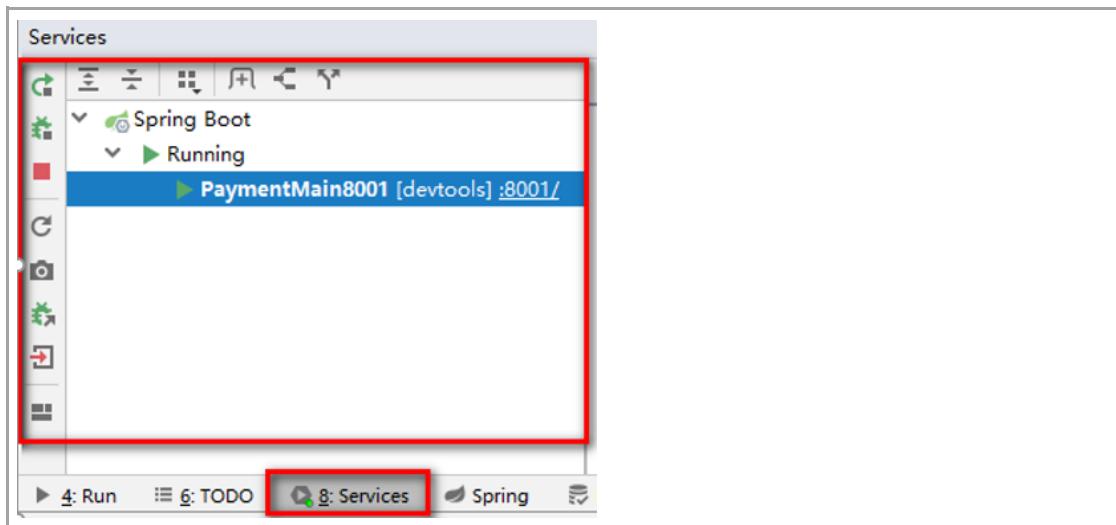
2. postman 测试 post 请求

The screenshot shows the Postman interface with a POST request to `http://localhost:8001/payment/create?serial=atguigu003`. The 'Params' tab shows a parameter `serial` with value `atguigu003`. The 'Body' tab shows the response body:

```
1 [ {  
2     "code": 200,  
3     "message": "插入成功",  
4     "data": 1  
5 }
```

3. 快速运行设置

The screenshot shows the 'Run/Debug Configurations' dialog in IntelliJ IDEA. The 'Templates' section is highlighted with a red box. A red box also surrounds the '+' button and the tooltip text: 'Click the + button to create a new configuration based on templates'. Other visible options include 'Confirm rerun with process termination' and 'Temporary configurations limit: 5'.



3.4.7. 开发步骤-小总结

- 1) 建 module
- 2) 改 POM
- 3) 写 YML
- 4) 主启动
- 5) 业务类

3.4.8. 热部署 Devtools

1. Adding devtools to your project

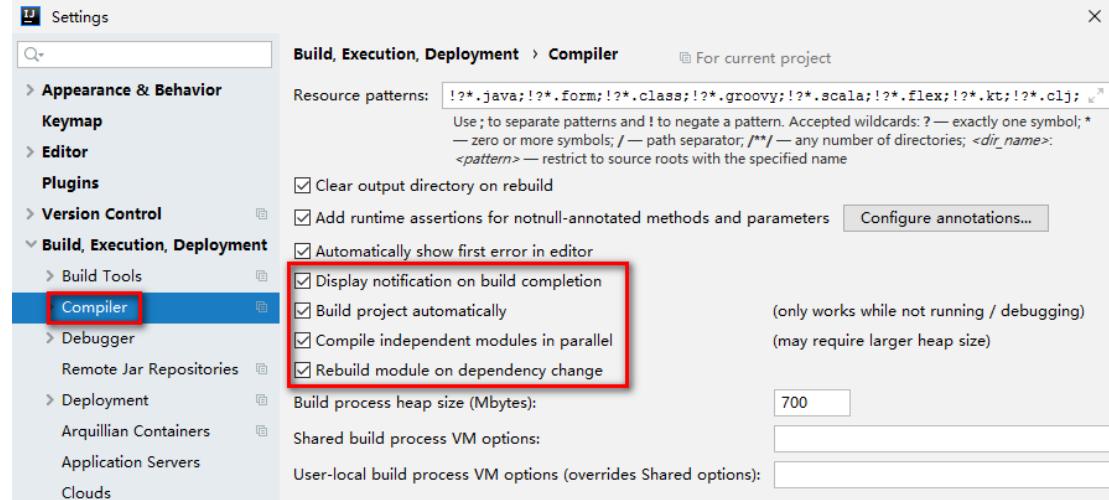
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

2. Adding plugin to your pom.xml

下一段配置黏贴到父工程当中的 pom 里

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

3. Enabling automatic build



4. Update the value of

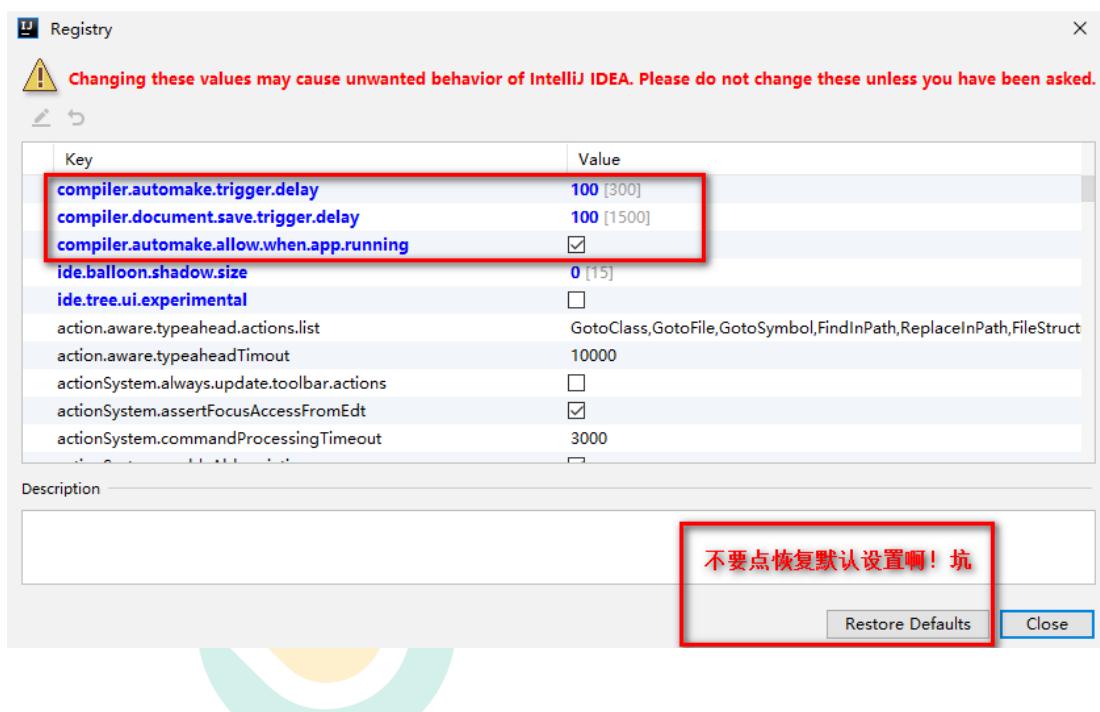
Ctrl+Shift+Alt+/选择 Registry...

Maintenance

1. Registry...
2. Switch Boot JDK...
3. UI Debugger
4. Experimental features...
5. Dump Lookup Element Weights to Log Ctrl+Alt+Shift+W

compiler.automake.allow.when.app.running -> 自动编译

compile.document.save.trigger.delay -> 自动更新文件；它主要是针对静态文件如 JS
CSS 的更新，将延迟时间减少后，直接按 F5 刷新页面就能看到效果！



5. 重启 IDEA

3.5. 微服务架构编码构建-Rest 微服务-【服务消费者】

3.5.1. 建 cloud-consumer-order80

3.5.2. 改 POM

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

    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloud-consumer-order80</artifactId>

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

```
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

<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>
</dependency>

</dependencies>
</project>
```

3.5.3. 写 YML

```
server:
  port: 80
spring:
  application:
    name: cloud-consumer-order80
```

3.5.4. 主启动

```
package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class OrderMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderMain80.class,args);
    }
}
```

3.5.5. 业务类

1. 创建 entities

(将 cloud-provider-payment8001 工程下的 entities 包下的两个实体类复制过来)

2. RestTemplate

- 是什么

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

- 官网及使用

官网地址：<https://docs.spring.io/spring-framework/docs/5.2.2.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html>

使用 RestTemplate 访问 Restful 接口非常的简单粗暴无脑。 (url, requestMap, ResponseBean.class) 这三个参数分别代表 REST 请求地址、请求参数、Http 响应转换被转换成的对象类型。

3. config 配置类

ApplicationContextConfig

```
package com.atguigu.springcloud.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

//@Configuration
@Configuration
public class ApplicationContextConfig {

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

}
```

4. 创建 controller

```
package com.atguigu.springcloud.controller;

import com.atguigu.springcloud.entities.CommonResult;
import com.atguigu.springcloud.entities.Payment;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
import javax.annotation.Resource;

@RestController
@Slf4j
public class OrderController {
```

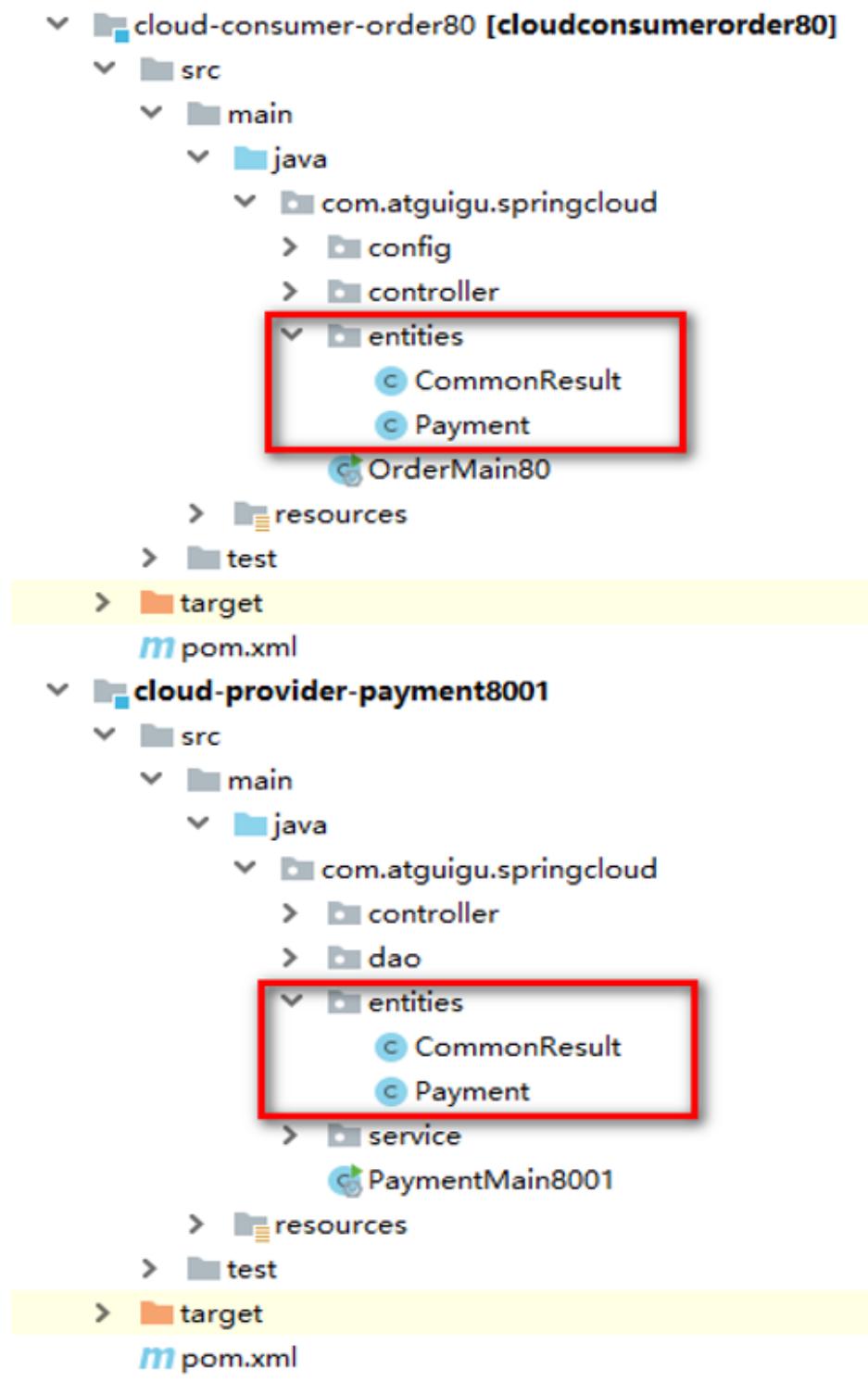
```
public static final String PAYMENT_URL = "http://localhost:8001";  
  
 @Resource  
 private RestTemplate restTemplate;  
  
 @PostMapping("/consumer/payment/create")  
 public CommonResult<Payment> create(@RequestBody Payment payment){  
     return  
     restTemplate.postForObject(PAYMENT_URL + "/payment/create", payment, CommonRe  
sult.class); //写操作  
 }  
  
 @GetMapping("/consumer/payment/get/{id}")  
 public CommonResult<Payment> getPayment(@PathVariable("id") Long id){  
     return  
     restTemplate.getForObject(PAYMENT_URL + "/payment/get/" + id, CommonResult.class)  
 ;  
 }  
 }
```

3.5.6. 测试

1. 先启动 cloud-provider-payment8001
2. 再启动 cloud-consumer-order80
3. <http://localhost/consumer/payment/get/32>
4. 不要忘记@RequestBody 注解
5. 服务提供者接口方法需要增加@RequestBody 注解(踩雷 or 破雷); 否则, 接收不
到数据。

3.6. 工程重构

3.6.1. 观察问题



3.6.2. 新建：cloud-api-commons

3.6.3. POM

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

    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloud-api-commons</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>cn.hutool</groupId>
            <artifactId>hutool-all</artifactId>
            <version>5.1.0</version>
        </dependency>
    </dependencies>
```

```
</project>
```

3.6.4. entities

Payment 实体

CommonResult 通用封装类

3.6.5. maven 命令 clean install

3.6.6. 订单 80 和支付 8001 分别改造

删除各自的原先有过的 entities

各自黏贴 POM 内容，依赖于 cloud-api-commons 公共项目

```
<dependency>
    <groupId>com.atguigu</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>${project.version}</version>
</dependency>
```

4. Eureka 服务注册与发现

4.1. Eureka 基础知识

4.1.1. 什么是服务治理

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

在传统的 RPC 远程调用框架中，管理每个服务与服务之间依赖关系比较复杂、所以需要进行服务治理，管理服务与服务之间依赖关联，以实现服务调用，负载均衡、容错等，实现服务发现与注册。

4.1.2. 什么是服务注册

Eureka 采用了 CS 的设计架构，Eureka Server 作为服务注册功能的服务器，它是服务注册中心。

而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。这样系统的维护人员可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。

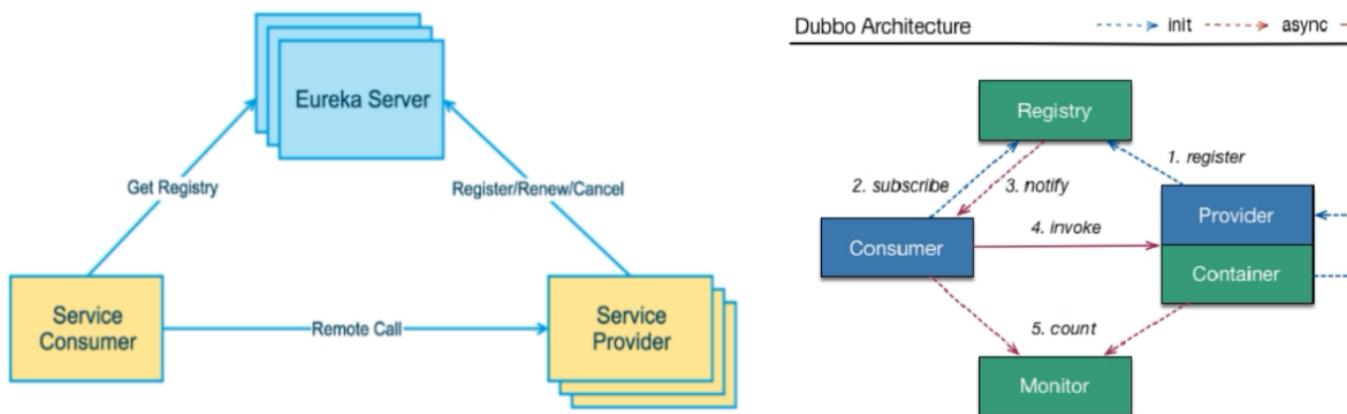
在服务注册与发现中，有一个注册中心。当服务器启动的时候，会把当前自己服务器的信息，比如：服务通讯地址等以别名方式注册到注册中心上。

另一方（消费者服务），以该别名的方式去注册中心上获取到实际的服务通讯地址，然后，再实现本地 RPC 远程调用。

RPC 远程调用框架核心设计思想：在于注册中心，因为使用注册中心管理每个服务与服务之间的一个依赖关系（服务治理概念）。

在任何 RPC 远程框架中，都会有一个注册中心（存放服务地址相关信息（接口地址））。

下左图是Eureka系统架构，右图是Dubbo的架构，请对比



4.1.3. Eureka 两组件

- Eureka Server 提供服务注册服务

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

- **Eureka Client 通过注册中心进行访问**

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

4.2. 单机 Eureka 构建步骤

4.2.1. IDEA 生成 eurekaServer 端服务注册中心

1. 建 Module：cloud-eureka-server7001

2. 改 POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloud-eureka-server7001</artifactId>

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

```
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>

<dependency>
    <groupId>com.atguigu</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>${project.version}</version>
</dependency>
<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.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
</dependency>
</dependencies>
</project>
```

3. 写 YML

```
server:  
  port: 7001  
  
eureka:  
  instance:  
    hostname: localhost  
  
client:  
  register-with-eureka: false  
  fetchRegistry: false  
  service-url:  
    defaultZone: http://localhost:7001/eureka
```

4. 主启动

```
@EnableEurekaServer
```

5. 测试

<http://localhost:7001/>

4.2.2. 服务提供者

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

1. 建 Module: cloud-provider-payment8001

2. 改 POM

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

```
</dependency>
```

3. 写 YML

```
eureka:  
  client:  
    register-with-eureka: true  
    fetchRegistry: true  
    service-url:  
      defaultZone: http://localhost:7001/eureka
```

4. 主启动

`@EnableEurekaClient`

5. 测试

先启动 EurekaServer

<http://localhost:7001/>

4.2.3. 服务消费者

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

1. 建 Module: cloud-consumer-order80

2. POM

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

3. 写 YML

```
eureka:  
  client:  
    register-with-eureka: true  
    fetchRegistry: true  
    service-url:  
      defaultZone: http://localhost:7001/eureka
```

4. 主启动

@EnableEurekaClient

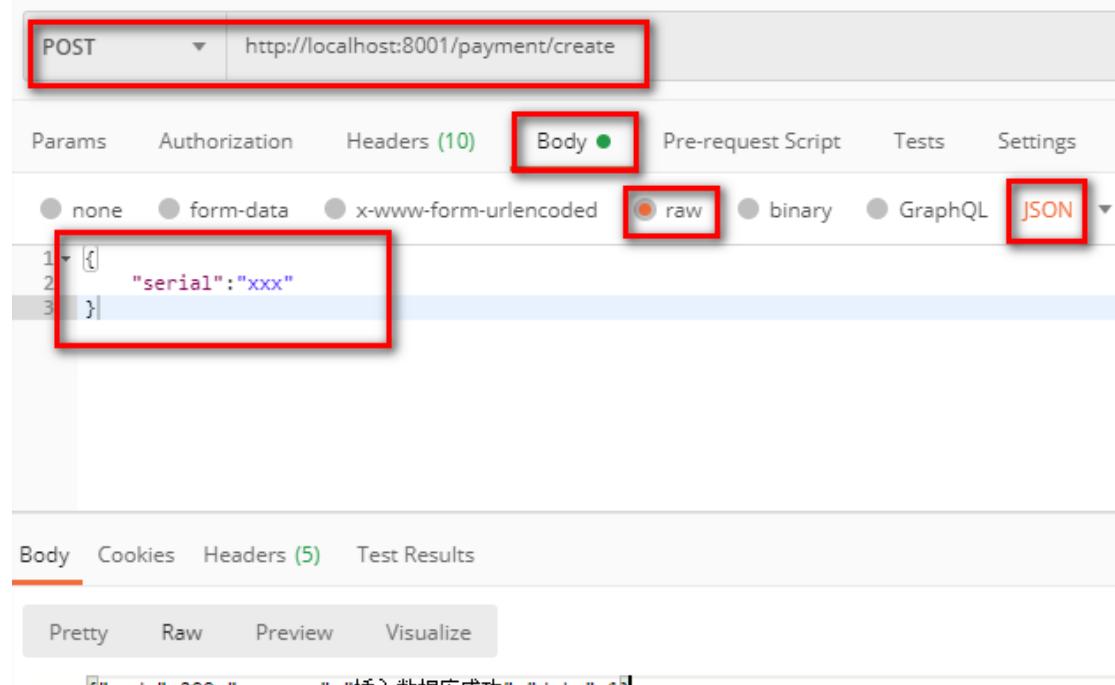
5. 测试

- 1) 先要启动 EurekaServer, 7001 服务
- 2) 再要启动服务提供者 8001 服务和服务消费者 80 服务
- 3) eureka 服务器

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.8:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.8:cloud-payment-service:8001

- 4) 测试查询: <http://localhost/consumer/payment/get/31>
- 5) 测试添加: postman 测试添加
- 6) 测试 8001 服务和 80 服务效果一样



5. Ribbon 负载均衡服务调用

5.1. 概述

5.1.1. 是什么

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

简单的说，Ribbon 是 Netflix 发布的开源项目，主要功能是提供客户端的软件负载均衡算法和服务调用。

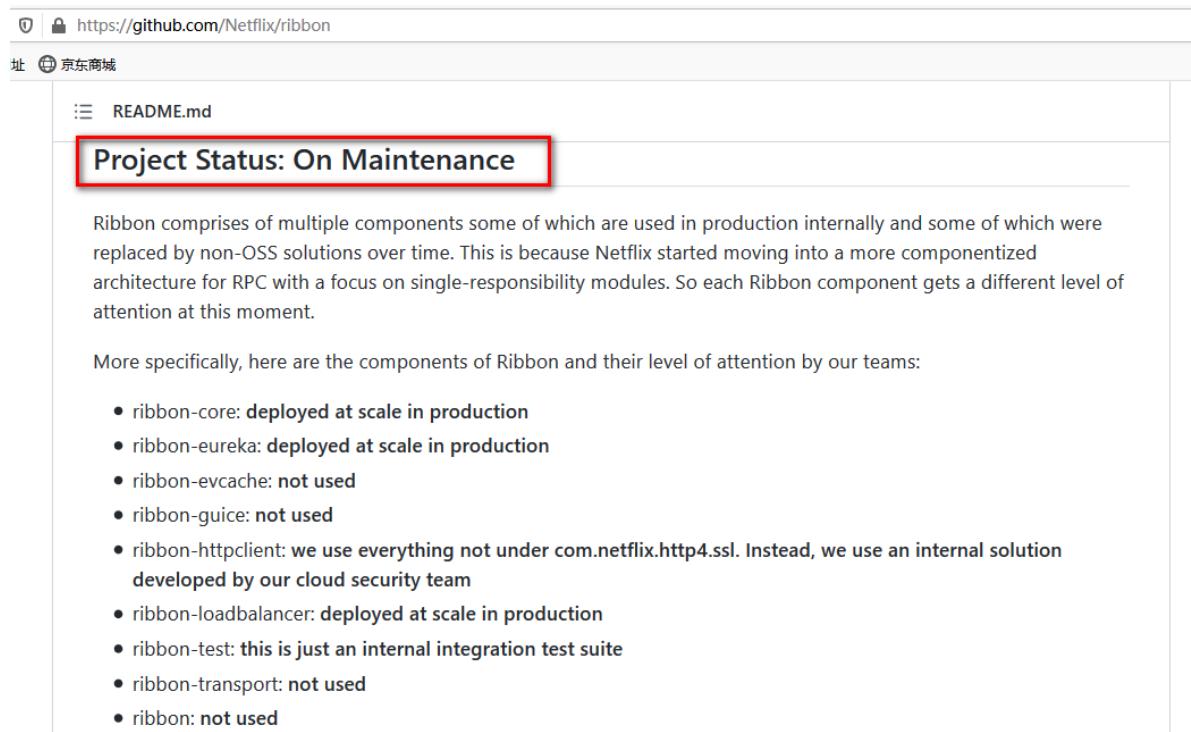
Ribbon 客户端组件提供一系列完善的配置项，如：连接超时，重试等。

简单的说，就是在配置文件中列出 Load Balancer(简称 LB)后面所有的机器，Ribbon 会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。

5.1.2. 官网资料

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

Ribbon 目前也进入维护模式



Project Status: On Maintenance

Ribbon comprises of multiple components some of which are used in production internally and some of which were replaced by non-OSS solutions over time. This is because Netflix started moving into a more componentized architecture for RPC with a focus on single-responsibility modules. So each Ribbon component gets a different level of attention at this moment.

More specifically, here are the components of Ribbon and their level of attention by our teams:

- ribbon-core: deployed at scale in production
- ribbon-eureka: deployed at scale in production
- ribbon-evcache: not used
- ribbon-guice: not used
- ribbon-httpclient: we use everything not under com.netflix.http4.ssl. Instead, we use an internal solution developed by our cloud security team
- ribbon-loadbalancer: deployed at scale in production
- ribbon-test: this is just an internal integration test suite
- ribbon-transport: not used
- ribbon: not used

- 未来替换方案
 - Spring Cloud LoadBalancer

5.1.3. 能干嘛

1. LB (负载均衡)

- 1) 简单的说就是将用户的请求平均分配到多个服务器上，从而达到系统的 HA(高可用)。
- 2) 常见的负载均衡有软件 Nginx, LVS, 硬件 F5 等。
- 3) **Ribbon 的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器端完成的。

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

4) 集中式 LB

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

5) 进程内 LB

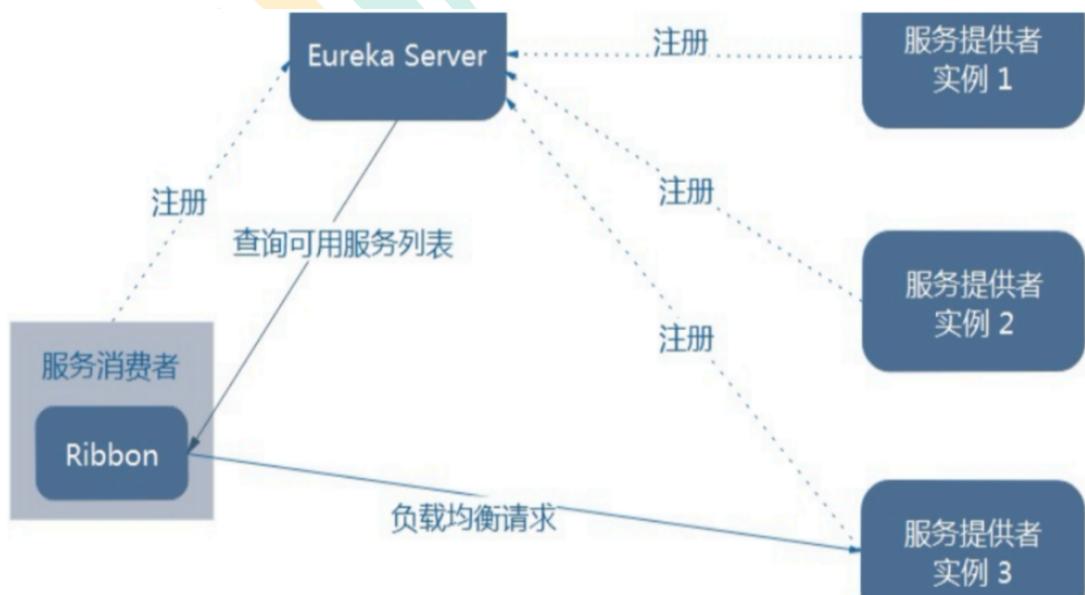
- 将 LB 逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。
- Ribbon 就属于进程内 LB，它只是一个类库，**集成于消费方进程**，消费方通过它来获取到服务提供方的地址。

2. 一句话

Ribbon=负载均衡+RestTemplate 调用

5.2. Ribbon 负载均衡演示

5.2.1. 架构说明



Ribbon 在工作时分成两步：

第一步，先选择 EurekaServer，它优先选择在同一个区域内负载较少的 server。

第二步，再根据用户指定的策略，在从 server 取到的服务注册列表中选择一个地址。其中 Ribbon 提供了多种策略。比如：轮询、随机和根据响应时间加权。

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

5.2.2. POM

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

注意：这个不需要手动引用，Eureka 客户端自带 Ribbon

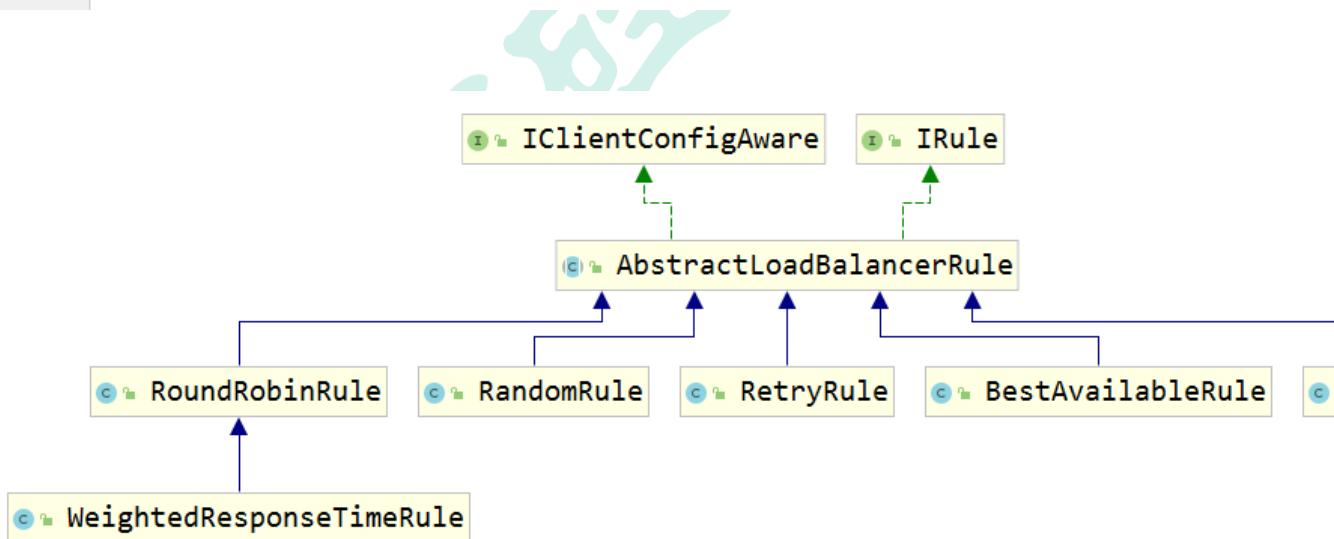


The screenshot shows a dependency tree for a project named 'cloud-consumer-order80'. The tree is visualized with colored bars indicating dependency status: green for direct dependencies and orange for runtime dependencies. The 'Dependencies' section of the tree includes the following entries:

- com.atguigu.springcloud:cloud-api-commons:1.0-SNAPSHOT
- org.springframework.boot:spring-boot-starter-web:2.3.6.RELEASE
- org.springframework.boot:spring-boot-starter-actuator:2.3.6.RELEASE
- org.springframework.boot:spring-boot-devtools:2.3.6.RELEASE (runtime)
- org.projectlombok:lombok:1.16.18
- org.springframework.boot:spring-boot-starter-test:2.3.6.RELEASE (test)
- org.springframework.cloud:spring-cloud-starter-netflix-eureka-client:2.2.6.RELEASE
 - org.springframework.cloud:spring-cloud-starter:2.2.6.RELEASE
 - org.springframework.cloud:spring-cloud-netflix-hystrix:2.2.6.RELEASE
 - org.springframework.cloud:spring-cloud-netflix-eureka-client:2.2.6.RELEASE
 - com.netflix.eureka:eureka-client:1.10.7
 - com.netflix.eureka:eureka-core:1.10.7
 - org.springframework.cloud:spring-cloud-starter-netflix-archaius:2.2.6.RELEASE
 - org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.6.RELEASE
 - org.springframework.cloud:spring-cloud-starter-loadbalancer:2.2.6.RELEASE
 - com.netflix.ribbon:ribbon-eureka:2.3.0
 - com.thoughtworks.xstream:xstream:1.4.13
 - org.springframework.cloud:spring-cloud-starter-zipkin:2.2.6.RELEASE

5.3. Ribbon 核心组件 IRule

```
IRule.java x
28  public interface IRule{
29      /**
30      * choose one alive server from lb.allServers or
31      * lb.upServers according to key
32      *
33      * @return choosen Server object. NULL is returned if none
34      * server is available
35      */
36
37      public Server choose(Object key);
38
39      public void setLoadBalancer(ILoadBalancer lb);
40
41      public ILoadBalancer getLoadBalancer();
42  }
```



5.3.1. IRule: 根据特定算法从服务列表中选取一个要访问的服务

1. com.netflix.loadbalancer.RoundRobinRule 轮询，默认策略。
2. com.netflix.loadbalancer.RandomRule 随机

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

5.3.2. 如何替换

- 修改 cloud-consumer-order80
- 注意配置细节

官方文档明确给出警告：

<https://docs.spring.io/spring-cloud-netflix/docs/2.2.6.RELEASE/reference/html/#customizing-the-ribbon-client>

7.2. Customizing the Ribbon Client

You can configure some bits of a Ribbon client by using external properties in `<client>.ribbon.*`, which is similar to using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in `CommonClientConfigKey` (part of ribbon-core).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`, as shown in the following example:

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

JAVA

In this case, the client is composed from the components already in `RibbonClientConfiguration`, together with any in `CustomConfiguration` (where the latter generally overrides the former).



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

这个自定义配置类不能放在`@ComponentScan`所扫描的当前包下以及子包下，否则我们自定义的这个配置类就会被所有的 Ribbon 客户端所共享，达不到特殊化订制的目的了。

5.3.3. 新建 package (注意：包的位置)

com.atguigu.myrule

5.3.4. 上面包下新建 MySelfRule 规则类

```
package com.atguigu.myrule;  
  
import com.netflix.loadbalancer.IRule;  
import com.netflix.loadbalancer.RandomRule;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class MySelfRule {
```

```
@Bean  
public IRule myRule(){  
    return new RandomRule(); //定义为随机  
}  
}
```

5.3.5. 主启动类添加@RibbonClient

```
package com.atguigu.springcloud;  
  
import com.atguigu.myrule.MySelfRule;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.cloud.netflix.ribbon.RibbonClient;  
  
{@EnableEurekaClient  
@SpringBootApplication  
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)  
public class OrderMain80 {  
    public static void main(String[] args) {  
        SpringApplication.run(OrderMain80.class, args);  
    }  
}}
```

5.3.6. 测试

<http://localhost/consumer/payment/get/31>

5.4. Ribbon 负载均衡算法

5.4.1. 原理

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

```
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.....

6. OpenFeign 服务接口调用

6.1. 概述

6.1.1. OpenFeign 是什么

- Feign 是一个声明式的 web 服务客户端，让编写 web 服务客户端变得非常容易，只需创建一个接口并在接口上添加注解即可
- SpringCloud 对 Feign 进行了封装，使其支持了 SpringMVC 标准注解和 HttpMessageConverters。Feign 可以与 Eureka 和 Ribbon 组合使用以支持负载均衡。
- <https://docs.spring.io/spring-cloud-openfeign/docs/2.2.6.RELEASE/reference/html/>

6.1.2. 能干嘛

- Feign 能干什么？

Feign 旨在使用编写 Java Http 客户端变得更容易。

前面在使用 Ribbon+RestTemplate 时，利用 RestTemplate 对 Http 请求的封装处理，形成了一套模板化的调用方法。

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端额调用。所以，Feign 在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。

在 Feign 的实现下，我们只需创建一个接口并使用注解的方式来配置它（以前是 DAO 接口上面标注 Mapper 注解，现在是一个微服务接口上面标注一个 Feign 注解即可），即可完成对服务提供方的接口绑定，简化了使用 Spring Cloud Ribbon 时，自动封装服务调用客户端的开发量。

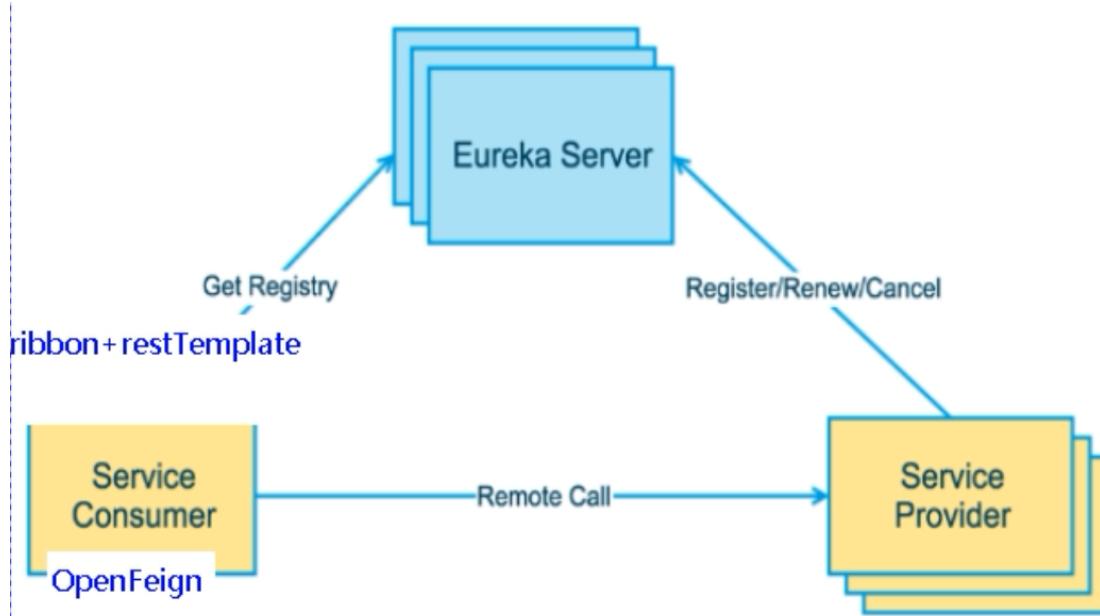
- Feign 集成了 Ribbon

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

- Feign 和 OpenFeign 两者区别

Feign	OpenFeign
<p>Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端 Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。Feign的使用方式是：使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务</p>	<p>OpenFeign是Spring Cloud 在Feign的基础上支持，如@ReqesMapping等等。OpenFeign的@Fei SpringMVC的@RequestMapping注解下的接口，式产生实现类，实现类中做负载均衡并调用其他服务</p>
<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-feign</artifactId> </dependency></pre>	<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-openfe</dependency></pre>

6.2. OpenFeign 使用步骤



6.2.1. 接口+注解

微服务调用接口 + @FeignClient

6.2.2. 新建 Module: cloud-consumer-feign-

order80

6.2.3. POM

注意: openFeign 也是自带 ribbon

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>cloud2021</artifactId>
```

```
<groupId>com.atguigu.springcloud</groupId>
<version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<artifactId>cloud-consumer-feign-order80</artifactId>

<dependencies>
    <!--openfeign-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
        <groupId>com.atguigu</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>${project.version}</version>
    </dependency>
    <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.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
```

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

6.2.4. YML

```
server:
  port: 80
spring:
  application:
    name: cloud-consumer-feign-order80
eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

6.2.5. 主启动类

```
package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class OrderFeignMain80 {
    public static void main(String[] args) {
```

```
    SpringApplication.run(OrderFeignMain80.class,args);
}
}
```

6.2.6. 业务类

1. 业务逻辑接口+**@FeignClient** 配置调用 provider 服务
2. 新建 **PaymentFeignService** 接口并新增注解**@FeignClient**

```
package com.atguigu.springcloud.service;

import com.atguigu.springcloud.entities.CommonResult;
import com.atguigu.springcloud.entities.Payment;
import feign.Param;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@Component
@FeignClient(value = "CLOUD-PAYMENT-SERVICE")
public interface PaymentFeignService {
    @GetMapping(value = "/payment/get/{id}")
    public CommonResult getPaymentById(@PathVariable("id") Long id);
}
```

3. 控制层 Controller

```
package com.atguigu.springcloud.controller;

import com.atguigu.springcloud.entities.CommonResult;
import com.atguigu.springcloud.entities.Payment;
import com.atguigu.springcloud.service.PaymentFeignService;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import javax.annotation.Resource;

@RestController
public class OrderFeignController {

    @Resource
    private PaymentFeignService paymentFeignService; //调用远程的微服接口

    @GetMapping(value = "/consumer/payment/get/{id}")
    public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id){
        return paymentFeignService.getPaymentById(id);
    }
}
```

6.2.7. 测试

1. 先启动 Eureka7001
2. 再启动 2 个微服务 8001/8002
3. 启动 OpenFeign 微服务：cloud-consumer-feign-order80
4. <http://localhost/consumer/payment/get/31>
5. Feign 自带负载均衡配置项

6.2.8. 小总结

The screenshot illustrates the integration of a Feign client and an Eureka service. On the left, a portion of the `PaymentController.java` file is shown, specifically the `getPaymentById()` method. This method uses a Feign client to call the `getPaymentById()` endpoint of the `CLOUD-PAYMENT-SERVICE`. A red arrow points from the Feign client code to the Eureka service registration on the right. On the right, the Eureka UI shows the `CLOUD-PAYMENT-SERVICE` application registered with two instances. A blue arrow points from the Feign client code to the service instance in the Eureka UI, indicating that the client is using the information provided by Eureka to resolve the service address.

```
@FeignClient(value = "CLOUD-PAYMENT-SERVICE")
public interface PaymentFeignService {
    @GetMapping(value = "/payment/get/{id}")
    CommonResult<Payment> getPaymentById(@PathVariable("id") Long id);
}

PaymentController.java
PaymentController > getPaymentById()

    @GetMapping(value = "/payment/get/{id}")
    public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id) {
        Payment payment = paymentService.getPaymentById(id);
        log.info("*****查询结果:{} ", payment);
        if (payment != null) {
            return new CommonResult< Payment >(code: 200, message: "查询成功" + "\t 服务端口: " + serverPort, payment);
        } else {
            return new CommonResult< Payment >(code: 404, message: "没有对应记录,查询ID: " + id, data: null);
        }
    }
}

Instances currently registered
Application          AIs
CLOUD-PAYMENT-SERVICE n/a (2)

General Info
Name
total-avail-memory
environment
num-of-cpus
```

6.3. OpenFeign 超时控制

6.3.1. 超时设置，故意设置超时演示出错情况

1. 服务提供方 8001 故意写暂停程序

```
@GetMapping(value = "/payment/feign/timeout")
public String paymentFeignTimeout(){
    try { TimeUnit.SECONDS.sleep(3); }catch (Exception e) {e.printStackTrace();} //单位秒
    return serverPort;
}
```

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

```
@GetMapping(value = "/payment/feign/timeout")
public String paymentFeignTimeout();
```

3. 服务消费方 80 添加超时方法 OrderFeignController

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

4. 测试

<http://localhost/consumer/payment/feign/timeout>

错误页面，OpenFeign 默认等待一秒钟，超过后报错

← → ⌂ ⌂ ① localhost/payment/feign/timeout

Whitelabel Error Page

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

Tue Apr 07 17:51:13 CST 2020

OpenFeign默认等待1秒,超时报错

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

Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

feign.RetryableException: Read timed out executing GET http://CLOUD-PAYMENT-SERVICE/payment/feign/timeout

```
at feign.FeignException.errorExecuting(FeignException.java:213)
at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:115)
at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:80)
at feign.ReflectiveFeign$FeignInvocationHandler.invoke(ReflectiveFeign.java:103)
at com.sun.proxy.$Proxy11.paymentFeignTimeout(Unknown Source)
at com.atguigu.springcloud.controller.OrderFeignController.paymentFeignTimeout(OrderFeignController.java:28)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:497)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:190)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)
```

6.3.2. 是什么

默认 Feign 客户端只等待一秒钟，但是，服务端处理需要超过 1 秒钟，导致 Feign 客户端不想等待了，直接报错。

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，也即 Ribbon 的超时时间，因为 Feign 集成了 Ribbon 进行负载均衡。

6.3.3. YML 中需要开启 OpenFeign 客户端超时控制

Feign 设置超时时间

使用 Feign 调用接口分两层，ribbon 的调用和 hystrix 的调用，所以 ribbon 的超时时间和 Hystrix 的超时时间的结合就是 Feign 的超时时间

```
#设置 Feign 客户端超时时间 (openfeign 默认支持 ribbon)

ribbon:
    ReadTimeout: 3000
    ConnectTimeout: 3000
    MaxAutoRetries: 1 #同一台实例最大重试次数,不包括首次调用
    MaxAutoRetriesNextServer: 1 #重试负载均衡其他的实例最大重试次数,不包括首次调用
    OkToRetryOnAllOperations: false #非 Get 请求是否重试
```

```
#hystrix 的超时时间

hystrix:
    command:
        default:
            execution:
                timeout:
                    enabled: true
                    isolation:
                        thread:
                            timeoutInMilliseconds: 9000
```

一般情况下都是 ribbon 的超时时间 (<) hystrix 的超时时间 (因为涉及到 ribbon 的重试机制)

因为 ribbon 的重试机制和 Feign 的重试机制有冲突，所以源码中默认关闭 Feign 的重试机制，源码如下

```
package feign;
import java.util.concurrent.TimeUnit;

public interface Retriever extends Cloneable {
    Retriever NEVER_RETRY = new Retriever() {
        public void continueOrPropagate(RetryableException e) { throw e; }

        public Retriever clone() { return this; }
    };
}
```

要开启 Feign 的重试机制如下：(Feign 默认重试五次 源码中有)

```
@Bean
Retriever feignRetriever() {
    return new Retriever.Default();
}
```

根据上面的参数计算重试的次数：

$(\text{MaxAutoRetries} + 1) * (\text{MaxAutoRetriesNextServer} + 1)$ —— 产生 4 次调用

如果在重试期间，时间超过了 hystrix 的超时时间，便会立即执行熔断，fallback。所以要根据上面配置的参数计算 hystrix 的超时时间，使得在重试期间不能达到 hystrix 的超时时间，不然重试机制就会没有意义

hystrix 超时时间的计算： $(1 + \text{MaxAutoRetries} + \text{MaxAutoRetriesNextServer}) * \text{ReadTimeout}$ 即按照以上的配置 hystrix 的超时时间应该配置为 $(1+1+1) * 3 = 9$ 秒

当 ribbon 超时后且 hystrix 没有超时，便会采取重试机制。当 OkToRetryOnAllOperations 设置为 false 时，只会对 get 请求进行重试。如果设置为 true，便会对所有的请求进行重试，如果是 put 或 post 等写操作，如果服务器接口没做幂等性，会产生不好的结果，所以 OkToRetryOnAllOperations 慎用。

如果不配置 ribbon 的重试次数，默认会重试一次

注意：



默认情况下，GET 方式请求无论是连接异常还是读取异常，都会进行重试

非 GET 方式请求，只有连接异常时，才会进行重试

6.4. OpenFeign 日志打印功能

6.4.1. 日志打印功能

6.4.2. 是什么

Feign 提供了日志打印功能，我们可以通过配置来调整日志级别，从而了解 Feign 中 Http 请求的细节。说白了就是对 Feign 接口的调用情况进行监控和输出。

6.4.3. 日志级别

NONE：默认的，不显示任何日志

BASIC：仅记录请求方法、RUL、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

6.4.4. 配置日志 bean

```
package com.atguigu.springcloud.config;

import feign.Logger;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FeignConfig {

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

6.4.5. YML 文件里需要开启日志的 Feign 客户端

6.4.6. 后台日志查看

<http://localhost/consumer/payment/get/31>

```
[PaymentFeignService#getPaymentById] <--- HTTP/1.1 200 (434ms)
[PaymentFeignService#getPaymentById] connection: keep-alive
[PaymentFeignService#getPaymentById] content-type: application/json
[PaymentFeignService#getPaymentById] date: Tue, 07 Apr 2020 11:08:11 GMT
[PaymentFeignService#getPaymentById] keep-alive: timeout=60
[PaymentFeignService#getPaymentById] transfer-encoding: chunked
[PaymentFeignService#getPaymentById]
[PaymentFeignService#getPaymentById] {"code":200,"message":"查询成功,port=8002","data":{"id":31,"serial":"尚硅谷
[PaymentFeignService#getPaymentById] <--- END HTTP (88-byte body)
```

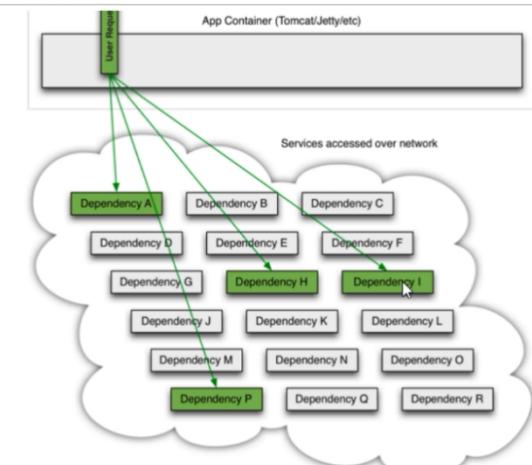
7. Hystrix 断路器

7.1. 概述

7.1.1. 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，每一个依赖关系在某些时候将不可避免的失败。

分布式系统面临的问题



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



服务雪崩

多个微服务之间调用的时候，假如微服务 A 调用微服务 B 和微服务 C，微服务 B 和微服务 C 又调用其他的微服务，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的时间过长或者不可用，对微服 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的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**。

7.1.2. 是什么

Hystrix 是一个用于处理分布式系统的**延迟和容错**的开源库，在分布式系统里，许多依赖不可避免的会**调用失败**，比如**超时、异常等**，

Hystrix 能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障**，以提高分布式系统的弹性。

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

7.1.3. 能干嘛

- 服务降级
- 服务熔断

- 接近实时的监控
- . . .

7.1.4. Hystrix 宣布，停更进维

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

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

京东商城

☰ README.md

Hystrix Status

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

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

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

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

7.2. Hystrix 重要概念

7.2.1. 服务降级 Fallback

- 服务器忙，请稍候再试，不让客户端等待并立刻返回一个友好提示
- 哪些情况会触发降级
 - 程序运行异常
 - 超时自动降级
 - 服务熔断触发服务降级

- 线程池/信号量打满也会导致服务降级
- 人工降级

7.2.2. 服务熔断 Breaker

- 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
- 就是保险丝
 - 服务的降级->进而熔断->恢复调用链路

7.2.3. 服务限流 Flowlimit

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

7.3. hystrix 案例

7.3.1. 构建

1. 新建 Module: cloud-provider-hystrix-payment8001

2. POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>cloud2021</artifactId>
    <groupId>com.atguigu.springcloud</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>cloud-provider-hystrix-payment8001</artifactId>

  <dependencies>
    <!--新增 hystrix-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
      <groupId>com.atguigu</groupId>
      <artifactId>cloud-api-commons</artifactId>
      <version>${project.version}</version>
    
```

```
</dependency>
<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.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<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>
</dependency>
</dependencies>
</project>
```

3. YML

```
server:
  port: 8001

spring:
  application:
    name: cloud-hystrix-payment-service

eureka:
  client:
```

```
register-with-eureka: true  
fetch-registry: true  
service-url:  
defaultZone: http://localhost:7001/eureka/
```

4. 主启动

```
package com.atguigu.springcloud;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
@SpringBootApplication  
  
@EnableEurekaClient  
public class PaymentHystrixMain8001 {  
    public static void main(String[] args) {  
        SpringApplication.run(PaymentHystrixMain8001.class,args);  
    }  
}
```

5. 业务类

Service/ServiceImpl

```
package com.atguigu.springcloud.service;  
  
public interface PaymentService {  
    public String paymentInfo_OK(Integer id);  
    public String payment_Timeout(Integer id);  
}
```

```
package com.atguigu.springcloud.service.impl;  
  
import org.springframework.stereotype.Service;  
import java.util.concurrent.TimeUnit;  
  
@Service
```

```
public class PaymentServiceImpl implements PaymentService {  
  
    //成功  
    public String paymentInfo_OK(Integer id){  
        return "线程池: "+Thread.currentThread().getName()+"  paymentInfo_OK,id:  
        "+id+"\t"+"哈哈哈" ;  
    }  
  
    //失败  
    public String payment_Timeout(Integer id){  
        int timeNumber = 3;  
        try { TimeUnit.SECONDS.sleep(timeNumber); }catch (Exception e)  
        {e.printStackTrace();}  
        return "线程池: "+Thread.currentThread().getName()+"  
paymentInfo_TimeOut,id: "+id+"\t"+"呜呜呜"+ " 耗时(秒)" +timeNumber;  
    }  
}
```

Controller

```
package com.atguigu.springcloud.controller;

import com.atguigu.springcloud.service.PaymentService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import javax.annotation.Resource;

@RestController
@Slf4j
public class PaymentController {

    @Resource
    private PaymentService paymentService;

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

    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfo_OK(@PathVariable("id") Integer id){
        String result = paymentService.paymentInfo_OK(id);
        log.info("*****result:"+result);
        return result;
    }

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id){
        String result = paymentService.payment_Timeout(id);
        log.info("*****result:"+result);
        return result;
    }
}
```

6. 正常测试

- 启动 eureka7001

- 启动 cloud-provider-hystrix-payment8001
- 访问

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

每次调用耗费 3 秒钟: <http://localhost:8001/payment/hystrix/timeout/31>

- 上述 module 均 OK

以上述为根基平台，从正确->错误->降级熔断->恢复

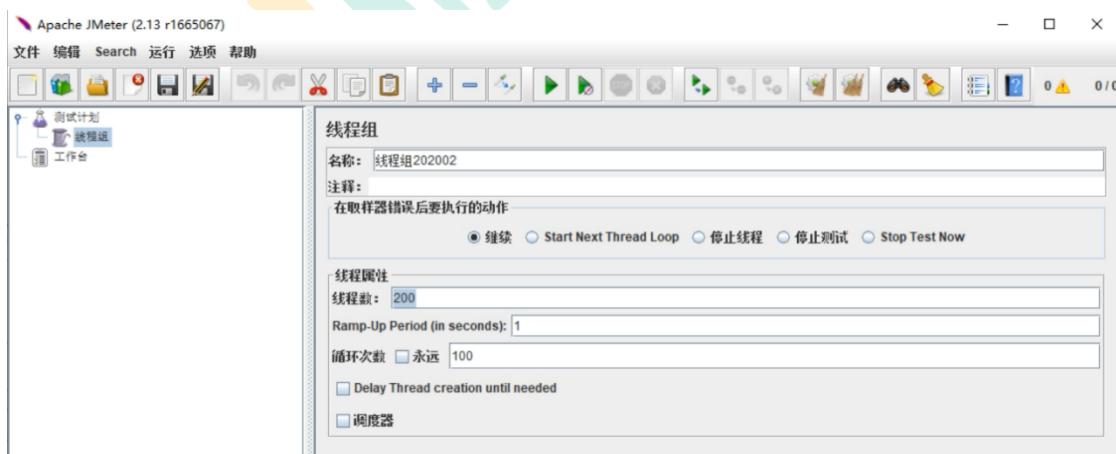
7.3.2. 高并发测试

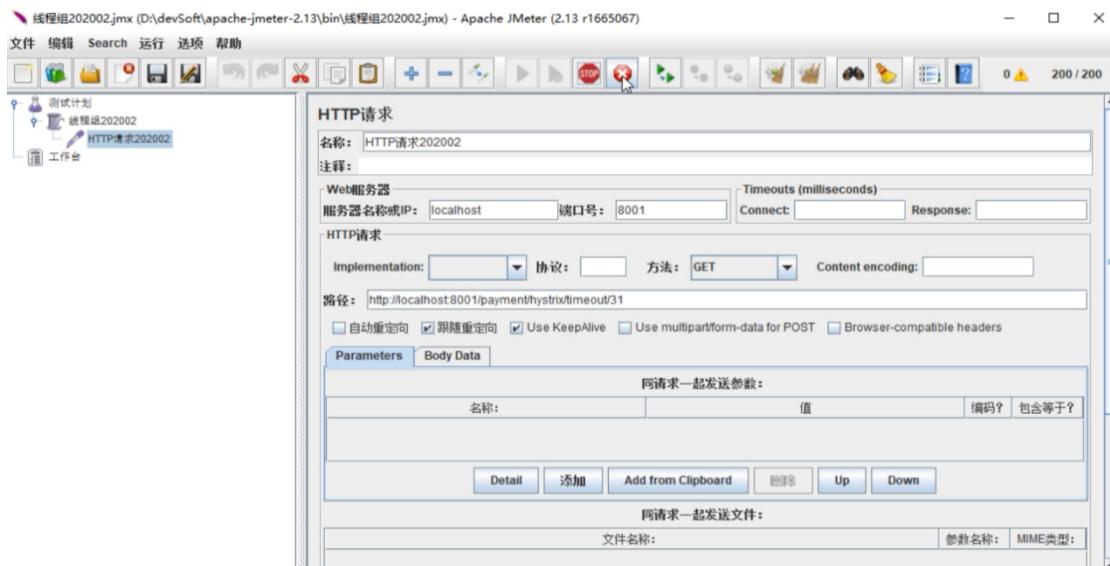
1. 上述在非高并发情形下，还能勉强满足 but.....

2. Jmeter 压测测试

下载地址: <https://archive.apache.org/dist/jmeter/binaries/>

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





- 压测的过程中再来访问一下微服务

<http://localhost:8001/payment/hystrix/ok/31>

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

- 演示结果

- 两个都在自己转圈圈
- 为什么会被卡死

tomcat 的默认的工作线程数被打满了，没有多余的线程来分解压力和处理。

3. Jmeter 压测结论

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

4. 看热闹不嫌弃事大，80 新建加入：cloud-consumer-feign-hystrix-order80

1) 新建：cloud-consumer-feign-hystrix-order80

2) POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloud-consumer-feign-hystrix-order80</artifactId>

    <dependencies>
        <!--新增 hystrix-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
            <groupId>com.atguigu.springcloud</groupId>
            <artifactId>cloud-api-commons</artifactId>
```

```
<version>${project.version}</version>
</dependency>
<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.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<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>
</dependency>
</dependencies>
</project>
```

3) YML

```
server:
  port: 80

spring:
  application:
    name: cloud-consumer-hystrix-order

eureka:
```

```
client:  
  register-with-eureka: true  #表示不向注册中心注册自己  
  fetch-registry: true  #表示自己就是注册中心，职责是维护服务实例，并不需要去检索服  
务  
  service-url:  
    defaultZone: http://localhost:7001/eureka/
```

4) 主启动

```
package com.atguigu.springcloud;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.openfeign.EnableFeignClients;  
  
@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients  
public class OrderHystrixMain80 {  
    public static void main(String[] args) {  
        SpringApplication.run(OrderHystrixMain80.class,args);  
    }  
}
```

5) 业务类

PaymentHystrixService

```
package com.atguigu.springcloud.service;  
  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
  
@Component  
@FeignClient("CLOUD-HYSTRIX-PAYMENT-SERVICE")  
public interface PaymentHystrixService {
```

```
@GetMapping("/payment/hystrix/ok/{id}")
public String paymentInfo_OK(@PathVariable("id") Integer id);

@GetMapping("/payment/hystrix/timeout/{id}")
public String payment_Timeout(@PathVariable("id") Integer id);
}
```

OrderHystrixController

```
package com.atguigu.springcloud.controller;

import com.atguigu.springcloud.service.PaymentHystrixService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import javax.annotation.Resource;

@RestController
@Slf4j
public class OrderHystrixController {
    @Resource
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/consumer/payment/hystrix/ok/{id}")
    public String paymentInfo_OK(@PathVariable("id") Integer id){
        String result = paymentHystrixService.paymentInfo_OK(id);
        log.info("*****result:"+result);
        return result;
    }

    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id){
        String result = paymentHystrixService.payment_Timeout(id);
        log.info("*****result:"+result);
        return result;
    }
}
```

6) 正常测试

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

7) 高并发测试

- 2W 个线程压 8001
- 消费端 80 微服务再去访问正常的 OK 微服务 8001 地址
- <http://localhost/consumer/payment/hystrix/timeout/32>
- 消费者 80, 呜呜呜
 - 要么转圈圈等待
 - 要么消费端报超时错误



Whitelabel Error Page

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

Tue Feb 04 17:08:55 CST 2020

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

```
Read timed out executing GET http://CLOUD-PROVIDER-HYSTRIX-PAYMENT/payment/hystrix/ok/32
feign.RetryableException: Read timed out executing GET http://CLOUD-PROVIDER-HYSTRIX-PAYMENT/payment/hystrix/ok/32
    at feign.FeignException.errorExecuting(FeignException.java:213)
    at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:115)
    at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:80)
    at feign.ReflectiveFeign$FeignInvocationHandler.invoke(ReflectiveFeign.java:103)
    at com.sun.proxy.$Proxy161.paymentInfo OK(Unknown Source)
```

7.3.3. 故障现象和导致原因

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

7.3.4. 上诉结论

- 正因为有上述故障或不佳表现，才有我们的降级/容错/限流等技术诞生

7.3.5. 如何解决？解决的要求

- 超时导致服务器变慢（转圈）
 - 超时不再等待
- 出错（宕机或程序运行出错）
 - 出错要有兜底
- 解决
 - 对方服务（8001）超时了，调用者（80）不能一直卡死等待，必须有服务降级
 - 对方服务（8001）down 机了，调用者（80）不能一直卡死等待，必须有服务降级
 - 对方服务（8001）OK，调用者（80）自己出故障或有自我要求（自己的等待时间小于服务提供者），自己处理降级

7.3.6. 服务降级

1. 降低配置

@HystrixCommand

2. 8001 先从自身找问题

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

3. 8001fallback

业务类启用阶级处理：使用@HystrixCommand 注解来干活。

```
package com.atguigu.springcloud.service.impl;

import com.atguigu.springcloud.service.PaymentService;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
```

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
import org.springframework.stereotype.Service;
import java.util.concurrent.TimeUnit;

@Service
public class PaymentServiceImpl implements PaymentService {

    @Override
    public String paymentInfo_OK(Integer id) {
        return "线程池: "+Thread.currentThread().getName()+" paymentInfo_OK,id=" + id
+ " \t O(∩_∩)O 哈哈~";
    }

    //超时降级演示
    @HystrixCommand(fallbackMethod =
    "payment_TimeoutHandler",commandProperties = {

        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="5
000") //5 秒钟以内就是正常的业务逻辑
    })
    @Override
    public String payment_Timeout(Integer id) {
        //int timeNumber = 3; //小于等于 3 秒算是正常情况
        int timeNumber = 15; //模拟非正常情况
        //int i = 1/0 ; //模拟非正常情况
        try {
            TimeUnit.SECONDS.sleep(timeNumber);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "线程池: "+Thread.currentThread().getName()+" payment_Timeout,id=" + id + " \t O(╥﹏╥)O 耗时: " + timeNumber;
    }

    //兜底方法，上面方法出问题,我来处理，返回一个出错信息
    public String payment_TimeoutHandler(Integer id) {
        return "线程池: "+Thread.currentThread().getName()+" payment_TimeoutHandler,系统繁忙,请稍后再试\t O(╥﹏╥)O ";
    }
}
```

```
}
```

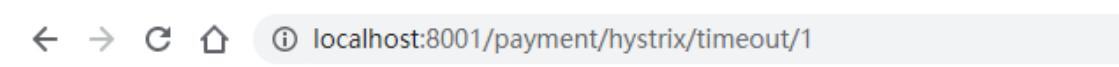
一旦调用服务方法失败并抛出了错误信息后，会自动调用@HystrixCommand 标注好的 fallbackMethod 调用类中的指定方法

主启动类激活

@EnableCircuitBreaker

测试超时和算数异常，都会走兜底方法——服务降级

<http://localhost:8001/payment/hystrix/timeout/1>



← → C ⌂ ⓘ localhost:8001/payment/hystrix/timeout/1

线程池：HystrixTimer-1 payment_TimeoutHandler, 系统繁忙，请稍后再试 o(╥﹏╥)o

4. 80fallback

1) 80 订单微服务，也可以更好的保护自己，自己也依样画葫芦进行客户端降级保护。注意：服务降级可以在服务提供者侧，也可以在服务消费者侧。更多是在服务消费者侧。

2) 题外话，切记

我们自己配置过的热部署方式对 java 代码的改动明显，但对@HystrixCommand 内属性的修改建议重启微服务

3) YML

```
feign:  
hystrix:  
enabled: true #如果处理自身的容错就开启。开启方式与生产端不一样。
```

4) 主启动

@EnableHystrix

5) 业务类：OrderHystrixController

```
//超时降级演示  
@HystrixCommand(fallbackMethod =  
"payment_TimeoutHandler", commandProperties = {  
  
@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1")  
})
```

```
500")//超过 1.5 秒就降级自己
})
@GetMapping("/consumer/payment/hystrix/timeout/{id}")
public String paymentInfo_TimeOut(@PathVariable("id") Integer id){
    //int age= 1/0;
    String result = paymentHystrixService.payment_Timeout(id);
    log.info("*****result:"+result);
    return result;
}

//兜底方法，上面方法出问题,我来处理，返回一个出错信息
public String payment_TimeoutHandler(Integer id) {
    return "我是消费者 80,对方支付系统繁忙请 10 秒后再试。或自己运行出错，请检查自己。";
}
```

6) 测试超时

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

我是消费者80,对方支付系统繁忙请10秒后再试。或自己运行出错，请检查自己。

5. 目前问题

每个业务方法对应一个兜底的方法，**代码膨胀，代码耦合**

统一通用处理和自定义独立处理的分开

6. 解决问题

1) 每个方法配置一个? ? ? 膨胀

feign 接口系列

@DefaultProperties(defaultFallback = "")

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

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

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

    public String payment_Global_FallbackMethod()
```

说明

@DefaultProperties(defaultFallback = "")

1:1 每个方法配置一个服务降级方法，技术上可以，但实际上傻 X

1:N 除了个别重要核心业务有专属，其它普通的可以通过

@DefaultProperties(defaultFallback = "")**统一跳转**到统一处理结果页面

通用的和独享的各自分开，避免了代码膨胀，合理减少了代码量

controller 配置

```
package com.atguigu.springcloud.controller;

import com.atguigu.springcloud.service.PaymentHystrixService;
import com.netflix.hystrix.contrib.javanica.annotation.DefaultProperties;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixProperty;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import javax.annotation.Resource;

@RestController
@Slf4j
@DefaultProperties(defaultFallback = "payment_Global_FallbackMethod") //全局的
public class OrderHystrixController {

    @Resource
    private PaymentHystrixService paymentHystrixService;

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

    @HystrixCommand
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id){
        int age = 10/0;
        String result = paymentHystrixService.paymentInfo_TimeOut(id);
        return result;
    }

    //兜底方法
    public String paymentTimeOutFallbackMethod(@PathVariable("id") Integer id){
        return "我是消费者 80， 对付支付系统繁忙请 10 秒钟后再试或者自己运行出错请检查
自己,(T_T)";
    }

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

测试结果

← → ⌛ ⌂ ⓘ localhost/consumer/payment/hystrix/timeout/1

Global异常处理信息，请稍后再试。(╥_╥)

2) 和业务逻辑混一起？？？混乱

服务降级，客户端去调用服务端，碰上服务端宕机或关闭

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

未来我们要面对的异常

- 运行
- 超时
- 宕机

再看我们的业务类 PaymentController

修改 cloud-consumer-feign-hystrix-order80

根据 cloud-consumer-feign-hystrix-order80 已经有的 PaymentHystrixService 接口，重新新建一个类 (PaymentFallbackService) 实现该接口，统一为接口里面的方法进行异常处理

PaymentFallbackService 类实现 PaymentFeignClientService 接口

```
package com.atguigu.springcloud.service;

import org.springframework.stereotype.Component;

@Component
public class PaymentFallbackService implements PaymentHystrixService {
    @Override
    public String paymentInfo_OK(Integer id) {
        return "-----PaymentFallbackService fall back-paymentInfo_OK , (╥_╥)";
    }

    @Override
```

```
public String payment_Timeout(Integer id) {  
    return "-----PaymentFallbackService fall back-paymentInfo_TimeOut , (╥_╥)";  
}  
}
```

YML

```
feign:  
  hystrix:  
    enabled: true #如果处理自身的容错就开启。开启方式与生产端不一样。
```

PaymentFeignClientService 接口

```
package com.atguigu.springcloud.service;  
  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.stereotype.Component;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
  
@Component  
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT",fallback = PaymentFallbackService.class)  
public interface PaymentHystrixService {  
  
    @GetMapping("/payment/hystrix/ok/{id}")  
    public String paymentInfo_OK(@PathVariable("id") Integer id);  
  
    @GetMapping("/payment/hystrix/timeout/{id}")  
    public String paymentInfo_TimeOut(@PathVariable("id") Integer id);  
}
```

3) 测试

单个 eureka 先启动 7001

PaymentHystrixMain8001 启动

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

故意关闭微服务 8001

客户端自己调用提升

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

7.3.7. 服务熔断

1. 断路器

一句话就是家里保险丝

2. 熔断是什么

熔断机制概述

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

当检测到该节点微服务调用响应正常后，恢复调用链路。

在 SpringCloud 框架里，熔断机制通过 Hystrix 实现。Hystrix 会监控微服务间调用的状态，当失败的调用到一定阈值，缺省是 **10 秒内 20 次** 调用并有 50% 的失败情况，就会**启动熔断机制**。熔断机制的注解是 @HystrixCommand

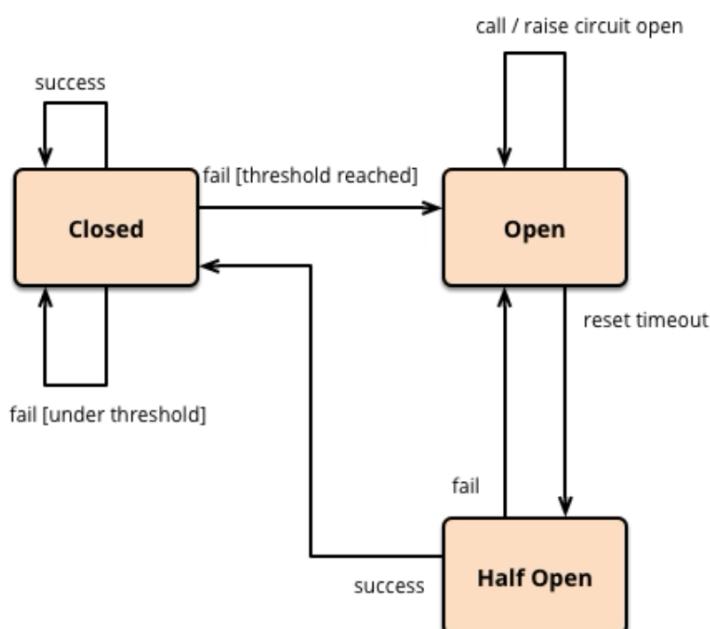
3. 大神论文：

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

← → ⌂ martinfowler.com/bliki/CircuitBreaker.html

end

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



4. 实操

1) 修改 cloud-provider-hystrix-payment8001

2) PaymentServiceImpl

com.netflix.hystrix.HystrixCommandProperties

```
//服务熔断
@HystrixCommand(fallbackMethod =
"paymentCircuitBreaker_fallback",commandProperties = {
    @HystrixProperty(name = "circuitBreaker.enabled",value = "true"), //是否开启断
路器
```

```
@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value =  
"10"), //当在配置时间窗口内达到此数量，打开断路，默认 20 个  
@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value =  
"10000"), //断路多久以后开始尝试是否恢复，默认 5s  
@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value =  
"60"), //出错百分比阈值，当达到此阈值后，开始短路。默认 50%  
}  
public String paymentCircuitBreaker(Integer id){  
    if (id < 0){  
        throw new RuntimeException("*****id 不能负数");  
    }  
    String serialNumber = IdUtil.simpleUUID(); //hutool.cn 工具包  
  
    return Thread.currentThread().getName()+"\t" + "调用成功,流水号: " +serialNumber;  
}  
public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id){  
    return "id 不能负数，请稍候再试(╥_╥)/~~~ id: " +id;  
}
```

3) PaymentController

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

4) 测试

自测 cloud-provider-hystrix-payment8001

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

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

一次正确一次错误 trytry

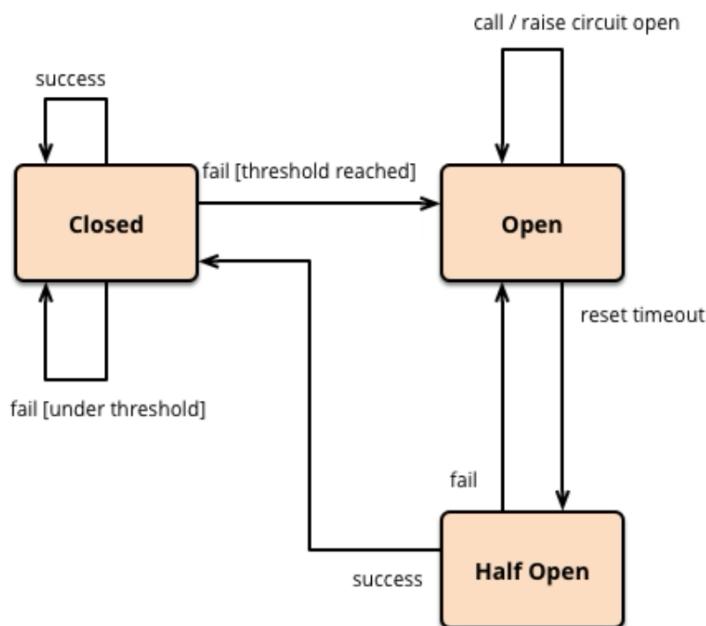
重点测试

多次错误（狂点），然后慢慢正确，发现刚开始不满足条件，就算是正确的访问地址也不能进行访问，需要慢慢的恢复链路

5. 原理（小总结）

1) 大神结论

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



2) 熔断类型

熔断打开

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

熔断关闭

熔断关闭不会对服务进行熔断

熔断半开

部分请求根据规则调用当前服务，如果请求成功且符合规则则认为当前服务恢复正常，
关闭熔断

3) 官网断路器流程图

① 官网步骤

② 断路器在什么情况下开始起作用

//服务熔断

```
@HystrixCommand(fallbackMethod =  
"paymentCircuitBreaker_fallback",commandProperties = {  
  
    @HystrixProperty(name = "circuitBreaker.enabled",value = "true"), //是否开启断路  
    器  
  
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value = "20"),  
    //当快照时间窗（默认 10 秒）内达到此数量才有资格打开断路， 默认 20 个  
  
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value =  
    "50000"), //断路多久以后开始尝试是否恢复， 默认 5s  
  
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value = "50"),  
    //出错百分比阈值，当达到此阈值后，开始短路。默认 50%  
})
```

涉及到断路器的三个重要参数：**快照时间窗、请求总数阈值、错误百分比阈值。**

配置属性参考：<https://github.com/Netflix/Hystrix/wiki/Configuration>

1、快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的 10 秒。

2、请求总数阈值：在快照时间窗内，必须满足请求总数阈值才有资格熔断。**默认 20**，意味着在 10 秒内，如果该 hystrix 命令的调用次数不足 20 次，即使所有的请求都超时或其他原因失败，断路器都不会打开。

3、错误百分比阈值：当请求总数在快照时间窗内超过了阈值，比如发生了 30 次调用，如果在这 30 次调用，有 15 次发生了超时异常，也就是超过 50% 的错误百分比，在**默认设定 50%**阈值情况下，这时候就会将断路器打开。

③ 断路器开启或者关闭的条件

当满足一定阈值的时候（默认 10 秒内超过 20 个请求次数）

当失败率达到一定的时候（默认 10 秒内超过 50% 请求失败）

到达以上阈值，断路器将会开启

当开启的时候，所有请求都不会进行转发

一段时间之后（默认是 5 秒），这个时候断路器是半开状态，会让其中一个请求进行转发。如果成功，断路器会关闭，若失败，继续开启。重复 4 和 5

④ 断路器打开之后

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

2：原来的主逻辑要如何恢复呢？

对于这一个问题，hystrix 也为我们实现了自动恢复功能。

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



让天下没有难学的技术

⑤ All 配置



All配置

```
//=====
//=====
@HystrixCommand(fallbackMethod = "str_fallbackMethod",
    groupKey = "strGroupCommand",
    commandKey = "strCommand",
    threadPoolKey = "strThreadPool",

    commandProperties = {
        // 设置隔离策略, THREAD 表示线程池 SEMAPHORE: 信号池隔离
        @HystrixProperty(name = "execution.isolation.strategy", value = "THREAD"),
        // 当隔离策略选择信号池隔离的时候, 用来设置信号池的大小 (最大并发数)
        @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests", value = "10"),
        // 配置命令执行的超时时间
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "10"),
        // 是否启用超时时间
        @HystrixProperty(name = "execution.timeout.enabled", value = "true"),
        // 执行超时的时候是否中断
        @HystrixProperty(name = "execution.isolation.thread.interruptOnTimeout", value = "true"),
        // 执行被取消的时候是否中断
        @HystrixProperty(name = "execution.isolation.thread.interruptOnCancel", value = "true"),
        // 允许回调方法执行的最大并发数
        @HystrixProperty(name = "fallback.isolation.semaphore.maxConcurrentRequests", value = "10"),
        // 服务降级是否启用, 是否执行回调函数
        @HystrixProperty(name = "fallback.enabled", value = "true"),
        // 该属性用来设置在滚动时间窗中, 断路器熔断的最小请求数。例如, 默认该值为 20 的时候,
        // 如果滚动时间窗(默认10秒)内仅收到了19个请求, 即使这19个请求都失败了, 断路器也不会打开。
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "20"),
        // 该属性用来设置在滚动时间窗中, 表示在滚动时间窗中, 在请求数量超过
        // circuitBreaker.requestVolumeThreshold 的情况下, 如果错误请求数的百分比超过50,
        // 就把断路器设置为“打开”状态, 否则就设置为“关闭”状态。
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "50"),
        // 该属性用来设置当断路器打开之后的休眠时间窗。休眠时间窗结束后,
        // 会将断路器置为“半开”状态, 尝试熔断的请求命令, 如果依然失败就将断路器继续设置为“打开”状态,
        // 如果成功就设置为“关闭”状态。
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "5000"),
        // 断路器强制打开
        @HystrixProperty(name = "circuitBreaker.forceOpen", value = "false"),
        // 断路器强制关闭
        @HystrixProperty(name = "circuitBreaker.forceClosed", value = "false"),
        // 滚动时间窗设置, 该时间用于断路器判断健康度时需要收集信息的持续时间
        @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value = "10000"),
        // 该属性用来设置滚动时间窗统计指标信息时划分“桶”的数量, 断路器在收集指标信息的时候会根据
        // 设置的时间窗长度拆分成多个“桶”来累计各度量值, 每个“桶”记录了一段时间内的采集指标。
        // 比如 10 秒内拆分成 10 个“桶”收集这样, 所以 timeInMilliseconds 必须能被 numBuckets 整除。否则会抛异常
        @HystrixProperty(name = "metrics.rollingStats.numBuckets", value = "10"),
        // 该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算。如果设置为 false, 那么所有的概要统计都
        @HystrixProperty(name = "metrics.rollingPercentile.enabled", value = "false"),
        // 该属性用来设置百分位统计的滚动窗口的持续时间, 单位为毫秒。
        @HystrixProperty(name = "metrics.rollingPercentile.timeInMilliseconds", value = "60000"),
        // 该属性用来设置百分位统计滚动窗口中使用“桶”的数量。
        @HystrixProperty(name = "metrics.rollingPercentile.numBuckets", value = "60000"),
        // 该属性用来设置在执行过程中每个“桶”中保留的最大执行次数。如果在滚动时间窗内发生超过该设定值的执行,
        // 就从最初的位置开始重写。例如, 将该值设置为100, 滚动窗口为10秒, 若在10秒内一个“桶”中发生了500次执行,
        // 那么该“桶”中只保留最后的100次执行的统计。另外, 增加该值的大小将会增加内存量的消耗, 并增加排序百分位数
        @HystrixProperty(name = "metrics.rollingPercentile.bucketSize", value = "100"),
        // 该属性用来设置采集影响断路器状态的健康快照(请求的成功、错误百分比)的间隔等待时间。
        @HystrixProperty(name = "metrics.healthSnapshot.intervalInMilliseconds", value = "500"),
        // 是否开启请求缓存
        @HystrixProperty(name = "requestCache.enabled", value = "true"),
        // HystrixCommand的执行和事件是否打印日志到 HystrixRequestLog 中
        @HystrixProperty(name = "requestLog.enabled", value = "true"),
    },
    threadPoolProperties = {
        // 该参数用来设置执行命令线程池的核心线程数, 该值也就是命令执行的最大并发量
        @HystrixProperty(name = "coreSize", value = "10"),
        // 该参数用来设置线程池的最大队列大小。当设置为 -1 时, 线程池将使用 SynchronousQueue 实现的队列,
        // 否则将使用 LinkedBlockingQueue 实现的队列。
        @HystrixProperty(name = "maxQueueSize", value = "-1")
```

7.3.8. 服务限流

后面讲

7.4. 服务监控 hystrixDashboard

7.4.1. 概述

除了隔离依赖服务的调用以外，Hystrix 还提供了准实时的调用监控(Hystrix Dashboard)，Hystrix 会持续地记录所有通过 Hystrix 发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求多少成功，多少失败等。Netflix 通过 hystrix-metrics-event-stream 项目实现了对以上指示的监控。Spring Cloud 也提供了 Hystrix Dashboard 的整合，对监控内容转化成可视化界面。

7.4.2. 仪表盘 9001

1. 新建 Module: cloud-consumer-hystrix-dashboard9001

2. POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
```

```
<modelVersion>4.0.0</modelVersion>
<artifactId>cloud-consumer-hystrix-dashboard9001</artifactId>

<dependencies>
    <!--新增 hystrix dashboard-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <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>
    </dependency>
</dependencies>
</project>
```

3. YML

```
server:
  port: 9001
hystrix:
  dashboard:
    proxy-stream-allow-list: "*"
```

4. HystrixDashboardMain9001+新注解

@EnableHystrixDashboard

```
package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

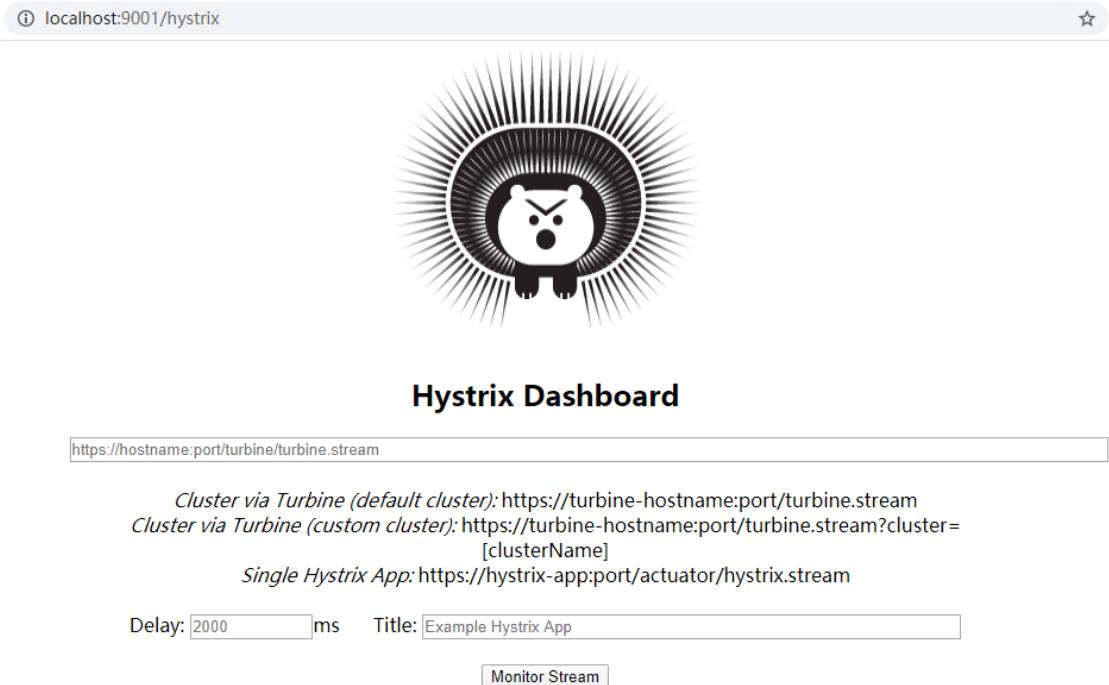
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardMain9001 {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardMain9001.class,args);
    }
}
```

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

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

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

<http://localhost:9001/hystrix>



7.4.3. 断路器演示

1. 修改 cloud-provider-hystrix-payment8001

注意：新版本 Hystrix 需要在主启动类 MainAppHystrix8001 中指定监控路径

```
/**  
 *此配置是为了服务监控而配置，与服务容错本身无关，springcloud 升级后的坑  
 *ServletRegistrationBean 因为 springboot 的默认路径不是"/hystrix.stream"，  
 *只要在自己的项目里配置上下面的 servlet 就可以了  
 */  
@Bean  
public ServletRegistrationBean getServlet() {  
    HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();  
    ServletRegistrationBean registrationBean = new  
    ServletRegistrationBean(streamServlet);  
    registrationBean.setLoadOnStartup(1);  
    registrationBean.addUrlMappings("/hystrix.stream");  
    registrationBean.setName("HystrixMetricsStreamServlet");  
    return registrationBean;  
}
```

Unable to connect to Command Metric Stream

404

2. 监控测试

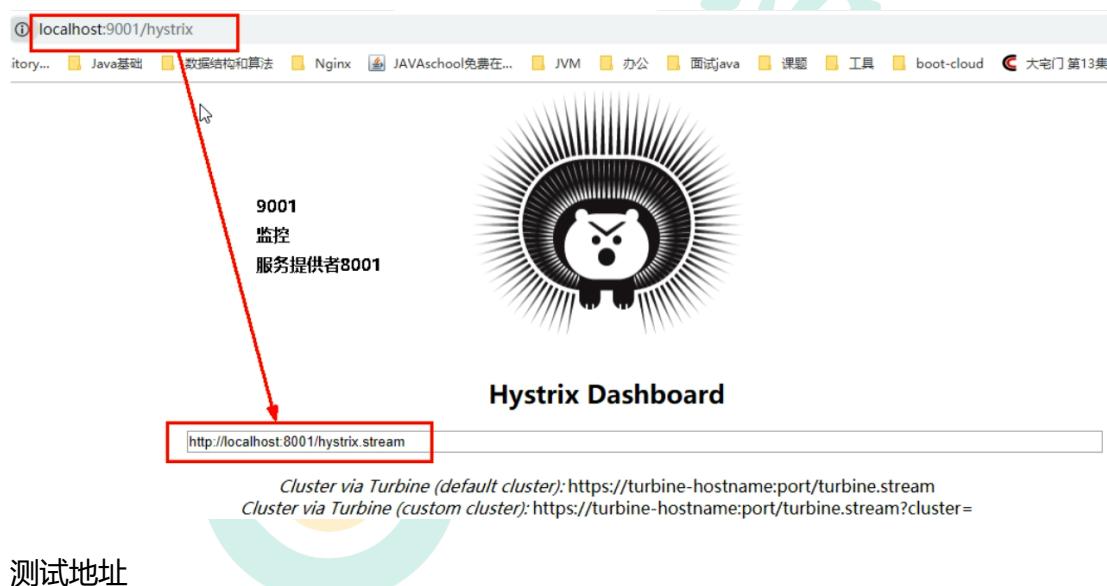
1) 启动 1 个 eureka

2) 观察监控窗口

9001 监控 8001

<http://localhost:9001/hystrix>

<http://localhost:8001/hystrix.stream>



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

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

上述测试通过

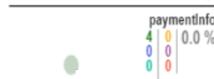
ok

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

Hystrix Stream: T1

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %


 Host: 0.4/s
 Cluster: 0.4/s
 Circuit Closed

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

Thread Pools Sort: Alphabetical | Volume |

 Host: 0.4/s
 Cluster: 0.4/s

 Active 0 Max Active 1
 Queued 0 Executions 4
 Pool Size 10 Queue Size 5

监控结果，失败
localhost:9001/hystrix/monitor?stream=http%3A%2F%2Flocalhost%3A8001%2Fhystrix.stream&delay=2000&title=T3

Hystrix Stream: T3


Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %


 Host: 0.0/s
 Cluster: 0.0/s

 Hosts 1 90th 1ms
 Median 0ms 99th 2ms
 Mean 0ms 99.5th 2ms

Thread Pools Sort: Alphabetical | Volume |

 Host: 0.0/s
 Cluster: 0.0/s

 Active 0 Max Active 0
 Queued 10 Executions 0
 Pool Size 5 Queue Size 5

如何看
7色

Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %

1圈

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

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

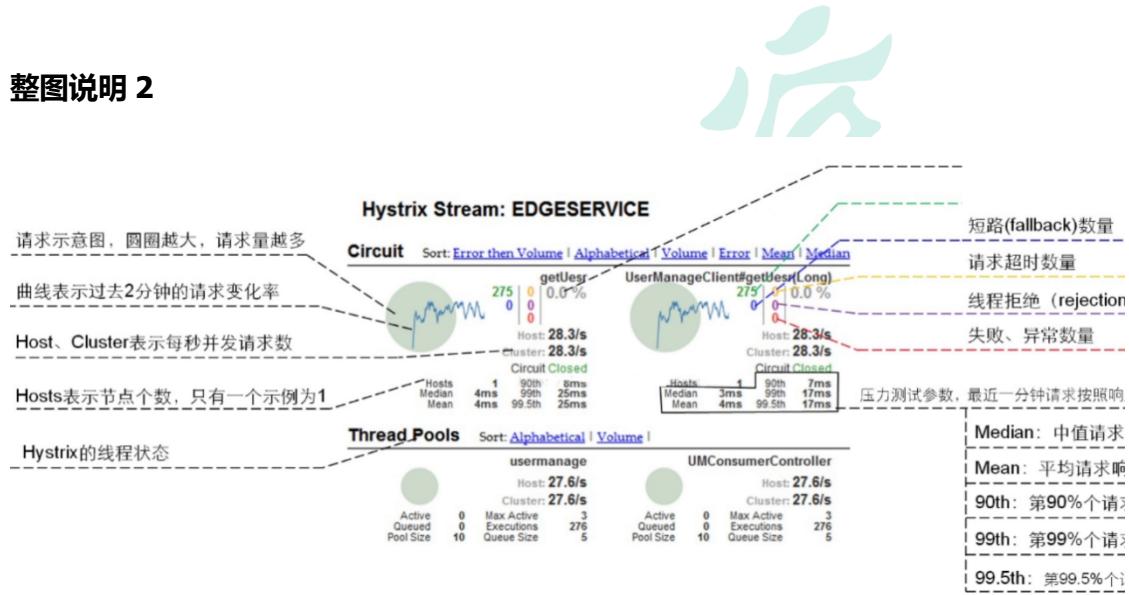
1线

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

整图说明



整图说明 2



搞懂一个才能看懂复杂的



8. Gateway 新一代网关

8.1. 概述简介

8.1.1. 官网

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.6.RELEASE/reference/html/>

The screenshot shows a browser window displaying the Spring Cloud Gateway documentation. The left sidebar contains a table of contents with 15 items, including 'How to Include Spring Cloud Gateway'. The main content area is titled '2.2.6.RELEASE' and discusses the project's aims and security features. It includes two warning boxes about the gateway's dependencies and runtime requirements.

Table of Contents:

- 1. How to Include Spring Cloud Gateway
- 2. Glossary
- 3. How It Works
- 4. Configuring Route Predicate Factories and Gateway Filter Factories
- 5. Route Predicate Factories
- 6. GatewayFilter Factories
- 7. Global Filters
- 8. HttpHeadersFilters
- 9. TLS and SSL
- 10. Configuration
- 11. Route Metadata Configuration
- 12. Http timeouts configuration
- 13. Reactor Netty Access Logs
- 14. CORS Configuration
- 15. Actuator API

2.2.6.RELEASE

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2.2.6.RELEASE, and Spring Cloud 2.2.6.RELEASE. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns such as: security, monitoring/metrics, and resiliency.

1. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project, use the starter with a group ID of `org.springframework.cloud` and artifact ID `spring-cloud-starter-gateway`. See the [Spring Cloud Project page](#) for details on setting up your build for the Spring Cloud Release Train.

If you include the starter, but you do not want the gateway to be enabled, set `spring.cloud.gateway.enabled=false`.

! Spring Cloud Gateway is built on [Spring Boot 2.x](#), [Spring WebFlux](#), and [Project Reactor](#). As a consequence, it depends on many of the same familiar synchronous libraries (Spring Data and Spring Security, for example) and patterns you would expect from using Spring Boot. If you are unfamiliar with these projects, we suggest you take a look at their documentation to familiarize yourself with some of the new concepts before working with Spring Cloud Gateway.

! Spring Cloud Gateway requires the Netty runtime provided by Spring Boot and Spring Webflux. It does not work with a traditional Servlet Container or when built as a WAR.

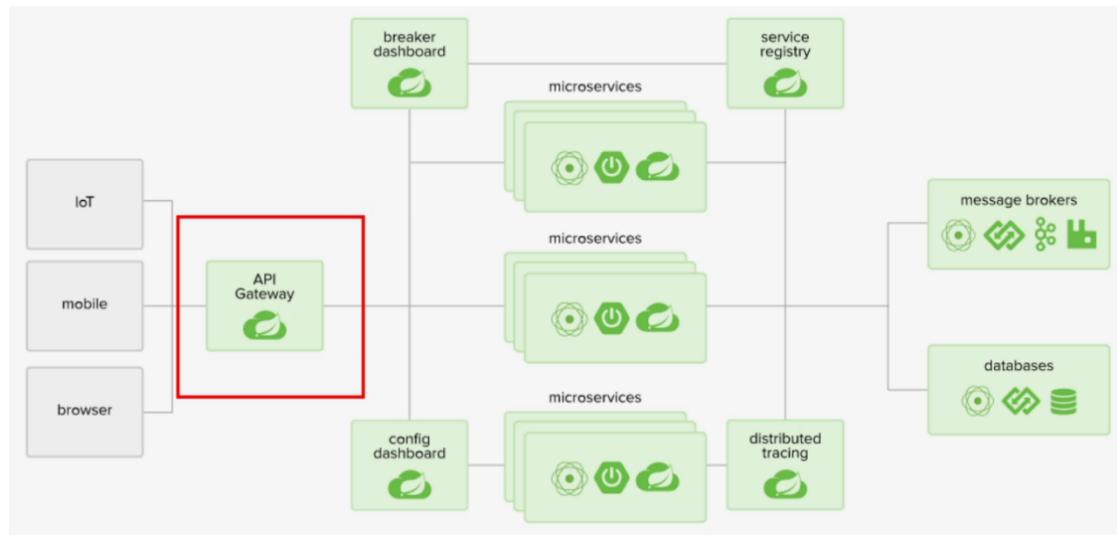
8.1.2. 是什么

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

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

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代替 Zuul，那就是 SpringCloud Geteway；

8.1.2.1. 一句话：Geteway 是原 Zuul1.x 版的替代



8.1.2.2. 概述

Gateway 是在 spring 生态系统之上构建的 API 网关服务，基于 Spring5，SpringBoot2 和 Project Reactor 等技术。

Gateway 旨在提供一种简单而有效的方式来对 API 进行**路由**，以及提供一些强大的**过滤器**功能，例如：**熔断、限流、重试**等

SpringCloud Gateway 是 SpringCloud 的一个全新项目，**基于 Spring5.X+SpringBoot2.X 和 Project Reactor 等技术**开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

为了提升网关的性能，SpringCloud Gatway 是基于 WebFlux 框架实现的，而 **WebFlux 框架底层则使用了高性能的 Reactor 模式通讯框架 Netty**。

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

8.1.2.3. 一句话

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

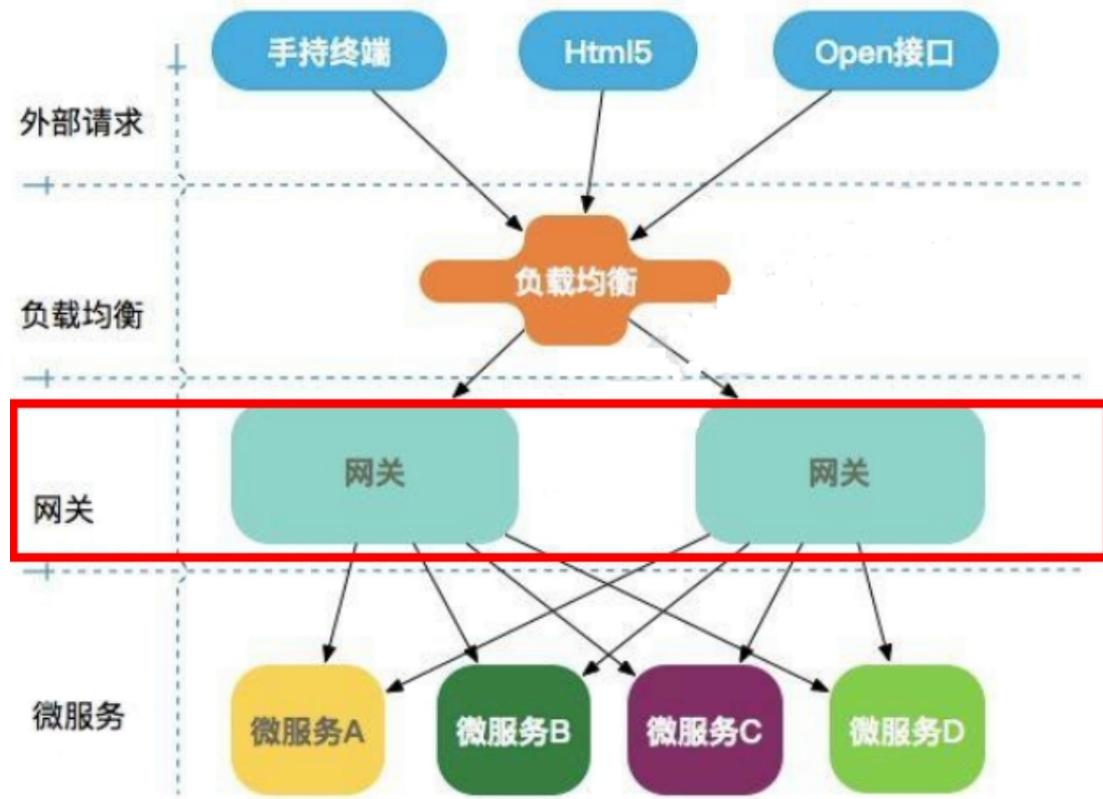
源码架构

- ✓  cloud-gateway-gateway9527
 -  Lifecycle
 -  Plugins
 - ✓  Dependencies
 - ✓  org.springframework.cloud:spring-cloud-starter-gateway:2.2.6.RELEASE
 -  org.springframework.cloud:spring-cloud-starter:2.2.6.RELEASE
 -  org.springframework.cloud:spring-cloud-gateway-server:2.2.6.RELEASE
 - ✓  org.springframework.boot:spring-boot-starter-webflux:2.3.6.RELEASE
 -  org.springframework.boot:spring-boot-starter:2.3.6.RELEASE (omitted for duplicate)
 -  org.springframework.boot:spring-boot-starter-json:2.3.6.RELEASE
 - ✓  org.springframework.boot:spring-boot-starter-reactor-netty:2.3.6.RELEASE
 - ✓  io.projectreactor.netty:reactor-netty:0.9.14.RELEASE
 -  io.netty:netty-codec-http:4.1.54.Final
 -  io.netty:netty-codec-http2:4.1.54.Final
 -  io.netty:netty-handler:4.1.54.Final
 -  io.netty:netty-handler-proxy:4.1.54.Final
 -  io.netty:netty-transport-native-epoll:linux-x86_64:4.1.54.Final
 -  io.projectreactor:reactor-core:3.3.11.RELEASE (omitted for duplicate)
 -  org.springframework:spring-web:5.2.11.RELEASE
 -  org.springframework:spring-webflux:5.2.11.RELEASE
 -  org.synchronoss.cloud:nio-multipart-parser:1.1.0

8.1.3. 能干嘛

- 反向代理
- 鉴权
- 流量控制
- 熔断
- 日志监控
-

8.1.4. 微服务架构中网关在哪里



8.2. 三大核心概念

8.2.1. 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

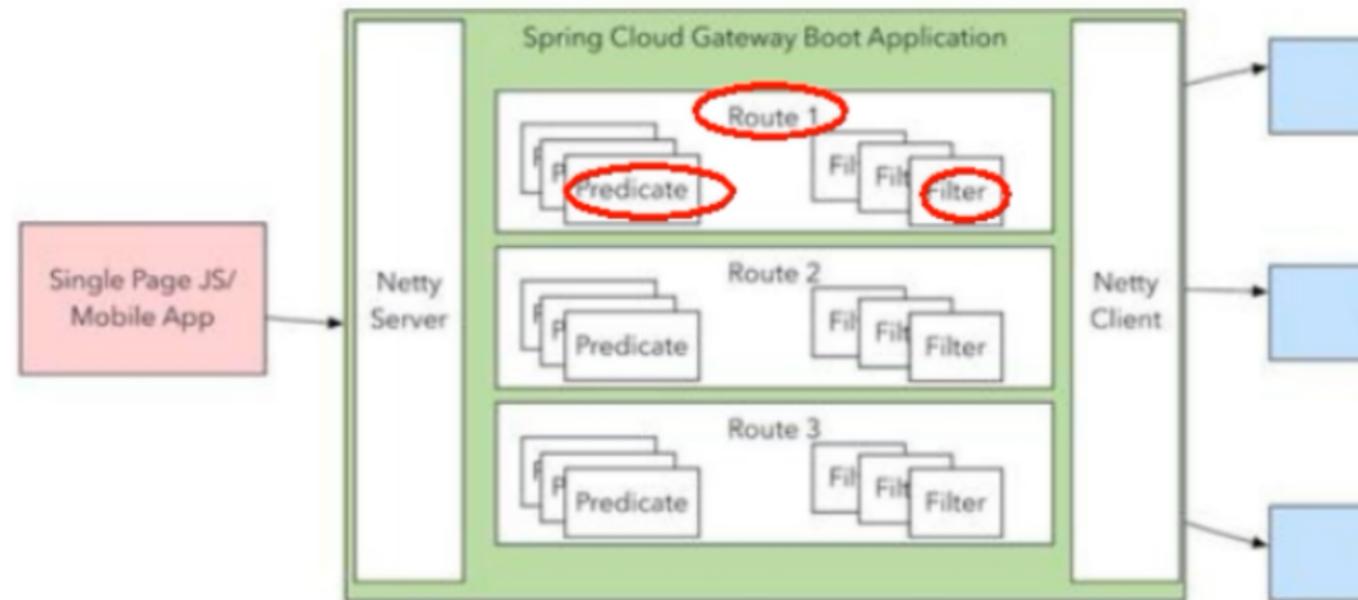
8.2.2. Predicate (断言)

参考的是 java8 的 `java.util.function.Predicate` 开发人员可以匹配 HTTP 请求中的所有内容（例如请求头或请求参数），如果请求与断言相匹配则进行路由

8.2.3. Filter(过滤)

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

8.2.4. 总体



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

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

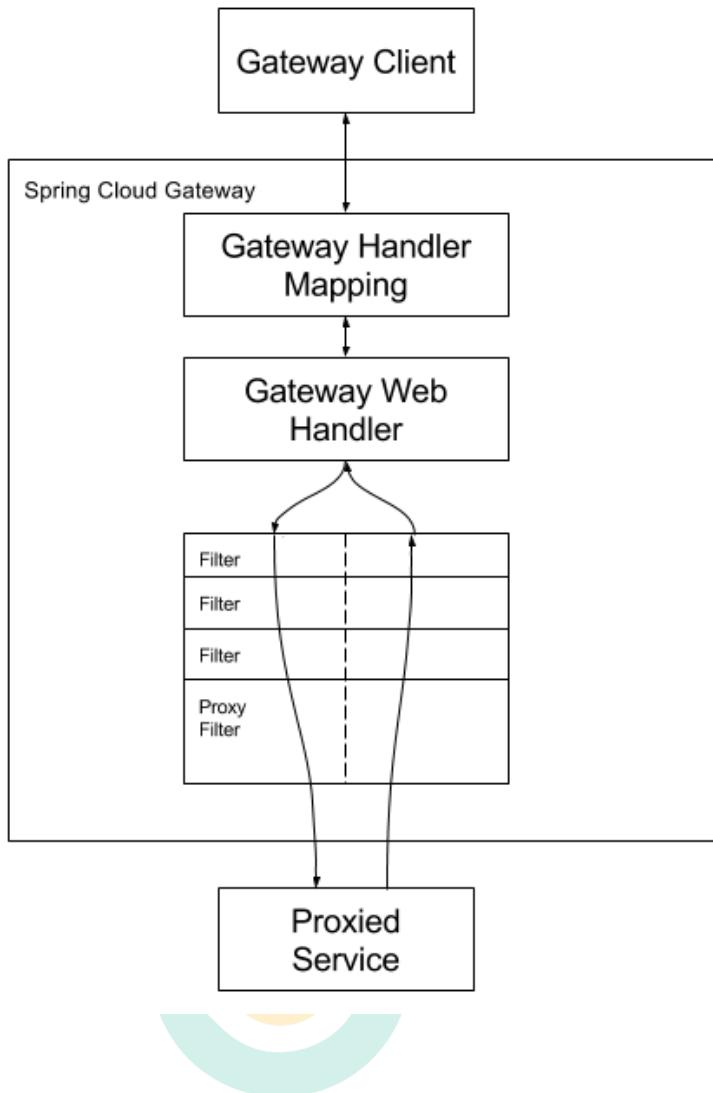
8.3. Gateway 工作流程

8.3.1. 官网总结

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.6.RELEASE/reference/html/>

3. How It Works

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



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request 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 made. After the proxy request is made, the "post" filter logic is executed.

客户端向 Spring Cloud Gateway 发出请求。然后在 Gateway Handler Mapping 中找到与请求匹配的路由，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给我们实际的服务执行业务逻辑，然后返回。

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

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

8.3.2. 核心逻辑：路由转发+执行过滤器链

8.4. 入门配置

8.4.1. 新建 Module：cloud-gateway-

gateway9527

8.4.2. POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloud-gateway-gateway9527</artifactId>

    <dependencies>
        <!--新增 gateway，不需要引入 web 和 actuator 模块-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
```

```
</dependency>
<dependency>
    <groupId>com.atguigu</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<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>
</dependency>
</dependencies>
</project>
```

8.4.3. YML

```
server:
  port: 9527
spring:
```

```
application:  
  name: cloud-gateway  
  
eureka:  
  instance:  
    hostname: cloud-gateway-service  
  client:  
    register-with-eureka: true  
    fetch-registry: true  
    service-url:  
      defaultZone: http://localhost:7001/eureka
```

8.4.4. 业务类

无

8.4.5. 主启动类

```
package com.atguigu.springcloud;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
@SpringBootApplication  
@EnableEurekaClient  
public class GateWayMain9527 {  
    public static void main(String[] args) {  
        SpringApplication.run(GateWayMain9527.class, args);  
    }  
}
```

8.4.6. 9527 网关如何做路由映射呢？？？

我们目前不想暴露 8001 端口，希望在 8001 外面套一层 9527

8.4.7. YML 新增网关配置

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      routes:
        - id: payment_routh #路由的 ID, 没有固定规则但要求唯一, 建议配合服务名
          uri: http://localhost:8001 #匹配后提供服务的路由地址
          predicates:
            - Path=/payment/get/** #断言,路径相匹配的进行路由

        - id: payment_routh2
          uri: http://localhost:8001
          predicates:
            - Path=/payment/lb/** #断言,路径相匹配的进行路由

eureka:
  instance:
    hostname: cloud-gateway-service
  client:
    service-url:
      register-with-eureka: true
      fetch-registry: true
      defaultZone: http://localhost:7001/eureka
```

8.4.8. 测试

启动 7001: cloud-eureka-server7001

启动 8001: cloud-provider-payment8001

启动 9527 网关: cloud-gateway-gateway9527

访问说明

The screenshot shows two code editors side-by-side. On the left is the `application.yml` configuration file, and on the right is the `PaymentController.java` Java code.

application.yml:

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      routes:
        - id: payment_routh #payment_route
          uri: http://localhost:8001
          predicates:
            - Path=/payment/get/**
```

PaymentController.java:

```
int result = paymentService.create(payment);
Log.info("*****插入操作返回结果:" + result);

if(result > 0)
{
  return new CommonResult( code: 200, message: "成功")
} else {
  return new CommonResult( code: 444, message: "失败")
}

@GetMapping(value = "/payment/get/{id}")
public CommonResult<Payment> getPaymentById(@PathVariable("id") Integer id) {
  Payment payment = paymentService.getPaymentById(id);
  Log.info("*****查询结果:{}" , payment);
  if (payment != null) {
    return new CommonResult( code: 200, message: "成功")
  } else {
    return new CommonResult( code: 444, message: "失败")
  }
}
```

A red box highlights the `Path=/payment/get/**` entry in the `application.yml` routes section, and another red box highlights the `@GetMapping` annotation in the `PaymentController.java` code.

添加网关前: <http://localhost:8001/payment/get/31>

添加网关后: <http://localhost:9527/payment/get/31>

8.5. 通过微服务名实现动态路由

默认情况下 Gateway 会根据注册中心的服务列表，以注册中心上微服务名为路径创建动态路由进行转发，从而实现动态路由的功能

8.5.1. 启动

一个 eureka7001+两个服务提供者 8001/8002

8.5.2. POM

8.5.3. YML

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
```

```
discovery:  
  locator:  
    enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由  
  routes:  
    - id: payment_routh #路由的 ID，没有固定规则但要求唯一，建议配合服务名  
      #uri: http://localhost:8001 #匹配后提供服务的路由地址  
      uri: lb://cloud-payment-service  
      predicates:  
        - Path=/payment/get/** #断言,路径相匹配的进行路由  
  
    - id: payment_routh2  
      #uri: http://localhost:8001 #匹配后提供服务的路由地址  
      uri: lb://cloud-payment-service  
      predicates:  
        - Path=/payment/lb/** #断言,路径相匹配的进行路由  
  
eureka:  
  instance:  
    hostname: cloud-gateway-service  
  client:  
    service-url:  
      register-with-eureka: true  
      fetch-registry: true  
      defaultZone: http://localhost:7001/eureka
```

需要注意的是 uri 的协议为 lb，表示启用 Gateway 的负载均衡功能。

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

8.5.4. 测试

<http://localhost:9527/payment/lb>

8001/8002 两个端口切换

8.6. Predicate 的使用

8.6.1. 是什么

启动我们的 gateway9527，查看启动日志

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

8.6.2. Route Predicate Factories 这个是什么东东？

The screenshot shows a browser window displaying the Spring Cloud Gateway documentation at cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.0.RELEASE/reference/html/#gateway-request-predicates-factories. The page is titled '5. Route Predicate Factories'. A red box highlights this title. Below it, a box contains the text: '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.' Another red box highlights the '5.1. The After Route Predicate Factory' section. This section describes the 'After' route predicate factory, which matches requests after a specified datetime. An example application.yml configuration is shown:

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

Spring Cloud Gateway 将路由匹配作为 Spring WebFlux HandlerMapper 基础框架的一部分。

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

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

所有这些谓词都匹配 HTTP 请求的不同属性。多种谓词工厂可以组合，并通过逻辑 and 。

8.6.3. 常用的 Route Predicate

1. After Route Predicate



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]
```

ZonedDateTime zonedDateTime = ZonedDateTime.now();

System.out.println(zonedDateTime);

▪ - After=2020-03-
08T10:59:34.102+08:00[Asia/Shanghai]

• 测试：没到时间进行测试报错

← → C ⌂ ⓘ localhost:9527/payment/lb

Whitelabel Error Page

This application has no configured error view, so you are seeing this as a fallback.

Wed Apr 08 14:13:40 CST 2020
[07f798df] There was an unexpected error (type=Not Found, status=404).
org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND
at org.springframework.web.reactive.resource.ResourceWebHandler.lambda\$handle\$0(ResourceWebHandler.java:325)
Suppressed: reactor.core.publisher.FluxOnAssembly\$OnAssemblyException:
Error has been observed at the following site(s):
|_ checkpoint ⇢ org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]
|_ checkpoint ⇢ HTTP GET "/payment/lb" [ExceptionHandlingWebHandler]
Stack trace:
at org.springframework.web.reactive.resource.ResourceWebHandler.lambda\$handle\$0(ResourceWebHandler.java:325)
at org.springframework.web.reactive.resource.ResourceWebHandler\$\$Lambda\$720/1701101299.get(Unknown Source)

2.Before Route Predicate

- YML
 - - Before=2020-03-08T10:59:34.102+08:00[Asia/Shanghai]

3.Between Route Predicate

- YML
 - - Between=2020-03-08T10:59:34.102+08:00[Asia/Shanghai] , 2020-03-08T10:59:34.102+08:00[Asia/Shanghai]

4.Cookie Route Predicate

5.4. The Cookie Route Predicate Factory

The `Cookie` route predicate factory takes two parameters, the cookie `name` and a `regexp` (which is a Java regular expression). This predicate matches cookies that have the given name and whose values match the regular expression. The following example configures a cookie route predicate factory:

Example 4. application.yml

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

YAML

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

- YML
 - - Cookie=username,atguigu #并且 Cookie 是 username=atguigu 才能访问
- 不带 cookies 访问

```
D:\Server\curl-7.69.1-win64-mingw\bin>curl http://localhost:9527/payment/lb  
8002
```

- 带上 cookies 访问
 - curl 下载地址: <https://curl.haxx.se/download.html>

```
D:\Server\curl-7.69.1-win64-mingw\bin>curl http://localhost:9527/payment/lb --coo  
8002  
D:\Server\curl-7.69.1-win64-mingw\bin>curl http://localhost:9527/payment/lb --coo  
{"timestamp":"2020-04-08T06:33:37.192+0000","path":"/payment/lb","status":404,"er  
amework.web.server.ResponseStatusException: 404 NOT_FOUND\r\n\tat org.springframework.  
er.java:325)\r\n\tSuppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyExc  
...\r\n\tat org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebF  
ebHandler]\nStack trace:\r\n\tat org.springframework.web.reactive.resource.Reso
```

- 加入 curl 返回中文乱码（帖子）：

<https://blog.csdn.net/leedee/article/details/82685636>

5.Header Route Predicate

5.5. The Header Route Predicate Factory

The `Header` route predicate factory takes two parameters, the header `name` and a `regexp` (which is a Java regular expression). This predicate matches with a header that has the given name whose value matches the regular expression. The following example configures a header route predicate:

Example 5. application.yml

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

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

- YML
 - Header=X-Request-Id, \d+ #请求头中要有 X-Request-Id 属性并且值为整数的正则表达式

```
D:\Server\curl-7.69.1-win64-mingw\bin>curl http://localhost:9527/payment/lb -H "X-Request-Id:123"  
8001  
D:\Server\curl-7.69.1-win64-mingw\bin>curl http://localhost:9527/payment/lb -H "X-Request-Id:123"  
{"timestamp":"2020-04-08T06:42:30.791+0000","path":"/payment/lb","status":404,"error": "Not Found", "message":null, "requestId": "f24e3f0f", "trace": "org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND\r\n\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\tSuppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: \nError has been observed at the following site(s):\n\tin [lCheckpoint] \n\tat org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]\n\tin [lCheckpoint] \n\t\tHTTP GET '/payment/lb' [ExceptionHandlingWebHandler]\nStack trace:\r\n\tat org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]\n\tin [lCheckpoint] \n\t\tHTTP GET '/payment/lb' [ExceptionHandlingWebHandler]\n\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\tin [lCheckpoint] \n\t\tget(Unknown Source)\r\n\tat reactor.core.publisher.MonoRefer.subscribe(MonoRefer.
```

6.Host Route Predicate

YML: - Host=**.atguigu.com

7.Method Route Predicate

YML: - Method=GET

8.Path Route Predicate

YML:

9. Query Route Predicate

YML: - Query=username, \d+ #要有参数名称并且是正整数才能路由

10.小总结

- All

```
server:  
port: 9527  
  
spring:  
application:  
name: cloud-gateway  
cloud:  
gateway:  
discovery:  
locator:  
enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由  
routes:  
- id: payment_routh #路由的 ID，没有固定规则但要求唯一，建议配合服务名  
#uri: http://localhost:8001 #匹配后提供服务的路由地址  
uri: lb://cloud-payment-service  
predicates:  
- Path=/payment/get/** #断言,路径相匹配的进行路由  
  
- id: payment_routh2  
#uri: http://localhost:8001 #匹配后提供服务的路由地址  
uri: lb://cloud-payment-service  
predicates:
```

```
- Path=/payment/lb/** #断言,路径相匹配的进行路由
 #- After=2020-03-08T10:59:34.102+08:00[Asia/Shanghai]
 #- Cookie=username,zhangshuai #并且 Cookie 是 username=zhangshuai 才能访问
 #- Header=X-Request-Id, \d+ #请求头中要有 X-Request-Id 属性并且值为整数的正则表达式
 #- Host=*.atguigu.com
 #- Method=GET
 #- Query=username, \d+ #要有参数名称并且是正整数才能路由

eureka:
instance:
hostname: cloud-gateway-service
client:
service-url:
register-with-eureka: true
fetch-registry: true
defaultZone: http://localhost:7001/eureka
```

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

8.7. Filter 的使用

8.7.1. 是什么

路由过滤器可用于修改进入的 HTTP 请求和返回的 HTTP 响应，路由过滤器只能指定路由进行使用。

SpringCloud Gateway 内置了多种路由过滤器，他们都由 `GatewayFilter` 的工厂类来产生。

8.7.2. Spring Cloud Gateway 的 Filter

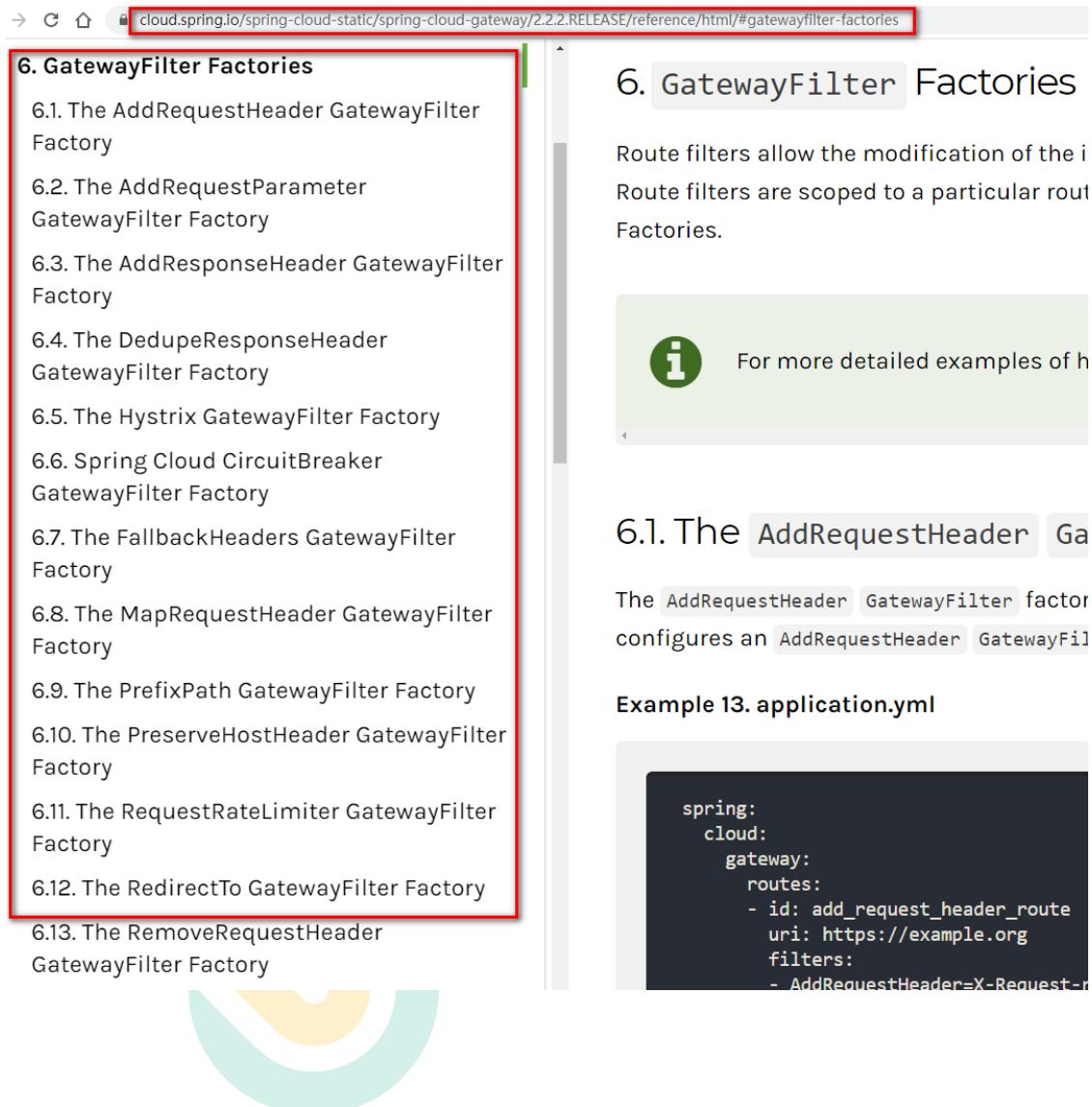
1. 生命周期, Only Two

- pre
 - 在业务逻辑之前
- post
 - 在业务逻辑之后

2. 种类, Only Two

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.6.RELEASE/reference/html/>

1) GatewayFilter (31种之多)



→ C ⌂ cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.2.RELEASE/reference/html/#gatewayfilter-factories

6. GatewayFilter Factories

- 6.1. The AddRequestHeader GatewayFilter Factory
- 6.2. The AddRequestParamter GatewayFilter Factory
- 6.3. The AddResponseHeader GatewayFilter Factory
- 6.4. The DedupeResponseHeader GatewayFilter Factory
- 6.5. The Hystrix GatewayFilter Factory
- 6.6. Spring Cloud CircuitBreaker GatewayFilter Factory
- 6.7. The FallbackHeaders GatewayFilter Factory
- 6.8. The MapRequestHeader GatewayFilter Factory
- 6.9. The PrefixPath GatewayFilter Factory
- 6.10. The PreserveHostHeader GatewayFilter Factory
- 6.11. The RequestRateLimiter GatewayFilter Factory
- 6.12. The RedirectTo GatewayFilter Factory
- 6.13. The RemoveRequestHeader GatewayFilter Factory

6. GatewayFilter Factories

Route filters allow the modification of the i
Route filters are scoped to a particular rout
Factories.

 For more detailed examples of h

6.1. The AddRequestHeader Ga

The `AddRequestHeader` `GatewayFilter` factor
configures an `AddRequestHeader` `GatewayFil`

Example 13. application.yml

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

2) GlobalFilter

7. Global Filters

- 7.1. Combined Global Filter and GatewayFilter Ordering
- 7.2. Forward Routing Filter
- 7.3. The LoadBalancerClient Filter
- 7.4. The ReactiveLoadBalancerClientFilter
- 7.5. The Netty Routing Filter
- 7.6. The Netty Write Response Filter
- 7.7. The RouteToRequestUrl Filter
- 7.8. The Websocket Routing Filter
- 7.9. The Gateway Metrics Filter
- 7.10. Marking An Exchange As Routed

8.7.3. 常用的 GatewayFilter

AddRequestParamter

YML

```
spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true #开启从注册中心动态创建路由的功能
      lower-case-service-id: true #使用小写服务名，默认是大写
    routes:
      - id: payment_routh #payment_route #路由的ID，没有固定规则但要求唯一，建议配合服务名
    uri: lb://cloud-provider-payment #匹配后的目标服务地址，供服务的路由地址
    #uri: http://localhost:8001 #匹配后提供服务的路由地址
    filters:
      - AddRequestParameter=X-Request-Id,1024 #过滤器工厂会在匹配的请求头加上一对请求头
    predicates:
      - Path=/paymentInfo/**           # 断言，路径相匹配的进行路由
      - Method=GET,POST
```

```
eureka:
  instance:
    hostname: cloud-gateway-service
```

省略

...

8.7.4. 自定义过滤器

1. 自定义全局 GlobalFilter

两个主要接口介绍

implements GlobalFilter , Ordered

2. 能干嘛

全局日志记录

统一网关鉴权

.....

3. 案例代码

```
package com.atguigu.springcloud.filter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;
import java.util.Date;

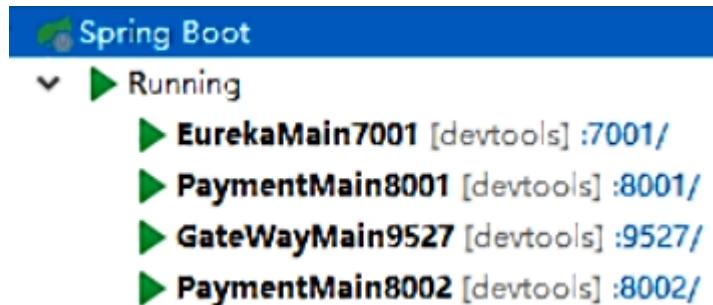
@Component
@Slf4j
public class MyLogGateWayFilter implements GlobalFilter,Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        log.info("*****come in MyLogGateWayFilter: " + new Date());
        String uname = exchange.getRequest().getQueryParams().getFirst("username");
        if(StringUtils.isEmpty(uname)){
            log.info("*****用户名为 Null 非法用户,(ㄒ_ㄒ)");
            exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
            return exchange.getResponse().setComplete();
        }
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return 0;
    }
}
```

4. 测试

启动



正确: <http://localhost:9527/payment/lb?username=z3>

错误: <http://localhost:9527/payment/lb?uname=z3>

9. SpringCloud Sleuth 分布式链路请求跟踪

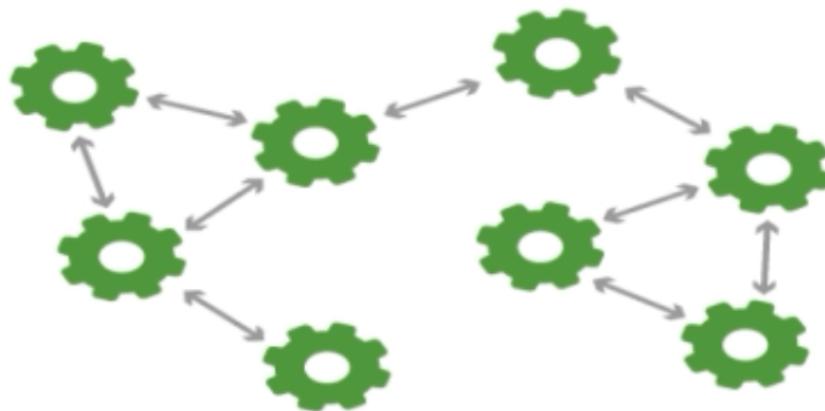
9.1. 概述

9.1.1. 为什么会出现这个技术？需要解决哪些问题？

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服务节点调用来协同产生最后的请求结果，**每一个前端请求都会形成一个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.

9.1.2. 是什么

Spring Cloud Sleuth 提供了一套完整的服务跟踪的解决方案

在分布式系统中提供追踪解决方案并且兼容支持了 zipkin(负责展现)

<https://docs.spring.io/spring-cloud-sleuth/docs/2.2.6.RELEASE/reference/html/>

<https://zipkin.io/>

Zipkin Find a trace Dependencies ENGLISH Search by trace ID DOWNLOAD

ROUTING: post /location/update/v4

Duration: 131.848ms Services: 10 Depth: 4 Total Spans: 13 Trace ID: a03ee8fff1dcdb9b9

ROUTING post /location/update/v4 [131.848ms]

P_MAIN/API_PRC post api proxy proxy [125ms]

- MEMCACHE get my_cache_name_v2 [993μs]
- YELP-MAIN txn: user_get_basic_and_scout_info [3.884ms]
- MYSQL begin [445μs]
- MEMCACHE get user_details_cache-20150901 [1.068ms]
- MEMCACHE get_multi my_cache_name_v1 [233μs]
- MYSQL commit [374μs]

MOBILE_API post /location/update/v4 [56ms]

- MEMCACHE get_multi mobile_api_nonce [1.066ms]
- MEMCACHE set mobile_api_nonce [1.026ms]
- get [3ms]
- post [14ms]

Annotations

Tags

- ecosystem prod
- habitat uswest1aprod
- http.uri.client /location/update/v4
- region uswest1-prod
- response_status_code 200

Select... ▾

account
alice
app_db
auth
auth-service
backend
blt
bookie



platformapi

Uses (traced requests)

Service Name	Call	Error
execution	14	0
bookie	17	0
gizmo	52	0
paperboy	1	0

Used (traced requests)

Service Name	Call	Error
strongman	30	0
stlogin	53	0

9.2. 搭建链路监控步骤

Spring Cloud Sleuth 和 OpenZipkin（也称为 Zipkin）集成。Zipkin 是一个分布式跟踪平台，可用于跟踪跨多个服务调用的事务。Zipkin 允许开发人员以图形方式查看事务占用的时间量，并分解在调用中涉及的每个微服务所用的时间。在微服务架构中，Zipkin 是识别性能问题的宝贵工具。

建立 Spring Cloud Sleuth 和 Zipkin 涉及 4 项操作：

- 将 Spring Cloud Sleuth 和 Zipkin JAR 文件添加到捕获跟踪数据的服务中；
- 在每个服务中配置 Spring 属性以指向收集跟踪数据的 Zipkin 服务器；
- 安装和配置 Zipkin 服务器以收集数据；
- 定义每个客户端所使用的采样策略，便于向 Zipkin 发送跟踪信息。

9.2.1. zipkin

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

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

 京东商城

 README.md

The quickest way to get started is to fetch the [latest released server](#) as a self-contained executable jar. Note that the Zipkin server requires minimum JRE 8. For example:

```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
java -jar zipkin.jar
```

You can also start Zipkin via Docker.

```
# Note: this is mirrored as ghcr.io/openzipkin/zipkin
docker run -d -p 9411:9411 openzipkin/zipkin
```

Once the server is running, you can view traces with the Zipkin UI at http://your_host:9411/zipkin/.

```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
java -jar zipkin.jar
```



C:\Windows\System32\cmd.exe - java -jar zipkin-server-2.23.4.jar

(c) Microsoft Corporation。保留所有权利。

```
D:\Server>java -jar zipkin-server-2.23.4.jar
```

00
0000
000000
00000000
000000000000
00000000000000
00000000 0000000
000000 0000000
000000 0000000
000000 0 0 000000
000000 00 00 000000
0000000 0000 0000 00000000
000000 00000 00000 00000000
000000 000000 000000 00000000
00000000 00 00 00000000
00000000000000 00 00 00000000000000
00000000000000 00000000000000
00000000000000 000000000000

:: version 2.23.4 :: commit 3827477 ::

```
2021-09-29 11:51:42.503 INFO [/] 32968 --- [oss-http-*:9411] c.l.a.s.Server : Serving HTTP at /0:0:0:0:0:0:0:9411 - http://127.0.0.1:9411/
```

运行控制台

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



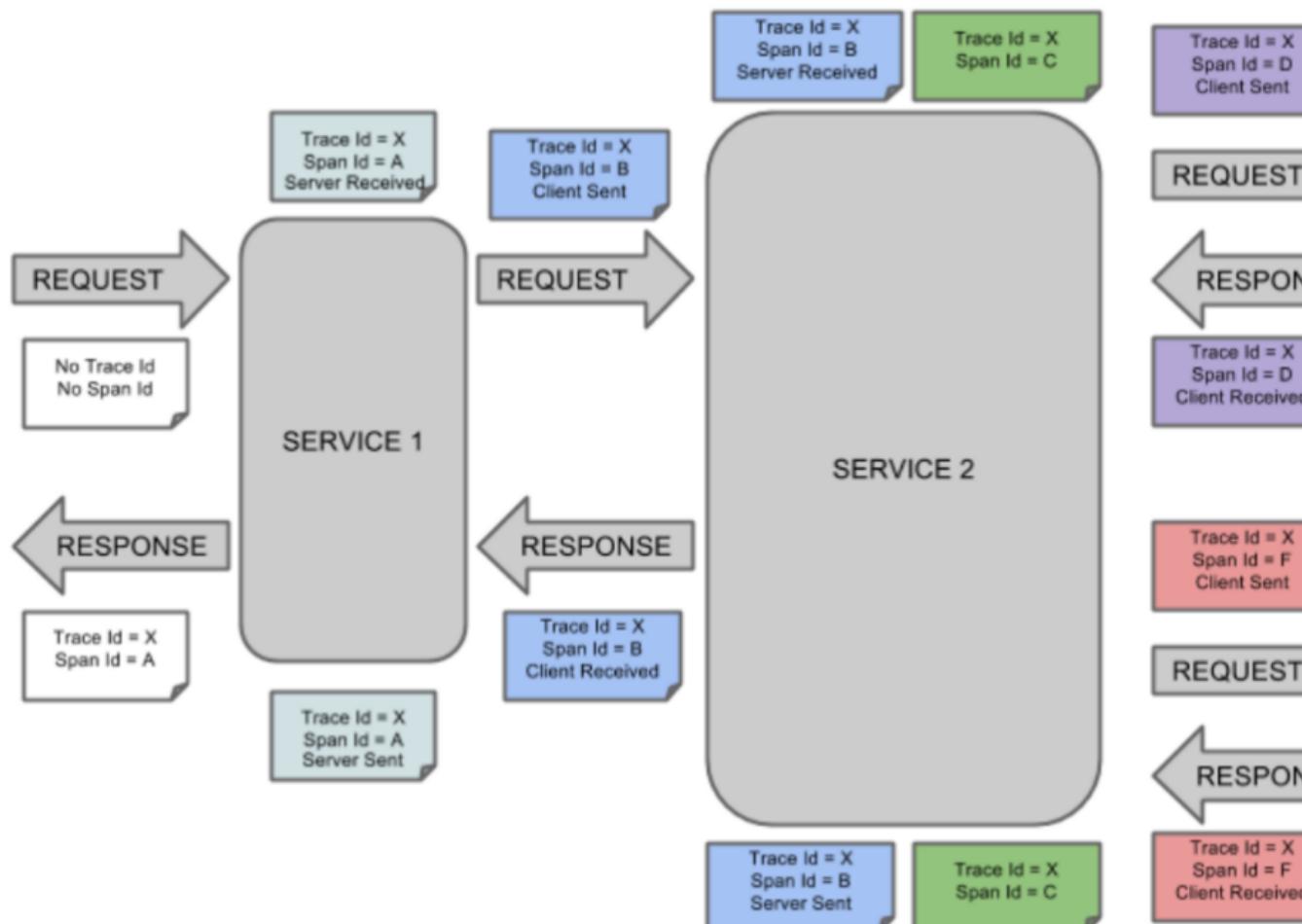
让天下没有难学的技术

The screenshot shows the Zipkin web interface at localhost:9411/zipkin/. The top navigation bar includes links for '找到一个痕迹' (Find a Trace), '依赖' (Dependencies), and language selection '中文 (简体)'. A search bar at the top right is labeled 'Search by trace ID'. Below the search bar are buttons for '+', 'RUN QUERY', and settings. The main area features a large magnifying glass icon and the text '查询trace链路' (Query trace path). A placeholder message says '请在搜索栏中选择条件，然后点击按钮进行查找' (Please select conditions in the search bar and click the button to search).

术语

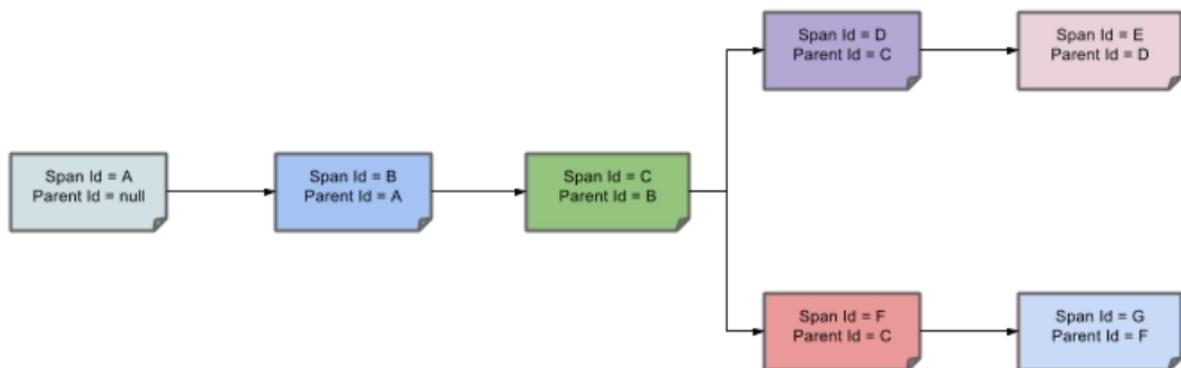
完整的调用链路

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

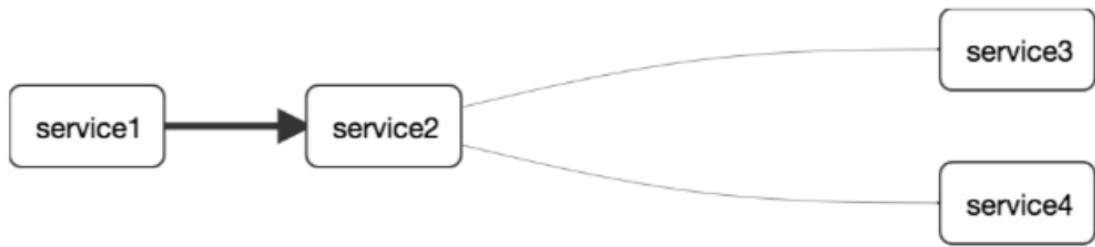


上图 what

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



整个链路的依赖关系如下：



名词解释

Trace:类似于树结构的 Span 集合，表示一条调用链路，存在唯一标识

span:表示调用链路来源，通俗的理解 span 就是一次请求信息

9.2.2. 服务提供者

1. 修改：cloud-provider-payment8001

2. POM

```
<!--包含了 sleuth+zipkin-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

3. YML

```
server:
  port: 8001

spring:
  application:
    name: cloud-payment-service
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      #采样率值介于 0~1 之间，1 表示全部采样
      probability: 1
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/cloud2021?useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: root

  mybatis:
    mapperLocations: classpath:/mapper/*.xml
    type-aliases-package: com.atguigu.springcloud.entities

eureka:
  client:
    register-with-eureka: true
    fetchRegistry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

4. 业务类 PaymentController

```
@GetMapping("/payment/zipkin")
public String paymentZipkin(){
```

```
        return "hi ,i'am paymentzipkin server, welcome to atguigu, O(n_n)O 哈哈~";  
    }
```

9.2.3. 服务消费者（调用方）

1. 修改：cloud-consumer-order80

2. POM

```
<!--包含了 sleuth+zipkin-->  
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-zipkin</artifactId>  
</dependency>
```

3. YML

```
server:  
  port: 80  
  
spring:  
  application:  
    name: cloud-order-service  
  zipkin:  
    base-url: http://localhost:9411  
  sleuth:  
    sampler:  
      probability: 1  
  
eureka:  
  client:  
    #表示是否将自己注册进 EurekaServer 默认为 true。  
    register-with-eureka: false  
    #是否从 EurekaServer 抓取已有的注册信息，默认为 true。单节点无所谓，集群必须设置  
    为 true 才能配合 ribbon 使用负载均衡  
    fetchRegistry: true
```

```
service-url:  
    #单机  
    defaultZone: http://localhost:7001/eureka  
    #集群  
    #defaultZone:  
http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka # 集群版
```

4. 业务类 OrderController

```
//==> zipkin+sleuth  
 @GetMapping("/consumer/payment/zipkin")  
 public String paymentZipkin(){  
     String result =  
 restTemplate.getForObject("http://localhost:8001" + "/payment/zipkin/", String.class);  
     return result;  
 }
```

9.2.4. 依次启动 eureka7001/8001/80

80 调用 8001 几次测试下

9.2.5. 打开浏览器访问: <http://localhost:9411>

会出现以下界面

查看



让天下没有难学的技术

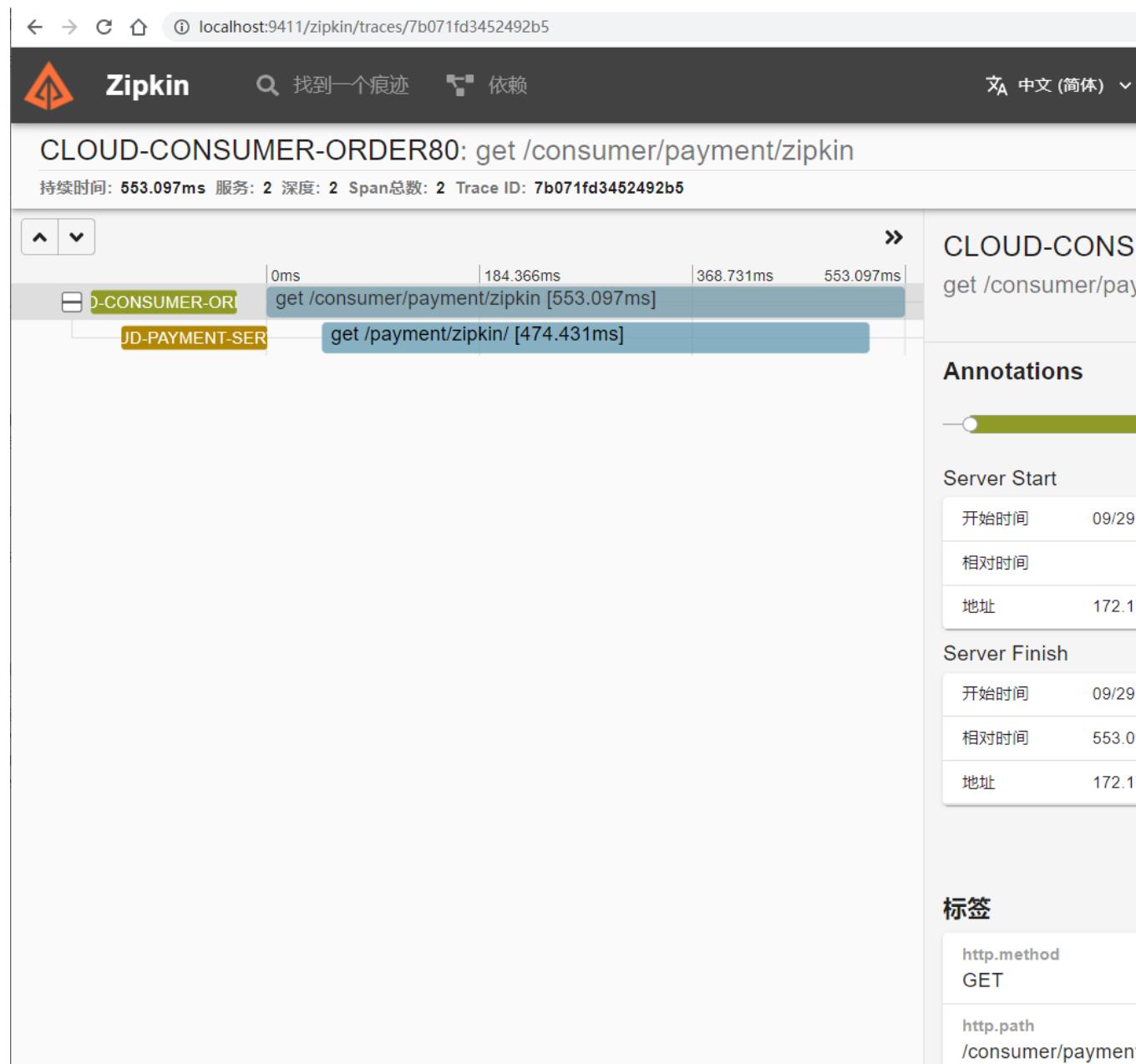
← → ⌂ ⌂ ⓘ localhost:9411/zipkin/?lookback=15m&endTs=1632891662289&limit=10

Zipkin 找到一个痕迹 依赖 中文 (简体) ▾

+ 10 条结果 EXPAND ALL COLLAPSE ALL

Root	开始时间	Sp
cloud-consumer-order80: get /consumer/payment/zipkin	a few seconds ago (09/29 13:00:48:213)	
Trace ID: 7b071fd3452492b5		
cloud-consumer-order80 (2) cloud-payment-service (1)		

点击【SHOW】按钮，查看依赖关系。



10. SpringCloud Alibaba 入门简介

10.1. why 会出现 SpringCloud alibaba

Spring Cloud Netflix 项目进入维护模式

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

 spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now

Spring Cloud Netflix Projects Entering Maintenance Mode

Recently, Netflix [announced](#) that Hystrix is entering maintenance mode. Ribbon has been in a [similar state](#) since 2016. Although Hystrix and Ribbon are now in maintenance mode, they are still deployed at scale at Netflix.

The Hystrix Dashboard and Turbine have been superseded by Atlas. The last commits to these project are 2 years and 4 years ago respectively. Zuul 1 and Archaius 1 have both been superseded by later versions that are not backward compatible.

The following Spring Cloud Netflix modules and corresponding starters will be placed into maintenance mode:

1. spring-cloud-netflix-archaius
2. spring-cloud-netflix-hystrix-contract
3. spring-cloud-netflix-hystrix-dashboard
4. spring-cloud-netflix-hystrix-stream
5. spring-cloud-netflix-hystrix
6. spring-cloud-netflix-ribbon
7. spring-cloud-netflix-turbine-stream
8. spring-cloud-netflix-turbine
9. spring-cloud-netflix-zuul

This does not include the Eureka or concurrency-limits modules.

What Is Maintenance Mode?

Placing a module in maintenance mode means that the Spring Cloud team will no longer be adding new features to the module. We will fix blocker bugs and security issues, and we will also consider and review small pull requests from the community.

We intend to continue to support these modules for a period of **at least** a year from the general availability of the [Greenwich release train](#).

说明：



Spring Cloud Netflix 项目进入维护模式

最近，Netflix 宣布 Hystrix 正在进入维护模式。自 2016 年以来，Ribbon 已处于类似状态。

虽然 Hystrix 和 Ribbon 现已处于维护模式，但它们仍然在 Netflix 大规模部署。

Hystrix Dashboard 和 Turbine 已被 Atlas 取代。这些项目的最后一次提交是 2 年和 4 年前。Zuul1 和 Archaius1 都被后来不兼容的版本所取代。

这不包括 Eureka 或并发限制模块。

什么是维护模式？

将模块置于维护模式，意味着 SpringCloud 团队将不会再向模块添加新功能。我们将修复 block 级别的 bug 以及安全问题，我们也会考虑并审查社区的小型 pull request。

我们打算继续支持这些模块，直到 Greenwich 版本被普遍采用至少一年。

进入维护模式意味着什么？

Spring Cloud Netflix 将不再开放新的组件

我们都应该知道 SpringCloud 版本迭代算是比较快的，因而出现了很多重大 ISSU 都还来不及 Fix 就又推出另一个 Release 了。

进入维护模式意思就是目前以致以后一段时间 SpringCloud netflix 提供的服务和功能就这么多了，不再开发新的组件和功能了。以后将以维护和 Merge 分支 Pull Request 为主。

新组件功能将以其他替代的方式实现



Replacements

We recommend the following as replacements for the functionality provided by these modules.

Current	Replacement
Hystrix	Resilience4j
Hystrix Dashboard / Turbine	Micrometer + Monitoring System
Ribbon	Spring Cloud Loadbalancer
Zuul 1	Spring Cloud Gateway
Archaius 1	Spring Boot external config + Spring Cloud Config

Look for a future blog post on Spring Cloud Loadbalancer and integration with a new Netflix project Concurrency Limits.

10.2. SpringCloud alibaba 带来了什么？

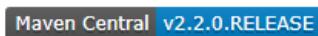
10.2.1. 是什么

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

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

github.com/alibaba/spring-cloud-alibaba/blob/master/README-zh.md

Spring Cloud Alibaba

 PASSED  Maven Central v2.2.0.RELEASE  codecov 31%  license Apache 2

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 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中间件来迅速搭建分布式应用系统。

10.2.2. 能干嘛

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

10.2.3. 去哪下

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2.2.6.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

10.2.4. 怎么玩

一整套解决方案，简单理解就是替换 Netflix 那一套

Sentinel: 把流量作为切入点，从**流量控制、熔断降级、系统负载保护**等多个维度保护服务的稳定性。

Nacos: 一个更易于构建云原生应用的动态**服务发现、配置管理**和服务管理平台。

RocketMQ: 一款开源的**分布式消息系统**，基于高可用分布式集群技术，提供低延时的、高可靠的**消息发布与订阅**服务。

Dubbo: Apache Dubbo™ 是一款高性能 Java RPC 框架。

Seata: 阿里巴巴开源产品，一个易于使用的高性能**分布式事务**解决方案。

Alibaba Cloud ACM: 一款在分布式架构环境中对应用配置进行集中管理和推送的应用配置中心产品。

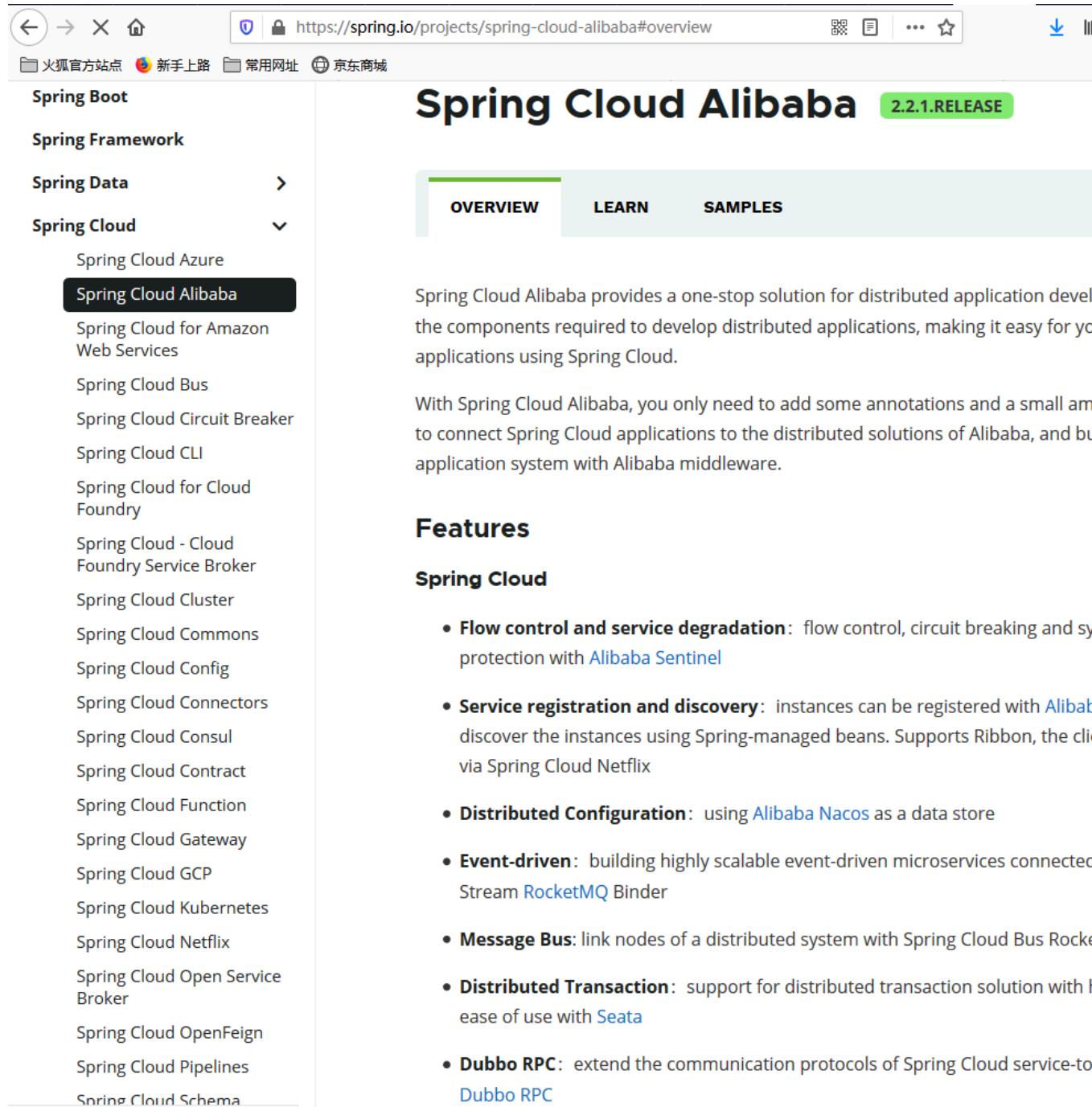
Alibaba Cloud OSS: 阿里云**对象存储服务（Object Storage Service，简称 OSS）**，是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。

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。

10.3.SpringCloud alibaba 学习资料获取

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



The screenshot shows the official website for Spring Cloud Alibaba at <https://spring.io/projects/spring-cloud-alibaba#overview>. The page is titled "Spring Cloud Alibaba 2.2.1.RELEASE". It features a navigation bar with tabs for "OVERVIEW" (which is active), "LEARN", and "SAMPLES". The main content area starts with a brief introduction: "Spring Cloud Alibaba provides a one-stop solution for distributed application development, the components required to develop distributed applications, making it easy for you to build distributed applications using Spring Cloud." Below this, there's a section titled "Features" with a list of ten bullet points detailing various features like flow control, service registration, and distributed transactions.

Spring Cloud Alibaba provides a one-stop solution for distributed application development, the components required to develop distributed applications, making it easy for you to build distributed applications using Spring Cloud.

With Spring Cloud Alibaba, you only need to add some annotations and a small amount of configuration to connect Spring Cloud applications to the distributed solutions of Alibaba, and build distributed applications with Alibaba middleware.

Features

Spring Cloud

- **Flow control and service degradation:** flow control, circuit breaking and system protection with [Alibaba Sentinel](#)
- **Service registration and discovery:** instances can be registered with [Alibaba Registry](#), discover the instances using Spring-managed beans. Supports Ribbon, the client-side load balancer, via Spring Cloud Netflix
- **Distributed Configuration:** using [Alibaba Nacos](#) as a data store
- **Event-driven:** building highly scalable event-driven microservices connected via [Apache RocketMQ](#) Binder
- **Message Bus:** link nodes of a distributed system with Spring Cloud Bus RocketMQ
- **Distributed Transaction:** support for distributed transaction solution with [Seata](#) for ease of use with [Spring Cloud Transactional Consistency](#)
- **Dubbo RPC:** extend the communication protocols of Spring Cloud service-to-service communication with [Dubbo RPC](#)

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。

SpringCloud Alibaba进入了SpringCloud官方孵化器，而且毕业了

英文

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

中文

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

11. SpringCloud Alibaba Nacos 服务注册和

配置中心

11.1.Nacos 简介

11.1.1. 为什么叫 Nacos

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

11.1.2. 是什么

一个更易于构建云原生应用的动态服务发现，配置管理和服务管理中心

Nacos: Dynamic Naming and Configuration Service

Nacos 就是注册中心+配置中心的组合

等价于：Nacos = Eureka+Config+Bus

11.1.3. 能干嘛

替代 Eureka 做服务注册中心

替代 Config 做服务配置中心

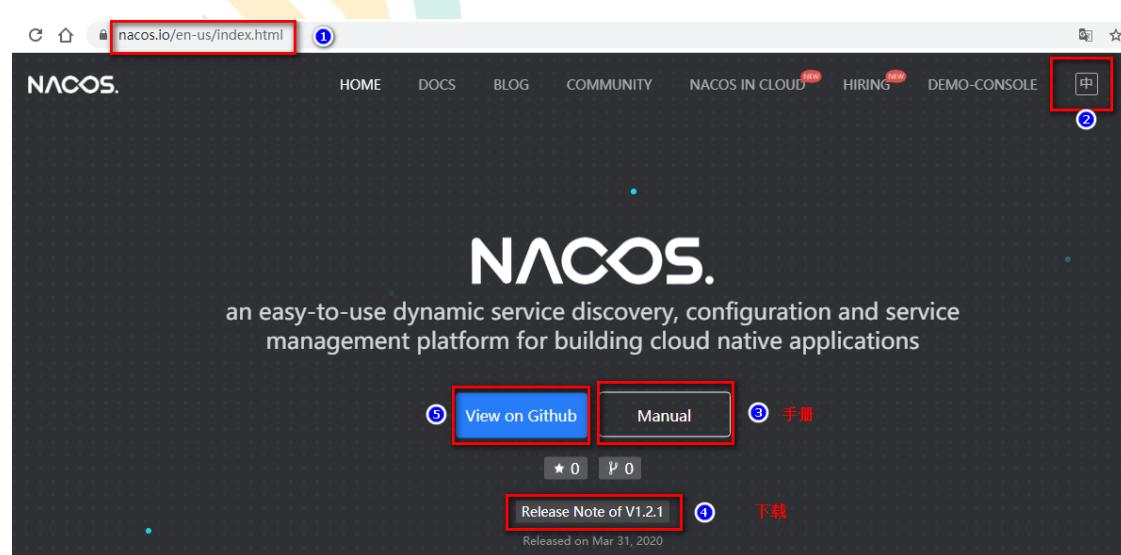
11.1.4. 去哪下

<https://github.com/alibaba/Nacos>



官网文档

<https://nacos.io/zh-cn/index.html>





让天下没有难学的技术

该截图展示了 Nacos 官方网站的首页。顶部有网站地址 nacos.io/zh-cn/docs/what-is-nacos.html。上方导航栏包括：首页、文档、博客、社区、企业版Nacos、招贤纳士、控制台样例和一个带“En”字样的按钮。右侧有一个带有铅笔图标和“Nacos 文档”的蓝色横幅。

该截图展示了 GitHub 上 alibaba/nacos 仓库的 Tags 页面。仓库名称“alibaba / nacos”被红色框选。下方是主要功能菜单：Code、Issues (307)、Pull requests (9)、Actions、Projects (1)、Wiki、Security 和 Insights。在“Tags”标签页下，列出了以下版本：

- 2.0.3 (2023-07-28) - 5a4d433: zip, tar.gz, Notes, Downloads
- 2.0.2 (2023-06-11) - 1fac5c8: zip, tar.gz, Notes, Downloads
- 2.0.1 (2023-04-29) - d1a8180: zip, tar.gz, Notes, Downloads
- 1.4.2 (2023-04-29) - 50c7318: zip, tar.gz, Notes, Downloads (此条目被红色框选)

https://spring-cloud-alibaba-group.github.io/github-pages/hoxton/en-us/index.html#_spring_cloud_alibaba_nacos_discovery

The screenshot shows a web browser displaying the official Spring Cloud Alibaba documentation. The URL in the address bar is https://spring-cloud-alibaba-group.github.io/github-pages/hoxton/en-us/index.html#_spring_cloud_alibaba_nacos_discovery. The page content includes a sidebar with navigation links for introduction, dependency management, and various discovery mechanisms like Eureka, Zookeeper, Consul, and Nacos. The main content area discusses "How to Introduce Nacos Discovery for service registration/discovery". It provides instructions to use the starter with group ID `com.alibaba.cloud` and artifact ID `spring-cloud-starter-alibaba-nacos-discovery`. A code snippet shows the dependency configuration:

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

11.1.5. 各种注册中心比较

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

据说 nacos 在阿里巴巴内部有超过 10 万的实例运行，已经过了类似双十一等各种大型流量的考验

CAP 原则又称 CAP 定理，指的是在一个分布式系统中，一致性 (Consistency) 、可用性 (Availability) 、分区容错性 (Partition tolerance) 。CAP 原则指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。

11.2. 安装并运行 Nacos

11.2.1. 本地 Java8+Maven 环境已经 OK

11.2.2. 先从官网下载 Nacos

<https://github.com/alibaba/nacos/releases/tag/1.4.2>

11.2.3. 解压安装包，直接运行 bin/startup.cmd -m

standalone

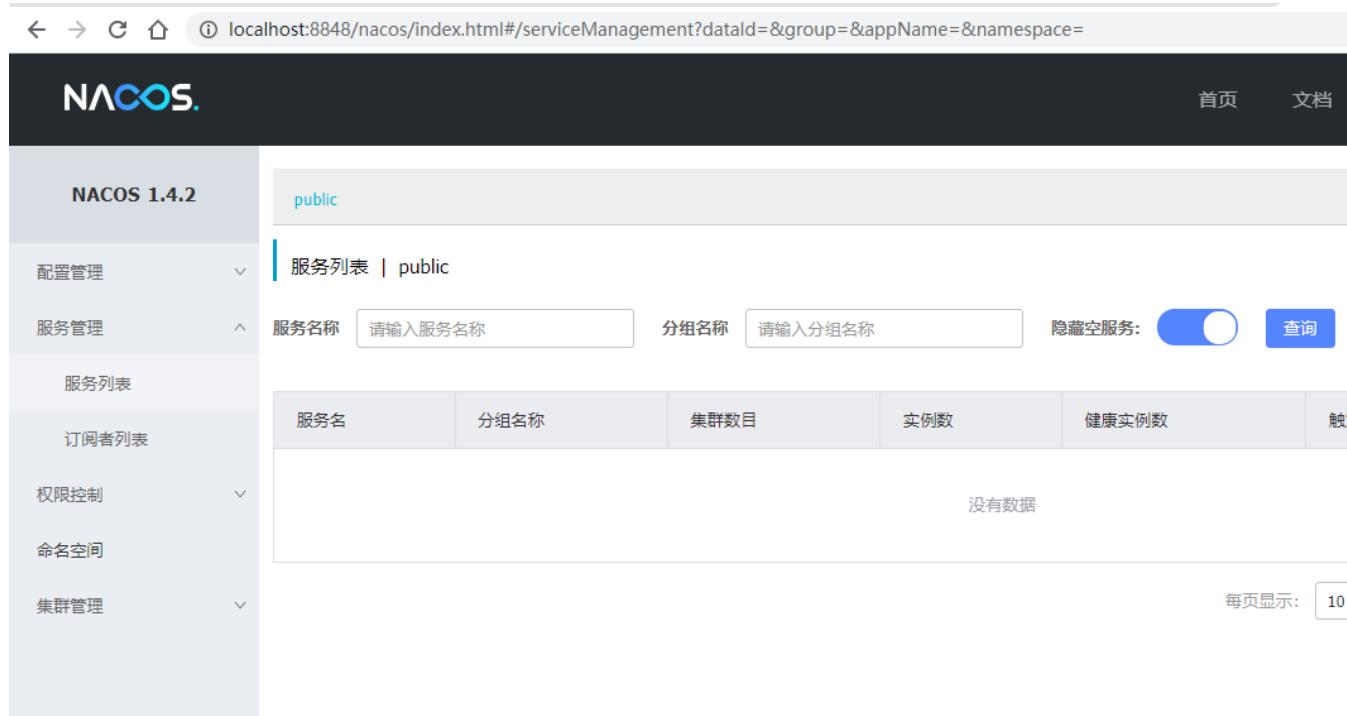
默认：MODE="cluster"集群方式启动，如果单机启动需要设置-m standalone 参数，否则，启动失败。

11.2.4. 命令运行成功后直接访问

http://localhost:8848/nacos

默认账号密码都是 nacos

11.2.5. 登录结果页面



The screenshot shows the Nacos 1.4.2 service management interface. The left sidebar has a 'NACOS 1.4.2' header and categories: 配置管理, 服务管理 (selected), 服务列表, 订阅者列表, 权限控制, 命名空间, and 集群管理. The main area shows a 'public' service group with a '服务列表 | public' heading. It includes search fields for '服务名称' and '分组名称', a toggle for '隐藏空服务', and a '查询' button. A table with columns '服务名', '分组名称', '集群数目', '实例数', and '健康实例数' is present but empty. A message '没有数据' is displayed. At the bottom right, there are buttons for '每页显示: 10'.

11.3.Nacos 作为服务注册中心演示

11.3.1. 官网文档

<https://spring.io/projects/spring-cloud-alibaba#learn>

Spring Boot

Spring Framework

Spring Data >

Spring Cloud > **Spring Cloud Alibaba**

Spring Cloud Azure

Spring Cloud for Amazon Web Services

Spring Cloud Bus

Spring Cloud Circuit Breaker

Spring Cloud CLI

Spring Cloud for Cloud Foundry

Spring Cloud - Cloud Foundry Service Broker

Spring Cloud Cluster

Spring Cloud Alibaba 2.2.1.RELEASE

OVERVIEW LEARN SAMPLES

Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** you can achieve with them.

Version	Status	Link
2.2.1.RELEASE	CURRENT GA	Reference Doc.
2.1.2.RELEASE	GA	Reference Doc.
2.0.2.RELEASE	GA	Reference Doc.
1.5.0.RELEASE	GA	Reference Doc.

11.3.2. 基于 Nacos 的服务提供者

1. 新建 Module: cloudalibaba-provider-payment9001

2. POM

父 POM

```
<!--spring cloud alibaba 2.2.6.RELEASE-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>
    <version>2.2.6.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

▪ 本模块 POM

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

```
<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.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<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>
</dependency>
</dependencies>
```

3. YML

```
server:
port: 9001

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

```
management:  
endpoints:  
web:  
exposure:  
include: '*' #默认只公开了/health 和/info 端点，要想暴露所有端点只需设置成星号
```

4. 主启动

```
package com.atguigu.springcloud.alibaba;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
  
@EnableDiscoveryClient  
@SpringBootApplication  
public class PaymentMain9001 {  
    public static void main(String[] args) {  
        SpringApplication.run(PaymentMain9001.class,args);  
    }  
}
```

5. 业务类

```
package com.atguigu.springcloud.alibaba.controller;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class PaymentController{  
    @Value("${server.port}")  
    private String serverPort;  
  
    @GetMapping(value = "/payment/nacos/{id}")  
    public String getPayment(@PathVariable("id") Long id) {
```

```
        return "nacos registry, serverPort: "+ serverPort+"\t id"+id;
    }
}
```

6. 测试

<http://localhost:9001/payment/nacos/1>

nacos 控制台



The screenshot shows the Nacos 1.4.2 control panel. The left sidebar has tabs for Configuration Management, Service Management, Subscribers List, Permission Control, Namespace, and Cluster Management. The main area is titled 'public' and shows a table for service lists. The table has columns: Service Name, Group Name, Cluster Count, and Instance Count. One entry is visible: nacos-payment-provider under DEFAULT_GROUP with 1 instance.

服务名	分组名称	集群数目	实例数
nacos-payment-provider	DEFAULT_GROUP	1	1

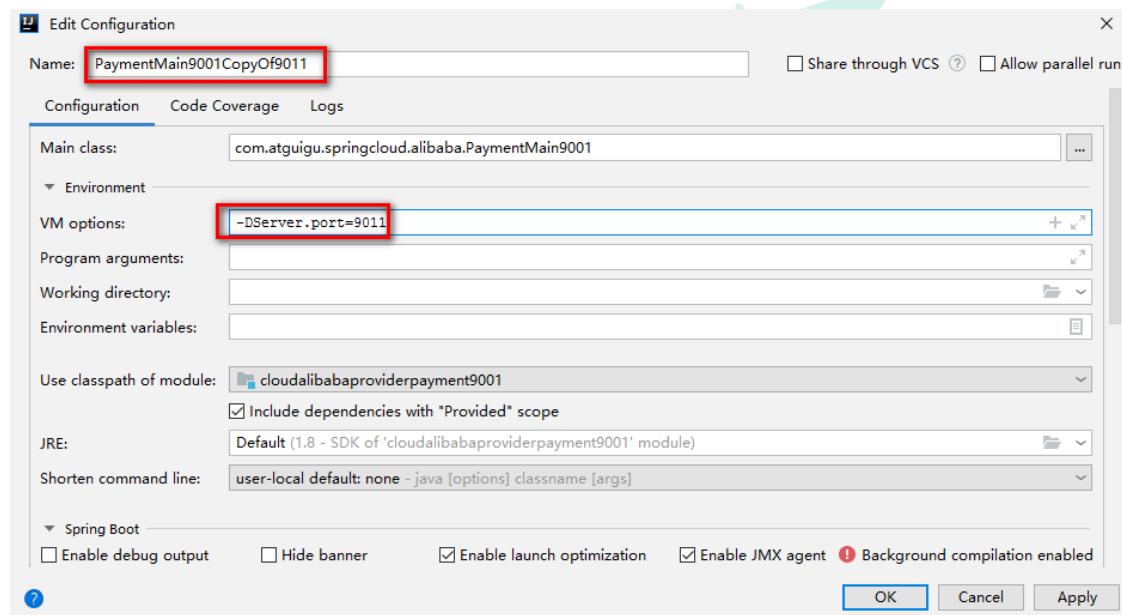
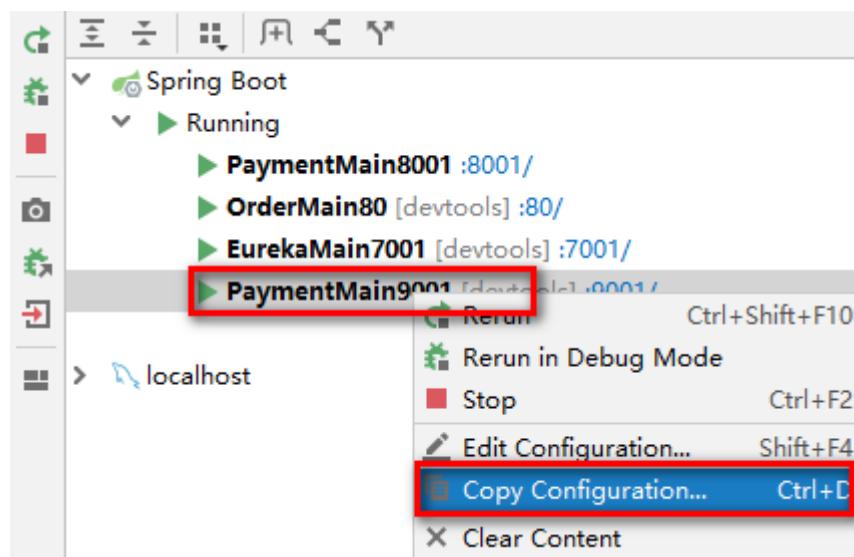
nacos 服务注册中心+服务提供者 9001 都 ok 了

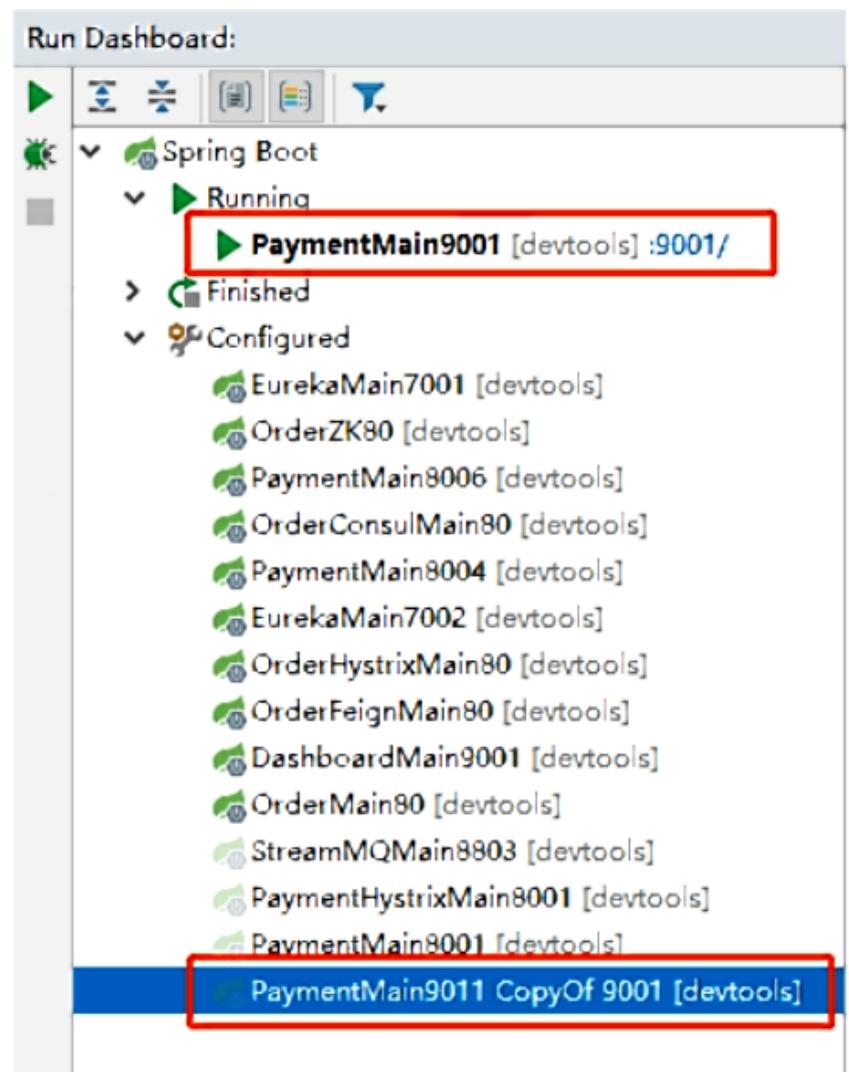
7. 为了下一章节演示 nacos 的负载均衡，参照 9001 新建 9002

新建 cloudalibaba-provider-payment9002

9002 其他步骤你懂的

或者取巧不想新建重复体力劳动，直接拷贝虚拟端口映射



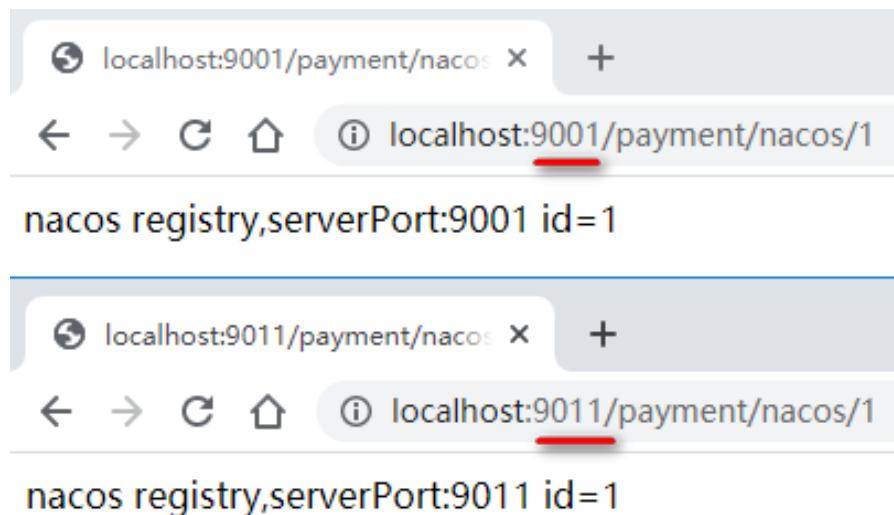


The screenshot shows the Nacos service management interface at localhost:8848/nacos/#/serviceManagement?dataId=&group=&appName=&namespace=public.

The left sidebar shows navigation options: 配置管理 (Configuration Management), 监听查询 (Monitoring & Query), 服务管理 (Service Management), 订阅者列表 (Subscriber List), and 命名空间 (Namespace). The '服务管理' section is expanded, and the '服务列表' item is highlighted with a red box.

The main content area displays a table for the 'public' service group. The table has columns: 服务名 (Service Name), 分组名称 (Group Name), 集群数目 (Cluster Count), 实例数 (Instance Count), 健康实例数 (Healthy Instance Count), 触发保护阈值 (Trigger Protection Threshold), and 操作 (Operations).

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



The image shows two browser windows side-by-side. Both windows have the address bar set to `localhost:9001/payment/nacos/1`. The first window's address bar has a red underline under the URL. The second window's address bar also has a red underline under the URL. Both windows display the text "nacos registry,serverPort:9001 id=1".

11.3.3. 基于 Nacos 的服务消费者

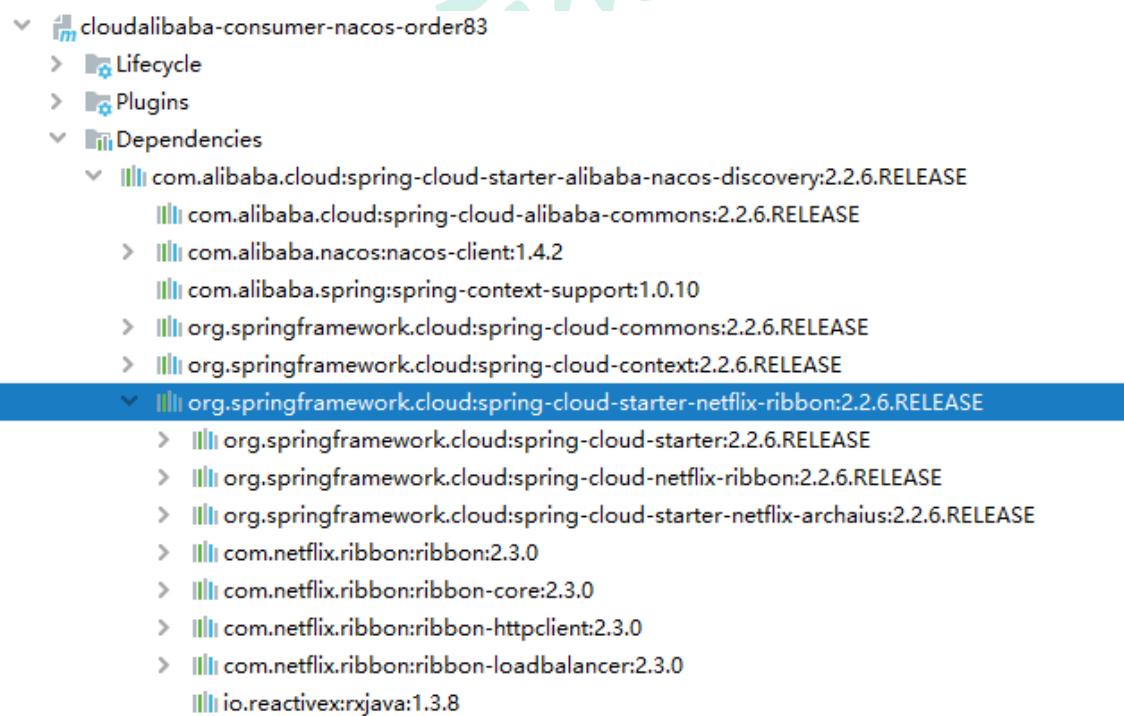
1. 新建 Module: `cloudalibaba-consumer-nacos-order83`

2. POM

```
<dependencies>
    <!--SpringCloud alibaba nacos -->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>com.atguigu.springcloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>${project.version}</version>
    </dependency>
    <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>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<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>
</dependency>
</dependencies>
```

为什么 nacos 支持负载均衡



3. YML

```
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(注册成功进 nacos 的微服务提供者【可选】，注意：  
      nacos-payment-provider 含有 IP 和端口)  
      service-url:  
        nacos-user-service: http://nacos-payment-provider
```

4. 主启动



```
package com.atguigu.springcloud.alibaba;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
  
@EnableDiscoveryClient  
@SpringBootApplication  
public class OrderNacosMain83{  
    public static void main(String[] args){  
        SpringApplication.run(OrderNacosMain83.class,args);  
    }  
}
```

5. 业务类

ApplicationContextBean

```
package com.atguigu.springcloud.alibaba.config;
```

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class ApplicationContextConfig{
    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
```

6. OrderNacosController

```
package com.atguigu.springcloud.alibaba.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
import javax.annotation.Resource;

@RestController
@Slf4j
public class OrderNacosController{
    @Resource
    private RestTemplate restTemplate;

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

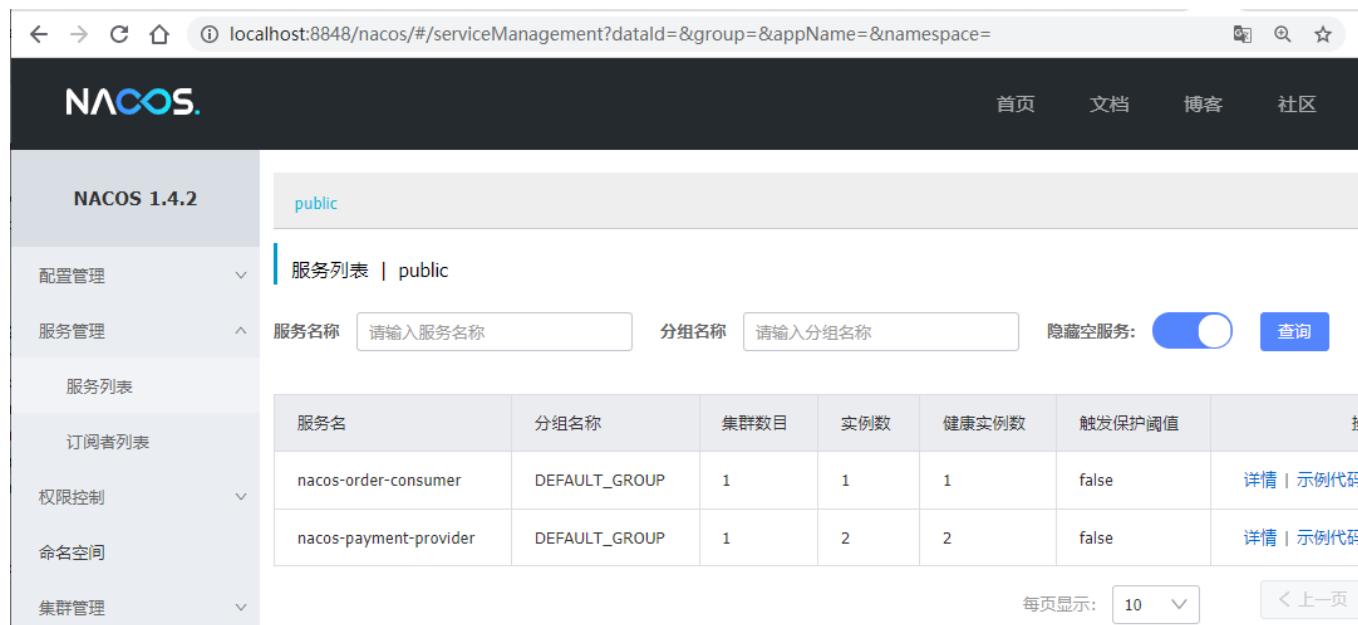
    @GetMapping(value = "/consumer/payment/nacos/{id}")
    public String paymentInfo(@PathVariable("id") Long id){
        return
    }
}
```

```
restTemplate.getForObject(serverURL+ "/payment/nacos/" +id, String.class);
}

}
```

7. 测试

nacos 控制台



The screenshot shows the Nacos 1.4.2 service management interface. The left sidebar has a tree structure with nodes like '配置管理', '服务管理' (selected), '服务列表', '订阅者列表', '权限控制', '命名空间', and '集群管理'. The main content area is titled '服务列表 | public'. It includes search fields for '服务名称' and '分组名称', a toggle for '隐藏空服务', and a '查询' button. Below these are two service entries in a table:

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

At the bottom right, there are buttons for '每页显示: 10' and '< 上一页'.

<http://localhost:83/consumer/payment/nacos/1>

83 访问 9001/9002, 轮询负载 OK

11.3.4. 服务注册中心对比

1. Nacos 和 CAP

CAP 原则又称 CAP 定理，指的是在一个分布式系统中， Consistency (一致性) 、 Availability (可用性) 、 Partition tolerance (分区容错性) ，三者不可得兼。

一致性 (C) : 在分布式系统中的所有数据备份，在同一时刻是否同样的值。 (等同于所有节点访问同一份最新的数据副本)

可用性 (A) : 在集群中一部分节点故障后, 集群整体是否还能响应客户端的读写请求。
(对数据更新具备高可用性)

分区容忍性 (P) : 以实际效果而言, 分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性, 就意味着发生了分区的情况, 必须就当前操作在 C 和 A 之间做出选择。

CAP 原则的精髓就是要么 AP, 要么 CP, 要么 AC, 但是不存在 CAP。

如果在某个分布式系统中数据无副本, 那么系统必然满足强一致性条件, 因为只有唯一数据, 不会出现数据不一致的情况, 此时 C 和 P 两要素具备, 但是如果系统发生了网络分区状况或者宕机, 必然导致某些数据不可以访问, 此时可用性条件就不能被满足, 即在此情况下获得了 CP 系统, 但是 CAP 不可同时满足。

因此在进行分布式架构设计时, 必须做出取舍。当前一般是通过[分布式缓存](#)中各节点的[最终一致性](#)来提高系统的性能, 通过使用多节点之间的[数据异步复制技术](#)来实现集群化的数据一致性。

Nacos 与其他注册中心特性对比

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

2. Nacos 支持 AP 和 CP 模式的切换

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

11.4.Nacos 作为服务配置中心演示

11.4.1. Nacos 作为配置中心-基础配置

1. 创建 Module: cloudalibaba-config-nacos-client3377

2. POM

```
<dependencies>
    <!--nacos-config-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
    </dependency>
    <!--nacos-discovery-->
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <!--web + actuator-->
    <dependency>
        <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>
    <!--一般基础配置-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <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>
</dependency>
</dependencies>
```

3. YML

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

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

bootstrap.yml

```
server:
  port: 3377

spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #服务注册中心地址
      config:
        server-addr: localhost:8848 #配置中心地址
        file-extension: yaml #指定 yaml 格式的配置 (yml 和 yaml 都可以)

      #${spring.application.name}-#${spring.profile.active}.#${spring.cloud.nacos.config.file-extension}
      #nacos-config-client-dev.yaml (一定要与 file-extension 值保持一致)
```

application.yml

```
spring:
```

```
profiles:  
active: dev #表示开发环境
```

4. 主启动

```
package com.atguigu.springcloud.alibaba;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
  
@EnableDiscoveryClient  
@SpringBootApplication  
public class NacosConfigClientMain3377{  
    public static void main(String[] args) {  
        SpringApplication.run(NacosConfigClientMain3377.class, args);  
    }  
}
```

5. 业务类:ConfigClientController

```
package com.atguigu.springcloud.alibaba.controller;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.cloud.context.config.annotation.RefreshScope;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
//@RefreshScope //通过 SpringCloud 原生注解@RefreshScope 实现配置自动更新  
public class ConfigClientController{  
    @Value("${config.info}") //对应 nacos 配置:nacos-config-client-dev.yaml  
    private String configInfo;  
  
    @GetMapping("/config/info")  
    public String getConfigInfo() {  
        return configInfo;  
    }  
}
```

{}

6. 在 Nacos 中添加配置信息

Nacos 中的匹配规则

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

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

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` 字分。

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

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

- `prefix` 默认为 `spring.application.name` 的值，也可以通过配置项 `spring.cloud.nacos.config.prefix` 来配置。
- `spring.profile.active` 即为当前环境对应的 profile，详情可以参考 [Spring Boot 文档](#)。注 `spring.profile.active` 为空时，对应的连接符 `-` 也将不存在，`dataId` 的拼接格式变成 `${prefix}.${file-extension}`
- `file-extension` 为配置内容的数据格式，可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。目前只支持 `properties` 和 `yaml` 类型。

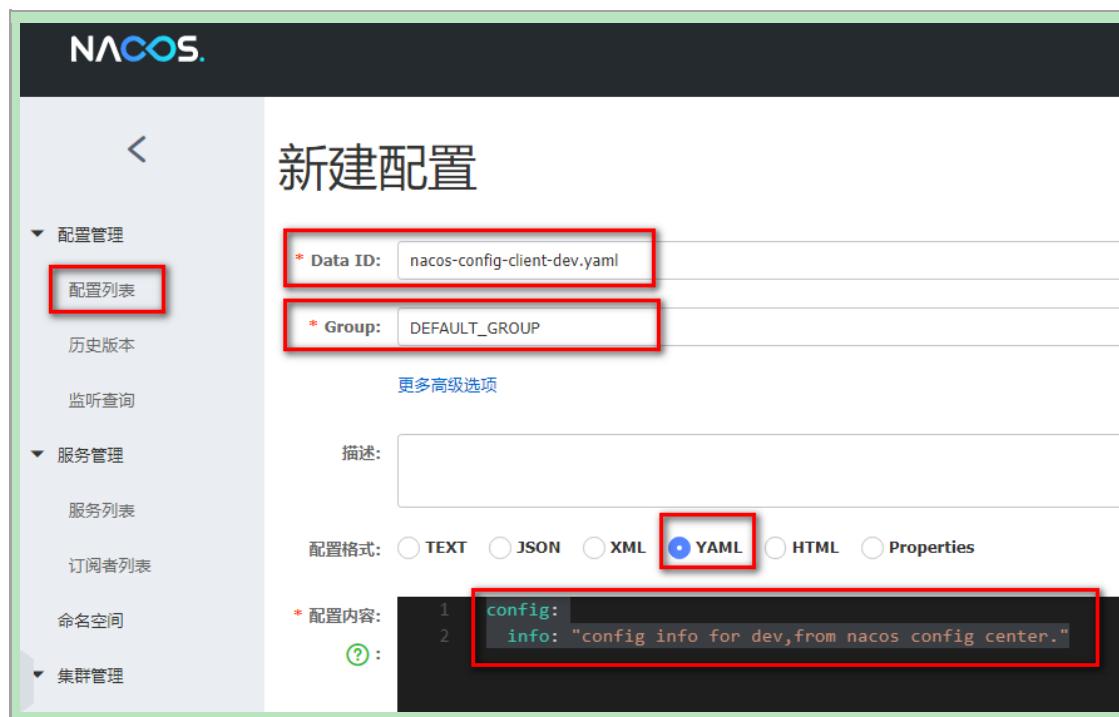
最后公式：

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

`nacos-config-client-dev.yaml`

Nacos 界面配置对应

```
config:  
  info: "config info for dev,from nacos config center."
```



设置 DataId

公式: \${spring.application.name}-
\${spring.profile.active}.\${spring.cloud.nacos.config.file-extension}

小总结说明



历史配置：Nacos 会记录配置文件的历史版本默认保留 30 天

7. 测试

启动前需要在 nacos 客户端-配置管理-配置管理栏目下有没有对应的 yaml 配置文件

运行 cloud-config-nacos-client3377 的主启动类

调用接口查看配置信息: <http://localhost:3377/config/info>

8. 自带动态刷新

修改 Nacos 中的 **yaml** 配置文件，查看配置已经刷新

11.4.2. Nacos 作为配置中心-分类配置

1. 问题

多环境多项目管理

问题 1

- 实际开发中，通常一个系统会准备
- dev 开发环境
- test 测试环境
- prod 生产环境
- 如何保证指定环境启动时服务能正确读取到 Nacos 上相应环境的配置文件呢？

问题 2

- 一个大型分布式微服务系统会有很多微服务子项目
- 每一个微服务项目又会相应的开发环境、测试环境、预发环境、正式环境....
- 那怎么对这些微服务配置进行管理呢？

2. Nacos 的图形化管理界面

- 配置管理

localhost:8848/nacos/#/configurationManagement?dataId=&group=&appName=&name... 🔍 ⭐ 🌐 🎯 📁

NACOS.

NACOS 1.4.2 public

配置管理 | public 查询结果: 共查询到 1 条满足要求的配置。

Data ID: 添加通配符**进行模糊查询 Group: 添加通配符**进行模糊查询 **查询** 高级查询

	Data Id ↓↑	Group ↓↑	归属应用: ↓↑	操作
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP		详情 示例代码 编辑 删除

配置列表 历史版本 监听查询 服务管理 权限控制 命名空间 集群管理

每页显示: 10 < 上一页 1

localhost:8848/nacos/#/namespace?dataId=&group=&appName=&namespace= 🔍 ⭐ 🌐 🎯 📁

NACOS.

NACOS 1.4.2 命名空间

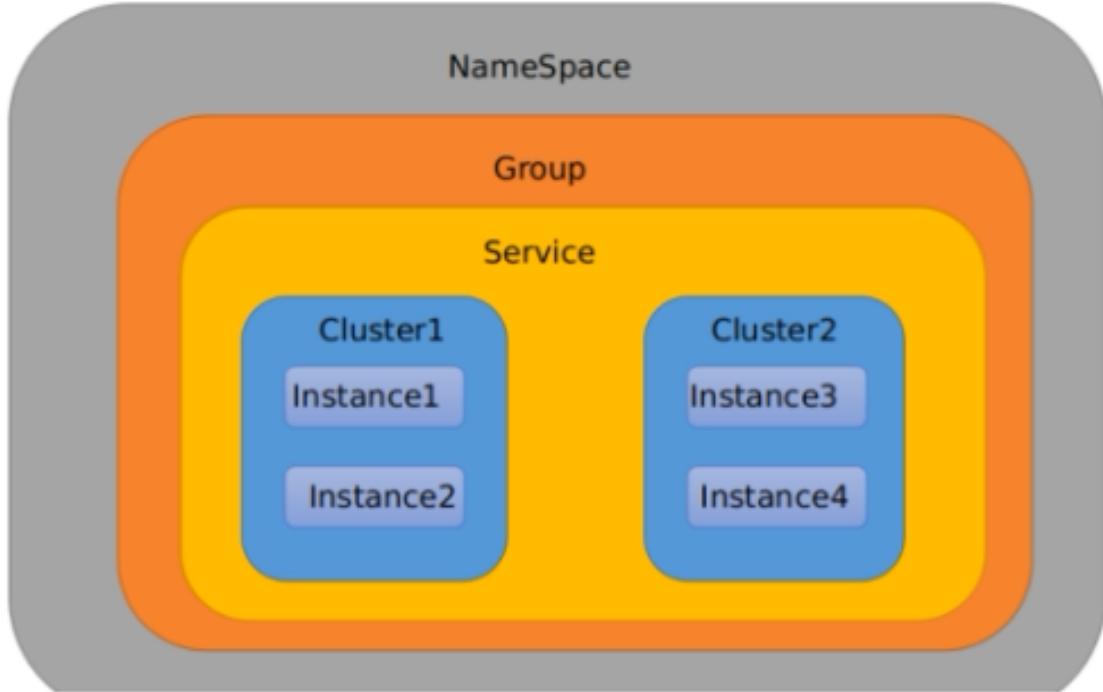
新建命名空间

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		0	详情 删除 编辑

配置管理 服务管理 服务列表 订阅者列表 权限控制 命名空间 集群管理

3. Namespace+Group+Data ID 三者关系？为什么这么设计？

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。



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

- Nacos 默认的命名空间是 public, Namespace 主要用来实现隔离。

比方说我们现在有三个环境：**开发、测试、生产环境**，我们就可以创建**三个 Namespace**，不同的 Namespace 之间是隔离的。

- Group 默认是 **DEFAULT_GROUP**, Group 可以把不同的微服务划分到同一个分组里面去。Service 就是微服务；一个 Service 可以包含多个 Cluster(集群)，Nacos 默认 Cluster 是 **DEFAULT**，Cluster 是对指定微服务的一个虚拟划分。

比方说为了容灾，将 Service 微服务分别部署在了**杭州机房**和**广州机房**，这时就可以给杭州机房的 Service 微服务起一个集群名称(**HZ**)，给广州机房的 Service 微服务起一个集群名字(**GZ**)，还可以尽量让同一个机房的微服务互相调用，以提升性能。

- 最后是 Instance，就是微服务的实例。

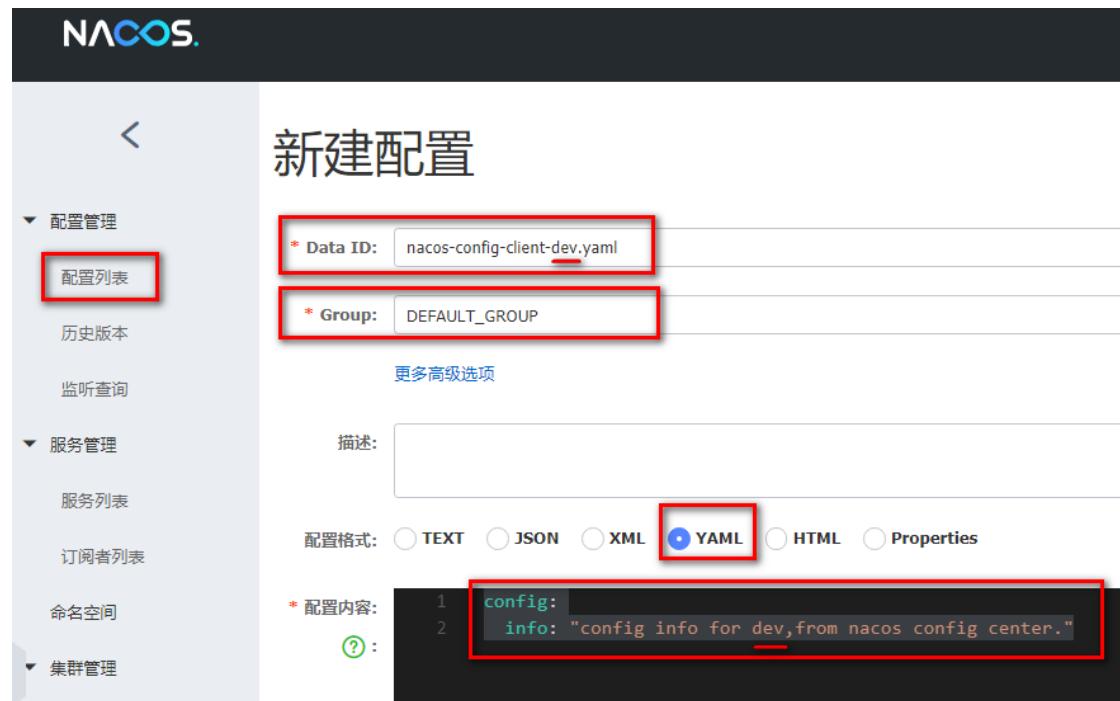
4. Case

1) DataID 方案

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

默认空间+默认分组+**新建 dev 和 test 两个 DataID**

新建 dev 配置 DataID



The screenshot shows the Nacos configuration management interface. On the left sidebar, under '配置管理' (Configuration Management), the '配置列表' (Configuration List) item is selected and highlighted with a red box. In the main content area, the title '新建配置' (Create Configuration) is displayed. The configuration details are as follows:

- * Data ID: nacos-config-client-dev.yaml
- * Group: DEFAULT_GROUP
- 更多高级选项 (More Advanced Options)
- 描述: (Description)
- 配置格式: TEXT, JSON, XML, **YAML** (selected), HTML, Properties. The 'YAML' option is highlighted with a red box.
- * 配置内容: (Configuration Content)

```
1 config:  
2   info: "config info for dev,from nacos config center."
```

新建 test 配置 DataID



The screenshot shows the Nacos configuration management interface. On the left sidebar, under '配置管理' (Configuration Management), the '配置列表' (Configuration List) item is selected and highlighted with a red box. In the main content area, the title '新建配置' (Create Configuration) is displayed. The configuration details are as follows:

- * Data ID: nacos-config-client-test.yaml
- * Group: DEFAULT_GROUP
- 更多高级选项 (More Advanced Options)
- 描述: (Description)
- 配置格式: TEXT, JSON, XML, **YAML** (selected), HTML, Properties. The 'YAML' option is highlighted with a red box.
- * 配置内容: (Configuration Content)

```
1 config:  
2   info: "config info for test,from nacos config center."
```

通过 spring.profile.active 属性就能进行多环境下配置文件的读取

The screenshot shows the IntelliJ IDEA interface with the project 'MainApp3377' open. The left sidebar displays the project structure under 'cloud-config-nacos-client3377'. The right pane shows two configuration files: 'bootstrap.yml' and 'application.yml'. A red box highlights the 'application.yml' file. Red arrows point from the text '配置是什么就加载什么' to the 'active' profiles section in both files. The 'application.yml' file contains the following configuration:

```
1 # Nacos注册配置, application.yml
2 spring:
3   profiles:
4     active: test
5     #active: dev
```

测试

<http://localhost:3377/config/info>

配置是什么就加载什么

localhost:3377/config/info

config info for test,from nacos config center.

2) Group 方案

通过 Group 实现环境区分

新建 Group

The screenshot shows the Nacos Configuration Center interface. On the left, there's a sidebar with '配置管理' (Configuration Management) selected, showing '配置列表' (Configuration List) highlighted with a red box. The main area is titled '新建配置' (Create Configuration). It has fields for 'Data ID' (nacos-config-client-info.yaml), 'Group' (DEV_GROUP), and '描述' (Description: DEV_GROUP Config). Below these are options for '配置格式' (Configuration Format) with 'YAML' selected (highlighted with a red box). The '配置内容' (Configuration Content) field shows the YAML configuration:

```
1 config:
2   info: nacos-config-client-info.yaml DEV_GROUP
```

新建配置

配置管理

配置列表 (选中)

历史版本

监听查询

服务管理

服务列表

订阅者列表

命名空间

集群管理

* Data ID: nacos-config-client-info.yaml

* Group: TEST_GROUP

更多高级选项

描述: TEST_GROUP Config

配置格式: TEXT JSON XML YAML (选中) HTML Properties

* 配置内容:

```
1 config:  
2   info: nacos-config-client-info.yaml TEST_GROUP
```

在 nacos 图形界面控制台上面新建配置文件 DataID

配置管理 | public 查询结果: 共查询到 4 条满足要求的配置。

Data ID: 模糊查询请输入 Data ID Group: 模糊查询请输入 Group

	Data Id	Group
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP
<input type="checkbox"/>	nacos-config-client-test.yaml	DEFAULT_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	DEV_GROUP
<input type="checkbox"/>	nacos-config-client-info.yaml	TEST_GROUP

删除 导出选中的配置 克隆

bootstrap+application

```

bootstrap.yml
1 server:
2   port: 3377
3
4 spring:
5   application:
6     name: nacos-config-client
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #服务注册中心地址
11    config:
12      server-addr: localhost:8848 #配置中心地址
13      file-extension: yaml #指定yaml格式的配置
14      group: TEST_GROUP
15

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

```

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

3) Namespace 方案

新建 dev/test 的 Namespace

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		4 / 200	详情 删除 编辑
test	0330ec04-4083-40d2-81bd-1609a8fc66ec	0 / 200	详情 删除 编辑
dev	9f62d48c-ef2e-4d83-a9fb-c9db5833f93b	0 / 200	详情 删除 编辑

回到服务管理-服务列表查看

public | test | dev **切换不同的命名空间查看微服务**

服务列表 | public

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阀值	操作
nacos-config-client	DEFAULT_GROUP	1	1	1	false	详情 示例代码 删除

创建服务

按照域名配置填写

nacos-config-client-dev.yaml

```
config:  
  info: 9f62d48c-ef2e-4d83-a9fb-c9db5833f93b DEFAULT_GROUP nacos-config-client-dev.yaml
```

public | test | **dev**

配置管理 | dev 9f62d48c-ef2e-4d83-a9fb-c9db5833f93b 查询结果 : 共查询3条

Data ID:	模糊查询请输入Data ID	Group:	模糊查询请输入Group
<input type="checkbox"/>	Data Id	Group	
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEFAULT_GROUP	
<input type="checkbox"/>	nacos-config-client-dev.yaml	DEV_GROUP	
<input type="checkbox"/>	nacos-config-client-dev.yaml	TEST_GROUP	

删除 **导出选中的配置** **克隆**

YML

bootstrap

```
# nacos 配置
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
        group: DEV_GROUP
        namespace: 7d8f0f5a-6a53-4785-9686-dd460158e5d4

# ${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
# nacos-config-client-dev.yaml
```

application

```
spring:
  profiles:
    active: dev # 表示开发环境
    #active: test # 表示测试环境
    #active: info
```

图解：

```
bootstrap.yml
1 server:
2   port: 3377
3
4 spring:
5   application:
6     name: nacos-config-client
7   cloud:
8     nacos:
9       discovery:
10      server-addr: localhost:8848 #服务注册中心地址
11    config:
12      server-addr: localhost:8848 #配置中心地址
13      file-extension: yaml #指定yaml格式的配置
14    group: DEV_GROUP
15    namespace: 9f62d48c-ef2e-4d83-a9fb-c9db5833f93b
16

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

测试结果：

```
localhost:3377/config/info
9f62d48c-ef2e-4d83-a9fb-c9db5833f93b DEV_GROUP nacos-config-client-dev.yaml
```

11.5.Nacos 集群和持久化配置（重要）

11.5.1. 官网说明

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

1. 官网架构图

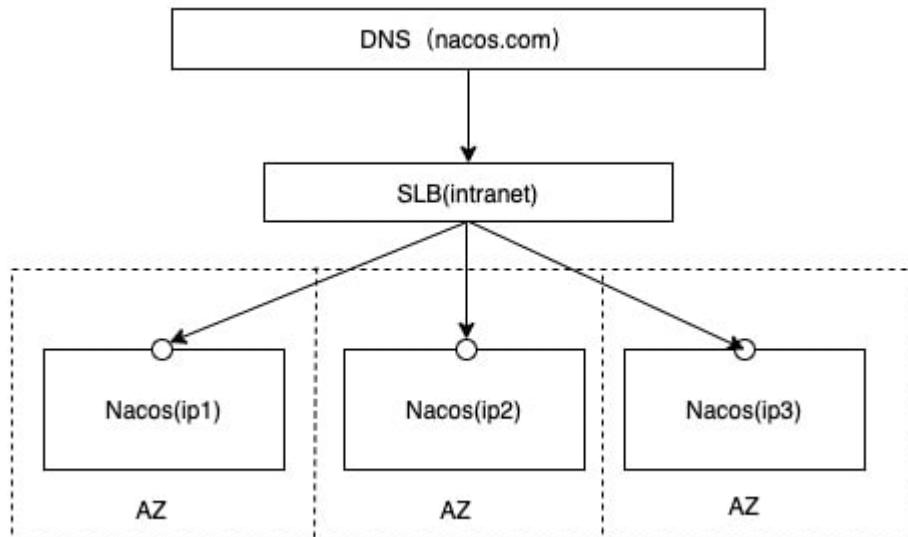
集群部署架构图

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

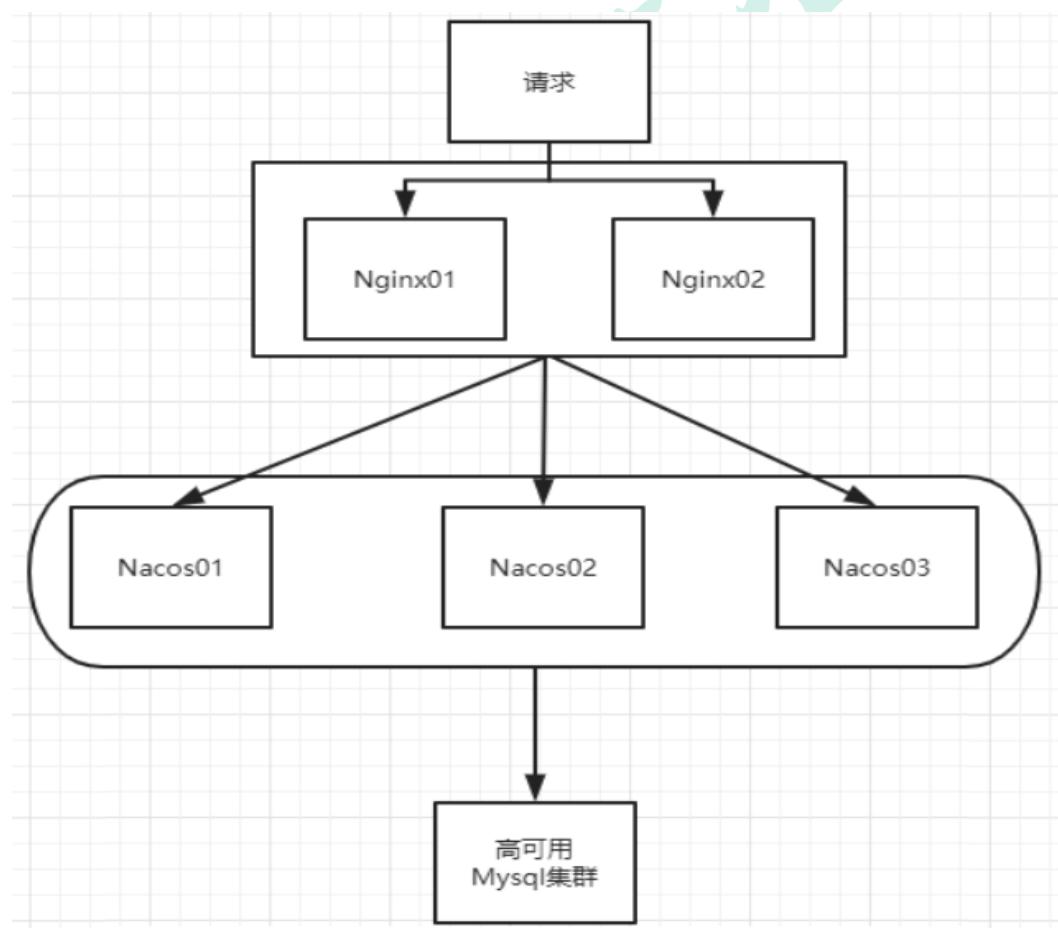
<http://ip1:port/openAPI> 直连 ip 模式，机器挂则需要修改 ip 才可以使用。

<http://SLB:port/openAPI> 挂载 SLB 模式(内网 SLB，不可暴露到公网，以免带来安全风险)，直连 SLB 即可，下面挂 server 真实 ip，可读性不好。

<http://nacos.com:port/openAPI> 域名 + SLB 模式(内网 SLB，不可暴露到公网，以免带来安全风险)，可读性好，而且换 ip 方便，推荐模式



2. 上图官网翻译，真实情况



3. 说明

默认 Nacos 使用嵌入式数据库实现数据的存储。所以，如果启动多个默认配置下的 Nacos 节点，数据存储是存在一致性问题的。

为了解决这个问题，**Nacos 采用了集中式存储的方式来支持集群化部署，目前只支持 MySQL 的存储。**

<https://github.com/alibaba/nacos/blob/master/distribution/conf/application.properties>

<https://github.com/alibaba/nacos/blob/master/distribution/conf/nacos-mysql.sql>

Nacos 支持三种部署模式

- 单机模式 - 用于测试和单机试用。
- 集群模式 - 用于生产环境，确保高可用。

- 多集群模式 - 用于多数据中心场景。

单机模式下运行 Nacos

Linux/Unix/Mac

- Standalone means it is non-cluster Mode. * sh [startup.sh](#) -m standalone

Windows

- Standalone means it is non-cluster Mode. * cmd startup.cmd -m standalone

单机模式支持mysql

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

- 1.安装数据库，版本要求：5.6.5+
- 2.初始化mysql数据库，数据库初始化文件：nacos-mysql.sql
- 3.修改conf/application.properties文件，增加支持mysql数据源配置（目前只支持mysql），添加mysql数据源的url、用户名和密码。

```
spring.datasource.platform=mysql  
db.num=1  
db.url.0=jdbc:mysql://11.162.196.16:3306/nacos_devtest?characterEncoding=utf8&connectTimeout=1000&socketTimeout=30000&autoReconnect=true&useSSL=false  
db.user=nacos_devtest  
db.password=youdontknow
```

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



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

11.5.2. Nacos 持久化配置解释

1. Nacos 默认自带的是嵌入式数据库 **derby**

<https://github.com/alibaba/nacos/blob/develop/config/pom.xml>

2. derby 到 mysql 切换配置步骤

nacos-server-1.4.2\nacos\conf 目录下找到 sql 脚本

nacos-mysql.sql

执行脚本

nacos-server-1.4.2\nacos\conf 目录下找到 **application.properties**

```
spring.datasource.platform=mysql  
  
db.num=1  
db.url.0=jdbc:mysql://localhost:3306/nacos_config?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&serverTimezone=UTC  
db.user=root  
db.password=root
```

3. 启动 nacos，可以看到是个全新的空记录界面，以前是记录进 derby

4. 测试：新建配置，发现配置信息写入了 MySQL 数据库

11.5.3. Linux 版 Nacos+MySQL 生产环境配置

1. 预计需要，1 个 nginx+3 个 nacos 注册中心+1 个 mysql

2. Nacos 下载 linux 版本

1. 预备环境准备

请确保是在环境中安装使用：

1. 64 bit OS Linux/Unix/Mac 推荐使用Linux系统。
2. 64 bit JDK 1.8+；[下载配置](#)。
3. Maven 3.2.x+；[下载配置](#)。
4. 3个或3个以上Nacos节点才能构成集群。

<https://github.com/alibaba/nacos/releases/tag/1.4.2>

nacos-server-1.4.2.tar.gz

3. 集群配置步骤（重点）

1. 创建 /opt/nacoscluster 目录，解压 3 个节点

```
/opt/nacoscluster
[root@atguigu nacoscluster]# ll
总用量 12
drwxr-xr-x. 8 root root 4096 9月 29 14:00 nacos8848
drwxr-xr-x. 8 root root 4096 9月 29 14:02 nacos8849
drwxr-xr-x. 8 root root 4096 9月 29 14:03 nacos8850
```

2.Linux 服务器上 mysql 数据库配置

- 其中一个节点 conf/目录下，找到 nacos_mysql.sql 文件，创建数据库并导入表结构

3.三个节点 conf/application.properties 配置

修改每一个节点下 server.port，分别设置为:8848,8849,8850，并添加以下配置

```
spring.datasource.platform=mysql

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

4. 三个节点/conf 下配置 cluster.conf

```
192.168.137.150:8848
192.168.137.150:8849
192.168.137.150:8850
```

5.启动三个节点前，修改内存大小

启动时，需要修改 startup.sh 文件内存大小，否则，内存可能不够用。

```
93     JAVA_OPT="\${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=256m"
94     JAVA_OPT="\${JAVA_OPT} -XX:-OmitStackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/tmp/nacos-heapdump.hprof"
95     JAVA_OPT="\${JAVA_OPT} -XX:-UseLargePages"
```

6.启动三个节点

```
[root@atguigu bin]# ./startup.sh
/opt/jdk1.8.0_152/bin/java -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=12
or -XX:HeapDumpPath=/opt/nacoscluster/nacos8848/logs/java_heapdump.hprof -XX:-UseLa
_152/lib/ext -Xloggc:/opt/nacoscluster/nacos8848/logs/nacos_gc.log -verbose:gc -XX:
XX:NumberOfGCLogFile=10 -XX:GCLogFileSize=100M -Dloader.path=/opt/nacoscluster/nac
ster/nacos8848 -jar /opt/nacoscluster/nacos8848/target/nacos-server.jar --spring.co
coscluster/nacos8848/conf/nacos-logback.xml --server.max-http-header-size=524288
nacos is starting with cluster
nacos is starting, you can check the /opt/nacoscluster/nacos8848/logs/start.out

[root@atguigu bin]# ./startup.sh
/opt/jdk1.8.0_152/bin/java -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=12
or -XX:HeapDumpPath=/opt/nacoscluster/nacos8849/logs/java_heapdump.hprof -XX:-UseLa
_152/lib/ext -Xloggc:/opt/nacoscluster/nacos8849/logs/nacos_gc.log -verbose:gc -XX:
XX:NumberOfGCLogFile=10 -XX:GCLogFileSize=100M -Dloader.path=/opt/nacoscluster/nac
ster/nacos8849 -jar /opt/nacoscluster/nacos8849/target/nacos-server.jar --spring.co
coscluster/nacos8849/conf/nacos-logback.xml --server.max-http-header-size=524288
nacos is starting with cluster
nacos is starting, you can check the /opt/nacoscluster/nacos8849/logs/start.out

[root@atguigu bin]# ./startup.sh
/opt/jdk1.8.0_152/bin/java -server -Xms256m -Xmx256m -Xmn128m -XX:MetaspaceSize=12
or -XX:HeapDumpPath=/opt/nacoscluster/nacos8850/logs/java_heapdump.hprof -XX:-UseLa
_152/lib/ext -Xloggc:/opt/nacoscluster/nacos8850/logs/nacos_gc.log -verbose:gc -XX:
XX:NumberOfGCLogFile=10 -XX:GCLogFileSize=100M -Dloader.path=/opt/nacoscluster/nac
ster/nacos8850 -jar /opt/nacoscluster/nacos8850/target/nacos-server.jar --spring.co
coscluster/nacos8850/conf/nacos-logback.xml --server.max-http-header-size=524288
nacos is starting with cluster
nacos is starting, you can check the /opt/nacoscluster/nacos8850/logs/start.out
```

查看进程: ps -ef | grep nacos



```
[root@atguigu bin]# ps -ef|grep nacos
root      65386      1 99 14:07 pts/4    00:01:05 /opt/jdk1.8.0_152/bin/java -server
ackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/nacoscluster/.dirs=/opt/jdk1.8.0_152/jre/lib/ext:/opt/jdk1.8.0_152/lib/ext -Xloggc:/opt/nacoscluster/XX:+PrintGCTimeStamps -XX:+UseGLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLog
nacos8850/plugins/cmdb -Dnacos.home=/opt/nacoscluster/nacos8850 -jar /opt/nacoscluster/nacos8850/conf/ --logging.config=/opt/nacoscluster/nacos8850/conf/nacos-log
root      65459      1 99 14:07 pts/3    00:01:00 /opt/jdk1.8.0_152/bin/java -server
ackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/nacoscluster/.dirs=/opt/jdk1.8.0_152/jre/lib/ext:/opt/jdk1.8.0_152/lib/ext -Xloggc:/opt/nacoscluster/XX:+PrintGCTimeStamps -XX:+UseGLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLog
nacos8849/plugins/cmdb -Dnacos.home=/opt/nacoscluster/nacos8849 -jar /opt/nacoscluster/nacos8849/conf/ --logging.config=/opt/nacoscluster/nacos8849/conf/nacos-log
root      65706      1 99 14:07 pts/2    00:00:25 /opt/jdk1.8.0_152/bin/java -server
ackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/nacoscluster/.dirs=/opt/jdk1.8.0_152/jre/lib/ext:/opt/jdk1.8.0_152/lib/ext -Xloggc:/opt/nacoscluster/XX:+PrintGCTimeStamps -XX:+UseGLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLog
nacos8848/plugins/cmdb -Dnacos.home=/opt/nacoscluster/nacos8848 -jar /opt/nacoscluster/nacos8848/conf/ --logging.config=/opt/nacoscluster/nacos8848/conf/nacos-log
root      65986  60491  0 14:08 pts/2    00:00:00 grep --color=auto nacos
```

登录任意节点查看集群：

The screenshot shows the Nacos 1.4.2 web interface. The top navigation bar includes icons for back, forward, refresh, and a warning sign indicating '不安全' (Insecure). The URL is 192.168.6.100:8848/nacos/#/clusterManagement?dataId=&group=&appName=&namespace=.

The left sidebar has a tree structure with 'NACOS 1.4.2' at the top, followed by '配置管理', '服务管理', '命名空间', and '集群管理'. Under '集群管理', there is a '节点列表' node.

The main content area has a header 'public' and a sub-header '节点列表 | public'. It features a search bar with '节点Ip' placeholder and a '查询' button. Below is a table with three rows, each representing a node:

节点Ip	节点状态	节点元数据
192.168.6.100:8848	UP	> 节点元数据
192.168.6.100:8849	UP	> 节点元数据
192.168.6.100:8850	UP	> 节点元数据

7.Nginx 的配置，由它作为负载均衡器

修改 nginx 的配置文件：vim /usr/local/nginx/conf/nginx.conf

nginx.conf

```
upstream nacoscluster{
    server localhost:8848;
    server localhost:8849;
    server localhost:8850;
}

server{
    listen 1111;
    server_name localhost;
    location / {
        proxy_pass http://nacoscluster;
    }
....省略
```

启动 Nginx:

```
[root@atguigu sbin]# ./nginx -c /usr/local/nginx/conf/nginx.conf
[root@atguigu sbin]# ps -ef|grep nginx
root      3766      1  0 11:39 ?        00:00:00 nginx: master process ./nginx -c /usr/local/nginx/conf/nginx.conf
nobody    3767  3766  0 11:39 ?        00:00:00 nginx: worker process
root      3771  3627  0 11:39 pts/2    00:00:00 grep --color=auto nginx
```

6. 截止到此处，1个 Nginx+3个 nacos 注册中心+1个 mysql

测试通过 nginx 访问 nacos

<http://192.168.6.100:1111/nacos>

← → ⌛ ⏺ 🔍 不安全 | 192.168.6.100:1111/nacos/#/clusterManagement?dataId=&group=&appName=&namespace=

NACOS.

NACOS 1.4.2

public

节点列表 | public

节点IP: 查询

节点IP	节点状态	节点元数据
192.168.6.100:8848	UP	> 节点元数据
192.168.6.100:8849	UP	> 节点元数据
192.168.6.100:8850	UP	> 节点元数据

新建一个配置测试

配置详情

* Data ID:

* Group:

更多高级选项

描述:

* MD5:

* 配置内容:

```
1 config:
  2   info: test
```

linux 服务器的 mysql 插入一条记录

```
mysql> SELECT * FROM config_info \G;
***** 1. row *****
    id: 1
  data_id: nacos-config-client-dev.yaml
group_id: DEFAULT_GROUP
  content: config:
info: test
      md5: 7d6ace71e952a1434b16b450f66fa6ba
gmt_create: 2021-09-29 02:07:00
gmt_modified: 2021-09-29 02:07:00
src_user: NULL
  src_ip: 192.168.137.150
app_name:
tenant_id: 49c0e556-7fee-444f-b15a-92da96be1112
  c_desc: NULL
  c_use: NULL
effect: NULL
      type: yaml
  c_schema: NULL
1 row in set (0.00 sec)
```

5. 测试微服务从配置中心获取配置

微服务 nacos-config-client 启动注册进 nacos 集群
Yml

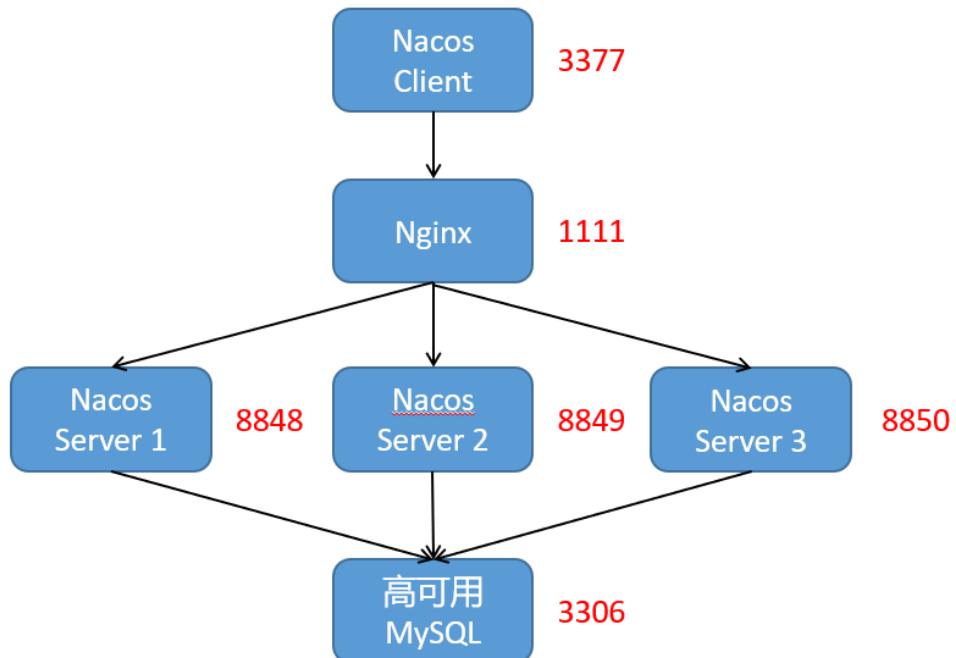
```
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        #server-addr: localhost:8848 #服务注册中心地址
        server-addr: 192.168.137.150:1111
      config:
        #server-addr: localhost:8848 #配置中心地址
        server-addr: 192.168.137.150:1111 #配置中心地址
        file-extension: yaml #指定yaml格式的配置(yml和yaml都可以)
        group: DEFAULT_GROUP
      namespace: 49c0e556-7fee-444f-b15a-92da96be1112
```

访问 URL:

← → C ⌂ ⓘ localhost:3377/config/info

test

6. 高可用小总结



12. SpringCloud Alibaba Sentinel 实现熔断与限流

12.1. Sentinel 介绍

12.1.1. 官网

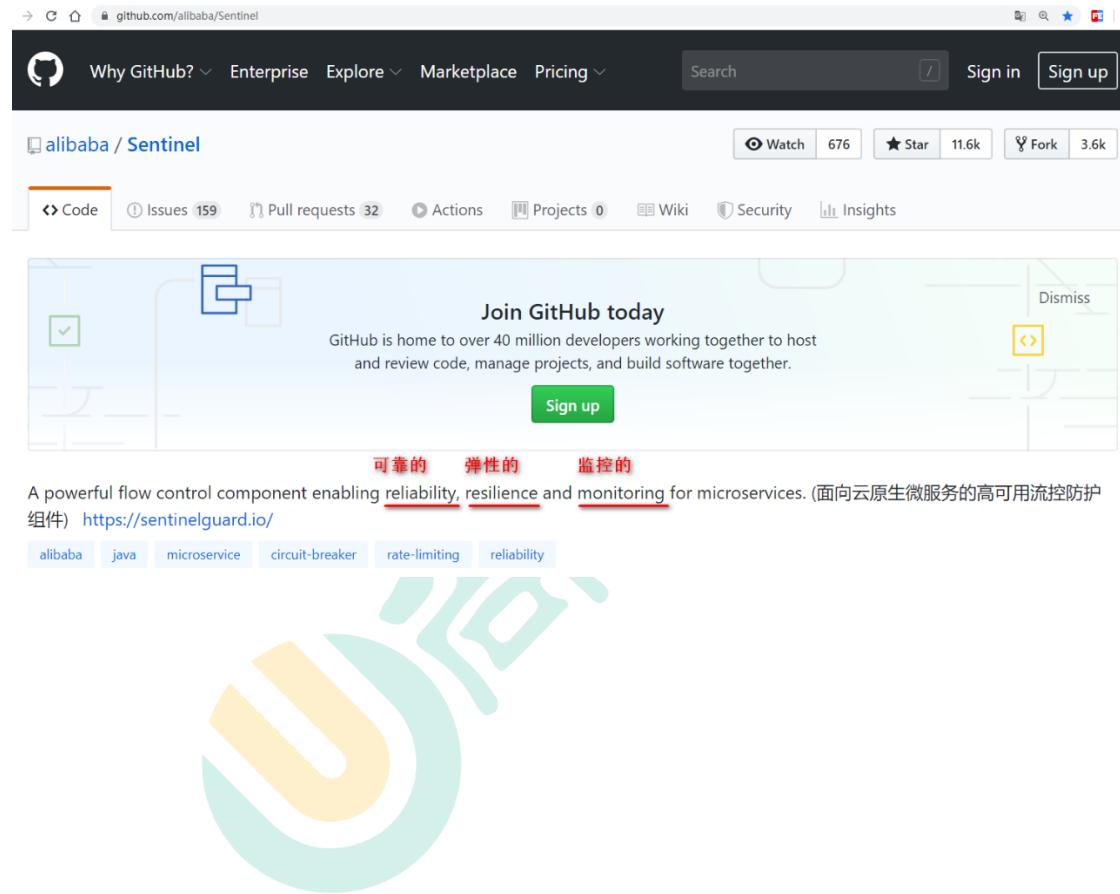
<https://github.com/alibaba/Sentinel>

中文

<https://github.com/alibaba/Sentinel/wiki/%E4%BB%8B%E7%BB%8D>

12.1.2. 是什么

一句话解释，之前我们讲解过的 Hystrix





Sentinel: 分布式系统的流量防卫兵

Sentinel 是什么？

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

Sentinel 具有以下特征：

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

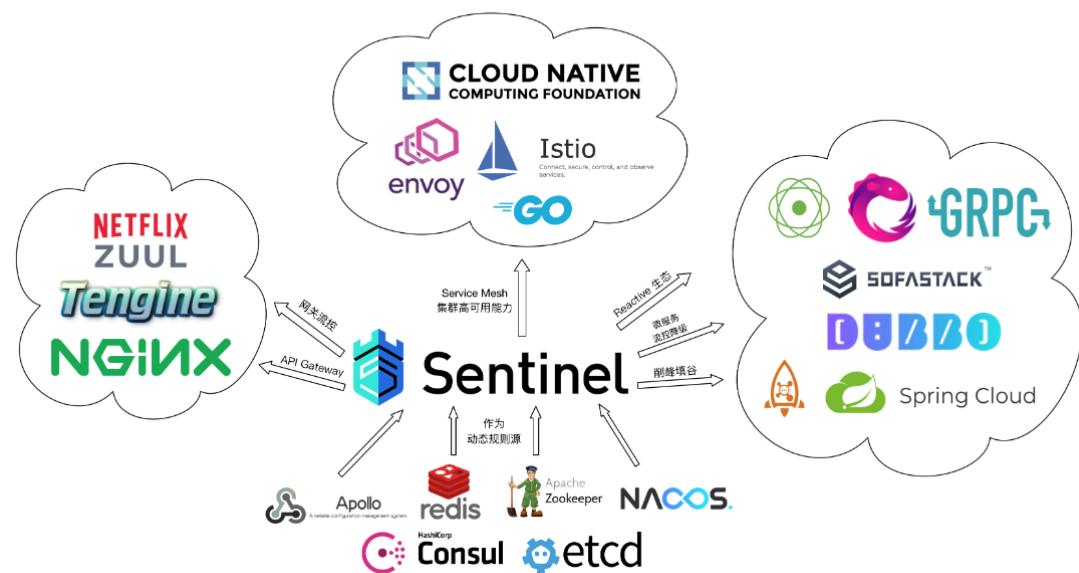


12.1.3. 能干嘛

Sentinel 的主要特性：



Sentinel 的开源生态：

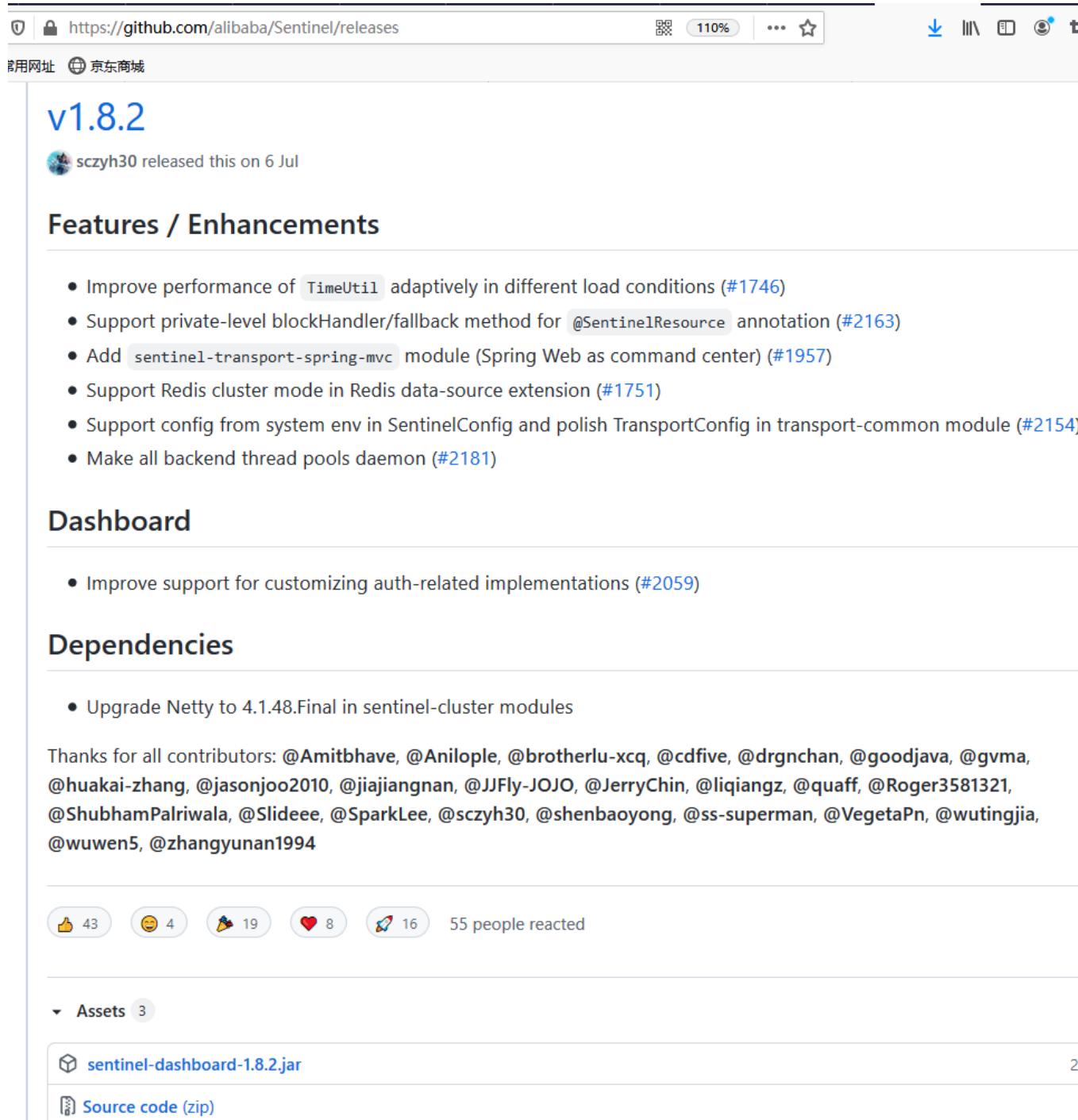


Sentinel 分为两个部分：

- 核心库 (Java 客户端) 不依赖任何框架/库，能够运行于所有 Java 运行时环境，同时对 Dubbo / Spring Cloud 等框架也有较好的支持。
- 控制台 (Dashboard) 基于 Spring Boot 开发，打包后可以直接运行，不需要额外的 Tomcat 等应用容器。

12.1.4. 去哪下

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



v1.8.2

sczyh30 released this on 6 Jul

Features / Enhancements

- Improve performance of `TimeUtil` adaptively in different load conditions ([#1746](#))
- Support private-level blockHandler/fallback method for `@SentinelResource` annotation ([#2163](#))
- Add `sentinel-transport-spring-mvc` module (Spring Web as command center) ([#1957](#))
- Support Redis cluster mode in Redis data-source extension ([#1751](#))
- Support config from system env in SentinelConfig and polish TransportConfig in transport-common module ([#2154](#))
- Make all backend thread pools daemon ([#2181](#))

Dashboard

- Improve support for customizing auth-related implementations ([#2059](#))

Dependencies

- Upgrade Netty to 4.1.48.Final in sentinel-cluster modules

Thanks for all contributors: @Amitbhave, @Anilople, @brotherlu-xcq, @cdfive, @drgnchan, @goodjava, @gvma, @huakai-zhang, @jasonjoo2010, @jiajiangnan, @JJFly-JOJO, @JerryChin, @liqiangz, @quaff, @Roger3581321, @ShubhamPalriwala, @Slideee, @SparkLee, @sczyh30, @shenbaoyong, @ss-superman, @VegetaPn, @wutingjia, @wuwen5, @zhangyunan1994

43 4 19 8 16 55 people reacted

Assets 3

- [sentinel-dashboard-1.8.2.jar](#)
- [Source code \(zip\)](#)

12.1.5. 怎么玩

https://spring-cloud-alibaba-group.github.io/github-pages/hoxton/en-us/index.html#_spring_cloud_alibaba_sentinel

5.2. How to Use Sentinel

If you want to use Sentinel in your project, please use the starter with the group ID as `com.alibaba.cloud` and the artifact ID as `spring-cloud-starter-alibaba-sentinel`.

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

- 服务使用中的各种问题
 - 服务雪崩
 - 服务降级
 - 服务熔断
 - 服务限流

12.2. 安装 Sentinel 控制台

12.2.1. sentinel 组件由 2 部分组成

Sentinel 分为两个部分：

核心库（Java 客户端）不依赖任何框架/库，能够运行于所有 Java 运行时环境，同时对 Dubbo / Spring Cloud 等框架也有较好的支持。

控制台（Dashboard）基于 Spring Boot 开发，打包后可以直接运行，不需要额外的 Tomcat 等应用容器。

- 后台
- 前台 8080

12.2.2. 安装步骤

1. 下载

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

下载到本地 sentinel-dashboard-1.8.2.jar

2. 运行命令

前提

java8 环境 OK

8080 端口不能被占用

命令

java -jar sentinel-dashboard-1.8.2.jar

3. 访问 sentinel 管理界面

<http://localhost:8080>

登录账号密码均为 sentinel

12.3. 初始化演示工程

12.3.1. 启动 Nacos8848 成功

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

12.3.2. 案例

1. 创建 Module: cloudalibaba-sentinel-service8401

2. POM

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

    <parent>
        <artifactId>cloud2021</artifactId>
        <groupId>com.atguigu.springcloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>cloudalibaba-sentinel-service8401</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.atguigu</groupId>
            <artifactId>cloud-api-commons</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba.csp</groupId>
            <artifactId>sentinel-datasource-nacos</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
        </dependency>
    </dependencies>

```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<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.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>4.6.3</version>
</dependency>
<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>
</dependency>
</dependencies>
</project>
```

3. YML

```
server:
```

```
port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
  sentinel:
    transport:
      dashboard: localhost:8080
      port: 8719 #默认 8719, 应用与 Sentinel 控制台交互的端口, 应用本地会起一个该端口占用 HttpServer

  management:
    endpoints:
      web:
        exposure:
          include: '*'
```

配置控制台信息

```
application.yml

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

这里的 `spring.cloud.sentinel.transport.port` 端口配置会在应用对应的机器上启动一个 Http Server , 该 Server 会与 Sentinel 控制台做交互。比如 Sentinel 控制台添加了一个限流规则 , 会把规则数据 push 给这个 Http Server 接收 , Http Server 再将规则注册到 Sentinel 中。

<https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html#production-ready-endpoints>

* 可以用来表示所有的端点 , 例如 , 通过HTTP公开所有的端点 , 除了env和beans端点 , 使用如下的属性 :

```
1 | management.endpoints.web.exposure.include=*
2 | management.endpoints.web.exposure.exclude=env,beans
```

4. 主启动

```
package com.atguigu.springcloud.alibaba;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class MainApp8401{
    public static void main(String[] args) {
        SpringApplication.run(MainApp8401.class, args);
    }
}
```

5. 业务类 FlowLimitController

```
package com.atguigu.springcloud.alibaba.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

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

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

12.3.3. 启动 Sentinel8080

```
java -jar sentinel-dashboard-1.8.2.jar
```

12.3.4. 启动微服务 8401

12.3.5. 启动 8401 微服务后查看 sentinel 控制台

空空如也，啥都没有

Sentinel 采用的**懒加载**说明

执行一次访问即可

<http://localhost:8401/testA>

<http://localhost:8401/testB>

效果

← → ⌂ ⌂ ① localhost:8080/#/dashboard/metric/cloudalibaba-sentinel-service

Sentinel 控制台 1.8.2

用户名 搜索

首页

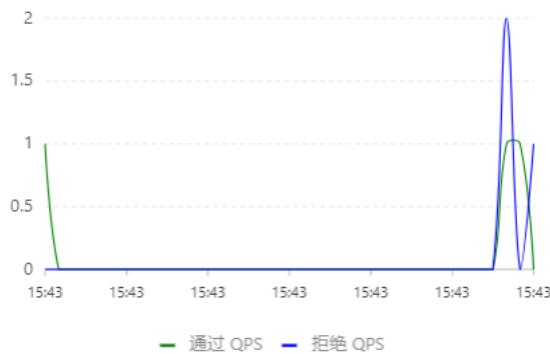
cloudalibaba-sentinel-service (1/1) ▾

- 实时监控
- 簇点链路
- 流控规则
- 熔断规则
- 热点规则
- 系统规则
- 授权规则
- 集群流控
- 机器列表

cloudalibaba-sentinel-service

实时监控

/testA



时间	通过 QPS
15:43:56	0.0
15:43:55	1.0
15:43:54	1.0
15:43:20	1.0
-	-
-	-

结论

sentinel8080 正在监控微服务 8401

12.4. 流控规则

12.4.1. 基本介绍



进一步解释说明

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

12.4.2. 流控模式

1. 直接（默认）

- 直接->快速失败
 - 系统默认
- 测试 QPS
 - 配置及说明
 - 表示 1 秒钟内查询 1 次就是 OK, 若超过次数 1, 就直接-快速失败, 报默认错误



- 快速点击访问: <http://localhost:8401/testA>
- 结果
 - Blocked by Sentinel (flow limiting)
- 测试线程数



- 快速点击访问: <http://localhost:8401/testA>
- 结果
 - 不会出现 Blocked by Sentinel (flow limiting) (线程处理请求很快)

```
@GetMapping("/testA")
public String testA() {
    try {
        Thread.sleep( millis: 800 );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "-----testA";
}
```

但是，在映射方法里添加 sleep 后，同样也会出现 Blocked by Sentinel (flow limiting) 默认提示信息。

- 思考？？？
 - 直接调用默认报错信息，技术方面 OK but，是否应该有我们自己的后续处理？
 - 类似有一个 fallback 的兜底方法？

2. 关联

- 是什么?
 - 当关联的资源达到阈值时，就限流自己

- 当与 A 关联的资源 B 达到阈值后，就限流自己
- B 惹事，A 挂了

3. 配置 A

- 设置效果：

- 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闲置配置的的资源名。



4. postman 模拟并发密集访问 testB

- 访问 testB 成功

GET http://localhost:8401/testB

http://localhost:8401/testB

Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (5) Test Results Status: 200 OK Time: 10ms Size: 175 B Save Response

Pretty Raw Preview Visualize Text

1 -----testB

- postman 里新建多线程集合组，将请求保存到集合组

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests). Learn more about creating collections.

Request name: http://localhost:8401/testB

Request description (Optional)

Select a collection or folder to save to:

Collection2020.4.8 + Create Folder

Cancel Save to Collection2020.4.8

Send Save

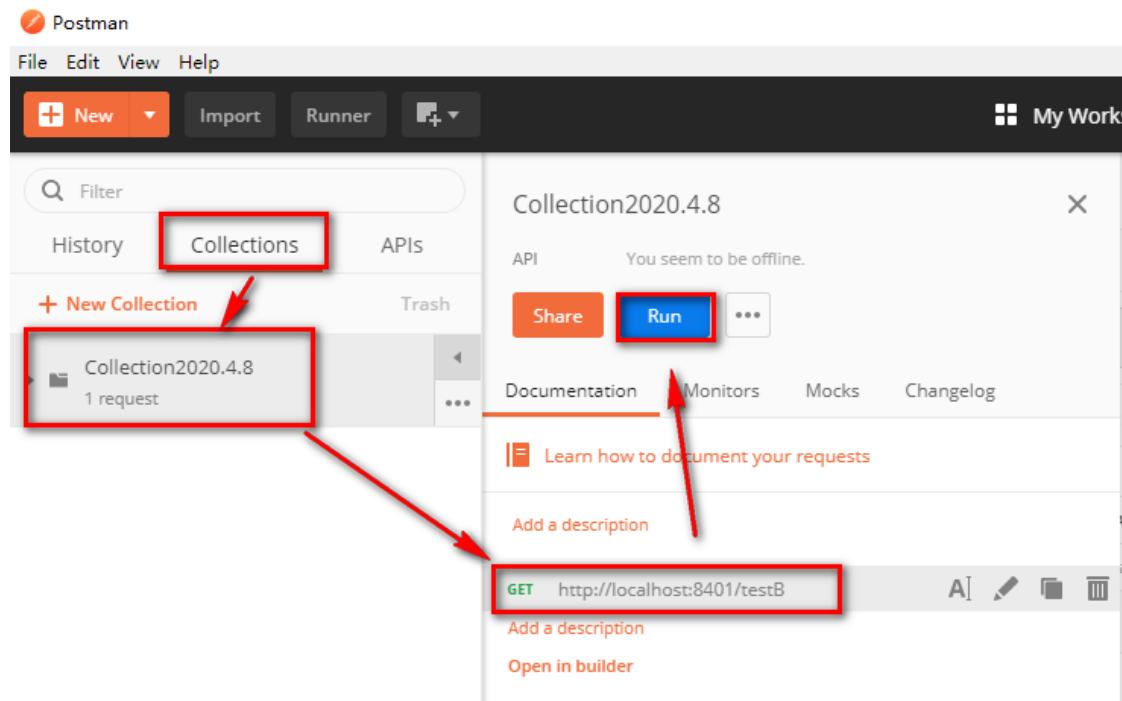
No Environment Comments 0 Examples 0

DESCRIPTION

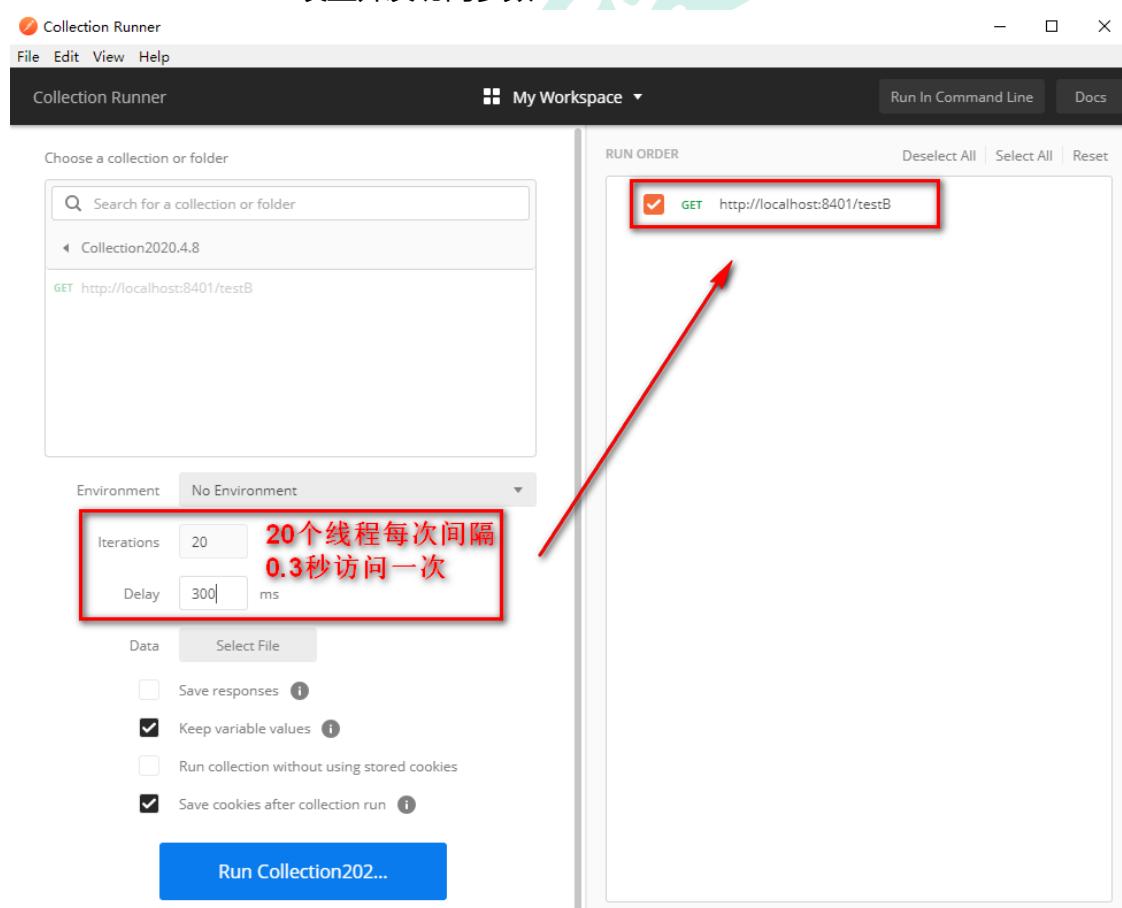
Description

Status: 200 OK Time: 12ms Size: 175 B Save Response

- 运行线程集合组



▪ 设置并发访问参数



- Run
 - 大批量线程高并发访问 B，导致 A 失效了
- 运行后发现 testA 挂了
 - 点击访问 <http://localhost:8401/testA>
 - 结果
 - Blocked by Sentinel (flow limiting)

5. 链路

- 多个请求调用了同一个微服务
- 家庭作业试试

12.4.3. 流控效果

1. 直接->快速失败（默认的流控处理）

- 直接失败，抛出异常：Blocked by Sentinel (flow limiting)
- 源码：
`com.alibaba.csp.sentinel.slots.block.flow.controller.DefaultController`

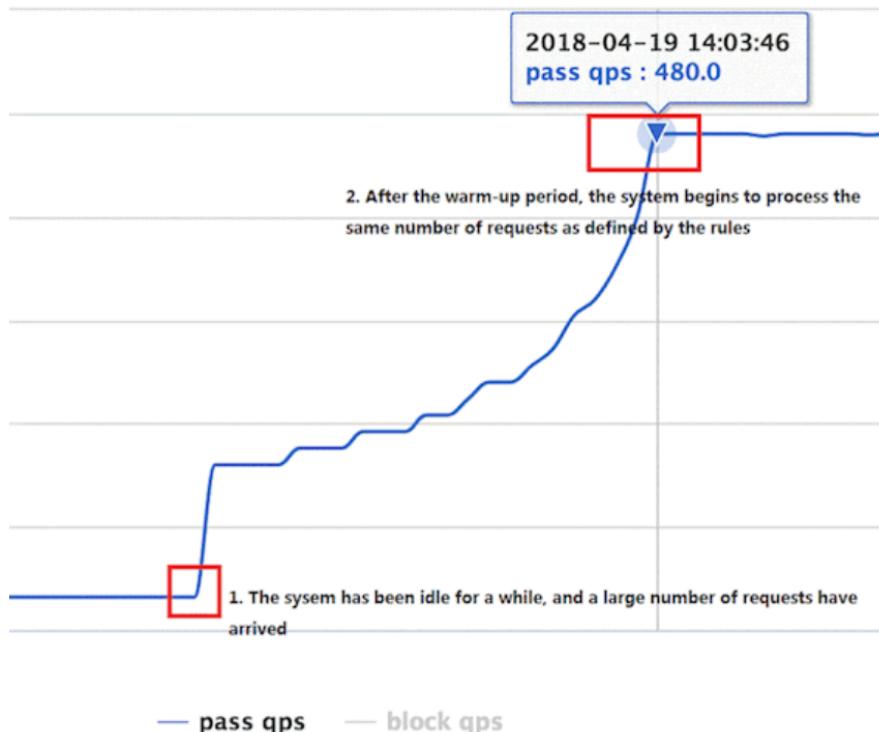
2. 预热

- 说明
 - 公式：阈值除以 coldFactor（默认值为 3），经过预热时长后才会达到阈值
- 官网：
<https://github.com/alibaba/Sentinel/wiki/%E6%B5%81%E9%87%8F%E6%A7%E5%88%B6#%E5%9F%BA%E4%BA%8Eqps%E5%B9%B6%E5%8F%91%E6%95%B0%E7%9A%84%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6>

Warm Up

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

通常冷启动的过程系统允许通过的 QPS 曲线如下图所示：



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。

- 限流 冷启动
<https://github.com/alibaba/Sentinel/wiki/%E9%99%90%E6%B5%81---%E5%86%B7%E5%90%AF%E5%8A%A8>
- 源码
 - com.alibaba.csp.sentinel.slots.block.flow.controller.WarmUpController

```
public WarmUpController(double count, int warmUpPeriodInSec) {  
    construct(count, warmUpPeriodInSec, coldFactor: 3);  
}
```

- Warmup 配置

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。

 - 案例：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，即阈值刚开始为 3；然后过了 5 秒后阈值才慢慢升高，恢复到 10



- 多次点击 <http://localhost:8401/testB>
- 刚开始不行，后续慢慢 OK
- 应用场景
 - 如：秒杀系统在开启的瞬间，会有很多流量上来，很有可能把系统打死，预热方式就是为了保护系统，可慢慢的把流量放进来，慢慢的把阈值增长到设置的阈值。

3. 排队等待

- 匀速排队，让请求以均匀的速度通过，阈值类型必须设置成 QPS，否则无效。
- 设置含义：/testB 每秒 1 次请求，超过的话就排队等待，等待的超时时间为 20000 毫秒。

编辑流控规则

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

[关闭高级选项](#)

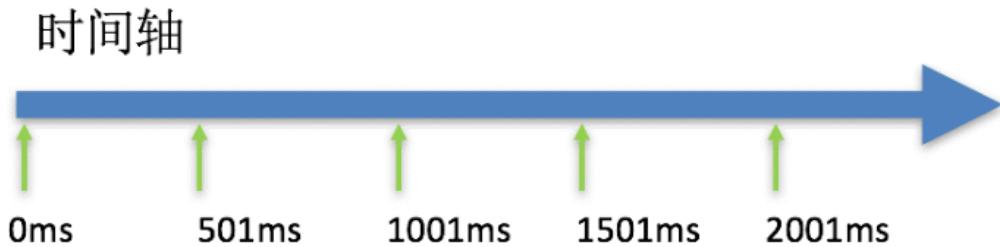
[保存](#) [取消](#)

- 官网

匀速排队

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

该方式的作用如下图所示：



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。

- 源码：
com.alibaba.csp.sentinel.slots.block.flow.controller.RateLimiterController
- 测试
 - 增加打印语句

```
@GetMapping("/testB")
public String testB() {
    log.info(Thread.currentThread().getName()+"\t ...testB");
    return "-----testB";
}
```

- 增加线程组：直接 10 个线程并发，排队被依次处理

The screenshot shows the 'Collection Runner' interface. On the left, there's a search bar and a list of collections, with 'Collection2020.4.8' selected. On the right, under 'RUN ORDER', a 'GET http://localhost:8401/testB' request is selected. A red arrow points from the 'Iterations' field in the configuration area below to this selection. The configuration area includes fields for 'Iterations' (set to 10) and 'Delay' (set to 0 ms), both of which are highlighted with a red box. Other configuration options like 'Save responses', 'Keep variable values', and 'Save cookies after collection run' are also visible.

12.5. 降级规则

12.5.1. 官网

<https://github.com/alibaba/Sentinel/wiki/%E7%86%94%E6%96%AD%E9%99%8D%E7%BA%A7>

hub.fastgit.org/alibaba/Sentinel/wiki/熔断降级

熔断策略

Sentinel 提供以下几种熔断策略：

- 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），当单位统计时长 (`statIntervalMs`) 内请求数目大于设置的最小请求数目的比例大于阈值，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。
- 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，并且异常率大于阈值，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 $[0.0, 1.0]$ 。
- 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。

12.5.2. 基本介绍

- 整体介绍

新增熔断规则

资源名: /testA

熔断策略: 慢调用比例 异常比例 异常数

最大 RT: RT (毫秒) 比例阈值: 取值 [0.0, 1.0]

熔断时长: 熔断时长(s) s 最小请求数: 最小请求数目

统计时长: 统计时长(ms) ms

新增并继续添加 新增 取消

编辑熔断规则

资源名: /testA

熔断策略: 慢调用比例 异常比例 异常数

比例阈值: 取值范围 [0.0,1.0]

熔断时长: 熔断时长(s) s 最小请求数: 最小请求数目

统计时长: 统计时长(ms) ms

保存 取消



12.5.3. 降级策略实战

1. 慢调用比例

- 是什么

- 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。

- 测试

- 代码

```
@GetMapping("/testA")
public String testA() {

    try {
        Thread.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
    }  
  
    return "-----testA";  
  
}
```

▪ 配置



- 访问测试: <http://localhost:8401/testA>
- 5 秒内打进 10 个请求, 由于每次请求都大于 RT, 并且比例阈值 100%, 所以, 熔断器打开。

← → C ⌂ ⓘ localhost:8401/testA

Blocked by Sentinel (flow limiting)

2. 异常比例

• 是什么

- 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) 下面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] 100%。

• 测试

- 代码

```
@GetMapping("/testB")
public String testB() {
    int age = 10/0;
    return "-----testB";
}
```

- 配置



- 访问测试: <http://localhost:8401/testB>
- 5 秒内打进 10 个请求，由于每次请求都抛异常，异常比例阈值 100% 超过 50%，所以，熔断器打开，10s 后半开。如果再次访问有异常，则继续熔断。

← → C ⌂ ⓘ localhost:8401/testB

Blocked by Sentinel (flow limiting)

3. 异常数

- 是什么

- 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。

- 测试

- 代码

```
@GetMapping("/testB")
public String testB(){
    int age = 10/0;
    return "-----testB 测试异常数";
}
```

- 配置

编辑熔断规则

资源名	/testB		
熔断策略	<input type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input checked="" type="radio"/> 异常数		
异常数	5		
熔断时长	10 s	最小请求数	10
统计时长	5000 ms		

保存 取消

- 访问测试: <http://localhost:8401/testB>
- 5 秒内打进 10 个请求, 由于每次请求都抛异常, 异常数超过 5 个, 所以, 熔断器打开, 10s 后半开。如果再次访问有异常, 则继续熔断。

← → C ⌂ ⓘ localhost:8401/testB

Blocked by Sentinel (flow limiting)

12.6.热点 key 限流

12.6.1. 基本介绍

- 是什么



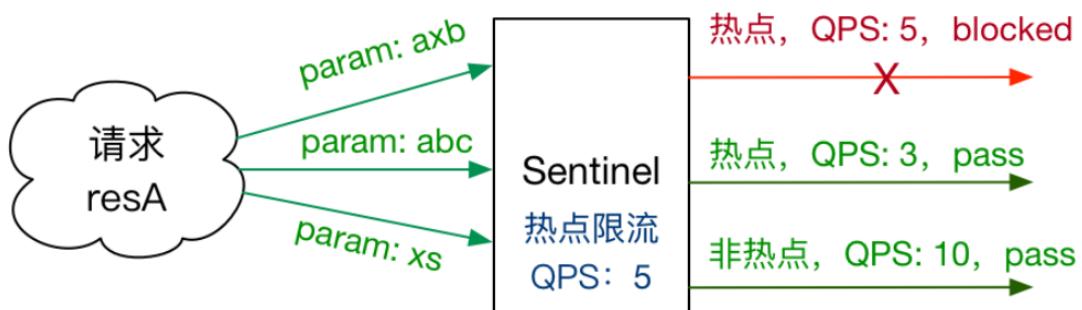
12.6.2. 官网

<https://github.com/alibaba/Sentinel/wiki/热点参数限流>

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

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

热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



Sentinel 利用 LRU 策略统计最近最常访问的热点参数，结合令牌桶算法来进行参数级别的流控。热点参数限流支持集群模式。

12.6.3. 承上启下复习

- 壳底方法
- 分为系统默认和客户自定义，两种
 - 之前的 case，限流出问题后，都是用 sentinel 系统默认的提示：**Blocked by Sentinel(flow limiting)**
 - 我们能不能自定义？类似 hystrix,某个方法出现问题了，就找对应的壳底降级方法？
 - 结论
 - **从@HystrixCommand 到@SentinelResource**

12.6.4. 代码

```
@GetMapping("/testHotKey")
@SentinelResource(value = "testHotKey",blockHandler = "deal_testHotKey")
public String testHotKey(@RequestParam(value = "p1",required = false) String p1,
                        @RequestParam(value = "p2",required = false) String p2) {
    //int age = 10/0;
    return "-----testHotKey";
}

//壳底方法
public String deal_testHotKey (String p1, String p2, BlockException exception){
    return "-----deal_testHotKey,o(╥﹏╥)o";
}
```

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

12.6.5. 配置

- 配置

新增热点规则

资源名	testHotKey		
限流模式	QPS 模式		
参数索引	0		
单机阈值	1	统计窗口时长	1 秒
是否集群	<input type="checkbox"/>		
高级选项			
<input type="button" value="新增"/> <input type="button" value="取消"/>			

热点参数限流规则

资源名	参数索引	流控模式	阈值	是否集群	例外项目数	操作
testHotKey	0	QPS	1	否	0	<input type="button" value="编辑"/> <input type="button" value="删除"/>

○ 默认

- @SentinelResource(value = "testHotKey")
- 异常打到了前台用户界面，不友好

localhost:8401/testHotKey?p1=a

Whitelabel Error Page

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

Tue Feb 25 10:59:39 CST 2020

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

No message available

```
java.lang.reflect.UndeclaredThrowableException
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:757)
    at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:93)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186)
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.proceed(CglibAopProxy.java:747)
    at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:689)
    at com.atguigu.springcloud.alibaba.controller.FlowLimitController$$EnhancerBySpringCGLIB$$27a2a4e4.testHotKey(<generated>)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
```

○ 自定义

- @SentinelResource(value = "testHotKey",
blockHandler = "deal_testHotKey") //value 值与资源名一致即可
- 方法 testHostKey 里面第一个参数只要 QPS 超过每秒 1 次，马上降级处理

- 测试
 - error (1秒1下可以, 但是, 超过则降级, 和 p1 参数有关)
<http://localhost:8401/testHotKey?p1=abc>
 - error (1秒1下可以, 但是, 超过则降级, 和 p1 参数有关)
<http://localhost:8401/testHotKey?p1=abc&p2=33>
 - right (狂点不会触发降级, 与 p2 参数无关)
<http://localhost:8401/testHotKey?p2=abc>

12.6.6. 参数例外项

- 上述案例演示了第一个参数 p1, 当 QPS 超过 1 秒 1 次点击后马上被限流
- 特殊情况
 - 普通
 - 超过 1 秒钟一个后, 达到阈值 1 后马上被限流
 - 我们期望 p1 参数当它是某个特殊值时, 它的限流值和平时不一样
 - 特例
 - 假如当 p1 的值等于 5 时, 它的阈值可以达到 200
- 配置
 - 添加按钮不能忘

编辑热点规则

`@SentinelResource(value="testHotKey".blockHandler = "deal_testHotKey")`

资源名	testHotKey
限流模式	QPS 模式
参数索引	0 限制索引为0的参数, 每秒访问量是1次
单机阈值	1
统计窗口时长	1 秒

是否集群

参数例外项

参数类型	java.lang.String 只能是简单类型
参数值	5 索引为0的参数, 参数值为5时, 每秒访问量限制200次
限流阈值	200
<button>+添加</button>	

参数值	参数类型	限流阈值	操作
5	java.lang.String	200	删除

[关闭高级选项](#)

[保存](#) [取消](#)

- 测试

<http://localhost:8401/testHotKey?p1=5> 对
<http://localhost:8401/testHotKey?p1=3> 错

- 当 p1 等于 5 的时候, 阈值变为 200
- 当 p1 不等于 5 的时候, 阈值就是平常的 1

- 前提条件

- 热点参数的注意点, 参数必须是基本类型或者 String

参数例外项

参数类型

参数值

参数值

5

int
double
java.lang.String
long
float
char
byte

12.6.7. 其他

- 手贱添加异常看看....
- @SentinelResource
 - 处理的是 Sentinel 控制台配置的违规情况，有 blockHandler 方法配置的兜底处理
- RuntimeException
 - Int age = 10/0;这个是 java 运行时报出的运行时异常 RuntimeException, @SentinelResource 不管
- 总结:
 - @SentinelResource 主管配置出错，运行出错该走异常走异常

12.7. 系统规则

12.7.1. 是什么

<https://github.com/alibaba/Sentinel/wiki/%E7%B3%BB%E7%BB%9F%E8%87%AA%E9%80%82%E5%BA%94%E9%99%90%E6%B5%81>

12.7.2. 各项配置参数说明

系统规则

系统保护规则是从应用级别的入口流量进行控制，从单台机器的 load、CPU 使用率、平均 RT、入口 QPS 和并发线程数等几个维度监控应用指标，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

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

系统规则支持以下的模式：

- **Load 自适应**（仅对 Linux/Unix-like 机器生效）：系统的 load1 作为启发指标，进行自适应系统保护。当系统 load1 超过一定的启发值，且系统当前的并发线程数超过估算的系统容量时才会触发系统保护（BBR 阶段）。系统容量由系统的 `maxQps` `minRt` 估算得出。设定参考值一般是 `CPU cores * 2.5`。
- **CPU usage**（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0），比较灵敏。
- **平均 RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **并发线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。

12.7.3. 配置全局 QPS

The screenshot shows the Sentinel Control Console interface. The main title is "Sentinel 控制台 1.8.2". On the left sidebar, there are several navigation items: 应用名 (Application Name), 搜索 (Search), 首页 (Home), 实时监控 (Real-time Monitoring), 簇点链路 (Cluster Link), 流控规则 (Flow Control Rules), 熔断规则 (Circuit Breaker Rules), 热点规则 (Hotspot Rules), 系统规则 (System Rules), 授权规则 (Authorization Rules), and 帮助和支持 (Help and Support). The current application selected is "cloudalibaba-sentinel-service" (1/1). In the center, there is a search bar with placeholder text "请输入关键字" and a refresh button. Below the search bar, there is a button labeled "+ 新增系统规则" (Add System Rule). A modal dialog titled "新增系统保护规则" (Add System Protection Rule) is open. It contains two input fields: "阈值类型" (Threshold Type) with radio buttons for LOAD (selected), RT, 线程数 (Threads), 入口 QPS (Inbound QPS), and CPU 使用率 (CPU Usage); and "阈值" (Threshold) with a text input field containing "[0, ~)的正整数" (Positive integer range [0, ~)). At the bottom of the dialog are two buttons: "新增" (Add) and "取消" (Cancel).

12.8. @SentinelResource

12.8.1. 按资源名称限流+后续处理

- 启动 Nacos 成功
- 启动 Sentinel 成功
- Module
 - cloudalibaba-sentinel-service8401
 - POM
 - YML

```
server:  
  port: 8401  
  
spring:  
  application:  
    name: cloudalibaba-sentinel-service  
  cloud:  
    nacos:  
      discovery:  
        server-addr: localhost:8848  
    sentinel:  
      transport:  
        dashboard: localhost:8080  
        port: 8719 #默认 8719, 应用与 Sentinel 控制台交互的端口, 应用本地会起一个该端口占用的 HttpServer  
  
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'
```

- 业务类 RateLimitController

```
package com.atguigu.springcloud.alibaba.controller;

import com.alibaba.csp.sentinel.annotation.SentinelResource;
import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.atguigu.springcloud.alibaba.entities.CommonResult;
import com.atguigu.springcloud.alibaba.entities.Payment;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class RateLimitController{
    @GetMapping("/byResource")
    @SentinelResource(value = "byResource",blockHandler = "handleException")
    public CommonResult byResource(){
        return new CommonResult(200,"按资源名称限流测试 OK",new Payment(2020L,"serial001"));
    }
    public CommonResult handleException(BlockException exception){
        return new CommonResult(444,exception.getClass().getCanonicalName() + "\t服务不可用");
    }
}
```

- 主启动

```
package com.atguigu.springcloud.alibaba;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class MainApp8401{
    public static void main(String[] args) {
        SpringApplication.run(MainApp8401.class, args);
    }
}
```

- 配置流控规则

- 配置步骤



- 图形配置和代码关系
- 表示 1 秒钟内查询次数大于 1，就跑到我们自定义的处理，限流
- 测试
 - 1 秒钟点击 1 下，OK
 - 超过上述问题，疯狂点击，返回了自己定义的限流处理信息，限流发生

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

- 额外问题
 - 此时关闭微服务 8401 看看
 - Sentinel 控制台，流控规则消失了？？？？
 - 临时/持久？

12.8.2. 按照 Url 地址限流+后续处理

- 通过访问的 URL 来限流，会返回 Sentinel 自带默认的限流处理信息
- 业务类 RateLimitController

```
@GetMapping("/rateLimit/byUrl")
@SentinelResource(value = "byUrl")
public CommonResult byUrl(){
    return new CommonResult(200,"按 url 限流测试 OK",new
Payment(2020L,"serial002"));
}
```

- 访问一次
- Sentinel 控制台配置



- 测试
 - 疯狂点击 <http://localhost:8401/rateLimit/byUrl>
 - 结果
 - 会返回 Sentinel 自带的限流处理结果

← → ⏪ ⓘ localhost:8401/rateLimit/byUrl

Blocked by Sentinel (flow limiting)

12.8.3. 上面兜底方法面临的问题

- 系统默认的，没有体现我们自己的**业务要求**。
- 依照现有条件，我们自定义的处理方法又和业务代码**耦合**在一起，不直观。
- 每个业务方法都增加一个兜底的，那代码**膨胀**加剧。
- **全局统一**的处理方法没有体现。

12.8.4. 客户自定义限流处理逻辑

- 创建 **customerBlockHandler** 类用于自定义限流处理逻辑
- 自定义限流处理类
 - 方法必须是 **public static** 修饰的。

```
package com.atguigu.springcloud.alibaba.myhandler;

import com.alibaba.csp.sentinel.slots.block.BlockException;
import com.atguigu.springcloud.entities.CommonResult;

public class CustomerBlockHandler {
    public static CommonResult handleException(BlockException exception){
        return new CommonResult(2020,"自定义限流处理信息.... CustomerBlockHandler -- - 1");
    }

    public static CommonResult handleException2(BlockException exception){
        return new CommonResult(2020,"自定义限流处理信息.... CustomerBlockHandler -- - 2");
    }
}
```

- RateLimitController

```
@GetMapping("/rateLimit/customerBlockHandler")
@SentinelResource(value = "customerBlockHandler",
    blockHandlerClass = CustomerBlockHandler.class, blockHandler =
    "handleException2")
public CommonResult customerBlockHandler(){
    return new CommonResult(200,"按客户自定义",new Payment(2020L,"serial003"));
}
```

- 启动微服务后先调用一次

<http://localhost:8401/rateLimit/customerBlockHandler>

- Sentinel 控制台配置
- 测试后我们自定义的出来了
- 进一步说明

```
45 @GetMapping("/rateLimit/customerBlockHandler")
46 @SentinelResource(value = "customerBlockHandler",
47     blockHandlerClass = CustomerBlockHandler.class, blockHandler = "handleException2")
48 public CommonResult customerBlockHandler()
49 {
50     return new CommonResult( code: 200, message: "按客户自定义限流处理逻辑");
51 }
52
53 CustomerBlockHandler.java
54
55 CustomerBlockHandler
56
57 public class CustomerBlockHandler
58 {
59     /**
60      * handleException
61      */
62     public static CommonResult handleException(BlockException exception){
63         return new CommonResult( code: 2020, message: "客户自定义的限流处理信息.....CustomerBlockHandler");
64     }
65
66     /**
67      * handleException2
68      */
69     public static CommonResult handleException2(BlockException exception){
70         return new CommonResult( code: 2020, message: "客户自定义的限流处理信息2.....CustomerBlockHandler--2");
71     }
72 }
```

12.8.5. 更多注解属性说明

<https://github.com/alibaba/Sentinel/wiki/%E6%B3%A8%E8%A7%A3%E6%94%AF%E6%8C%81>

@SentinelResource 注解

注意：注解方式埋点不支持 private 方法。

`@SentinelResource` 用于定义资源，并提供可选的异常处理和 fallback 配置项。`@SentinelResource` 注解包含以下属性：

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

注：1.6.0 之前的版本 `fallback` 函数只针对降级异常（`DegradeException`）进行处理，**不能针对业务异常进行处理**。

特别地，若 `blockHandler` 和 `fallback` 都进行了配置，则被限流降级而抛出 `BlockException` 时只会进入 `blockHandler` 处理逻辑。若未配置 `blockHandler`、`fallback` 和 `defaultFallback`，则被限流降级时会将 `BlockException` 直接抛出（若方法本身未定义 `throws BlockException` 则会被 JVM 包装一层 `UndeclaredThrowableException`）。

```
/**  
 * @SentinelResource 与 Hystrix 组件中的@HystrixCommand 注解作用类似的。  
 *      value = "byResourceName" 用于设置资源名称，只有根据资源名称设置的规则，才能执行 blockHandler 所引用降级方法。  
 */
```

```
* 如果按照映射路径进行规则配置，返回默认降级消息：Blocked by Sentinel  
(flow limiting)  
  
* blockHandler 用于引用降级方法。  
  
* blockHandlerClass 用于引用降级方法的处理器类。注意：降级方法必须是  
static 的。否则，无法解析  
  
* blockHandler + blockHandlerClass 只处理配置违规，进行降级处理。代码出现  
异常，不执行的。  
  
*  
  
* blockHandler + fallback 同时存在，配置违规，代码也有异常，这时，走  
blockHandler 配置文件降级处理  
  
*  
  
* exceptionsToIgnore 设置特定异常不需要降级处理。  
  
*/  
  
@RequestMapping("/fallback/{id}")  
  
@SentinelResource(value = "byFallbackName", blockHandler =  
"handleException3",  
  
blockHandlerClass = RateLimitControllerHandler.class,  
  
fallback = "handleException2", fallbackClass =  
RateLimitControllerHandler.class,  
  
exceptionsToIgnore=IllegalArgumentException.class  
)  
  
public CommonResult<Payment> fallback(@PathVariable("id") Long id) {  
  
    if (id == 4) {  
  
        throw new IllegalArgumentException ("IllegalArgumentException,非法参数异常....");  
    }  
  
    if (id== -1) {  
  
        CommonResult<Payment> result = new CommonResult<>(444,"数据不存在  
,null);
```

```
        throw new NullPointerException ("NullPointerException,该 ID 没有对应记录,  
空指针异常");  
  
    }  
  
    CommonResult<Payment> result = new CommonResult<>(200,"数据已经获取  
",new Payment(id,"test"+1));  
  
    return result;  
  
}
```

12.9. 熔断框架比较

	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离 (并发线程数限流)	线程池隔离/信号量隔 离	信号量隔高
熔断降级策略	基于响应时间、异常比率、异常数	基于异常比率	基于异常比率、响应时 间
实时统计实现	滑动窗口 (LeapArray)	滑动窗口 (基于 RxJava)	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
流量整形	支持预热模式、匀速器模式、预热排队模 式	不支持	简单的 Rate Limiter 模式
系统自适应保护	支持	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、查 看秒级监控、机器发现等	简单的监控查看	不提供控制台，可对接 其它监控系统

12.10. 规则持久化

12.10.1. 是什么

一旦我们重启应用，Sentinel 规则将消失，生产环境需要将配置规则进行持久化

12.10.2. 怎么玩

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

12.10.3. 步骤

1. 修改：cloudalibaba-sentinel-service8401

2. POM

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

3. YML

```
server:
  port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
```

```
nacos:  
discovery:  
    server-addr: localhost:8848 #Nacos 服务注册中心地址  
sentinel:  
    transport:  
        dashboard: localhost:8080 #配置 Sentinel dashboard 地址  
        port: 8719  
datasource:  
    ds1:  
        nacos:  
            server-addr: localhost:8848  
            dataId: cloudalibaba-sentinel-service  
            groupId: DEFAULT_GROUP  
            data-type: json  
            rule-type: flow  
  
management:  
endpoints:  
    web:  
        exposure:  
            include: '*'
```



4. 添加 Nacos 业务规则配置

新建配置

* 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: [?](#) :

```
1 [ [ { "resource": "/testA", "limitApp": "default", "grade": 1, "count": 1, "strategy": 0, "controlBehavior": 0, "clusterMode": false } ] ]
```

• 内容解析

```
[ [ { "resource": "/testA", "limitApp": "default", "grade": 1, "count": 1, "strategy": 0, "controlBehavior": 0, "clusterMode": false } ] ]
```

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，QPS 模式（1）或并发线程数模式（0）	QPS 模式
limitApp	流控针对的调用来源	default，代表不区分调用来源
strategy	调用关系限流策略：直接、链路、关联	根据资源本身（直接）
controlBehavior	流控效果（直接拒绝 / 排队等待 / 慢启动模式），不支持按调用关系限流	直接拒绝
clusterMode	是否集群限流	否

5. 启动 8401 后刷新 sentinel 发现业务规则有了

The screenshot shows the Sentinel Control Console interface. The title bar says "Sentinel 控制台 1.7.0". Below it, there's a search bar with "应用名" and "搜索" buttons. The main area has a blue header "cloudalibaba-sentinel-service". On the left, there's a sidebar with "实时监控", "簇点链路", and a red-bordered "流控规则" button. The main content area shows a table titled "流控规则" with one record: "资源名: /rateLimit/byUrl, 来源应用: default, 流控模式: 直接, 阈值类型: QPS, 阈值: 1, 阈值模式: 单机, 流控效果: 快速失败". A red arrow points from the "流控规则" button in the sidebar to the table in the main area.

6. 快速访问测试接口

<http://localhost:8401/testA>

默认

← → C ⌂ ⓘ localhost:8401/testA

Blocked by Sentinel (flow limiting)

7. 停止 8401 再看 sentinel

cloudalibaba-sentinel-service

+ 新增流控规则

资源名	来源应用	流控模式	阈值类型	阈值	阈值模式	流控效果	操作
停机后发现流控规则没有了							

共 0 条记录, 每页 10 条记录

8. 重新启动 8401 再看 sentinel

扎一看还是没有，稍等一会儿

多次调用

<http://localhost:8401/testA>

重新配置出现了，持久化验证通过