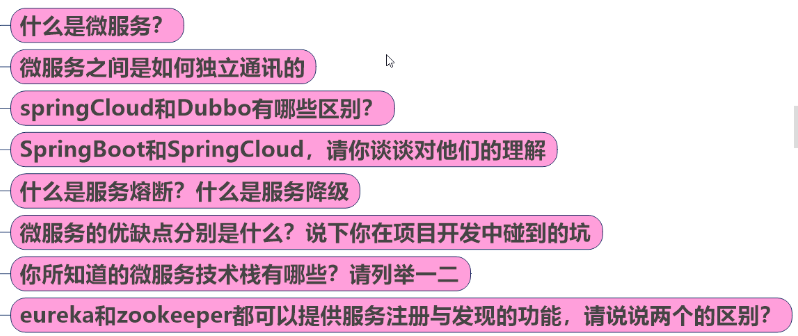
课程前提，会使用：springMVC+Spring+SpringBoot+MyBatis+Maven+git…

SpringCloud五大神兽 尚硅谷周阳老师（金句频出）

新技术都是站在老技术的基础概念、功能、思想上进行迭代升级和演化。

天上飞的理念必然有落地的实现。

企业里最火爆的Zookeeper+Dubbo模式的服务化应用拆分和分布式部署。

Spring和Dubbo的区别：

1：通讯机制不一样，Dubbo是基于RPC远程过程调用，而微服务Cloud是基于Rest调用。

# 微服务

## 1.1、基本概念

2014年左右马丁福勒提出。微服务是一种架构模式或者说是一种架构风格。它提倡将单一应用程序划分成一组小的服务，每个服务运行在其自己独立的进程中，服务之间相互协调配合，为用户提供最终价值。服务之间采用轻量级的通讯机制相互沟通，通常是基于Http的Restful API。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外。应该尽量避免同一的，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，可以是不同的语言来编写服务，也可以使用不同的数据库存储。

技术维度：微服务化的核心就是将传统的一站式应用，根据业务拆成一个一个的服务，彻底的去耦合，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者销毁，拥有自己独立的数据库。分模块，独立部署。

微服务强调的是一个个的个体，每个个体完成一个具体的任务或务功能的大小。

专业的事情交个专业的人去做，尽量降低耦合度

## 1.2、微服务优缺点

优：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者业务需求；2、开发简单，提高效率，小团队开发，容易被一个开发人员理解；3、微服务是松耦合的，是有功能意义的服务，无论是在开发阶段还是在部署阶段都是独立的；4、可以使用不同的语言，容易集成第三方；5、微服务只是业务逻辑代码，不会和HTML,CSS或其他界面组件混合；6、每个微服务都有自己的存储能力，可以有自己的数据库，也可以统一数据库。

缺：开发人员要处理分布式系统的复杂性，多服务运维难度会随着服务的增加，运维压力也在增加，系统部署依赖，服务间通讯成本，数据一致性，系统集成测试，性能监控等等。

## 1.3、微服务技术栈

多种技术的集合体，讨论一个分布式微服务架构的话，它需要有哪些维度？

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

## 1.4、为什么选择SpringCloud作为微服务架构？

微服务不是仅仅只有SpringCloud一种落地技术来实现。

阿里：2012年Dubbo梁飞团队被打散 HSF

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 功能点  服务框架 | 主流分布式架构 | | | | |
| Netfix/SpringCloud | Motan | gRPC | Thrift | Dubbo |
| 定位 | 完整的微服务框架 | RPC框架，但整合了ZK或者Consul,实现集群环境的基本注册发现 | RPC框架 | RPC框架 | 微服务框架 |
| 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 | 是 | 是 | 是 |
| 支持多语言 | 是 | 否 | 是 | 是 | 是 |
| 服务注册/发现 | 是Eureka服务注册表，Karyon服务端框架支持服务自注册和健康检查 | 是Zookeeper/Consul | 否 | 否 | 是 |
| 负载均衡 | 是 服务端Zuul+客户端Ribbon | 是 客户端 | 否 | 否 | 是 |
| 配置服务 | NetFlix Archaius SpringCloudConfig Server集中配置 | 是 Zookeeper提供 | 否 | 否 | 否 |

# 二、SpringCloud入门概述

## 2.1、SpringCloud是什么，能干什么，去哪下，怎么玩，使用情况？

1、Spring的官网越来越简洁，越来越少的技术，简洁而不简单，JavaEE占80%的开发都是Spring社区技术的使用。

2、分布式系统的演化：构建一个分布式系统并不需要特别的复杂，Spring能够提供一种简单的程序模型，包含了多种分布式系统中的模式和设计理念。SpringCloud是一堆技术的集合体。

3、SpringCloud，基于SpringBoot提供了一整套的微服务解决方案，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。

4、SpringCloud利用SpringBooot的开发便利性巧妙的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捷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、分布式回话等等，它们都可以用SpringBoot的开发风格做到一键启动和部署。

5、SpringCloud并没有重复制造轮子，它只是将目前各家公司开发比较成熟、经得起考验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，最终给开发者流出了一套简单易懂、易部署和维护的分布式系统开发工具包。

总结：SpringCloud=分布式微服务架构下的一站式解决方案，是各种微服务框架落地技术的集合体，俗称微服务全家桶

## 2.2、SpringCloud和SpringBoot是什么关系？

SpringCloud更加偏向宏观，SpringBoot偏向于微观。SpringCloud依赖于SpringBoot，SpringBoot可以作为独立开发项目使用，但是SpringCloud离不开SpringBoot。SpringBoot专注于快速开发单个个体微服务，SpringCloud是专注于全局的微服务协调治理框架，它将SpringBoot开发的一个个单体微服务整个并且管理起来。

## 2.3、Dubbo是怎么演化到SpringCloud的？那些优缺点让你去技术选型？

应用服务化拆分+消息中间件

对比社区活跃度，SpringCloud的更高一些

对比结果：

|  |  |  |
| --- | --- | --- |
|  | **Dubbo** | **SpringCloud** |
| 服务注册中心 | Zookeeper | Spring Cloud Netflitx 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 |

最大的区别：SpringCloud抛弃了Dubbo的RPC通讯，采用的是基于HTTP的REST方式，严格的来说，这两种方式各有优势，REST不具备服务调用的性能，但却比RPC更加灵活，服务提供方和调用方的依赖只靠一纸契约，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。很明显的是，SpringCloud的功能相比Dubbo更加强大，并且兼容Spring生态圈。而Dubbo构建微服务架构时候提供了更高的选择自由度，但随之而来的是出问题的可能性也很高，选择起来就要求对各个“部件”有足够的了解。遇到问题时候中小型公司没有技术高手去修改Dubbo的源码+周边的一整套解决方案，并不是每一个公司都有阿里大牛+真实的线上生产环境测试过。

## 2.4、总结Dubbo与Spring Cloud

Dubbo重启维护之后的负责人之一是 刘军，他的说法是：

Dubbo的定位始终是一款RPC框架，而Spring Cloud的目标是微服务架构下的一站式解决方案，Dubbo可以类比到Netflix OSS技术栈，而Spring Cloud集成了Netflix OSS 作为分布式服务治理解决方案。当前由于RPC协议，注册中心元数据不能匹配等问题，在面临微服务基础框架选型时Dubbo与Spring Cloud只能二选一。Dubbo之后会积极寻求适配到Spring Cloud生态，比如作为Spring Cloud的二进制通信方案来发挥Dubbo的性能优势，或者Dubbo通过模块化以及对HTTP的支持适配到SpringCloud。

## 2.5、能干嘛

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

Service registry 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)

## 2.6、网址

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

官方中文文档： <https://springcloud.cc/spring-cloud-netflix.html>

官方配置文档： <https://springcloud.cc/spring-cloud-dalston.html>

SpringCloud中国社区： <http://springcloud.cn/>

SpringCloud中文网： <http://springcloud.cc/>

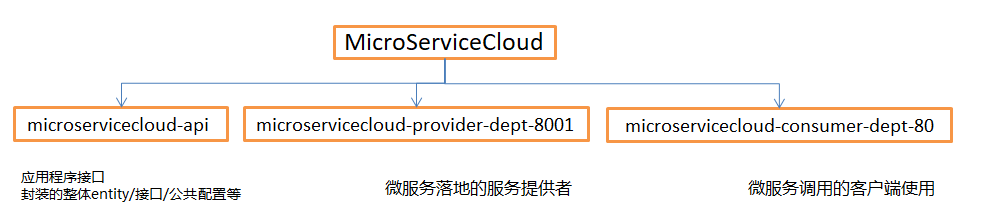
## 2.7、国内谁在用

中国联通子公司，上海米么金服，指点无限，易保软件，广州简法网络，猪八戒，东软，深圳魔令，广州万表网，视觉中国，上海秦苍买单侠，大连爱油科技，卖货郎，广发银行，拍拍货。

# 三、SpringCloud微服务搭建环境

## 3.1、项目结构

一个MicroServiceProject带着多个Module子模块，一个ServiceCloud父工程下，初次带着3个子模块（Module）



一处定义，多处使用，利用Maven的GAV坐标。每一个服务的后面跟上服务名和端口号。

整体父工程Project、公共子模块Module、部门微服务提供者Module、部门微服务消费者Module

## 3.2、SpringCloud和SpringBoot版本选用

SpringCloud的版本是伦敦地铁站的名字，可能优秀的程序员都会去挤地铁吧！

SpringCloud：Dalston.SR1

SpringBoot：1.5.9.RELEASE

## **3.3、父工程**

microservicecloud

packaging选择pom，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，类似于一个抽象父类。

## 3.4、API工程Module

microservicecloud-api

### 3.4.1、创建和定义

新建microservicecloud-api，修改pom文件，新建部门Entity且配合lombok使用，mvn clear install后给其他模块引用，达到通用的目的。也即需要用到部门实体的话，不用每一个工程都定义一份，直接引用米板即可。

新建的是Maven的Module

Lombok解决的问题是：在构造一个entity的时候，可能会加字段，减字段，构造方法和set()和get()方法随时会改变，使用lombok可以更好的解决。

工程创建好之后到pom文件上，先运行 maven –clean再运行maven –install，Install的时候要选择jdk不要选择jre不然install编译会报错。

### 3.4.2、Lombok插件的安装

在IDEA里面：

Setting 🡪 Plugins 搜索lombok，之后install 重启即可。

在Eclipse里面：

下载lonbok.jar文件，放到跟eclipse.ini同级目录，在命令行输 java -jar lombok.jar

再选择eclipse.exe 安装之后重启即可。

## 3.5、服务提供者Module

microservicecloud-provider-dept-8001

### 3.5.1、步骤

①创建module；②解决pom文件；③yml配置文件的安装和编写；④在工程下面根目录创建mybatis文件夹和mybatis.cfg.xml文件；⑤新建数据表，运行建表脚本什么的；⑥DAO层的接口；⑦dao层接口实现，Mabatis使用的是Mapper，去写xml配置文件。

⑧Dao层，Service层，Controller层；⑨主启动类，会手写吗？；

### 3.5.2、开发思维

1、相同的代码出现了第三次就应该将它提出来！

2、既然要前后端分离，那么Controller里面大部分的情况应该是只返回数据的，也就是Json字符串，所以在Controller层使用@RestController注解。

3、restful风格的访问在前面一定要有一个名词，表示要对哪一种rest表达性资源进行状态转移。 哪一个资源得说明白，对这个资源进行什么样的操作也要说清楚。举例：/dept/add，/dept/get/{id}，/dept/list

4、SpringBoot主启动类，@SpringBootApplication注解标注在类上，main方法里面 SpringApplication.run(“类名.clsaa”,args);

5、访问地址：

<http://localhost:8001/dept/get/1>

<http://localhost:8001/dept/list>

6、过程：maven先解决pom.xml 🡪 application.yml 🡪 数据库脚本 🡪 Dao层接口 Mapper 🡪 Dao层实现 Mapper.xml 🡪 Service层接口 🡪 Service层实现 🡪 Controller层 🡪 主启动类 🡪浏览器端看效果

## 3.6、服务消费者Module

### 3.6.1、新建module

跟之前新建的一样，打包方式是jar。

SpringBoot逐渐的优化了Spring框架，之前在Spring框架使用的是applicationContext.xml配置文件，在SpringBoot里面使用的是@Configuration注解，在基于注解的Spring开发中有讲解。

### 3.6.2、RestTemplate

RestTemplate提供了多种便捷访问远程Http服务的方法，是一种Spring高度抽象以后的针对于restful服务调用的模板类，更加的简洁高效，主要是Spring提供的用于访问Rest服务的客户端模板工具集。把Rest调用又封装了一层。

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

### 3.6.3、输入访问地址测试

当服务消费者microservicecloud-consumer-dept-80的主启动类以及coltroller层代码写好之后，在浏览器测试的连接是：<http://localhost/consumer/dept/get/4>，<http://localhost/consumer/dept/list>

页面上返回Json数据即为基本环境搭建成功。

开发到此，之前的所有内容都是Maven，Spring，SpringMVC，SpringBoot，MyBatis知识的复习，因为SpringCloud是基于微服务集群的配置，接下来才是真正的核心内容。

在Eclipse里面跑微服务程序很吃内存，随着微服务的增加，没有8G很难跑起来。

# 四、Eureka服务注册与发现

## 4.1、Eureka简介

NetFlix在设计Eureka的时候遵守的就是AP原则。

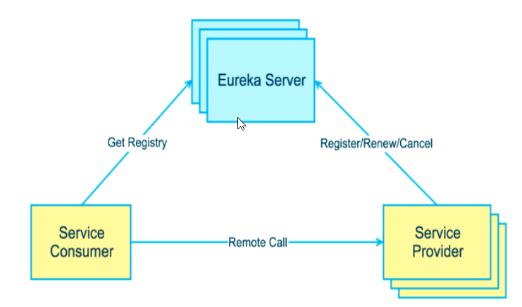
Eureka是Netflix的一个子模块，也是核心模块之一。Eureka是一个基于REST的服务，用于定位服务，以实现云端中的数据层服务发现和故障转移。服务注册与发现对于微服务架构来说是非常中药包的，有了服务注册与发现，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件。功能类似于dubbo的注册中心，如Zookeeper。

## 4.2、Eureka原理解析

### 4.2.1、概述

Eureka采用的是C-S 架构。Eureka Server作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用Eureka的客户端连接到Eureka Server并维持心跳连接。这样系统维护人员就可以通过Eureka Server来监控系统中各个微服务是否运行正常。Spring Cloud的一些其他模块，例如Zuul就可以通过Eureka Server来发现系统中的其他微服务，并执行相关逻辑。

### 4.2.2、Eureka Server和Eureka Client



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Eureka Server的交互，客户端同时也具备一个内置的、使用轮询（round-robin）负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳（默认周期是30秒）。如果Eureka Server在多个心跳周期内没有接收到某个接的心跳，Eureka Server会从服务注册表中把这个服务节点删除（默认90秒）。

### 4.2.3、三大角色

1、Eureka Server提供服务注册和发现

2、Server Provider服务提供方将自身服务注册到Eureka，从而使服务消费方能够找到

3、Service Consumer服务消费方从Eureka获取注册服务列表，从而能够消费服务

## 4.3、Eureka工程落地实现

### 4.3.2、使用新组建三步走

1、新增一个相关的maven坐标

2、在主启动类上面标注启动该新组建技术想关的注解

3、编写业务代码

例：Eureka

1、<!--eureka-server服务端 -->

<dependency>

<groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

2、@EnableEurekaServer // EurekaServer服务器端启动类,接受其它微服务注册进来

### 4.3.1、新建Eureka服务注册中心Module

新建microservicecloud-eureka-7001Module也就是eureka模块，步骤同样是 pom🡪

Yml 🡪 EurekaServer7001\_App主启动类 🡪 测试

### 4.3.3、测试

<http://localhost:7001/> 访问能看到东西就算Eureka服务注册中心配置成功。主要的两个文件就是pom，yml,然后在SpringBoot的启动类上标注@EnableEurekaServer即可。

## 4.4、将一个服务提供者注册进入Eureka

也就是在microservicecloud-provider-dept-8001服务提供者的项目下面依次修改： pom.xml 🡪 application.yml 🡪 主启动类上加注解 @EnableEurekaClient

1、在pom文件中增加maven依赖

<!-- 将微服务provider侧注册进eureka -->

<dependency>

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

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

2、在yml文件中加入配置

eureka:

client: #客户端注册进eureka服务列表内

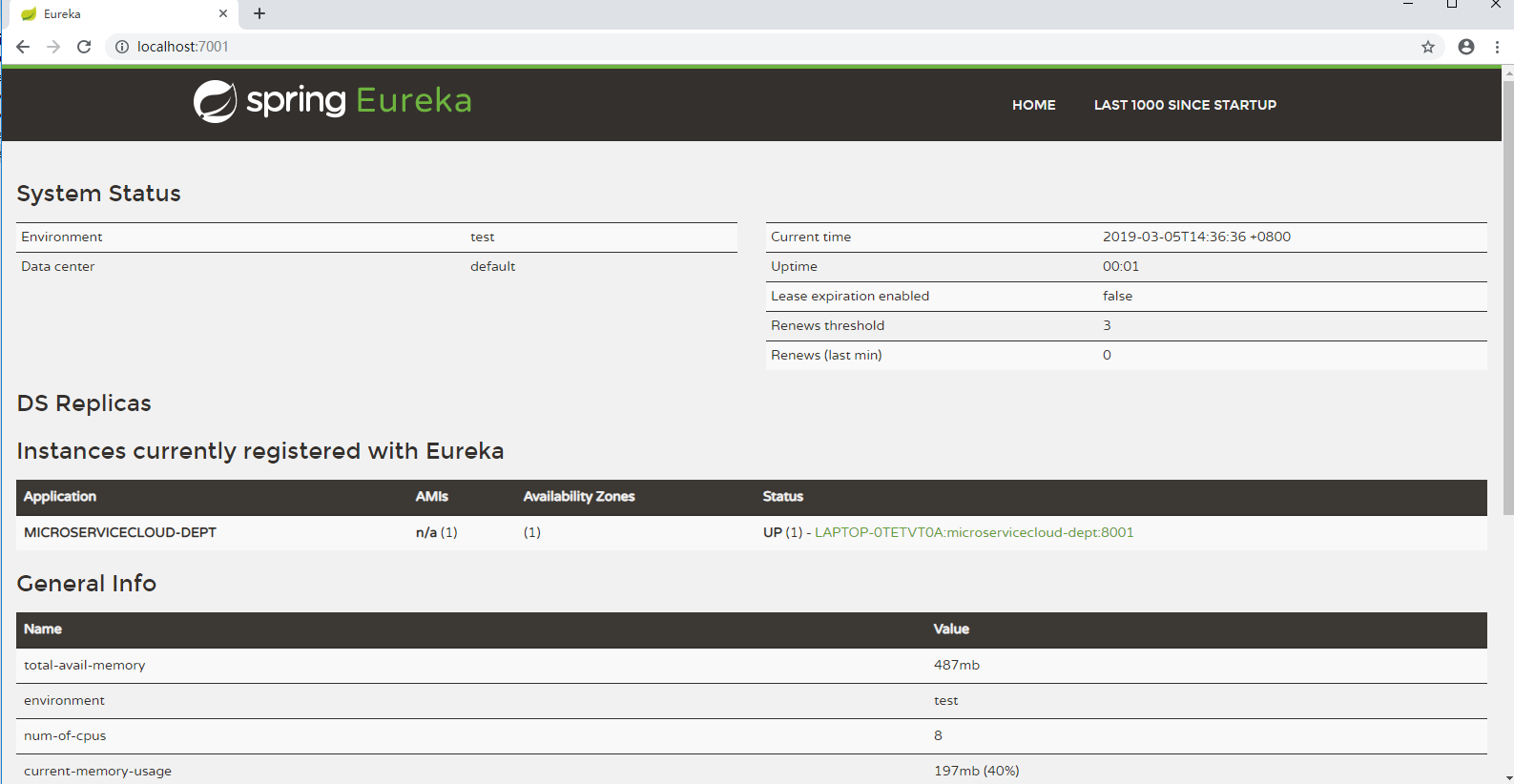
service-url:

defaultZone: http://localhost:7001/eureka #单机版

3、在服务提供者的主启动类上标注 @EnableEurekaClient 注解

启动测试： <http://localhost:7001/> 服务列表里面有东西了就表示成功，启动的时候Eureka先启动，连接它的客户端后启动。

Application的名称name是在yml文件中定义的。



## 4.5、Actuator与注册微服务信息完善

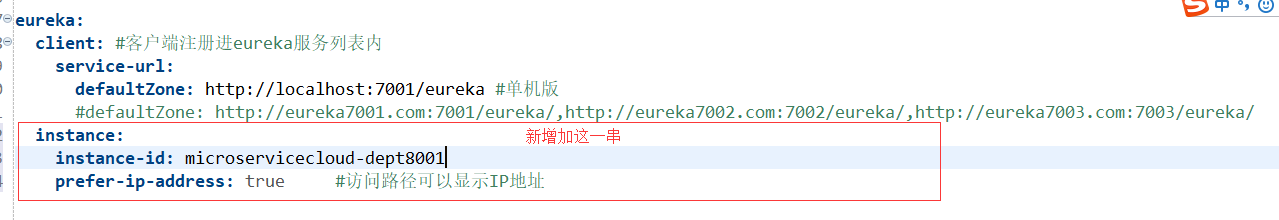
### 4.5.1主机名称和超连接IP地址修改

新增以下信息

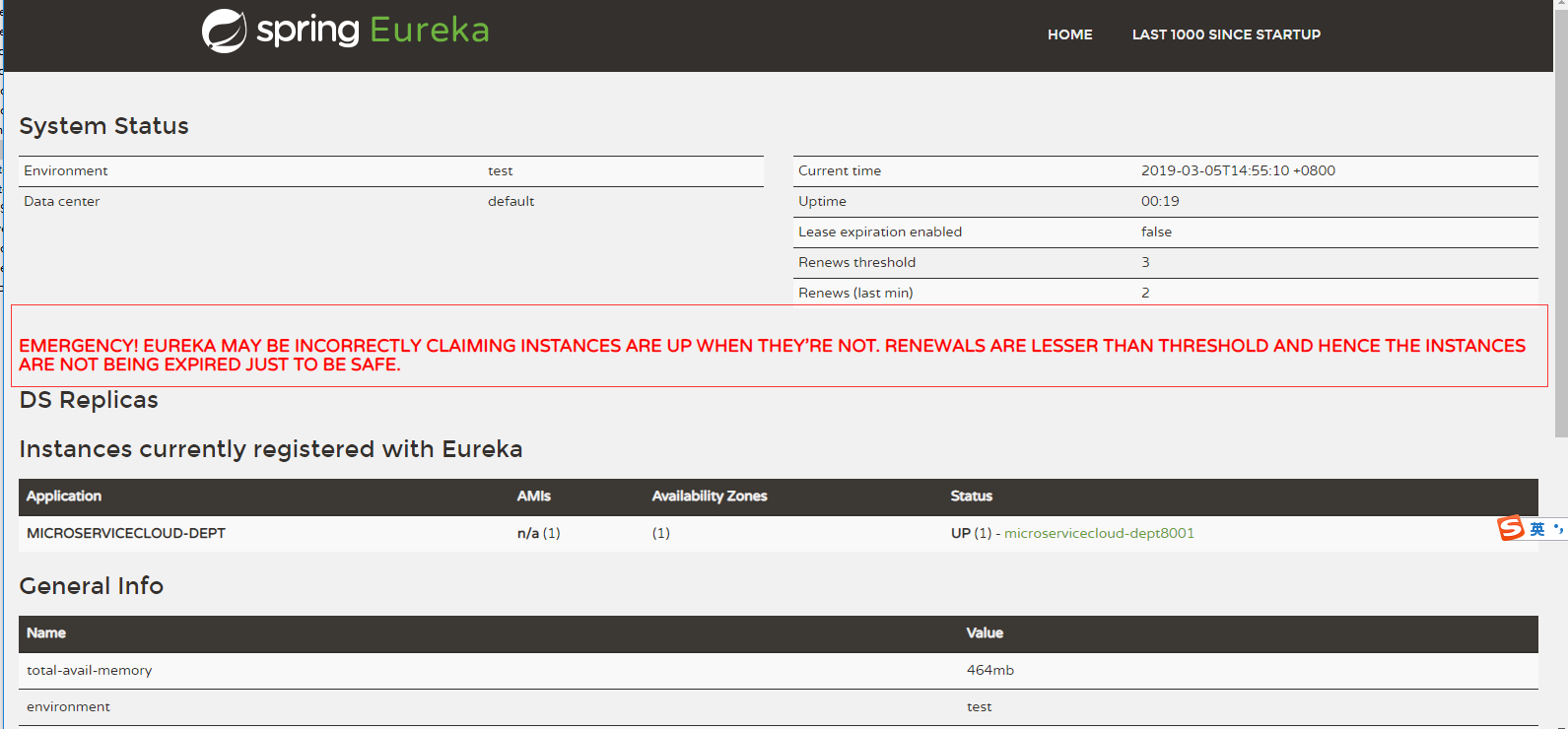
instance:

instance-id: microservicecloud-dept8001 #自定义服务名称信息

prefer-ip-address: true #访问路径可以显示IP地址



修改之后，中间的红色字体是Eureka的自我保护



### 4.5.2、微服务Info内容详细信息

点击超链接，希望看到关于这个微服务的一些信息。

1. 修改服务提供者microservicecloud-provider-dept-8001 的pom文件，新增依赖。

<!-- actuator监控信息完善 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

1. 在总体父工程修改pom.xml文件，添加构建build 信息

<build>

<finalName>microservicecloud</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>

<delimit>$</delimit>

</delimiters>

</configuration>

</plugin>

</plugins>

</build>

添加完成之后就是这样子的：



1. 修改服务提供者的yml配置文件

新增如下信息：

info:

app.name: learn-microservicecloud

app.author: xiaosha

app.info: Dept-Servicr-Provider 提供部门服务

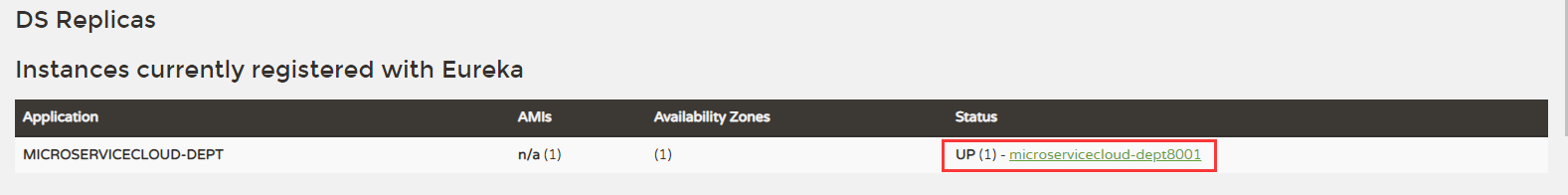
build.artifactId: $project.artifactId$

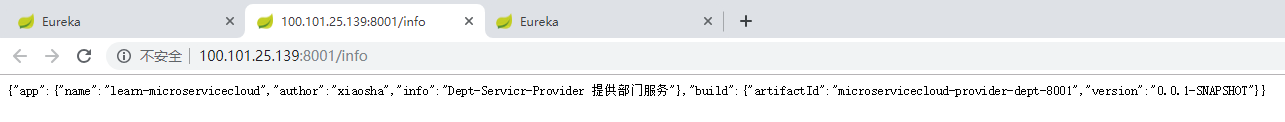
build.version: $project.version$

改成这种样子：



点击超链接会显示一些关于这个微服务的信息。





## 4.6、Eureka的自我保护机制

某个时刻某一个微服务不可用了，Eureka不会立即清理，依旧会对改微服务的信息进行保存。

默认情况下，如果Eureka Server在一定时间内没有接受到某个微服务实例的心跳，Eureka Server 将会注销该实例（默认90秒）。但是当网络分区故障时，微服务与Eureka之间无法正常通讯，以上行为可能变得非常危险了，因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护”来解决这个问题，当Eureka Server 节点在短时间内丢失过多的客户端时（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。一旦进入该模式，Eureka Server就会保护服务注册表中的信息，不再删除注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该Eureka Server会自动推出保护模式。

在自我保护模式中。

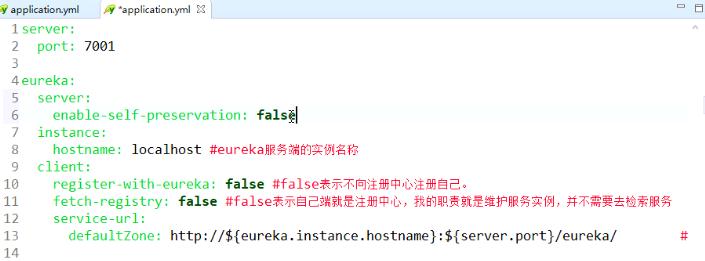
在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例，当它收到的心跳数重新恢复到阀值以上时，该Eureka Server节点就会自动推出自我保护模式。它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话总结就是：好死不如赖活着。

综上，自我保护模式就是一种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，不健康的都会保留），也不盲目注销仍盒健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

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.

翻译：紧急情况！Eureka可能错误地声称实例已经启动，而事实并非如此。续约低于阈值，因此实例不会为了安全而过期。

在SpringCloud中，可以使用 eureka.server.enable-self-preservation = false 禁止自我保护模式。



是去7001，也就是Eureka Server端配的。

## 4.7、Eureka服务发现Discovery

操作的还是microservicecloud-provider-dept-8001 这个Module，也就是服务提供者。

对于注册进入Eurake里面的微服务，可以通过服务发现来获得该服务的信息。

1. 在Controller层加入代码，遍历所以注册进入Eureka的微服务信息，将一些信息输出。

/\*\* @Autowired

private DiscoveryClient client;

\* 这个接口的目的是：遍历Eureka中的所有微服务信息

\*/

@RequestMapping(value = "/dept/discovery", method = RequestMethod.GET)

public Object discovery()

{

List<String> list = client.getServices();

System.out.println("\*\*\*\*\*\*\*\*\*\*" + list);

List<ServiceInstance> srvList = client.getInstances("MICROSERVICECLOUD-DEPT");

for (ServiceInstance element : srvList) {

System.out.println(element.getServiceId() + "\t" + element.getHost() + "\t" + element.getPort() + "\t"

+ element.getUri());

}

return this.client;

}

DiscoveryClient导入的包是这个：org.springframework.cloud.client.discovery.DiscoveryClient;

1. 在主启动类上添加注解 @EnableDiscoveryClient //服务发现
2. 这时候可以自测了，启动Eureka Server，再启动服务提供者，访问在Controller里面写好的对外暴露的接口。在自己项目下测试的路径是： <http://localhost:8001/dept/discovery>
3. 自测通过之后就可以在服务消费端使用了，也就是直接在Controller层添加如下代码：

其他的任何地方都不用改。

// 测试@EnableDiscoveryClient,消费端可以调用服务发现

@RequestMapping(value = "/consumer/dept/discovery")

public Object discovery()

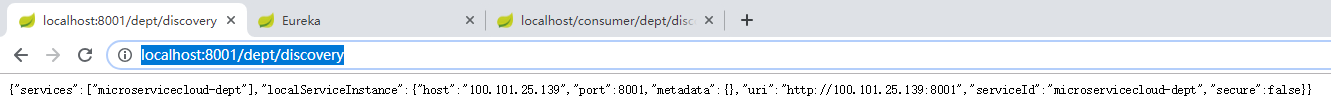
{

return restTemplate.getForObject(REST\_URL\_PREFIX + "/dept/discovery", Object.class);

}

1. 启动测试，需要把Eureka服务注册中心启动，服务提供者启动，服务消费者启动，在消费端的端口下去访问。 <http://localhost/consumer/dept/discovery>





## 4.8、配置Eureka Server集群

通常不是配置一个Eureka就可以的，需要多配置几个，才满足高可用，也避免单Eureka宕机之后的问题。

1、新建microservicecloud-eureka-7002、microservicecloud-eureka-7003两个工程，也就是7001、7002、7003共同组成Eureka集群；

2、参照7001搞定pom文件、主启动类文件；

3、去hosts文件添加如下信息；

127.0.0.1 eureka7001.com

127.0.0.1 eureka7002.com

127.0.0.1 eureka7003.com

4、yml文件的写法

eureka:

instance:

hostname: eureka7001.com

client:

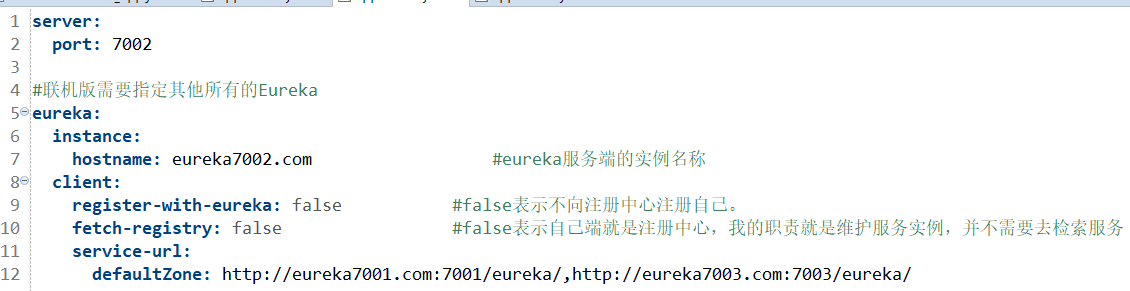
register-with-eureka: false

fetch-registry: false

service-url:

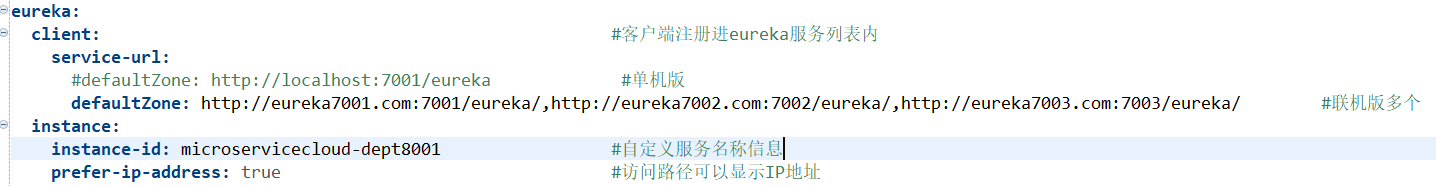
defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/





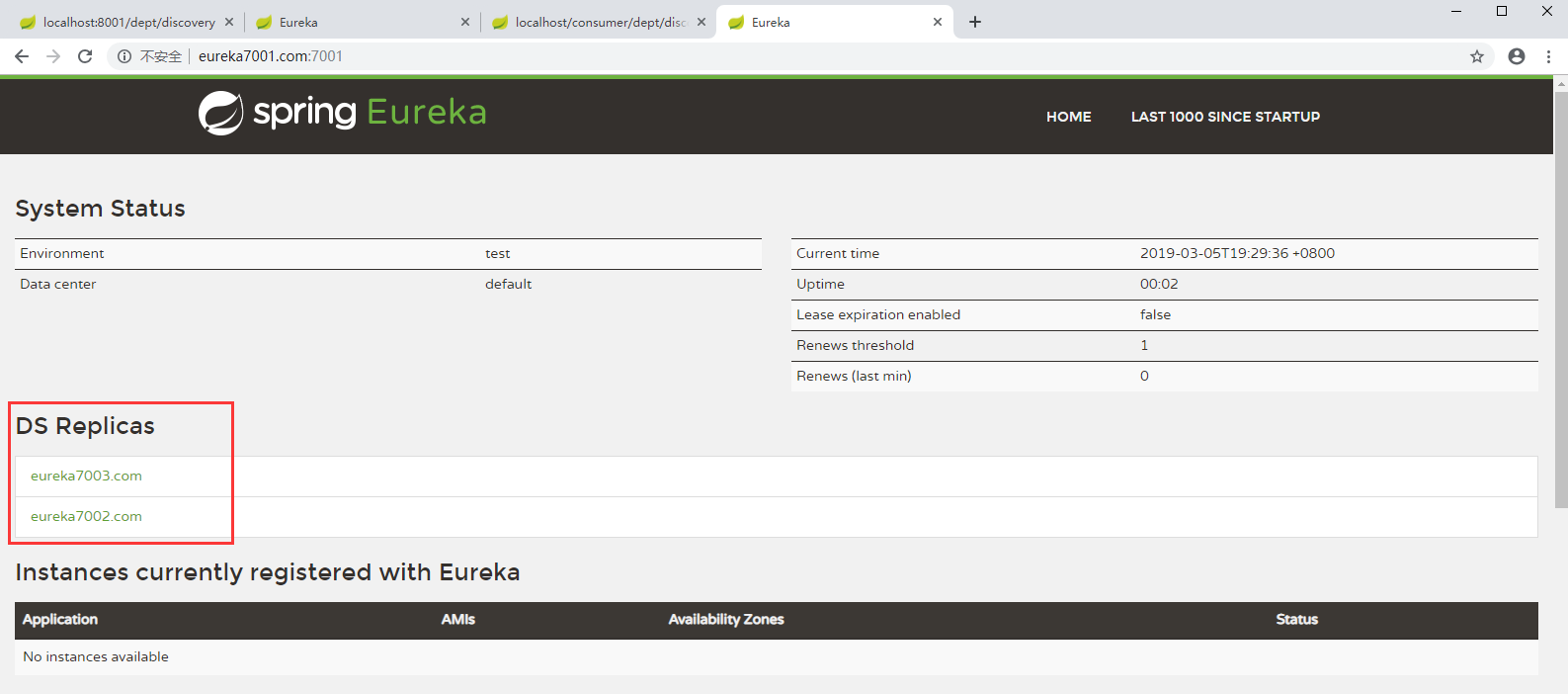


1. 修改服务提供者8001的yml配置文件



1. 启动验证，先启动7001，7002，7003，再启动8001

访问地址是：<http://eureka7001.com:7001/> 看到这个东西就说明没问题了。



## 4.9、对比Zookeeper和Eureka

或者说，作为服务注册中心，Eureka比Zookeeper好在哪里？

在传统的关系数据库里面（MysSQL、Oracle、SQLServer），有一个很重要的概念，就是ACID

A:Atomicity 原子性

C:Consistency 一致性

I:Isolation独立性

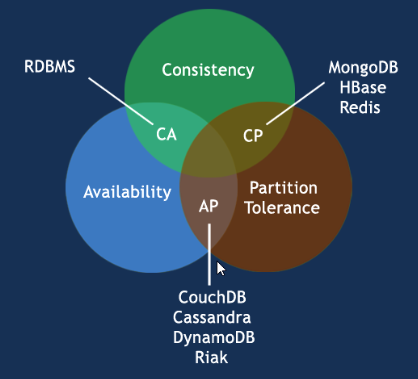
D:Durability 持久性

而作为NoSQL数据库（Redis,MongDB），与之对应的理念就是CAP

C:Consistency强一致性

A:Availability 可用性

P:Partition tolerance 分区容错性



答题关键点：Eureka遵守AP，Zookeeper遵守CP。

在一个分布式系统里面，没办法同时满足强一致、高可用和分区容错性，即CAP,也就是3进2，3个里面选择两个，也就是一个分布式系统只能同时较好的满足两个。由于现在分布式系统在部署的时候绝对不可能只有一个一台机器，也就意味着，分区容错性必须满足，也就是P绝对要被占有。初次之外还有一个原因是，由于部署在不同的机器上，由于网络传输必然会出现延迟丢包等问题，所以，分区容错性是一个分布式系统必须要实现的。

所以只能在一致性和可用性之间进行权衡，没有NoSQL系统能够同时保证这三点。

考题：如果是京东、淘宝这样的网站，只能保证AP，不能选择CP。理由如下：

要以保证网站正常运作而选择A,P用来精确统计数据。

### Zookeeper保证CP

当向注册中心查询服务列表的时候，我们可以容忍注册中心返回的是几分钟前的注册信息，但是不能接受服务直接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，剩余的节点会重新选举leader。问题在于，选举leader的时间太长，30~120s，并且选举期间整个zookeeper集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署情况下，因为网络问题而使得zk集群失去master节点是较大概率会发生的事，虽然服务能够很快恢复，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AP

Eureka在设计的时候就是优先保证可用性的。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的节点，只要有一台Eureka还在，就能保证注册服务可用（保证了可用性），只不过查到的信息可能不是最新的（不能保证强一致性）。除此之外，Eureka还有一种自我保护机制，如果再15分钟内超过85%d的节点都没有正常的心跳，那么Eureak就会认为客户端与注册中心出现； 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有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，但是暂时不会被同步到其他的节点上（即优先保证当前节点依然可用）
3. 当网络稳定的时候，当前实例的新的注册信息才会被同步到其他节点上

对比结论：Eureka可用很好的应对网络故障导致的部分节点失去联络，而不会像Zookeeper那样使得整个注册服务瘫痪。

# 五、Ribbon负载均衡

## 5.1、LB简介

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客户端组件提供一系列完善的配置项如连接超时、重试等。简单的说，就是在配置文件中列出Load Balance(简称LB)后面所有的机器，Ribbon会自动的帮助你基于某种规则（比如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自己定义你的负载均衡算法。

LB，即负载均衡（Load Balance）,在微服务或分布式集群中经常使用到的一种应用。负载均衡简单的说就是将用户的请求平均的分摊到多个服务商，从而达到系统HA(搞可用)，

常见的负载均衡软件有Nginx，LVS，硬件F5等。相应的在中间件，例如：Dubbo和SpringCloud中均给我们提供了负载均衡，SpringCloud的负载均衡算法可用自定义。

集中式的负载均衡偏硬件，进程式的负载均衡偏软件。

所谓集中式就是在服务的消费方和服务的提供方之间使用独立的LB设施（可用是硬件，如F5,也可以是软件，如Nginx），由该设施负责把访问请求通过某种策略转发至服务提供方。

F5贵啊。

所谓进程式，就是将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择一个合适的服务器。Ribbon就术语进程内LB，它只是一个类库，集成与消费方，消费方通过它来获取到服务提供方的地址。

## 5.2、Ribbon落地实现

实现步骤：主要是对microservicecloud-consumer-dept-80工程的修改。

1、在客户端，也就是服务消费端的pom文件中添加依赖，其中会有一个Eureka的坐标，其实是因为Ribbon需要跟Eureka整合。在pom文件中添加以下GAV坐标：

<dependency>

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

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

2、在microservicecloud-consumer-dept-80服务消费者工程的yml文件中添加以下信息：

eureka:

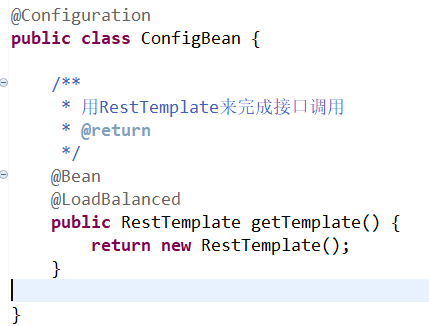
client:

register-with-eureka: false

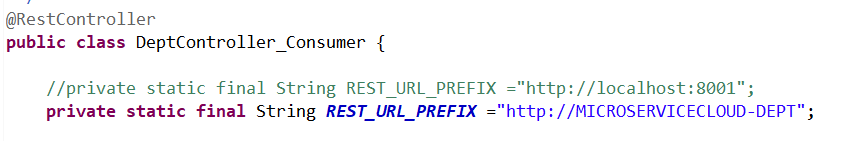
service-url:

defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

1. 因为Ribbon是一套客户端负载均衡工具，而在服务消费端，并没有写任何的业务逻辑代码，而是通过RestTemplate去根据URL获取在服务提供方提供的数据。既然是客户端负载均衡工具，是不是也就意味着要对这个真正调用的组件加一点什么东西呢？在想一步，我们是通过写一个配置类，使用@Bean来把这个RestTemplate加载到容器中的，所以，只需要在@Configuration类里面，对这个RestTemplate加一个注解@LoadBalanced即可。如下：



1. 在主启动类上标注@EnableEurekaClient注解
2. 修改客户端访问类，也就是之前Controller类的访问路径之前是被写死的。给他变成微服务名字。



1. 集群启动测试，顺序依次会是7001,7002,7003三个Eureka服务注册发现中心，8001服务提供者，80服务消费者。

测试路径：<http://localhost/consumer/dept/list> <http://localhost/consumer/dept/get/1>

有数据显示在页面上即为成功配置。

Ribbon和Eureka整合之后Consumer可以直接调用服务而不再需要关心地址和端口号。

有了这个基础之后才真正的开始微服务负载均衡编程。

## 5.3、Ribbon负载均衡

Ribbon钻工作的时候分成两步：

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

第二部再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。Ribbon没有指定的话，默认的负载均衡策略会是轮询。

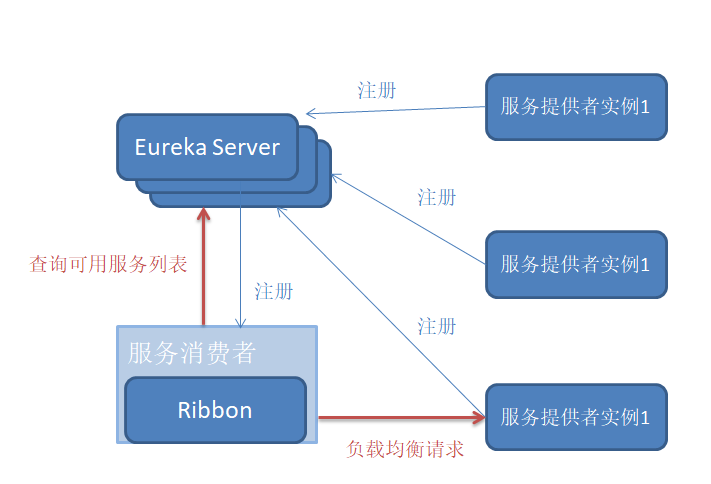
每个微服务可以连不同的数据库。

新建两个工程8002、8003作为新的微服务提供者，同样参照8001去修改pom文件、之后新建其他的两个数据库、然后修改yml文件，修改微服务端口号和所连接的数据库名称，但是尤其是对外暴露的统一的服务名称不能修改。



启动测试：先启动三个Eureka集群，载启动三个微服务提供者，再启动三个微服务，并且各自去测试看看是否通过。测试连接是： <http://localhost:8001/dept/list> <http://localhost:8002/dept/list> <http://localhost:8003/dept/list> 自测的结果是：直接访问对应的微服务，那么查到的数据就会是相应库里面的数据。

测试负载均衡的连接：<http://localhost/consumer/dept/list> 这样查询数据它每次连接数据库就会是遵循了负载均衡算法的。由于没有设置负载均衡算法，所以是以轮询的方式去调用对应的服务的。



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结合使用只是其中的一个案例。

## 5.4、Ribbon核心组件IRule

IRule:根据特定算法从服务列表中选取一个需要访问的服务。经常使用的有：

1、RoundRobinRule：轮询。

2、RandomRule：随机。

3、AvailabilityFilteringRule：先会过滤掉由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阀值得服务，然后对剩余的服务列表按照轮询策略进行访问。

4、WeightedResponseTimeRule：根据平均响应时间计算所有服务的权重，响应时间越快的服务权重越大，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，等统计信息足够会切换到WeightResponseTimeRule。

5、RetryRule：按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定的时间内会进行重试，获取可用的服务。

6、BestAvailableRule：会优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。

7、ZoneAvoidandeRule：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。

在80服务消费方定义负载均衡算法策略，也就是在配置类中注入一个负载均衡Bean,如下：

/\*\*用在配置文件中定义的随机算法，代替轮询算法，使得cunsumer调用provider的时候选择随机调用

\*/

@Bean

public IRule myRule() {

return new RandomRule();

}

## 5.5、自定义负载均衡算法

比较难。

1. 在80也就是服务消费者的主启动类上面标注@RibbonClient注解。

@RibbonClient(name="MICROSERVICECLOUD-DEPT",configuration=MySelfRule.class)

1. 踩坑

官方文档给出了明确的警告，这个自己定义的配置类不能放在@ComponentScan所扫描的当前包下以及子包下，否则我们定义的这个配置类就会被所有的Ribbon客户端所共享，也就是说我们达不到特殊化定制的目的了。

所以新建包：com.learn.myrule，将MySelfRule.class新建在这个包下边。

1. 新的需求，以前轮询的算法是每次轮询1，现在要求8001轮询5次之后再去8002轮询5次，再去8003轮询5次。解决办法是，直接去看官网的代码然后自己改造一遍。

GitHub源码地址：<https://github.com/Netflix/ribbon/blob/master/ribbon-loadbalancer/src/main/java/com/netflix/loadbalancer/RandomRule.java>

可以浏览器打开对比着看，改造部分关键代码如下：

// private int total = 0; // 总共被调用的次数，目前要求每台被调用5次

// private int currentIndex = 0; // 当前提供服务的机器号

if(total < 5){//调用次数从0到4，如果是第5次会进else这个分支

server = upList.get(currentIndex);

total++;

}else {

total = 0;

currentIndex++;

if(currentIndex >= upList.size()){

//只有三个机器 数组里面是0,1,2所以当执行到第3台机器，

//currentIndex加到3的时候,将currentindex置为0

currentIndex = 0;

}

}

4、把自己定义的负载均衡算法配置到Bean容器中。

@Configuration

public class MySelfRule {

@Bean

public IRule myRule() {

return new RandomRule\_ZY(); //自定义随机，5次一轮询

}

}

# 六、Feign负载均衡

## 6.1、基本概念

Feign是一个申明式WebService客户端。使用Feign能让编写Web Servicek客户端更加简单，它的使用方法是定义一个接口，然后在上面添加注解。同时也支持JAX-RS标准的注解。Feign也支持可拔插式的编码器和解码器。Spring Cloud 对Feign进行了封装，使其支持了Spring MVC标准和HttpMessageConverters。Feign可以与Eureka和Ribbin组合使用以支持负载均衡。

Feign是一个申明式的Web服务客户端，使得编写Web服务变得非常容易，只需要创建一个接口，然后在上面添加注解即可。

之前使用Ribbon进行负载均衡，功能很强大，甚至可以自己定义负载均衡算法。

Feign能干什么？

Feign的宗旨是使得在编写Java Http客户端更加容易。前面在使用Ribbon + RestTemplate的时候，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，由于服务以来的调用可能不止一处，往往一个接口会被多出调用，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以来服务的调用。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且使用注解的方式来配置它（以前是Dao接口上面标注Mapper注解，现在是一个微服务接口上面标注一个Feign注解即可），即可完成对服务提供方的接口绑定，简化了使用Spring Cloud Ribbon时，自动封装服务调用客户端的开发量。

面向服务接口调用微服务，接口上面导注解。

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

Feign通过接口的方法调用Rest服务（之前是使用Ribbon+RestTemplate）,该请求发送给Eureka服务器（http://MICROSERVICECLOUT-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，所以也支持负载均衡的作用。

## 6.2、Feign使用步骤

参考微服务microservicecloud-consumer-dept-80新建servicecloud-consumer-dept-feign工程。

1、新建Module、拷贝写好的yml\启动类\pom文件、修改主启动类名字。

修改pom文件，添加以下信息：

<dependency>

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

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

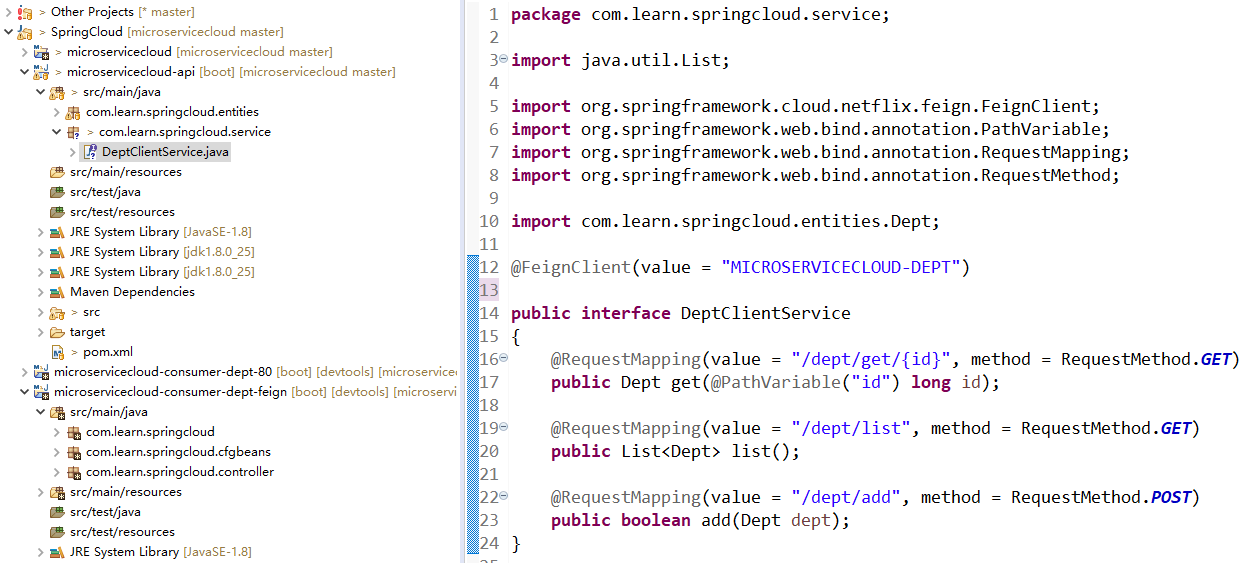
2、修改API工程

在pom的xml文件中添加feign相关依赖：



在API工程上新建service层的接口，新建接口DeptClientService

因为API工程是公共的，修改了之后需要maven –clean和maven –install



启动测试，不要把80服务消费者启动起来。只是启动Feign。

测试连接 <http://localhost/consumer/dept/list>

# 七、Hystrix断路器

Hystrix断路器，也叫熔断器，在SpringCloud里面主要负责做服务熔断，服务降级的处理，思想跟Spring里面环绕通知、前置通知等有相似之处。程序出现异常、长时间调用失败、调用不恰当、超时等。为了避免长时间没有响应而导致全局系统的瘫痪、挂起、死机。

## 7.1、在分布式系统中面临的问题

复杂分布式系统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会失败。系统不可避免的发生超时和占有紧张的情况。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”，依赖的层级越来越多，依赖程度加深加重加宽。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的调用就会占有越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更加糟糕的是，这些应用程序还可能导致服务器之间延迟增加，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者系统。

## 7.2、Hystrix是什么？

Hystrix是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里面，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现问题的情况下，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和高可用。

“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（类似于熔断保险丝），向调用方返回一个符合预期的、可处理多大的备选响应（FallBack）,而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要的占用，从而避免了故障在分布式字体中的蔓延，乃至雪崩。

## 7.3、Hystrix官网资料

<https://github.com/Netflix/Hystrix/wiki/How-To-Use>

## 7.4、服务熔断

### 7.4.1、服务熔断是什么？

熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务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框架里熔断机制提供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。

### 7.4.2、编码测试

1、参考8001新建microservicecloud-provider-dept-hystrix-8001这个Module

2、修改pom文件，在8001服务提供者的pom基础上添加以下依赖：

<!-- hystrix -->

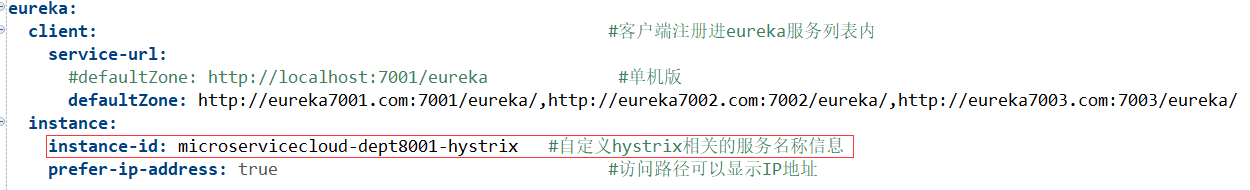
<dependency>

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

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

3、修yml文件



1. 拷贝其他文件，修改controller层代码如下：



1. 修改主启动类为DeptProvider8001\_Hystrix\_App添加@EnableCircuitBreaker注解。



1. 启动测试

先启动3个Eureka，然后是这个带服务熔断的服务提供者，然后是服务消费者。然后查询一条不再数据库里面的数据。 链接是： <http://localhost/consumer/dept/get/10>

不难发现，只要每有一个主业务处理逻辑，就需要一个FallBack方法，方法膨胀和高度耦合。

## 7.5、服务降级

### 7.5.1、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。所谓降级，一般是从整体复核考虑。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，返回一个缺省的值，这样做，虽然服务水平下降，但是好歹可用，总比直接宕机强。

服务降级的处理是在客户端实现的，与服务端没有关系。把主业务和降级之后的异常处理做成一个切面，通过支入的方式去执行这些程序。业务逻辑里面及不应该带有这么多横切性质的异常处理信息。业务和熔断处理的解耦合。

### 7.5.2、编码测试

1. 修改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。
2. 在这个新建的类上面添加@Component注解



1. 修改API工程的DeptClientService，意思就是，只要是这个接口里面的方法，只要服务出现了熔断、降级，去找实现了FallBackFactory的这个类去，也就是上面这个类。



1. 去到服务消费的Feign工程的yml文件，添加如下信息：

feign:

hystrix:

enabled: true



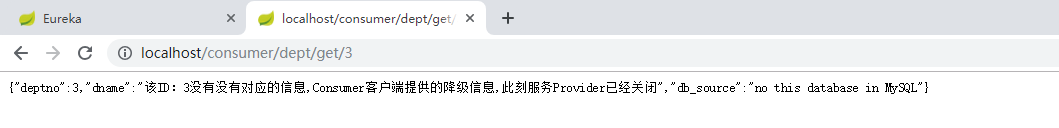
1. 启动测试

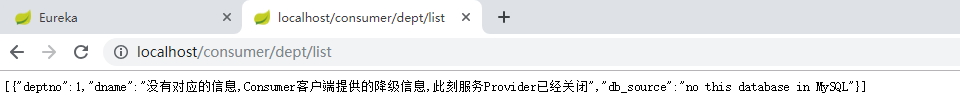
先启动3个Eureka7001,7002,7003、再启动8001微服务提供者，再启动服务消费者dept-feign

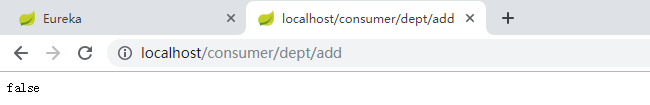
正常访问测试：

故意关闭微服务提供者8001

再去客户端调用：







此时，服务点provider已经down掉了，但是由于做了服务降级处理，让客户端在服务端不可用的时候也会获得提示信息而不会挂起耗死服务器。

## 7.6、服务监控HystrixDashboard

随着服务上线，微服务的工程越来越多，每一个微服务每个时间段访问的压力是多少，如果这个时候有些图形化的数据能够直观的看到，有助于帮助于我们构建高可用的微服务架构体系。

### 7.6.1、服务监控室什么？

除了隔离依赖服务的调用之外，Hystrix还提供了准时的调用监控（Hystrix Dashboard），Hystrix会持续的记录所有通过Hystrix发起的请求的执行信息，并且统计报表和图形的形式展现给用户，包括美妙执行多少请求多少成功，多少失败等。Netflix通过hystrix-metrics-stream项目实现了对以上指标的监控。SpringCloud也提供了Hystrix Dashboardd的整合，对监控内容转化成可视化界面。

### 7.6.2、使用Hystrix Dashboard

1、新建microservicecloud-consumer-hystrix-dashboard工程

2、修改pom文件、yml文件

对比其他的服务消费者，pom文件增加以下信息：

<!-- hystrix和 hystrix-dashboard相关 -->

<dependency>

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

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

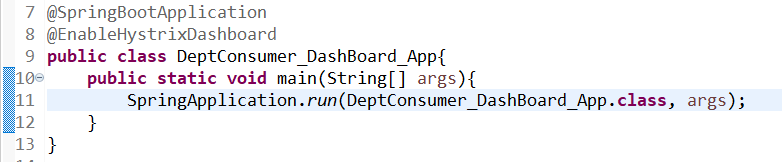
yml文件也只有以下内容：

server:

port: 9001



1. 新建主启动类文件，并且在主启动类上面添加注解@EnableHystrixDashboard



1. 要保证所有微服务都有以下maven依赖。

<!-- actuator监控信息完善 -->

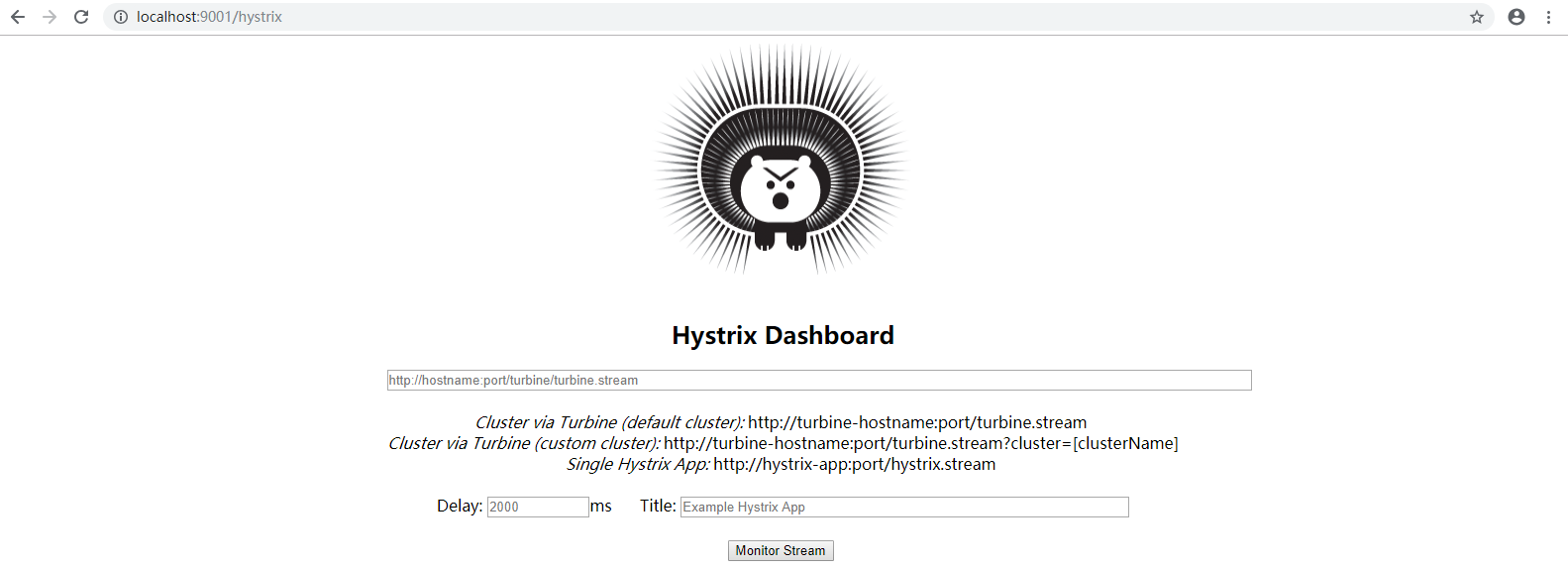
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

1. 启动这个新建的项目，浏览器访问地址： <http://localhost:9001/hystrix>



### 7.6.3、测试让“大管家”豪猪兄Hystrix监控服务的健康状况.

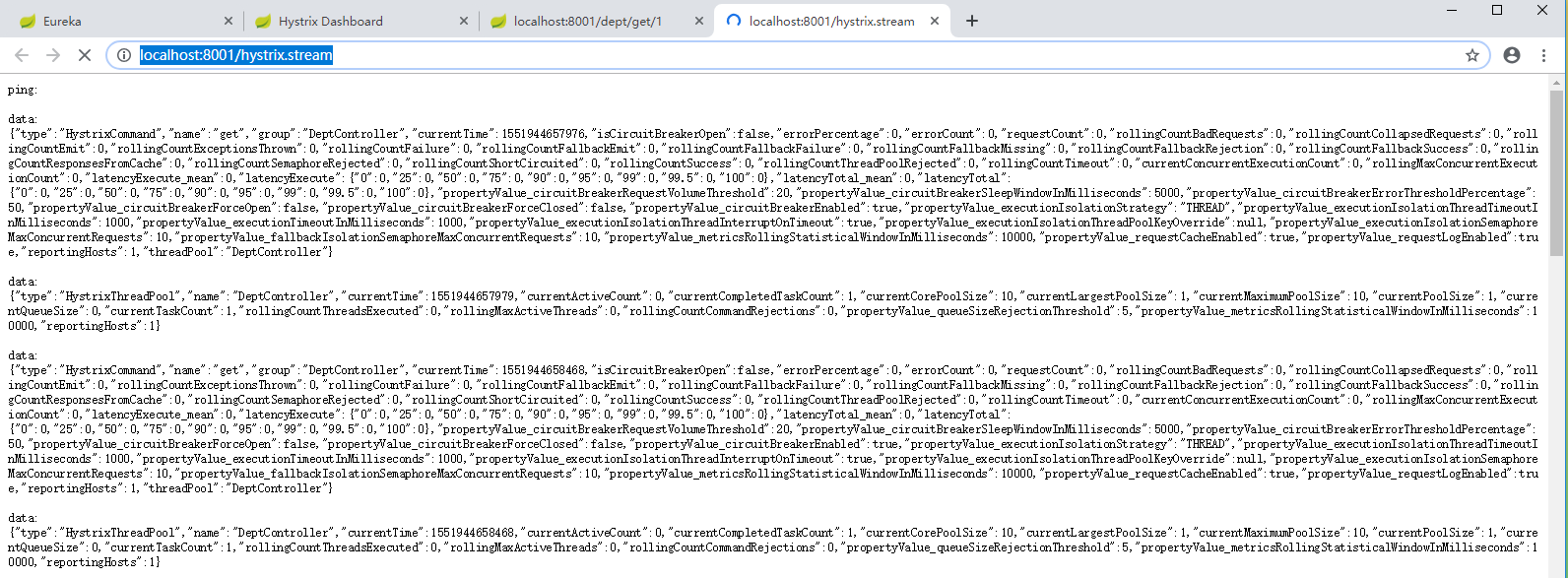
1、启动hystrix-dashboard让它去健康服务消费端

2、启动三个Eureka集群

3、启动服务提供者hystrix-8001服务提供者，这个是带服务熔断的，只写了一个get的，因为会耦合，因为异常处理也在里面。

<http://localhost:8001/dept/get/1> 查询正确的数据

<http://localhost:8001/hystrix.stream> 它会一直去ping这个微服务，获取数据，不会停。

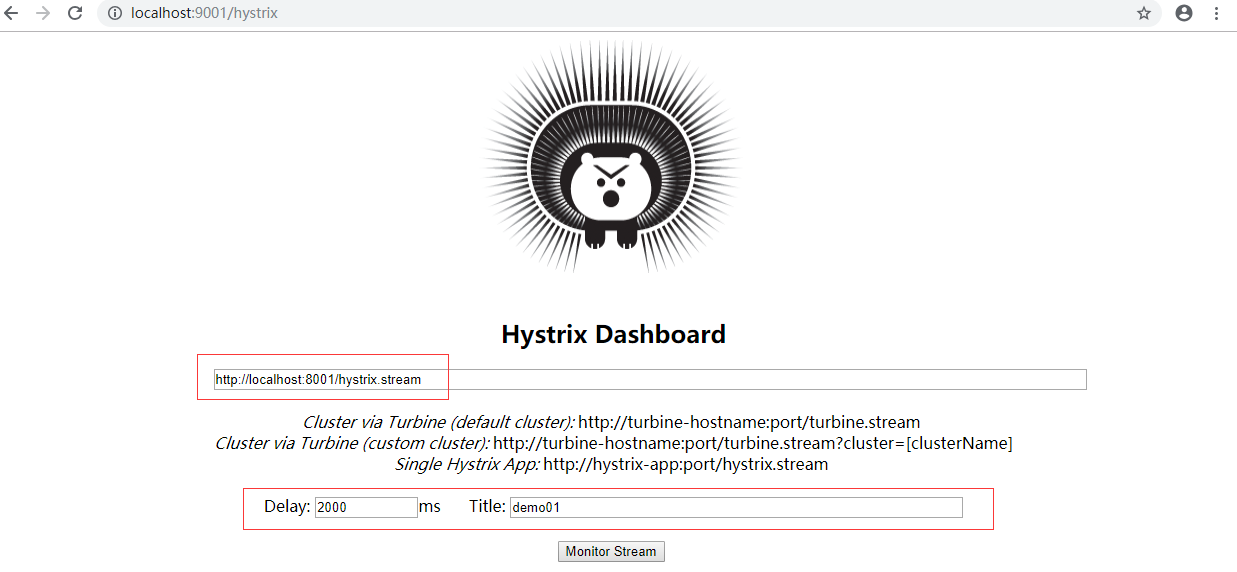


### 7.6.4、怎么监控怎么看？

填那个<http://localhost:9001/hystrix>链接的参数就可以打开监控页面了。

Delay：该参数用来控制服务器上轮询监控信息的延迟时间，默认是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Hystrix Strean之后的内退，默认会使用具体监控实例的URL，可以通过配置该信息来展示更合适的标题。



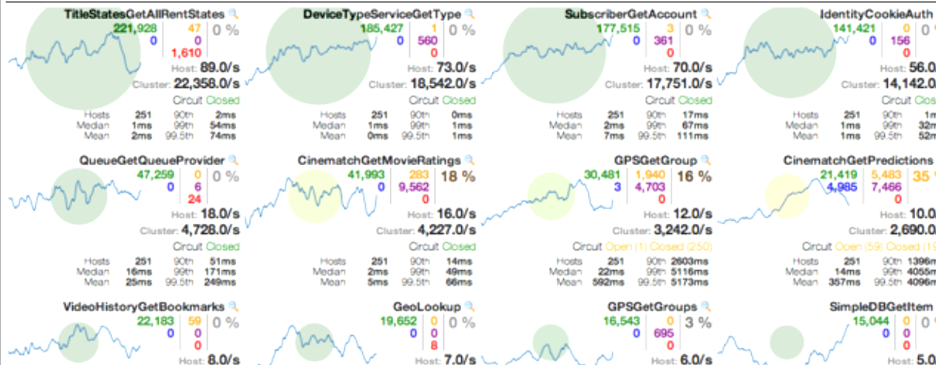
多次去发送这个请求：<http://localhost:8001/dept/get/1>



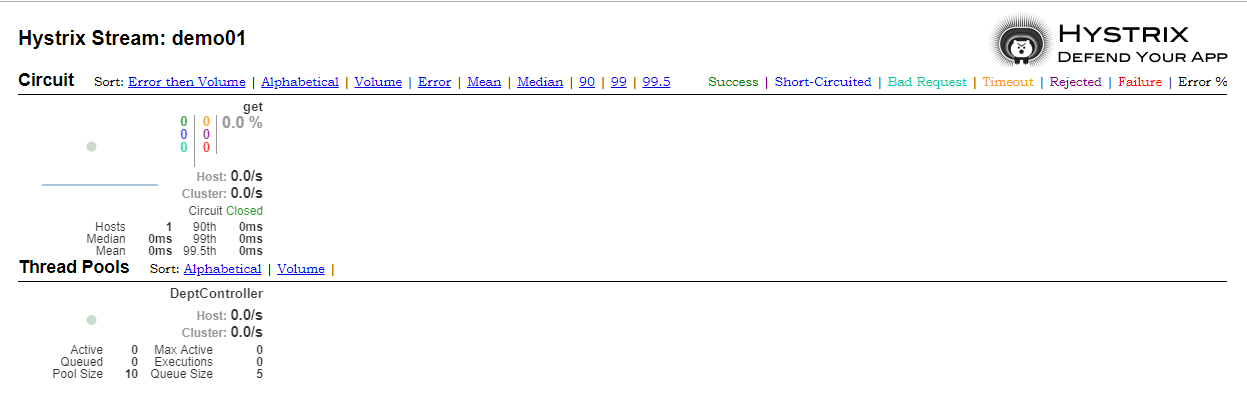
实心圆：一共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康从绿色<黄色<橙色<红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆也就越大。所以通过该实心圆的展示，就可以在大量实例中快速发现故障实例和高压力实例。

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

一般在企业的运维部门的画面：



然后这是我的………。。。



# 八、Zuul路由网管

## 8.1、概述

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和干预，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最终还是会注册进Eureka，Zuul提供=代理+路由+过滤三大功能。**

## 8.2、路由的基本配置

1、新建microservicecloud-zuul-gateway-9527路由网关项目

2、pom文件除了基本的内容之外，需要新增加以下内容：

<dependency>

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

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<dependency>

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

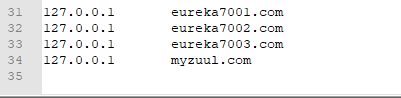
<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

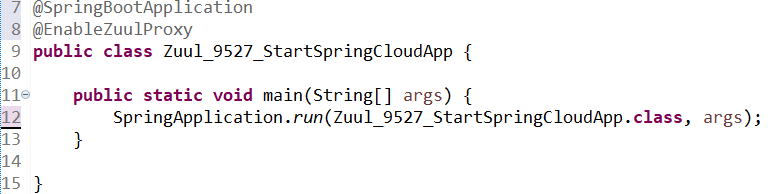
1. application.yml文件内容：



1. 去修改hosts文件，添加 127.0.0.1 myzuul.com 这里不太明白为什么是这个



1. 新建主启动类 Zuul\_9527\_StartSpringCloudApp，标注新注解@EnableZuulProxy

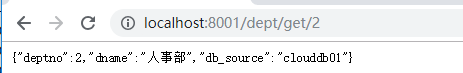


6、启动测试

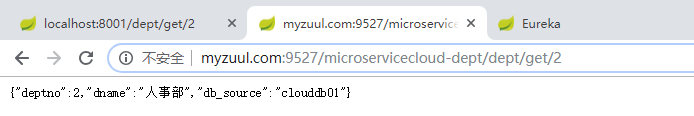
启动顺序：3个Eureka集群，一个8001，一个zuul

测试：

先测试不用路由的链接： <http://localhost:8001/dept/get/2>



再测试启用路由的链接： <http://myzuul.com:9527/microservicecloud-dept/dept/get/2>



## 8.3、路由访问映射规则

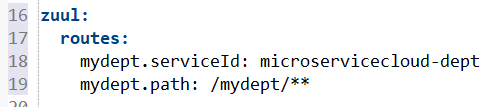
针对zuul-gateway-9527进行修改，就相当于把真实的微服务名字隐藏起来，对外暴露一个虚假的名字。

1、

没有修改之前的访问链接是：

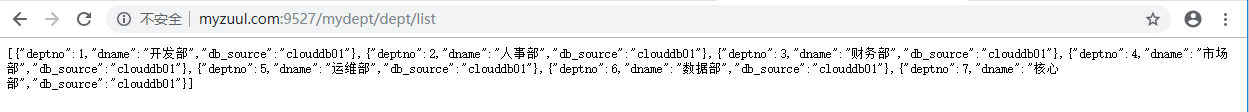
<http://myzuul.com:9527/microservicecloud-dept/dept/list>

修改yml文件，增加以下信息



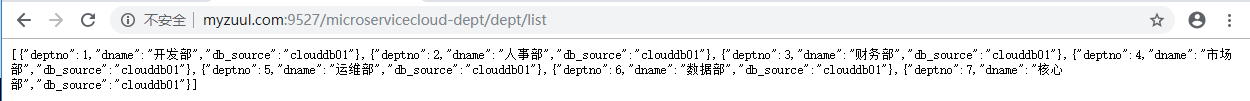
修改之后的访问链接是：

<http://myzuul.com:9527/mydept/dept/list>

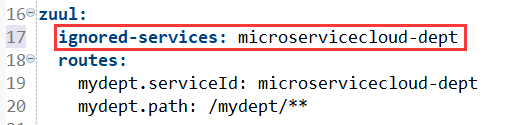


但是这个时候真实的名字还是能够访问：

<http://myzuul.com:9527/microservicecloud-dept/dept/list>



2、将原来的路径设置了不可以访问，yml文件中

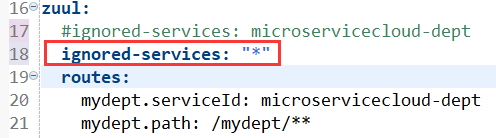


此时再用<http://myzuul.com:9527/microservicecloud-dept/dept/list>

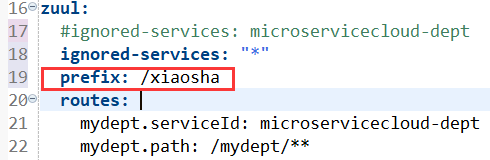
路径去访问就不再可行：



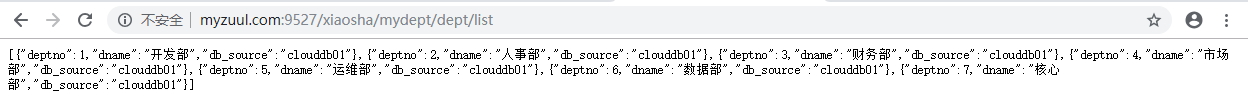
但这样只是忽略了一个微服务名称，当项目庞大起来的时候，就需要忽略全部，可能是因为分布式集群只有一个服务网关吧。如果想把真实的服务地址全部忽略，在yml文件中这样写即可：



Zull还可以配置统一的前缀：



访问路径就变成了： <http://myzuul.com:9527/xiaosha/mydept/dept/list>



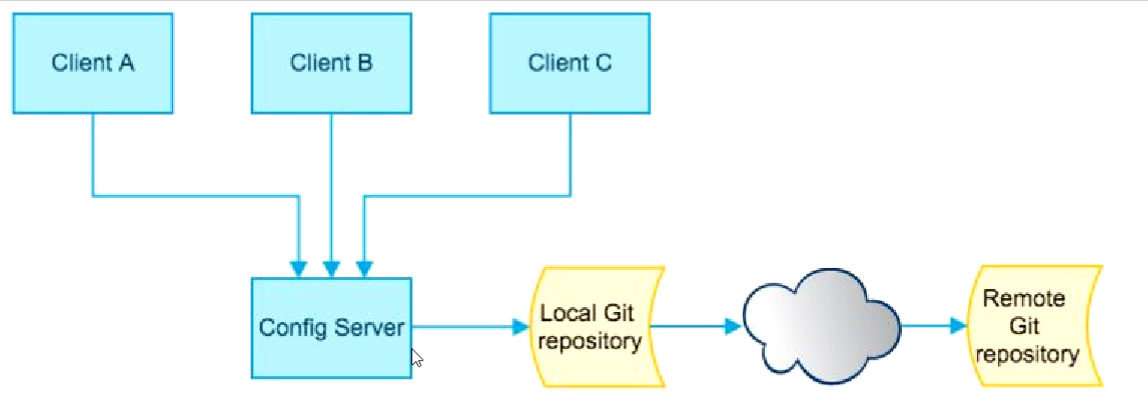
# 九、Spring Cloud Config分布式配置中心

## 9.1、基本概念

微服务意味着要将单体应用中的业务拆分成一个个的子服务，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 Server来解决这个问题。我们需要一套集中式配置文件的管理和落地。

### 9.1.1是什么？

Spring Cloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境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整体版本的管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

### 9.1.2、能干什么？

1、集中管理配置文件。

2、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/beat/release。

3、运行期间动态调整配置，不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自己的配置信息。

4、当配置发生变化的时候，服务不需要重新启动既可以感知到配置的变化并应用新的配置。

5、将配置信息以REST接口的形式暴露。

### 9.1.3、与GitHub整合

由于Spring Cloud Config默认是使用Git来存储配置文件，也有使用SVN的，所以推荐使用Git，而且使用的是http/https的访问形式。

## 9.2、Spring Cloud Config服务端的配置

1、在GitHub上新建一个仓库。

2、在本地文件夹下建一个application.yml文件，字符编码选择UTF-8。

spring:

profiles:

active:

- dev

---

spring:

profiles: dev

application:

name: microservicecloud-config-learn-dev

---

spring:

prifiles: test

application:

name: microservicecloud-config-learn-test

3、新建microservicecloud-config-config-3344配置中心Module

4、pom文件：

<dependencies>

<!-- springCloud Config -->

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<!-- 避免Config的Git插件报错：org/eclipse/jgit/api/TransportConfigCallback -->

<dependency>

<groupId>org.eclipse.jgit</groupId>

<artifactId>org.eclipse.jgit</artifactId>

<version>4.10.0.201712302008-r</version>

</dependency>

<!-- 图形化监控 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<!-- 熔断 -->

<dependency>

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

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-config</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-starter-test</artifactId>

</dependency>

<!-- 热部署插件 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

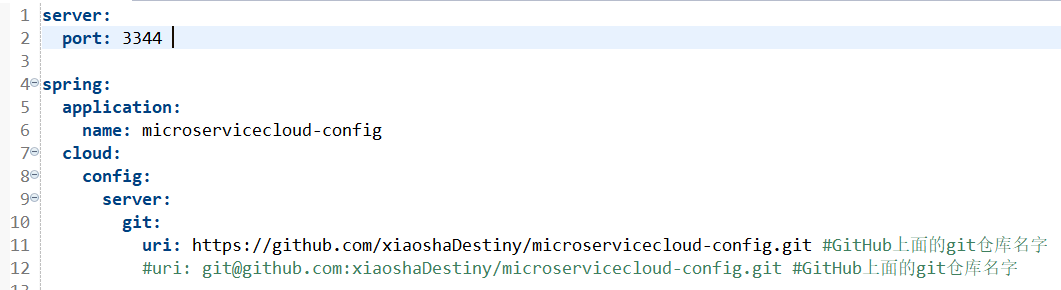
<artifactId>spring-boot-devtools</artifactId>

</dependency>

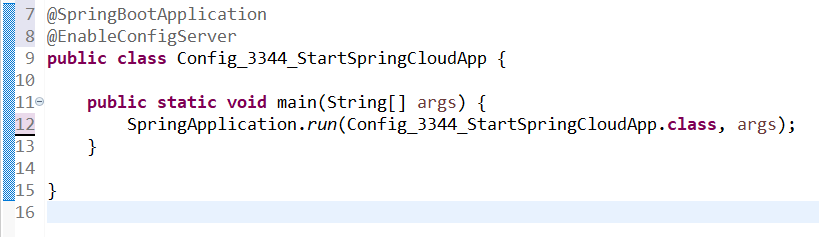
</dependencies>

</project>

5、application.yml文件



6、主启动类Config\_3344\_StartSpringCloudApp，并且在上面标注@EnableConfigServer

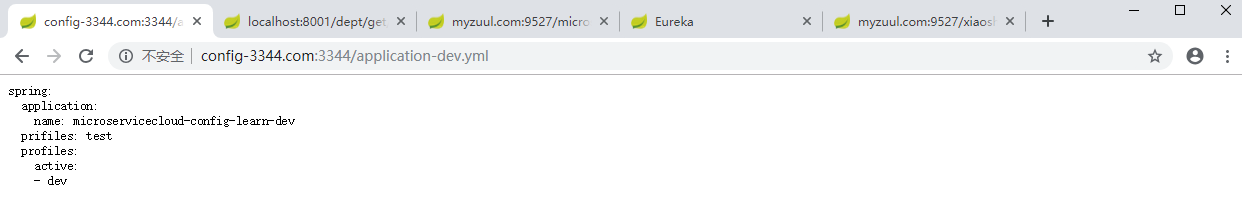


7、修改hosts文件 127.0.0.1 config-3344.com

8、测试通过Config微服务是否可以从Github上获取配置内容

只启动3344这个项目，测试链接：

开发环境： <http://config-3344.com:3344/application-dev.yml>

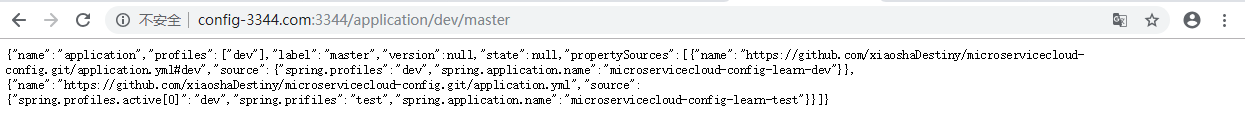


测试环境： <http://config-3344.com:3344/application-test.yml>

随便输的一个环境： <http://config-3344.com:3344/application-acvvv.yml>

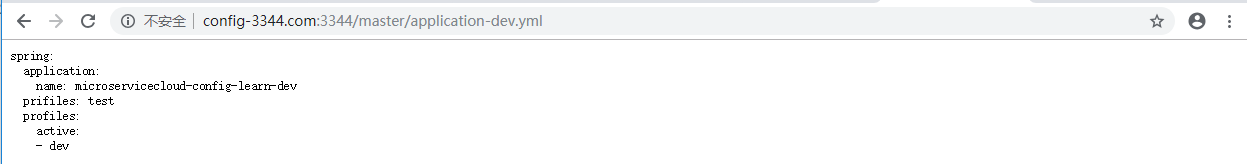
1. 返回json字符串

<http://config-3344.com:3344/application/dev/master>



1. 还可以把分支写在前面

<http://config-3344.com:3344/master/application-dev.yml>



## 9.3、Spring Cloud Config客户端配置与测试

1、新建一个microservicecloud-config-client.yml文件，上传到github的远程仓库上。

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microservicecloud-config-client

eureka:

client:

service-url:

defaultZone: http://eureka7001.com:7001/eureka/

---

server:

port: 8202

spring:

profiles: test

application:

name: microservicecloud-config-client

eureka:

client:

service-url:

defaultZone: http://eureka7001.com:7001/eureka/

2、新建一个microservicecloud-config-client-3355的Config 客户端Module

3、修改pom

<dependencies>

<!-- SpringCloud Config客户端 -->

<dependency>

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

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-config</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-starter-test</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

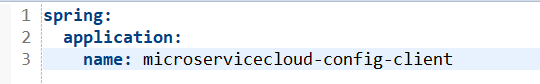
</project>

1. application.yml文件

spring:

application:

name: microservicecloud-config-client



1. **新建bootstrap.yml**

spring:

cloud:

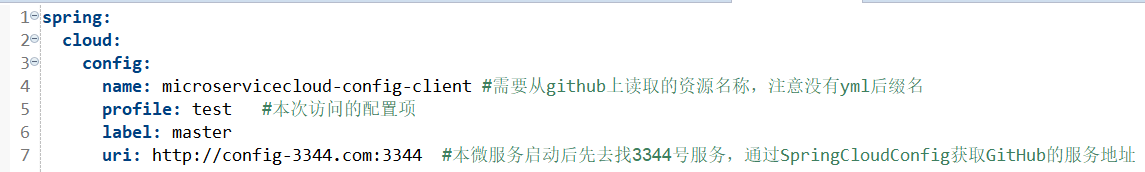
config:

name: microservicecloud-config-client #需要从github上读取的资源名称，注意没有yml后缀名

profile: test #本次访问的配置项

label: master

uri: http://config-3344.com:3344 #本微服务启动后先去找3344号服务，通过SpringCloudConfig获取GitHub的服务地址



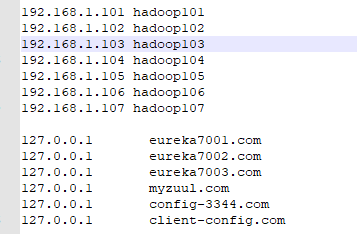
## 9.4、bootstrap.yml

**application.yml 是用户级别的资源配置项**

**bootstrap.yml 是系统级别的资源配置项，优先级更高**

SpringCloud会创建一个 Bootstrap Context作为Spring应用的Application Context的父级上下文。初始化的时候，Bootstrap Context 负责从外部资源加载配置属性并且解析配置。这两个上下文共享一个从外部获取的Environment。 Bootstrap属性有高优先级，默认情况下，它们不会被本地配置覆盖。Bootstrap context和Application Context有着不同的约定。所以增加了一个bootstrap.yml文件，保证 Bootstrap Context 和Application Context配置的分离。

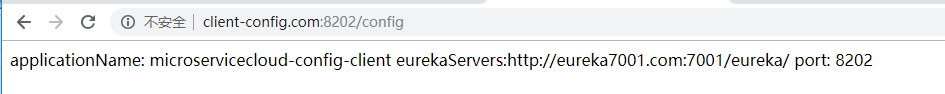
1. hosts文件



1. 启动测试

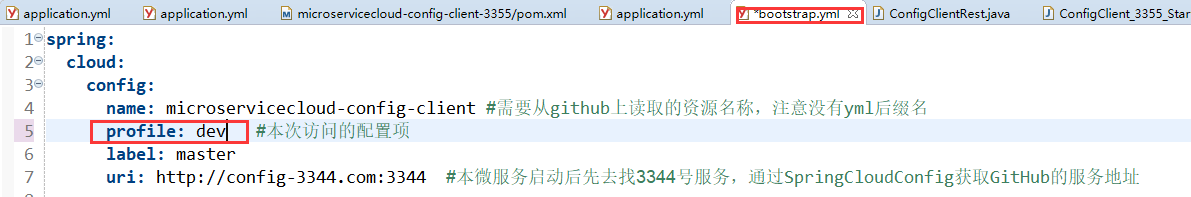
先启动3344，再启动3355，访问链接：

<http://client-config.com:8202/config>

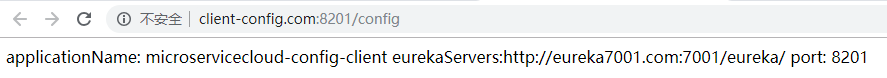


1. 测试dev

修改bootstrap文件：



<http://client-config.com:8201/config>



少了一节，没事，不遗憾，还要学的东西很多。

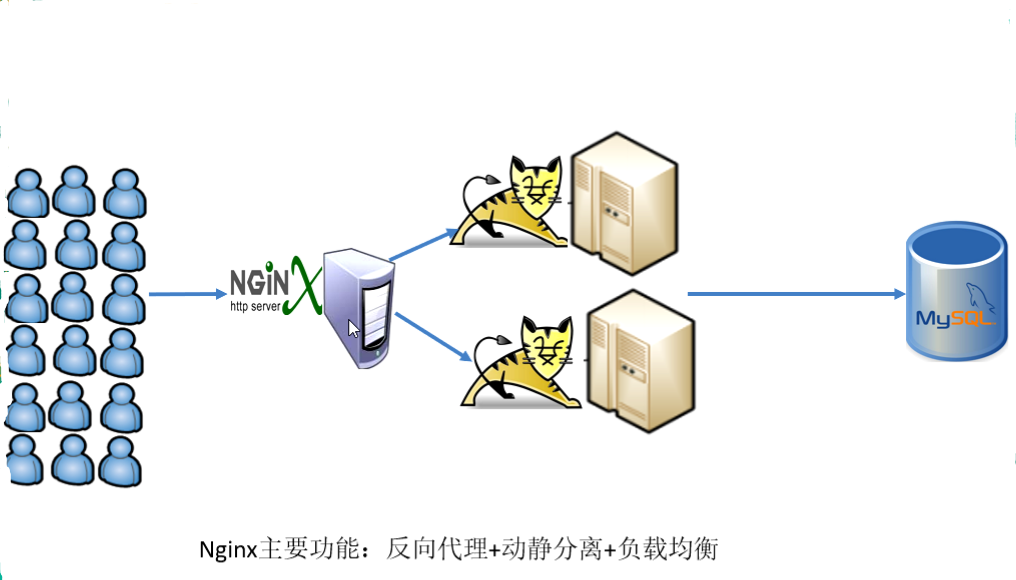
# 十、总结与展望

10.1、互联网架构的演变，这些技术怎么出来的？

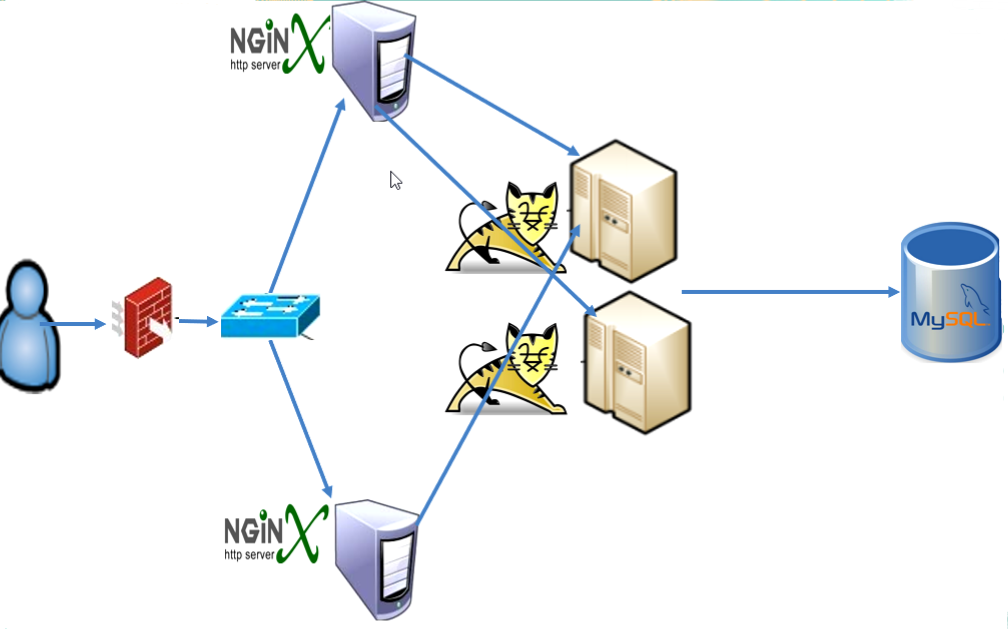
1. 单机集中构建网站
2. JSP和Servlet换成了SSM框架

随着访问量的增加，单台衣服武器已经无法满足需求，在假设数据库服务器没有压力的情况下，我们可以把应用服务器从一台变成了多台，把用户请求分散到不太的服务器中，从而提高负载能力。此时变成了Tomcat集群，当然就出现了一堆新的问题：

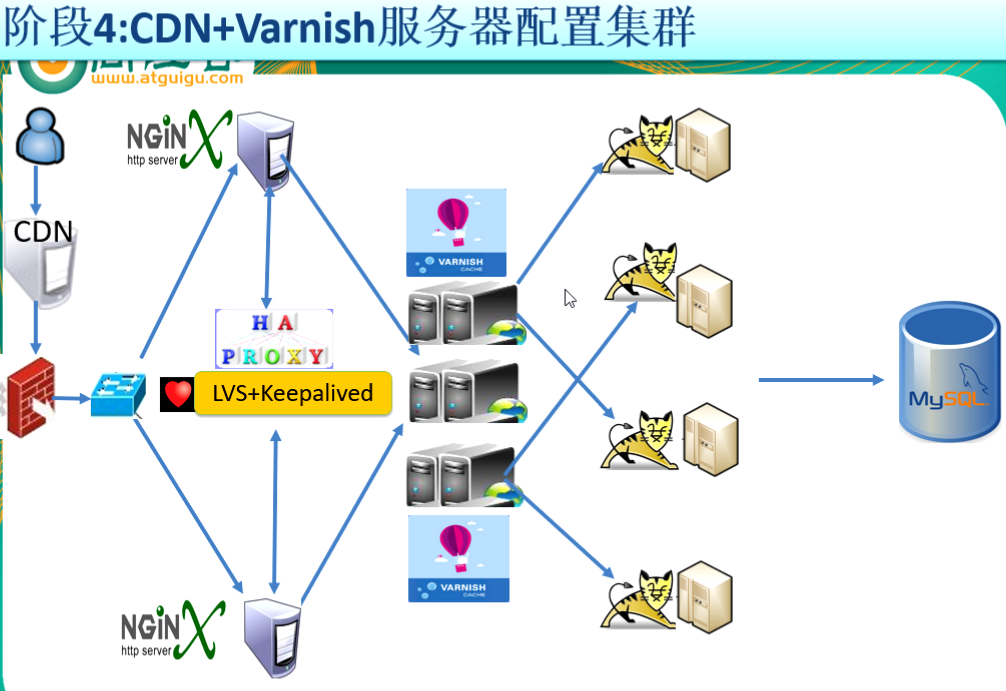
1. 用户的请求由谁来转发到具体的应用服务器。Nginx
2. 有什么样的转发算法？轮询，随机，最小访问，IP取模、取余数等
3. 应用服务器如何返回用户请求？直连，IP隧道
4. 用户如果每次访问到的服务器不一样，如何维护session的一致性？Nginx+Redis来解决session一致性的问题。

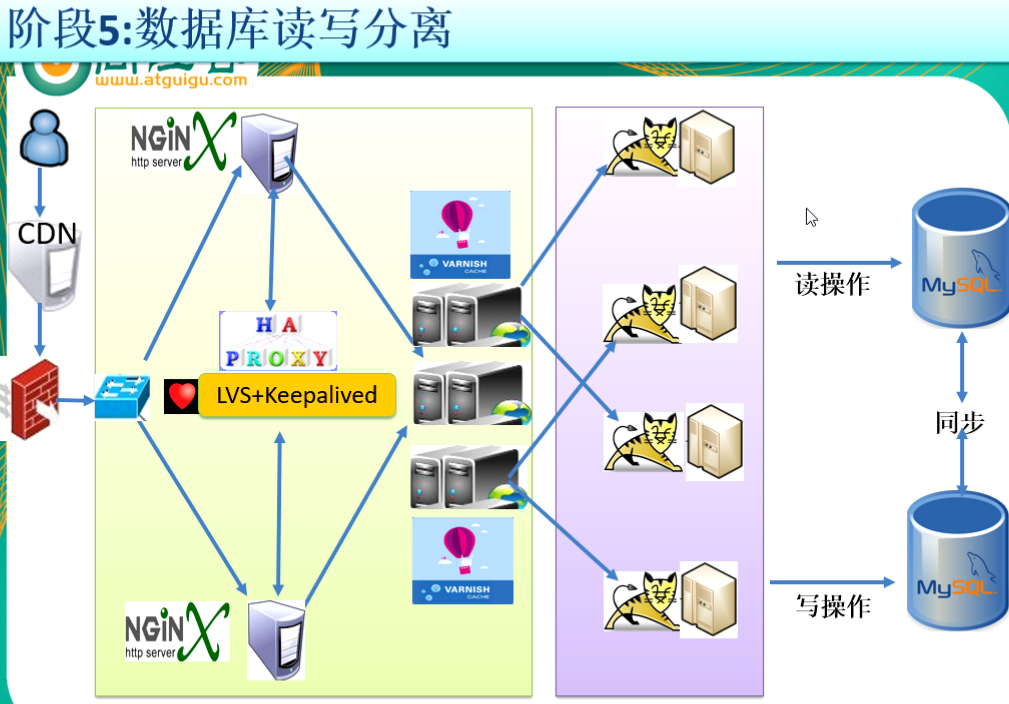


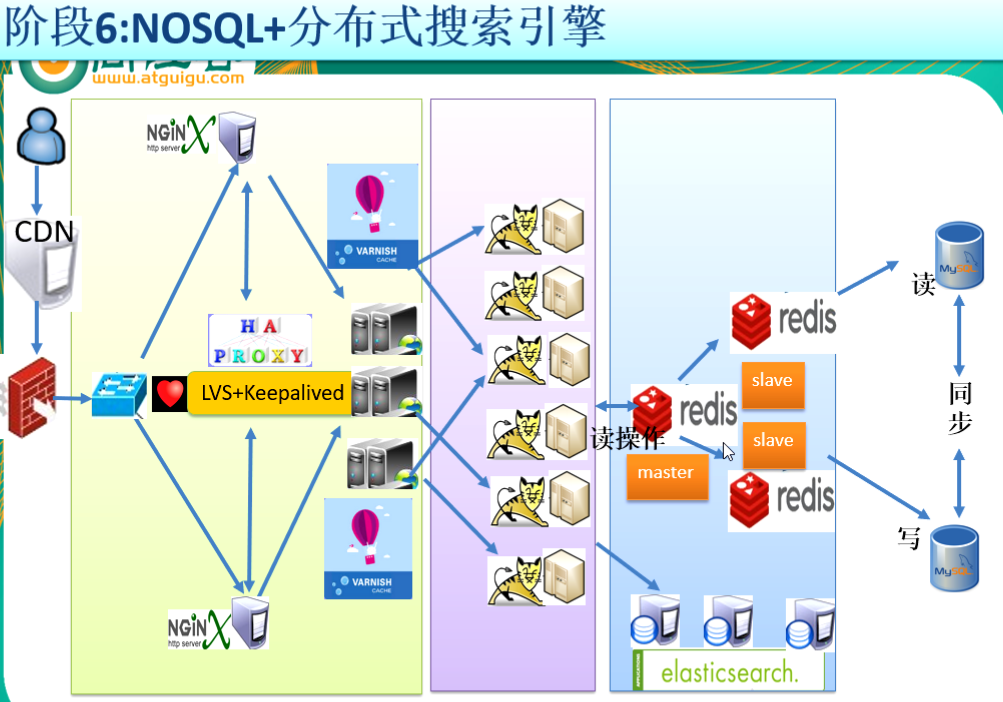
但是这样当Nginx宕机了以后就会崩溃，所以慢慢演化为Nginx+应用服务器配置集群+HA

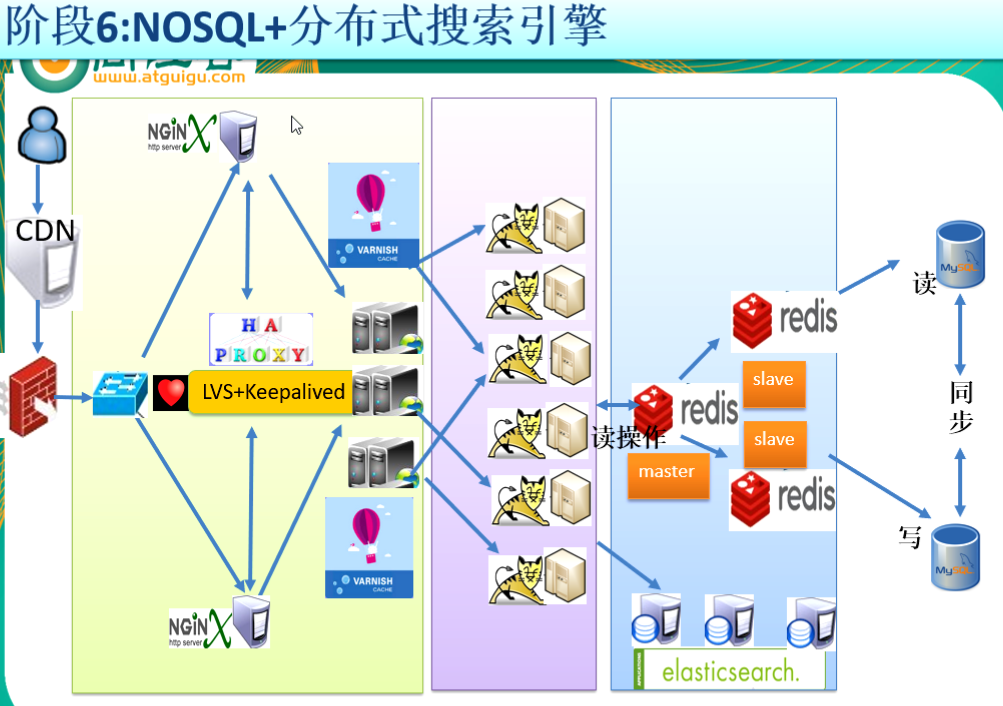


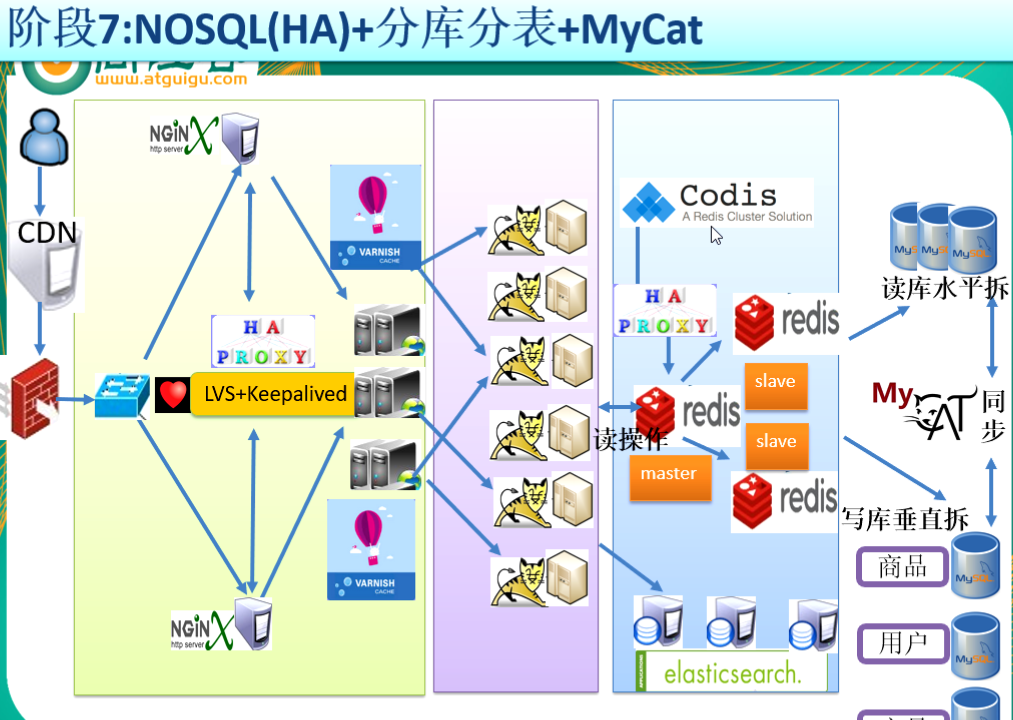
为了保证集群高可用，使用了LVS(Linux虚拟化)+Keepalived，用LVS实现负载均衡，用Keepalived实现高可用。

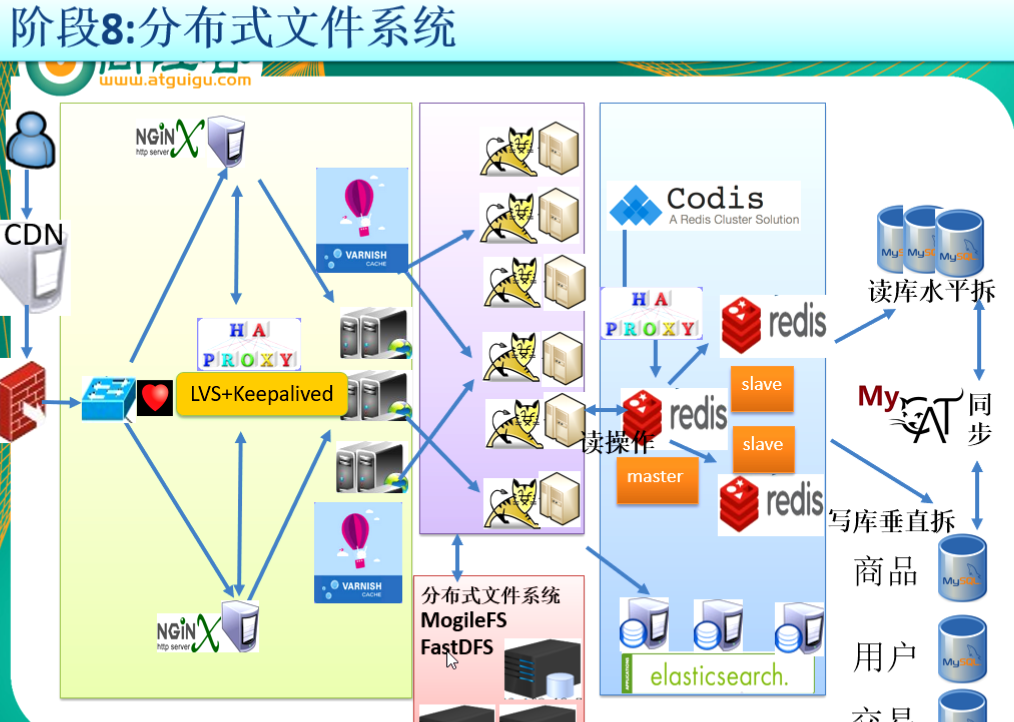


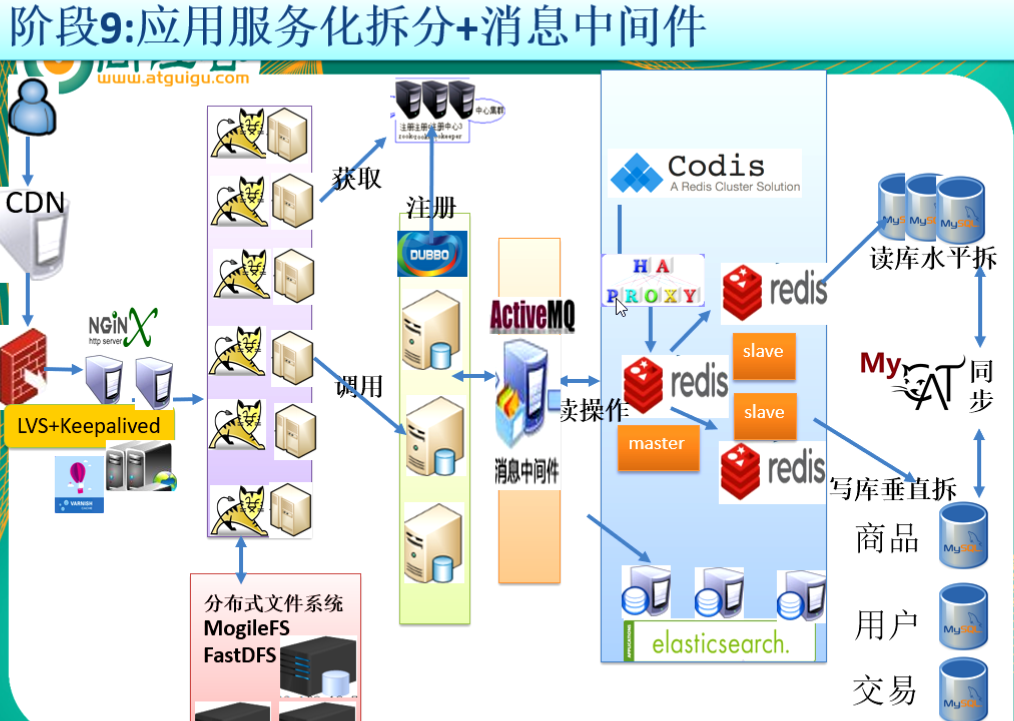














安全管理，shiro等。



# 附：踩坑

## 1、SpringCloud踩坑

1. 服务消费者和服务提供者都要映入web模块，不然会报这个错：Unregistering JMX-exposed beans on shutdown
2. Description Resource Path Location Type Archive for required library:'C:/develop/m2/org/codehaus/jettison/jettison/1.3.7/jettison-1.3.7.jar' in project 'microservicecloud-eureka-7001' cannot be read or is not a valid ZIP file microservicecloud-eureka-7001 Build path Build Path Problem

老问题，还是Jar包不对，然后在7001项目上出现了一个红色感叹号，但是没有报错，启动就会报找不到主类的错误。把问题Jar包删除，重新Update下载能解决。

1. API工程的pom文件莫名其妙报错，重新install之后再去Update一下。
2. 启动顺序有讲究，不能瞎鸡儿乱启动服务。

## 2、基本操作踩坑

1. 在使用RestTemplate的时候一定要注意那三个参数分别代表什么意思，List反射的时候不能带泛型。
2. MyBatis的配置里面有实体类entities的路径，记得要修改。

API Gateway 路由网管zuul

Breaker dashboard 熔断的服务监控Hystrix断路器

Service registry 服务注册

主管服务注册与发现的Eureka

主管负载均衡的Ribbon 还有Feign也可以 怎么选呢？

断路器Hystrix

路由网管zuul

分布式配置中心SpringCloudConfig

面试常问：

Eureak和Zookeeper的区别？

SpringCloud和Dubbo的区别？谈谈对这两个框架的认识？