# 面试问题

值传递和引用传递

高可用性”（High Availability）通常来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性。

Svn

枚举类型

hashMap实现原理

solr增量更新

Redis数据类型

MySQL分析SQL语句

MybatisPuls（Mybatis增强版）

消息中间件

不想创建实体类，又要查询数据该怎么办

Struts2和SpringMvc的区别

JDBC和MyBatis的区别

MyBatis的好处

Java虚拟机里面栈和队列

Dubbo和SpringCloud的区别：

Dubbo因为停更了5年，很多技术只能使用第三方

SpringCloud完完全全使用Spring社区里的技术做到真正的一站式开发

如果说要对比的话那么Dubbo可以说是一台组装机，而SpringCloud则是一体机

O2O是什么:线上和线下

服务雪崩：

a依赖了b，b又依赖了c...，一系列链式依赖，其中一环出错了导致请求响应时间太长而一直卡在那里，同一时间多个请求访问导致整个系统崩溃。解决方案：服务熔断，服务降级

微服务跟分布式的区别：

微服务是一种架构，分布式是一种系统部署方案

服务熔断和访问降级的区别

服务熔断是：

举个例子解释，生活中每家每户都在用电，小明家的电线因为故障导致了小明家停电了。而小李、小张家的电是正常使用的。电力公司没有因为小明家有故障线路而停掉其他人家的电，同时小明家没有使用有故障的电路的电。这时即为熔断。熔断的目的是当A服务模块中的某块程序出现故障后为了不影响其他客户端的请求而做出的及时回应。

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

单个方法，单个接口出现问题，对单个方法进行熔断

服务降级是：

举个例子解释，我们去银行排队办理业务，大部分的银行分为普通窗口、特殊窗口（VIP窗口，老年窗口）。某一天银行大厅排普通窗口的人巨多。这时特殊窗口贴出告示说某时刻之后再开放。那么这时特殊窗口的工作人员就可以空出来去帮其他窗口办理业务，提高办事效率，已达到解决普通窗口排队的人过的目的。这时即为降级，降级的目的是为了解决整体项目的压力，而牺牲掉某一服务模块而采取的措施。

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

整个（单个服务）服务不可用了，整体

# SpringCloud

注意写的时候一定一定一定要注意依赖版本（巨坑）

内存最低8G不然还是别搞了

测试的时候启动服务一定要按顺序启动不然容易出错

开发的三个阶段

单机版 ：耦合度搞会互相影响 All in one

分布式 ：每个模块单独运行，不会互相影响

开发模式：

前后端分离：只需要管理后端，给前端H5工程师就按照约定Rest地址+输入参数格式和报文约定+输出参数

## 全栈工程师：

全部自己定义

微服务：一个个可以独立运行小Module，强调的是一个个个体，每个个体完成一个具体的任务或者是功能

微服务架构：微服务体系结构风格是一种将单个应用程序开发为一个oaadsuite的方法，每个oaadsuite运行在itswaotend中，与nhtwemec hanrm通信，通常是一个HTTP资源。这些服务的集中管理非常少，可以用不同的编程编写使用不同的数据存储技术。

## SpringCloud是什么：

分布式微服务架构下的一站式解决方案，是各个微服务框架落地技术的集合体。

## SpringCloud和SpringBoot的关系：

SpringBoot是微观是微服务里面的一个个服务

SpringCloud是宏观（微服务）

他们存在一个依赖关系SpringBoot可以独立使用开发项目SpringCloud离不开SpringBoot

SpringBoot专注于快速开发方便的开发单个个体微服务

SpringCloud是关注全局微服务协调治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来。

去哪里下载：

能干嘛：

怎么玩

SpringCloud国内使用的情况

什么是微服务:模式架构或者说是一只架构风格，它将单一的应用划分为一组微小的服务，每个服务运行在自己的进程中

## 微服务之间是如何独立通讯的:

## springCloud和Dubbo有哪些区别

微服务SpringCloud是基于RESTful调用，Dubbo是基于RPC远程过程调用

## SpingBoot和SpringCloud谈谈理解

## 什么是服务熔断？服务降级

## Dubbo是怎么到SpringCloud的？

## 微服务的优缺点是什么：

优点：每个服务足够内聚，足够小，代码容易理解这样能聚焦一个指定的业务功能或者业务需求，开发简单，开发效率高，松耦合，可以用不同的语言开发，易于和第三方集成，微服务只是业务逻辑的代码，不会和HTML,CSS或者其他界面组件混合。每个微服务都有自己的储存能力，可以有自己的数据库，也可以有统一的数据库。

缺点：复杂性，运维难，系统部署依赖，服务器通信成本，数据一致性，系统集成测试，性能监……

微服务技术站有哪些:（微服务技术栈：多种技术的集合体）():条目

Springboot、Spring、SpringgMVC (服务开发)

Neflix公司的Archaius、阿里的Diamond等（服务器配置与管理）

Eureka、Consul、Zookeeper等(服务注册与发现)

Rest、RPC、Grpc(服务调用)

Hystrix、Envoy等（服务熔断器）

Ribbon、Nginx等（负载均衡）

Feign等（服务接口调用

客户端服务的简化工具）

Kafka、RabbitMQ、ActiveMQ等（消息队列）

SpringCloudConfig、Chef等（服务配置管理中心）

Zuul等（服务路由API网关）

Zabbix、Nagios、Metrics、Spectator等（服务监控）

Zipkin、Brave、Dapper等（全链路追踪）

Docker、OpenStack、Kubernetes等(数据流操作开发包)

SpringCloudBus（事件消息总栈）

……

为什么会选择SpringCloudBus作为微服务框架：

选型依据：

整体解决方案和框架的成熟度

社区热度

可维护性

学习曲线

Eureka和zookeeper的可以提供服务注册与发现功能两者的区别

SpringCloud中文网：<https://www.springcloud.cc/spring-cloud-netflix.html>

SpringCloud地址：<https://spring.io/projects/spring-cloud>

微服务创始人博客：<https://martinfowler.com/articles/microservices.html>

SpringCloudAPI说明：<https://www.springcloud.cc/spring-cloud-dalston.html>

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

SpringCoud中文网: <https://springcloud.cc>

## 调用Rest服务，需要RestTemplate:

完整内容参考项目80

提供了多种便捷的访问远程Http访问的方法

是一种简答便捷的访问restful服务模板类，是Spring提供的用于访问Rest服务的客户端模板工具类（把rest的调用封装了）

使用restTemplate访问restful接口非常简单

（url，requestMap，ResponseBean.class）这三个参数分别代表Rest请求地址、请求参数、HTTP响应转化被转换成的对象类型（返回值的类，告诉RestTemplate该以什么类型返回）

## 微服务技术理解：

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。

从技术角度看技术一种小而独立的处理过程，类似进程概念，能够自行单独启动或者销毁，拥有自己独立的数据库。

# Eureka服务注册与发现：

## Eureka是什么 Netflix在设计Eureka时遵守的就是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，以实现云端中间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重要的，有了服务发现与注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了。功能类似于dubbo的注册中心，比如Zookeeper

## Eureka的基本框架：

SpringCloud非主流Netflix公司开发的Eureka模块来实现服务注册和发现

Eureka采用了C-S的设计架构。EurekaServer作为服务注册功能的服务器，它是服务注册中心。

## EurekaServer提供服务注册服务：

（CS架构）

各个节点启动后，会EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到

EurekaClient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（round-robin）负载算法的负载均衡器，在应用启动后，将会EurekaServer发送心跳（默认周期为30秒）。如果EurekaServer在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除（默认90秒）

## Eureka三大角色：

EurekaServer 提供服务注册和发现

ServiceProvider（服务提供方） 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

ServiceConsumer （服务消费者） 服务消费方从Eureka获取注册服务列表，从而能够消费服务

## 引入cloud的一个新技术组件(服务端)

基本有三步

完整内容产考7001

1. 新增一个相关的maven坐标

Eureka:

*<!-- eurekaServer服务端 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-eureka-server</**artifactId**>  
</**dependency**>

Yml加入配置：

*#单机版*

**server:  
 port:** 7001  
**eureka:  
 instance:  
 hostname:** localhost *#eureka服务端的实例名称* **client:  
 register-with-eureka: false** *#false表示不向注册中心注册自己* **fetch-registry: false** *#false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务* **service-url:  
 defaultZone:** http://${**eureka.instance.hostname**}:${**server.port**}/eureka/ *#设置与EurekaServer交互的地址查询服务和注册服务都需要依赖这个地址  
#这里的${}都是引用的上面的配置*

1. 在主启动类上面，标注的启动该新组件技术的相关注解标签

@EnableEurekaServer(代表服务端)

1. java业务逻辑编码

## 将已经有的微服务注册进eureka中心:

完整内容参考项目8001

在微服务提供者端加入依赖（Client）

*<!-- 将微服务provider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**>

在yml加入配置:

**eureka:（根）  
 client:** *#客户端注册进eureka服务列表内* **service-url:   
 defaultZone:** <http://localhost:7001/eureka>

(这里是7001里面配置文件里配置的通过这里进行关联)

在启动类上加上@EnableEurekaClient(代表客户端)

*//本服务启动后会自动注册进eureka服务中*

## 微服务配置和完善:

在Instances currently registered with Eureka里面显示的东西

### 主机名称服务名称的修改：

**instance:（子）  
 instance-id:** microservicecloud-dept8001 *#在eureka内的Status的链接名*

*加在eureka后面*

### 访问信息有IP信息提示：

**（子）**

**prefer-ip-address: true** *#eureka页面中鼠标移到上一步的链接是显示ip 加在上一步的下面*

### 微服务info内容详细信息:

在需要完善的微服务里面加上依赖

*<!—监控信息完善 主管监控和信息配置 -->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>

在总的父工程修改pom.xml添加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**>$</**delimit**>(可以解读进行$中的内容)  
 </**delimiters**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

修改子工程中的配置文件yml

**info:（根）***#这里是进入eureka里面单击服务连接进入的页面配置* **app.name:** atguigu-microservicecloud *#服务名称*  
 **company.name:** [www.atguigu.com](http://www.atguigu.com) *#官网网址*  
 **build.artifactId:** $project.artifactId$ *#域名*  
 **build.version:** $project.version$ *#版本*

### eureka自我保护：

在某时刻某一个微服务不可用了，eureka不会立刻清理，依旧会对该微服务的信息进行保存

默认情况下，如果EurekaServer在一定时间内没有收到某个微服务的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生时，微服务与EurekaServer之间无法进行通信，以上行为可能变得非常危险了——因为微服务本身是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入模式，EurekaServer就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会销毁任何微服务）。当网络故障恢复后，该EurekaServer节点就会退出自我保护模式 （设计哲学：宁可保留错误的服务注册信息，也不盲目注册任何可能健康的服务实例），可以让Eureka集群更加的健壮、稳定

#### 关闭保护模式：

在SpringCloud中，可以使用：

eureka.server.enable-self-preservation=false

禁用自我保护模式（不推荐禁用）

## 服务发现Discovery：

参考8001

在服务提供者的controller中添加

@Autowired  
**private** DiscoveryClient **client**; //服务发现

@RequestMapping(value = **"/dept/discovery"**,method = RequestMethod.***GET***)  
**public** Object discovery(){  
 List<String> list = **client**.getServices(); *//获取Eureka里面注册的微服务* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*"**+list);  
 List<ServiceInstance> srvList = **client**.getInstances(**"MICROSERVICECLOUD.DEPT"**);*//获取指定名称的微服务* **for** (ServiceInstance element:srvList){  
 System.***out***.println(element.getServiceId()+**"\t"**+element.getHost()+**"\t"**+element.getPort()+**"\t"**+element.getUri());  
 *//分别对应 ID 主机名 端口 访问地址* }  
 **return this**.**client**;  
}

在启动类上添加 @EnableDiscoveryClient *//开启服务发现*

进行测试

在消费者端添加

*//测试@EnableDisconveryClient，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**);  
}

然后进行测试

## Eureka集群配置

集群就是在不同的机器上面配置一套相同的服务来对外进行服务（高可用）

创建几个项目将依赖导入

*<!-- eurekaServer服务端 -->* <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-eureka-server</**artifactId**>  
 <**version**>1.4.5.RELEASE</**version**>  
 </**dependency**>  
 *<!-- 热部署 -->  
<!-- <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>Springloaded</artifactId>  
 </dependency>-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
 </**dependency**>

配置启动类

@SpringBootApplication  
@EnableEurekaServer *//启动该EurekaServer组件技术的相关注解标签***public class** Eureka\_7001\_App {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Eureka\_7001\_App.**class**,args);  
 }  
}

配置映射

找到C:\Windows\System32\drivers\etc路径下的hosts文件

修改映射配置添加进hosts文件：

127.0.0.1 eureka7001.com

127.0.0.1 eureka7002.com

127.0.0.1 eureka7003.com

同一个名字有三个不同的别名

修改yml文件：

*#集群版***server:  
 port:** 7001  
**eureka:  
 instance:  
 hostname:** eureka7001.com *#eureka服务端的实例名称在C:\Windows\System32\drivers\etc配置了实际上就是localhost* **client:  
 register-with-eureka: false** *#false表示不向注册中心注册自己* **fetch-registry: false** *#false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务* **service-url:** *#单机 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与EurekaServer交互的地址查询服务和注册服务都需要依赖这个地址* **defaultZone:** <http://eureka7002.com:7002/eureka/,http://eureka7003.com/eureka/> *#集群内有多少个就配多少个因为在自己的服务里面使用这里是和其他两个服务进行关联*

修改服务提供者的yml（8001）

**eureka:  
 client:** *#客户端注册进eureka服务列表内* **service-url:** *#当Eureka是单击版的时候 defaultZone: http://localhost:7001/eureka  
 #集群* **defaultZone:** <http://eureka7001.com/eureka/,http://eureka7002.com/eureka/,http://eureka7003.com/eureka/>

启动服务

在启动的过程中可能会报错

Caused by: java.net.SocketTimeoutException: Read timed out

没关系这个只是连接找到了只是没有数据返回所有超时了才报错是因为服务还没有启动完可以忽略

访问：

[http://eureka7001.com:7001](http://eureka7001.com:7001/)

[http://eureka7002.com:7002](http://eureka7002.com:7002/)

[http://eureka7003.com:7003](http://eureka7003.com:7003/)

## 作为服务注册中心，Eureka比Zookeeper好在哪里

### ACID (关系型数据库):

A (Atomicity)原子性

C (Conslstency)一致性

I（Isolation）独立性

D (Durability) 持久性

### CAP (NOSQL数据库):

C:Consistency (强一致性)

A:Availability (可用性)

P:Partition tolerance (分区容错性)

CAP的核心理论是：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，因此，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，通常在可扩展性上不太强大

CP – 满足一致性，分区容忍性的系统，通常性能不是特别高

AP – 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些

分布式项目一般都是CP或者是AP

因为由于当前网络硬件肯定会出现延迟丢包等问题，所以分区忍性必须要实现

所以我们只能在一致性和可用性之间进行权衡，没有NOSQL系统能同时保证这三点

问题：针对双11当天，你觉得你来选择的话？

你选AP?CP?

答案：AP

理由：P绝对要因为是分布式项目，A当天数据访问量大为了网站不崩溃所以选AP,等到双11过去了，再改回CP

各个NOSQL数据库特性:

Redis,MongoDB,HBase: CP

CouchDB,Cassandra,DynamoDB,Riak: AP

RDBMS: CP

Zookeeper保证的是CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用也就是说，服务注册功能对可用性的要求高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长了，30~120s，且选举期间这个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

Zk集群失去master节点是较大概率会发生的事情，虽然服务能够最终恢复，但是漫长的选举时间导致的注册时间不可用是不能容忍的。

Eureka则是AP

Eureka看明白了一点，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注册或发现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，就能保证注册服务可用（保证可用性），只不过查的的信息可能不是最新的（不保证数据的一致性）。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过80%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

# Ribbon负载均衡

### 是什么：

Ribbon是基于NetflixRibbon实现的一套客户端，负载均衡的工具。

简单的，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LoadBalancer（简称LB）后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。

### 能干嘛：

LB（负载均衡）：

LB,即负载均衡（LoadBalance），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.

常见的负载均衡有软件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的负载均衡算法可以自定义。

集中式LB:

偏硬件

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

进程式LB:

偏软件

将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址

## Ribbon配置初步（客户端）

完成真正的通过微服务名字从eureka上找到并访问服务：

在消费者模块里加上Ribbon的相关依赖

*<!-- Ribbo相关依赖 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-eureka</**artifactId**>*<!-- 后面没跟东西如Server说明这个一个客户端，因为Ribbon需要和Eureka结合使用 -->*</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-ribbon</**artifactId**>*<!-- Ribbon -->*</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>*<!-- 使Ribbon和Eureka联系的中间件 -->*</**dependency**>

修改yml配置

**eureka:  
 client:  
 register-with-eureka: false  
 service-url:  
 defaultZone:** http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

在ConfigBean配置类的getRestTenplate方法上加上@LoadBalanced表示开启客户端负载均衡（客户端每次去访问Eureka的时候自带负载均衡）

@LoadBalanced *//SpringCloud Ribbon是基于Neetflix Ribbon实现的一套客户端负载均衡工具*

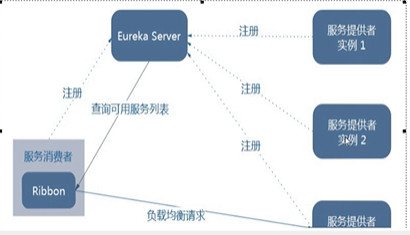
在启动类上加上:

@EnableEurekaClient*//和Eureka进行整合*

修改controller

*//服务提供方端口  
//private static final String REST\_URL\_PREFIX="http://localhost:8001"; //不规范因为这是直接调用的服务提供者***private static final** String ***REST\_URL\_PREFIX***=**"http://MICROSERVICECLOUD-DEPT"**; *//通过微服务名调用Eureka上的微服务这里的名字是微服务提供者里面定义的微服务名字*

### Ribbon负载均衡



Ribbon在工作时分成两步

第一步选择EurekaServer，它优先选择同一个区域内负载均衡较少的server

第二步再根据用户指定的策略，再从server取到服务注册列表中选择一个地址。

其中Ribbon提供了很多种策略：比如轮询，随机和根据响应时间权限

新建两个服务提供者：8002 8003 内部配置参考8001配置文件

修改端口号

新建8002 8003的数据库（每个微服务提供者都来自不同的数据库）

修改yml中的数据库名称

（对外暴露的服务访问名字不能修改必须是同一个）

启动测试Ribbon默认采用轮询的方式进行负载均衡通过消费者来访问

### Ribbon核心组件IRule

IRule：根据特定算法中从服务列表中选取一个要访问的服务

自带了7种算法：

RoundRobinRule 轮询

RandomRule 随机

AvailbilityFilteringRule 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表安装轮询策略进行访问

WeightedResponseTimeRule 根据平均响应时间计算所有的服务权重，响应时间越快服务权重越大被选中的概率越高，刚启动如果统计信息不足，则使用RoundRobinRule策略，等统计信息足够，会切换到WeightedResponseTimeRule

RetryRule 先按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定时间内会进行重试，获取可用的服务

BestAvailableRule 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务

ZoneAvoidanceRule 默认规则，复合判断server所在区域的性能server的可用性选择服务器

更改负载均衡算法

在消费者配置类中加上

@Bean  
**public** IRule myRule(){  
 *//return new RoundRobinRule();//达到的目的，用我们重新选择的随机算法替代默认的轮询* **return new** RandomRule();*//随机*}

### Ribbon自定义算法

在消费者启动类上加上

@RibbonClient(name = **"microservicecloud.dept"**,configuration = MySelfRule.**class**)*//定制负载均衡算法*

官方文档明确给出了警告:

这个自定义配置类（MySelfRule）不能放在@ComponentScan所扫描的当前包下及子包下（就是不能和SpringBootApplication同包，不能让它扫描到），否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享，也就是说我们达不到特殊化定制的目的了

MySelfRule配置如下:

@Configuration  
**public class** MySelfRule {  
 @Bean  
 **public** IRule myRule(){  
 **return new** RandomRule();*//Ribbon默认是轮询，我自定义为随机* }  
}

问题:依旧使用轮询策略，但是加上新需求，每个服务器要求被调用5次。也即以前是每台机器一次，现在是每台机器5次

产考源码（地址）：<https://github.com/Netflix/ribbon/blob/master/ribbon-loadbalancer/src/main/java/com/netflix/loadbalancer/RandomRule.java>

**package** config;  
  
**import** com.netflix.client.config.IClientConfig;  
**import** com.netflix.loadbalancer.AbstractLoadBalancerRule;  
**import** com.netflix.loadbalancer.ILoadBalancer;  
**import** com.netflix.loadbalancer.Server;  
  
**import** java.util.List;  
**import** java.util.Random;  
**import** java.util.concurrent.ThreadLocalRandom;  
  
**public class** RandomRule\_ZY **extends** AbstractLoadBalancerRule {  
  
 *//total = 0 //当total数==5了以后，指针才能往下走  
 //index = 0 //当前对外提供的服务器地址  
 //total需要重新至为0，但是已经达到过5次，index = 1  
 //分析我们5次，但是微服务只有8001，8002,8003三台  
 //* **private int total** = 0; *//总共被调用的次数，目的要求每台被调用5次* **private int currentIndex** = 0; *//当前提供服务的机器号* **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();  
 List<Server> allList = lb.getAllServers();  
  
 **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);  
 server = upList.get(index);\*/*

*//主要配置这一段* **if**(**total**<5){  
 server = upList.get(**currentIndex**);  
 **total**++;  
 }**else**{  
 **total** = 0;  
 **currentIndex**++;  
 **if**(**currentIndex** >= upList.size()){  
 **currentIndex** = 0;  
 }  
 }  
  
 **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) {  
  
 }  
}

修改源码后将MySelfRule改为

@Configuration  
**public class** MySelfRule {  
 @Bean  
 **public** IRule myRule(){  
 *// return new RandomRule();//Ribbon默认是轮询，我自定义为随机* **return new** RandomRule\_ZY();*//自定义轮询* }  
}

## Feign负载均衡

Feign是一个声明式客户端。使用Feign能让编写WebService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也支持可插拔式的编码器和解码器。SpringCoud对Feign进行了封装，使其支持了SpringMVC标准注解和HttpMessageConverterters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

只需要创建一个接口，然后在上面添加注解即可。

参考官网:<https://github.com/OpenFeign/feign>

之前用Ribbon进行负载均衡，功能很强大甚至可以自己自定义算法

### Feign是怎么出来的？

大部分的大家都可以接受，直接调用微服务来进行访问。

大家目前都习惯面新街口编程，比如WebServer接口，比如DAO接口，这个已经是大家的规范了。

所以现在有两种方式：

微服务名字获得调用地址 --------Ribbon

通过接口+注解，来获得我们的使用服务

通过社区其他程序员提出的，还是统一的面向接口编程的套路 --------Feign

### Feign能干什么：

Feign指在编写JavaHttp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，形成了一套模板的调用方法。但是在实际开发中，由于对于服务依赖的调用可能不止一处，往往一个接口Juin被多处调用，所有通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用，所以，Feign在此基础上做进一步封装，由他来帮助我们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（以前的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），即完成对服务提供方的接口绑定，简化了使用SpringCloudRibbon时，自动封装服务调用客户端的开发量。

### 使用Feign的步骤

创建新的子模块（服务消费者）

参考microservicecloud-consumer-dept-80

新建子模块microservicecloud-consumer-dept-feign：

修改启动类名称DeptConsumer80\_Feign\_App

删除Ribbon配置文件

修改pom.xml添加对Feign的支持

*<!-- feign相关依赖 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-feign</**artifactId**>  
</**dependency**>

修改microservicecloud-api（存放公共资源）工程

POM:

*<!-- feign相关依赖 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-feign</**artifactId**>  
</**dependency**>

新建DeptClientService接口并新增注解@FeignClient  
@FeignClient(value = **"microservicecloud.dept"**)*//标识对那一个微服务进行面向接口Feign的编码***public interface** DeptClientService {  
 @RequestMapping(value = **"/dept/get/{id}"**,method = RequestMethod.***GET***)  
 **public** Dept get(@PathVariable(**"id"**) **long** id);  
 @RequestMapping(value = **"/dept/list"**,method = RequestMethod.***GET***)  
 **public** List<Dept> list();  
 @RequestMapping(value = **"/dept/add"**,method = RequestMethod.***POST***)  
 **public boolean** add(Dept dept);  
}

Clean清空编译文件

Install 提交到Maven仓库

Microservicecloud-consumer-dept-feign工程修改Controller，添加上一步新建的DeptClientService接口

@RestController  
**public class** DeptController\_Consumer {  
  
 @Autowired  
 **private** DeptClientService **service**;  
  
 @RequestMapping(value = **"/consumer/dept/get/{id}"**)  
 **public** Dept get(@PathVariable(**"id"**) Long id){  
 **return this**.**service**.get(id);  
 }  
 @RequestMapping(value = **"/consumer/dept/list"**)  
 **public** List<Dept> list(){  
 **return this**.**service**.list();  
 }  
 @RequestMapping(value = **"/consumer/dept/add"**)  
 **public** Object add(Dept dept){  
 **return this**.**service**.add(dept);  
 }

面向接口编程替代了以前的模板

在服务消费者主启动类上添加注解

@EnableFeignClients(basePackages = {**"cn.sang"**})  
 @ComponentScan(**"cn.sang"**)

### Feign集成了Ribbon

利用Ribbon维护了MicroServiceCloud-Dept的服务列表信息，并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并且声明式的方法，优雅简单的实现了服务调用。

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

该请求发送给Eureka服务器(http://MICROSERVICECLOUD.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Ribbon技术，所以也支持负载均衡作用

## Hystrix断路器（熔断器）

分布式系统面临的问题:

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

### 服务雪崩：

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其他的微服务，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资源都在几秒钟内饱和，比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟加载，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，这些都标识需要对故障和延迟继续隔离和管理，以便单个依赖关系失败，不能取消整个应用程序或系统。

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

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

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

### 服务熔断：

熔断机制是对应雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间过长时，会进行服务降级，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的响应正常后恢复调用链路，在SpringCloud框架熔断机制通过Hystrix实现。Hystrix会监控微服务调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand

### 搭建步骤：

创建新的子模块microservicecloud-provider-dept-8001（参考8001）

添加新的依赖

*<!-- hystrix -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-hystrix</**artifactId**>  
</**dependency**>

修改YML配置文件

**instance-id:** microservicecloud-dept8001 改成

**instance-id:** microservicecloud-dept8001-hystrix

**instance:  
 instance-id:** microservicecloud-dept8001-hystrix *#在eureka内的Status的链接名* **prefer-ip-address: true** *#显示IP*

@RestController  
**public class** DeptController {  
 @Autowired  
 **private** DeptService **deptService**;  
  
 @RequestMapping(value = **"/dept/get/{id}"**,method = RequestMethod.***GET***)  
 @HystrixCommand(fallbackMethod = **"processHystrix\_Get"**)  
 **public** Dept get(@PathVariable(**"id"**) Long id){  
 Dept dept = **this**.**deptService**.get(id);  
 **if**(**null** == dept){  
 **throw new** RuntimeException(**"该ID:"**+id+**"没有对应的信息"**);  
 }  
 **return** dept;  
 }  
  
 **public** Dept processHystrix\_Get(@PathVariable(**"id"**) Long id){  
 **return new** Dept().setDeptno(id).setDname(**"该ID:"**+id+**"没有对应的想，null--@HystrixCommand"**).setDb\_source(**"no this database MySQL"**);  
 }  
  
}

Controller 代码：

修改主启动类名称：DeptProvider8001\_Hystrix\_App

在主启动类上加上注解:

@EnableCircuitBreaker*//对hystrix熔断机制的支持*

启动服务三个Eureka，和一个microservicecloud-dept8001-hystrix，和一个消费者

# 服务降级

整体资源快不够了，忍痛将某些服务先关掉，待渡过难关，再开启回来。

服务的降级处理是在客户端完成的，与服务端没有关系。

### 配置步骤：

修改microservicecloud-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

@Component  
**public class** DeptClientServiceFallbackFactory **implements** FallbackFactory<DeptClientService> {  
  
 @Override  
 **public** DeptClientService create(Throwable throwable) {  
 **return new** DeptClientService() {  
 @Override  
 **public** Dept get(**long** id) {  
 **return new** Dept().setDeptno(id).setDname(**"该ID:"**+id+**"没有对应的信息，Consumer客户端提供的降级信息，此刻服务Provider已经关闭"**).setDb\_source(**"no this database in MySQL"**);  
 }  
  
 @Override  
 **public** List<Dept> list() {  
 **return null**;  
 }  
  
 @Override  
 **public boolean** add(Dept dept) {  
 **return false**;  
 }  
 };  
 }

在类上加上@Component注解不然没用

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

@FeignClient(name = **"microservicecloud.dept"**,fallbackFactory = DeptClientServiceFallbackFactory.**class**)*//服务熔断*

再clean，install

修改yml：

**feign:  
 hystrix:  
 enabled:** true

测试

3个eureka先启动

微服务microservicecloud-provider-dept-8001启动

Microservicecloud-consumer-dept-feign启动

正常访问测试 http：//localhost/consumer/dept/get/1

故意关闭微服务microservicecloud-provider-dept-8001

客户端自己调用提示

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

## 服务监控hystrixDashboarb：

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

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

### Case步骤:

新建工程microservicecoud-consumer-hystrix-dashboard

Pom：

<**dependency**>  
 <**groupId**>cn.sang</**groupId**>  
 <**artifactId**>microservicecloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**dependency**>  
*<!-- 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：

**server:  
 port:** 9001

主启动类改名+新注解@EnableHystrixDashboard

@EnableHystrixDashboard*//开启仪表盘监控注解*

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

监控依赖配置:

*<!-- 这个是init页面的必须要的依赖 监控 actuator监控信息完善 -->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>

启动microservicecloud-consumer-hystrix-dashboard该服务监控消费端

输入网址<http://localhost:9001/hystrix>弹出页面则配置成功

启动三个eureka集群

启动microservicecloud-provider-dept-hystrix-8001:

启动相关的微服务工程

输入测试网址<http://localhost:8001/dept/get/1>

<http://localhost:8001/hystrix.stream>（json文本形式）

监控测试：

多次刷新<http://localhost:8001/dept/get/1>

观察监控窗口:



填写监控地址

监控结果

如何看：

7色

1圈:

实心圆:共有两种含义。它通过颜色的变化代表了实例的监控程度

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送变化，流量越大该实心圆就越大。所以通过该实心圆的展示，就可以在大量的实例中快速的发现故障实例和好压力。

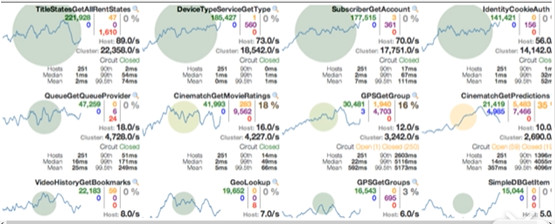
1线

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

整图说明:



搞懂一个才能看懂复杂的：（公司开发）



## zuul路由网关

### 概述：

是什么：

Zuul是Netflix开源的微服务网关，他可以和Eureka,Ribbon,Hystrix等组件配合使用。Zuul组件的核心是一系列的过滤器，这些过滤器可以完成以下功能：

身份认证和安全:识别每一个资源的验证要求 审查和监控

动态路由：动态将请求路由到不同后端集群

压力测试：组建增加指定集群流量，检测性能

负载分配：为不同负载类型分配对应容量

静态响应处理：边缘位置响应，避免转发到内部集群

多区域弹性

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

注意Zuul服务最终还是会注册进Eureka

提供=代理+路由+过滤三大功能

能干嘛：

过滤：每个客户端用户请求微服务应用提供的接口时，它们权限往往都有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而服务路由并没有限制权限这样的功能，所有请求被毫无保留的转发到具体应用并返回结果，为了实现对客户端请求的安全校验签名和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的的过滤器或拦截器。不过这样会增加日后的维护难度，而且代码也变得冗余。所以最好的做法是将这些检验剥离出去，构建一个独立的鉴权服务。

路由：不依赖服务发现机制，通过在配置文件中具体指定每个路由表达式与服务实例的映射关系来实现API网关对外部请求的路由。这里我们需要根 根据服务实例的数量采用不同方式的配置来实现路由规则 。

源码地址：<https://github.com/Netflix/zuul/wiki/Getting-Started>

### 路由基本配置：

路由访问映射规则：

新建Modulemicroservicecloud-zuul-gateway-9527

POM：

*<!-- zuul路由网关 -->*<**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**>  
*<!-- actuator监控 -->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>  
*<!-- hystrix容错 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-hystrix</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
</**dependency**>  
*<!-- 日常标配 -->*<**dependency**>  
 <**groupId**>cn.sang</**groupId**>  
 <**artifactId**>microservicecloud-api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
</**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.boot</groupId>  
 <artifactId>springloaded</artifactId>  
</dependency>-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
</**dependency**>

YML:

**server:  
 port:** 9527  
  
**spring:  
 application:  
 name:** microservicecloud-zuul-gateway  
  
**eureka:  
 client:  
 service-url:  
 defaultZone:** http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eureka7003.com:7003/eureka  
 **instance:  
 instance-id:** sang-9527.com  
 **prefer-ip-address: true  
  
info:  
 app.name:** sang-microcloud  
 **company.name:** www.sang.com  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.version$

hosts修改：127.0.0.1 myzuul.com

主启动类:

@EnableZuulProxy开启Zuul代理

启动：

三个Eueeka集群

一个服务提供类microservicecloud-provide-8001

一个路由

测试

<http://eureka7001.com:7001> （查看注册中心）

<http://localhost:8001/dept/get/1>（不用路由）

<http://myzuul.com:9527/microservicecloud.dept/dept/get/2>（启用路由）

### 路由访问映射规则:

工程microservicecloud-zuul-gateway-9527

代理名称

Yml：

**zuul:  
 routes:  
 mydept.serviceId:** microservicecloud.dept *#真实服务名* **mydept.path:** /mydept/\*\* *#别名  
#原来访问是http://myzuul.com:9527/microservicecloud.dept/dept/get/2  
#现在可以用http://myzuul.com:9527/mydept/dept/get/2*

问题：

路由访问OK：

*<http://myzuul.com:9527/mydept/dept/get/2>*

原路径访问OK：

*<http://myzuul.com:9527/microservicecloud.dept/dept/get/2>*

原服务真实名忽略:

**zuul:  
 ignored-services:** microservicecloud.dept

*#忽略真实服务名"\*"忽略所有微服务真实名称*  
 **routes:  
 mydept.serviceId:** microservicecloud.dept *#真实服务名* **mydept.path:** /mydept/\*\* *#别名*

设置统一公共前缀

**zuul:  
 prefix:** /sang *#前缀* **ignored-services:** microservicecloud.dept *#忽略真实服务名"\*"忽略所有微服务真实名称* **routes:  
 mydept.serviceId:** microservicecloud.dept *#真实服务名* **mydept.path:** /mydept/\*\* *#别名*

这个时候：

路由访问NO：

*<http://myzuul.com:9527/mydept/dept/get/2>*

原路径访问NO：

*<http://myzuul.com:9527/microservicecloud.dept/dept/get/2>*

带路由网关+前缀+别名访问OK:

*[http://myzuul.com:9527/sang/mydept/dept/get/2](http://myzuul.com:9527/mydept/dept/get/2)*

最后YML:

**server:  
 port:** 9527  
  
**spring:  
 application:  
 name:** microservicecloud-zuul-gateway  
  
**zuul:  
 prefix:** /sang *#前缀* **ignored-services:** microservicecloud.dept *#忽略真实服务名"\*"忽略所有微服务真实名称* **routes:  
 mydept.serviceId:** microservicecloud.dept *#真实服务名* **mydept.path:** /mydept/\*\* *#别名  
#原来访问是http://myzuul.com:9527/microservicecloud.dept/dept/get/2  
#现在可以用http://myzuul.com:9527/mydept/dept/get/2***eureka:  
 client:  
 service-url:  
 defaultZone:** http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eureka7003.com:7003/eureka  
 **instance:  
 instance-id:** sang-9527.com  
 **prefer-ip-address: true  
  
info:  
 app.name:** sang-microcloud  
 **company.name:** www.sang.com  
 **build.artifactId:** $project.artifactId$  
 **build.version:** $project.version$

## SpringCloudConfig分布式配置中心：

### 概述：

分布式系统面临的---配置问题：

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，每个服务的粒度很小，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。SpringCloud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，上百个配置文件的管理....../(TOT)/~~

是什么：

SpringCloudConfig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是所有环境提供了一个中心化的外部配置。

SpringCloudConfig分为服务端和客户端两部分。

服务端有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/加密信息等访问接口

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

能干吗：

与GitHub整合配置：

由于SpringCloudConfig默认使用Git来存储配置文件(也有其他方式，比如支持SVN和本单文件),但是最推荐的还是Git，而且使用的是http://https访问的形式。

### SpringCloudConfig服务端配置：

用自己的GitHub账号在GitHub上新建一个名为microservicecloud-config的新Repository

由上一步获得SHH协议的git地址：[git@github.com:wosiniDad/microservicecloud-config.git](mailto:git@github.com:wosiniDad/microservicecloud-config.git)

#### 本地硬盘目录上新建git仓库并clone：

本地地址（自己设置）我的D:\SpringCloudGit

Git命令：git clone [git@github.com:wosiniDad/microservicecloud-config.git](mailto:git@github.com:wosiniDad/microservicecloud-config.git)

本地D:\SpringCloudGit\microservicecloud-config里面新建一个application.yml

#### Yml内容：

spring:

profiles:

active:

- dev

---

spring:

profiles: dev #开发环境

application:

name: microservicecloud-config-sang-dev

---

spring:

profiles: test #测试环境

application:

name: microservicecloud-config-sang-test

#请保存为UTF-8格式

保存的格式必须为UTF-8

#### 将上一步的YML文件推送到gethub上：

git add .

git commit -m “init yml”

Git push origin master

#### 新建Module模块microservicecloud-config-3344

POM:

*<!-- springCloudConfig -->*<**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**>  
*<!-- 图像化监控 -->*<**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**>  
*<!--<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>springloaded</artifactId>  
</dependency>-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
</**dependency**>

YML:

**server:  
 port:** 3344  
  
**spring:  
 application:  
 name:** microservicecloud-config  
 **cloud:  
 config:  
 server:  
 git:  
 uri:** [git@github.com:wosiniDad/microservicecloud-config.git](mailto:git@github.com:wosiniDad/microservicecloud-config.git)

*#GitHub上面的git仓库*

修改主启动类Config\_3344\_StartSpringCloudApp

添加注解：@EnableConfigServer *//开启配置中心*

windows下修改hosts文件，增加映射：

127.0.0.1 config-3344.com

测试通过Config微服务是否可以从GitHub上获取配置内容：

启动微服务3344

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

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

<http://config-3344.com:3344/application-xxx.yml> (不存在的配置)

配置读取规则:

/{application}-{profie}.yml

/{application}/{proflie}[/{label}]

/{lable}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

它即为Cloud的配置中心模块

### SpringCloudConfig客户端配置与测试：

在本地D:\SpringCloudGit\microservicecloud-config路径下新建文件

microservicecloud-config-client.yml

microservicecloud-config-client.yml内容：

spring:

profiles:

active:

- dev

---

server:

port: 8201

spring:

profiles: dev

application:

name: microservicecloud-config-client

eureka:

client:

server-url:

defauthZone: http://eureka-dev.com:7001/eureka/

---

server:

port: 8202

spring:

profiles: test

application:

name:microservicecloud-config-client

eureka:

client:

server-url:

defauthZone: <http://eureka-test.com:7001/eureka/>

将上一步提交到GitHub中

新建microservicecloud-config-client-3355

POM:

<**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.boot</**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-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**>

bootstrap.yml配置：

application.yml是用户级的资源配置项

bootstrap.yml是系统级的，优先级更高

SpringCloud会创建一个BootstrapContext，作为Spring应用的ApplicationContext的父上下文。初始化的时候。BootstrapContext负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取Environment。Bootstrap属性有优先级，默认情况下，它们不会被本地覆盖。Bootstrapcontext和ApplicationContext有着不同的约定

所以新增了一个bootstrap.yml文件，保证BootstrapContext和ApplicationContext配置的分离。

Bootstrap.yml内容:

**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:** dev *#本次访问的配置源* **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*

Application.yml内容:

**spring:  
 application:  
 name:** microservicecloud-config-client

新建rest类，验证是否能从CitHub上读取配置：

@RestController  
**public class** ConfigClientRest {  
 @Value(**"${spring.application.name}"**)  
 **private** String **applicationName**;  
  
 @Value(**"${eureka.client.server-url.defauthZone}"**)  
 **private** String **eurekaServers**;  
  
 @Value(**"${server.port}"**)  
 **private** String **port**;  
  
 @RequestMapping(**"/config"**)  
 **public** String getConfig(){  
 String str = **"applicationName: "**+**applicationName**+**"\t eurekaServers:"**+**eurekaServers**+**"\t port:"**+**port**;  
 System.***out***.println(**"\*\*\*\*\*\*str:"**+str);  
 **return "applicationName: "**+**applicationName**+**"\t eurekaServers: "**+**eurekaServers**+**"\t prot: "**+**port**;  
 }  
}

主启动类ConfigClient\_3355\_StartSpringCloudApp：

@SpringBootApplication  
**public class** ConfigClient\_3355\_StartSpringCloudApp {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ConfigClient\_3355\_StartSpringCloudApp.**class**,args);  
 }  
}

测试：

启动Config配置中心3344微服务并自测:http://config.3344.com:3344/application-config

启动3355作为Client准备访问

Bootstrap.Yml里面的profile值是什么，决定从github上读取什么：

假如目前是profile:dev:

Dev默认在github上对应的端口就是8201:http://client-config.com:8201/config

假如目前是profile:test:

Test默认在github上对应的端口就是8202:http://client-config.com:8202/config

成功实现了客户端3355访问SpringCloudConfig3344通过GitHub获取配置信息

## SpringCloudConfig配置实战：

### 目前情况：

1. Config服务端配置OK并且测试通过，我们可以和config+GitHub进行配置修改并获得内容
2. 此时我们做一个eureka服务+一个Dept访问的微服务，将两个微服务的配置统一由github获得实现统一配置分布式管理，完成多环境变更。

### Git文件本地配置：

在本地D:\SpringCloudGit\microservicecloud-config路径下新建文件

Microservicecloud-config-eureka-client.yml:

spring:

profiles:

active:

- dev

---

server:

port: 7001 #注册中心占用7001端口，冒号后面必须有空格

spring:

profiles: dev

application:

name: microservicecloud-config-eureka-client

eureka:

instance:

hostname: eureka7001.com #冒号后面必须有空格

client:

register-with-eureka: false #当前eureka-server自己不注册进服务列表中

fetch-registry: false #不通过eureka获取注册信息

server-url:

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

---

server:

port: 7001 #注册中心占用7001端口，冒号后面必须有空格

spring:

profiles: test

application:

name: microservicecloud-config-eureka-client

eureka:

instance:

hostname: eureka7001.com #冒号后面必须有空格

client:

register-with-eureka: false #当前eureka-server自己不注册进服务列表中

fetch-registry: false #不通过eureka获取注册信息

server-url:

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

在本地D:\SpringCloudGit\microservicecloud-config路径下新建文件

Microservicecloud-config-dept-client.yml:

spring:

profiles:

active:

- dev

---

server:

port: 8001

mybatis:

config-location: classpath:mybatis/mybatis.cfg.xml #mybatis配置文件所在路径

type-aliases-package: cn.sang.pojo #所有Entity别名所在包

mapper-locations:

classpath: mybatis/mapper/\*\*/\*.xml #mapper映射文件

spring:

profiles: dev

application:

name: microservicecloud-config-dept-client #对外暴露的微服务名字

datasource:

type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型

driver-class-name: org.gjt.mm.mysql.Driver #mysql驱动包

url: jdbc:mysql://localhost:3306/cloudDB01 #数据库名称

username: root

password: 123456

dbcp2:

min-idle: 5 #数据库连接池的最小连接数

initial-size: 5 #初始化连接数

max-total: 5 #最大连接数

max-wait-millis: 200 #等待连接获取的最大超时时间

eureka:

client: #客户端注册进eureka服务列表内

service-url:

#当Eureka是单击版的时候 defaultZone: http://localhost:7001/eureka

#集群

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

instance:

instance-id: microservicecloud-dept8001 #在eureka内的Status的链接名

prefer-ip-address: true #显示IP

info: #这里是进入eureka里面单击服务连接进入的页面配置

app.name: fw-microservicecloud

company.name: www.fw.com

build.artifactId: $project.artifactId$

build.version: $project.version$

---

server:

port: 8001

mybatis:

config-location: classpath:mybatis/mybatis.cfg.xml #mybatis配置文件所在路径

type-aliases-package: cn.sang.pojo #所有Entity别名所在包

mapper-locations:

classpath: mybatis/mapper/\*\*/\*.xml #mapper映射文件

spring:

profiles: test

application:

name: microservicecloud-config-dept-client #对外暴露的微服务名字

datasource:

type: com.alibaba.druid.pool.DruidDataSource #当前数据源操作类型

driver-class-name: org.gjt.mm.mysql.Driver #mysql驱动包

url: jdbc:mysql://localhost:3306/cloudDB02 #数据库名称

username: root

password: 123456

dbcp2:

min-idle: 5 #数据库连接池的最小连接数

initial-size: 5 #初始化连接数

max-total: 5 #最大连接数

max-wait-millis: 200 #等待连接获取的最大超时时间

eureka:

client: #客户端注册进eureka服务列表内

service-url:

#当Eureka是单击版的时候 defaultZone: http://localhost:7001/eureka

#集群

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

instance:

instance-id: microservicecloud-dept8001 #在eureka内的Status的链接名

prefer-ip-address: true #显示IP

info: #这里是进入eureka里面单击服务连接进入的页面配置

app.name: fw-microservicecloud

company.name: www.fw.com

build.artifactId: $project.artifactId$

build.version: $project.version$

### Config版的eureka服务端：

新建工程micrsoervicecloud-config-eureka-client-7001

POM：

*<!-- SpringCloudConfig配置 -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-eureka-server</**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**>

Bootstrap.yml:

**spring:  
 cloud:  
 config:  
 name:** microservicecloud-config-eureka-client *#需要从github上读取的资源名称，注意没有yml后缀名* **profile:** dev  
 **label:** master  
 **uri:** http://config-3344.com:3344 *#SpringCloudConfig获取的服务地址*

Application.yml:

**spring:  
 application:  
 name:** microservicecloud-config-eureka-client

主启动类Config\_Git\_EurekaServerApplication

测试：

先启动microservicecloud-config-3344微服务，保证Config总配置是OK的

在启动microservicecloud-config-eurekaclient-7001微服务

<http://eureka7001.com:7001/>

出现eureka主页表示成功启动

Config版的dept微服务：

参考之前的8001拷贝后新建工程micorservicecloud-config-dept-client-8001

POM:

*<!-- SpringCloudConfig配置 -->*<**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-eureka</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>druid</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>ch.qos.logback</**groupId**>  
 <**artifactId**>logback-core</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>1.3.1</**version**>  
</**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：修改后立即生效 -->*<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>springloaded</**artifactId**>  
 <**version**>1.2.7.RELEASE</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
</**dependency**>

Bootstrap.yml:

**spring:  
 cloud:  
 config:  
 name:** microservicecloud-config-dept-client *#需要从github上读取资源名称*

*#profile配置是什么就取什么配置dev or test  
 #profile：dev* **profile:** test  
 **label:** master  
 **uri:** http://config-3344.com:3344 *#SpringCloudConfig获取的服务地址*

Application.yml：

**spring:  
 application:  
 name:** microservicecloud-config-dept-client

主启动类及其它一套业务逻辑代码：

参考：micorservicecloud-config-dept-client-8001

配置说明：

主要看bootstrap.yml

profile:属性具体值是什么，从而确定它能从github上取得什么样的配置

加入配置dev找左图，test找右图，具体各自数据库不同，从而依据配置得到分布式配置的目的

测试：

Test配置默认访问：

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

可以看到数据库配置是02

Test配置默认访问：

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

可以看到数据库配置是01

技术总结：

