1.微服务架构

什么是微服务?

维基上对其定义为：一种软件开发技术- 面向服务的体系结构（SOA）架构样式的一种变体，将应用程序构造为一组松散耦合的服务。在微服务体系结构中，服务是细粒度的，协议是轻量级的。

微服务（或微服务架构）是一种云原生架构方法，其中单个应用程序由许多松散耦合且可独立部署的较小组件或服务组成。这些服务通常

- 有自己的**堆栈**，包括数据库和数据模型；
- 通过REST API，事件流和消息代理的组合相互通信；
- 和它们是按业务能力组织的，分隔服务的线通常称为有界上下文。

尽管有关微服务的许多讨论都围绕体系结构定义和特征展开，但它们的价值可以通过相当简单的业务和组织收益更普遍地理解：

- 可以更轻松地更新代码。
- 团队可以为不同的组件使用不同的堆栈。
- 组件可以彼此独立地进行缩放，从而减少了因必须缩放整个应用程序而产生的浪费和成本，因为单个功能可能面临过多的负载。

1.1. Java语言相关微服务框架

Dubbo

Dubbo是由阿里巴巴开源的分布式服务化治理框架，通过RPC请求方式访问。Dubbo是在阿里巴巴的电商平台中逐渐探索演进所形成的，经历过复杂业务的高并发挑战，比Spring Cloud的开源时间还要早。目前阿里、京东、当当、携程、去哪等一些企业都在使用Dubbo。

Dropwizard

Dropwizard将Java生态系统中各个问题域里最好的组建集成于一身，能够快速打造一个Rest风格的后台，还可以整合Dropwizard核心以外的项目。国内现在使用Dropwizard还很少，资源也不多，但是与Spring Boot相比，Dropwizard在轻量化上更有优势，同时如果用过Spring，那么基本也会使用Spring Boot。

Akka

Akka是一个用Scala编写的库，可以用在有简化编写容错、高可伸缩性的Java和Scala的Actor模型，使用Akka能够实现微服务集群。

Vert.x/Lagom/ReactiveX/Spring 5

这四种框架主要用于响应式微服务开发，响应式本身和微服务没有关系，更多用于提升性能上，但是可以和微服务相结合，也可以提升性能。

Spring Boot 与 Spring Cloud

Spring Boot的设计目的是简化新Spring应用初始搭建以及开发过程，2017年有64.4%的受访者决定使用Spring Boot，可以说是最受欢迎的微服务开发框架。利用Spring Boot开发的便捷度简化分布式系统基础设施的开发，比如像配置中心、注册、负载均衡等方面都可以做到一键启动和一键部署。

Spring Cloud是一个系列框架的合计，基于HTTP（s）的RETS服务构建服务体系，Spring Cloud能够帮助架构师构建一整套完整的微服务架构技术生态链。

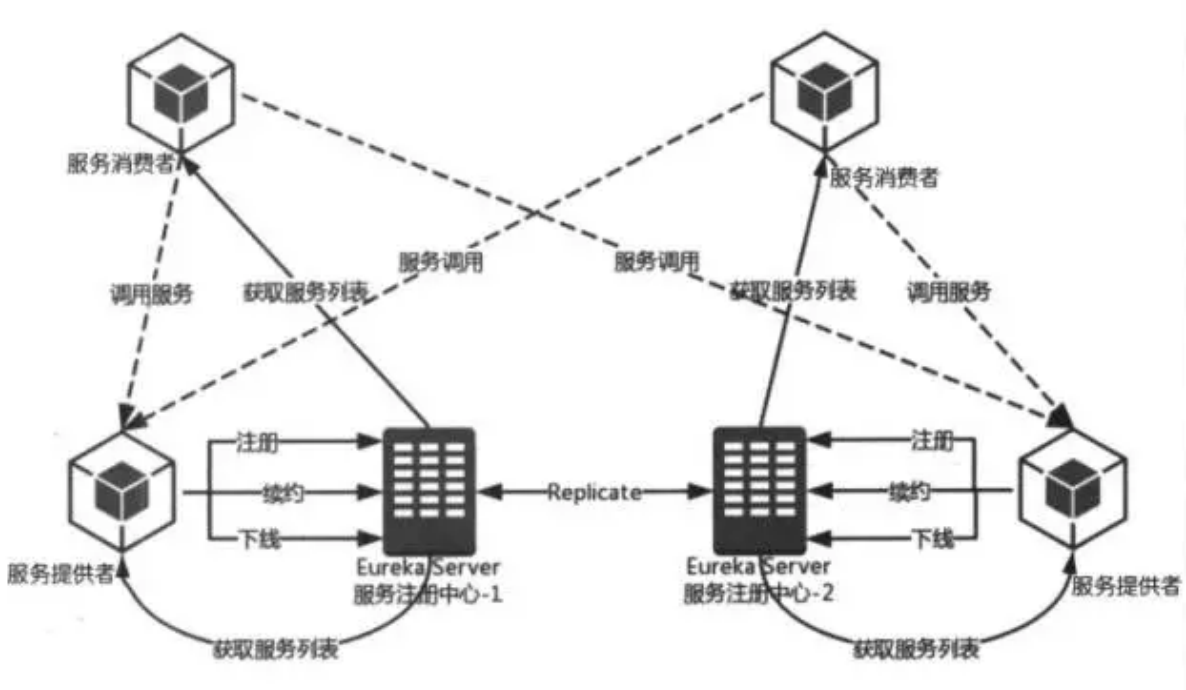
2. Springcloud 介绍

2.1 介绍

在介绍Spring Cloud 全家桶之前，首先要介绍一下Netflix ， Netflix 是一个很伟大的公司，在Spring Cloud 项目中占着重要的作用，Netflix 公司提供了包括Eureka、Hystrix、Zuul、Archaius等在内的很多组件，在微服务架构中至关重要，Spring在Netflix 的基础上，封装了一系列的组件，命名为：Spring Cloud Eureka、Spring Cloud Hystrix、Spring Cloud Zuul等，下边对各个组件进行分别得介绍： （1）Spring Cloud Eureka

我们使用微服务，微服务的本质还是各种API接口的调用，那么我们怎么产生这些接口、产生了这些接口之后如何进行调用那？如何进行管理哪？

答案就是Spring Cloud Eureka，我们可以将自己定义的API 接口注册到Spring Cloud Eureka上，Eureka负责服务的注册于发现，如果学习过Zookeeper的话，就可以很好的理解，Eureka的角色和 Zookeeper的角色差不多，都是服务的注册和发现，构成Eureka体系的包括：服务注册中心、服务提供者、服务消费者。



这里写图片描述

上图中描述了（图片来源于网络）：

1、两台Eureka服务注册中心构成的服务注册中心的主从复制集群； 2、然后服务提供者向注册中心进行注册、续约、下线服务等； 3、服务消费者向Eureka注册中心拉去服务列表并维护在本地（这也是客户端发现模式的机制体现！）； 4、然后服务消费者根据从Eureka服务注册中心获取的服务列表选取一个服务提供者进行消费服务。

（2）Spring Cloud Ribbon

在上Spring Cloud Eureka描述了服务如何进行注册，注册到哪里，服务消费者如何获取服务生产者的服务信息，但是Eureka只是维护了服务生产者、注册中心、服务消费者三者之间的关系，真正的服务消费者调用服务生产者提供的数据是通过Spring Cloud Ribbon来实现的。

在（1）中提到了服务消费者是将服务从注册中心获取服务生产者的服务列表并维护在本地的，这种客户端发现模式的方式是服务消费者选择合适的节点进行访问服务生产者提供的数据，这种选择合适节点的过程就是Spring Cloud Ribbon完成的。

Spring Cloud Ribbon客户端负载均衡器由此而来。

（3）Spring Cloud Feign

上述（1）、（2）中我们已经使用最简单的方式实现了服务的注册发现和服务的调用操作，如果具体的使用Ribbon调用服务的话，你就可以感受到使用Ribbon的方式还是有一些复杂，因此Spring Cloud Feign应运而生。

Spring Cloud Feign 是一个声明web服务客户端，这使得编写Web服务客户端更容易，使用Feign 创建一个接口并对它进行注解，它具有可插拔的注解支持包括Feign注解与JAX-RS注解，Feign还支持可插拔的编码器与解码器，Spring Cloud 增加了对 Spring MVC的注解，Spring Web 默认使用了HttpMessageConverters，Spring Cloud 集成 Ribbon 和 Eureka 提供的负载均衡的HTTP客户端 Feign。

简单的可以理解为：Spring Cloud Feign 的出现使得Eureka和Ribbon的使用更为简单。

（4）Spring Cloud Hystrix

我们在（1）、（2）、（3）中知道了使用Eureka进行服务的注册和发现，使用Ribbon实现服务的负载均衡调用，还知道了使用Feign可以简化我们的编码。但是，这些还不足以实现一个高可用的微服务架构。

例如：当有一个服务出现了故障，而服务的调用方不知道服务出现故障，若此时调用放的请求不断的增加，最后就会等待出现故障的依赖方 相应形成任务的积压，最终导致自身服务的瘫痪。

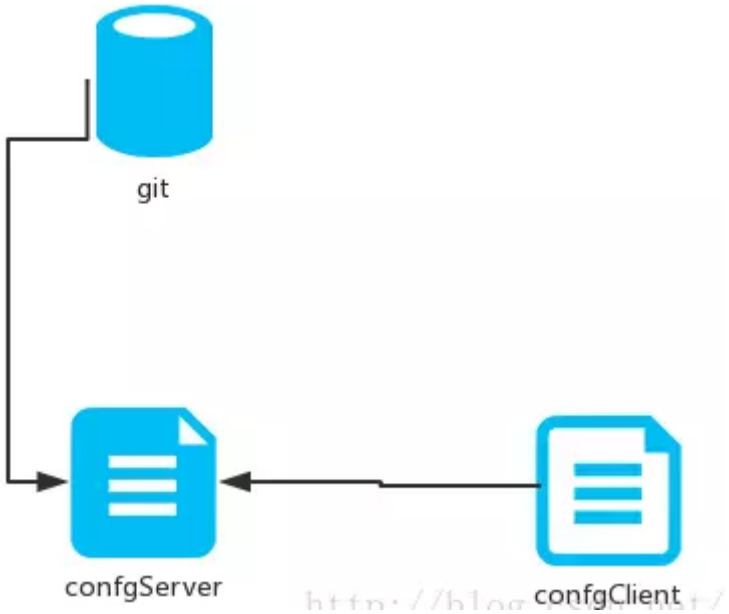
Spring Cloud Hystrix正是为了解决这种情况的，防止对某一故障服务持续进行访问。Hystrix的含义是：断路器，断路器本身是一种开关装置，用于我们家庭的电路保护，防止电流的过载，当线路中有电器发生短路的时候，断路器能够及时切换故障的电器，防止发生过载、发热甚至起火等严重后果。

（5）Spring Cloud Config

对于微服务还不是很多的时候，各种服务的配置管理起来还相对简单，但是当成百上千的微服务节点起来的时候，服务配置的管理变得会复杂起来。

分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件Spring Cloud Config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在Cpring Cloud Config 组件中，分两个角色，一是Config Server，二是Config Client。

Config Server用于配置属性的存储，存储的位置可以为Git仓库、SVN仓库、本地文件等，Config Client用于服务属性的读取。



（6）Spring Cloud Zuul

我们使用Spring Cloud Netflix中的Eureka实现了服务注册中心以及服务注册与发现；而服务间通过Ribbon或Feign实现服务的消费以及均衡负载；通过Spring Cloud Config实现了应用多环境的外部化配置以及版本管理。为了使得服务集群更为健壮，使用Hystrix的融断机制来避免在微服务架构中个别服务出现异常时引起的故障蔓延。

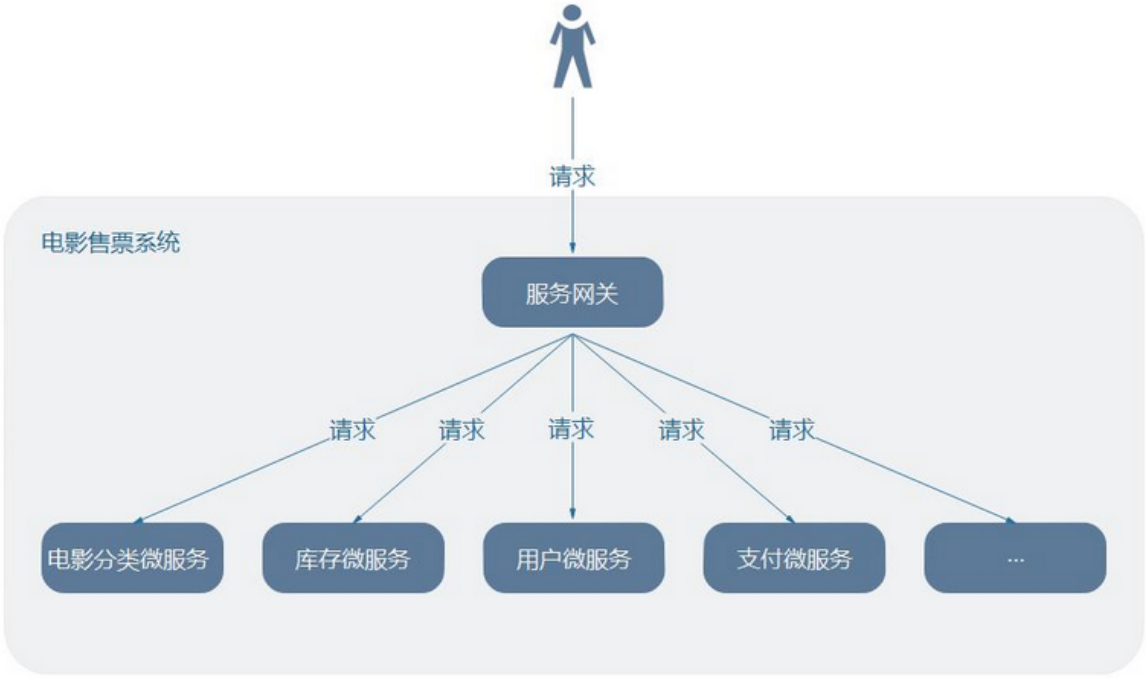
先来说说这样架构需要做的一些事儿以及存在的不足：

1、首先，破坏了服务无状态特点。为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。

2、其次，无法直接复用既有接口。当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。 面对类似上面的问题，我们要如何解决呢？下面进入本文的正题：服务网关！

为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器，它就是本文将来介绍的：服务网关。

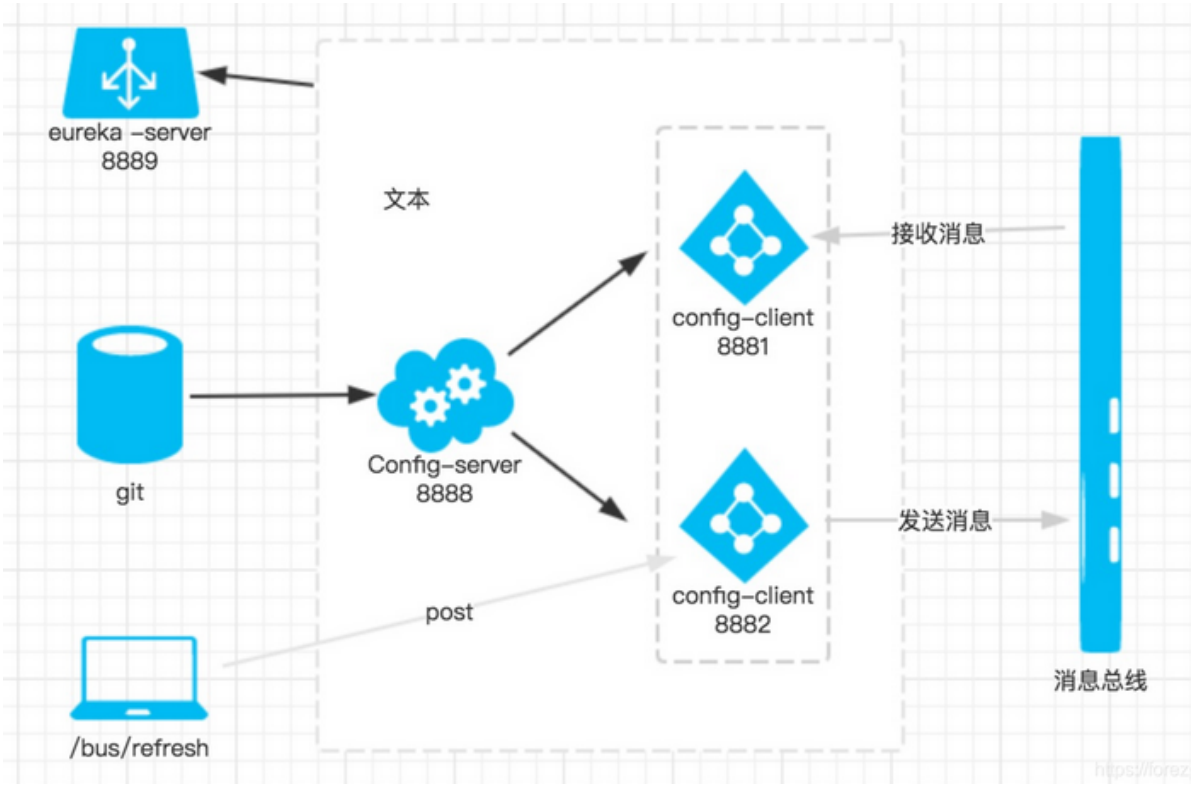
服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。



（7）Spring Cloud Bus

在（5）Spring Cloud Config中，我们知道的配置文件可以通过Config Server存储到Git等地方，通过Config Client进行读取，但是我们的配置文件不可能是一直不变的，当我们的配置文件放生变化的时候如何进行更新哪?

一种最简单的方式重新一下Config Client进行重新获取，但Spring Cloud绝对不会让你这样做的，Spring Cloud Bus正是提供一种操作使得我们在不关闭服务的情况下更新我们的配置。

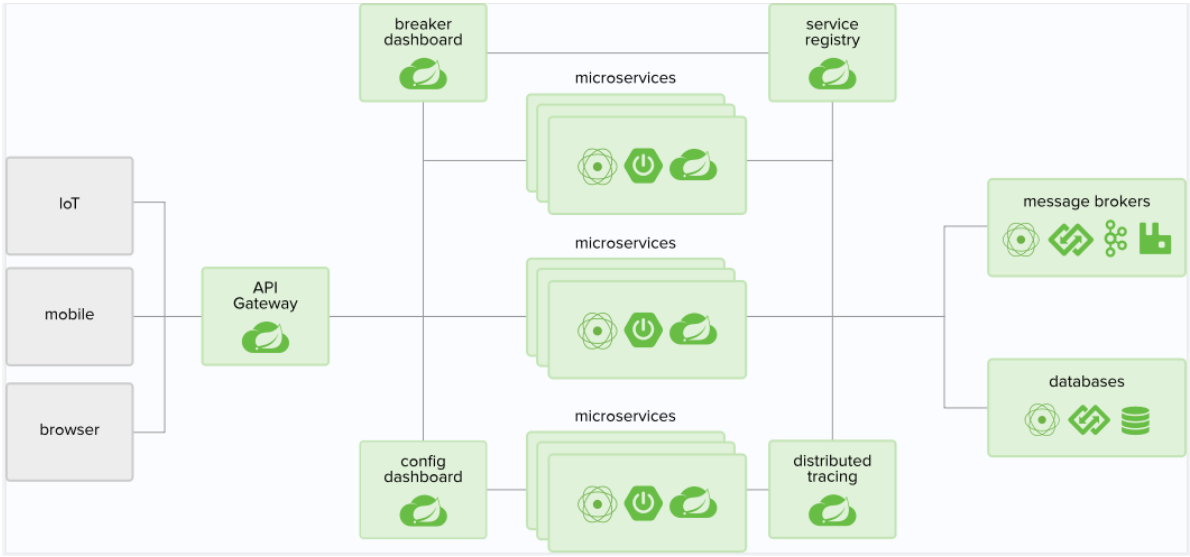


Spring Cloud Bus官方意义：消息总线。

当然动态更新服务配置只是消息总线的一个用处，还有很多其他用处。

https://docs.spring.io/spring-cloud/docs/Hoxton.SR9/reference/htmlsingle/

SpringCloud架构



2.2 关于Cloud各种组件的停更、升级、替换

Netflix 开源发生了什么?

2018年6月底，Eureka 2.0 开源工作宣告停止，继续使用风险自负。

```
Eureka 2.0 (Discontinued)

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

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

2018年11月底，Hystrix 宣布不再在开源版本上提供新功能。

2018年12月，Spring官方宣布Netflix的相关项目进入维护模式（Maintenance Mode）。

Spring官方对什么是Maintenance Mode给出了定义：

一旦进入维护模式，Spring Cloud将不会对该组件添加新功能，但我们会继续修复相关的bug和一些安全性问题，同时考虑和review一些小的PR。我们将在Greenwich Release后的至少一年内提供以上支持。

SpringCloud新闻

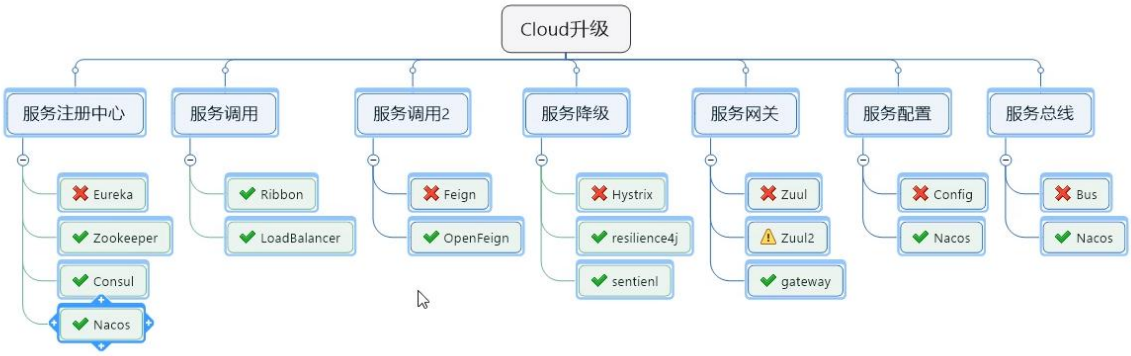
https://spring.io/blog/2018/12/12/spring-cloud-greenwich-rc1-available-now#spring-cloud-netflix-projects-entering-maintenance-mode

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

- spring-cloud-netflix-archaius
- spring-cloud-netflix-hystrix-contract
- spring-cloud-netflix-hystrix-dashboard
- spring-cloud-netflix-hystrix-stream
- spring-cloud-netflix-hystrix
- spring-cloud-netflix-ribbon
- spring-cloud-netflix-turbine-stream
- spring-cloud-netflix-turbine

- spring-cloud-netflix-zuul

停更的解决方案：



3. 技术选型

3.1. Springboot版本选择

> git源码地址

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

> SpringBoot2.0新特性 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

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

> Springcloud和Springboot之间的依赖关系如何看

https://spring.io/projects/spring-cloud#overview

Table 1. Release train Spring Boot compatibility

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

> 依赖 更详细的版本对应查看方法

https://start.spring.io/actuator/info

```
1  {
```

```
    "git": {
        "branch": "f8d94d9224f93bc01a553d1bf18f223aba939ec4",
        "commit": {
            "id": "f8d94d9",
            "time": "2021-06-18T15:04:41Z"
        }
    },
    "build": {
        "version": "0.0.1-SNAPSHOT",
        "artifact": "start-site",
        "versions": {
            "spring-boot": "2.5.1",
            "initializr": "0.11.0-SNAPSHOT"
        },
        "name": "start.spring.io website",
        "time": "2021-06-18T15:05:31.390Z",
        "group": "io.spring.start"
    },
    "bom-ranges": {
        "azure": {
            "2.2.4": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "3.2.0": "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
            "3.5.0": "Spring Boot >=2.4.0.M1 and <2.5.0-M1"
        },
        "codecentric-spring-boot-admin": {
            "2.2.4": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "2.3.1": "Spring Boot >=2.3.0.M1 and <2.5.0-M1"
        },
        "solace-spring-boot": {
            "1.0.0": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "1.1.0": "Spring Boot >=2.3.0.M1 and <2.6.0-M1"
        },
        "solace-spring-cloud": {
            "1.0.0": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "1.1.1": "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
            "2.1.0": "Spring Boot >=2.4.0.M1 and <2.6.0-M1"
        },
        "spring-cloud": {
            "Hoxton.SR11": "Spring Boot >=2.2.0.RELEASE and <2.3.999.BUILD-
SNAPSHOT",
            "Hoxton.BUILD-SNAPSHOT": "Spring Boot >=2.3.999.BUILD-SNAPSHOT and
<2.4.0.M1",
            "2020.0.0-M3": "Spring Boot >=2.4.0.M1 and <=2.4.0.M1",
            "2020.0.0-M4": "Spring Boot >=2.4.0.M2 and <=2.4.0-M3",
            "2020.0.0": "Spring Boot >=2.4.0.M4 and <=2.4.0",
            "2020.0.3": "Spring Boot >=2.4.1 and <2.5.2-SNAPSHOT",
            "2020.0.4-SNAPSHOT": "Spring Boot >=2.5.2-SNAPSHOT"
        },
        "spring-cloud-alibaba": {
            "2.2.1.RELEASE": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1"
        },
        "spring-cloud-gcp": {
            "2.0.3": "Spring Boot >=2.4.0-M1 and <2.5.0-M1"
        },
        "spring-cloud-services": {
            "2.2.6.RELEASE": "Spring Boot >=2.2.0.RELEASE and <2.3.0.RELEASE",
            "2.3.0.RELEASE": "Spring Boot >=2.3.0.RELEASE and <2.4.0-M1",
            "2.4.1": "Spring Boot >=2.4.0-M1 and <2.5.0-M1"
```

```
        },
        "spring-geode": {
            "1.2.12.RELEASE": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "1.3.12.RELEASE": "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
            "1.4.7": "Spring Boot >=2.4.0-M1 and <2.5.0-M1",
            "1.5.1": "Spring Boot >=2.5.0-M1"
        },
        "vaadin": {
            "14.6.3": "Spring Boot >=2.1.0.RELEASE and <2.6.0-M1"
        },
        "wavefront": {
            "2.0.2": "Spring Boot >=2.1.0.RELEASE and <2.4.0-M1",
            "2.1.1": "Spring Boot >=2.4.0-M1 and <2.5.0-M1",
            "2.2.0": "Spring Boot >=2.5.0-M1"
        }
    },
    "dependency-ranges": {
        "native": {
            "0.9.0": "Spring Boot >=2.4.3 and <2.4.4",
            "0.9.1": "Spring Boot >=2.4.4 and <2.4.5",
            "0.9.2": "Spring Boot >=2.4.5 and <2.5.0-M1",
            "0.10.0": "Spring Boot >=2.5.0-M1 and <2.5.2-M1",
            "0.10.1-SNAPSHOT": "Spring Boot >=2.5.2-M1 and <2.6.0-M1"
        },
        "okta": {
            "1.4.0": "Spring Boot >=2.2.0.RELEASE and <2.4.0-M1",
            "1.5.1": "Spring Boot >=2.4.0-M1 and <2.4.1",
            "2.0.1": "Spring Boot >=2.4.1 and <2.5.0-M1",
            "2.1.0": "Spring Boot >=2.5.0-M1 and <2.6.0-M1"
        },
        "mybatis": {
            "2.1.4": "Spring Boot >=2.1.0.RELEASE and <2.5.0-M1",
            "2.2.0": "Spring Boot >=2.5.0-M1"
        },
        "camel": {
            "3.3.0": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "3.5.0": "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
            "3.10.0": "Spring Boot >=2.4.0.M1 and <2.5.0-M1"
        },
        "open-service-broker": {
            "3.1.1.RELEASE": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1",
            "3.2.0": "Spring Boot >=2.3.0.M1 and <2.4.0-M1",
            "3.3.0": "Spring Boot >=2.4.0-M1 and <2.5.0-M1"
        }
    }
}
```

本次使用的cloud版本为Hoxton.SR9，SpringBoot版本为 **2.3.2.RELEASE**

库	版本	说明
SpringCloud	Hoxton.SR8	
SpringBoot	**2.3.2.RELEASE**	
spring-cloud-alibaba	2.2.5.RELEASE	
Java	8	
Maven	3.6+	

4. 微服务架构编码构建

> 项目仓库地址： **https://gitee.com/harbin-university-18-java1/spring-cloud-learning**

4.1 构建父工程

微服务cloud整体聚合父工程Project

- New Project
- 聚合总父工程名字
- Maven选版本
- 工程名字
- 字符编码
- 注解生效激活
- java编译版本选8
- File Type过滤

pom.xml

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

    <groupId>org.jshand.cloud</groupId>
    <artifactId>springcloud-java1-2018</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>

        <java.version>1.8</java.version>
        <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>
        <mysql.version>5.1.49</mysql.version>
        <mybatis.version>2.1.3</mybatis.version>
        <lombok.version>1.18.16</lombok.version>
        <druid.version>1.1.16</druid.version>
        <mybatis.spring.boot.version>1.3.0</mybatis.spring.boot.version>
    </properties>


    <!--控制子工程 版本号。 不实际依赖  子模块继承之后，提供作用：锁定版本+子modlue不用写groupId和version -->
    <dependencyManagement>
        <dependencies>
            <!--spring boot 2.3.2-->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.3.2.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--spring cloud Hoxton.SR8-->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Hoxton.SR8</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--spring cloud alibaba 2.2.5.RELEASE
            https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E
            -->
            <dependency>
                <groupId>com.alibaba.cloud</groupId>
                <artifactId>spring-cloud-alibaba-dependencies</artifactId>
                <version>2.2.5.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <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>
        <resources>
            <resource>
                <directory>src/main/java</directory>
                <includes>
                    <include>**/*.xml</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/resources</directory>
                <includes>
                    <include>**/*.*</include>
                </includes>
            </resource>
        </resources>

        <plugins>
            <!--启动热部署-->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <fork>true</fork>
                    <addResources>true</addResources>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

关于dependencyManagement

Maven 使用dependencyManagement 元素来提供了一种管理依赖版本号的方式。 通常会在一个组织或者项目的最顶层的父POM 中看到dependencyManagement 元素。

使用pom.xml 中的dependencyManagement 元素能让所有在子项目中引用一个依赖而不用显式的列出版本号。 Maven 会沿着父子层次向上走，直到找到一个拥有dependencyManagement 元素的项目，然后它就会使用这个 dependencyManagement 元素中指定的版本号。

这样做的好处就是：如果有多个子项目都引用同一样依赖，则可以避免在每个使用的子项目里都声明一个版本号，这样当想升级或切换到另一个版本时，只需要在顶层父容器里更新，而不需要一个一个子项目的修改；另外如果某个子项目需要另外的一个版本，只需要声明version就可。

- dependencyManagement里只是声明依赖，并不实现引入，因此子项目需要显示的声明需要用的依赖。
- 如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且version和scope都读取自父pom;
- 如果子项目中指定了版本号，那么会使用子项目中指定的jar版本。

4.2 Rest微服务工程构建

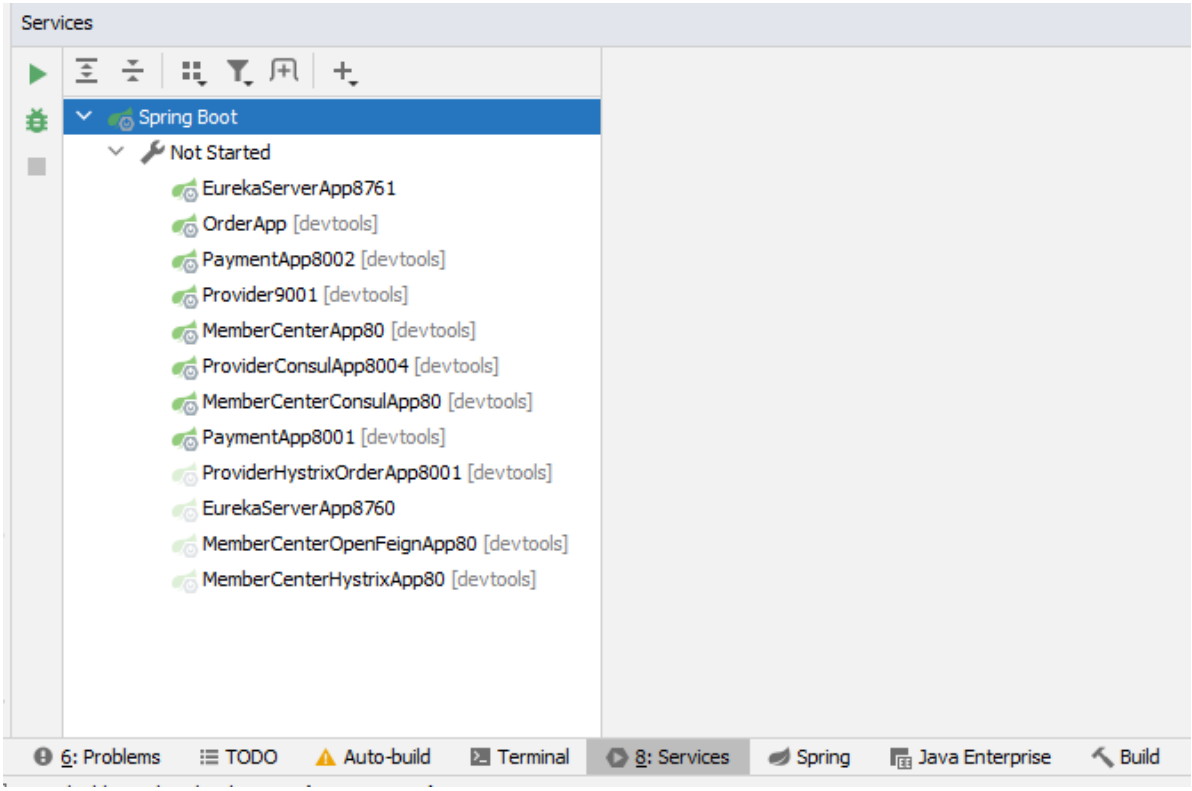
搭建一些消费提供者

- 改POM
- 写YML
- 业务类
- 主启动
-
  - controller
- 测试

```
postman模拟post
```

- 运行 通过修改idea的workspace.xml的方式来快速打开Run Dashboard窗口，开启Services

```
<component name="RunDashboard">
  <option name="configurationTypes">
    <set>
      <option value="SpringBootApplicationConfigurationType" />
    </set>
  </option>
</component>
```



5. 远程调用（客户端）

常用的几种RPC调用方式包括http client、RestTemplate、openFigen

此处演示HttpComponents和普通的RestTemplate，下一章演示RestTemplate集合Eureka做负载均衡、以及openFigen方式远程调用

- 创建modules（cloud-rpc-clients）
- 修改pom

```
<dependencies>
        <!-- 提供web 内嵌的Tomcat-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!--测试框架 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

        <!--热部署的工具 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>

        <!--
https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->
        <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
        </dependency>
```

```
25
26        </dependencies>
```

- 测试httpclient访问 payment接口
  - httpclient快速开始请参考： http://hc.apache.org/httpcomponents-client-4.5.x/quickstart.html
- RestTemplate访问接口
  - Spring提供的接口
- 修改yml
- 业务类
- 启动类
- 测试
- 封装统一的返回结果

6. Eureka的服务注册与发现

6.1 Eureka基础介绍

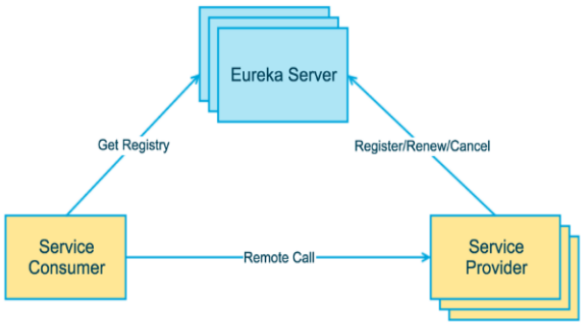
1. 什么是服务治理

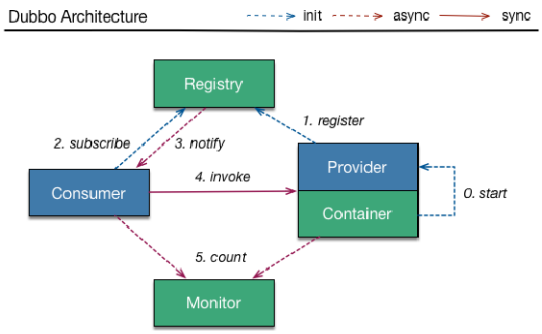
Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务治理

在传统的rpc远程调用框架中，管理每个服务与服务之间依赖关系比较复杂，管理比较复杂，所以需要使用服务治理，管理服务于服务之间依赖关系，可以实现服务调用、负载均衡、容错等，实现服务发现与注册。

2.什么是服务注册与发现

Eureka采用了CS的设计架构，Eureka Server 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka的客户端连接到 Eureka Server并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。 在服务注册与发现中，有一个注册中心。当服务器启动的时候，会把当前自己服务器的信息 比如 服务地址通讯地址等以别名方式注册到注册中心上。另一方（消费者|服务提供者），以该别名的方式去注册中心上获取到实际的服务通讯地址，然后再实现本地RPC调用RPC远程调用框架核心设计思想：在于注册中心，因为使用注册中心管理每个服务与服务之间的一个依赖关系(服务治理概念)。在任何rpc远程框架中，都会有一个注册中心(存放服务地址相关信息(接口地址)) 下左图是Eureka系统架构，右图是Dubbo的架构，请对比





3.Eureka包含两个组件

> Eureka Server和Eureka Client

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

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

6.2 Eureka常用属性配置

Spring Cloud Eureka 属性作用

配置参数	默认值	说明
服务注册中心配置		Bean类：org.springframework.cloud.netflix.eureka.server.EurekaServerConfigBean
eureka.server.enable-self-preservation	false	关闭注册中心的保护机制，Eureka 会统计15分钟之内心跳失败的比例低于85%将会触发保护机制，不剔除服务提供者，如果关闭服务注册中心将不可用的实例正确剔除
服务实例类配置		Bean类：org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean
eureka.instance.prefer-ip-address	false	不使用主机名来定义注册中心的地址，而使用IP地址的形式，如果设置了 eureka.instance.ip-address 属性，则使用该属性配置的IP，否则自动获取除环路IP外的第一个IP地址
eureka.instance.ip-address		IP地址
eureka.instance.hostname		设置当前实例的主机名称
eureka.instance.appname		服务名，默认取 spring.application.name 配置值，如果没有则为 unknown
eureka.instance.lease-renewal-interval-in-seconds	30	定义服务续约任务（心跳）的调用间隔，单位：秒
eureka.instance.lease-expiration-duration-in-seconds	90	定义服务失效的时间，单位：秒
eureka.instance.status-page-url-path	/info	状态页面的URL，相对路径，默认使用 HTTP 访问，如果需要使用 HTTPS则需要使用绝对路径配置
eureka.instance.status-page-url		状态页面的URL，绝对路径
eureka.instance.health-check-url-path	/health	健康检查页面的URL，相对路径，默认使用 HTTP 访问，如果需要使用 HTTPS则需要使用绝对路径配置
eureka.instance.health-check-url		健康检查页面的URL，绝对路径
服务注册类配置		Bean类：org.springframework.cloud.netflix.eureka.EurekaClientConfigBean
eureka.client.service-url.		指定服务注册中心地址，类型为 HashMap，并设置有一组默认值，默认的Key为 defaultZone；默认的Value为 http://localhost:8761/eureka ，如果服务注册中心为高可用集群时，多个注册中心地址以逗号分隔。 如果服务注册中心加入了安全验证，这里配置的地址格式为： http://%3Cusername%3E:%3Cpassword%3E@localhost:8761/eureka 其中 为安全校验的用户名； 为该用户的密码
eureka.client.fetch-registery	true	检索服务；false不会向Eureka Server注册中心获取注册信息
eureka.client.registery-fetch-interval-seconds	30	从Eureka服务器端获取注册信息的间隔时间，单位：秒
eureka.client.register-with-eureka	true	启动服务注册；false不会向Eureka Server注册中心注册自己的信息
eureka.client.eureka-server-connect-timeout-seconds	5	连接 Eureka Server 的超时时间，单位：秒

配置参数	默认值	说明
eureka.client.eureka-server-read-timeout-seconds	8	读取 Eureka Server 信息的超时时间，单位：秒
eureka.client.filter-only-up-instances	true	获取实例时是否过滤，只保留UP状态的实例
eureka.client.eureka-connection-idle-timeout-seconds	30	Eureka 服务端连接空闲关闭时间，单位：秒
eureka.client.eureka-server-total-connections	200	从Eureka 客户端到所有Eureka服务端的连接总数
eureka.client.eureka-server-total-connections-per-host	50	从Eureka客户端到每个Eureka服务主机的连接总数

6.3 单机搭建Eureka

1.搭建Eureka

- 建Module
- 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>spring-cloud-java1</artifactId>
        <groupId>com.neuedu.cloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>server-eureka-7001</artifactId>


    <dependencies>
        <!--eureka-server-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>
        <!--boot 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>
        <!--一般通用配置-->
        <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>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>


</project>
```

关于Eureka的依赖

```
<!--以前的老版本（当前使用2018） -->
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

<!--现在新版本 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- 写YML

```
server:
  port: 7001

eureka:
  instance:
    hostname: localhost #eureka服务端的实例名称
  client:
    #false表示不向注册中心注册自己。
    register-with-eureka: false
    #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
    fetch-registry: false
    service-url:
    #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。
    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

- 主启动
    - @EnableEurekaServer

```
package org.jshand;
```

```
2
3  import org.slf4j.Logger;
4  import org.slf4j.LoggerFactory;
5  import org.springframework.boot.SpringApplication;
6  import org.springframework.boot.autoconfigure.SpringBootApplication;
7  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
8
9
10 @SpringBootApplication
11 @EnableEurekaServer
12 public class EurekaServerApp7001 {
13     private static final Logger log =
   LoggerFactory.getLogger(EurekaServerApp7001.class);
14
15     public static void main(String[] args) {
16         SpringApplication.run(EurekaServerApp7001.class, args);
17         log.info("Eureka Server Started （EurekaServerApp7001服务端启动成功）");
18     }
19 }
```

- 测试

```
1   http://localhost:7001/
2     结果页面
```

如果Idea中没有出现Service视图，找到项目目录.idea>workspace.xml

```
1  <component name="RunDashboard">
2      <option name="configurationTypes">
3        <set>
4          <option value="SpringBootApplicationConfigurationType" />
5        </set>
6      </option>
7    </component>
```

Spring Cloud Eureka 属性作用

配置参数	默认值	说明
服务注册中心配置		Bean类：org.springframework.cloud.netflix.eureka.server.EurekaServerConfigBean
eureka.server.enable-self-preservation	false	关闭注册中心的保护机制，Eureka 会统计15分钟之内心跳失败的比例低于85%将会触发保护机制，不剔除服务提供者，如果关闭服务注册中心将不可用的实例正确剔除
服务实例类配置		Bean类：org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean
eureka.instance.prefer-ip-address	false	不使用主机名来定义注册中心的地址，而使用IP地址的形式，如果设置了 eureka.instance.ip-address 属性，则使用该属性配置的IP，否则自动获取除环路IP外的第一个IP地址
eureka.instance.ip-address		IP地址
eureka.instance.hostname		设置当前实例的主机名称
eureka.instance.appname		服务名，默认取 spring.application.name 配置值，如果没有则为 unknown
eureka.instance.lease-renewal-interval-in-seconds	30	定义服务续约任务（心跳）的调用间隔，单位：秒
eureka.instance.lease-expiration-duration-in-seconds	90	定义服务失效的时间，单位：秒
eureka.instance.status-page-url-path	/info	状态页面的URL，相对路径，默认使用 HTTP 访问，如果需要使用 HTTPS则需要使用绝对路径配置
eureka.instance.status-page-url		状态页面的URL，绝对路径
eureka.instance.health-check-url-path	/health	健康检查页面的URL，相对路径，默认使用 HTTP 访问，如果需要使用 HTTPS则需要使用绝对路径配置
eureka.instance.health-check-url		健康检查页面的URL，绝对路径
服务注册类配置		Bean类：org.springframework.cloud.netflix.eureka.EurekaClientConfigBean
eureka.client.service-url.		指定服务注册中心地址，类型为 HashMap，并设置有一组默认值，默认的Key为 defaultZone；默认的Value为 http://localhost:8761/eureka ，如果服务注册中心为高可用集群时，多个注册中心地址以逗号分隔。 如果服务注册中心加入了安全验证，这里配置的地址格式为： http://%3Cusername%3E:%3Cpassword%3E@localhost:8761/eureka 其中 为安全校验的用户名； 为该用户的密码
eureka.client.fetch-registery	true	检索服务；false不会向Eureka Server注册中心获取注册信息
eureka.client.registery-fetch-interval-seconds	30	从Eureka服务器端获取注册信息的间隔时间，单位：秒
eureka.client.register-with-eureka	true	启动服务注册；false不会向Eureka Server注册中心注册自己的信息
eureka.client.eureka-server-connect-timeout-seconds	5	连接 Eureka Server 的超时时间，单位：秒

配置参数	默认值	说明
eureka.client.eureka-server-read-timeout-seconds	8	读取 Eureka Server 信息的超时时间，单位：秒
eureka.client.filter-only-up-instances	true	获取实例时是否过滤，只保留UP状态的实例
eureka.client.eureka-connection-idle-timeout-seconds	30	Eureka 服务端连接空闲关闭时间，单位：秒
eureka.client.eureka-server-total-connections	200	从Eureka 客户端到所有Eureka服务端的连接总数
eureka.client.eureka-server-total-connections-per-host	50	从Eureka客户端到每个Eureka服务主机的连接总数

2. 将提供者注册到Eureka

- 改POM

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

- 写YML

```
server:
  port: 8081

spring:
  application:
    name: payment

eureka:
  client:
      #表示是否将自己注册进EurekaServer默认为true。
    register-with-eureka: true
      #是否从EurekaServer抓取已有的注册信息，默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
    fetchRegistry: true
    service-url:
        defaultZone: http://localhost:7001/eureka
  instance:
    #在注册中心，控制台，访问的时候 地址IP地址（超链接的地址）
    prefer-ip-address: true
    #主机名字,在列表中使用
    hostname: 192.168.77.11
    #在eureka列表中查看微服务时使用如下格式
    instance-id: ${eureka.instance.hostname}:${spring.application.name}:${server.port}
```

- 主启动

```
@EnableEurekaClient
```

- 测试

```
先要启动EurekaServer
  http://localhost:7001/
  微服务注册名配置说明
```

- 自我保护机制

Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 会将这些实例保护起来，让这些实例不会过期，但是在保护期内如果服务刚好这个服务提供者非正常下线了，此时服务消费者就会拿到一个无效的服务实例，此时会调用失败，对于这个问题需要服务消费者端要有一些容错机制，如重试，断路器等。

我们在单机测试的时候很容易满足心跳失败比例在 15 分钟之内低于 85%，这个时候就会触发 Eureka 的保护机制，一旦开启了保护机制，则服务注册中心维护的服务实例就不是那么准确了，此时我们可以使用 `eureka.server.enable-self-preservation=false` 来关闭保护机制，这样可以确保注册中心中不可用的实例被及时的剔除。

3. 在Eureka中注册消费者

- POM

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

- YML

```
server:
  port: 80
spring:
  application:
    name: order

eureka:
  client:
    #表示是否将自己注册进EurekaServer默认为true。
    register-with-eureka: true
    #是否从EurekaServer抓取已有的注册信息，默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
    fetchRegistry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
  instance:
    #在注册中心，控制台，访问的时候 地址IP地址（超链接的地址）
    prefer-ip-address: true
    #主机名字,在列表中使用
    hostname: 192.168.77.11
    #在eureka列表中查看微服务时使用如下格式
    instance-id: ${eureka.instance.hostname}:${spring.application.name}:${server.port}
```

- 主启动

```
@EnableEurekaClient
```

- 测试

```
先要启动EurekaServer，7001服务
  再要启动服务提供者provider，8001服务
  eureka服务器
```

4. 使用Eureka+RestTemplate做服务调用

> 声明RestTemplate对象

在注入的RestTemplate中声明@LoadBanlance

@LoadBalanced //使用@LoadBalanced注解赋予RestTemplate负载均衡的能力(默认是ribbon)

```
@Configuration
public class Appconfig {

    @Bean
    @LoadBalanced  //负载均衡，封装了 负载的策略，
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }

}
```

> 服务调用

在调用的时候将ip：port替换成 application.name

```
@GetMapping("/order/{id}")
    CommonResult<Payment> order(@PathVariable("id") String orderId){

        //调用支付的接口
        String url = OrderConstants.SERVICE_PAYMENT+"/pay/"+orderId;
        CommonResult<Payment> result = restTemplate.getForObject(url,
CommonResult.class);

        return result;
    }
```

5.模拟服务集群

上面模拟的是一个服务提供者，为了负载提高系统可用性，此时可以将服务提供者多部署几台，分别上线。

将 `cloud-provider-payment-8081` 代码复制一份，修改端口为8082，并添加相同的方法

```
    /**
         *  http://127.0.0.1:8081/provider/echo/clouds
         */
        @Value("${server.port}")
        private String port;

        @GetMapping("/echo/{param}")
        String echo(@PathVariable String param){
            return param+"-provider: "+port;
        }
```

在服务消费方添加方法调用echo方法

```
@GetMapping("/echo/{param}")
String echo(@PathVariable String param){
    String url = OrderConstants.SERVICE_PAYMENT+"/echo/"+param;
    String result = restTemplate.getForObject(url, String.class);
    return result;
}
```

通过浏览器 直接访问服务消费者， `http://192.168.77.11/consumer/echo/aaa` ,结果会在 `8081` 、`和8082之间切换`

```
aaa-provider: 8082`、`aaa-provider: 8081
```

6.4 手动的服务发现Discovery

在App类上添加 `@EnableDiscoveryClient` 注解

使用DiscoveryClient手动访问服务;

```
package com.neuedu.cloud.controller;

import com.netflix.discovery.converters.Auto;
import com.neuedu.cloud.common.OrderConstants;
import com.neueud.cloud.common.CommonResult;
import com.neueud.cloud.entity.Payment;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
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 java.util.List;

/**
 * 项目 spring-cloud-java1
 *
 * @author 张金山
 * @version 1.0
 * 说明 订单接口
```

```
 * @date 2021/6/23 13:36
 */
@RestController
@Slf4j
public class OrderControlelr {
    @Autowired
    RestTemplate restTemplate; //才有能力将应用的 名字 转换成 实际 ip地址


    /**
     * http://192.168.77.11/order/506
     * @param orderId
     * @return
     */
    @GetMapping("/order/{id}")
    CommonResult<Payment> order(@PathVariable("id") String orderId){

        //调用支付的接口
        String url = OrderConstants.SERVICE_PAYMENT+"/pay/"+orderId;
        CommonResult<Payment> result = restTemplate.getForObject(url,
CommonResult.class);

        return result;
    }

    @GetMapping("/echo/{param}")
    String echo(@PathVariable String param){
        String url = OrderConstants.SERVICE_PAYMENT+"/echo/"+param;
        String result = restTemplate.getForObject(url, String.class);
        return result;
    }


    /**
     * 手动发现服务
      */
    @Autowired
    DiscoveryClient discoveryClient;  //httpclient

    /**
     * 手动发现服务
     * @return
     */
    @GetMapping("/discoveryClient")
    String discoveryClient(){
        List<String> services = discoveryClient.getServices();
        StringBuffer stringBuffer = new StringBuffer();

        for (String service : services) {
           log.info("服务名称:{}",service);
            stringBuffer.append("服务名称:"+service);

            List<ServiceInstance> instance =
discoveryClient.getInstances(service);
            for (ServiceInstance serviceInstance : instance) {
                /***
                 * http://192.168.77.11:80/order/506
                 */
```

```
82
83                  log.info("ServiceId:{}, getHost:{},Port :{}",
84                          serviceInstance.getServiceId(),
85                          serviceInstance.getHost(),
86                          serviceInstance.getPort()
87                  );
88              }
89
90          }
91          return stringBuffer.toString();
92      }
93
94  }
```

6.5 集群搭建Eureka

为了实现Eureka的高可用，搭建Eureka注册中心集群 ， 实现负载均衡+故障容错

准备工作

找到C:\Windows\System32\drivers\etc路径下的hosts文件,或者使用swichhosts软件 修改映射配置添加进hosts文件

```
127.0.0.1 eureka7001.com
127.0.0.1 eureka7002.com
C:\Users\Administrator>ping eureka7001.com

正在 Ping eureka7001.com [127.0.0.1] 具有 32 字节的数据:
来自 127.0.0.1 的回复: 字节=32 时间<1ms TTL=64
来自 127.0.0.1 的回复: 字节=32 时间<1ms TTL=64
来自 127.0.0.1 的回复: 字节=32 时间<1ms TTL=64
来自 127.0.0.1 的回复: 字节=32 时间<1ms TTL=64

127.0.0.1 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms，最长 = 0ms，平均 = 0ms

C:\Users\Administrator>
```

- 新建server-eureka-7002
- 改POM ,跟7001相同
- 写YML(以前单机)
  - 端口
  - hostname
  - 7001 需要修改defaultZone
  - 7002 需要修改defaultZone
- 在微服务中配置注册到集群中（yml）

```
        defaultZone:
    http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka  # 集群版
```

- 测试,先启动两个Eureka服务

- 查看两个Eureka控制台，是否存在服务 `http://eureka7001.com:7001/` 、`http://eureka7002.com:7002/`
- 在启动order、payment三个微服务。
- 再测试将注册中心的某一个停止，测试远程服务调用是否可用

6.6 自我保护模式

什么会产生Eureka自我保护机制？ 为了防止EurekaClient可以正常运行，但是 与 EurekaServer网络不通情况下，EurekaServer不会立刻将EurekaClient服务剔除

什么是自我保护模式？ 默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生(延时、卡顿、拥挤)时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。

在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。 它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话讲解：好死不如赖活着

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留）也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

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

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

心跳检测和服务端自我保护

> 客户端

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

服务端

```
# 关闭自我保护机制，保证不可用的服务被及时剔除
enable-self-preservation: false
# 如果2秒内没有收到某个微服务的心跳，那就剔除该微服务，单位为毫秒
eviction-interval-timer-in-ms: 2000
```

7. Ribbon负载均衡调用

7.1 介绍

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

简单的说，Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法和服务调用。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer（简称LB）后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们很容易使用Ribbon实现自定义的负载均衡算法。

7.2 RestTemplate+Ribbon实现负载均衡

7.3 核心组件IRule

7.3.1 核心组件自带的算法

7.3.2 如何替换默认规则

由于Ribbon是客户端负载算法，所以需要修改微服务客户端（远程调用发起方）。

- 在客户端定义Ribbon的配置类，用于在上下文中注册规则类。(根据 **官网** 说明,**自定义的类最好在@ComponentScan注解扫描不到的包**，这样的方便个性化配置，否则会导致调用所有服务都是用同一个规则) 自定义配置类如下：

```
@Configuration
public class RibbonConfig {
    @Bean
    public IRule ribbonRule(){
        System.out.println("准备负载规则....");
        return new RandomRule();
    }
}
```

- 在主启动类上使用@RibbonClient注解用于定义某一个服务所使用的的配置

```
@RibbonClient(name = "payment",configuration = RibbonConfig.class)
```

其中name为调用服务的名称，configuration为上述配置类

7.3.3 RoundRobinRule轮训算法源码分析

```
/*
 *
 * Copyright 2013 Netflix, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
```

```
 */
package com.netflix.loadbalancer;

import com.netflix.client.config.IClientConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * The most well known and basic load balancing strategy, i.e. Round Robin Rule.
 *
 * @author stonse
 * @author Nikos Michalakis <nikos@netflix.com>
 *
 */
public class RoundRobinRule extends AbstractLoadBalancerRule {

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

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

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

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

    public Server choose(ILoadBalancer lb, Object key) {
        if (lb == null) {
            log.warn("no load balancer");
            return null;
        }

        Server server = null;
        int count = 0;
        while (server == null && count++ < 10) {
            List<Server> reachableServers = lb.getReachableServers();
            List<Server> allServers = lb.getAllServers();
            int upCount = reachableServers.size();
            int serverCount = allServers.size();

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

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

            if (server == null) {
                /* Transient. */
```

```
                Thread.yield();
                continue;
            }

            if (server.isAlive() && (server.isReadyToServe())) {
                return (server);
            }

            // Next.
            server = null;
        }

        if (count >= 10) {
            log.warn("No available alive servers after 7 tries from load
balancer: "
                    + lb);
        }
        return server;
    }

    /**
     * Inspired by the implementation of {@link
AtomicInteger#incrementAndGet()}.
     *
     * @param modulo The modulo to bound the value of the counter.
     * @return The next value.
     */
    private int incrementAndGetModulo(int modulo) {
        for (;;) {
            int current = nextServerCyclicCounter.get();
            int next = (current + 1) % modulo;
            if (nextServerCyclicCounter.compareAndSet(current, next))
                return next;
        }
    }

    @Override
    public Server choose(Object key) {
        return choose(getLoadBalancer(), key);
    }

    @Override
    public void initWithNiwsConfig(IClientConfig clientConfig) {
    }
}
```

上述代码中 `incrementAndGetModulo` 方法涉及到自旋锁，请参考 **博客**

7.3.4 自定义算法

当有需要时可以自定义负载均衡算法，仿照RoundRobinRule定义算法，每5秒切换一次微服务

```
package org.jshand.ribbon;

import com.netflix.client.config.IClientConfig;
import com.netflix.loadbalancer.AbstractLoadBalancerRule;
import com.netflix.loadbalancer.ILoadBalancer;
import com.netflix.loadbalancer.Server;
```

```java
import lombok.extern.slf4j.Slf4j;

import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

@Slf4j
public class MySelfRule extends AbstractLoadBalancerRule {

    private AtomicInteger nextServerCyclicCounter;
    long lastTime = 0;
    int second_interval = 5;
    int index = 0;

    public MySelfRule() {
        nextServerCyclicCounter = new AtomicInteger(0);
    }

    @Override
    public void initWithNiwsConfig(IClientConfig iClientConfig) {
        //TODO
    }

    @Override
    public Server choose(Object key) {
        ILoadBalancer lb = getLoadBalancer();
        List<Server> servers = lb.getReachableServers();

        //计算与上次时间的间隔描述
        long time = (System.currentTimeMillis() - lastTime) / 1000 ;

        //与上次间隔超过5秒则切换 index
        if (time >=second_interval) {
            index = incrementAndGetModulo(servers.size());
            lastTime = System.currentTimeMillis();
        }
        log.info(System.currentTimeMillis() +"\t"+lastTime+"\t"+time);

        return servers.get(index);//根据index返回数据
    }


    /**
     * 使用原子性int(AtomicInteger) 在总数中递增
     * @param serverCount
     * @return
     */
    private int incrementAndGetModulo(int serverCount) {
        for (; ; ) {
            int current = nextServerCyclicCounter.get();
            int next = (current + 1) % serverCount;
            if (nextServerCyclicCounter.compareAndSet(current, next))
                return next;
        }
    }

}
```

8. OpenFeign

8.1 介绍

> github官网 Spring-Cloud官网2

作为Spring Cloud的子项目之一，Spring Cloud OpenFeign以将`OpenFeign`集成到Spring Boot应用中的方式，为微服务架构下服务之间的调用提供了解决方案。首先，利用了OpenFeign的声明式方式定义Web服务客户端；其次还更进一步，通过集成`Ribbon`或`Eureka`实现负载均衡的HTTP客户端。

在Spring Cloud OpenFeign中，除了OpenFeign自身提供的标注（annotation）之外，还可以使用JAX-RS标注，或者Spring MVC标注。下面还是以OpenFeign标注为例介绍用法。

> OpenFeign的标注@FeignClient和@EnableFeignClients

OpenFeign提供了两个重要标注@FeignClient和@EnableFeignClients。

@FeignClient标注用于声明Feign客户端可访问的Web服务。

@EnableFeignClients标注用于修饰Spring Boot应用的入口类，以通知Spring Boot启动应用时，扫描应用中声明的Feign客户端可访问的Web服务。

> @EnableFeignClients标注的参数

- value, basePackages (默认{})
- basePackageClasses (默认{})
- defaultConfiguration (默认{})
- clients (默认{})

> @FeignClient标注的参数

- name, value (默认""），两者等价
- qualifier (默认"")
- url (默认"")
- decode404 (默认false)
- configuration (默认FeignClientsConfiguration.class)
- fallback (默认void.class)
- fallbackFactory (默认void.class)
- path (默认"")
- primary (默认true)

> @FeignClient标注的configuration参数

@FeignClient标注的configuration参数，默认是通过FeignClientsConfiguration类定义的，可以配置Client，Contract，Encoder/Decoder等。

FeignClientsConfiguration类中的配置方法及默认值如下：

- **feignContract**: SpringMvcContract
- **feignDecoder**: ResponseEntityDecoder
- **feignEncoder**: SpringEncoder

- **feignLogger**: Slf4jLogger
- **feignBuilder**: Feign.Builder
- **feignClient**: LoadBalancerFeignClient（开启Ribbon时）或默认的HttpURLConnection

> 定制@FeignClient标注的configuration类

@FeignClient标注的默认配置类为FeignClientsConfiguration，我们可以定义自己的配置类如下：

```
@Configuration
public class MyConfiguration {

    @Bean
    public Contract feignContract(...) {....}

    @Bean
    public Encoder feignEncoder() {...}

    @Bean
    public Decoder feignDecoder() {...}
    ...
}
```

然后在使用@FeignClient标注时，给出参数如下：

```
@FeignClient(name = "myServiceName", configuration = MyConfiguration.class, ...)
public interface MyService {

    @RequestMapping("/")
    public String getName();
    ...
}
```

定制@FeignClient标注的configuration类还可以有另一个方法，直接配置**application.yaml**文件即可，示例如下：

```
feign:
  client:
    config:
      feignName: myServiceName
        connectTimeout: 5000
        readTimeout: 5000
        loggerLevel: full
        encoder: com.example.MyEncoder
        decoder: com.example.MyDecoder
        contract: com.example.MyContract
```

8.2 feign 和OpenFeign的区别

> Feign

Feign是Springcloud组件中的一个轻量级Restful的HTTP服务客户端，Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。Feign的使用方式是：使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务

> OpenFeign

OpenFeign是springcloud在Feign的基础上支持了SpringMVC的注解，如@RequestMapping等等。OpenFeign的@FeignClient可以解析SpringMVC的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。

8.3 OpenFeign的使用

8.3.1创建modules(子项目)

类型	值
groupId	`org.jshand` (与父项目保持一致即可)
artifactId	`10-cloud-customer-openfeign-80` (也可随意)

8.3.2 修改pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>springcloud-202101</artifactId>
        <groupId>org.jshand</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <packaging>jar</packaging>
    <artifactId>10-cloud-customer-openfeign-80</artifactId>

    <dependencies>


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

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

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</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>
        </dependency>
    </dependencies>

</project>
```

8.3.3 application.yaml配置文件

```
server:
  port: 80

spring:
  application:
    name: cloud-member-center-service

eureka:
  client:
    serviceUrl:
      defaultZone: http://server8760.com:8760/eureka
  instance:
    hostname: 127.0.0.1
    instance-id: order-80
    prefer-ip-address: true
```

8.3.4 主启动类

在主启动类上添加@EnableFeignClients注解，激活使用@FeignClient注解标注的Feign客户端。

```
package org.jshand.cloud;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

/**
 * 使用OpenFeign作为客户端
 */
@SpringBootApplication
@Slf4j
@EnableFeignClients
public class MemberCenterOpenFeignApp {

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

```
19
20  }
```

8.3.5 定义Feign接口

定义调用远程服务的接口，并声明 @FeignClient注解，其中name或者value为微服务名称(即注册中心的服务名)

```
1   package org.jshand.cloud.service;
2
3   import org.springframework.cloud.openfeign.FeignClient;
4   import org.springframework.stereotype.Component;
5   import org.springframework.web.bind.annotation.PathVariable;
6   import org.springframework.web.bind.annotation.RequestMapping;
7
8   /**
9    * 定义OpenFeign客户端接口
10   */
11  @Component
12  @FeignClient(name="PAYMENT")
13  public interface PaymentService {
14
15      @RequestMapping(value = "/echo/{string}")
16      public String echo(@PathVariable String string);
17  }
```

8.3.6 定义Controller调用Feign接口

```
1   package org.jshand.cloud.controller;
2
3   import lombok.extern.slf4j.Slf4j;
4   import org.jshand.cloud.service.PaymentService;
5   import org.springframework.web.bind.annotation.PathVariable;
6   import org.springframework.web.bind.annotation.RequestMapping;
7   import org.springframework.web.bind.annotation.RestController;
8
9   import javax.annotation.Resource;
10
11  @RestController
12  @Slf4j
13  public class MemberCenterController {
14
15      @Resource
16      PaymentService paymentService;
17
18      /**
19       * http://127.0.0.1:80/openfeign/consumer/echo/abc
20       * @param string
21       * @return
22       */
23      @RequestMapping(value = "/openfeign/consumer/echo/{string}")
24      public String echo(@PathVariable String string){
25          return paymentService.echo(string);
26      }
27
28  }
```

8.3.7 浏览器测试

使用地址 `http://127.0.0.1:80/openfeign/consumer/echo/abc` 进行测试，能够正常访问微服务，并提供负载均衡的特性。截图略

8.4 OpenFeign超时控制

默认OpenFeign客户端等待时间为1秒钟，如果服务端超过1秒钟没有返回，则客户端直接抛出Timeout，为了避免这样的情况，有时候我们需要处理Feign客户端的超时控制。

为了模拟上述问题，在服务端控制器方法中添加如下代码（如果是集群，需要每个应用都加），让服务端处理时间延长

```
try {
    TimeUnit.SECONDS.sleep(3);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

添加完上述代码后，浏览器单独访问服务端成功没有问题，但是使用OpenFeign客户端调用时报如下错误

Whitelabel Error Page

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

Wed Jan 27 00:12:29 CST 2021
There was an unexpected error (type=Internal Server Error, status=500).
Read timed out executing GET http://PAYMENT/echo/abc
feign.RetryableException: Read timed out executing GET http://PAYMENT/echo/abc
at feign.FeignException.errorExecuting(FeignException.java:249)
at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:129)
at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:89)
at feign.ReflectiveFeign$FeignInvocationHandler.invoke(ReflectiveFeign.java:100)
at com.sun.proxy.$Proxy146.echo(Unknown Source)
at org.jshand.cloud.controller.MemberCenterController.echo(MemberCenterController.java:25)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:190)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:105)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:878)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:792)
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1040)
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:943)

为了解决上述问题，此时需要配置客户端。在客户端配置Feign客户端超时时间。其他OpenFeign配置参考 **链接**

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 5000
```

或者直接设置Ribbon的超时时间, 其他配置请参考 **连接**

```
ribbon:
  ConnectTimeout: 8000
  ReadTimeout: 8000
```

8.5 日志打印功能

OpenFeign提供了打印日志功能，提供了如下级别。开发过程中可以通过配置来调整日志级别，从而了解OpenFeign在调用http请求的过程中的细节。

- NONE： 默认的，不显示任何日志
- BASIC： 仅记录请求方法、URL、响应状态码以及执行时间
- HEADERS：除了BASIC 中自定义的信息外，还有请求和响应的信息头
- FULL： 除了HEADERS中定义的信息外， 还有请求和响应的正文以及元数据。

以Full类型为例，需要做如下功能：

8.5.1 定义配置类

定义配置类，用于提供日志类型

```java
package org.jshand.cloud.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;
    }
}
```

8.5.2 配置日志级别

在application.yml中配置FeignClient客户端类的日志级别

```yaml
logging:
  level:
    org.jshand.cloud.service.PaymentService: debug
```

测试后控制台打印如下日志

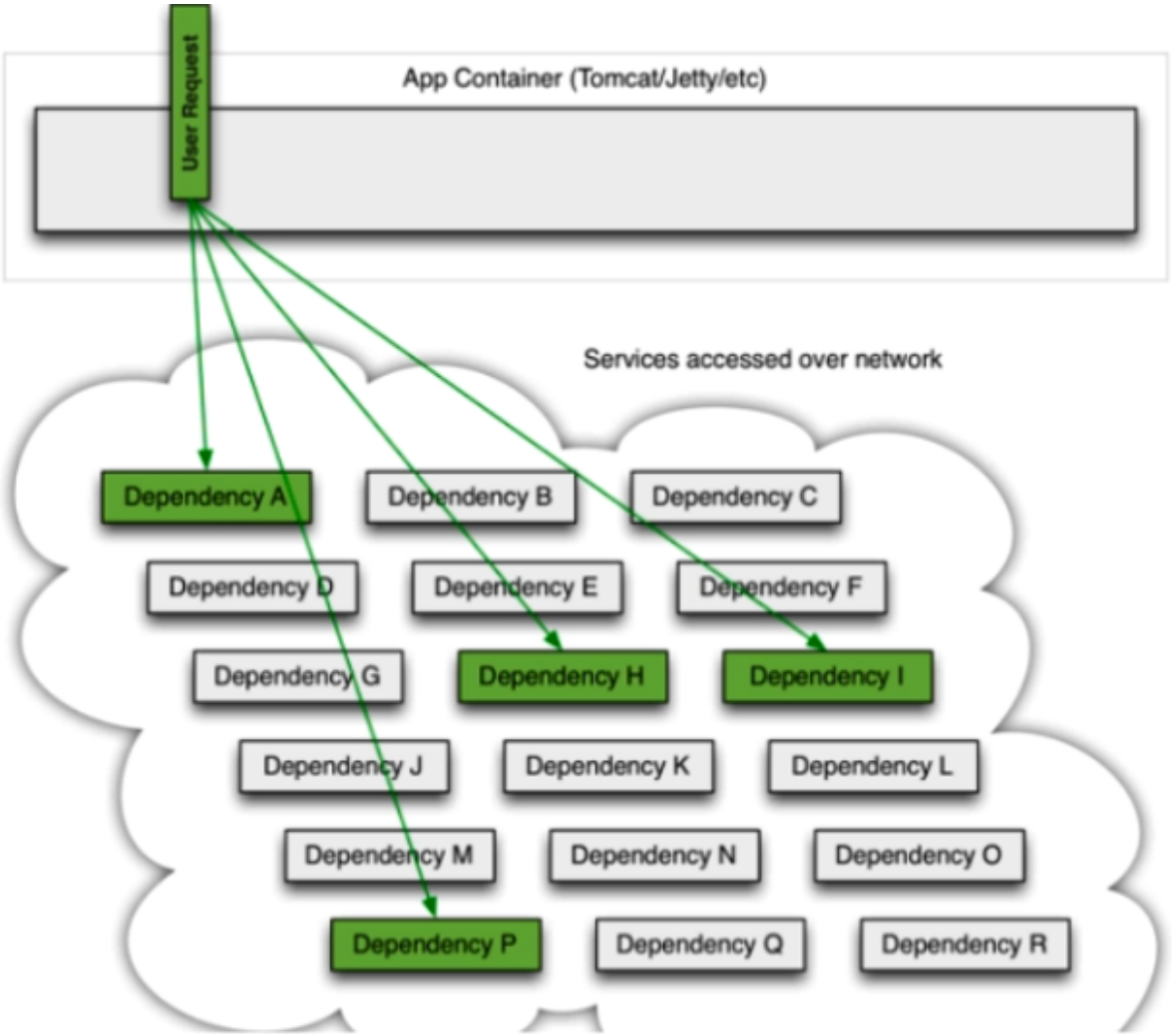
```
- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty  : Flipping property: PAYMENT.ribbon.ActiveConnectionsLimit to
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] <--- HTTP/1.1 200 (3124ms)
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] connection: keep-alive
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] content-length: 29
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] content-type: text/plain;charset=UTF-8
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] date: Tue, 26 Jan 2021 17:10:20 GMT
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] keep-alive: timeout=60
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo]
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] Provider serverPort: 8001    abc
- [p-nio-80-exec-1] org.jshand.cloud.service.PaymentService  : [PaymentService#echo] <--- END HTTP (29-byte body)
```

9.Hystrix 断路器

9.1 概述

> 分布式系统面临的问题

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



> 服务雪崩

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

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

一般情况对于服务依赖的保护主要有3中解决方案：

（1） `熔断模式`：这种模式主要是参考电路熔断，如果一条线路电压过高，保险丝会熔断，防止火灾。放到我们的系统中，如果某个目标服务调用慢或者有大量超时，此时，熔断该服务的调用，对于后续调用请求，不在继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

（2） `隔离模式`：这种模式就像对系统请求按类型划分成一个个小岛的一样，当某个小岛被火烧光了，不会影响到其他的小岛。例如可以对不同类型的请求使用线程池来资源隔离，每种类型的请求互不影响，如果一种类型的请求线程资源耗尽，则对后续的该类型请求直接返回，不再调用后续资源。这种模式使用场景非常多，例如将一个服务拆开，对于重要的服务使用单独服务器来部署，再或者公司最近推广的多中心。

（3） `限流模式`：上述的熔断模式和隔离模式都属于出错后的容错处理机制，而限流模式则可以称为预防模式。限流模式主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。这种模式不能解决服务依赖的问题，只能解决系统整体资源分配问题，因为没有被限流的请求依然有可能造成雪崩效应。

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

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

hystrix-github官网 https://github.com/netflix/hystrix/

> **停止更新**，进入维护期

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.

9.2 重要概念

9.2.1 服务降级 fallback

概念：服务器繁忙，请稍后重试，不让客户端等待并立即返回一个友好的提示。fallback

出现服务降级的情况：

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

9.2.2 服务熔断 break

概念： 类比 保险丝 ，达到最大访问后，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的提示。

服务熔断-->服务降级-->恢复调用链路

9.2.3 服务限流 flowlimit

概念： 秒杀高并发等操作，眼睛一窝蜂的过来拥挤，进行排队，一秒中N个，有序进行

9.3. 案例

9.3.1搭建带熔断的服务

首先使用Eureka单机版搭建正常的开发环境

- 创建项目，具体信息如下

Title	值	备注
artifactId	11-cloud-provider-hystrix-order-8001	
groupId	org.jshand.cloud	
version	1.0-SNAPSHOT	

- 修改pom

```yaml
server:
  port: 8001

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

eureka:
  instance:
    hostname: 97.0.0.1
    instance-id: cloud-order-hystrix-sericee-8001
    prefer-ip-address: on
  client:
    serviceUrl:
      defaultZone: http://server8760.com:8760/eureka/

logging:
  level:
    org.jshand.cloud: debug
```

- 主启动类

```java
package org.jshand.cloud;


import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * 带熔断器的微服务 主启动类
 */
@SpringBootApplication
@Slf4j
public class ProviderHystrixOrderApp8001 {
    public static void main(String[] args) {
        SpringApplication.run(ProviderHystrixOrderApp8001.class, args);
    }
}
```

- 定义Service层代码，一个正常返回，一个延迟3秒返回

```java
package org.jshand.cloud.service;

import org.springframework.stereotype.Service;

@Service
public class OrderService {
```

```
7
8       /**
9        * 立即处理完成并返回，模拟快速返回
10       * @param id
11       * @return
12       */
13      public String getOrderOk(Integer id){
14          return String.format(&quot;微服务(OK) 线程:%s,查询订单id:%d,(*￣
    ￣)&quot;,
15                      Thread.currentThread().getName(),
16                      id
17                  );
18
19      }
20
21      /**
22       * 延迟3秒的  模拟长时间执行的业务
23       * @param id
24       * @return
25       */
26      public String getOrderTimeout(Integer id){
27          int timesecond = 3;
28          try { Thread.sleep(timesecond);  } catch (InterruptedException e) {
      e.printStackTrace(); }
```

```
1       return String.format(&quot;微服务(Timeout(秒):%d, 线程:%s,查询订单id:%d,(*￣
    ￣)&quot;,
2               timesecond,
3               Thread.currentThread().getName(),
4               id
5       );
6
7    }
```

```
}
```

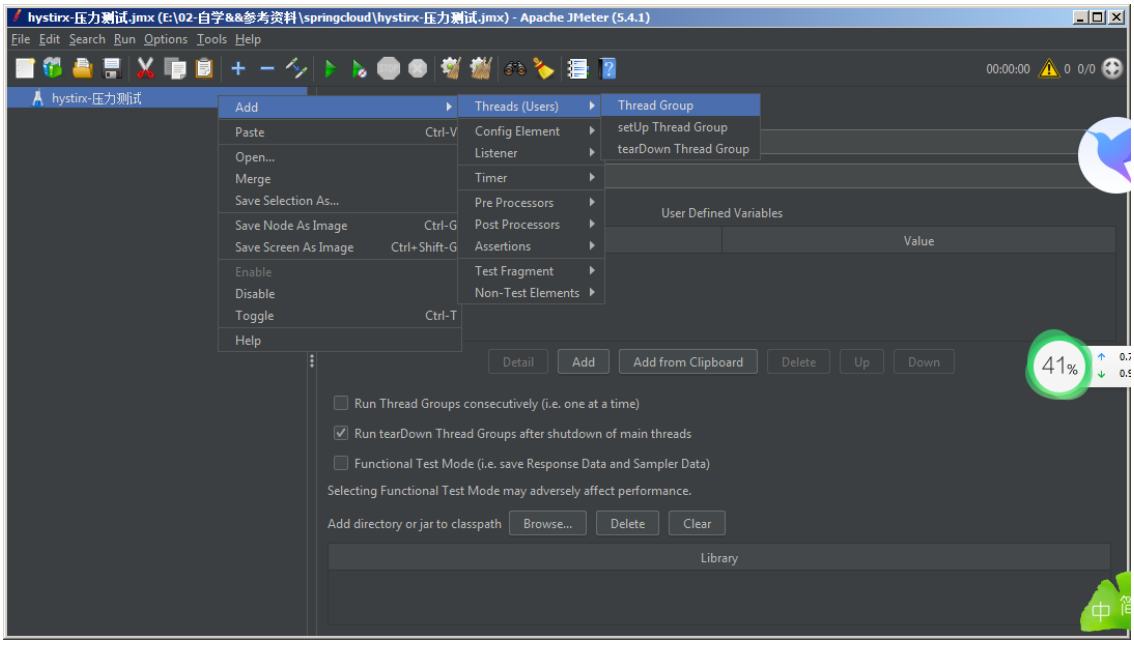
```
1   - 定义Controller
2
3   ```java
4   package org.jshand.cloud.controller;
5
6   import lombok.extern.slf4j.Slf4j;
7   import org.jshand.cloud.service.OrderService;
8   import org.springframework.web.bind.annotation.PathVariable;
9   import org.springframework.web.bind.annotation.RequestMapping;
10  import org.springframework.web.bind.annotation.RestController;
11
12  import javax.annotation.Resource;
13
14  @RestController
15  @Slf4j
16  public class OrderController {
17
18    @Resource
19    OrderService orderService;
20
21    /**
22     * 正常的
```

```
     *     http://127.0.0.1:8001/provider/hystrix/getOrderOk/1
     *     http://127.0.0.1:8001/provider/hystrix/getOrderOk/2
     *     http://127.0.0.1:8001/provider/hystrix/getOrderOk/3
     * @param id
     * @return
     */
    @RequestMapping("/provider/hystrix/getOrderOk/{id}")
    public String getOrderOk(@PathVariable  Integer id){
        String result = orderService.getOrderOk(id);
        log.info("************"+result);
        return result;
    }

    /**
     * 延迟返回的
     *      http://127.0.0.1:8001/provider/hystrix/getOrderTimeout/1
     *      http://127.0.0.1:8001/provider/hystrix/getOrderTimeout/2
     *      http://127.0.0.1:8001/provider/hystrix/getOrderTimeout/3
     * @param id
     * @return
     */
    @RequestMapping("/provider/hystrix/getOrderTimeout/{id}")
    public String getOrderTimeout(@PathVariable  Integer id){
        String result = orderService.getOrderTimeout(id);
        log.info("************"+result);
        return result;
    }

}
```

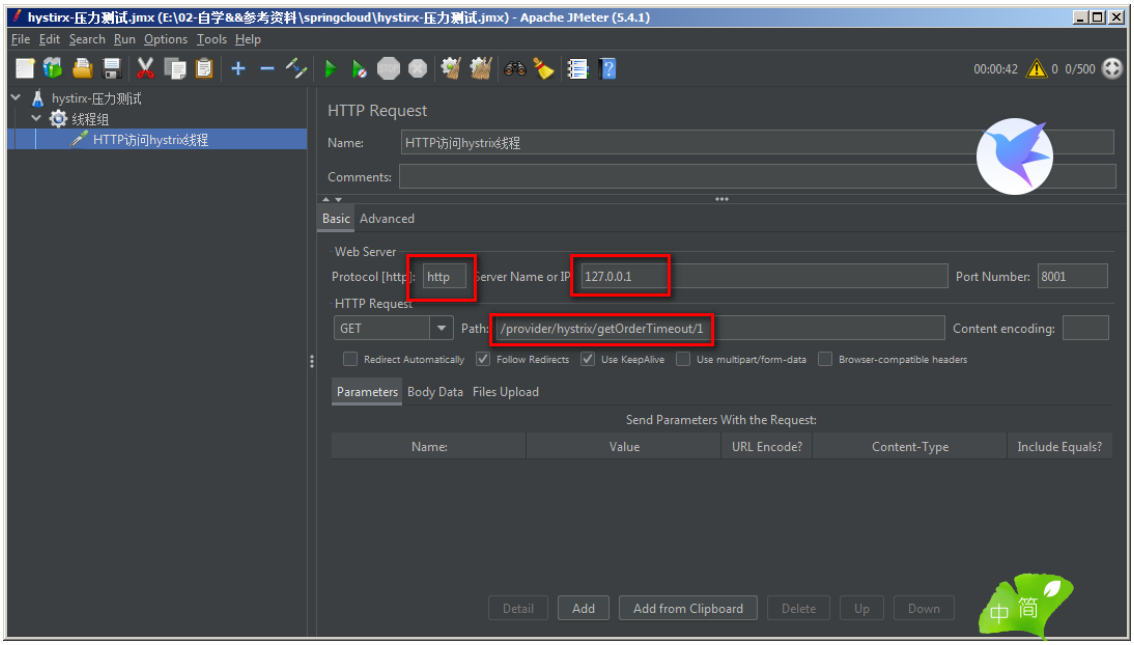
9.3.2 使用Jmeter压测

使用Apache-Jmeter工具模拟大规模访问getOrderTimeout

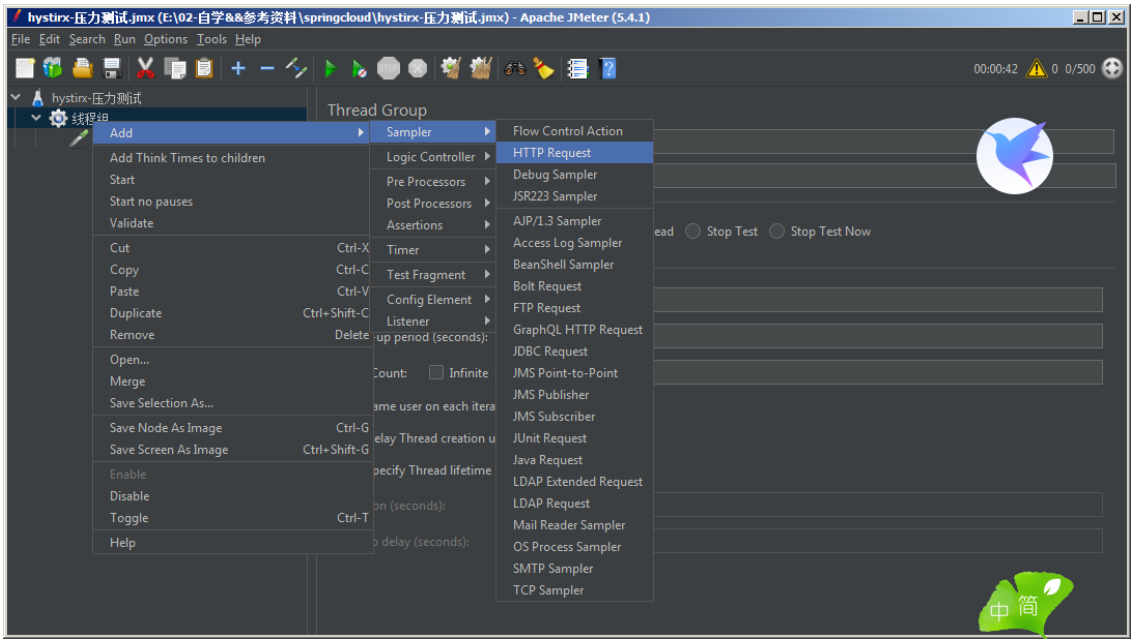
- 下载Jmeter
  - 下载地址 https://mirrors.tuna.tsinghua.edu.cn/apache//jmeter/binaries/apache-jmeter-5.4.1.zip
- 配置线程组 20000（2W)个线程, 视服务器配置可以适当调整

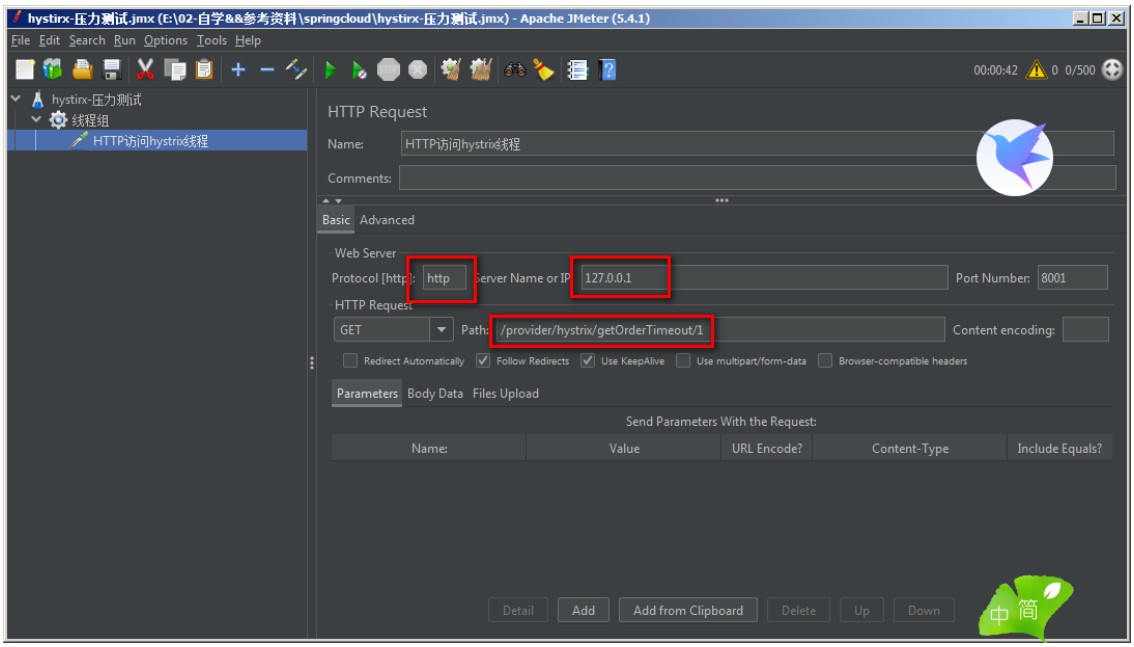


- 设置http请求路径
  - 创建http请求



  - 设置路径





- 测试
  - 启动jmeter压测后，发现**浏览器**正常访问的路径 http://127.0.0.1:8001/provider/hystrix/getOrderOk/1 也会变慢。 此时如果是consumer客户端发起的请求，可能导致服务超时，无法使用 。
- 卡顿原因
  - 由于压力测试工具大量消耗Tomcat线程池，导致服务器无法即时的处理 getOrderOk 的请求

9.3.3 创建消费方（用户中心）

key	value	remark
groupId	org.jshand.cloud	
artifactId	12-cloud-consumer-hystrix-order-80	
version	1.0-SNAPSHOT	

- 修改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>springcloud-202101</artifactId>
        <groupId>org.jshand.cloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>12-cloud-consumer-hystrix-membercenter-80</artifactId>

    <dependencies>

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

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

        <!--lombok工具-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>

        <!--spring-web工具-->
```

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

        <!--健康检查、监控-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

        <!-- springboot开发工具 、热部署 建议热部署时使用，optional为true意味着依赖不会传递给外部-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>

</project>
```

- 修改application.yml

```
server:
  port: 80

spring:
  application:
    name: cloud-member-center--hystrix-service

eureka:
  client:
    serviceUrl:
      defaultZone: http://server8760.com:8760/eureka
  instance:
    hostname: 127.0.0.1
    instance-id: cloud-member-center--hystrix-service-80
    #注册中心中点击超链接时使用ip地址
    prefer-ip-address: true


logging:
  level:
    #设置service目录 日志级别为 debug
    org.jshand.cloud.service: debug
```

- 主启动类

```
package org.jshand.cloud;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

/**
 * 使用hystrix熔断器的客户端 app启动类
```

```
 */
@SpringBootApplication
@Slf4j
@EnableFeignClients
public class MemberCenterHystrixApp80 {
    public static void main(String[] args) {
        SpringApplication.run(MemberCenterHystrixApp80.class, args);
    }
}
```

- 业务接口(Service)

```
package org.jshand.cloud.service;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@Component
@FeignClient(name="CLOUD-ORDER-HYSTRIX-SERICE")
public interface OrderService {

    @RequestMapping("/provider/hystrix/getOrderOk/{id}")
    public String getOrderOk(@PathVariable Integer id);

    @RequestMapping("/provider/hystrix/getOrderTimeout/{id}")
    public String getOrderTimeout(@PathVariable  Integer id);
}
```

- 定义Controller

```
package org.jshand.cloud.controller;

import lombok.extern.slf4j.Slf4j;
import org.jshand.cloud.service.OrderService;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

@RestController
@Slf4j
public class MemberCenterHystrixController {

    @Resource
    private OrderService orderService;


    /**
     * http://127.0.0.1:80/consumer/hystrix/getOrderOk/100
     * @param id
     * @return
     */
    @RequestMapping("/consumer/hystrix/getOrderOk/{id}")
    public String getOrderOk(@PathVariable Integer id){
        String result = orderService.getOrderOk(id);
```

```
27          log.info("******** result:"+result);
28          return result;
29      }
30
31      /**
32       * http://127.0.0.1:80/consumer/hystrix/getOrderTimeout/100
33       * @param id
34       * @return
35       */
36      @RequestMapping("/consumer/hystrix/getOrderTimeout/{id}")
37      public String getOrderTimeout(@PathVariable  Integer id){
38          String result = orderService.getOrderTimeout(id);
39          log.info("******** result:"+result);
40          return result;
41      }
42
43  }
```

- jmeter压测+客户端访问。consumer客户端出现同样的问题访问 http://127.0.0.1:80/consumer/hystrix/getOrderOk/100 ，同样出现转圈圈，甚至可能出现TImeout的情况.

Whitelabel Error Page

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

Wed Jan 27 19:34:57 CST 2021
There was an unexpected error (type=Internal Server Error, status=500).
Read timed out executing GET http://CLOUD-ORDER-HYSTRIX-SERICE/provider/hystrix/getOrderTimeout/100
feign.RetryableException: Read timed out executing GET http://CLOUD-ORDER-HYSTRIX-SERICE/provider/hystrix/getOrderTimeout/100
    at feign.FeignException.errorExecuting(FeignException.java:249)
    at feign.SynchronousMethodHandler.executeAndDecode(SynchronousMethodHandler.java:129)
    at feign.SynchronousMethodHandler.invoke(SynchronousMethodHandler.java:89)
    at feign.ReflectiveFeign$FeignInvocationHandler.invoke(ReflectiveFeign.java:100)

9.4

正是因为有上述的问题，所以才需要我们引入服务降级、容错、限流等技术。

> 超时导致服务器变慢（转圈）超时不再等待。

> 出错（宕机或服务运行出错），要有友好的提示，出错的解决方案。而不是直接报错错误。

- 对方服务（provider）超时，调用者（consumer）不能一直等待，需要服务降级
- 对方服务（provider）宕机了，调用者（consumer）不能直接报错，需要服务降级
- 对方服务（provider）ok，调用者（consumer）自己出故障了或者自己有要求（自己的等待时间小于服务提供者），自己需要服务降级。

9.5 服务降级 fallback

9.5.1 服务侧进行处理

1）在主启动类上 激活Hystrix，添加@EnableHystrix或者@EnableCircuitBreaker注解

```
package org.jshand.cloud;


import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;

/**
 * 带熔断器的微服务 主启动类
 */
@SpringBootApplication
@Slf4j
@EnableHystrix
public class ProviderHystrixOrderApp8001 {
    public static void main(String[] args) {
        SpringApplication.run(ProviderHystrixOrderApp8001.class, args);
    }
}
```

2）使用需要进行降级的程序上添加@HystrixCommand注解，进行处理

```
@HystrixCommand(
    fallbackMethod = "timeoutHandler",
    commandProperties = {
        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
value="3000")
})
```

- fallbackMethod 降级后调用的方法。
- commandProperties 设置降级属性
  - execution.isolation.thread.timeoutInMilliseconds 为超时时间，默认为1
  - 更多属性配置，参考 https://github.com/Netflix/Hystrix/wiki/Configuration
- PS: 修改注解，`尽量自己手动重启，不要热部署`，防止一些devtools对注解热部署不生效。

3）添加降级处理方法timeoutHandler

```
public String timeoutHandler(Integer id){
    return "服务异常或超时，请稍后再试";
}
```

4）除上述超时外也支持服务报错，可以手动将程序设置一个异常后同样也会调用降级的方法

9.5.2 消费侧处理

1. 开启Feign支持hystrix，由于consumer消费侧使用OpenFeign调用，需要开启Feign支持hystrix。在yaml文件中添加如下配置：

```
feign:
  hystrix:
    enabled: true
```

2） 主启动类添加激活hystrix注解 `@EnableHystrix` 或 `@EnableCircuitBreaker`

3）同消费侧处理方式一样，使用@

9.4 工作流程

9.5 服务监控hystrixDashBoard

10.Nacos

参考：

zookeeper的服务注册与发现

Consul的服务注册与发现

三个服务注册中心的异同