SpringCloud

聊聊现在和未来

这是我们学习到最后的微服务架构SpringCloud；

大家目前应该已经掌握如下知识

- MyBatis
- Spring
- SpringMVC
- SpringBoot
- Maven，Git
- Ajax，JSON
- Vue
- 。。。等等技术

聊聊现在和未来，回顾以前，架构！

```
三层架构 + MVC
    架构 -----> 结构

开发框架
    Spring
        IOC  AOP
            IOC: 控制翻转
                约泡：
                    泡温泉，泡茶....，泡友
                    附近的人，打招呼。加微信，聊聊，天天聊，---> 约泡
                浴场：  温泉，茶庄...
                    直接进温泉，就有人和你一起了！

                原来我们都是自己一步步操作，现在交给容器了！我们需要什么就去拿就可以了

            AOP：切面 （本质：动态代理）
                为了解决什么？不影响业务本来的情况下，实现动态增加功能，大量应用在日志，事务，等方面

        Spring是一个轻量级的Java开源框架，容器
        目的：解决企业开发的复杂性问题
        Spring是春天，觉他是春天，但是随着时间推移，也开始慢慢复杂起来，配置文件多！

    SpringBoot
        SpringBoot并不是新东西，就是Spring的升级版！
        新一代JavaEE的开发标准，开箱即用！-> 拿过来就可以用！
        它自动帮我们配置了非常多的东西，我们拿来即用！
        特性：约定大约配置！

随着公司体系越来越大，用户越来越多！

微服务架构 ---> 新架构
    模块化！功能化！
    用户，支付，签到...；
    人越来越多：一台服务器解决不了了；我们可以再增加一台服务器，   横向扩展
    假设A服务器占用了98%的资源，B服务器只占用了10%。 --负载均衡
```

```
    将原来的整体项目（all in one），分成模块化，用户管理就是一个单独的项目，签到也是一个单独的项目；项目之间需要进行通信？那如何通信？
    用户非常多，签到十分少！   给用户多一点服务器，给签到少一点服务器！

微服务架构问题？
    分布式会遇到的四个核心问题？
    1. 这么多服务，客户端该如何去访问？
    2. 这么多服务，服务之间如何进行通信？
    3. 这么多服务，如何治理呢？
    4. 服务挂了，怎么办？

解决方案：
    SpringCloud，是一套生态，就是来解决以上分布式架构的4个问题
    想使用SpringCloud，必须要掌握SpringBoot，因为SpringCloud是基于SpringBoot；

    1.Spring Cloud NetFlix，出来了一套解决方案！一站式解决方案。我们需要的东西它都有。
        api网关，zuul组件
        Feign-->HttpClient-->Http的通信方式，同步并阻塞
        服务注册与发现，Eureka
        熔断机制，Hystrix

        2018年年底，NetFlix宣布无限期停止维护，生态不再维护，就会脱节。

    2.Apache Dubbo zookeeper，第二套解决方案
        API：没有！要么找第三方组件，要么自己实现
        Dubbo是一个高性能的基于Java实现的RPC通信框架！ 2.6.X
        服务注册与发现，zookeeper：动物园管理者（Hadoop，Hive）
        没有：借助Hystrix

        Dubbo这个并不完善！

    3.Spring Cloud Alibaba 一站式解决方案！


目前又推出服务网格的概念
    服务网格：下一代微服务标准，Server Mesh
    代表解决方案：istio（你们未来可能需要掌握！）

万变不离其宗，一通百通！
    1.API网关，服务路由
    2.HTTP，RPC框架，异步调用
    3.服务注册与发现，高可用
    4.熔断机制，服务降级

如果，你们基于这些问题，研发出一套解决方案，也可叫SpringCloud！

为什么要解决这些问题？ 本质：网络是不可靠的！

程序猿！
```

> 问大家几个问题

1、什么是微服务？

2、微服务之间是如何独立通讯的？

3、SpringCloud 和 Dubbo有哪些区别?

4、SpringBoot和SpringCloud，请你谈谈对他们的理解

5、什么是服务熔断？什么是服务降级

6、微服务的优缺点是分别是什么？说下你在项目开发中遇到的坑

7、你所知道的微服务技术栈有哪些？请列举一二

8、eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别?

微服务概述

> 什么是微服务

什么是微服务? 微服务（Microservice Architecture）是近几年流行的一种架构思想，关于它的概念很难一言以蔽之。究竟什么是微服务呢？我们在此引用 ThoughtWorks 公司的首席科学家 Martin Fowler 于2014年提出的一段话：

原文：https://martinfowler.com/articles/microservices.html

汉化：https://www.cnblogs.com/liuning8023/p/4493156.html

就目前而言，对于微服务，业界并没有一个统一的，标准的定义

但通常而言，微服务架构是一种架构模式，或者说是一种架构风格， **它提倡将单一的应用程序划分成一组小的服务**，每个服务运行在其独立的自己的进程内，服务之间互相协调，互相配置，为用户提供最终价值。服务之间采用轻量级的通信机制互相沟通，每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境中，另外，应尽量避免统一的，集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言，工具对其进行构建，可以有一个非常轻量级的集中式管理来协调这些服务，可以使用不同的语言来编写服务，也可以使用不同的数据存储；

可能有的人觉得官方的话太过生涩，我们从技术维度来理解下：

微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事情，从技术角度看就是一种小而独立的处理过程，类似进程的概念，能够自行单独启动或销毁，拥有自己独立的数据库。

> 微服务与微服务架构

微服务

强调的是服务的大小，他关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用，狭义的看，可以看做是IDEA中的一个个微服务工程，或者Moudel

微服务架构

一种新的架构形式，Martin Fowler，2014提出！

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调，互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务于服务间采用轻量级的通信机制互相协作，每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境中，另外，应尽量避免统一的，集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言，工具对其进行构建。

> 微服务优缺点

优点

- 每个服务足够内聚，足够小，代码容易理解，这样能聚焦一个指定的业务功能或业务需求；
- 开发简单，开发效率提高，一个服务可能就是专一的只干一件事；
- 微服务能够被小团队单独开发，这个小团队是2~5人的开发人员组成；
- 微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。
- 微服务能使用不同的语言开发。
- 易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如jenkins，Hudson，bamboo
- 微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果。无需通过合作才能体现价值。
- 微服务允许你利用融合最新技术。
- **微服务只是业务逻辑的代码，不会和 HTML， CSS 或其他界面混合**
- **每个微服务都有自己的存储能力，可以有自己的数据库，也可以有统一数据库**

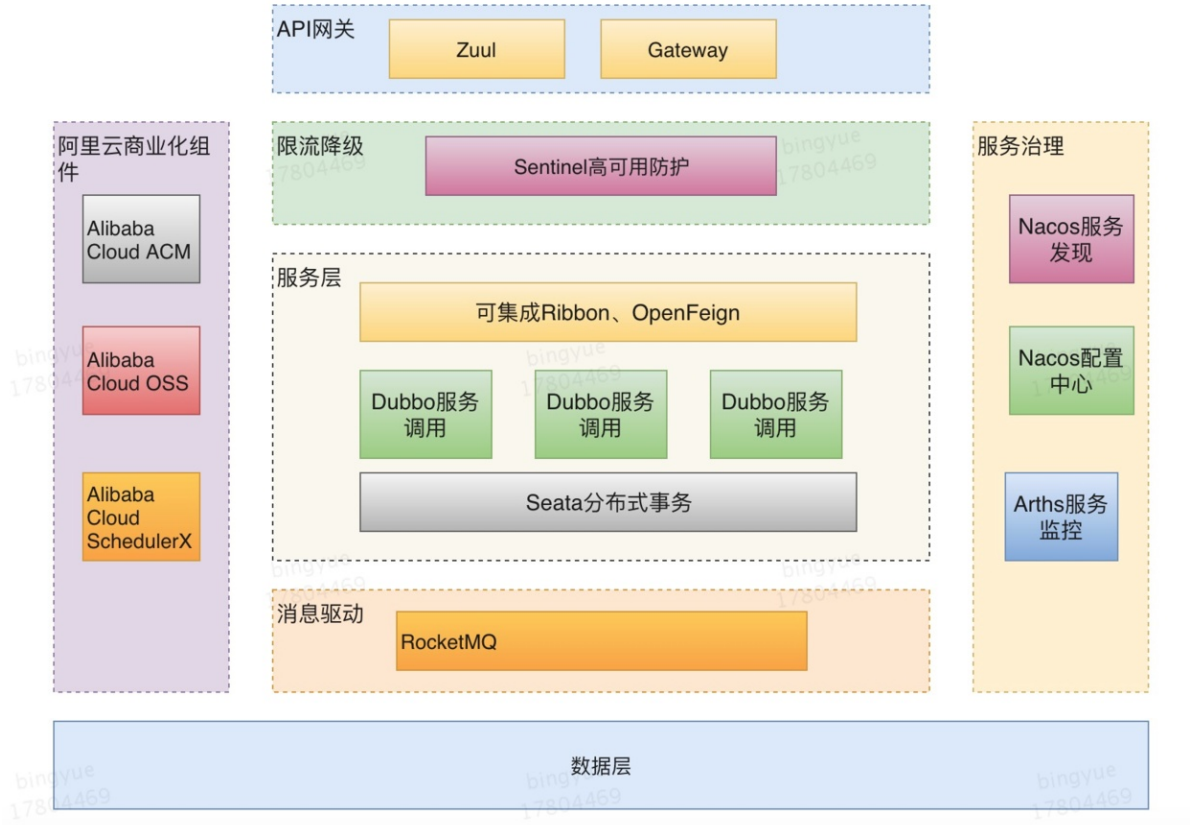
缺点：

- 开发人员要处理分布式系统的复杂性
- 多服务运维难度，随着服务的增加，运维的压力也在增大
- 系统部署依赖
- 服务间通信成本
- 数据一致性
- 系统集成测试
- 性能监控.....

> 微服务技术栈有哪些?

微服务条目	落地技术
服务开发	SpringBoot,Spring,SpringMVC
服务配置与管理	Netflix公司的Archaius、阿里的Diamond等
服务注册与发现	Eureka、Consul、Zookeeper等
服务调用	Rest、RPC、gRPC
服务熔断器	Hystrix、Envoy等
负载均衡	Ribbon、Nginx等
服务接口调用（客户端调用服务的简化工具）	Feign等
消息队列	Kafka、RabbitMQ、ActiveMQ等
服务配置中心管理	SpringCloudConfig、Chef等
服务路由（API网关）	Zuul等
服务监控	Zabbix、Nagios、Metrics、Specatator等
全链路追踪	Zipkin、Brave、Dapper等
服务部署	Docker、OpenStack、Kubernetes等
数据流操作开发包	SpringCloud Stream(封装与Redis，Rabbit，Kafka等发送接收消息)
事件消息总线	SpringCloud Bus

Spring Cloud Alibaba



> 为什么选择SpringCloud作为微服务架构

1、选型依据

- 整体解决方案和框架成熟度
- 社区热度
- 可维护性
- 学习曲线

2、当前各大IT公司用的微服务架构有哪些?

- 阿里：dubbo+HFS
- 京东：JSF
- 新浪：Motan
- 当当网 DubboX
- .....

3、各微服务框架对比

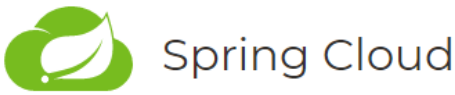
功能点/服务框架	Netflix/SpringCloud	Motan	gRPC	Thrift	Dubbo/DubboX
功能定位	完整的微服务框架	RPC框架，但整合了ZK或Consul，实现集群环境的基本服务注册/发现	RPC框架	RPC框架	服务框架
支持Rest	是，Ribbon支持多种可插拔的序列化选择	否	否	否	否
支持RPC	否	是（Hession2）	是	是	是
支持多语言	是（Rest形式）？	否	是	是	否
负载均衡	是（服务端zuul+客户端Ribbon），zuul-服务，动态路由，云端负载均衡Eureka（针对中间层服务器）	是（客户端）	否	否	是（客户端）
配置服务	Netfix Archaius，Spring Cloud Config Server集中配置	是（zookeeper提供）	否	否	否
服务调用链监控	是（zuul），zuul提供边缘服务，API网关	否	否	否	否
高可用/容错	是（服务端Hystrix+客户端Ribbon）	是（客户端）	否	否	是（客户端）
典型应用案例	Netflix	Sina	Google	Facebook	
社区活跃程度	高	一般	高	一般	2017年后重新开始维护，之前中断了5年
学习难度	中断	低	高	高	低
文档丰富程度	高	一般	一般	一般	高

功能点/服务框架	Netflix/SpringCloud	Motan	gRPC	Thrift	Dubbo/DubboX
其他	Spring Cloud Bus为我们的应用程序带来了更多管理端点	支持降级	Netflix内部在开发集成gRPC	IDL定义	实践的公司比较多

SpringCloud入门

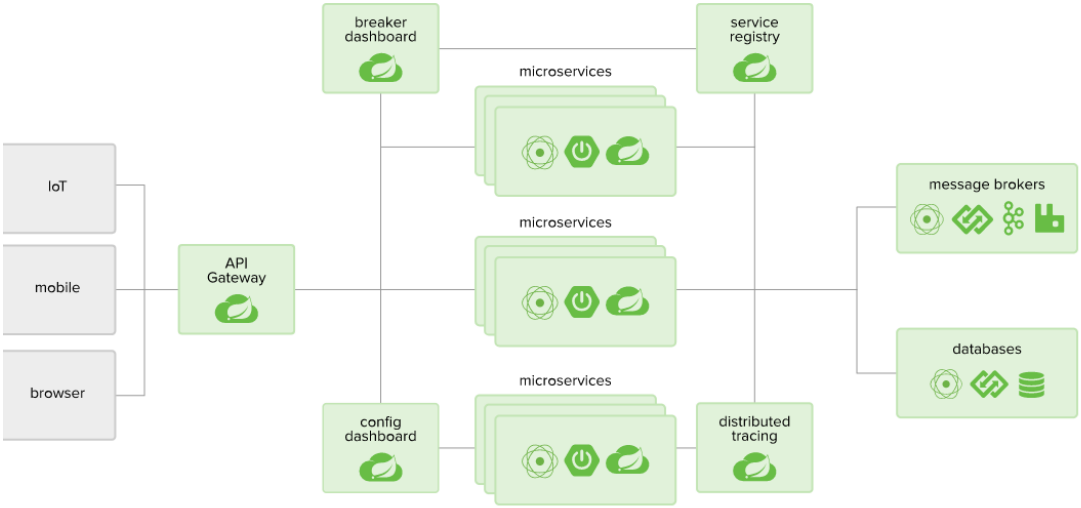
> SpringCloud是什么

Spring官网:https://spring.io/



COORDINATE ANYTHING: DISTRIBUTED SYSTEMS SIMPLIFIED

Building distributed systems doesn't need to be complex and error-prone. Spring Cloud offers a simple and accessible programming model to the most common distributed system patterns, helping developers build resilient, reliable, and coordinated applications. Spring Cloud is built on top of Spring Boot, making it easy for developers to get started and become productive quickly.



SpringCloud, 基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现，配置中心，全链路监控，服务网关，负载均衡，熔断器等组件，除了基于NetFlix的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。

SpringCloud利用SpringBoot的开发便利性，巧妙地简化了分布式系统基础设施的开发，SpringCloud为开发人员提供了快速构建分布式系统的一些工具，**包括配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，决策竞选，分布式会话等等**，他们都可以用SpringBoot的开发风格做到一键启动和部署。

SpringBoot并没有重复造轮子，它只是将目前各家公司开发的比较成熟，经得起实际考研的服务框架组合起来，通过SpringBoot风格进行再封装，屏蔽掉了复杂的配置和实现原理，**最终给开发者留出了一套简单易懂，易部署和易维护的分布式系统开发工具包**

SpringCloud 是 分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务全家桶。

> SpringCloud和SpringBoot关系

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供：配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，决策竞选，分布式会话等等集成服务。

SpringBoot可以离开SpringClooud独立使用，开发项目，但是SpringCloud离不开SpringBoot，属于依赖关系

SpringBoot专注于快速、方便的开发单个个体微服务，SpringCloud关注全局的服务治理框架

> Dubbo 和 SpringCloud 对比

	Dubbo	Spring
服务注册中心	Zookeeper	Spring Cloud Netfilx Eureka
服务调用方式	RPC	REST API
服务监控	Dubbo-monitor	Spring Boot Admin
断路器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

最大区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。
严格来说，这两种方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

品牌机与组装机的区别
很明显，Spring Cloud的功能比DUBBO更加强大，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而Spring Cloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

社区支持与更新力度
最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

总结：
曾风靡国内的开源 RPC 服务框架 Dubbo 在重启维护后，令许多用户为之雀跃，但同时，也迎来了一些质疑的声音。互联网技术发展迅速，Dubbo 是否还能跟上时代？Dubbo 与 Spring Cloud 相比又有何优势和差异？是否会有相关举措保证 Dubbo 的后续更新频率？

人物：Dubbo重启维护开发的刘军，主要负责人之一

刘军，阿里巴巴中间件高级研发工程师，主导了 Dubbo 重启维护以后的几个发版计划，专注于高性能 RPC 框架和微服务相关领域。曾负责网易考拉 RPC 框架的研发及指导在内部使用，参与了服务治理平台、分布式跟踪系统、分布式一致性框架等从无到有的设计与开发过程。

解决的问题域不一样：Dubbo的定位是一款RPC框架，Spring Cloud的目标是微服务架构下的一站式解决方案

> Dubbo 和 SpringCloud 对比

Distributed/versioned configuration （分布式/版本控制配置）

Service registration and discovery（服务注册与发现）

Routing（路由）

Service-to-service calls（服务到服务的调用）

Load balancing（负载均衡配置）

Circuit Breakers（断路器）

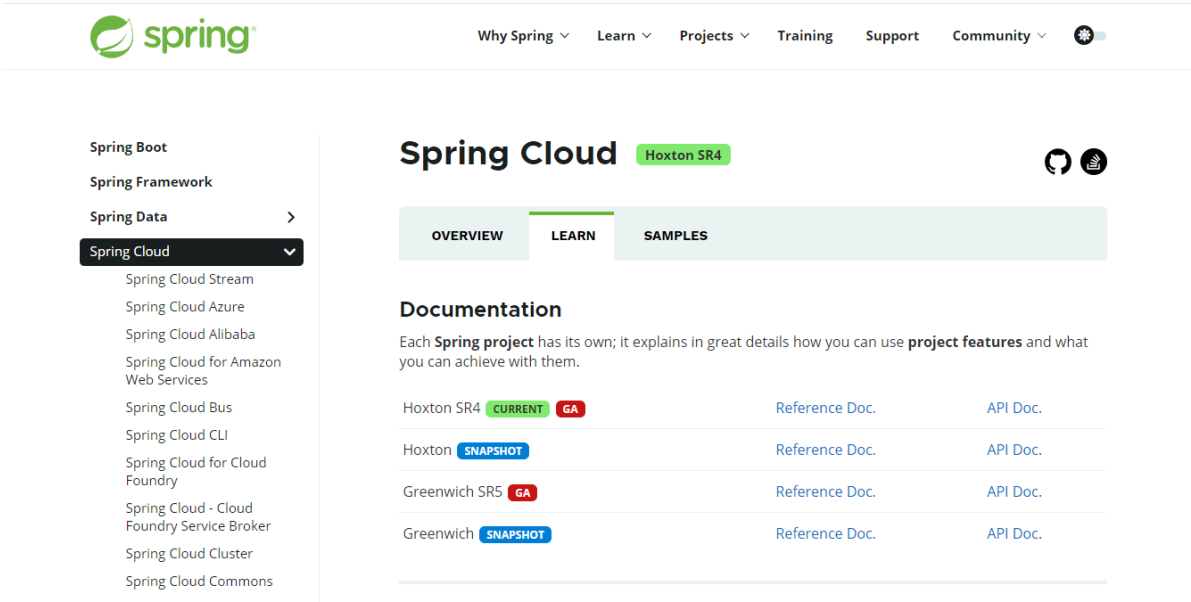
Distributed messaging（分布式消息管理）

....

> SpringCloud在哪下

官网：http://projects.spring.io/spring-cloud/

这玩意的版本号有点特别



Spring Cloud是一个由众多独立子项目组成的大型综合项目，每个子项目有不同的发行节奏，都维护着自己的发布版本号。Spring Cloud通过一个资源清单BOM（Bill of Materials）来管理每个版本的子项目清单。为避免与子项目的发布号混淆，所以没有采用版本号的方式，而是通过命名的方式。

这些版本名称的命名方式采用了伦敦地铁站的名称，同时根据字母表的顺序来对应版本时间顺序，比如：最早的Release版本：Angel，第二个Release版本：Brixton，然后是Camden、Dalston、Edgware，Finchley,目前最新的是 Hoxton 版本。

参考：

- https://springcloud.cc/spring-cloud-netflix.html
- 中文API文档：https://springcloud.cc/spring-cloud-dalston.html
- SpringCloud中国社区 http://springcloud.cn/
- SpringCloud中文网 https://springcloud.cc

上面和大家聊了这么多，希望大家能够认真吸收，这就是大家能够在面试中和别人的谈资。而且好像我也没说什么废话，之后的代码都会和上面的理论挂钩！ 所以需要认真掌握哈！

Rest微服务构建

> 总体介绍

我们会使用一个Dept部门模块做一个微服务通用案例

Consumer消费者（Client）通过REST调用Provider提供者（Server）提供的服务。

回忆Spring，SpringMVC，MyBatis等以往学习的知识。。。

Maven的分包分模块架构复习

```
一个简单的Maven模块结构是这样的：

-- app-parent：一个父项目（app-parent）聚合很多子项目（app-util，app-dao，app-web...）
  |-- pom.xml
  |
  |-- app-core
  ||----pom.xml
  |
  |-- app-web
  ||----pom.xml
  ......
```

一个父工程带着多个子Module子模块

SpringCloud父工程（Project）下初次带着3个子模块（Module）

- springcloud-api 【封装的整体entity / 接口 / 公共配置等】
- springcloud-provider-dept-8001【服务提供者】
- springcloud-consumer-dept-80【服务消费者】

> SpringCloud版本选择

大版本说明

Spring Boot	Spring Cloud	关系
1.2.x	Angel版本 (天使)	兼容Spring Boot 1.2.x
1.3.x	Brixton版本 (布里克斯顿)	兼容Spring Boot 1.3.x，也兼容Spring Boot 1.4.x
1.4.x	Camden版本 (卡姆登)	兼容Spring Boot 1.4.x，也兼容Spring Boot 1.5.x
1.5.x	Dalston版本 (多尔斯顿)	兼容Spring Boot 1.5.x，不兼容Spring Boot 2.0.x
1.5.x	Edgware版本 (埃奇韦尔)	兼容Spring Boot 1.5.x，不兼容Spring Boot 2.0.x
2.0.x	Finchley版本 (芬奇利)	兼容Spring Boot 2.0.x，不兼容Spring Boot 1.5.x
2.1.x	Greenwich版本 (格林威治)	兼容Spring Boot 2.1.x
2.2.x	Hoxton版本 (霍斯顿)	兼容Spring Boot 2.2.x

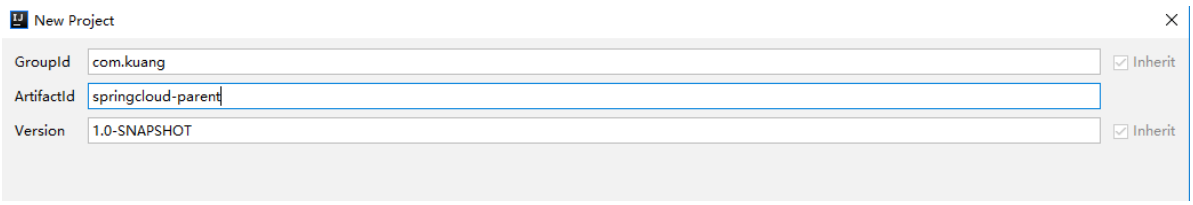
实际开发版本关系

spring-boot-starter-parent		spring-cloud-dependencies	
版本号	发布日期	版本号	发布日期
1.5.2.RELEASE	2017年3月	Dalston.RC1	2017年未知月
1.5.9.RELEASE	Nov, 2017	Edgware.RELEASE	Nov, 2017
1.5.16.RELEASE	Sep, 2018	Edgware.SR5	Oct, 2018
1.5.20.RELEASE	Apr, 2019	Edgware.SR5	Oct, 2018
2.0.2.RELEASE	May, 2018	Finchley.BUILD-SNAPSHOT	2018年未知月
2.0.6.RELEASE	Oct, 2018	Finchley.SR2	Oct, 2018
2.1.4.RELEASE	Apr, 2019	Greenwich.SR1	Mar, 2019

> 创建父工程

新建父工程Maven项目 springcloud-parent，切记Packageing是pom模式

主要是定义POM文件，将后续各个子模块公用的jar包等统一提取出来，类似一个抽象父类



pom.xml

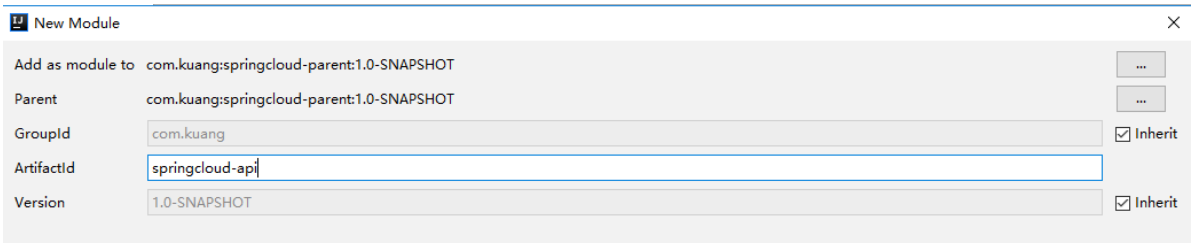
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         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>com.kuang</groupId>
    <artifactId>springcloud-parent</artifactId>
    <version>1.0-SNAPSHOT</version>


    <packaging>pom</packaging>

    <properties>
        <junit.version>4.12</junit.version>
        <log4j.version>1.2.17</log4j.version>
        <lombok.version>1.16.18</lombok.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <!-- spring-cloud-dependencies -->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Hoxton.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!-- spring-boot-dependencies -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.2.1.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!-- mysql-connector-java -->
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.1.47</version>
            </dependency>
            <!-- druid -->
            <dependency>
                <groupId>com.alibaba</groupId>
                <artifactId>druid</artifactId>
                <version>1.1.6</version>
            </dependency>
            <!-- mybatis-spring-boot-starter -->
            <dependency>
                <groupId>org.mybatis.spring.boot</groupId>
                <artifactId>mybatis-spring-boot-starter</artifactId>
                <version>1.3.0</version>
            </dependency>
            <dependency>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-core</artifactId>
                <version>1.2.3</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>
            </dependency>
        </dependencies>
    </dependencyManagement>

</project>
```

> 创建api公共模块

新建springcloud-api模块



可以观察发现，在父工程中多了一个Modules

编写springcloud-api 的 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-parent</artifactId>
        <groupId>com.kuang</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <!--当前Module的名字-->
```

```
    <artifactId>springcloud-api</artifactId>

    <!--当前Module需要到的jar包，按自己需求添加，如果父项目已经包含了，可以不用写版本号-->
    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
    </dependencies>

</project>
```

创建部门数据库脚本，数据库名：springcloud01

```
CREATE TABLE dept(
    deptno BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    dname VARCHAR(60),
    db_source VARCHAR(60)
);

INSERT INTO dept(dname,db_source) VALUES('开发部',DATABASE());
INSERT INTO dept(dname,db_source) VALUES('人事部',DATABASE());
INSERT INTO dept(dname,db_source) VALUES('财务部',DATABASE());
INSERT INTO dept(dname,db_source) VALUES('市场部',DATABASE());
INSERT INTO dept(dname,db_source) VALUES('运维部',DATABASE());

SELECT * FROM dept;
```

编写实体类，注意：实体类都序列化！

```
package com.kuang.springcloud.pojo;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.experimental.Accessors;

import java.io.Serializable;

@NoArgsConstructor
@Data
@Accessors(chain = true) //链式写法
public class Dept implements Serializable { //Dept(实体类)  orm   mysql->Dept(表)  类表关系映射

    private Long deptno; //主键
    private String dname; //部门名称
    //来自哪个数据库，因为微服务架构可以一个服务对应一个数据库，同一个信息被存到多个不同的数据库
    private String db_source;

    public Dept(String dname) {
        this.dname = dname;
    }

    /*
     链式写法：
        Dept dept = new Dept()
```

```
        dept.setDeptno(11L).setDname("school").setDb_source("DB01");
     **/
}
```

> 创建provider模块

新建springcloud-provider-dept-8001模块

编辑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-parent</artifactId>
        <groupId>com.kuang</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>springcloud-provider-dept-8001</artifactId>

    <dependencies>
        <!--引入自定义的模块，我们就可以使用这个模块中的类了-->
        <dependency>
            <groupId>com.kuang</groupId>
            <artifactId>springcloud-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jetty</artifactId>
        </dependency>
```

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

    </dependencies>

</project>
```

编辑 application.yml

```
server:
  port: 8001

#mybatis的配置
mybatis:
  config-location: classpath:mybatis/mybatis-config.xml
  type-aliases-package: com.kuang.springcloud.pojo
  mapper-locations:
   - classpath:mybatis/mapper/**/*.xml

#spring的相关配置
spring:
  application:
    name: springcloud-provider-dept
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource  # 数据源
    driver-class-name: org.gjt.mm.mysql.Driver    # mysql驱动
    url: jdbc:mysql://localhost:3306/springcloud01    #数据库名称
    username: root
    password: 123456
    dbcp2:
      min-idle: 5                                 #数据库连接池的最小维持连接数
      initial-size: 5                             #初始化连接数
      max-total: 5                                #最大连接数
      max-wait-millis: 200                        #等待连接获取的最大超时时间
```

根据配置新建mybatis-config.xml文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <settings>
        <!--开启二级缓存-->
        <setting name="cacheEnabled" value="true"/>
    </settings>

</configuration>
```

编写部门的dao接口

```java
package com.kuang.springcloud.dao;

import com.kuang.springcloud.pojo.Dept;
import org.apache.ibatis.annotations.Mapper;

import java.util.List;

@Mapper
public interface DeptDao {

    public boolean addDept(Dept dept); //添加一个部门

    public Dept queryById(Long id); //根据id查询部门

    public List<Dept> queryAll(); //查询所有部门

}
```

接口对应的Mapper.xml文件 mybatis\mapper\DeptMapper.xml

```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.kuang.springcloud.dao.DeptDao">

    <insert id="addDept" parameterType="Dept">
        insert into dept (dname,db_source) values (#{dname},DATABASE());
    </insert>

    <select id="queryById" resultType="Dept" parameterType="Long">
        select deptno,dname,db_source from dept where deptno = #{deptno};
    </select>

    <select id="queryAll" resultType="Dept">
        select deptno,dname,db_source from dept;
    </select>

</mapper>
```

创建Service服务层接口

```java
package com.kuang.springcloud.service;

import com.kuang.springcloud.pojo.Dept;

import java.util.List;

public interface DeptService {

    public boolean addDept(Dept dept); //添加一个部门

    public Dept queryById(Long id); //根据id查询部门

    public List<Dept> queryAll(); //查询所有部门

}
```

ServiceImpl实现类

```java
package com.kuang.springcloud.service.impl;

import com.kuang.springcloud.dao.DeptDao;
import com.kuang.springcloud.pojo.Dept;
import com.kuang.springcloud.service.DeptService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class DeptServiceImpl implements DeptService {

    //自动注入
    @Autowired
    private DeptDao deptDao;

    @Override
    public boolean addDept(Dept dept) {
        return deptDao.addDept(dept);
    }

    @Override
    public Dept queryById(Long id) {
        return deptDao.queryById(id);
    }

    @Override
    public List<Dept> queryAll() {
        return deptDao.queryAll();
    }

}
```

DeptController提供REST服务

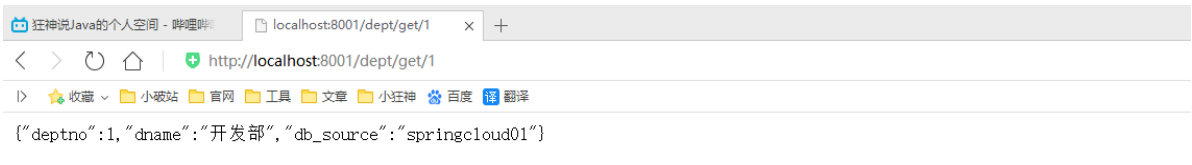
```java
package com.kuang.springcloud.controller;

import com.kuang.springcloud.pojo.Dept;
import com.kuang.springcloud.service.DeptService;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/dept")
public class DeptController {

    @Autowired
    private DeptService service;

    // @RequestBody
    // 如果参数是放在请求体中，传入后台的话，那么后台要用@RequestBody才能接收到
    @PostMapping("/add")
    public boolean addDept(@RequestBody Dept dept) {
        return service.addDept(dept);
    }

    @GetMapping("/get/{id}")
    public Dept get(@PathVariable("id") Long id) {
        return service.queryById(id);
    }

    @GetMapping("/list")
    public List<Dept> queryAll() {
        return service.queryAll();
    }

}
```

编写DeptProvider的主启动类

```java
package com.kuang.springcloud;

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

@SpringBootApplication
public class DeptProvider8001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptProvider8001.class,args);
    }
}
```

启动测试，注意编写细节：



> 创建consumer模块

新建springcloud-consumer-dept-80模块

编辑pom.xml

```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-parent</artifactId>
        <groupId>com.kuang</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>springcloud-consumer-dept-80</artifactId>
    <description>部门微服务消费者</description>

    <dependencies>
        <dependency>
            <groupId>com.kuang</groupId>
            <artifactId>springcloud-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!--热部署-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
    </dependencies>

</project>
```

application.yml 配置文件

```yaml
server:
  port: 80
```

新建一个ConfigBean包注入 RestTemplate!

```java
package com.kuang.springcloud.config;

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

@Configuration
public class ConfigBean {

    @Bean
```

```
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }

}
```

创建Controller包，编写DeptConsumerController类

```
package com.kuang.springcloud.controller;

import com.kuang.springcloud.pojo.Dept;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.List;

@RestController
public class DeptConsumerController {
    //理解：消费者，不应该有service层

    //  使用RestTemplate访问restful接口非常的简单粗暴且无脑
    //  （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
    @Autowired
    private RestTemplate restTemplate;

    private static final String REST_URL_PREFIX = "http://localhost:8001";

    @RequestMapping("/consumer/dept/add")
    public boolean add(Dept dept){
        return
restTemplate.postForObject(REST_URL_PREFIX+"/dept/add",dept,Boolean.class);
    }

    @RequestMapping("/consumer/dept/get/{id}")
    public Dept get(@PathVariable("id") Long id){
        return
restTemplate.getForObject(REST_URL_PREFIX+"/dept/get/"+id,Dept.class);
    }

    @RequestMapping("/consumer/dept/list")
    public List<Dept> list(){
        return
restTemplate.getForObject(REST_URL_PREFIX+"/dept/list",List.class);
    }

}
```

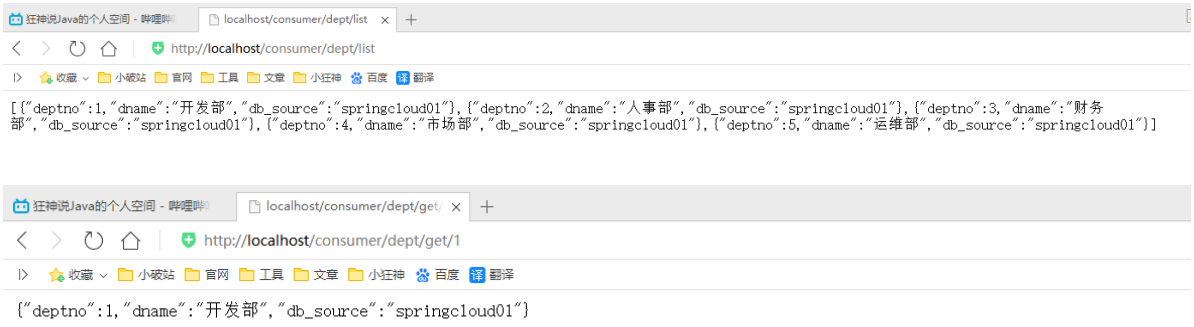
了解RestTemplate：

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

使用RestTemplate访问restful接口非常的简单粗暴且无脑
（url，requsetMap，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
```

主启动类

```java
package com.kuang.springcloud;

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

@SpringBootApplication
public class DeptConsumer80 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumer80.class,args);
    }
}
```

访问测试：



Eureka服务注册与发现

> 什么是Eureka

Eureka：怎么读?

Netflix 在设计Eureka 时，遵循的就是AP原则

```
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。
```

Eureka是Netflix的一个子模块，也是核心模块之一。Eureka是一个基于REST的服务，用于定位服务，以实现云端中间层服务发现和故障转移，服务注册与发现对于微服务来说是非常重要的，有了服务发现与注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了，功能类似于Dubbo的注册中心，比如Zookeeper；

> 原理讲解

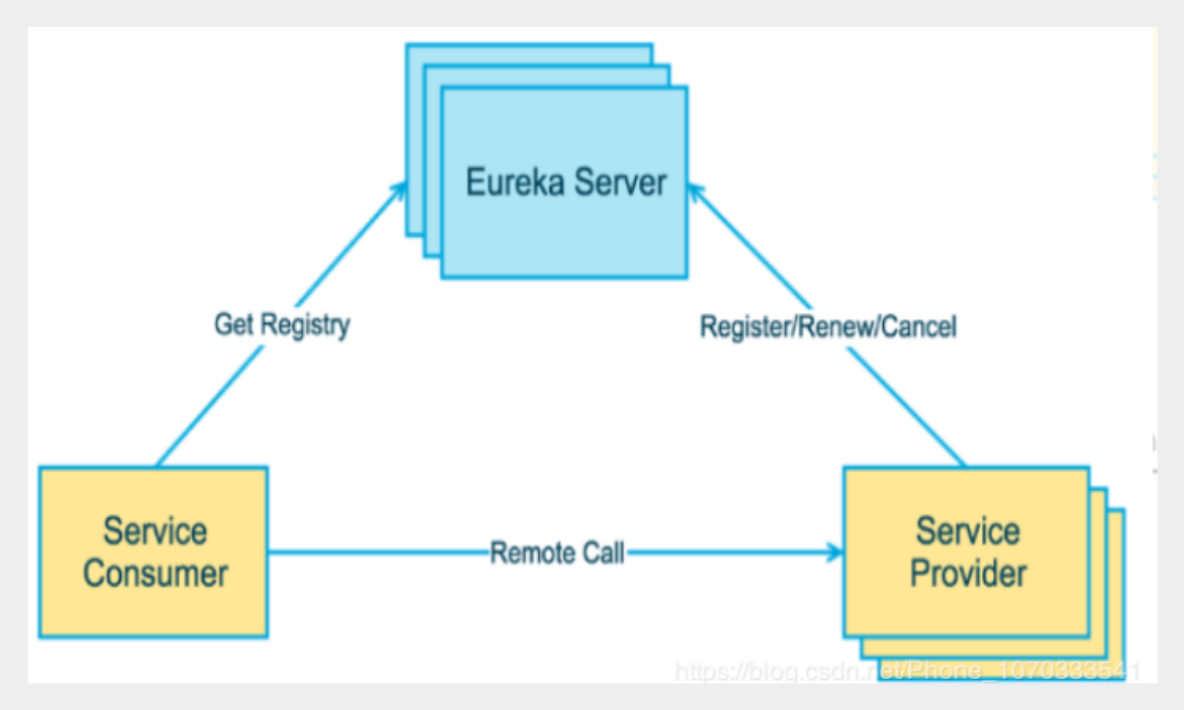
Eureka的基本架构：

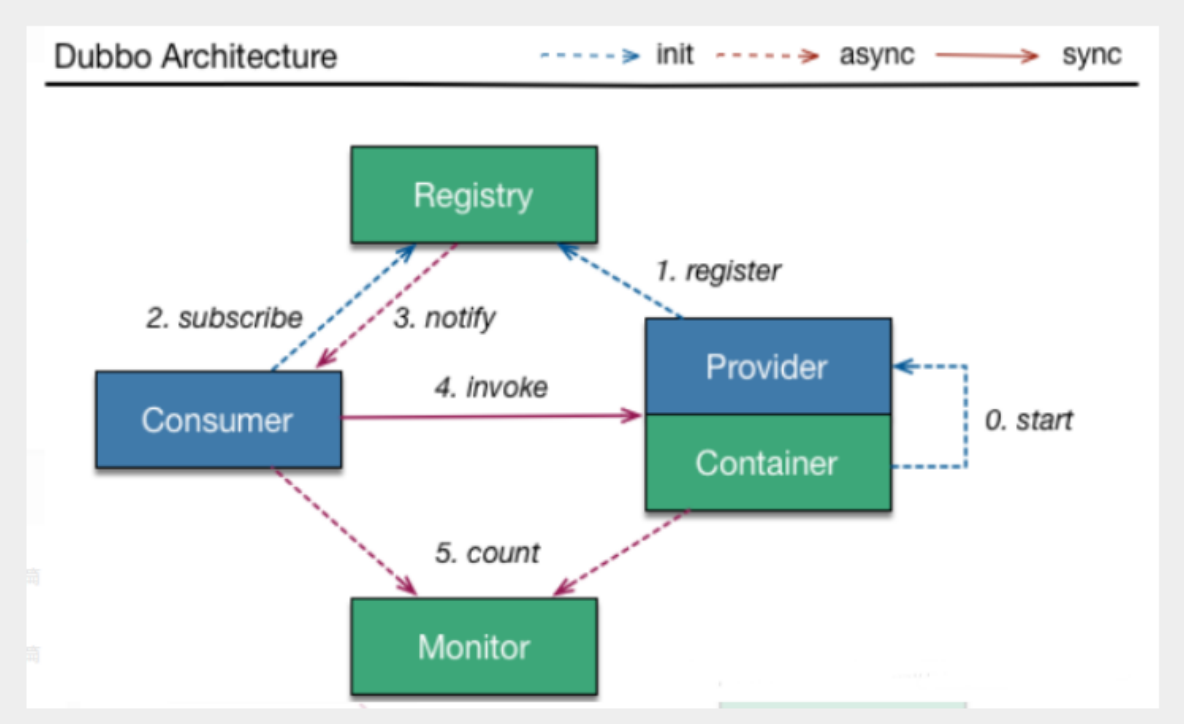
SpringCloud 封装了NetFlix公司开发的Eureka模块来实现服务注册和发现

Eureka采用了C-S的架构设计，EurekaServer 作为服务注册功能的服务器，他是服务注册中心

而系统中的其他微服务。使用Eureka的客户端连接到EurekaServer并维持心跳连接。这样系统的维护人员就可以通过EurekaServer来监控系统中各个微服务是否正常运行，SpringCloud的一些其他模块（比如Zuul）就可以通过EurekaServer来发现系统中的其他微服务，并执行相关的逻辑；

和Dubbo架构对比：





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

Eureka Server 提供服务注册服务，各个节点启动后，会在EurekaServer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的看到。

Eureka Client是一个Java客户端，用于简化EurekaServer的交互，客户端同时也具备一个内置的，使用轮询负载算法的负载均衡器。在应用启动后，将会向EurekaServer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掉（默认周期为90秒）

三大角色

- Eureka Server：提供服务的注册于发现。
- Service Provider：将自身服务注册到Eureka中，从而使消费方能够找到。
- Service Consumer：服务消费方从Eureka中获取注册服务列表，从而找到消费服务。

> 服务构建

建立springcloud-eureka-7001模块

编辑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-parent</artifactId>
        <groupId>com.kuang</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>springcloud-eureka-7001</artifactId>

    <dependencies>
        <!-- eureka-server服务端 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka-server</artifactId>
            <version>1.4.7.RELEASE</version>
        </dependency>
        <!--热部署-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
        </dependency>
    </dependencies>

</project>
```

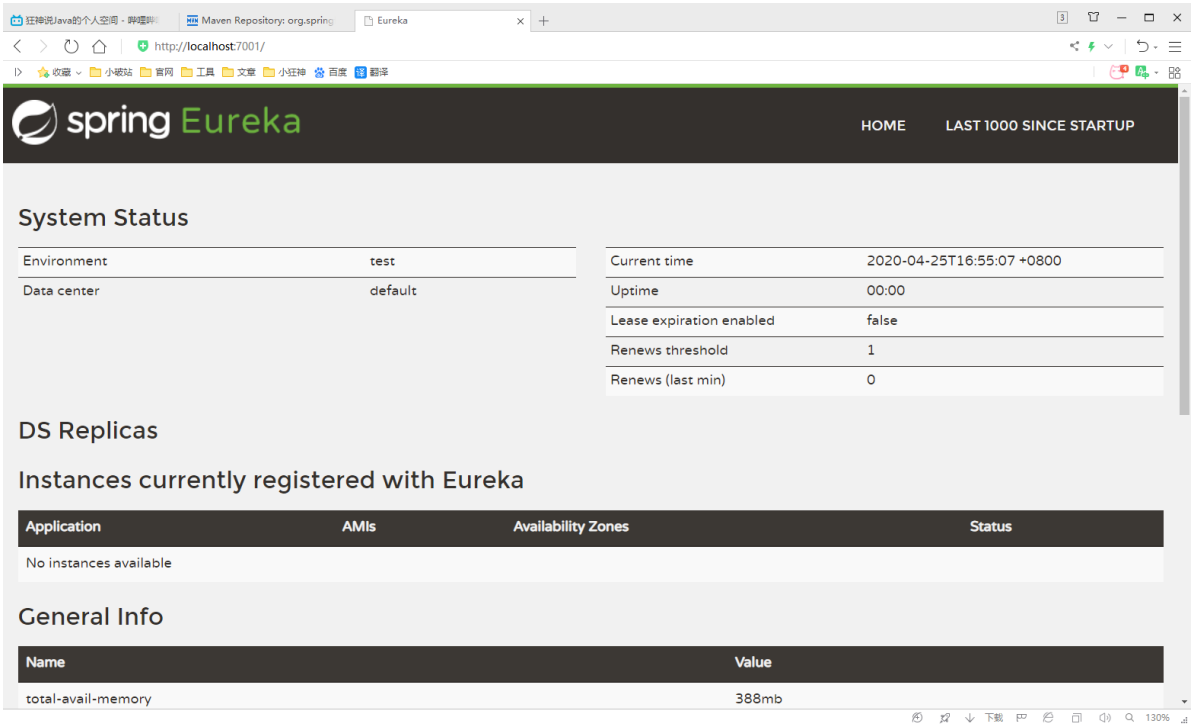
application.yml

```yaml
server:
  port: 7001

#Eureka配置
eureka:
  instance:
    hostname: localhost #eureka服务端的实例名称
  client:
    register-with-eureka: false # 是否将自己注册到Eureka服务器中，本身是服务器，无需注册
    fetch-registry: 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
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
      # 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defaultZone地址
```

编写主启动类

```java
package com.kuang.springcloud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer //EurekaServer服务器端启动类，接受其他微服务注册进来！
public class EurekaServer7001 {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServer7001.class,args);
    }

}
```

启动，访问测试：



System Status：系统信息

DS Replicas：服务器副本

Instances currently registered with Eureka：已注册的微服务列表

General Info：一般信息

Instance Info：实例信息

> Service Provider

将 8001 的服务入驻到 7001 的eureka中!

1、修改8001服务的pom文件，增加eureka的支持!

```
<!--将服务的provider注册到eureka中-->
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-eureka -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

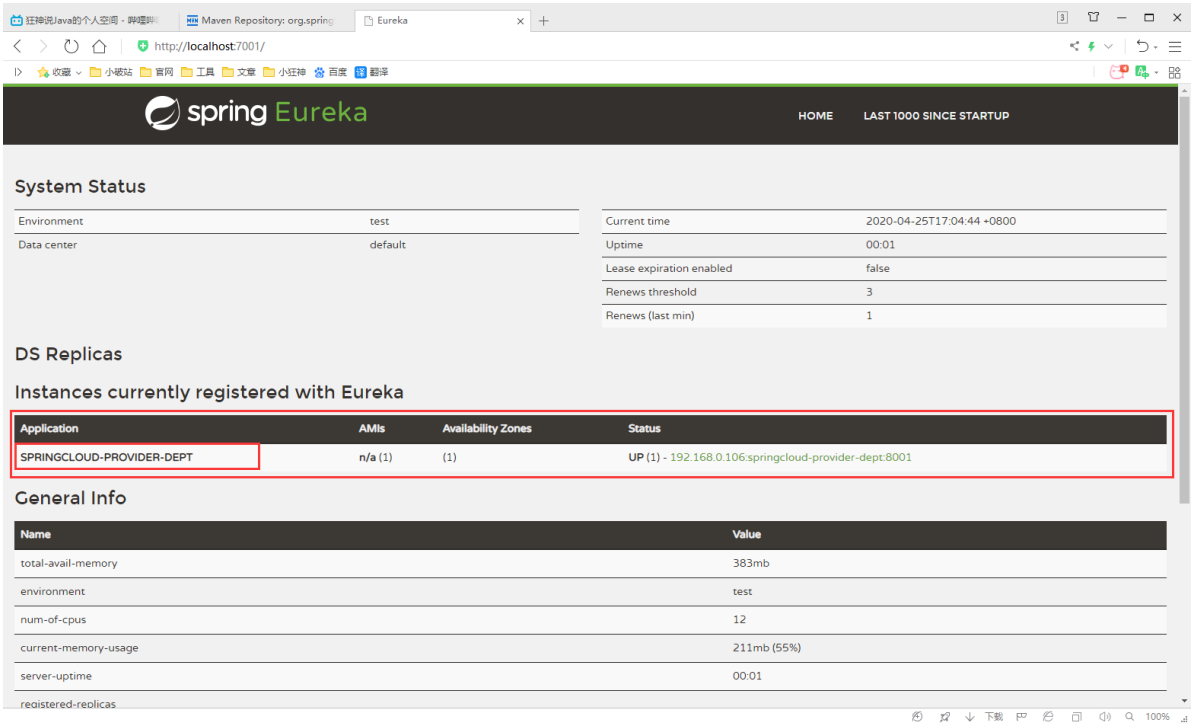
2、yaml 中配置 eureka 的支持

```
#eureka配置
eureka:
  client:
    service-url:
     defaultZone: http://localhost:7001/eureka
```

3、8001 的主启动类注解支持

```
@SpringBootApplication
@EnableEurekaClient // 本服务启动之后会自动注册进Eureka中!
public class DeptProvider8001 {

    public static void main(String[] args) {
        SpringApplication.run(DeptProvider8001.class,args);
    }

}
```

截止目前：服务端也有了，客户端也有了，启动7001，再启动8001，测试访问



> actuator与注册微服务信息完善

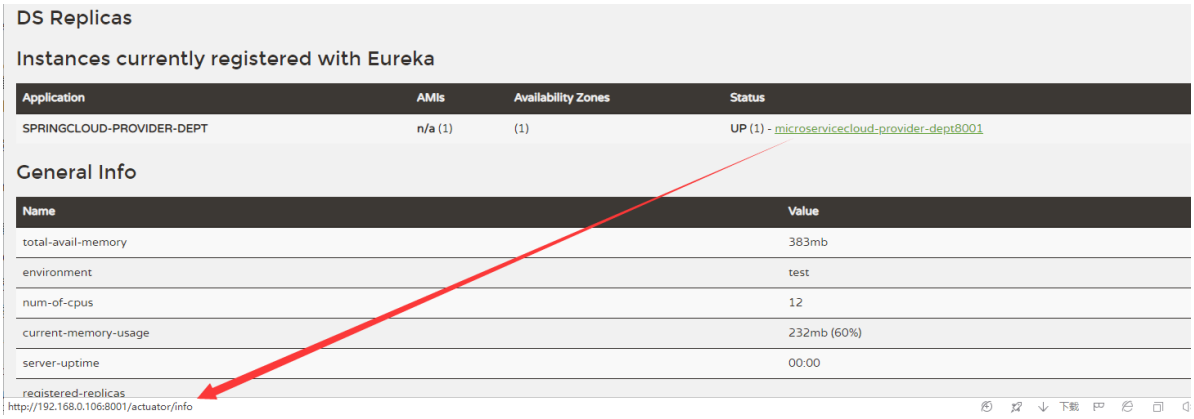
主机名称：服务名称修改

在8001的yaml中修改一下配置

```yaml
# eureka配置
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka
  instance:
    instance-id: springcloud-provider-dept8001 # 重点，和client平级
```

重启，刷新后查看结果！

访问信息有IP信息提示

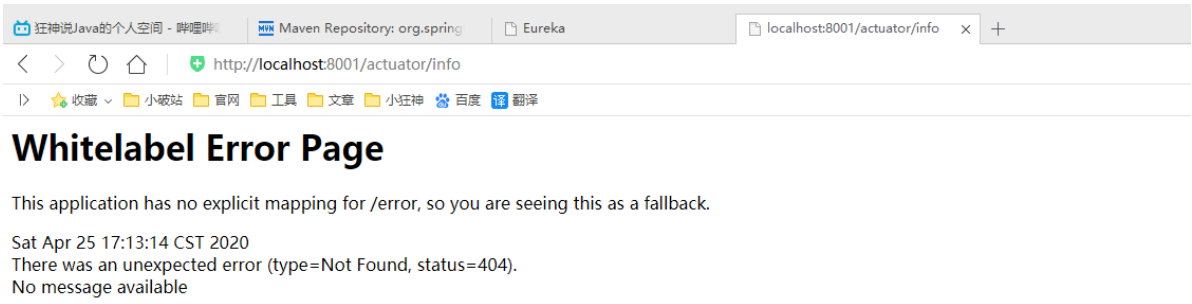


yaml中在增加一个配置

```yaml
#eureka配置
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka
  instance:
    instance-id: springcloud-provider-dept8001
    prefer-ip-address: true  # true访问路径可以显示IP地址
```

info内容构建

现在点击info，出现ERROR页面



修改8001的pom文件，新增依赖！

```xml
<!--actuator监控信息完善-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

在**总的父工程**中修改pom.xml添加build信息 【没必要做】

```xml
<build>
    <finalName>springcloud</finalName>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
    <plugins>
```

```
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-resources-plugin</artifactId>
            <configuration>
                <delimiters>
                    <!--以$开头的或者结尾的在src/main/resources路径下的就能读取-->
                    <delimit>$</delimit>
                </delimiters>
            </configuration>
        </plugin>
    </plugins>
</build>
```

然后回到我们的8001的yaml配置文件中修改增加信息

```
#info配置
info:
  app.name: kuang-springcloud
  company.name: www.kuangstudy.com
  build.artifactId: ${project.artifactId}
  build.version: ${project.version}
```

重启项目测试：7001、8001

> Eureka的自我保护机制

之前出现的这些红色情况，没出现的，修改一个服务名，故意制造错误！

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.

DS Replicas

自我保护机制：好死不如赖活着

一句话总结：某时刻某一个微服务不可以用了，eureka不会立刻清理，依旧会对该微服务的信息进行保存！

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时，微服务与Eureka之间无法正常通行，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服务**。Eureka通过 **自我保护机制** 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。一旦进入该模式，EurekaServer就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该EurekaServer节点会自动退出自我保护模式。

在自我保护模式中，EurekaServer会保护服务注册表中的信息，不再注销任何服务实例。当它收到的心跳数重新恢复到阈值以上时，该EurekaServer节点就会自动退出自我保护模式。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：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集群更加的健壮和稳定。

在SpringCloud中，可以使用`eureka.server.enable-self-preservation = false`禁用自我保护模式【不推荐关闭自我保护机制】

> 8001服务发现 Discovery

对于注册进eureka里面的微服务，可以通过服务发现来获得该服务的信息。【对外暴露服务】

修改springcloud-provider-dept-8001工程中的DeptController

新增一个方法

```
@GetMapping("/discovery")
public Object discovery(){
    //获得微服务列表清单
    List<String> list = client.getServices();
    System.out.println("client.getServices()==>"+list);

    //得到一个具体的微服务!
    List<ServiceInstance> serviceInstanceList =
client.getInstances("springcloud-provider-dept");

    for (ServiceInstance serviceInstance : serviceInstanceList) {
        System.out.println(
            serviceInstance.getServiceId()+"\t"+
            serviceInstance.getHost()+"\t"+
            serviceInstance.getPort()+"\t"+
            serviceInstance.getUri()
        );
    }
    return this.client;
}
```

主启动类增加一个注解

```java
@SpringBootApplication
@EnableEurekaClient // 本服务启动之后会自动注册进Eureka中!
@EnableDiscoveryClient //服务发现
public class DeptProvider8001 {

    public static void main(String[] args) {
        SpringApplication.run(DeptProvider8001.class,args);
    }

}
```

启动Eureka服务，启动8001提供者

访问测试 http://localhost:8001/dept/discovery

后台输出：

consumer访问服务

springcloud-consumer-dept-80

修改DeptConsumerController增加一个方法

```java
@GetMapping("/consumer/dept/discovery")
public Object discovery(){
    return
restTemplate.getForObject(REST_URL_PREFIX+"/dept/discovery",Object.class);
}
```

启动 80 项目进行测试!

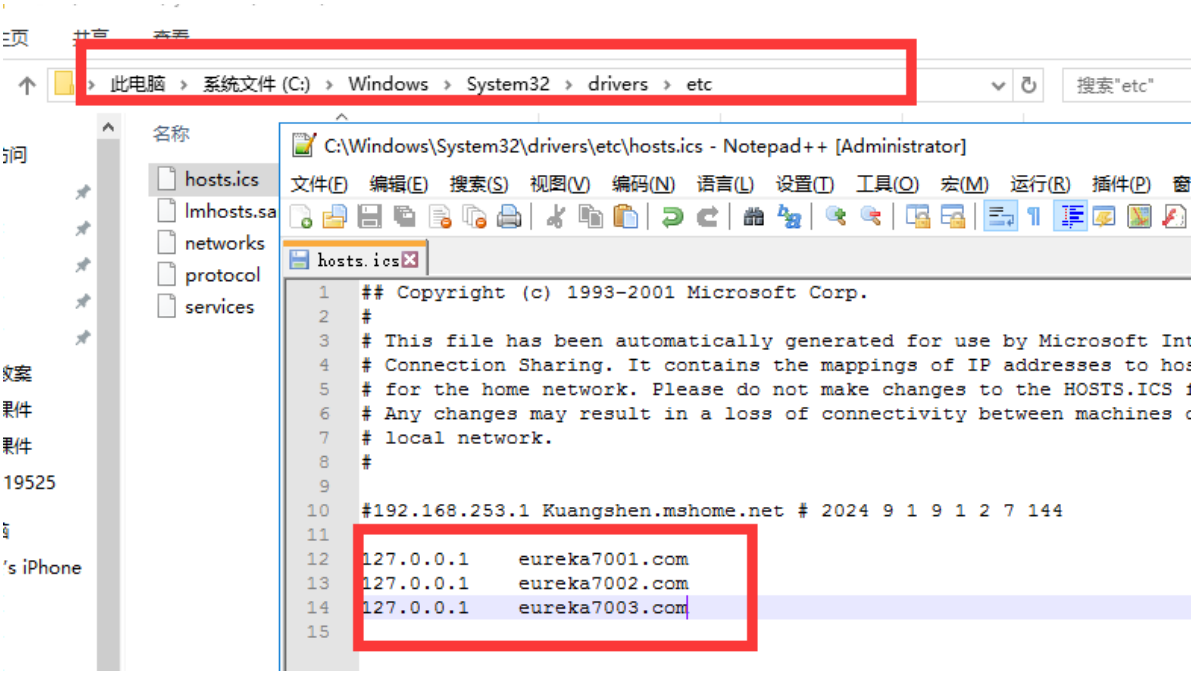
集群配置

新建工程springcloud-eureka-7002、springcloud-eureka-7003

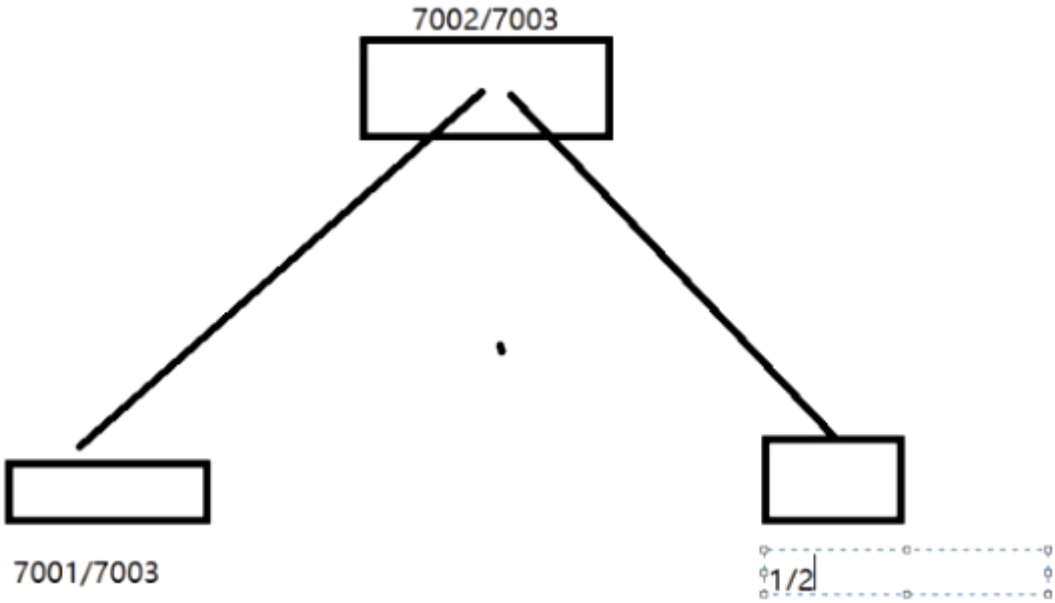
按照7001为模板粘贴POM

修改7002和7003的主启动类

修改映射配置，**windows域名映射**



集群配置分析



修改3个EurekaServer的yaml文件夹

7001：

```yaml
server:
  port: 7001

#Eureka配置
eureka:
  instance:
    hostname: eureka7001.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: 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
    service-url:
      # 单机 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
      # 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defaultZone地址
```

```
      defaultZone:
http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

7002:

```yaml
server:
  port: 7002

#Eureka配置
eureka:
  instance:
    hostname: eureka7002.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: 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
    service-url:
      # 单机 defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/
      # 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defaultZone地址
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7003.com:7003/eureka/
```

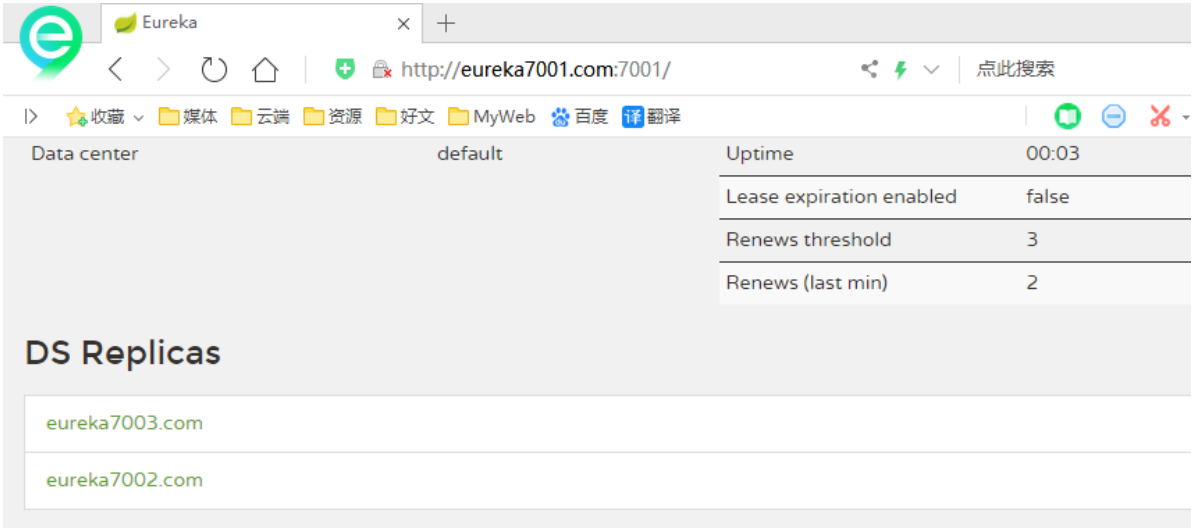
7003:

```yaml
server:
  port: 7003

#Eureka配置
eureka:
  instance:
    hostname: eureka7003.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: 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
    service-url:
      # 单机 defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/
      # 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defaultZone地址
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/
```

将8001微服务发布到1台eureka集群配置中，发现在集群中的其余注册中心也可以看到，但是平时我们保险起见，都发布！

```yaml
#eureka配置
eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
  instance:
    instance-id: microservicecloud-provider-dept8001
    prefer-ip-address: false  #true访问路径可以显示IP地址
```

启动集群测试！



> 对比Zookeeper

回顾CAP原则

RDBMS （Mysql、Oracle、sqlServer）===>ACID

NoSQL（redis、mongdb）===> CAP

ACID是什么?

- A（Atomicity）原子性
- C（Consistency） 一致性
- I （Isolation）隔离性
- D（Durability）持久性

CAP是什么?

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

CAP的三进二：CA、AP、CP

CAP理论的核心

- 一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求
- 根据CAP原理，将NoSQL数据库分成了满足CA原则，满足CP原则和满足AP原则三大类：
  - CA：单点集群，满足一致性，可用性的系统，通常可扩展性较差
  - CP：满足一致性，分区容错性的系统，通常性能不是特别高
  - AP：满足可用性，分区容错性的系统，通常可能对一致性要求低一些

> 作为服务注册中心，Eureka比Zookeeper好在哪里?

著名的CAP理论指出，一个分布式系统不可能同时满足C（一致性）、A（可用性）、P（容错性）。

由于分区容错性P在分布式系统中是必须要保证的，因此我们只能在A和C之间进行权衡。

- Zookeeper保证的是CP;
- Eureka保证的是AP;

Zookeeper保证的是CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30~120s，且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因为网络问题使得zk集群失去master节点是较大概率会发生的事件，虽然服务最终能够恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

Eureka保证的是AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。**Eureka各个节点都是平等的**，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时，如果发现连接失败，则会自动切换至其他节点，只要有一台Eureka还在，就能保住注册服务的可用性，只不过查到的信息可能不是最新的，除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点依然可用）
3. 当网络稳定时，当前实例新的注册信息会被同步到其他节点中

因此，Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪

Ribbon负载均衡

鸡汤：

理论 和 实践 是同样重要的！
因为在你们去面试的时候，需要有谈资！！！
你需要先进去，才能够拥有干活的机会，所以要耐着性子学习！

> 概述

Ribbon是什么?

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

简单的说，Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将NetFlix的中间层服务连接在一起。Ribbon的客户端组件提供一系列完整的配置项如：连接超时、重试等等。简单的说，就是在配置文件中列出LoadBalancer（简称LB：负载均衡）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，随机连接等等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法！

Ribbon能干嘛?

LB，即负载均衡（Load Balance），在微服务或分布式集群中经常用的一种应用。

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

常见的负载均衡软件有 Nginx，Lvs 等等

Dubbo、SpringCloud中均给我们提供了负载均衡，**SpringCloud的负载均衡算法可以自定义**

负载均衡简单分类：

- 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
  - 如之前学习的Nginx，由该设施负责把访问请求通过某种策略转发至服务的提供方！
- 进程式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！

Ribbon的github地址： https://github.com/NetFlix/ribbon

> Ribbon配置初步

复制springcloud-consumer-dept-80工程 springcloud-consumer-dept-ribbon-80

1、主启动类名字修改下，用作区分

```
@SpringBootApplication
public class DeptConsumerRibbon80 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumerRibbon80.class,args);
    }
}
```

2、修改pom.xml，添加Ribbon相关的配置

```
<!--Ribbon相关-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-ribbon -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

3、修改application.yml， 追加Eureka的服务注册地址

```yaml
server:
  port: 80

#Eureka配置
eureka:
  client:
    register-with-eureka: false  #false表示不向注册中心注册自己
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://
eureka7003.com:7003/eureka/
```

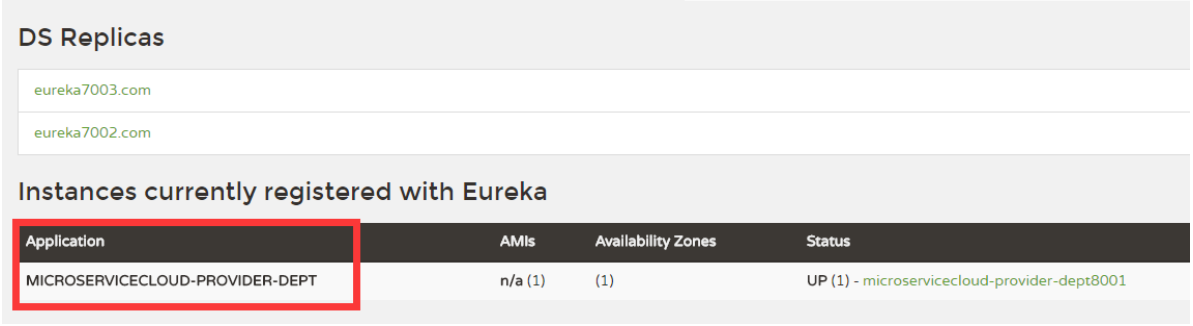
4、对里面的ConfigBean方法加上注解@LoadBalanced在 获得Rest时加入Ribbon的配置；

```
@Bean
@LoadBalanced //Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端负载均衡的工具
public RestTemplate getRestTemplate(){
    return new RestTemplate();
}
```

5、主启动类DeptConsumerRibbon80添加@EnableEurekaClient

```
@EnableEurekaClient
@SpringBootApplication
public class DeptConsumerRibbon80 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumerRibbon80.class,args);
    }
}
```

6、修改DeptConsumerController客户端访问类，之前的写的地址是写死的，现在需要变化！




```
//做Ribbon时候将此改为了微服务名字
//private static final String REST_URL_PREFIX = "http://localhost:8001";
private static final String REST_URL_PREFIX = "http://SPRINGCLOUD-PROVIDER-DEPT";
```

7、先启动3个Eureka集群后，在启动springcloud-provider-dept-8001并注册进eureka

8、启动 DeptConsumerRibbon80

9、测试

http://localhost/consumer/dept/get/1

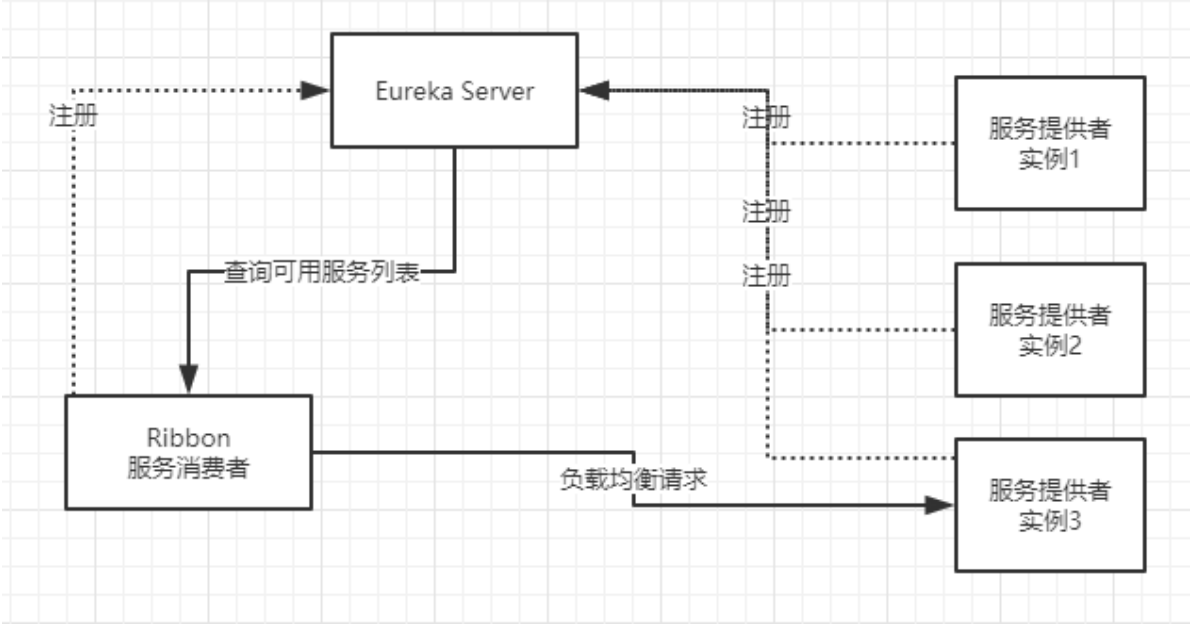
http://localhost/consumer/dept/list

10、小结

Ribbon和Eureka整合后Consumer可以直接调用服务而不用再关心地址和端口号！

> Ribbon负载均衡

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均衡较少的Server。

第二步在根据用户指定的策略，在从server去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**（默认），随机和根据响应时间加权重,,,等等

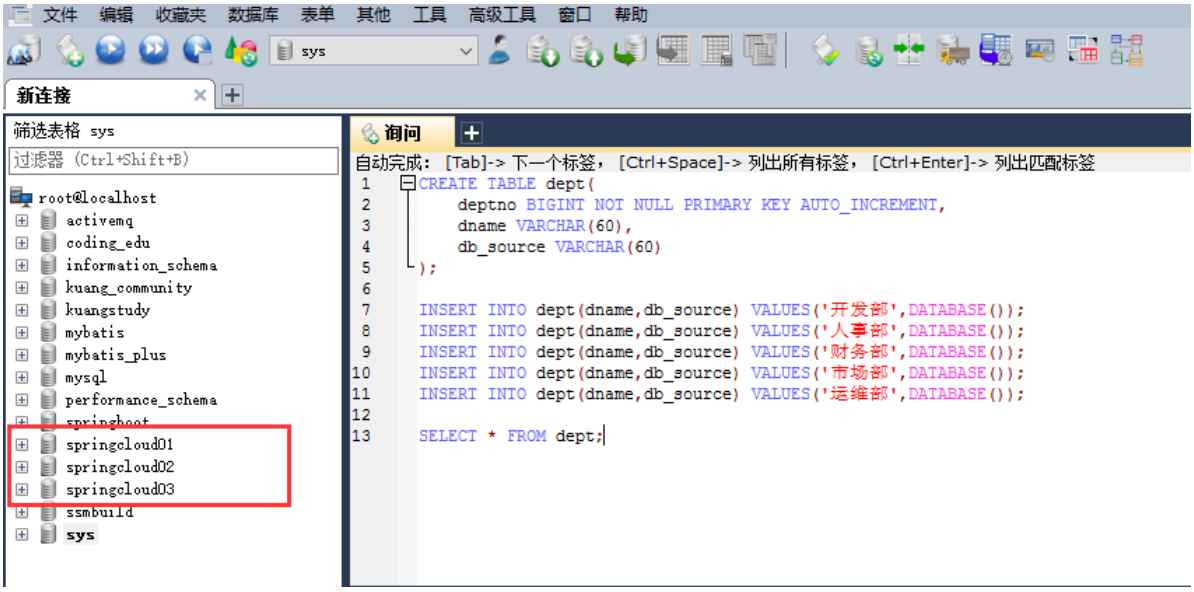
测试：

参考springcloud-provider-dept-8001，新建两份，分别为8002,8003！

全部复制完毕，修改启动类名称，修改端口号名称！

新建8002/8003数据库，各自微服务分别连接各自的数据库，复制DB1！

- 新建springcloud01
- 新建springcloud02
- 新建springcloud03



修改8002/8003各自的YML文件

- 端口
- 数据库连接
- 实例名也需要修改

```yaml
instance:
  instance-id: springcloud-provider-dept8003
```

- 对外暴露的统一的服务实例名【三个服务名字必须一致！】

```yaml
application:
  name: springcloud-provider-dept
```

启动3个Eureka集群配置区

启动3个Dept微服务并都测试通过

http://localhost:8001/dept/list

http://localhost:8002/dept/list

http://localhost:8003/dept/list

启动springcloud-consumer-dept-ribbon-80

客户端通过Ribbon完成负载均衡并访问上一步的Dept微服务

http://localhost/consumer/dept/list

多刷新几次注意观察结果！

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。

> Ribbon核心组件IRule

IRule：根据特定算法从服务列表中选取一个要访问的服务！

- RoundRobinRule【轮询】
- RandomRule【随机】
- AvailabilityFilterRule【会先过滤掉由于多次访问故障而处于断路器跳闸的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表按照轮询策略进行访问】
- WeightedResponseTimeRule【根据平均响应时间计算所有服务的权重，响应时间越快服务权重越大，被选中的概率越高，刚启动时如果统计信息不足，则使用RoundRobinRule策略，等待统计信息足够，会切换到WeightedResponseTimeRule】
- 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】
- 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】
- 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】

查看分析源码：

1. IRule
2. ILoadBalancer
3. AbstractLoadBalancer
4. AbstractLoadBalancerRule：这个抽象父类十分重要！核心
5. RoundRobinRule

分析一下方法

```
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);
        //每一次得到下一个ServerIndex，也就是所谓的轮询
        server = allServers.get(nextServerIndex);

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

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

```
33
34          // Next.
35          server = null;
36      }
37
38      if (count >= 10) {
39          log.warn("No available alive servers after 10 tries from load
 balancer: "
40                  + lb);
41      }
42      return server;
43  }
```

切换为随机策略实现试试，在ConfigBean中添加方法

```
1  @Bean
2  public IRule myRule(){
3      //使用我们重新选择的随机算法，替代默认的轮询!
4      return new RandomRule();
5  }
```

重启80服务进行访问测试，查看运行结果！【注意，可能服务长时间不使用会崩】

http://localhost/consumer/dept/list

测试：new RetryRule() 算法

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】

1、在运行期间关闭掉一个服务提供者8002

2、消费者再次测试！发现404后继续访问测试！看结果！！！

其余的不再挨个测试了，大家有时间可以去测试玩玩；

现在有一个新的需求，我们不需要这些默认的算法，我们需要自己重新定义，这该怎么办呢?

> 自定义Ribbon

修改springcloud-consumer-dept-ribbon-80

主启动类添加@RibbonClient注解

在启动该微服务的时候就能去加载我们自定义的Ribbon配置类，从而使配置类生效，例如:

```
1  @RibbonClient(name="SPRINGCLOUD-PROVIDER-
   DEPT",configuration=MySelfRule.class)
```

注意配置细节

官方文档明确给出了警告：

这个自定义配置类不能放在@ComponentScan所扫描的当前包以及子包下，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，也就是说达不到特殊化定制的目的了！

6.2 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 {
}
```

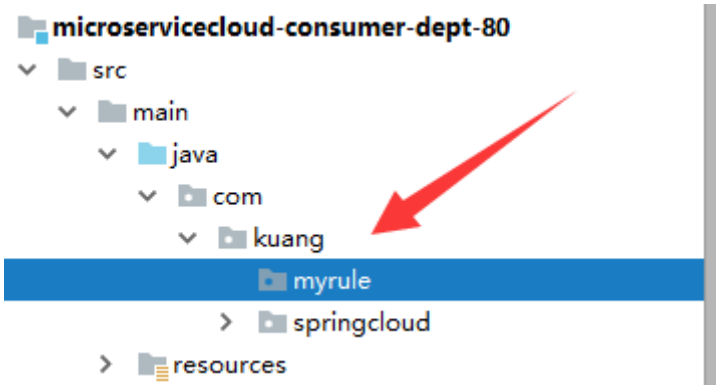
In this case, the client is composed from the components already in `RibbonClientConfiguration`, together with any in `CustomConfiguration` (where the latter generally overrides the former).

The `CustomConfiguration` clas 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`).

The following table shows the beans that Spring Cloud Netflix provides by default for Ribbon:

步骤

1. 由于有以上配置细节原因，我们建立一个包 com.kuang.myrule



2. 在这里新建一个自定义规则的Rubbion类

```
@Configuration
public class MySelfRule {

    @Bean
    public IRule myRule(){
        return new RandomRule(); //Ribbon默认是轮询，我们自定义为随机算法
    }

}
```

3. 在主启动类上配置我们自定义的Ribbon

```java
@SpringBootApplication
@EnableEurekaClient
//核心!
@RibbonClient(name="SPRINGCLOUD-PROVIDER-DEPT",configuration=MySelfRule.class)
public class DeptConsumer80_App {

    public static void main(String[] args) {
        SpringApplication.run(DeptConsumer80_App.class,args);
    }

}

```

4. 启动所有项目，访问测试，看看我们编写的随机算法，现在是否生效!

自定义规则深度解析

1、问题：依旧轮询策略，但是加上新需求，每个服务器要求被调用5次，就是以前每一个机器一次，现在每个机器5次；

2、解析源码：RandomRule.java ， IDEA直接点击进去，复制出来，变成我们自己的类 KuangRondomRule

仔细分析阅读源码

```java
public class KuangRondomRule extends AbstractLoadBalancerRule {

    @edu.umd.cs.findbugs.annotations.SuppressWarnings(value = "RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE")
    //ILoadBalancer选择的随机算法
    public Server choose(ILoadBalancer lb, Object key) {
        if (lb == null) {
            return null;
        }
        Server server = null;

        while (server == null) {
            //查看线程是否中断了
            if (Thread.interrupted()) {
                return null;
            }

            //Reachable:  可及；可到达；够得到
            List<Server> upList = lb.getReachableServers(); //活着的服务
            List<Server> allList = lb.getAllServers(); //获取所有的服务

            int serverCount = allList.size();
            if (serverCount == 0) {
                return null;
```

```
            }

            int index = chooseRandomInt(serverCount); //生成区间随机数!
            server = upList.get(index); //从活着的服务中，随机取出一个

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

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

            server = null;
            Thread.yield();
        }

        return server;

    }

    //随机
    protected int chooseRandomInt(int serverCount) {
        return ThreadLocalRandom.current().nextInt(serverCount);
    }

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

    @Override
    public void initWithNiwsConfig(IClientConfig clientConfig) {

    }
}
```

参考源码修改为我们需求要求的KuangRondomRule.java

```
public class KuangRondomRule extends AbstractLoadBalancerRule {



    // total = 0    当total数等于5以后，我们指针才能往下走
    // index = 0    当前对外提供服务的服务器地址
    // 如果total等于5，则index+1，将total重置为0即可!
    // 问题：我们只有3台机器，所有total>3 则将total置为0;

    private int total = 0;           //总共被调用的次数
    private int currentIndex = 0;    //当前提供服务的机器序号!

    //ILoadBalancer选择的随机算法
    public Server choose(ILoadBalancer lb, Object key) {
        if (lb == null) {
            return null;
        }
        Server server = null;

```

```
        while (server == null) {
            //查看线程是否中断了
            if (Thread.interrupted()) {
                return null;
            }

            //Reachable:  可及；可到达；够得到
            List<Server> upList = lb.getReachableServers(); //活着的服务
            List<Server> allList = lb.getAllServers(); //获取所有的服务

           int serverCount = allList.size();
            if (serverCount == 0) {
                return null;
            }

//              int index = chooseRandomInt(serverCount); //生成区间随机数!
//              server = upList.get(index); //从活着的服务中，随机取出一个

            //+====================================
            if (total<5){
                server = upList.get(currentIndex);
                total++;
            }else {
                total = 0;
                currentIndex++;
                if (currentIndex>=upList.size()){
                    currentIndex = 0;
                }
                server = upList.get(currentIndex);
            }
            //+====================================

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

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

            server = null;
            Thread.yield();
        }

        return server;

    }

    //随机
    protected int chooseRandomInt(int serverCount) {
        return ThreadLocalRandom.current().nextInt(serverCount);
    }

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

```
77
78      @Override
79      public void initWithNiwsConfig(IClientConfig clientConfig) {
80
81      }
82  }
83
```

调用，在我们自定义的IRule方法中返回刚才我们写好的随机算法类

```
1  @Configuration
2  public class MySelfRule {
3
4      @Bean
5      public IRule myRule(){
6          return new KuangRondomRule(); //Ribbon默认是轮询，我们自定义为
   KuangRondomRule
7      }
8
9  }
```

测试

Feign负载均衡

> 简介

feign是声明式的web service客户端，它让微服务之间的调用变得更简单了，类似controller调用service。

Spring Cloud集成了Ribbon和Eureka，可在使用Feign时提供负载均衡的http客户端。

只需要创建一个接口，然后添加注解即可！

feign ，主要是社区，大家都习惯面向接口编程。这个是很多开发人员的规范。调用微服务访问两种方法

1. 微服务名字 【ribbon】
2. 接口和注解 【feign 】

Feign能干什么?

- Feign旨在使编写Java Http客户端变得更容易
- 前面在使用Ribbon + RestTemplate时，利用RestTemplate对Http请求的封装处理，形成了一套模板化的调用方法。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的调用。所以，Feign在此基础上做了进一步封装，由他 来帮助我们定义和实现依赖服务接口的定义，在Feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它（类似于以前Dao接口上标注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springcloud-Dept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，优雅而且简单的实现了服务调用。

> Feign使用步骤

1、参考springcloud-consumer-dept-ribbon-80

2、新建springcloud-consumer-dept-feign-80

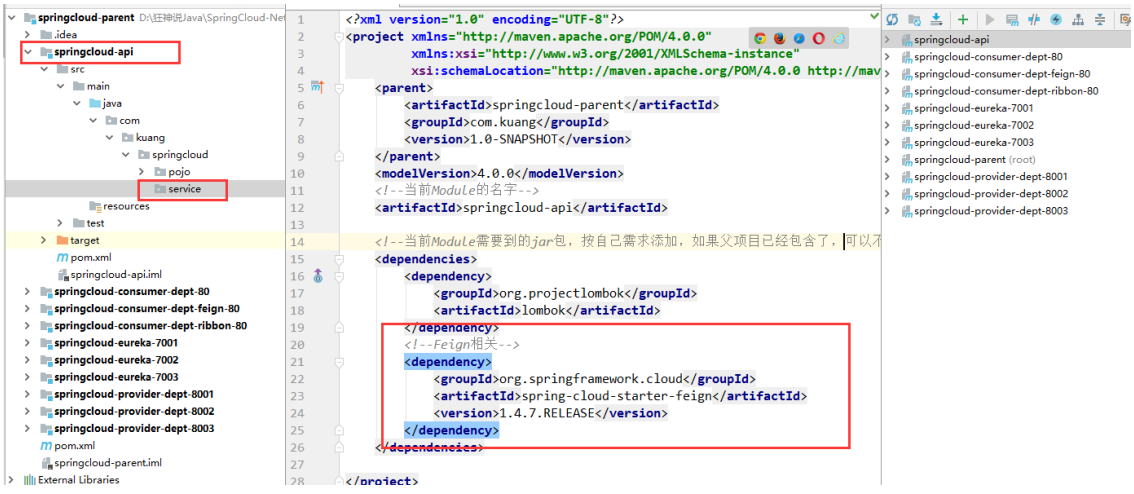
- 修改主启动类名称
- 将springcloud-consumer-dept-80的内容都拷贝到 feign项目中
- 删除myrule文件夹
- 修改主启动类的名称为 DeptConsumerFeign80

3、springcloud-consumer-dept-feign-80修改pom.xml， 添加对Feign的支持。

```
<!--Feign相关-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
```

4、修改springcloud-api工程

- pom.xml添加feign的支持
- 新建一个Service包



- 编写接口 DeptClientService，并增加新的注解@FeignClient。

```java
@FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT")
public interface DeptClientService {

    @GetMapping("/dept/get/{id}")
    public Dept queryById(@PathVariable("id") Long id); //根据id查询部门

    @GetMapping("/dept/list")
    public List<Dept> queryAll(); //查询所有部门

    @PostMapping(value = "/dept/add")
    public boolean addDept(Dept dept); //添加一个部门

}
```

- mvn清理一下

5、springcloud-consumer-dept-feign-80工程修改Controller，添加上一步新建的DeptClientService

```java
@RestController
public class DeptConsumerController {

    @Autowired
    private DeptClientService service = null;

    @RequestMapping("/consumer/dept/add")
    public boolean add(Dept dept){
        return this.service.addDept(dept);
    }

    @RequestMapping("/consumer/dept/get/{id}")
    public Dept get(@PathVariable("id") Long id){
         return this.service.queryById(id);
    }

    @RequestMapping("/consumer/dept/list")
    public List<Dept> list(){
        return this.service.queryAll();
    }

}
```

6、microservicecloud-consumer-dept-feign工程修改主启动类，开启Feign使用!

```java
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients(basePackages = {"com.kuang.springcloud"})
@ComponentScan("com.kuang.springcloud")
public class DeptConsumer80_Feign_App {

    public static void main(String[] args) {
        SpringApplication.run(DeptConsumer80_Feign_App.class,args);
    }

}
```

7、测试

- 启动eureka集群

- 启动8001，8002，8003
- 启动feign客户端
- 测试： http://localhost/consumer/dept/list
- 结论：Feign自带负载均衡配置项

> 小结

Feign通过接口的方法调用Rest服务 (之前是Ribbon+RestTemplate)

该请求发送给Eureka服务器 （http://MICROSERVICECLOUD-PROVIDER-DEPT/dept/list）

通过Feign直接找到服务接口，由于在进行服务调用的时候融合了Ribbon技术，所以也支持负载均衡作用！

feign其实不是做负载均衡的,负载均衡是ribbon的功能,feign只是集成了ribbon而已,但是负载均衡的功能还是feign内置的ribbon再做,而不是feign。
feign的作用的替代RestTemplate,性能比较低，但是可以使代码可读性很强。

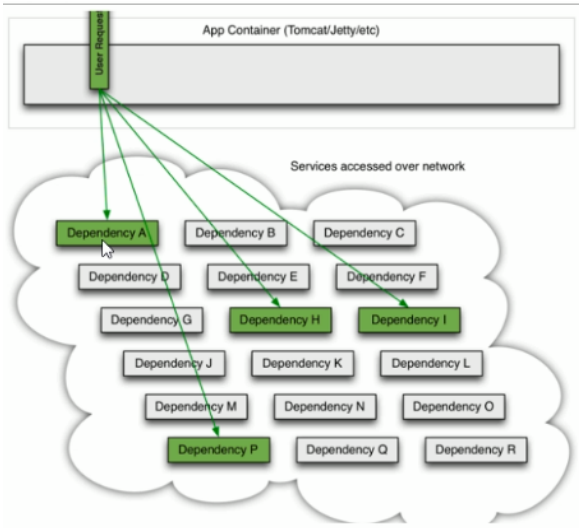
Hystrix断路器

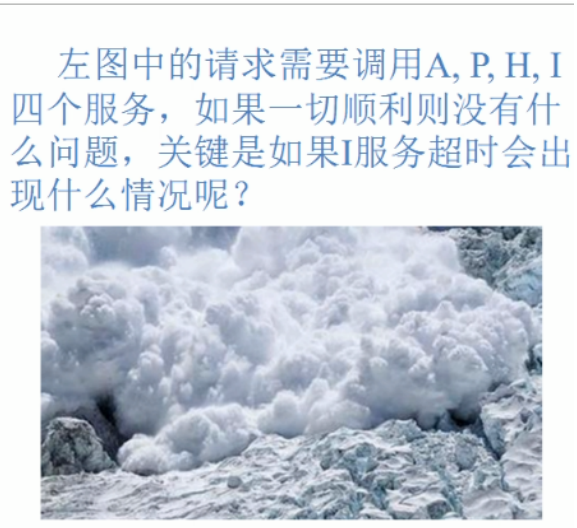
Hystrix怎么读?

> 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的失败！





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中内饱和。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。

我们需要 ·弃车保帅·

什么是Hystrix

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），**向调用方返回一个服务预期的，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法无法处理的异常，这样就可以保证了服务调用方的线程不会被长时间**，不必要的占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩.

能干嘛

- 服务降级
- 服务熔断
- 服务限流
- 接近实时的监控
- .....

官网资料

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

> 服务熔断

是什么

熔断机制是对应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况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。

熔断机制的注解是 @HystrixCommand。

参考springcloud-provider-dept-8001

- 新建springcloud-provider-dept-hystrix-8001
- 将之前8001的所有东西拷贝一份

修改pom

添加Hystrix的依赖

```
<!--Hystrix-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
```

修改yml

修改eureka实例的id

```
eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka
instance:
    instance-id: microservicecloud-provider-dept8001-hystrix #自定义服务名称
    prefer-ip-address: false   #true访问路径可以显示IP地址

#info配置
info:
  app.name: kuang-microservicecloud
```

修改DeptController

1、@HystrixCommand报异常后如何处理

```
//一旦调用服务方法失败并抛出了错误信息后
// 会自动调用HystrixCommand标注好的fallbackMethod调用类中指定方法
@HystrixCommand(fallbackMethod = "processHystrix_Get")
```

2、代码内容

```
@RestController
public class DeptController {

    @Autowired
    private DeptService service;

    //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调用类中指定方法
    @GetMapping("/dept/get/{id}")
    @HystrixCommand(fallbackMethod = "processHystrix_Get")
    public Dept get(@PathVariable("id") Long id) {
        Dept dept = service.queryById(id);
        if (dept==null){
            throw new RuntimeException("该id：“+id+"没有对应的的信息");
        }
        return dept;
    }

    public Dept processHystrix_Get(@PathVariable("id") Long id){
        return new Dept().setDeptno(id)
                .setDname("该id：“+id+"没有对应的信息，null--@HystrixCommand")
                .setDb_source("no this database in MySQL");
```

```
    }

}

```

修改主启动类添加新注解 @EnableCircuitBreaker

3、修改主启动类的名称为 DeptProviderHystrix8001

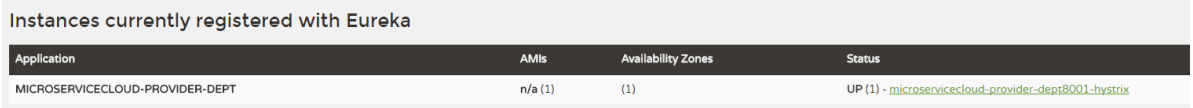
4、代码

```
@SpringBootApplication
@EnableEurekaClient //本服务启动之后会自动注册进Eureka中!
@EnableDiscoveryClient //服务发现
@EnableCircuitBreaker //对hystrix 熔断机制的支持 【==========new=======】
public class DeptProviderHystrix8001 {

    public static void main(String[] args) {
        SpringApplication.run(DeptProviderHystrix8001.class,args);
    }

}
```

测试

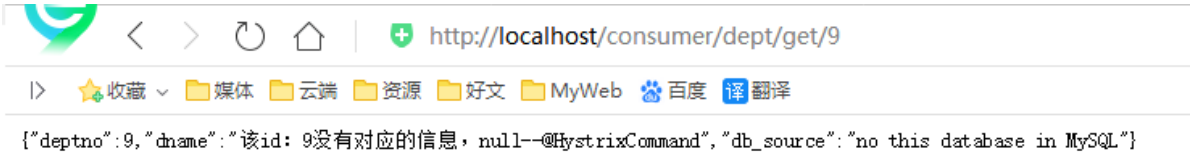
1、启动Eureka集群

2、启动主启动类 DeptProviderHystrix8001



3、启动客户端 springcloud-consumer-dept-80

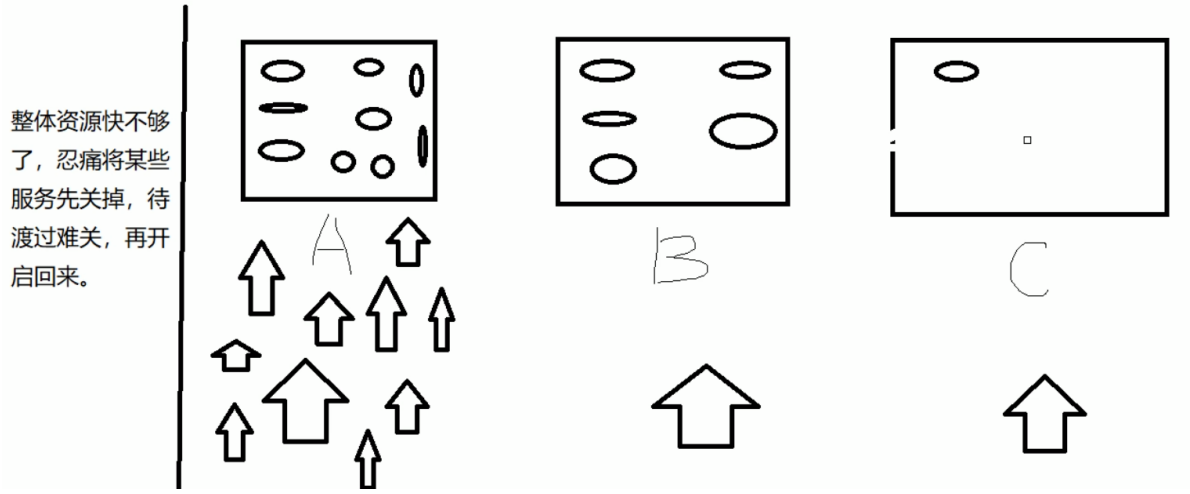
4、访问 http://localhost/consumer/dept/get/111



> 服务降级

是什么

整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。



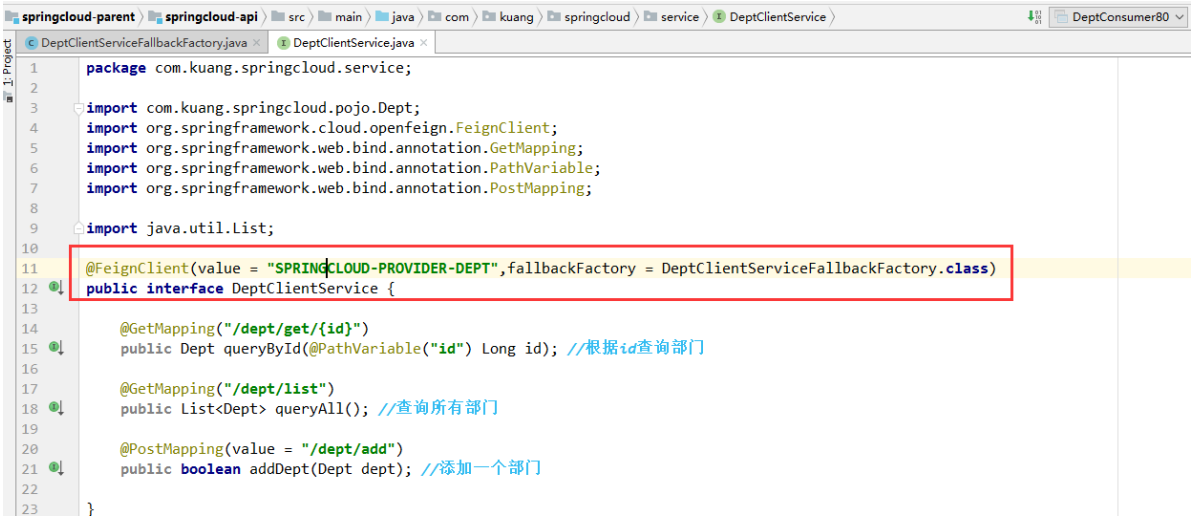
服务降级处理是在客户端实现完成的，与服务端没有关系

修改springcloud-api工程，根据已经有的DeptClientService接口新建一个实现了FallbackFactory接口的类DeptClientServiceFallbackFactory

【注意：这个类上需要@Component注解！！！】

```
@Component //千万不要忘记
public class DeptClientServiceFallbackFactory implements
FallbackFactory<DeptClientService> {

    @Override
    public DeptClientService create(Throwable throwable) {
        return new DeptClientService() {
            @Override
            public Dept queryById(Long id) {
                return new Dept().setDeptno(id)
                        .setDname("该id：“+id+"没有对应的信息，Consumer客户端提供的降级信息，此刻服务Provider已经关闭")
                        .setDb_source("no this database in MySQL");
            }

            @Override
            public List<Dept> queryAll() {
                return null;
            }

            @Override
            public boolean addDept(Dept dept) {
                return false;
            }
        };
    }

}
```

修改springcloud-api工程，DeptClientService接口在注解 @FeignClient中添加fallbackFactory属性值



```java
package com.kuang.springcloud.service;

import com.kuang.springcloud.pojo.Dept;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;

import java.util.List;

@FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT",fallbackFactory = DeptClientServiceFallbackFactory.class)
public interface DeptClientService {

    @GetMapping("/dept/get/{id}")
    public Dept queryById(@PathVariable("id") Long id); //根据id查询部门

    @GetMapping("/dept/list")
    public List<Dept> queryAll(); //查询所有部门

    @PostMapping(value = "/dept/add")
    public boolean addDept(Dept dept); //添加一个部门

}
```

```java
@FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT",fallbackFactory =
DeptClientServiceFallbackFactory.class)
public interface DeptClientService {

    @GetMapping("/dept/get/{id}")
    public Dept queryById(@PathVariable("id") Long id); //根据id查询部门

    @GetMapping("/dept/list")
    public List<Dept> queryAll(); //查询所有部门

    @PostMapping(value = "/dept/add")
    public boolean addDept(Dept dept); //添加一个部门

}
```

springcloud-api工程 mvn clean install

springcloud-consumer-dept-feign-80工程修改YML

```yaml
server:
  port: 80


feign:
  hystrix:
    enabled: true

#Eureka配置
eureka:
  client:
    register-with-eureka: false  #false表示不向注册中心注册自己
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com
```

```yaml
server:
  port: 80

feign:
  hystrix:
    enabled: true

#Eureka配置
eureka:
  client:
    register-with-eureka: false  #false表示不向注册中心注册自己
```

```
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http:/
/eureka7003.com:7003/eureka/

```

测试

1. 启动eureka集群
2. 启动 springcloud-provider-dept-hystrix-8001
3. 启动 springcloud-consumer-dept-feign-80
4. 正常访问测试
    - http://localhost/consumer/dept/get/1
5. 故意关闭微服务启动 springcloud-provider-dept-hystrix-8001
6. 客户端自己调用提示
    - http://localhost/consumer/dept/get/1

    - 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。

> 小结

服务熔断：一般是某个服务故障或者异常引起，类似现实世界中的 “保险丝”， 当某个异常条件被触发，直接熔断整个服务，而不是一直等到此服务超时！

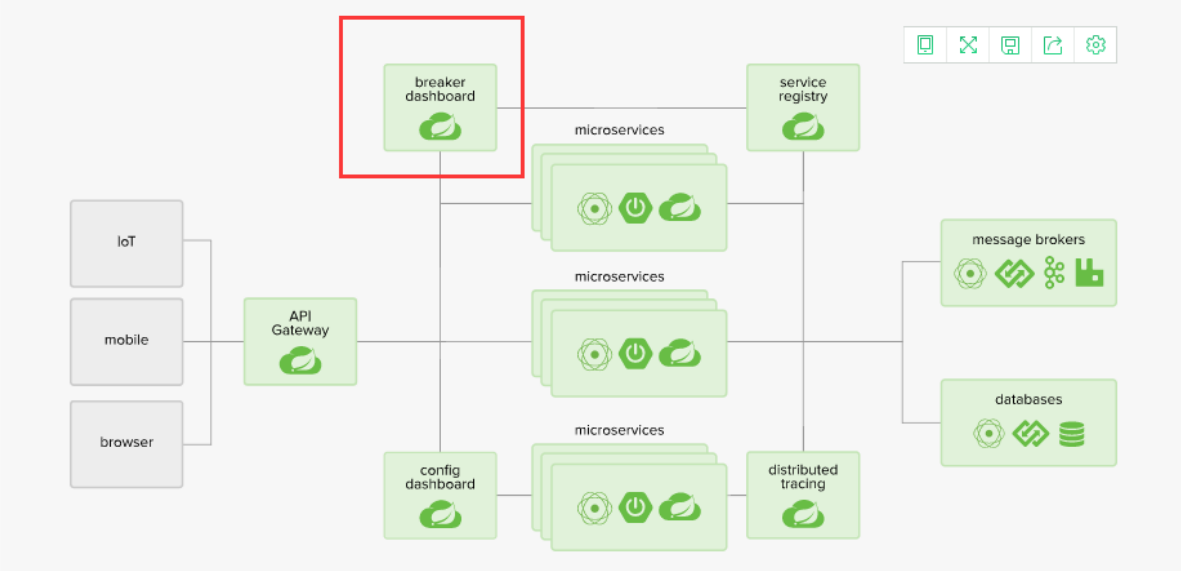
服务降级：所谓降级，一般是从整体负荷考虑，就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强。

> 服务监控

服务监控 hystrixDashboard

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

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



新建工程springcloud-consumer-hystrix-dashboard-9001

Pom.xml

复制之前80项目的pom文件，新增以下依赖！

```
<!--Hystrix-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    <version>1.4.7.RELEASE</version>
</dependency>
```

application.yaml配置

```
server:
  port: 9001
```

主启动类改名 + 新注解@EnableHystrixDashboard

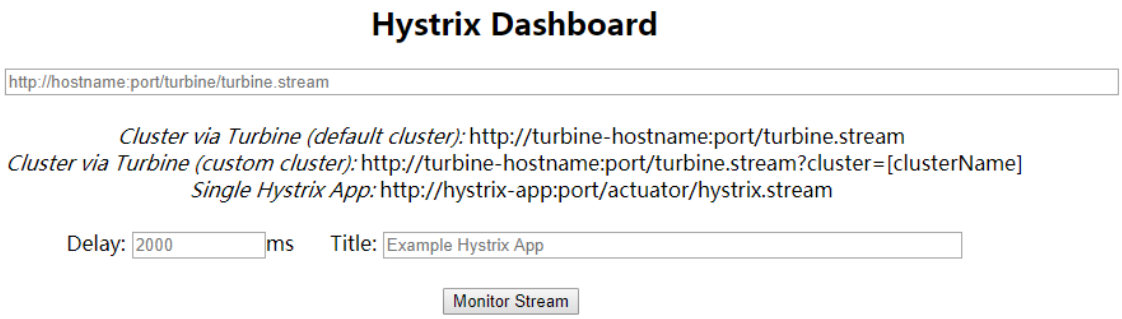
```
package com.kuang.springcloud;

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

@SpringBootApplication
@EnableHystrixDashboard
public class DeptConsumerDashBoardApp9001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumerDashBoardApp9001.class,args);
    }
}
```

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

```
<!--actuator监控信息完善-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

启动springcloud-consumer-hystrix-dashboard-9001该微服务监控消费端

http://localhost:9001/hystrix



这是动物是一只豪猪!

测试一

1. 启动eureka集群
2. 启动springcloud-consumer-hystrix-dashboard-9001
3. 在 springcloud-provider-dept-hystrix-8001 启动类中增加一个bean

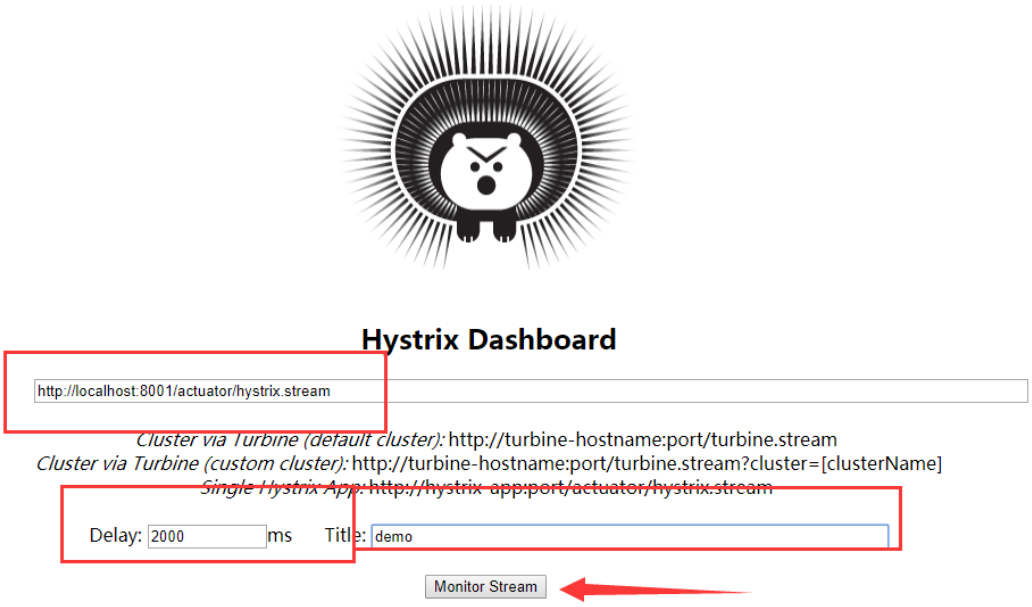
```
@Bean
public ServletRegistrationBean hystrixMetricsStreamServlet() {
    ServletRegistrationBean registration = new
ServletRegistrationBean(new HystrixMetricsStreamServlet());
    registration.addUrlMappings("/actuator/hystrix.stream");
    return registration;
}
```

4. 启动 springcloud-provider-dept-hystrix-8001
    1. http://localhost:8001/dept/get/1
    2. http://localhost:8001/actuator/hystrix.stream 【查看1秒一动的数据流】

监控测试

- 多次刷新 http://localhost:8001/dept/get/1

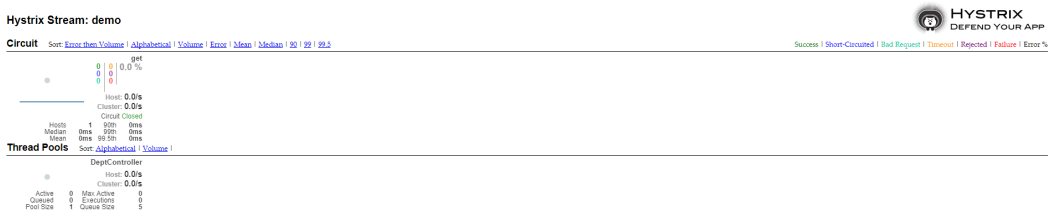
- 观察监控窗口，就是那个豪猪页面
  - 添加监控地址



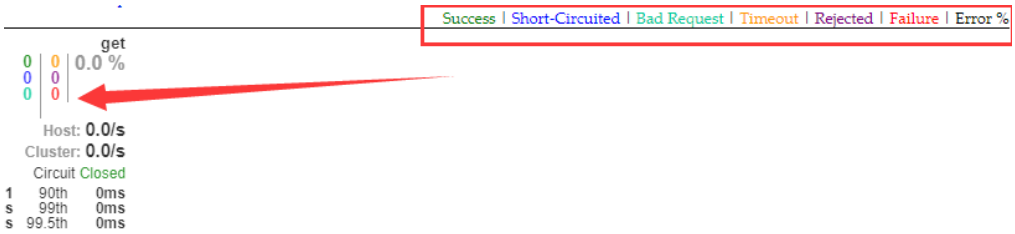
Delay：该参数用来控制服务器上轮询监控信息的延迟时间，默认为2000毫秒，可以通过配置该属性来降低客户端的网络和CPU消耗

Title： 该参数对应了头部标题HystrixStream之后的内容，默认会使用具体监控实例URL，可以通过配置该信息来展示更合适的标题。

  - 监控结果



  - 如何看
    - 7色

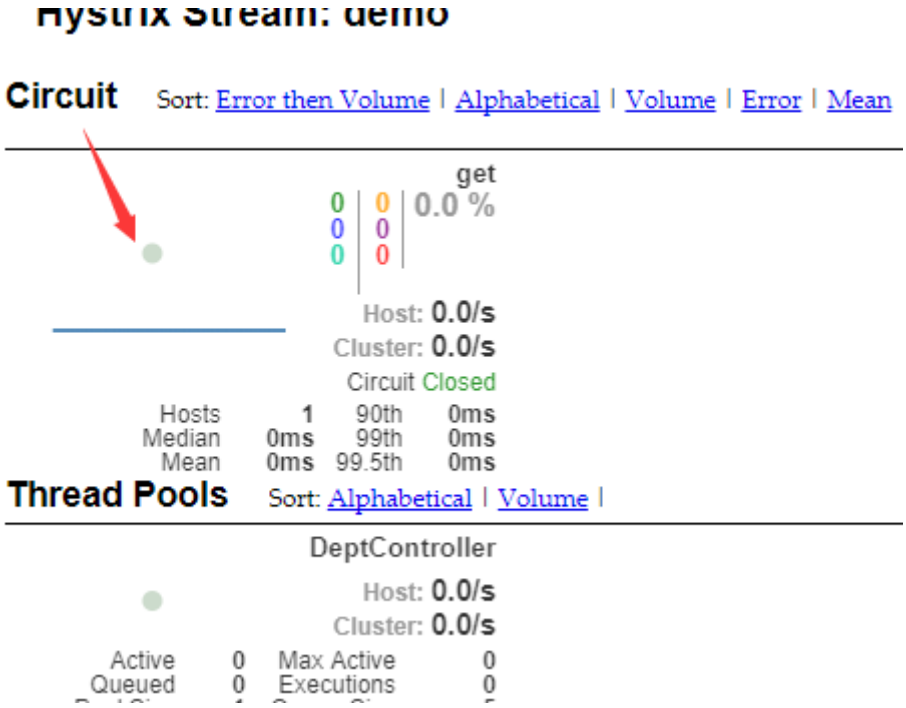


    - 一圈

      实心圆：公有两种含义，他通过颜色的变化代表了实例的健康程度

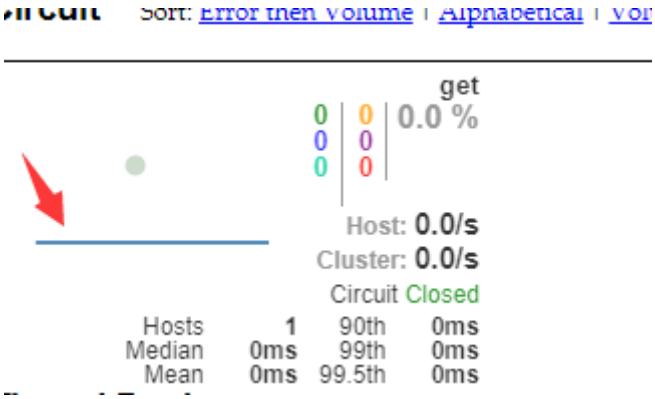
      它的健康程度从绿色<黄色<橙色<红色 递减

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

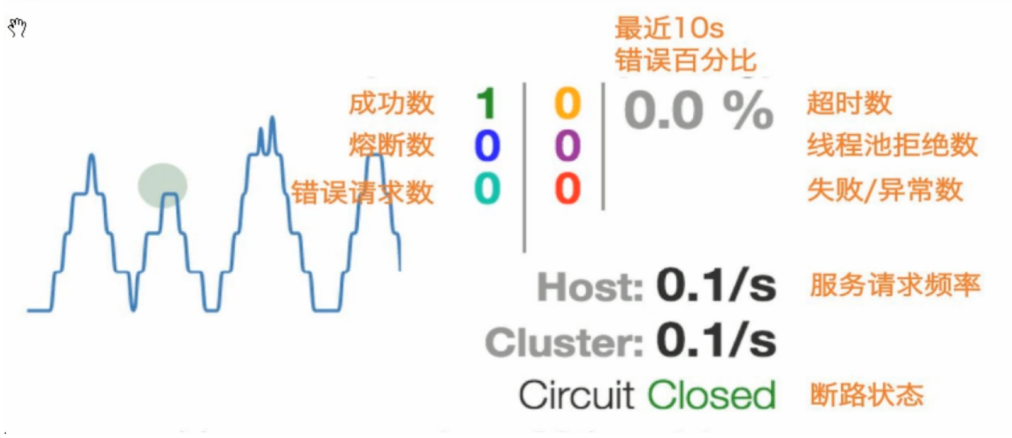


- 一线

  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



  - 搞懂一个才能看懂复杂的

Zuul路由网关

> 概述

什么是Zuul?

Zuul包含了对请求的路由和过滤两个最主要的功能：

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础，而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验，服务聚合等功能的基础。Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

注意：Zuul服务最终还是会注册进Eureka

提供：代理 + 路由 + 过滤 三大功能!

Zuul能干嘛?

- 路由
- 过滤

官网文档：https://github.com/Netflix/zuul

> 路由的基本配置

新建Module模块springcloud-zuul-gateway-9527

pom文件

添加Eureka和zuul的配置

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

```
        <version>1.4.7.RELEASE</version>
    </dependency>
    <!--eureka相关-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
        <version>1.4.7.RELEASE</version>
    </dependency>
</dependencies>
```

application.yaml配置

```
server:
  port: 9527

#spring的相关配置
spring:
  application:
    name: springcloud-zuul-gateway

#eureka配置
eureka:
  client:
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http:/
/eureka7003.com:7003/eureka/
  instance:
    instance-id: gateway9527.com
    prefer-ip-address: false  #true访问路径可以显示IP地址

#info配置
info:
  app.name: kuang-springcloud
  company.name: www.kuangstudy.com
  build.artifactId: ${project.artifactId}
  build.version: ${project.version}
```

hosts修改

路径：C:\Windows\System32\drivers\etc\hosts

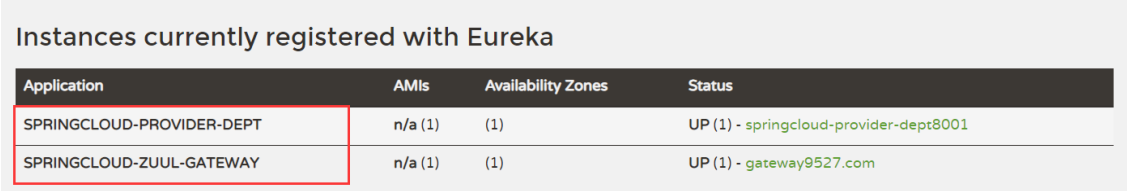
```
127.0.0.1   myzuul.com
```

主启动类

```
@SpringBootApplication
@EnableZuulProxy
public class SpringCloudZuulApp9527 {
    public static void main(String[] args) {
        SpringApplication.run(SpringCloudZuulApp9527.class,args);
    }
}
```

启动

- Eureka集群
- 一个服务提供类：springcloud-provider-dept-8001
- zuul路由
- 访问：http://localhost:7001/



测试

- 不用路由：http://localhost:8001/dept/get/2
- 使用路由：http://myzuul.com:9527/springcloud-provider-dept/dept/get/2
  - 网关 / 微服务名字 / 具体的服务

> 路由访问映射规则

问题：http://myzuul.com:9527/springcloud-provider-dept/dept/get/2 这样去访问的话，就暴露了我们真实微服务的名称！这不是我们需要的！怎么处理呢?

修改：springcloud-zuul-gateway-9527 工程

代理名称

yml配置修改，增加Zuul路由映射！

```yaml
#Zuul路由映射
zuul:
  routes:
    mydept.serviceId: springcloud-provider-dept
    mydept.path: /mydept/**
```

配置前访问：http://myzuul.com:9527/springcloud-provider-dept/dept/get/2

配置后访问：http://myzuul.com:9527/mydept/dept/get/2

问题，现在访问原路径依旧可以访问！这不是我们所希望的！

原真实服务名忽略

```yaml
# Zuul路由映射
zuul:
  ignored-services: springcloud-provider-dept  # 不能再使用这个服务名访问；ignored：忽略
  routes:
    mydept.serviceId: springcloud-provider-dept
    mydept.path: /mydept/**
```

测试：现在访问http://myzuul.com:9527/springcloud-provider-dept/dept/get/2 就访问不了了

上面的例子中，我们只写了一个，那要是有多个需要隐藏，怎么办呢?

```yaml
#Zuul路由映射
zuul:
  ignored-services: "*"  # 通配符 * ， 隐藏全部的!
  routes:
    mydept.serviceId: springcloud-provider-dept
    mydept.path: /mydept/**
```

设置统一公共前缀

```yaml
#Zuul路由映射
zuul:
  prefix: /kuang
  ignored-services: springcloud-provider-dept
  routes:
    mydept.serviceId: springcloud-provider-dept
    mydept.path: /mydept/**
```

访问：http://myzuul.com:9527/kuang/mydept/dept/get/2 ，加上统一的前缀！kuang，否则，就访问不了了!