狂神说SpringCloud学习笔记

兴趣使然的草帽路飞 VIP 分类：学习笔记 浏览：13596 评论：10 字体 皮肤 左侧展开 最后修改于： 2021/03/25 12:33:42

展开目录

笔记整理来源 B站UP主狂神说https://www.bilibili.com/video/BV1jJ411S7xr

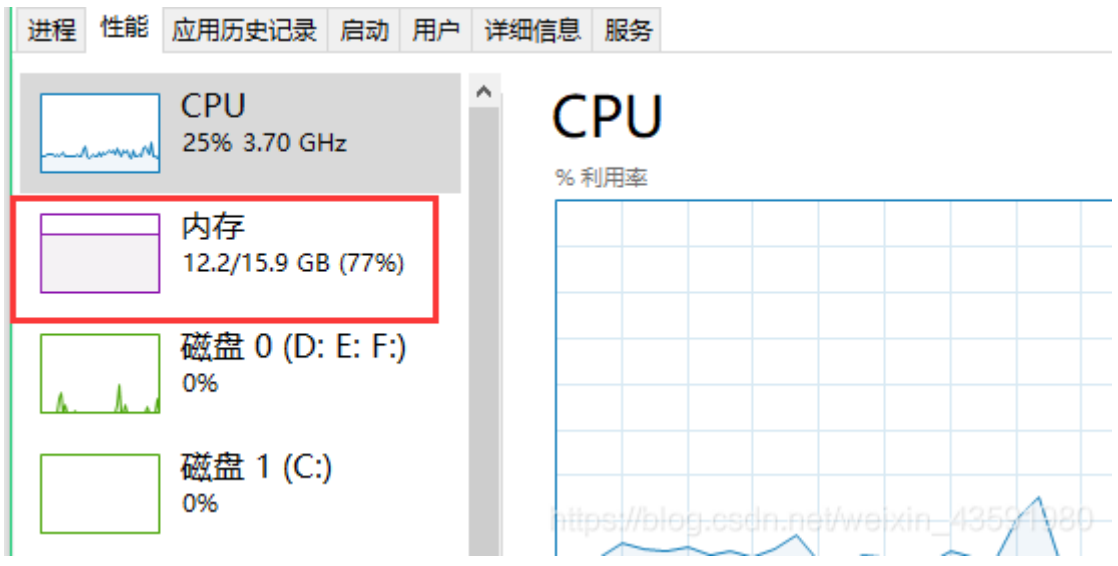
学习前言

1.1 学习前提

- 熟练使用SpringBoot 微服务快速开发框架
- 了解过Dubbo + Zookeeper 分布式基础

- 电脑配置内存不低于8G(我自己的是16G)

    - 给大家看下多个服务跑起来后的内存开销图：



1.2 文章大纲

> Spring Cloud 五大组件

- 服务注册与发现——**Netflix Eureka**

- 负载均衡：

    - 客户端负载均衡——**Netflix Ribbon**
    - 服务端负载均衡：——**Feign**(其也是依赖于Ribbon，只是将调用方式RestTemplete 更改成Service 接口)
- 断路器——**Netflix Hystrix**
- 服务网关——**Netflix Zuul**
- 分布式配置——**Spring Cloud Config**

1.3 常见面试题

1.1 什么是微服务?

1.2 微服务之间是如何独立通讯的?

1.3 SpringCloud 和 Dubbo有那些区别?

1.4 SpringBoot 和 SpringCloud，请谈谈你对他们的理解

1.5 什么是服务熔断？什么是服务降级?

1.6 微服务的优缺点分别是什么？说下你在项目开发中遇到的坑

TOP

1.7 你所知道的微服务技术栈有哪些？列举一二

1.8 Eureka和Zookeeper都可以提供服务注册与发现的功能，请说说两者的区别

...

2. 微服务概述

2.1 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，**它体长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(**HTTP**)互相沟通，每个服务都围绕着具体的业务进行构建，并且能狗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，应该根据业务上下文，选择合适的语言，工具(**Maven**)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。

> 再来从技术维度角度理解下：

- 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。

2.2 微服务与微服务架构

> 微服务

强调的是服务的大小，它关注的是某一个点，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，可以看作是IDEA中的一个个微服务工程，或者Moudel。IDEA 工具里面使用Maven开发的一个个独立的小Moudel，它具体是使用SpringBoot开发的一个小模块，专业的事情交给专业的模块来做，一个模块就做着一件事情。强调的是一个个的个体，每个个体完成一个具体的任务或者功能。

> 微服务架构

一种新的架构形式，Martin Fowler 于2014年提出。

微服务架构是一种架构模式，它体长将单一应用程序划分成一组小的服务，服务之间相互协调，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，服务与服务之间采用轻量级的通信机制**(如HTTP)**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**(如Maven)**对其进行构建。

2.3 微服务优缺点

> 优点

- 单一职责原则；
- 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，这个团队只需2-5个开发人员组成；
- 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；
- 微服务允许利用和融合最新技术；
- **微服务只是业务逻辑的代码，不会和HTML，CSS，或其他的界面混合；**
- **每个微服务都有自己的存储能力，可以有自己的数据库，也可以有统一的数据库；**

> 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；
- 系统部署依赖问题；
- 服务间通信成本问题；
- 数据一致性问题；
- 系统集成测试问题；
- 性能和监控问题；

2.4 微服务技术栈有那些?

微服务技术条目	落地技术
服务开发	**SpringBoot**、**Spring**、**SpringMVC**等
服务配置与管理	**Netfix**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**、PRC、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)	**Fegin**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
数据流操作开发包	**SpringCloud Stream**(封装与**Redis**，**Rabbit**，**Kafka**等发送接收消息)
时间消息总栈	**SpringCloud Bus**
服务部署	**Docker**、**OpenStack**、**Kubernetes**等

2.5 为什么选择SpringCloud作为微服务架构

01. 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

02. 当前各大IT公司用的微服务架构有那些?

- 阿里：dubbo+HFS
- 京东：JFS
- 新浪：Motan

- 当当网：DubboX

...

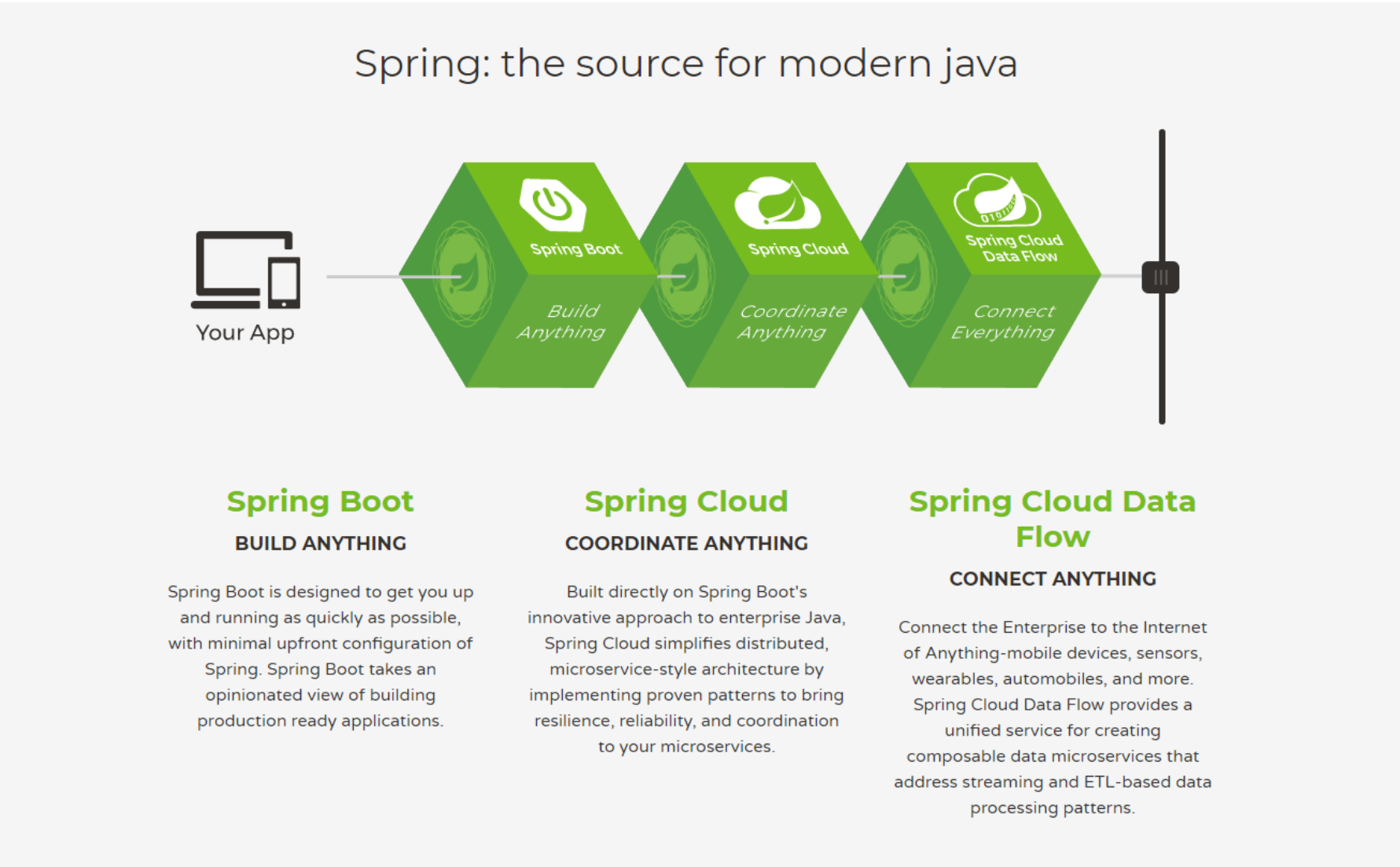
03. 各微服务框架对比

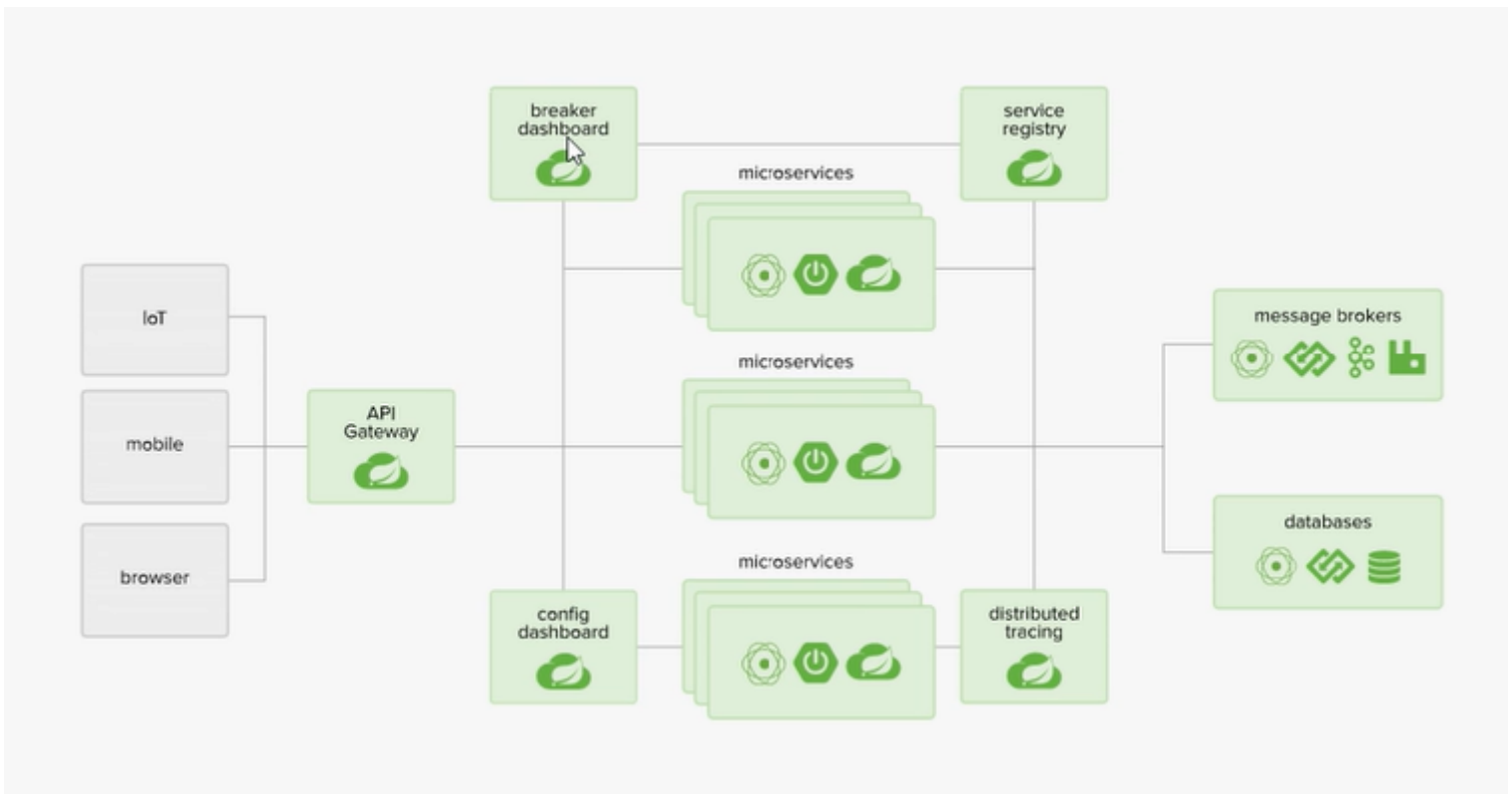
功能点/服务框架	Netflix/SpringCloud	Motan	gRPC	Thri t	Dubbo/DubboX
功能定位	完整的微服务框架	RPC框架，但整合了ZK或Consul，实现集群环境的基本服务注册发现	RPC框架	RPC框架	服务框架
支持Rest	是，Ribbon支持多种可拔插的序列号选择	否	否	否	否
支持RPC	否	是(Hession2)	是	是	是
支持多语言	是(Rest形式)	否	是	是	否
负载均衡	是(服务端zuul+客户端Ribbon)，zuul-服务，动态路由，云端负载均衡Eureka（针对中间层服务器）	是(客户端)	否	否	是(客户端)
配置服务	Netfix 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	中等	低	高	高	低
文档丰富程度	高	一般	一般	一般	高
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

3. SpringCloud入门概述

3.1 SpringCloud是什么?

Spring官网：https://spring.io/





3.2 SpringCloud和SpringBoot的关系

- SpringBoot专注于开苏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、为代理、事件总栈、全局锁、决策竞选、分布式会话等等集成服务；
- SpringBoot可以离开SpringCloud独立使用，开发项目，但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；

3.3 Dubbo 和 SpringCloud技术选型

1. 分布式+服务治理Dubbo

目前成熟的互联网架构，应用服务化拆分 + 消息中间件

2. Dubbo 和 SpringCloud对比

TOP

可以看一下社区活跃度：

https://github.com/dubbo

https://github.com/spring-cloud

对比结果：

	Dubbo	SpringCloud
服务注册中心	Zookeeper	Spring Cloud Netflix 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
消息总栈	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

最大区别：Spring Cloud 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，这个优点在当下强调快速演化的微服务环境下，显得更加合适。

品牌机和组装机的区别

社区支持与更新力度的区别

总结：二者解决的问题域不一样：Dubbo的定位是一款RPC框架，而SpringCloud的目标是微服务架构下的一站式解决方案。

3.4 SpringCloud能干嘛?

- 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
- ...

3.5 SpringCloud下载

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

版本号有点特别：

Spring Cloud Hoxton SR4

OVERVIEW | LEARN | SAMPLES

Documentation

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

Hoxton SR4 CURRENT GA	Reference Doc.	API Doc.
Hoxton SNAPSHOT	Reference Doc.	API Doc.
Greenwich SR5 GA	Reference Doc.	API Doc.
Greenwich SNAPSHOT	Reference Doc.	API Doc.

SpringCloud没有采用数字编号的方式命名版本号，而是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Realse版本：Angel，第二个Realse版本：Brixton，然后是Camden、Dalston、Edgware，目前最新的是Hoxton SR4 CURRENT GA通用稳定版。

自学参考书：

- SpringCloud Netflix 中文文档：https://springcloud.cc/spring-cloud-netflix.html
- SpringCloud 中文API文档(官方文档翻译版)：https://springcloud.cc/spring-cloud-dalston.html
- SpringCloud中国社区：http://springcloud.cn/
- SpringCloud中文网：https://springcloud.cc

4. SpringCloud Rest学习环境搭建：服务提供者

4.1 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。
- 回顾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)聚合了很多子项目(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Moudule子模块

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

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

4.2 SpringCloud版本选择

大版本说明

SpringBoot	SpringCloud	关系
1.2.x	Angel版本(天使)	兼容SpringBoot1.2x
1.3.x	Brixton版本(布里克斯顿)	兼容SpringBoot1.3x，也兼容SpringBoot1.4x
1.4.x	Camden版本(卡姆登)	兼容SpringBoot1.4x，也兼容SpringBoot1.5x
1.5.x	Dalston版本(多尔斯顿)	兼容SpringBoot1.5x，不兼容SpringBoot2.0x
1.5.x	Edgware版本(埃奇韦尔)	兼容SpringBoot1.5x，不兼容SpringBoot2.0x
2.0.x	Finchley版本(芬奇利)	兼容SpringBoot2.0x，不兼容SpringBoot1.5x
2.1.x	Greenwich版本(格林威治)	

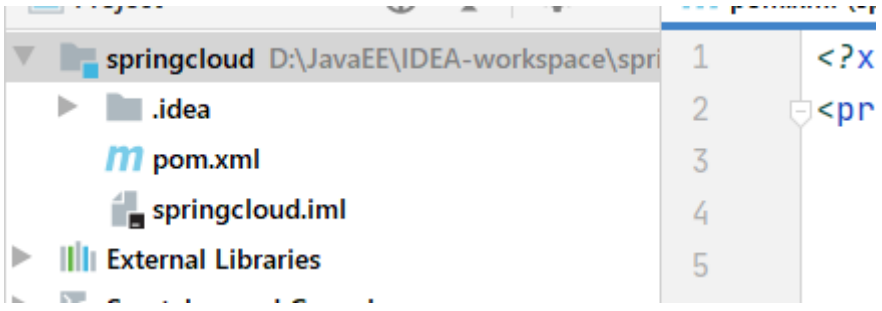
实际开发版本关系

spring-boot-starter-parent		spring-cloud-dependencles	
版本号	**发布日期**	**版本号**	**发布日期**
1.5.2.RELEASE	2017-03	Dalston.RC1	2017-x
1.5.9.RELEASE	2017-11	Edgware.RELEASE	2017-11
1.5.16.RELEASE	2018-04	Edgware.SR5	2018-10
1.5.20.RELEASE	2018-09	Edgware.SR5	2018-10
2.0.2.RELEASE	2018-05	Fomchiey.BULD-SNAPSHOT	2018-x
2.0.6.RELEASE	2018-10	Fomchiey-SR2	2018-10
2.1.4.RELEASE	2019-04	Greenwich.SR1	2019-03

使用后两个

4.3 创建父工程

- 新建父工程项目springcloud，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

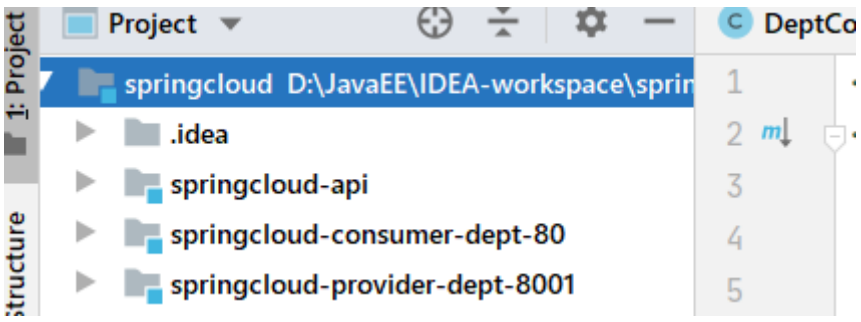


pom.xml

```xml
<?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.haust</groupId>
    <artifactId>springcloud</artifactId>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>springcloud-api</module>
        <module>springcloud-provider-dept-8001</module>
        <module>springcloud-consumer-dept-80</module>
        <module>springcloud-eureka-7001</module>
        <module>springcloud-eureka-7002</module>
        <module>springcloud-eureka-7003</module>
        <module>springcloud-provider-dept-8002</module>
        <module>springcloud-provider-dept-8003</module>
        <module>springcloud-consumer-dept-feign</module>
        <module>springcloud-provider-dept-hystrix-8001</module>
        <module>springcloud-consumer-hystrix-dashboard</module>
        <module>springcloud-zuul-9527</module>
        <module>springcloud-config-server-3344</module>
        <module>springcloud-config-client-3355</module>
        <module>springcloud-config-eureka-7001</module>
        <module>springcloud-config-dept-8001</module>
    </modules>

    <!--打包方式  pom-->
    <packaging>pom</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <junit.version>4.12</junit.version>
        <log4j.version>1.2.17</log4j.version>
        <lombok.version>1.16.18</lombok.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-alibaba-dependencies</artifactId>
                <version>0.2.0.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--springCloud的依赖-->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Greenwich.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--SpringBoot-->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.1.4.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--数据库-->
```

```
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.1.47</version>
            </dependency>
            <dependency>
                <groupId>com.alibaba</groupId>
                <artifactId>druid</artifactId>
                <version>1.1.10</version>
            </dependency>
            <!--SpringBoot 启动器-->
            <dependency>
                <groupId>org.mybatis.spring.boot</groupId>
                <artifactId>mybatis-spring-boot-starter</artifactId>
                <version>1.3.2</version>
            </dependency>
            <!--日志测试~-->
            <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>
```

父工程为springcloud，其下有多个子mudule，详情参考完整代码了解



springcloud-consumer-dept-80访问springcloud-provider-dept-8001下的controller使用REST方式

如**DeptConsumerController.java**

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:44
 * @Description:
 */
@RestController
public class DeptConsumerController {


    /**
     * 理解：消费者，不应该有service层~
     * RestTemplate .... 供我们直接调用就可以了！ 注册到Spring中
     * (地址：url, 实体：Map ,Class<T> responseType)
     * <p>
     * 提供多种便捷访问远程http服务的方法，简单的Restful服务模板~
     */
    @Autowired
    private RestTemplate restTemplate;

    /**
     * 服务提供方地址前缀
     * <p>
     * Ribbon:我们这里的地址，应该是一个变量，通过服务名来访问
     */
    private static final String REST_URL_PREFIX = "http://localhost:8001";
    //private static final String REST_URL_PREFIX = "http://SPRINGCLOUD-PROVIDER-DEPT";

    /**
     * 消费方添加部门信息
     * @param dept
     * @return
     */
    @RequestMapping("/consumer/dept/add")
    public boolean add(Dept dept) {

        // postForObject(服务提供方地址(接口),参数实体,返回类型.class)
        return restTemplate.postForObject(REST_URL_PREFIX + "/dept/add", dept, Boolean.class);
    }

    /**
     * 消费方根据id查询部门信息
     * @param id
     * @return
     */
    @RequestMapping("/consumer/dept/get/{id}")
    public Dept get(@PathVariable("id") Long id) {

        // getForObject(服务提供方地址(接口),返回类型.class)
        return restTemplate.getForObject(REST_URL_PREFIX + "/dept/get/" + id, Dept.class);
    }

    /**
     * 消费方查询部门信息列表
     * @return
     */
    @RequestMapping("/consumer/dept/list")
    public List<Dept> list() {

        return restTemplate.getForObject(REST_URL_PREFIX + "/dept/list", List.class);
    }
}
```

使用RestTemplete先需要放入Spring容器中

ConfigBean.java

```java
@Configuration
public class ConfigBean {
    //@Configuration -- spring  applicationContext.xml

    //配置负载均衡实现RestTemplate
    // IRule
    // RoundRobinRule 轮询
    // RandomRule 随机
    // AvailabilityFilteringRule ： 会先过滤掉，跳闸，访问故障的服务~，对剩下的进行轮询~
    // RetryRule ： 会先按照轮询获取服务~，如果服务获取失败，则会在指定的时间内进行，重试
    @Bean
    public RestTemplate getRestTemplate(){

        return new RestTemplate();
    }
}
```

springcloud-provider-dept-8001的dao接口调用springcloud-api模块下的pojo，可使用在springcloud-provider-dept-8001的pom文件导入springcloud-api模块依赖的方式：

```xml
<!--我们需要拿到实体类，所以要配置api module-->
        <dependency>
            <groupId>com.haust</groupId>
            <artifactId>springcloud-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
```

springcloud-consumer-dept-80和springcloud-provider-dept-8001的pom.xml和父工程下的依赖基本一样，直接看完整代码即可，不再添加重复笔记。

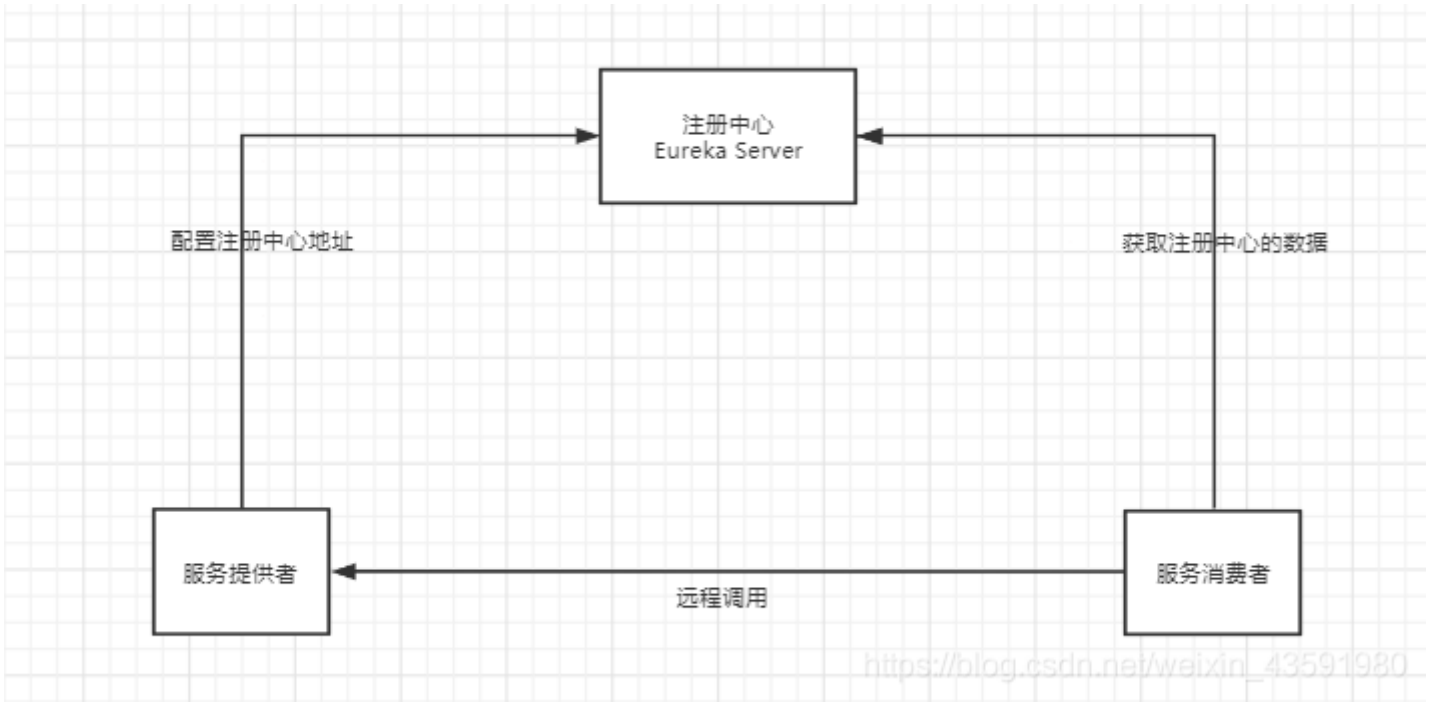
5. Eureka服务注册中心

5.1 什么是Eureka

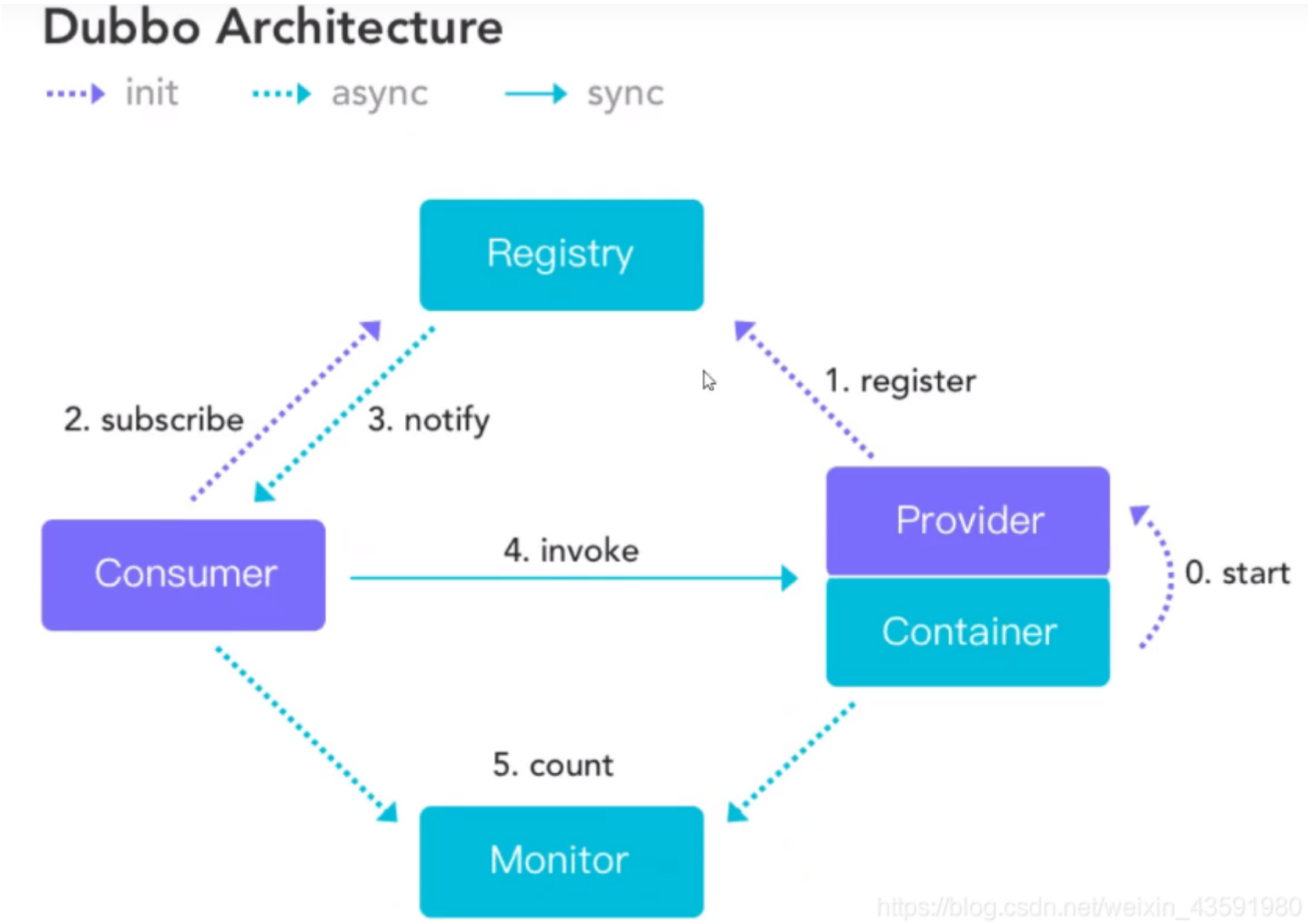
- Netflix在涉及Eureka时，遵循的就是API原则.
- Eureka是Netflix的有个子模块，也是核心模块之一。Eureka是基于REST的服务，用于定位服务，以实现云端中间件层服务发现和故障转移，服务注册与发现对于微服务来说是非常重要的，有了服务注册与发现，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了，功能类似于Dubbo的注册中心，比如Zookeeper.

5.2 原理理解

- **Eureka基本的架构**
    - Springcloud 封装了Netflix公司开发的Eureka模块来实现服务注册与发现 (对比Zookeeper).
    - 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，各个节点启动后，回在EurekaServer中进行注册，这样Eureka Server中的服务注册表中将会储存所有课用服务节点的信息，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90s).

- **三大角色**
  - Eureka Server：提供服务的注册与发现
  - Service Provider：服务生产方，将自身服务注册到Eureka中，从而使服务消费方能狗找到
  - 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
- 目前工程状况

```
     */
    public static final String PREFIX = "eureka.client";

    /**
     * Default Eureka URL.
     */
    public static final String DEFAULT_URL = "http://localhost:8761" + DEFAULT_PREFIX
            + "/";
```


5.3 构建步骤

1. eureka-server

01. springcloud-eureka-7001 模块建立

02. pom.xml 配置

```xml
<!--导包~-->
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-eureka-server -->
    <!--导入Eureka Server依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--热部署工具-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
</dependencies>
```

03. application.yml

```yaml
server:
  port: 7001

# Eureka配置
eureka:
  instance:
    # Eureka服务端的实例名字
    hostname: 127.0.0.1
  client:
    # 表示是否向 Eureka 注册中心注册自己(这个模块本身是服务器,所以不需要)
    register-with-eureka: false
    # fetch-registry如果为false,则表示自己为注册中心,客户端的化为 ture
    fetch-registry: false
    # Eureka监控页面~
    service-url:
      defaultZone: http://${
      eureka.instance.hostname}:${
      server.port}/eureka/
```

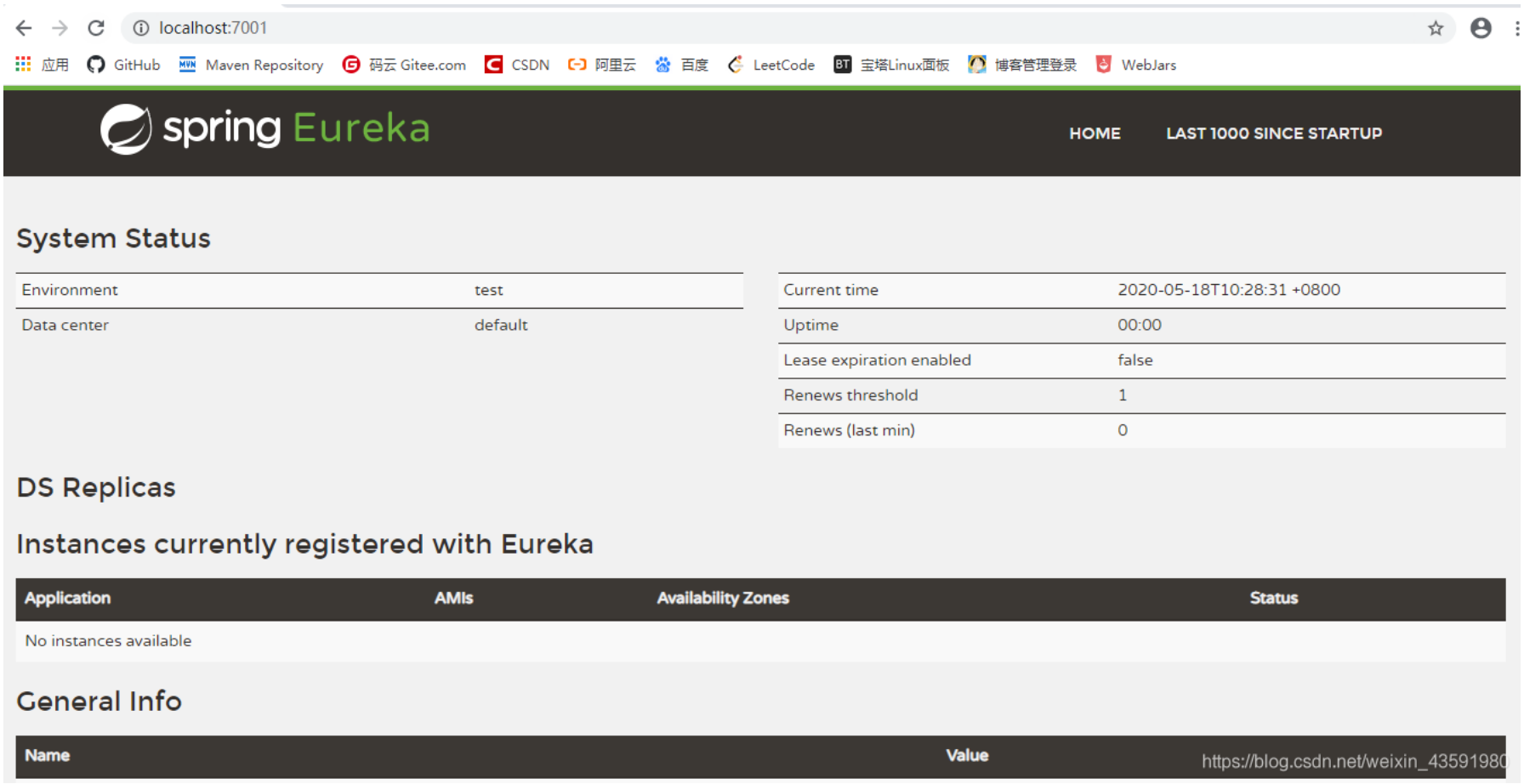
源码中Eureka的默认端口以及访问路径:

```
 */
public static final String PREFIX = "eureka.client";

/**
 * Default Eureka URL.
 */
public static final String DEFAULT_URL = "http://localhost:8761" + DEFAULT_PREFIX
        + "/";
```


01. 主启动类

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/18/10:26
 * @Description: 启动之后，访问 http://127.0.0.1:7001/
 */
@SpringBootApplication
// @EnableEurekaServer 服务端的启动类，可以接受别人注册进来~
@EnableEurekaServer
public class EurekaServer_7001 {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServer_7001.class,args);
    }
}
```

02. 启动成功后访问 http://localhost:7001/ 得到以下页面



2. eureka-client

调整之前创建的springlouc-provider-dept-8001

01. 导入Eureca依赖

```xml
<!--Eureka依赖-->
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-eureka -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
```

02. application中新增Eureca配置

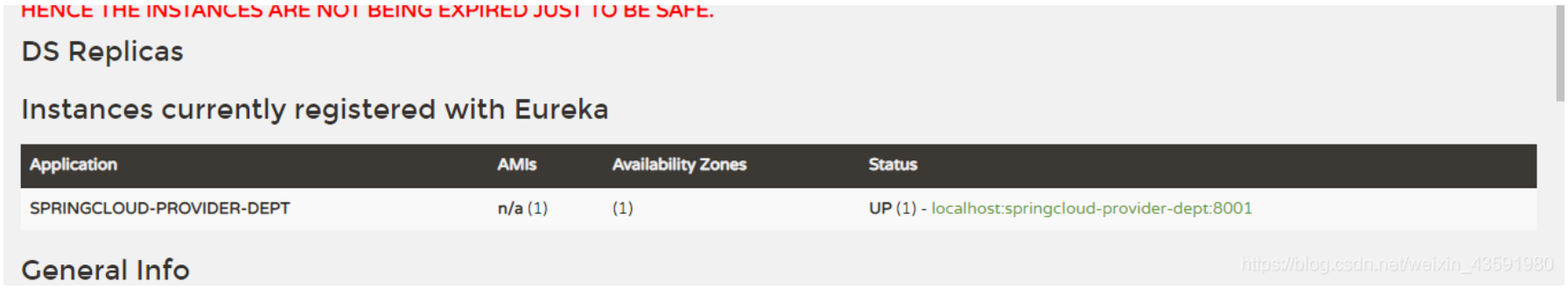
```yaml
# Eureka配置：配置服务注册中心地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/
```

03. 为主启动类添加@EnableEurekaClient注解

TOP

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:09
 * @Description: 启动类
 */
@SpringBootApplication
// @EnableEurekaClient 开启Eureka客户端注解，在服务启动后自动向注册中心注册服务
@EnableEurekaClient
public class DeptProvider_8001 {

    public static void main(String[] args) {

        SpringApplication.run(DeptProvider_8001.class,args);
    }
}
```

04. 先启动7001服务端后启动8001客户端进行测试，然后访问监控页http://localhost:7001/ 产看结果如图，成功

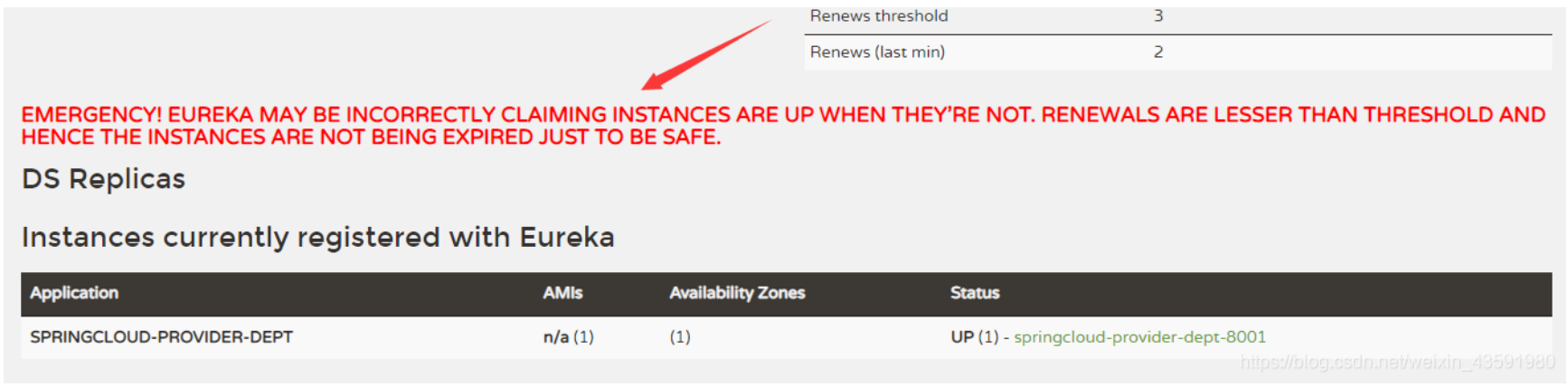


01. 修改Eureka上的默认描述信息

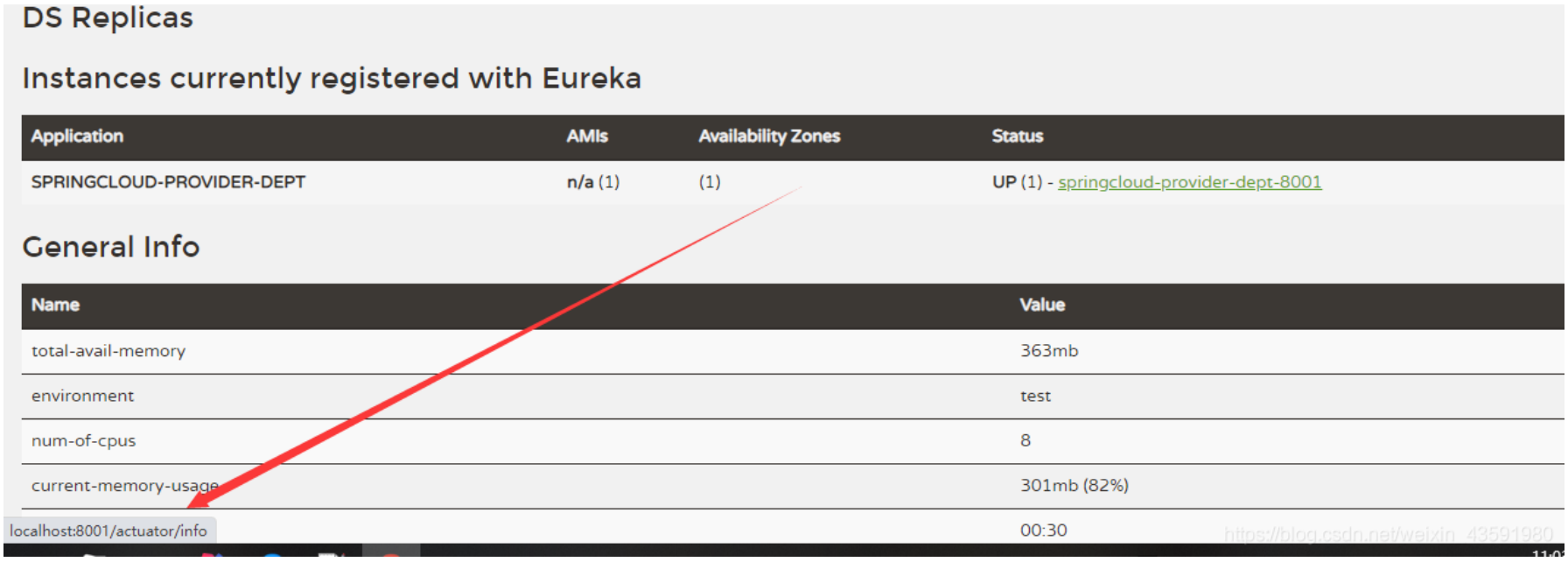
```yaml
# Eureka配置：配置服务注册中心地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/
  instance:
    instance-id: springcloud-provider-dept-8001 #修改Eureka上的默认描述信息
```

结果如图：

如果此时停掉springcloud-provider-dept-8001 等**30s**后 监控会开启保护机制：



02. 配置关于服务加载的监控信息



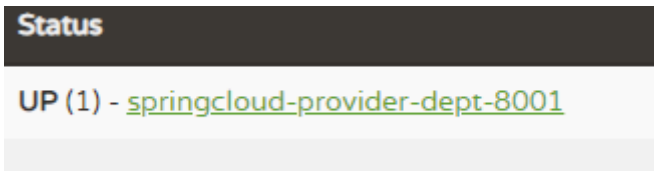
pom.xml中添加依赖

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

application.yml中添加配置

```
# info配置
info:
# 项目的名称
app.name: haust-springcloud
# 公司的名称
company.name: 河南科技大学西苑校区软件学院
```

此时刷新监控页，点击进入



跳转新页面显示如下内容：

3. EureKa自我保护机制：好死不如赖活着

一句话总结就是：**某时刻某一个微服务不可用，eureka不会立即清理，依旧会对该微服务的信息进行保存！**

- 默认情况下，当eureka server在一定时间内没有收到实例的心跳，便会把该实例从注册表中删除（**默认是90秒**），但是，如果短时间内丢失大量的实例心跳，便会触发eureka server的自我保护机制，比如在开发测试时，需要频繁地重启微服务实例，但是我们很少会把eureka server一起重启（因为在开发过程中不会修改eureka注册中心），**当一分钟内收到的心跳数大量减少时，会触发该保护机制**。可以在eureka管理界面看到Renews threshold和Renews(last min)，当后者（最后一分钟收到的心跳数）小于前者（心跳阈值）的时候，触发保护机制，会出现红色的警告：`EMERGENCY!EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT.RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEGING EXPIRED JUST TO BE SAFE.` 从警告中可以看到，eureka认为虽然收不到实例的心跳，但它认为实例还是健康的，eureka会保护这些实例，不会把它们从注册表中删掉。
- 该保护机制的目的是避免网络连接故障，在发生网络故障时，微服务和注册中心之间无法正常通信，但服务本身是健康的，不应该注销该服务，如果eureka因网络故障而把微服务误删了，那即使网络恢复了，该微服务也不会重新注册到eureka server了，因为只有在微服务启动的时候才会发起注册请求，后面只会发送心跳和服务列表请求，这样的话，该实例虽然是运行着，但永远不会被其它服务所感知。所以，eureka server在短

时间内丢失过多的客户端心跳时，会进入自我保护模式，该模式下，eureka会保护注册表中的信息，不在注销任何微服务，当网络故障恢复后，eureka会自动退出保护模式。自我保护模式可以让集群更加健壮。

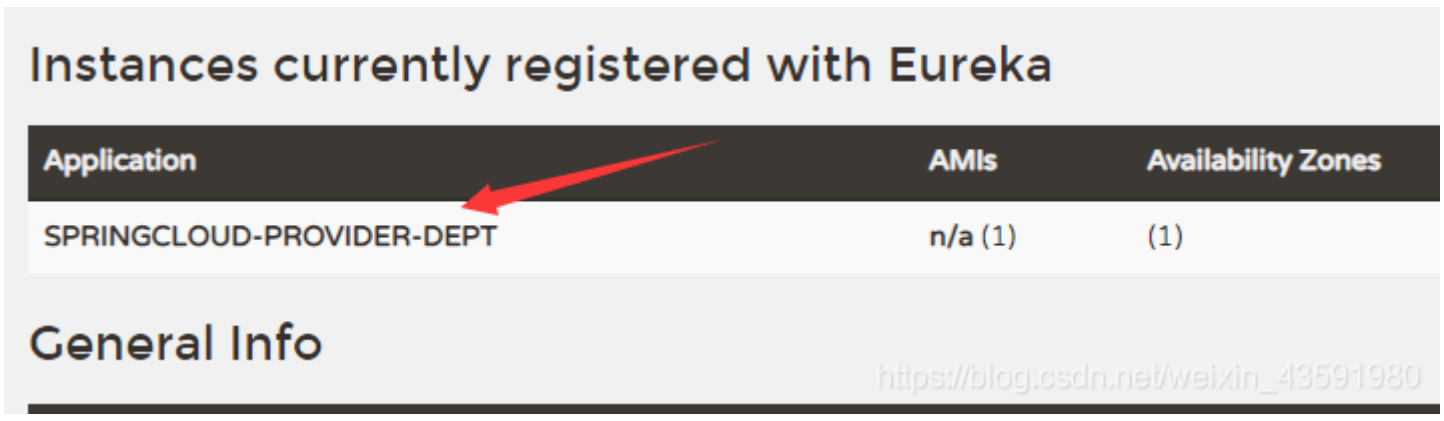
- 但是我们在开发测试阶段，需要频繁地重启发布，如果触发了保护机制，则旧的服务实例没有被删除，这时请求有可能跑到旧的实例中，而该实例已经关闭了，这就导致请求错误，影响开发测试。所以，在开发测试阶段，我们可以把自我保护模式关闭，只需在eureka server配置文件中加上如下配置即可：`eureka.server.enable-self-preservation=false`【不推荐关闭自我保护机制】

详细内容可以参考下这篇博客内容：https://blog.csdn.net/wudiyong22/article/details/80827594

4. 注册进来的微服务，获取一些消息（团队开发会用到）

DeptController.java新增方法

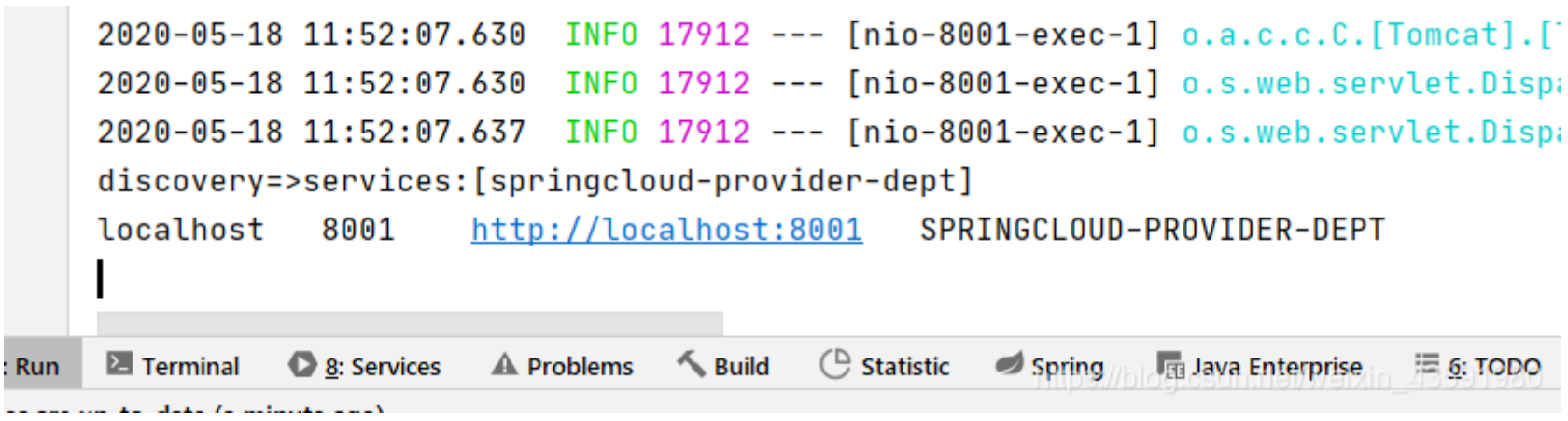
```
/**
 * DiscoveryClient 可以用来获取一些配置的信息，得到具体的微服务！
 */
@Autowired
private DiscoveryClient client;

/**
 * 获取一些注册进来的微服务的信息~，
 *
 * @return
 */
@GetMapping("/dept/discovery")
public Object discovery() {

    // 获取微服务列表的清单
    List<String> services = client.getServices();
    System.out.println("discovery=>services:" + services);
    // 得到一个具体的微服务信息,通过具体的微服务id，applicaioinName；
    List<ServiceInstance> instances = client.getInstances("SPRINGCLOUD-PROVIDER-DEPT");
    for (ServiceInstance instance : instances) {

        System.out.println(
                instance.getHost() + "\t" + // 主机名称
                        instance.getPort() + "\t" + // 端口号
                        instance.getUri() + "\t" + // uri
                        instance.getServiceId() // 服务id
        );
    }
    return this.client;
}
```



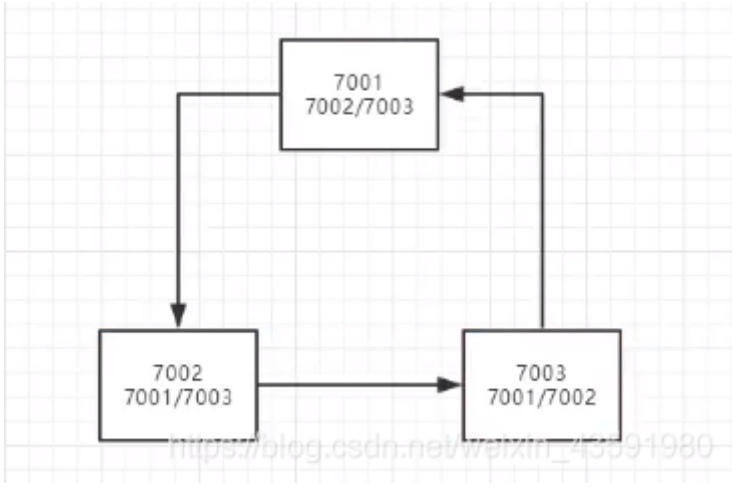
主启动类中加入@EnableDiscoveryClient 注解

```
@SpringBootApplication
// @EnableEurekaClient 开启Eureka客户端注解，在服务启动后自动向注册中心注册服务
@EnableEurekaClient
// @EnableEurekaClient 开启服务发现客户端的注解，可以用来获取一些配置的信息，得到具体的微服务
@EnableDiscoveryClient
public class DeptProvider_8001 {

    ...
}
```

结果如图：



5.4 Eureka：集群环境配置



1.初始化

新建springcloud-eureka-7002、springcloud-eureka-7003 模块

1.为pom.xml添加依赖 (与springcloud-eureka-7001相同)

```xml
<!--导包~-->
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-eureka-server -->
    <!--导入Eureka Server依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--热部署工具-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
</dependencies>
```

2.application.yml配置(与springcloud-eureka-7001相同)

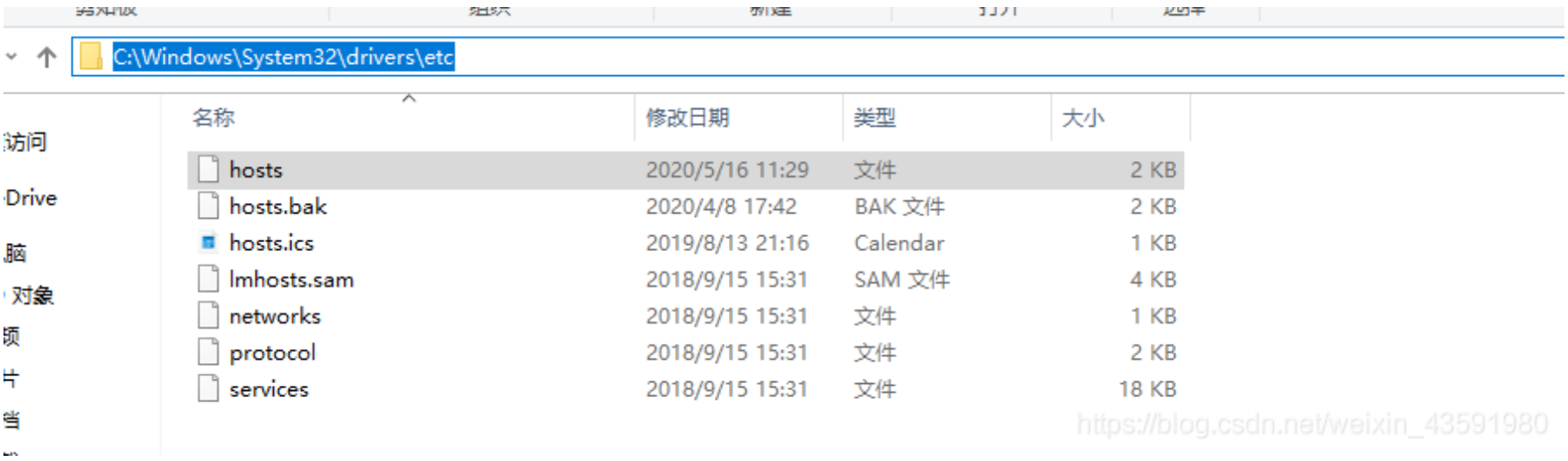
```yaml
server:
  port: 7003

# Eureka配置
eureka:
  instance:
    hostname: localhost # Eureka服务端的实例名字
  client:
    register-with-eureka: false # 表示是否向 Eureka 注册中心注册自己(这个模块本身是服务器,所以不需要)
    fetch-registry: false # fetch-registry如果为false,则表示自己为注册中心
    service-url: # 监控页面~
      # 重写Eureka的默认端口以及访问路径 --->http://localhost:7001/eureka/
      defaultZone: http://${
    eureka.instance.hostname}:${
    server.port}/eureka/
```

3.主启动类(与springcloud-eureka-7001相同)

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/18/10:26
 * @Description: 启动之后，访问 http://127.0.0.1:7003/
 */
@SpringBootApplication
// @EnableEurekaServer 服务端的启动类，可以接受别人注册进来~
public class EurekaServer_7003 {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServer_7003.class,args);
    }
}
```

2.集群成员相互关联

配置一些自定义本机名字，找到本机hosts文件并打开



在hosts文件最后加上，要访问的本机名称，默认是localhost

修改application.yml的配置，如图为springcloud-eureka-7001配置，springcloud-eureka-7002/springcloud-eureka-7003同样分别修改为其对应的名称即可

在集群中使springcloud-eureka-7001关联springcloud-eureka-7002、springcloud-eureka-7003

完整的springcloud-eureka-7001下的application.yml如下

```yaml
server:
  port: 7001

#Eureka配置
eureka:
  instance:
    hostname: eureka7001.com #Eureka服务端的实例名字
  client:
    register-with-eureka: false #表示是否向 Eureka 注册中心注册自己(这个模块本身是服务器,所以不需要)
    fetch-registry: false #fetch-registry如果为false,则表示自己为注册中心
    service-url: #监控页面~
      #重写Eureka的默认端口以及访问路径 --->http://localhost:7001/eureka/
      # 单机： defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
      # 集群（关联）：7001关联7002、7003
      defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

同时在集群中使springcloud-eureka-7002关联springcloud-eureka-7001、springcloud-eureka-7003

完整的springcloud-eureka-7002下的application.yml如下

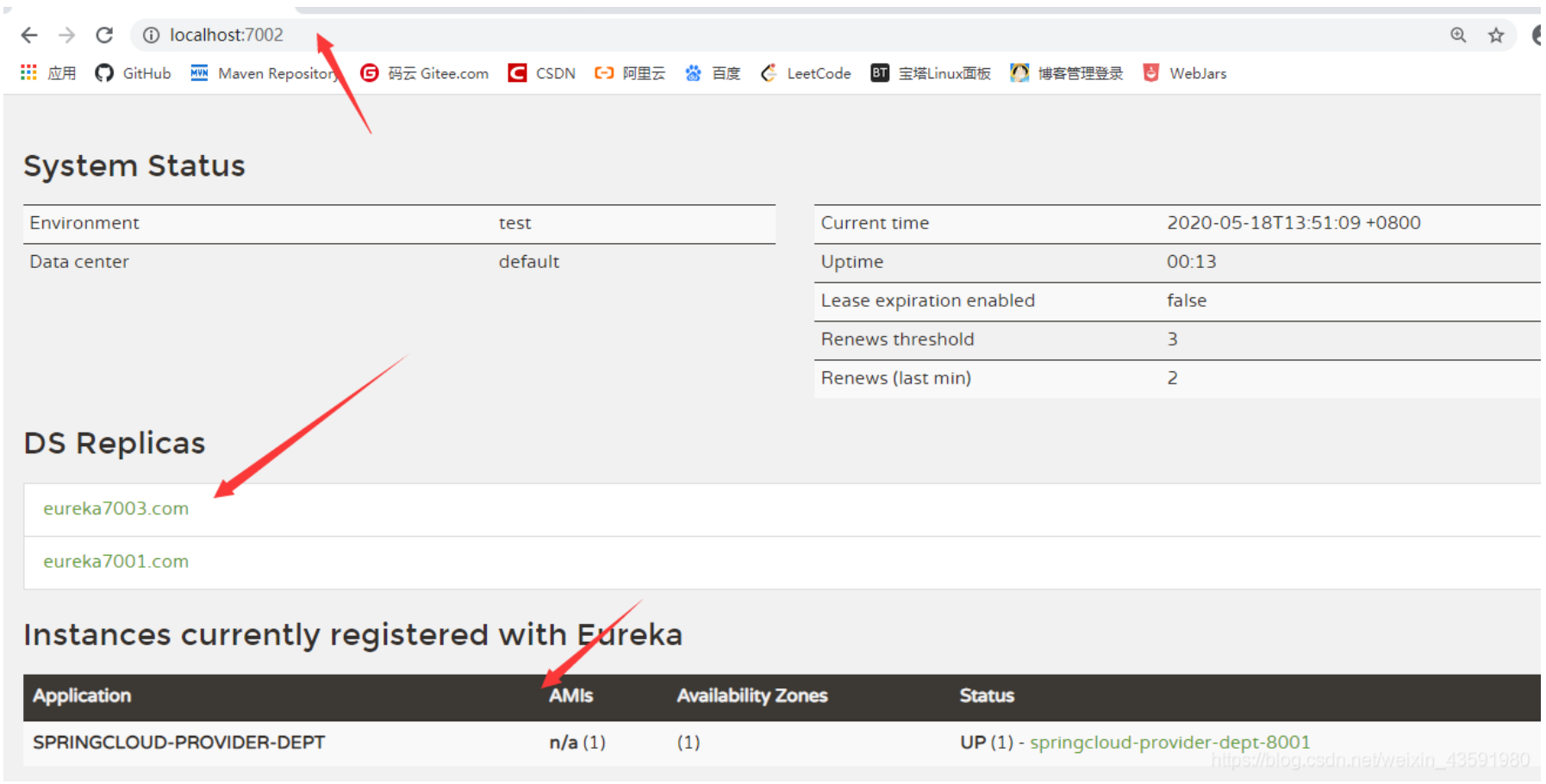
```yaml
server:
  port: 7002

#Eureka配置
eureka:
  instance:
    hostname: eureka7002.com #Eureka服务端的实例名字
  client:
    register-with-eureka: false #表示是否向 Eureka 注册中心注册自己(这个模块本身是服务器,所以不需要)
    fetch-registry: false #fetch-registry如果为false,则表示自己为注册中心
    service-url: #监控页面~
      #重写Eureka的默认端口以及访问路径 --->http://localhost:7001/eureka/
      # 单机： defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
      # 集群（关联）：7002关联7001、7003
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7003.com:7003/eureka/
```

springcloud-eureka-7003配置方式同理可得.

通过springcloud-provider-dept-8001下的yml配置文件，修改**Eureka配置：配置服务注册中心地址**

```yaml
# Eureka配置：配置服务注册中心地址
eureka:
  client:
    service-url:
      # 注册中心地址7001-7003
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
  instance:
    instance-id: springcloud-provider-dept-8001 #修改Eureka上的默认描述信息
```

这样模拟集群就搭建号了，就可以把一个项目挂载到三个服务器上了



5.5 对比和Zookeeper区别

1. 回顾CAP原则

RDBMS (MySQL\Oracle\sqlServer) ===> ACID

NoSQL (Redis\MongoDB) ===> CAP

2. ACID是什么?

- A (Atomicity) 原子性
- C (Consistency) 一致性
- I (Isolation) 隔离性

- D (Durability) 持久性

3. 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

4. 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，分区容错的系统，通常性能不是特别高
    - AP：满足可用性，分区容错的系统，通常可能对一致性要求低一些

5. 作为分布式服务注册中心，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再分布式系统中是必须要保证的，因此我们只能再A和C之间进行权衡。

- Zookeeper 保证的是 CP —> 满足一致性，分区容错的系统，通常性能不是特别高
- Eureka 保证的是 AP —> 满足可用性，分区容错的系统，通常可能对一致性要求低一些

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，但不能接收服务直接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zookeeper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zookeeper集群是不可用的，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zookeeper集群失去master节点是较大概率发生的事件，虽然服务最终能够恢复，但是，漫长的选举时间导致注册长期不可用，是不可容忍的。

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还有之中自我保护机制，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：

- Eureka不在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
- 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)
- 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

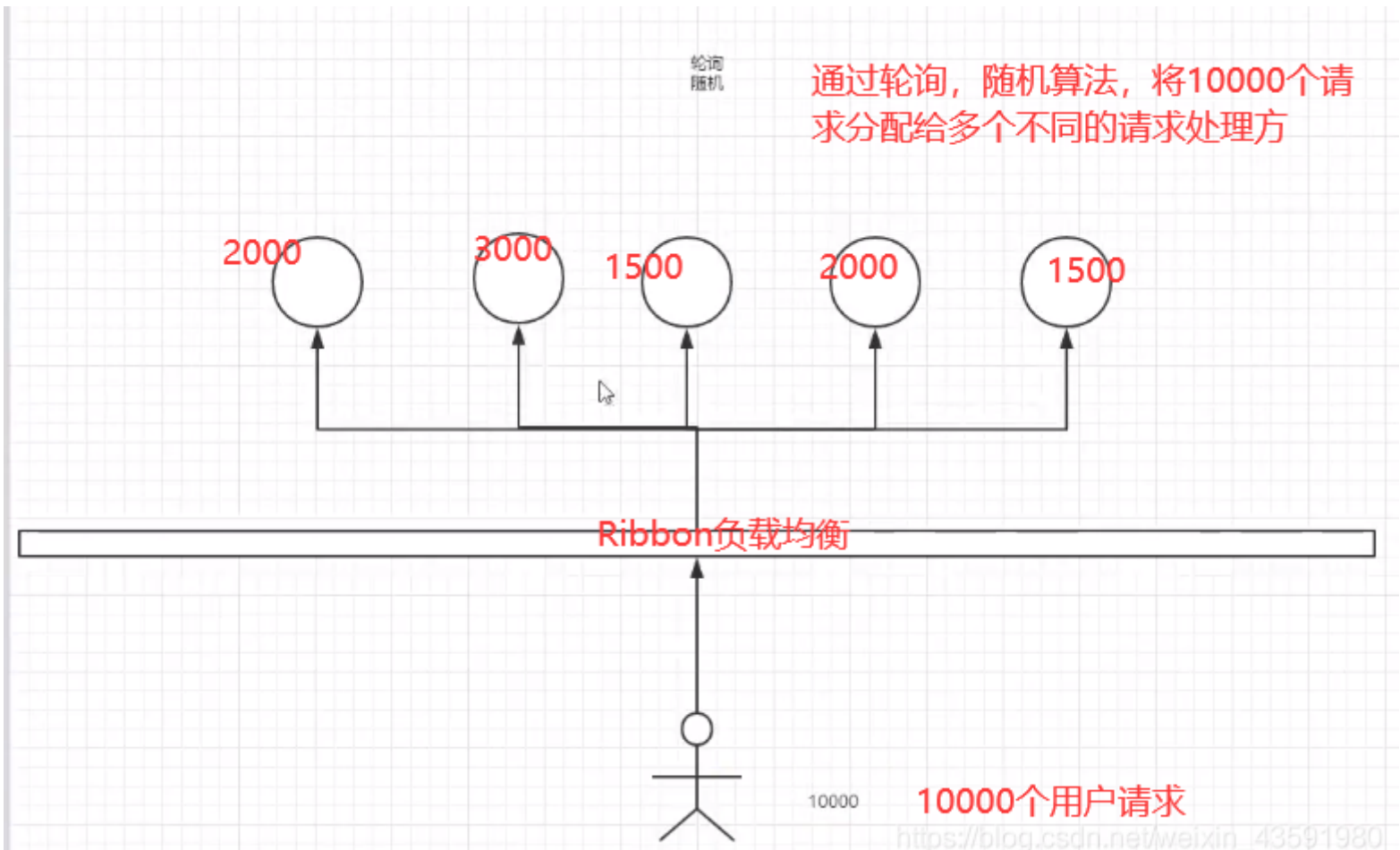
6. Ribbon：负载均衡(基于客户端)

6.1 负载均衡以及Ribbon

> 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。简单的说，就是在配置文件中列出 LoadBalancer (简称LB：负载均衡) 后面所有的及其，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 (LoadBalancer)，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
    - 即在服务的提供方和消费方之间使用独立的LB设施，如**Nginx(反向代理服务器)**，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！

6.2 集成Ribbon

springcloud-consumer-dept-80向pom.xml中添加Ribbon和Eureka依赖

```
<!--Ribbon-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<!--Eureka: Ribbon需要从Eureka服务中心获取要拿什么-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
```

在application.yml文件中配置Eureka

```
# Eureka配置
eureka:
  client:
    register-with-eureka: false # 不向 Eureka注册自己
    service-url: # 从三个注册中心中随机取一个去访问
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

主启动类加上@EnableEurekaClient注解，开启Eureka

```
//Ribbon 和 Eureka 整合以后，客户端可以直接调用，不用关心IP地址和端口号
@SpringBootApplication
@EnableEurekaClient //开启Eureka 客户端
public class DeptConsumer_80 {

    public static void main(String[] args) {

        SpringApplication.run(DeptConsumer_80.class, args);
    }
}
```

自定义Spring配置类：ConfigBean.java 配置负载均衡实现RestTemplate

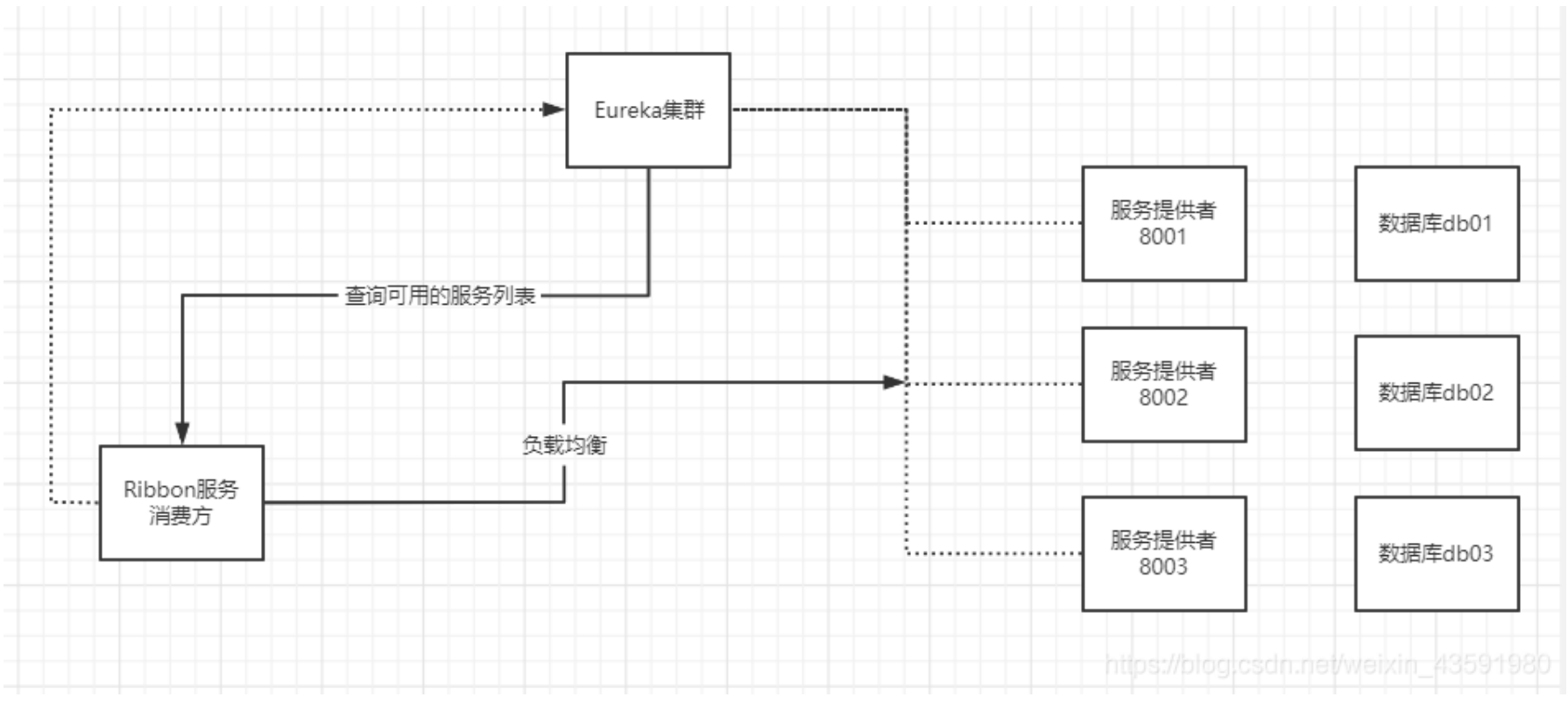
```
@Configuration
public class ConfigBean {
    //@Configuration -- spring  applicationContext.xml

    @LoadBalanced //配置负载均衡实现RestTemplate
    @Bean
    public RestTemplate getRestTemplate() {

        return new RestTemplate();
    }
}
```

修改conroller：DeptConsumerController.java

```
//Ribbon:我们这里的地址，应该是一个变量，通过服务名来访问
//private static final String REST_URL_PREFIX = "http://localhost:8001";
private static final String REST_URL_PREFIX = "http://SPRINGCLOUD-PROVIDER-DEPT";
```

6.3 使用Ribbon实现负载均衡

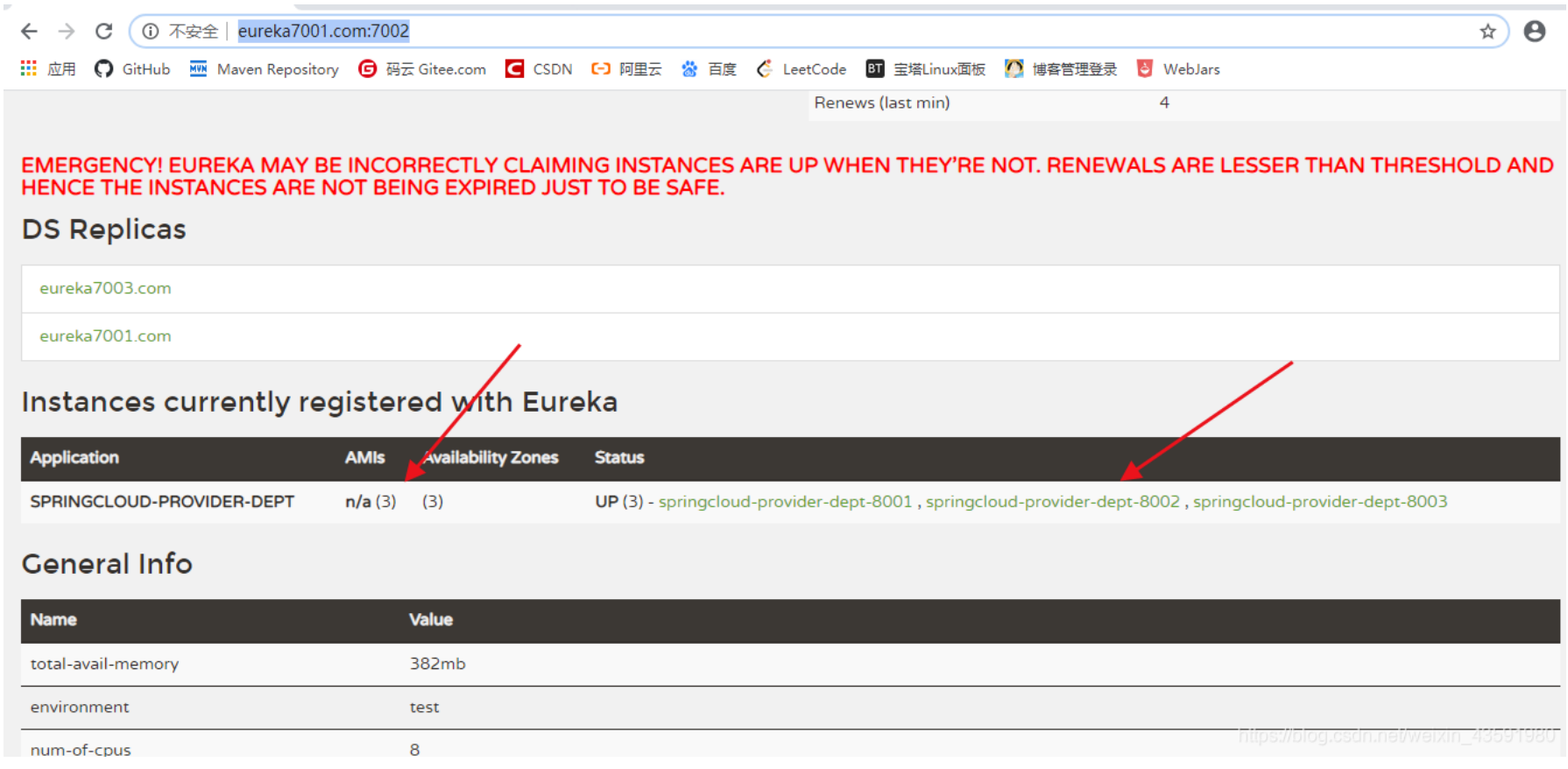
流程图：



1.新建两个服务提供者Moudle：springcloud-provider-dept-8003、springcloud-provider-dept-8002

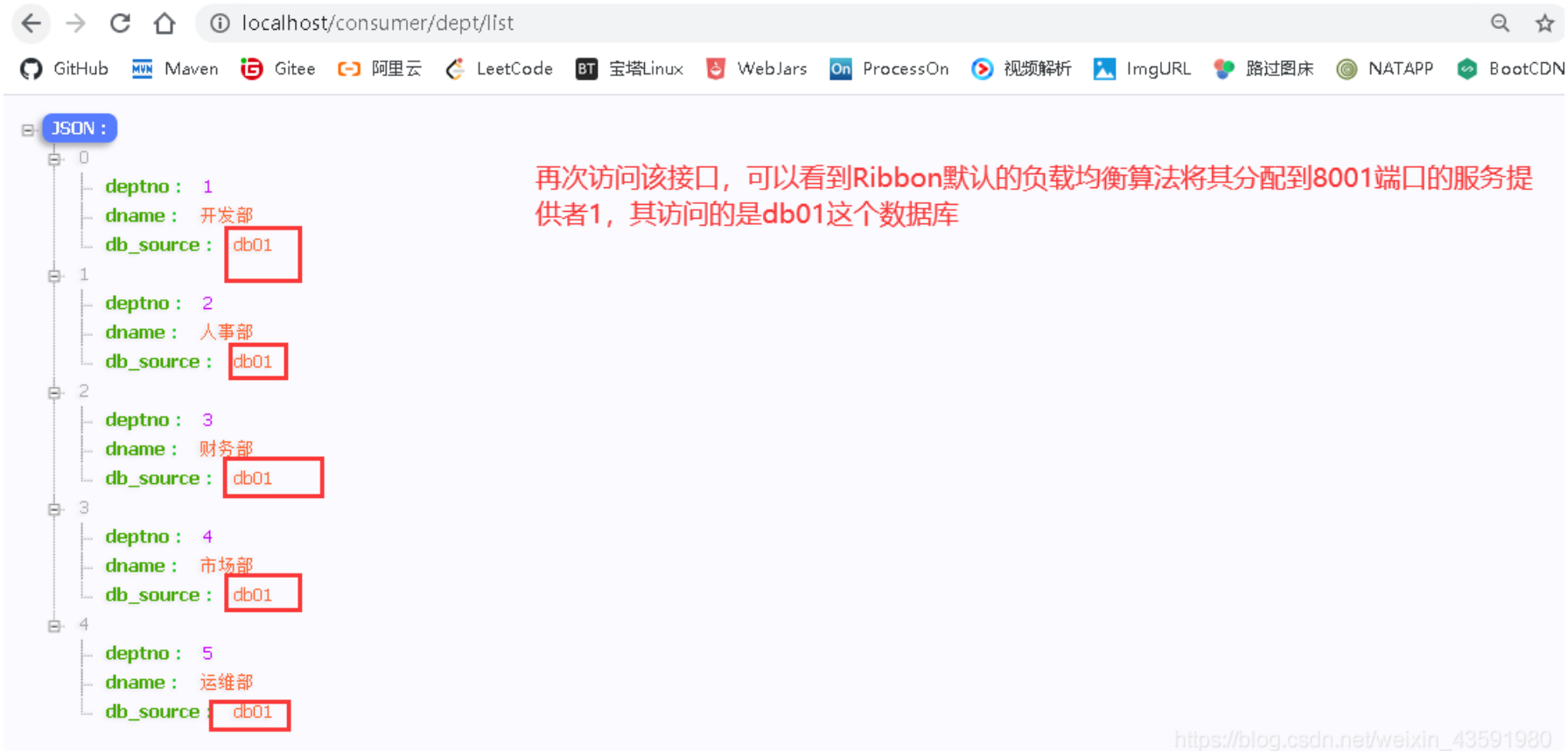
2.参照springcloud-provider-dept-8001 依次为另外两个Moudle添加pom.xml依赖 、resourece下的mybatis和application.yml配置，Java代码

3.启动所有服务测试(根据自身电脑配置决定启动服务的个数)，访问http://eureka7001.com:7002/查看结果



测试访问http://localhost/consumer/dept/list 这时候随机访问的是服务提供者8003

再次访问http://localhost/consumer/dept/list这时候随机的是服务提供者8001



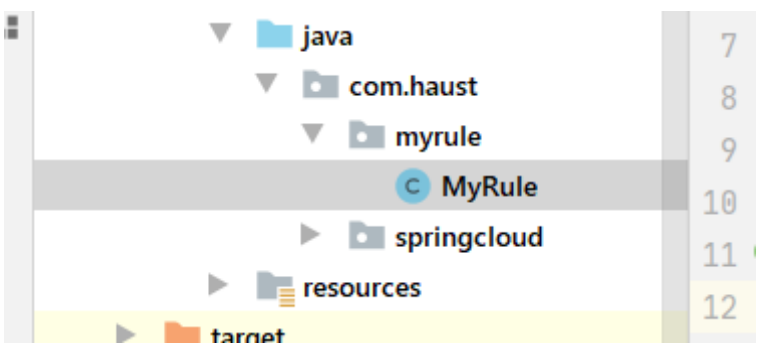
以上这种**每次访问http://localhost/consumer/dept/list随机访问集群中某个服务提供者，这种情况叫做轮询**，轮询算法在SpringCloud中可以自定义。

如何切换或者自定义规则呢?

在springcloud-provider-dept-80模块下的ConfigBean中进行配置，切换使用不同的规则

```
@Configuration
public class ConfigBean {
    //@Configuration -- spring  applicationContext.xml

    /**
     * IRule:
     * RoundRobinRule 轮询策略
     * RandomRule 随机策略
     * AvailabilityFilteringRule ： 会先过滤掉，跳闸，访问故障的服务~，对剩下的进行轮询~
     * RetryRule ： 会先按照轮询获取服务~，如果服务获取失败，则会在指定的时间内进行，重试
     */
    @Bean
    public IRule myRule() {

        return new RandomRule();//使用随机策略
        //return new RoundRobinRule();//使用轮询策略
        //return new AvailabilityFilteringRule();//使用轮询策略
        //return new RetryRule();//使用轮询策略
    }
}
```

也可以自定义规则，在myRule包下自定义一个配置类MyRule.java，注意：**该包不要和主启动类所在的包同级，要跟启动类所在包同级**:



MyRule.java

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/19/11:58
 * @Description: 自定义规则
 */
@Configuration
public class MyRule {


    @Bean
    public IRule myRule(){

        return new MyRandomRule();//默认是轮询RandomRule,现在自定义为自己的
    }
}
```

主启动类开启负载均衡并指定自定义的MyRule配置类

```java
//Ribbon 和 Eureka 整合以后，客户端可以直接调用，不用关心IP地址和端口号
@SpringBootApplication
@EnableEurekaClient
//在微服务启动的时候就能加载自定义的Ribbon类(自定义的规则会覆盖原有默认的规则)
@RibbonClient(name = "SPRINGCLOUD-PROVIDER-DEPT",configuration = MyRule.class)//开启负载均衡,并指定自定义的规则
public class DeptConsumer_80 {

    public static void main(String[] args) {

        SpringApplication.run(DeptConsumer_80.class, args);
    }
}
```

自定义的规则(这里我们参考Ribbon中默认的规则代码自己稍微改动)：MyRandomRule.java

```java
public class MyRandomRule extends AbstractLoadBalancerRule {


    /**
     * 每个服务访问5次则换下一个服务(总共3个服务)
     * <p>
     * total=0,默认=0,如果=5,指向下一个服务节点
     * index=0,默认=0,如果total=5,index+1
     */
    private int total = 0;//被调用的次数
    private int currentIndex = 0;//当前是谁在提供服务

    //@edu.umd.cs.findbugs.annotations.SuppressWarnings(value = "RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE")
    public Server choose(ILoadBalancer lb, Object key) {

        if (lb == null) {

            return null;
        }
        Server server = null;

        while (server == null) {

            if (Thread.interrupted()) {

                return null;
            }
            List<Server> upList = lb.getReachableServers();//获得当前活着的服务
            List<Server> allList = lb.getAllServers();//获取所有的服务

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

                /*
                 * No servers. End regardless of pass, because subsequent passes
                 * only get more restrictive.
                 */
                return null;
            }

            //int index = chooseRandomInt(serverCount);//生成区间随机数
            //server = upList.get(index);//从或活着的服务中,随机获取一个

            //====================自定义代码=========================

            if (total < 5) {

                server = upList.get(currentIndex);
                total++;
            } else {

                total = 0;
                currentIndex++;
                if (currentIndex > upList.size()) {

                    currentIndex = 0;
                }
                server = upList.get(currentIndex);//从活着的服务中,获取指定的服务来进行操作
            }

            //=======================================================

            if (server == null) {

                /*
                 * The only time this should happen is if the server list were
```

```
                 * somehow trimmed. This is a transient condition. Retry after
                 * yielding.
                 */
                Thread.yield();
                continue;
            }
            if (server.isAlive()) {

                return (server);
            }
            // Shouldn't actually happen.. but must be transient or a bug.
            server = null;
            Thread.yield();
        }
        return server;
    }

    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) {

        // TODO Auto-generated method stub
    }
}
```

7.Feign：负载均衡(基于服务端)

7.1 Feign简介

Feign是声明式Web Service客户端，它让微服务之间的调用变得更简单，类似controller调用service。SpringCloud集成了Ribbon和Eureka，可以使用Feigin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，主要是社区版，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

01. 微服务名字 【ribbon】
02. 接口和注解 【feign】

Feign能干什么?

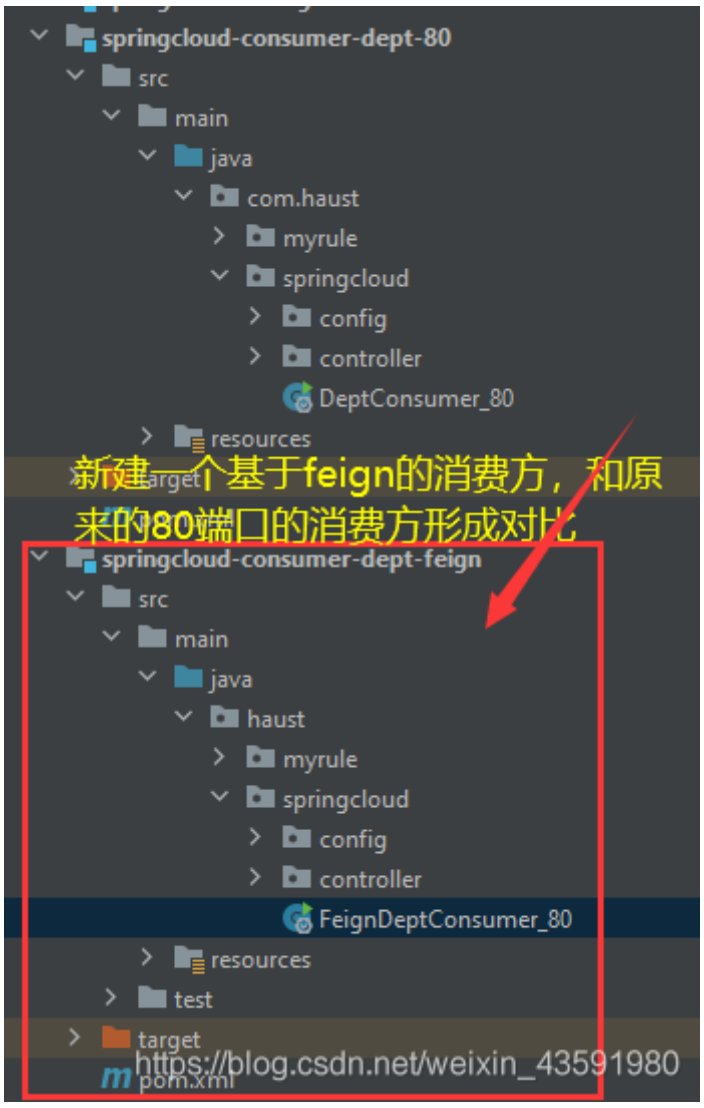
- Feign旨在使编写Java Http客户端变得更容易
- 前面在使用**Ribbon** + **RestTemplate**时，利用**RestTemplate**对Http请求的封装处理，形成了一套模板化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一个客户端类来包装这些依赖服务的调用。所以，**Feign**在此基础上做了进一步的封装，由他来帮助我们定义和实现依赖服务接口的定义，在Feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它 (类似以前Dao接口上标注Mapper注解，现在是一个微服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默认集成了Ribbon

- 利用**Ribbon**维护了MicroServiceCloud-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，优雅而简单的实现了服务调用。

7.2 Feign的使用步骤

01. 创建springcloud-consumer-fdept-feign模块



拷贝springcloud-consumer-dept-80模块下的pom.xml，resource，以及java代码到springcloud-consumer-feign模块，并添加feign依赖。

```
<!--Feign的依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
```

通过**Ribbon**实现：—原来的controller：**DeptConsumerController.java**

```
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:44
 * @Description:
 */
@RestController
public class DeptConsumerController {
```

```
    /**
     * 理解：消费者，不应该有service层~
     * RestTemplate .... 供我们直接调用就可以了！ 注册到Spring中
     * (地址：url, 实体：Map ,Class<T> responseType)
     * <p>
     * 提供多种便捷访问远程http服务的方法，简单的Restful服务模板~
     */
    @Autowired
    private RestTemplate restTemplate;

    /**
     * 服务提供方地址前缀
     * <p>
     * Ribbon:我们这里的地址，应该是一个变量，通过服务名来访问
     */
//    private static final String REST_URL_PREFIX = "http://localhost:8001";
    private static final String REST_URL_PREFIX = "http://SPRINGCLOUD-PROVIDER-DEPT";

    /**
     * 消费方添加部门信息
     * @param dept
     * @return
     */
    @RequestMapping("/consumer/dept/add")
    public boolean add(Dept dept) {

        // postForObject(服务提供方地址(接口),参数实体,返回类型.class)
        return restTemplate.postForObject(REST_URL_PREFIX + "/dept/add", dept, Boolean.class);
    }

    /**
     * 消费方根据id查询部门信息
     * @param id
     * @return
     */
    @RequestMapping("/consumer/dept/get/{id}")
    public Dept get(@PathVariable("id") Long id) {

        // getForObject(服务提供方地址(接口),返回类型.class)
        return restTemplate.getForObject(REST_URL_PREFIX + "/dept/get/" + id, Dept.class);
    }

    /**
     * 消费方查询部门信息列表
     * @return
     */
    @RequestMapping("/consumer/dept/list")
    public List<Dept> list() {

        return restTemplate.getForObject(REST_URL_PREFIX + "/dept/list", List.class);
    }
}
```

通过**Feign**实现：—改造后controller：**DeptConsumerController.java**

```
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:44
 * @Description:
 */
@RestController
public class DeptConsumerController {


```

```
    @Autowired
    private DeptClientService deptClientService;

    /**
     * 消费方添加部门信息
     * @param dept
     * @return
     */
    @RequestMapping("/consumer/dept/add")
    public boolean add(Dept dept) {

        return deptClientService.addDept(dept);
    }

    /**
     * 消费方根据id查询部门信息
     * @param id
     * @return
     */
    @RequestMapping("/consumer/dept/get/{id}")
    public Dept get(@PathVariable("id") Long id) {

        return deptClientService.queryById(id);
    }

    /**
     * 消费方查询部门信息列表
     * @return
     */
    @RequestMapping("/consumer/dept/list")
    public List<Dept> list() {

        return deptClientService.queryAll();
    }
}
```

Feign和**Ribbon**二者对比，前者显现出面向接口编程特点，代码看起来更清爽，而且**Feign**调用方式更符合我们之前在做SSM或者**SprngBoot**项目时，**Controller**层调用**Service**层的编程习惯！

主配置类：

```
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:47
 * @Description:
 */
@SpringBootApplication
@EnableEurekaClient
// feign客户端注解,并指定要扫描的包以及配置接口DeptClientService
@EnableFeignClients(basePackages = {
      "com.haust.springcloud"})
// 切记不要加这个注解，不然会出现404访问不到
//@ComponentScan("com.haust.springcloud")
public class FeignDeptConsumer_80 {

    public static void main(String[] args) {

        SpringApplication.run(FeignDeptConsumer_80.class, args);
    }
}
```

01. 改造springcloud-api模块

    pom.xml添加feign依赖

```xml
<!--Feign的依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
```

新建service包，并新建DeptClientService.java接口，

```
// @FeignClient:微服务客户端注解,value:指定微服务的名字,这样就可以使Feign客户端直接找到对应的微服务
@FeignClient(value = “SPRINGCLOUD-PROVIDER-DEPT”)
public interface DeptClientService {
```

```java
    @GetMapping("/dept/get/{id}")
    public Dept queryById(@PathVariable("id") Long id);

    @GetMapping("/dept/list")
    public Dept queryAll();

    @GetMapping("/dept/add")
    public Dept addDept(Dept dept);
}
```

7.3 Feign和Ribbon如何选择?

根据个人习惯而定，如果喜欢REST风格使用Ribbon；如果喜欢社区版的面向接口风格使用Feign.

Feign 本质上也是实现了 Ribbon，只不过后者是在调用方式上，为了满足一些开发者习惯的接口调用习惯!

下面我们关闭springcloud-consumer-dept-80 这个服务消费方，换用springcloud-consumer-dept-feign(端口还是80) 来代替：(依然可以正常访问，就是调用方式相比于Ribbon变化了)

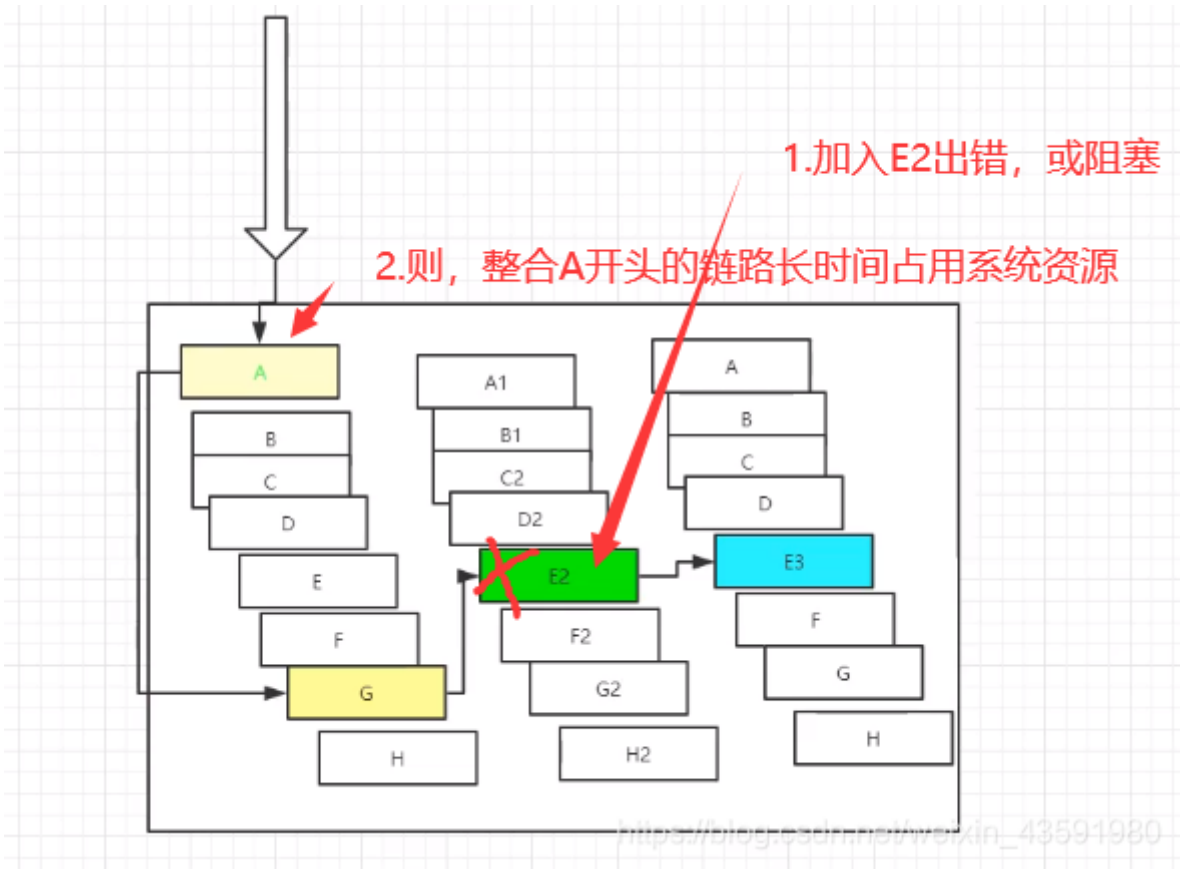
8. Hystrix：服务熔断

> 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失败！

8.1 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”。



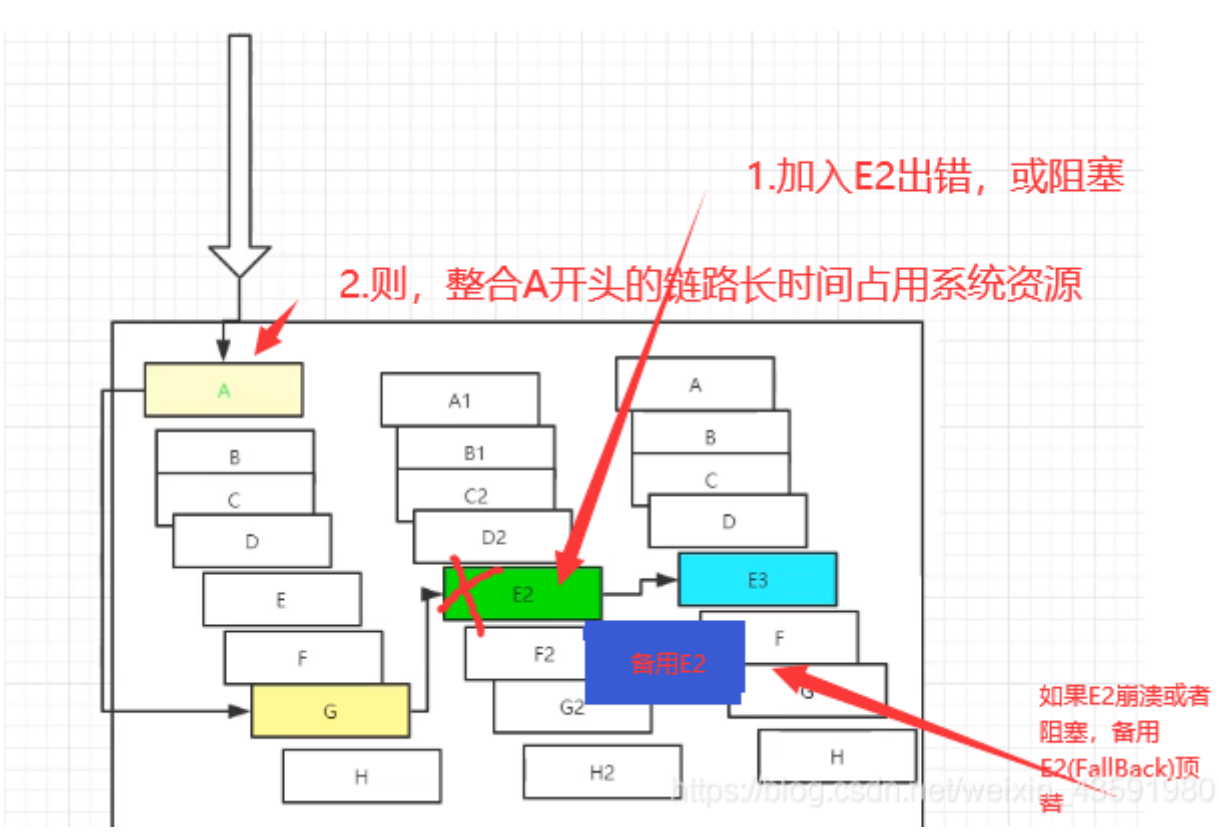
对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几十秒内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障，**这些都表示需要对故障和延迟进行隔离和管理，以达到单个依赖关系的失败而不影响整个应用程序或系统运行。**

我们需要，**弃车保帅**!

8.2 什么是Hystrix?

Hystrix是一个应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，不会导致整个体系服务失败，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。

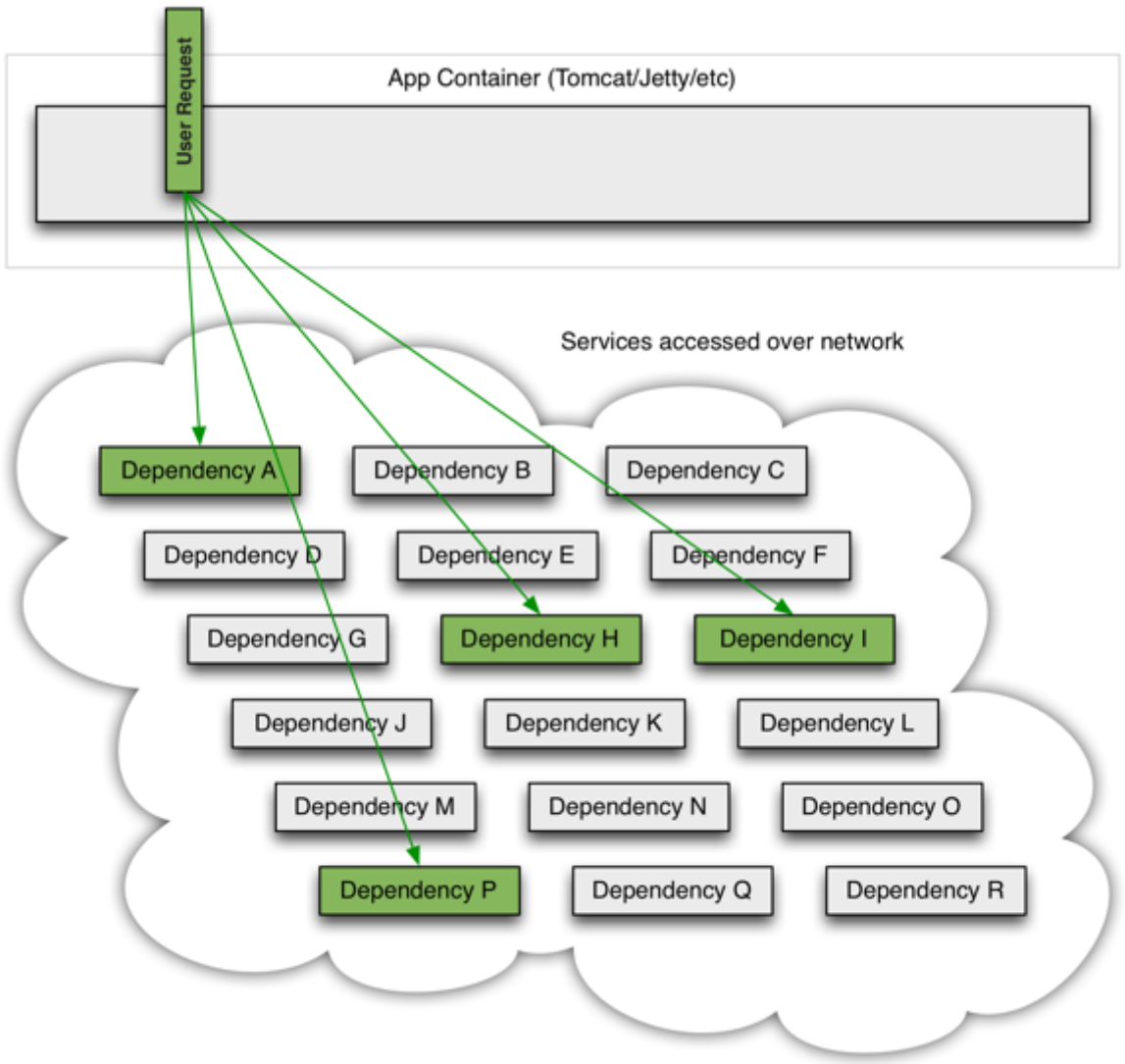


8.3 Hystrix能干嘛?

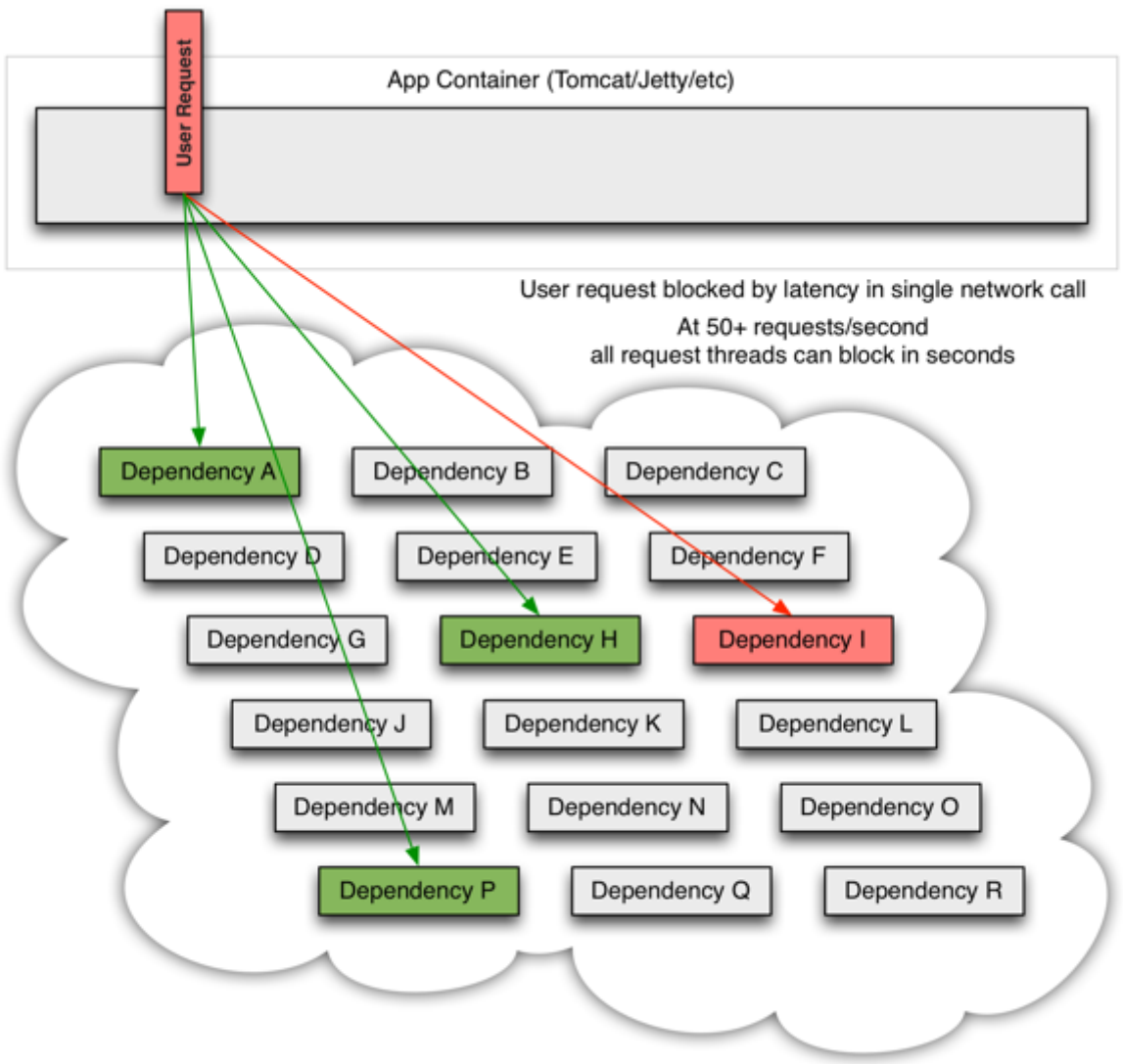
- 服务降级

- 服务熔断
- 服务限流
- 接近实时的监控
- ...

当一切正常时，请求流可以如下所示：

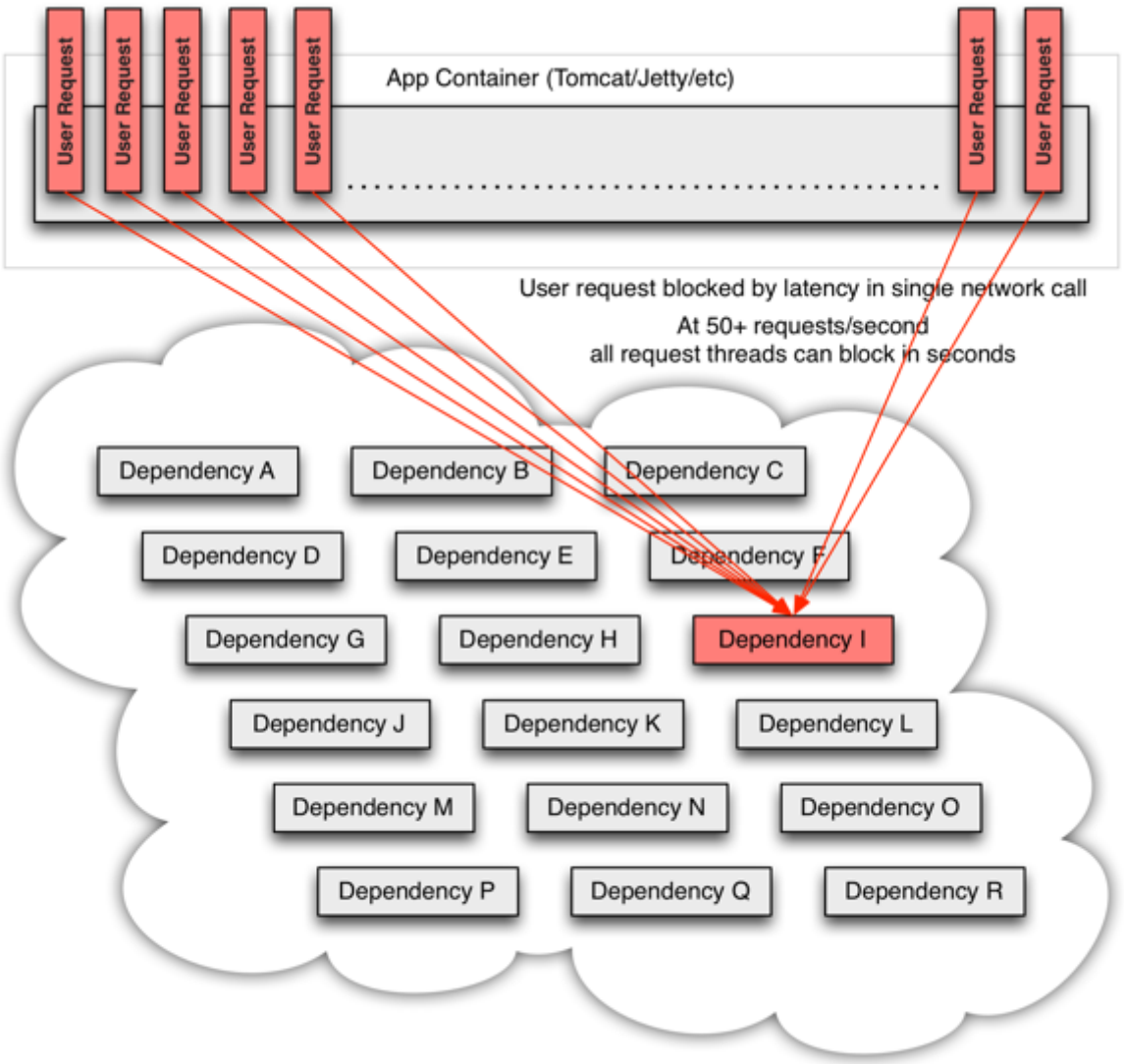


当许多后端系统中有一个潜在阻塞服务时，它可以阻止整个用户请求：

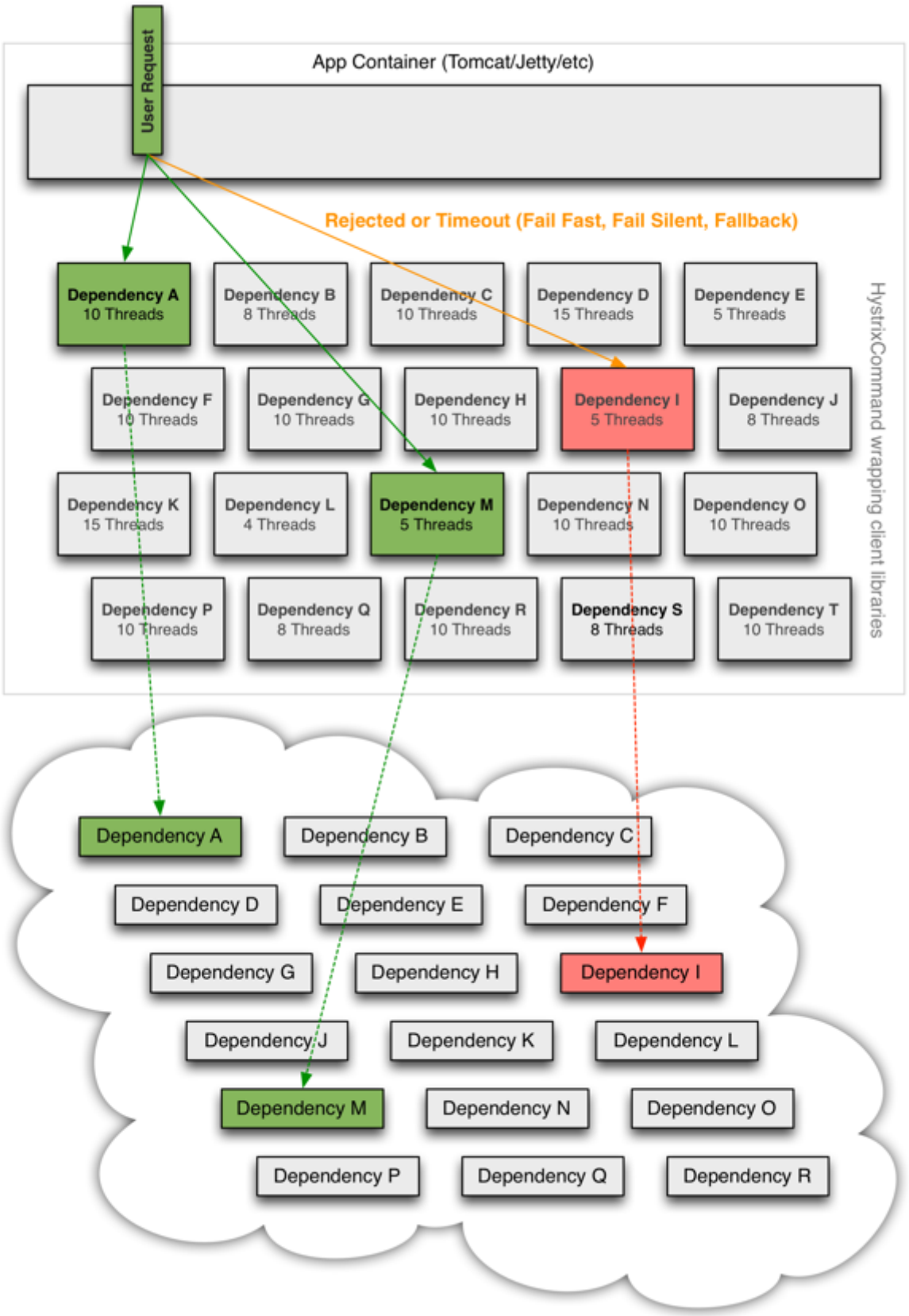


随着大容量通信量的增加，单个后端依赖项的潜在性会导致所有服务器上的所有资源在几秒钟内饱和。

应用程序中通过网络或客户端库可能导致网络请求的每个点都是潜在故障的来源。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，从而备份队列、线程和其他系统资源，从而导致更多跨系统的级联故障。



当使用**Hystrix**包装每个基础依赖项时，上面的图表中所示的体系结构会发生类似于以下关系图的变化。**每个依赖项是相互隔离的**，限制在延迟发生时它可以填充的资源中，并包含在回退逻辑中，该逻辑决定在依赖项中发生任何类型的故障时要做出什么样的响应：



官网资料：https://github.com/Netflix/Hystrix/wiki

8.4 服务熔断

什么是服务熔断?

熔断机制是赌赢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。**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阀值缺省是**5秒内20次调用失败，就会启动熔断机制。**熔断机制的注解是： `@HystrixCommand` 。

服务熔断解决如下问题：

- 当所依赖的对象不稳定时，能够起到快速失败的目的；
- 快速失败后，能够根据一定的算法动态试探所依赖对象是否恢复。

入门案例

新建springcloud-provider-dept-hystrix-8001模块并拷贝springcloud-provider-dept–8001内的**pom.xml、resource**和Java代码进行初始化并调整。

导入hystrix依赖

```xml
<!--导入Hystrix依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
```

调整yml配置文件

```yaml
server:
  port: 8001

# mybatis配置
mybatis:
  # springcloud-api 模块下的pojo包
  type-aliases-package: com.haust.springcloud.pojo
  # 本模块下的mybatis-config.xml核心配置文件类路径
  config-location: classpath:mybatis/mybatis-config.xml
  # 本模块下的mapper配置文件类路径
  mapper-locations: classpath:mybatis/mapper/*.xml

# spring配置
spring:
  application:
    #项目名
    name: springcloud-provider-dept
  datasource:
    # 德鲁伊数据源
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-8
    username: root
    password: root

# Eureka配置：配置服务注册中心地址
eureka:
  client:
    service-url:
      # 注册中心地址7001-7003
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
  instance:
    instance-id: springcloud-provider-dept-hystrix-8001 #修改Eureka上的默认描述信息
    prefer-ip-address: true #改为true后默认显示的是ip地址而不再是localhost

#info配置
info:
  app.name: haust-springcloud #项目的名称
  company.name: com.haust #公司的名称
```

prefer-ip-address: false:

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (2)	(2)	UP (2) - springcloud-provider-dept-8001 , springcloud-provider-dept-hystrix-8001

General Info

Name	Value
total-avail-memory	360mb
environment	test
num-of-cpus	8
current-memory-usage	264mb (73%)
	00:20

localhost:8001/actuator/info

prefer-ip-address: true:

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (2)	(2)	UP (2) - springcloud-provider-dept-8001 , springcloud-provider-dept-hystrix-8001

General Info

Name	Value
total-avail-memory	343mb
environment	test
num-of-cpus	8
current-memory-usage	84mb (24%)
	00:27

169.254.11.91:8001/actuator/info

修改controller

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:06
 * @Description: 提供Restful服务
 */
@RestController
public class DeptController {


    @Autowired
    private DeptService deptService;

    /**
     * 根据id查询部门信息
     * 如果根据id查询出现异常,则走hystrixGet这段备选代码
     * @param id
     * @return
     */
    @HystrixCommand(fallbackMethod = "hystrixGet")
    @RequestMapping("/dept/get/{id}")//根据id查询
    public Dept get(@PathVariable("id") Long id){

        Dept dept = deptService.queryById(id);
        if (dept==null){

            throw new RuntimeException("这个id=>"+id+",不存在该用户，或信息无法找到~");
        }
        return dept;
    }

    /**
     * 根据id查询备选方案(熔断)
     * @param id
     * @return
     */
    public Dept hystrixGet(@PathVariable("id") Long id){

        return new Dept().setDeptno(id)
                .setDname("这个id=>"+id+",没有对应的信息,null---@Hystrix~")
                .setDb_source("在MySQL中没有这个数据库");
    }
}
```

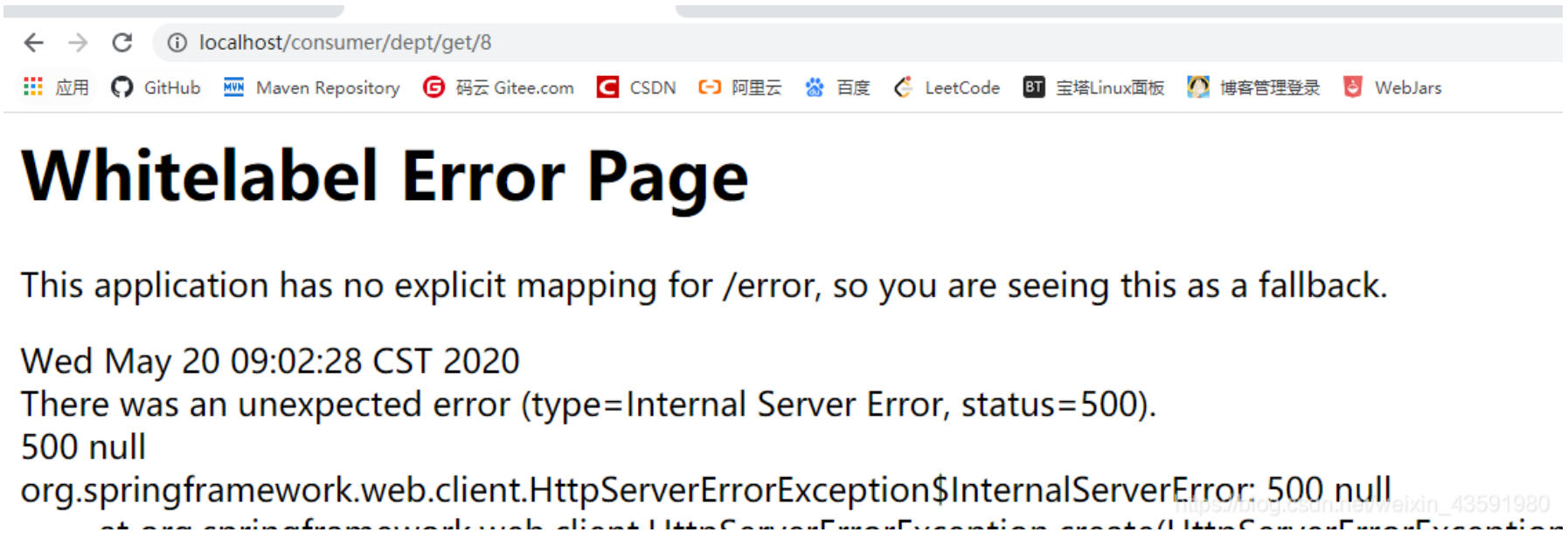
为主启动类添加对熔断的支持注解@EnableCircuitBreaker

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/17/22:09
 * @Description: 启动类
 */
@SpringBootApplication
@EnableEurekaClient // EnableEurekaClient 客户端的启动类，在服务启动后自动向注册中心注册服务
@EnableDiscoveryClient // 服务发现~
@EnableCircuitBreaker // 添加对熔断的支持注解
public class HystrixDeptProvider_8001 {

    public static void main(String[] args) {

        SpringApplication.run(HystrixDeptProvider_8001.class,args);
    }
}
```

测试:

使用熔断后，当访问一个不存在的id时，前台页展示数据如下:

而不适用熔断的springcloud-provider-dept–8001模块访问相同地址会出现下面状况:



因此，**为了避免因某个微服务后台出现异常或错误而导致整个应用或网页报错，使用熔断是必要的**

8.5 服务降级

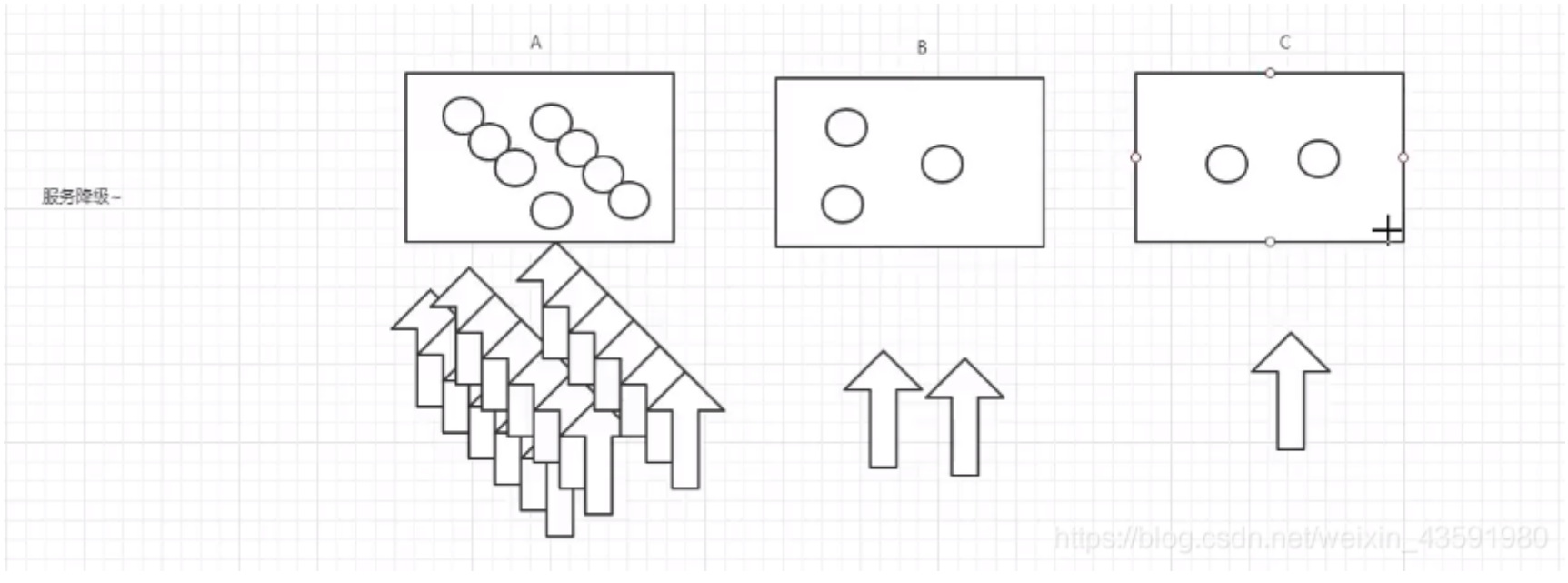
什么是服务降级?

服务降级是指 当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理，或换种简单的方式处理，从而释放服务器资源以保证核心业务正常运作或高效运作。说白了，**就是尽可能的把系统资源让给优先级高的服务**。

资源有限，而请求是无限的。如果在并发高峰期，不做服务降级处理，一方面肯定会影响整体服务的性能，严重的话可能会导致宕机某些重要的服务不可用。所以，一般在高峰期，为了保证核心功能服务的可用性，都要对某些服务降级处理。比如当双11活动时，把交易无关的服务统统降级，如查看蚂蚁深林，查看历史订单等等。

服务降级主要用于什么场景呢？当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，可以将一些 不重要 或 不紧急 的服务或任务进行服务的 延迟使用 或 暂停使用。

降级的方式可以根据业务来，可以延迟服务，比如延迟给用户增加积分，只是放到一个缓存中，等服务平稳之后再执行 ； 或者在粒度范围内关闭服务，比如关闭相关文章的推荐。



由上图可得，**当某一时间内服务A的访问量暴增，而B和C的访问量较少，为了缓解A服务的压力，这时候需要B和C暂时关闭一些服务功能，去承担A的部分服务，从而为A分担压力，叫做服务降级。**

服务降级需要考虑的问题

- 1）那些服务是核心服务，哪些服务是非核心服务
- 2）那些服务可以支持降级，那些服务不能支持降级，降级策略是什么
- 3）除服务降级之外是否存在更复杂的业务放通场景，策略是什么?

自动降级分类

1）超时降级：主要配置好超时时间和超时重试次数和机制，并使用异步机制探测回复情况

2）失败次数降级：主要是一些不稳定的api，当失败调用次数达到一定阀值自动降级，同样要使用异步机制探测回复情况

3）故障降级：比如要调用的远程服务挂掉了（网络故障、DNS故障、http服务返回错误的状态码、rpc服务抛出异常），则可以直接降级。降级后的处理方案有：默认值（比如库存服务挂了，返回默认现货）、兜底数据（比如广告挂了，返回提前准备好的一些静态页面）、缓存（之前暂存的一些缓存数据）

4）限流降级：秒杀或者抢购一些限购商品时，此时可能会因为访问量太大而导致系统崩溃，此时会使用限流来进行限制访问量，当达到限流阀值，后续请求会被降级；降级后的处理方案可以是：排队页面（将用户导流到排队页面等一会重试）、无货（直接告知用户没货了）、错误页（如活动太火爆了，稍后重试）。

入门案例

在springcloud-api模块下的service包中新建降级配置类DeptClientServiceFallBackFactory.java

```
/**
 * @Auther: csp1999
 * @Date: 2020/05/20/9:18
 * @Description: Hystrix服务降级 ~
 */
@Component
public class DeptClientServiceFallBackFactory implements FallbackFactory {


    @Override
    public DeptClientService create(Throwable cause) {

        return new DeptClientService() {

            @Override
            public Dept queryById(Long id) {

                return new Dept()
                        .setDeptno(id)
                        .setDname("id=>" + id + "没有对应的信息，客户端提供了降级的信息，这个服务现在已经被关闭")
                        .setDb_source("没有数据~");
            }
            @Override
            public List<Dept> queryAll() {

                return null;
            }

            @Override
            public Boolean addDept(Dept dept) {

                return false;
            }
        };
    }
}
```

在DeptClientService中指定降级配置类DeptClientServiceFallBackFactory

```java
@Component //注册到spring容器中
//@FeignClient:微服务客户端注解,value:指定微服务的名字,这样就可以使Feign客户端直接找到对应的微服务
@FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT",fallbackFactory =
DeptClientServiceFallBackFactory.class)//fallbackFactory指定降级配置类
public interface DeptClientService {


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

    @GetMapping("/dept/list")
    public List<Dept> queryAll();

    @GetMapping("/dept/add")
    public Boolean addDept(Dept dept);
}
```

在**springcloud-consumer-dept-feign**模块中开启降级：

```yaml
server:
  port: 80

# Eureka配置
eureka:
  client:
    register-with-eureka: false # 不向 Eureka注册自己
    service-url: # 从三个注册中心中随机取一个去访问
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

# 开启降级feign.hystrix
feign:
  hystrix:
    enabled: true
```

8.6 服务熔断和降级的区别

- **服务熔断—>服务端**：某个服务超时或异常，引起熔断~，类似于保险丝(自我熔断)
- **服务降级—>客户端**：从整体网站请求负载考虑，当某个服务熔断或者关闭之后，服务将不再被调用，此时在客户端，我们可以准备一个FallBackFactory，返回一个默认的值(缺省值)。会导致整体的服务下降，但是好歹能用，比直接挂掉强。
- 触发原因不太一样，服务熔断一般是某个服务（下游服务）故障引起，而服务降级一般是从整体负荷考虑；管理目标的层次不太一样，熔断其实是一个框架级的处理，每个微服务都需要（无层级之分），而降级一般需要对业务有层级之分（比如降级一般是从最外围服务开始）
- 实现方式不太一样，服务降级具有代码侵入性(由控制器完成/或自动降级)，熔断一般称为**自我熔断。**

熔断，降级，限流:

限流：限制并发的请求访问量，超过阈值则拒绝；

降级：服务分优先级，牺牲非核心服务（不可用），保证核心服务稳定；从整体负荷考虑；

熔断：依赖的下游服务故障触发熔断，避免引发本系统崩溃；系统自动执行和恢复

8.7 Dashboard 流监控

新建springcloud-consumer-hystrix-dashboard模块

添加依赖

```xml
<!--Hystrix依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<!--dashboard依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<!--Ribbon-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<!--Eureka-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
    <version>1.4.6.RELEASE</version>
</dependency>
<!--实体类+web-->
<dependency>
    <groupId>com.haust</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>
```

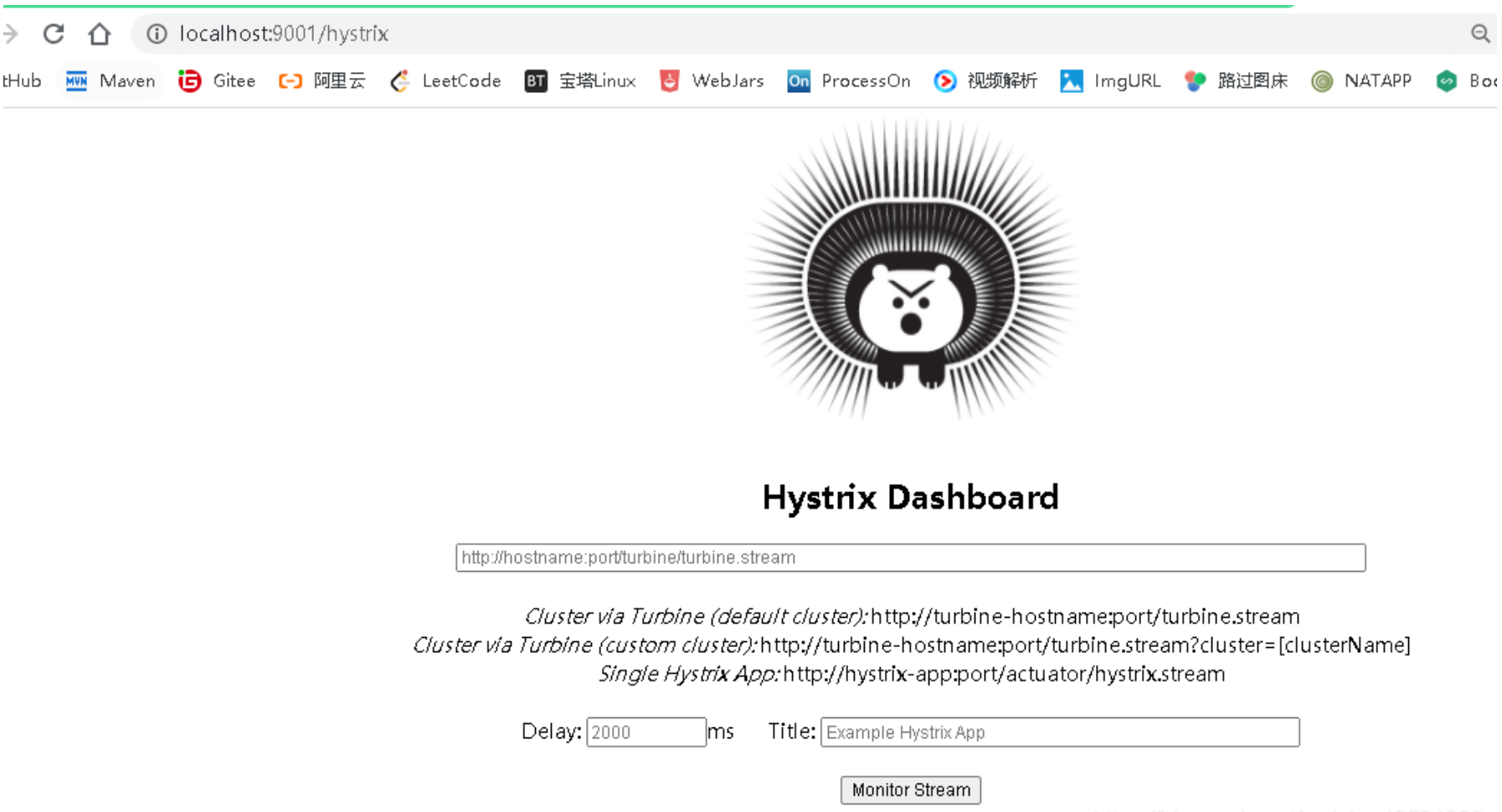
主启动类

```java
@SpringBootApplication
// 开启Dashboard
@EnableHystrixDashboard
public class DeptConsumerDashboard_9001 {

    public static void main(String[] args) {

        SpringApplication.run(DeptConsumerDashboard_9001.class,args);
    }
}
```

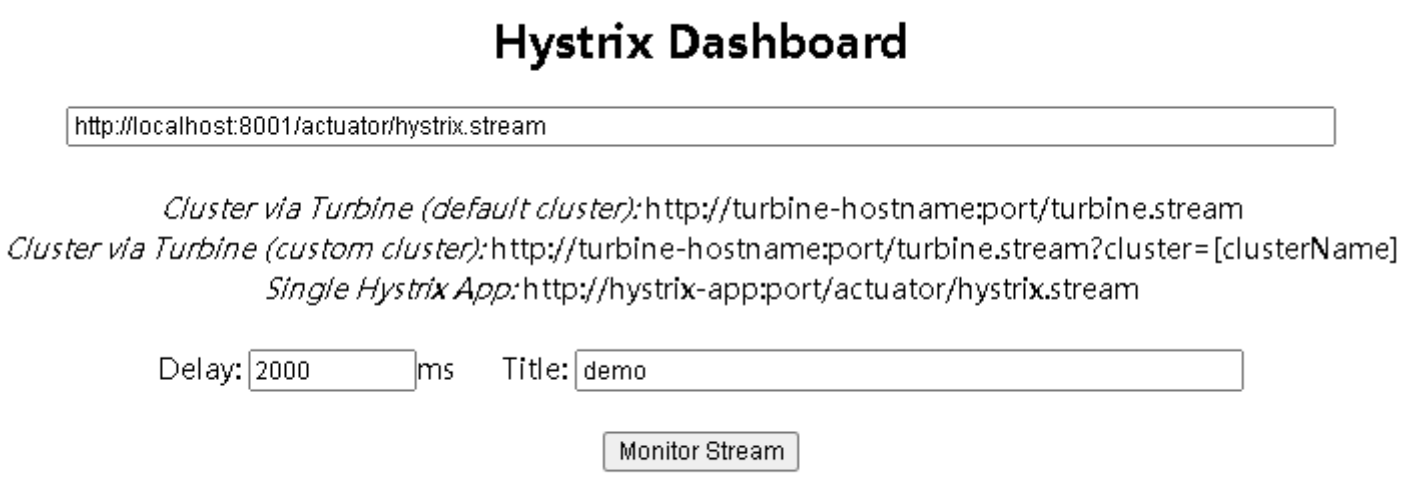
给springcloud-provider-dept-hystrix-8001模块下的主启动类添加如下代码,添加监控

```java
@SpringBootApplication
@EnableEurekaClient //EnableEurekaClient 客户端的启动类，在服务启动后自动向注册中心注册服务
public class DeptProvider_8001 {

    public static void main(String[] args) {

        SpringApplication.run(DeptProvider_8001.class,args);
    }

    //增加一个 Servlet
    @Bean
    public ServletRegistrationBean hystrixMetricsStreamServlet(){

        ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
HystrixMetricsStreamServlet());
        //访问该页面就是监控页面
        registrationBean.addUrlMappings("/actuator/hystrix.stream");

        return registrationBean;
    }
}
```

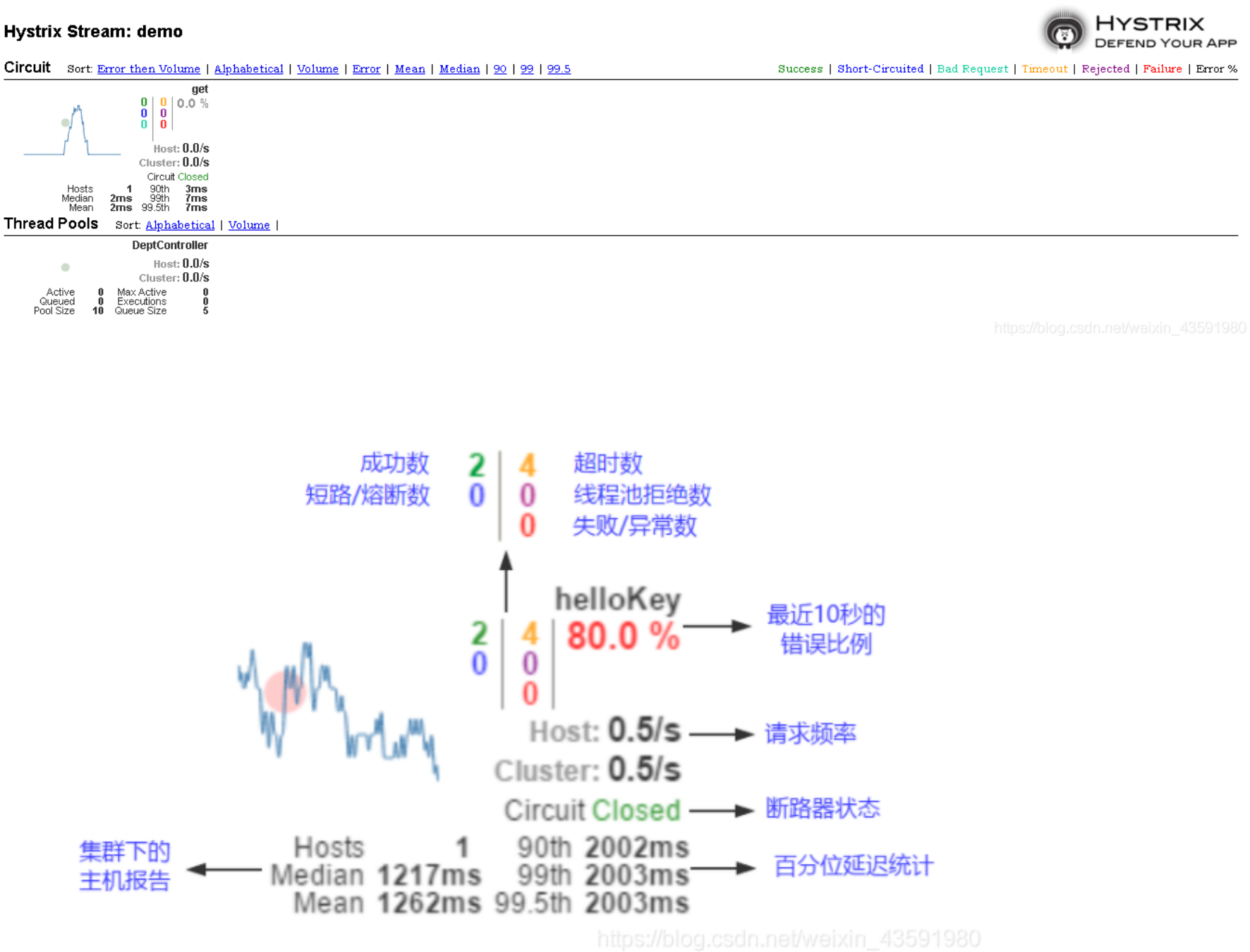
访问：http://localhost:9001/hystrix



进入监控页面：



效果如下图：



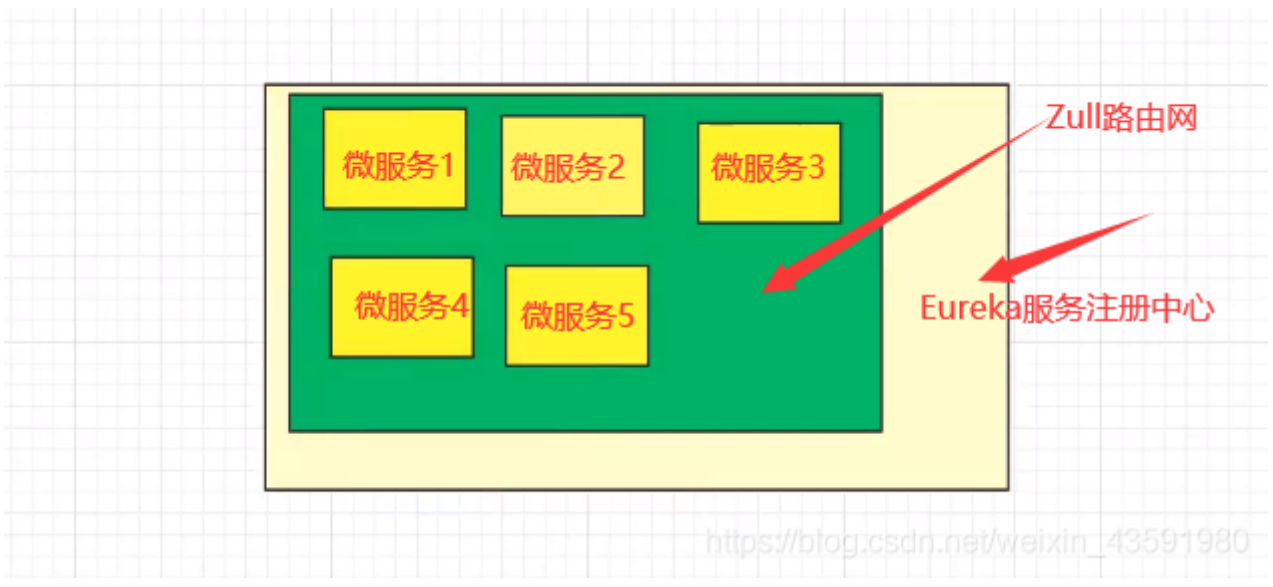
9. Zull路由网关

概述

> 什么是zuul?

Zull包含了对请求的**路由**(用来跳转的)和**过滤**两个最主要功能:

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

官方文档：https://github.com/Netflix/zuul/

入门案例

新建springcloud-zuul模块，并导入依赖

```
<dependencies>
    <!--导入zuul依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--Hystrix依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--dashboard依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboar</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--Ribbon-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-ribbon</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--Eureka-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
    <!--实体类+web-->
    <dependency>
        <groupId>com.haust</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>
```

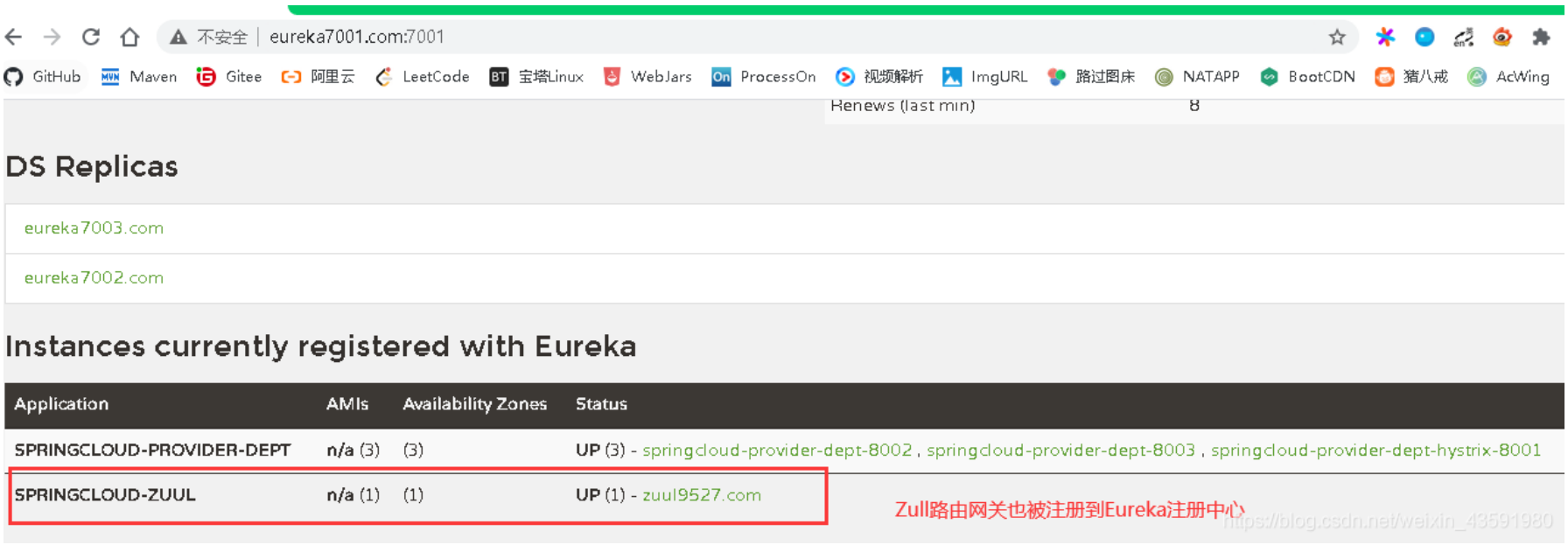
application.yml

```yaml
server:
  port: 9527

spring:
  application:
    name: springcloud-zuul #微服务名称

# eureka 注册中心配置
eureka:
  client:
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
  instance: #实例的id
    instance-id: zuul9527.com
    prefer-ip-address: true # 显示ip

info:
  app.name: haust.springcloud # 项目名称
  company.name: 河南科技大学西苑校区 # 公司名称

# zull 路由网关配置
zuul:
  # 路由相关配置
  # 原来访问路由 eg:http://www.cspStudy.com:9527/springcloud-provider-dept/dept/get/1
  # zull路由配置后访问路由 eg:http://www.cspstudy.com:9527/haust/mydept/dept/get/1
  routes:
    mydept.serviceId: springcloud-provider-dept # eureka注册中心的服务提供方路由名称
    mydept.path: /mydept/** # 将eureka注册中心的服务提供方路由名称 改为自定义路由名称
  # 不能再使用这个路径访问了，*： 忽略,隐藏全部的服务名称~
  ignored-services: "*"
  # 设置公共的前缀
  prefix: /haust
```

主启动类

```java
/**
 * @Auther: csp1999
 * @Date: 2020/05/20/20:53
 * @Description: Zull路由网关主启动类
 */
@SpringBootApplication
@EnableZuulProxy // 开启Zuul
public class ZuulApplication_9527 {


    public static void main(String[] args) {

        SpringApplication.run(ZuulApplication_9527.class,args);
    }
}
```

测试：



可以看出Zull路由网关被注册到Eureka注册中心中了！

上图是没有经过Zull路由网关配置时，服务接口访问的路由，可以看出直接用微服务(服务提供方)名称去访问，这样不安全，不能将微服务名称暴露！

所以经过Zull路由网关配置后，访问的路由为：

我们看到，微服务名称被替换并隐藏，换成了我们自定义的微服务名称mydept，同时加上了前缀haust，这样就做到了对路由fan访问的加密处理！

详情参考springcloud中文社区zuul组件 :https://www.springcloud.cc/spring-cloud-greenwich.html\#_router_and_filter_zuul

10. Spring Cloud Config 分布式配置

Dalston.RELEASE

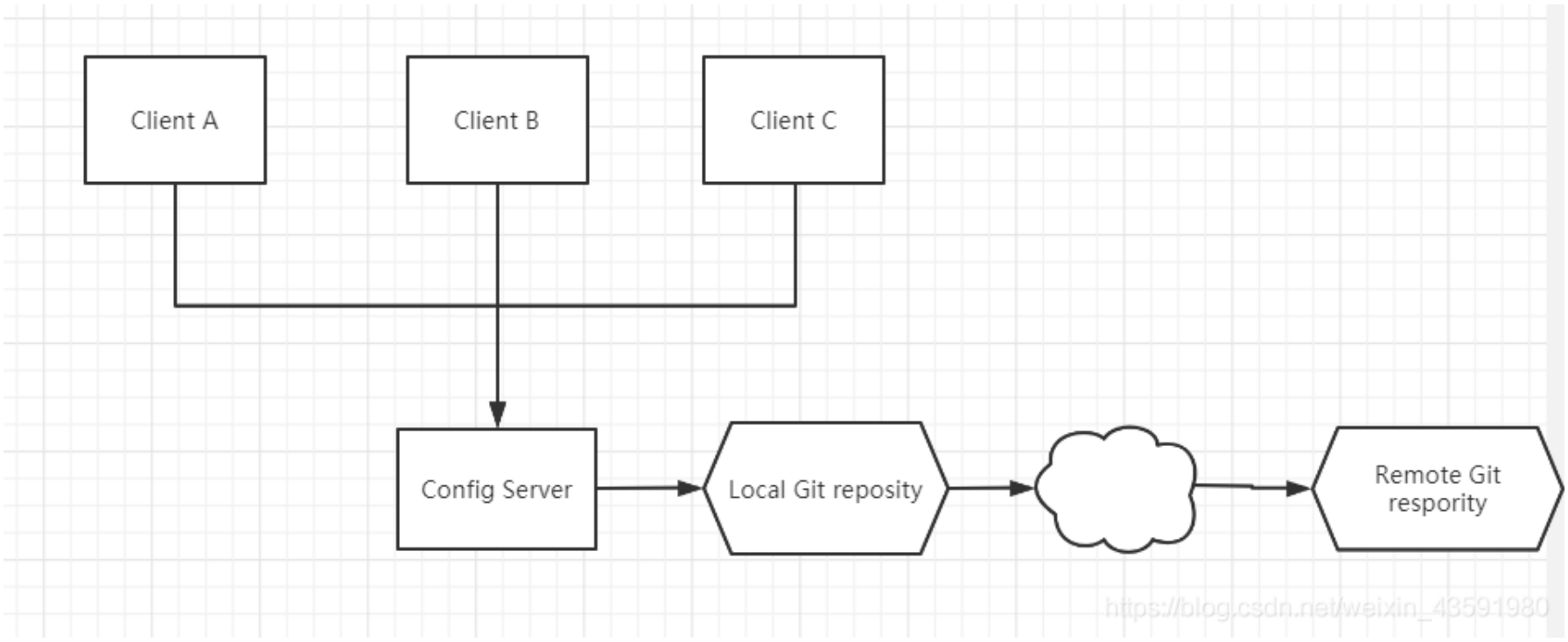
Spring Cloud Config为分布式系统中的外部配置提供服务器和客户端支持。使用Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念映射与Spring `Environment` 和 `PropertySource` 抽象相同，因此它们与Spring应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。很容易添加替代实现，并使用Spring配置将其插入。

概述

分布式系统面临的–配置文件问题

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务，由于每个服务都需要必要的配置信息才能运行，所以一套集中式的，动态的配置管理设施是必不可少的。spring cloud提供了configServer来解决这个问题，我们每一个微服务自己带着一个application.yml，那上百个的配置文件修改起来，令人头疼！

什么是SpringCloud config分布式配置中心?



spring cloud config 为微服务架构中的微服务提供集中化的外部支持，配置服务器为各个不同微服务应用的所有环节提供了一个**中心化的外部配置**。

spring cloud config 分为**服务端**和**客户端**两部分。

服务端也称为 **分布式配置中心**，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密，解密信息等访问接口。

客户端则是**通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息**。配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理。并且可用通过git客户端工具来方便的管理和访问配置内容。

spring cloud config 分布式配置中心能干嘛?

- 集中式管理配置文件
- 不同环境，不同配置，动态化的配置更新，分环境部署，比如 /dev /test /prod /beta /release
- 运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息
- 当配置发生变动时，服务不需要重启，即可感知到配置的变化，并应用新的配置
- 将配置信息以REST接口的形式暴露

spring cloud config 分布式配置中心与GitHub整合

由于spring cloud config 默认使用git来存储配置文件 (也有其他方式，比如自持SVN 和本地文件)，但是最推荐的还是git ，而且使用的是 http / https 访问的形式。

入门案例

服务端

新建springcloud-config-server-3344模块导入pom.xml依赖

```xml
<dependencies>
    <!--web-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--config-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
        <version>2.1.1.RELEASE</version>
    </dependency>
    <!--eureka-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
        <version>1.4.6.RELEASE</version>
    </dependency>
</dependencies>
```

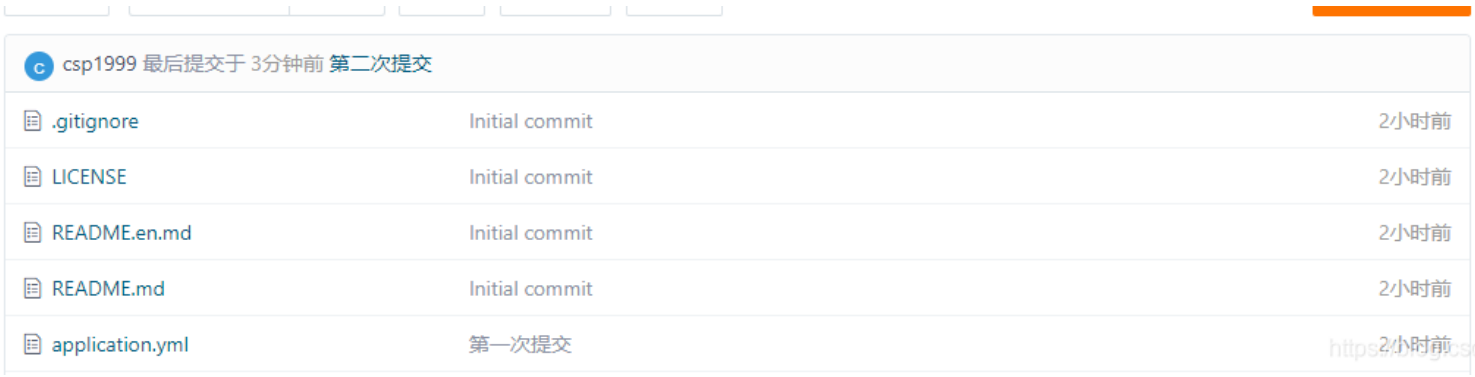
resource下创建application.yml配置文件，Spring Cloud Config服务器从git存储库（必须提供）为远程客户端提供配置：

```yaml
server:
  port: 3344

spring:
  application:
    name: springcloud-config-server
  # 连接码云远程仓库
  cloud:
    config:
      server:
        git:
          # 注意是https的而不是ssh
          uri: https://gitee.com/cao_shi_peng/springcloud-config.git
            # 通过 config-server可以连接到git，访问其中的资源以及配置~

# 不加这个配置会报Cannot execute request on any known server 这个错：连接Eureka服务端地址不对
# 或者直接注释掉eureka依赖 这里暂时用不到eureka
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

主启动类

```java
@EnableConfigServer // 开启spring cloud config server服务
@SpringBootApplication
public class Config_server_3344 {

    public static void main(String[] args) {

        SpringApplication.run(Config_server_3344.class,args);
    }
}
```

将本地git仓库springcloud-config文件夹下新建的application.yml提交到码云仓库：



定位资源的默认策略是克隆一个git仓库（在 `spring.cloud.config.server.git.uri` ），并使用它来初始化一个迷你 `SpringApplication` 。小应用程序的 `Environment` 用于枚举属性源并通过JSON端点发布。

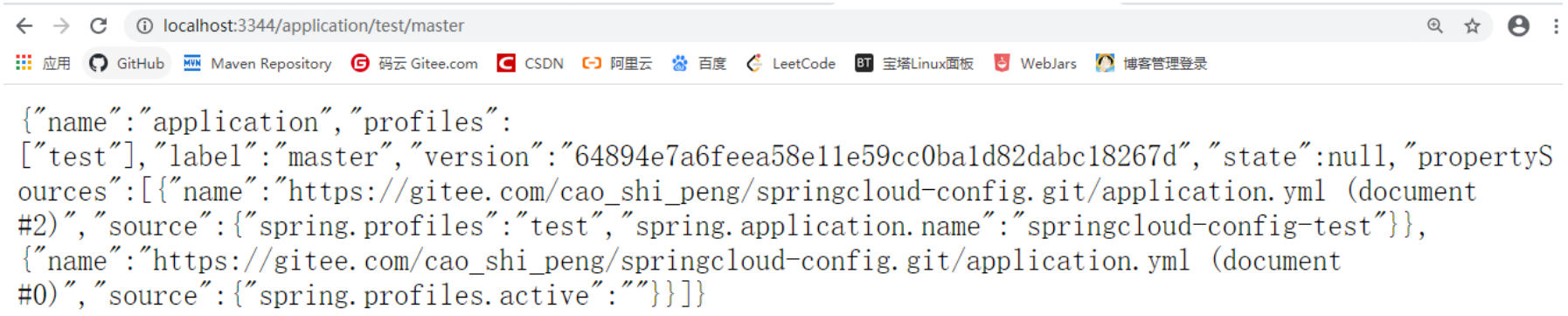
HTTP服务具有以下格式的资源：

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

其中“应用程序”作为 `SpringApplication` 中的 `spring.config.name` 注入（即常规的Spring Boot应用程序中通常是“应用程序”），“配置文件”是活动配置文件（或逗号分隔列表的属性），“label”是可选的git标签（默认为“master”）。

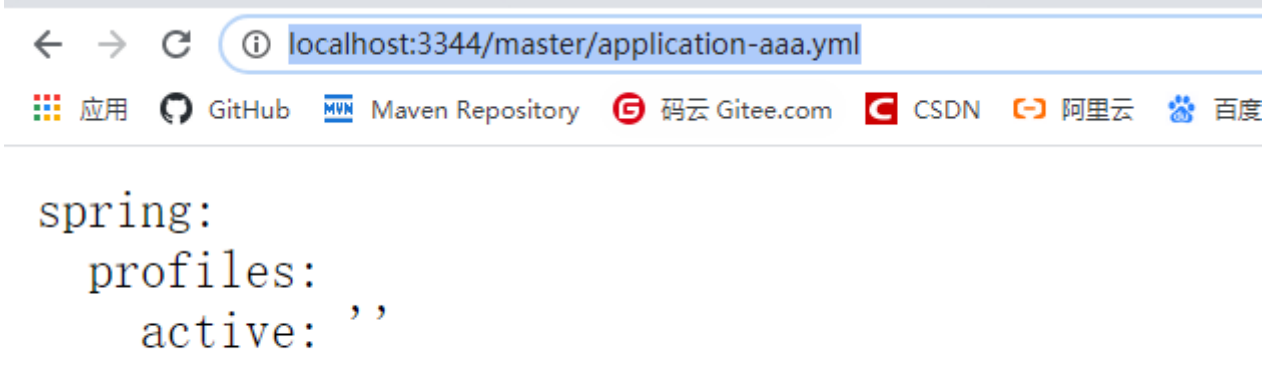
测试访问http://localhost:3344/application-dev.yml

测试访问 http://localhost:3344/application/test/master



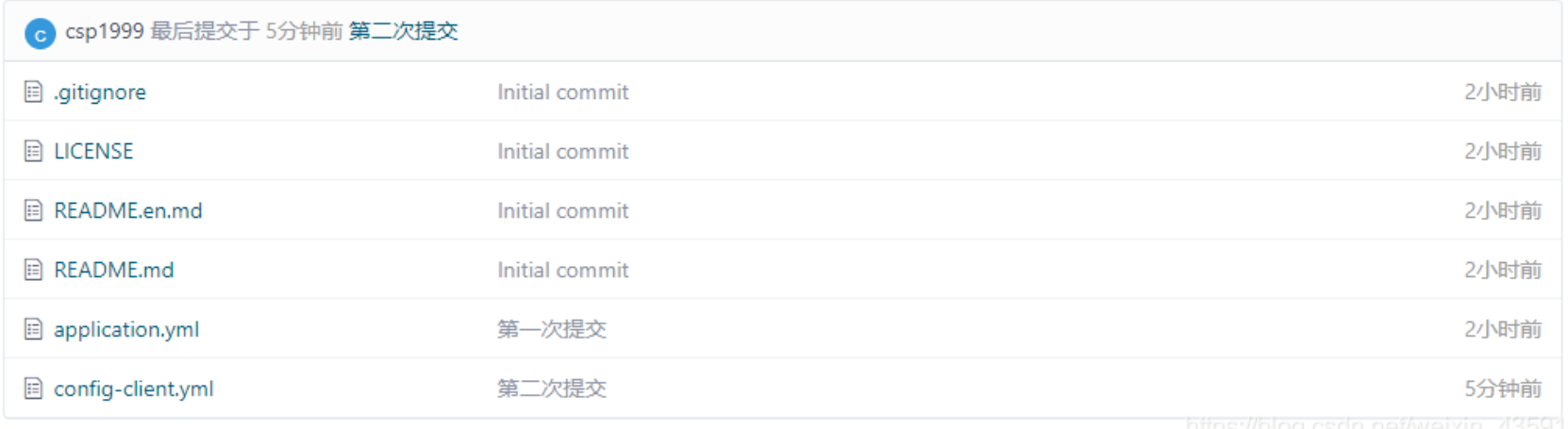
测试访问 http://localhost:3344/master/application-dev.yml

如果测试访问不存在的配置则不显示 如：http://localhost:3344/master/application-aaa.yml



客户端

将本地git仓库springcloud-config文件夹下新建的config-client.yml提交到码云仓库：



新建一个springcloud-config-client-3355模块，并导入依赖

```
<!--config-->
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-start -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>2.1.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

resources下创建application.yml和bootstrap.yml配置文件

TOP

bootstrap.yml 是系统级别的配置

```yaml
# 系统级别的配置
spring:
  cloud:
    config:
      name: config-client # 需要从git上读取的资源名称，不要后缀
      profile: dev
      label: master
      uri: http://localhost:3344
```

application.yml 是用户级别的配置

```yaml
# 用户级别的配置
spring:
  application:
    name: springcloud-config-client
```

创建controller包下的**ConfigClientController.java** 用于测试

```java
@RestController
public class ConfigClientController {


    @Value("${spring.application.name}")
    private String applicationName; //获取微服务名称

    @Value("${eureka.client.service-url.defaultZone}")
    private String eurekaServer; //获取Eureka服务

    @Value("${server.port}")
    private String port; //获取服务端的端口号


    @RequestMapping("/config")
    public String getConfig(){

        return "applicationName:"+applicationName +
         "eurekaServer:"+eurekaServer +
         "port:"+port;
    }
}
```

主启动类

```java
@SpringBootApplication
public class ConfigClient {

    public static void main(String[] args) {

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

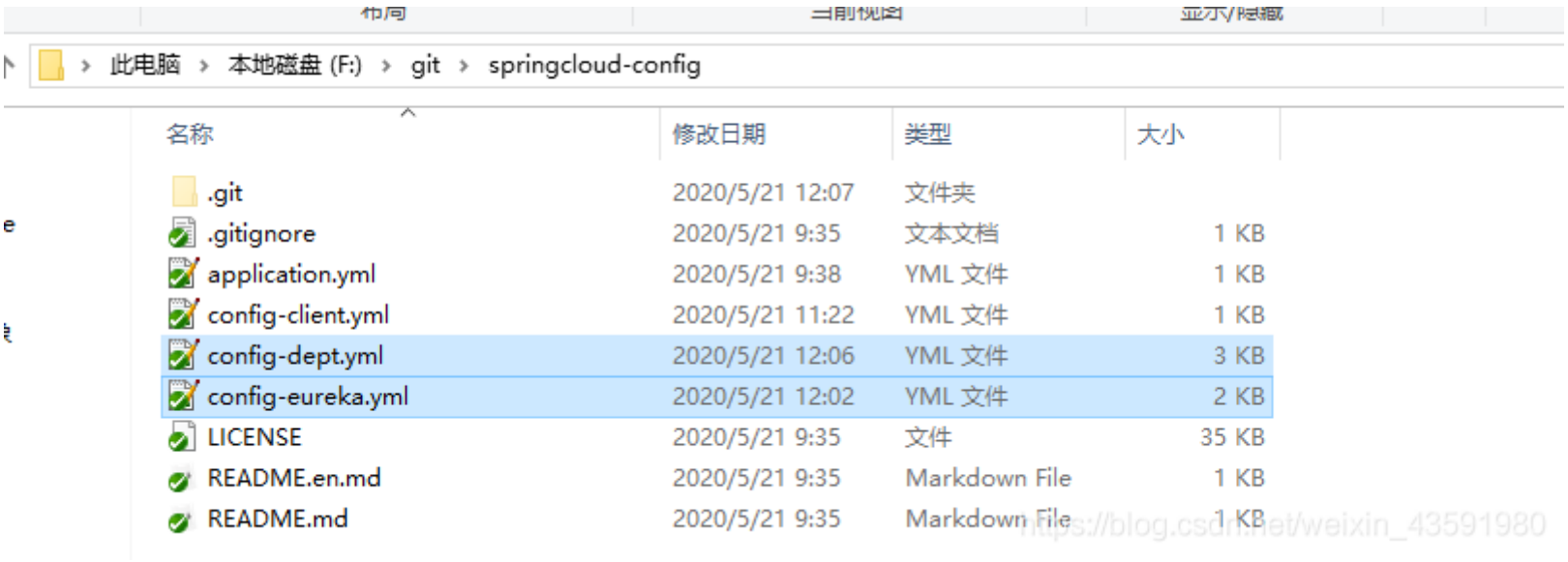
测试：

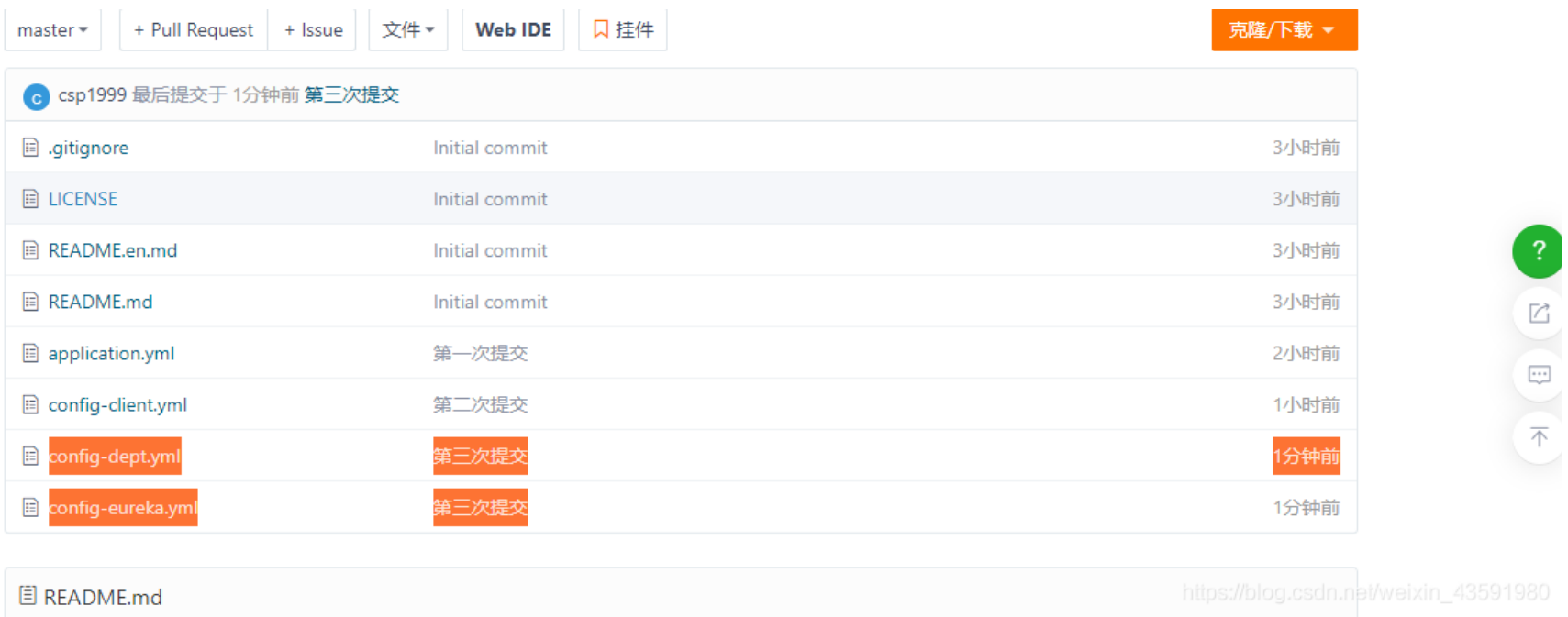
启动服务端Config_server_3344 再启动客户端ConfigClient

访问：http://localhost:8201/config/

小案例

本地新建config-dept.yml和config-eureka.yml并提交到码云仓库





这里配置文件内容不再列举直接到代码中看把。

新建springcloud-config-eureka-7001模块，并将原来的springcloud-eureka-7001模块下的内容拷贝的该模块。

1.清空该模块的application.yml配置，并新建bootstrap.yml连接远程配置

```yaml
spring:
  cloud:
    config:
      name: config-eureka # 仓库中的配置文件名称
      label: master
      profile: dev
      uri: http://localhost:3344
```

2.在pom.xml中添加spring cloud config依赖

```xml
<!--config-->
<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-config -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>2.1.1.RELEASE</version>
</dependency>
```

3.主启动类

```java
@SpringBootApplication
@EnableEurekaServer //EnableEurekaServer 服务端的启动类，可以接受别人注册进来~
public class ConfigEurekaServer_7001 {

    public static void main(String[] args) {

        SpringApplication.run(ConfigEurekaServer_7001.class,args);
    }
}
```

4.测试

第一步：启动 Config_Server_3344，并访问 http://localhost:3344/master/config-eureka-dev.yml 测试


```yaml
eureka:
  client:
    fetch-registry: false
    register-with-eureka: false
    service-url:
      defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
  instance:
    hostname: eureka7001.com
server:
  port: 7001
spring:
  application:
    name: springcloud-config-eureka
  profiles:
    active: ''
```


第二部：启动ConfigEurekaServer_7001，访问 http://localhost:7001/ 测试

显示上图则成功

新建springcloud-config-dept-8001模块并拷贝springcloud-provider-dept-8001的内容

同理导入spring cloud config依赖、清空application.yml 、新建bootstrap.yml配置文件并配置

```yaml
spring:
  cloud:
    config:
      name: config-dept
      label: master
      profile: dev
      uri: http://localhost:3344
```

主启动类

```java
@SpringBootApplication
@EnableEurekaClient //在服务启动后自动注册到Eureka中！
@EnableDiscoveryClient //服务发现~
@EnableCircuitBreaker //
public class ConfigDeptProvider_8001 {

    public static void main(String[] args) {

        SpringApplication.run(ConfigDeptProvider_8001.class,args);
    }

    //增加一个 Servlet
    @Bean
    public ServletRegistrationBean hystrixMetricsStreamServlet(){

        ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
HystrixMetricsStreamServlet());
        registrationBean.addUrlMappings("/actuator/hystrix.stream");
        return registrationBean;
    }
}
```

测试 (略)

感谢❤️

愿你

回复 2021-08-14 15:56:41

大佬，跟着p4教程写报错
Invalid bound statement (not found): com.you.springcloud.servcie.DeptService.queryAll
该怎么解决?

Chāo

回复 2021-07-22 17:37:55

提供一下自己的完整代码：https://gitee.com/chao-work/springcloud

芝麻糕 @Chāo

回复 2021-09-28 10:37:15

感谢大佬，懒得敲了😁

深拥为伴 @Chāo

回复 2021-07-23 23:47:24

感谢

范子豪

回复 2021-07-06 14:49:15

老哥，完整代码在哪里可以下载啊?

君兮

回复 2021-06-08 23:14:55

大佬你好强👍

Qīngchūn¹⁸😜ɔ🙏ʔc

回复 2021-06-03 11:53:03

👍牛气

兴趣使然的草帽路飞 @Qīngchūn¹⁸😜ɔ🙏ʔc

回复 2021-06-04 14:09:13

欢迎关注我的CSDN博客，兴趣使然的草帽路飞，互相学习哈!

没有更多了 (共7条 / 当前1页)

TOP