# Spring Cloud

## 微服务概述

微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合,每一个微服务提供单个业务功能的服务，一个服务做一件事，从技术角度看就是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销毁，拥有自己独立的数据库。

微服务强调的是服务的大小，它关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用，狭义的看，可以看作Eclipse里面的一个个微服务工程或者Module。

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的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，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

**微服务优点**：强模块化边界；可独立部署；技术多样性。

**微服务缺点**：分布式系统复杂性；最终一致性；运维复杂性。

### 微服务技术栈

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

### 微服务基础设施

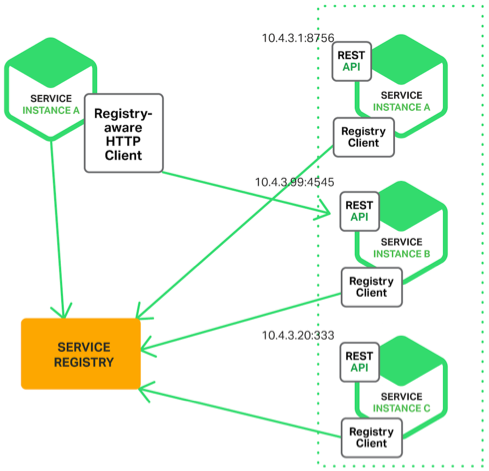
每项微服务基础设施都是一个平台、一个系统、一个解决方案，其过程和做业务系统类似，都需要经过需求分析、架构设计、开发、测试、部署上线等步骤。

#### 服务发现

微服务种类和数量很多，如果这些信息全部通过手工配置的方式写入各个微服务节点，首先配置工作量很大，配置文件可能要配几百上千行，几十个节点加起来后配置项就是几万几十万行了，人工维护这么大数量的配置项是一项灾难；其次是微服务节点经常变化，可能是由于扩容导致节点增加，也可能是故障处理时隔离掉一部分节点，还可能是采用灰度升级，先将一部分节点升级到新版本，然后让新老版本同时运行。不管哪种情况，都希望节点的变化能够及时同步到所有其他依赖的微服务。如果采用手工配置，是不可能做到实时更改生效的。因此，需要一套服务发现的系统来支撑微服务的自动注册和发现。

服务发现主要有两种实现方式：自理式和代理式。

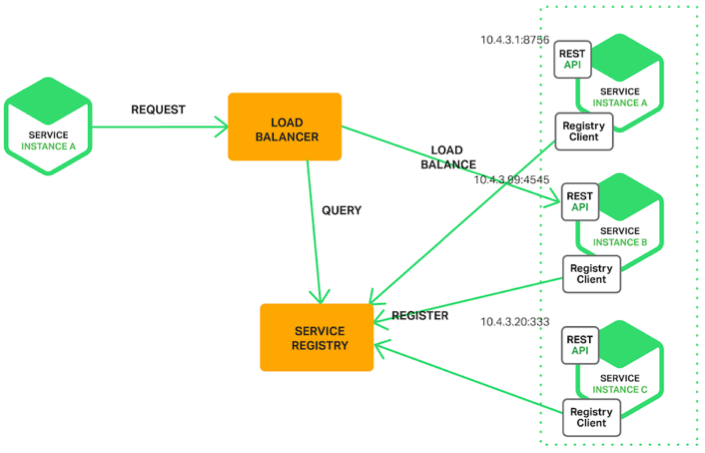
##### 自理式



自理式结构就是指每个微服务自己完成服务发现。例如，图中 SERVICE INSTANCE A 访问 SERVICE REGISTRY 获取服务注册信息，然后直接访问 SERVICE INSTANCE B。

自理式服务发现实现比较简单，因为这部分的功能一般通过统一的程序库或者程序包提供给各个微服务调用，而不会每个微服务都自己来重复实现一遍；并且由于每个微服务都承担了服务发现的功能，访问压力分散到了各个微服务节点，性能和可用性上不存在明显的压力和风险。

##### 代理式



代理式结构就是指微服务之间有一个负载均衡系统（图中的 LOAD BALANCER 节点），由负载均衡系统来完成微服务之间的服务发现。

代理式的方式看起来更加清晰，微服务本身的实现也简单了很多，但实际上这个方案风险较大。第一个风险是可用性风险，一旦 LOAD BALANCER 系统故障，就会影响所有微服务之间的调用；第二个风险是性能风险，所有的微服务之间的调用流量都要经过 LOAD BALANCER 系统，性能压力会随着微服务数量和流量增加而不断增加，最后成为性能瓶颈。因此 LOAD BALANCER 系统需要设计成集群的模式，但 LOAD BALANCER 集群的实现本身又增加了复杂性。

不管是自理式还是代理式，服务发现的核心功能就是服务注册表，注册表记录了所有的服务节点的配置和状态，每个微服务启动后都需要将自己的信息注册到服务注册表，然后由微服务或者 LOAD BALANCER 系统到服务注册表查询可用服务。

#### 服务路由

有了服务发现后，微服务之间能够方便地获取相关配置信息，但具体进行某次调用请求时，还需要从所有符合条件的可用微服务节点中挑选出一个具体的节点发起请求，这就是服务路由需要完成的功能。

服务路由和服务发现紧密相关，服务路由一般不会设计成一个独立运行的系统，通常情况下是和服务发现放在一起实现的。对于自理式服务发现，服务路由是微服务内部实现的；对于代理式服务发现，服务路由是由 LOAD BALANCER 系统实现的。无论放在哪里实现，服务路由核心的功能就是路由算法。常见的路由算法有：随机路由、轮询路由、最小压力路由、最小连接数路由等。

#### 服务容错

系统拆分为微服务后，单个微服务故障的概率变小，故障影响范围也减少，但是微服务的节点数量大大增加。从整体上来看，系统中某个微服务出故障的概率会大大增加，如果不及时处理故障，故障扩散开来就会导致看起来系统中很多服务节点都故障了，因此需要微服务能够自动应对这种出错场景，及时进行处理。否则，如果节点一故障就需要人工处理，投入人力大，处理速度慢；而一旦处理速度慢，则故障就很快扩散，所以需要服务容错的能力。

常见的服务容错包括请求重试、流控和服务隔离。通常情况下，服务容错会集成在服务发现和服务路由系统中。

#### 服务监控

系统拆分为微服务后，节点数量大大增加，导致需要监控的机器、网络、进程、接口调用数等监控对象的数量大大增加；同时，一旦发生故障，需要快速根据各类信息来定位故障。这两个目标如果靠人力去完成是不现实的。举个简单例子：收到用户投诉说业务有问题，如果此时采取人工的方式去搜集、分析信息，可能把几十个节点的日志打开一遍就需要十几分钟了，因此需要服务监控系统来完成微服务节点的监控。

**服务监控的主要作用有：**

1、实时搜集信息并进行分析，避免故障后再来分析，减少了处理时间。

2、服务监控可以在实时分析的基础上进行预警，在问题萌芽的阶段发觉并预警，降低了问题影响的范围和时间。

通常情况下，服务监控需要搜集并分析大量的数据，因此建议做成独立的系统，而不要集成到服务发现、API 网关等系统中。

#### 服务跟踪

服务监控可以做到微服务节点级的监控和信息收集，但如果需要跟踪某一个请求在微服务中的完整路径，服务监控是难以实现的。因为如果每个服务的完整请求链信息都实时发送给服务监控系统，数据量会大到无法处理。

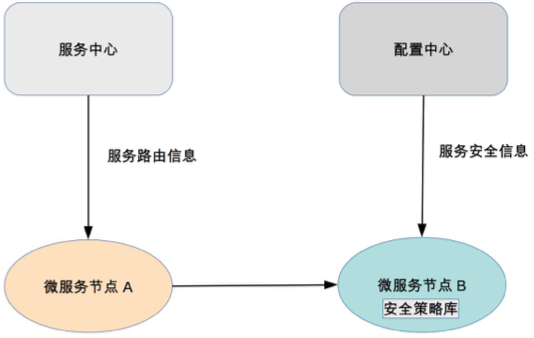
服务监控和服务跟踪的区别可以简单概括为宏观和微观的区别。例如，A 服务通过 HTTP 协议请求 B 服务 10 次，B 通过 HTTP 返回 JSON 对象，服务监控会记录请求次数、响应时间平均值、响应时间最高值、错误码分布这些信息；而服务跟踪会记录其中某次请求的发起时间、响应时间、响应错误码、请求参数、返回的 JSON 对象等信息。

目前无论是分布式跟踪还是微服务的服务跟踪，绝大部分请求跟踪的实现技术都基于 Google 的 Dapper 论文《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》

#### 服务安全

系统拆分为微服务后，数据分散在各个微服务节点上。从系统连接的角度来说，任意微服务都可以访问所有其他微服务节点；但从业务的角度来说，部分敏感数据或者操作，只能部分微服务可以访问，而不是所有的微服务都可以访问，因此需要设计服务安全机制来保证业务和数据的安全性。

服务安全主要分为三部分：接入安全、数据安全、传输安全。通常情况下，服务安全可以集成到配置中心系统中进行实现，即配置中心配置微服务的接入安全策略和数据安全策略，微服务节点从配置中心获取这些配置信息，然后在处理具体的微服务调用请求时根据安全策略进行处理。由于这些策略是通用的，一般会把策略封装成通用的库提供给各个微服务调用。基本架构如下：



#### 自动化测试

微服务将原本大一统的系统拆分为多个独立运行的“微”服务，微服务之间的接口数量大大增加，并且微服务提倡快速交付，版本周期短，版本更新频繁。如果每次更新都靠人工回归整个系统，则工作量大，效率低下，达不到“快速交付”的目的，因此必须通过自动化测试系统来完成绝大部分测试回归的工作。

自动化测试涵盖的范围包括代码级的单元测试、单个系统级的集成测试、系统间的接口测试，理想情况是每类测试都自动化。如果因为团队规模和人力的原因无法全面覆盖，至少要做到接口测试自动化。

#### 自动化部署

相比大一统的系统，微服务需要部署的节点增加了几倍甚至十几倍，微服务部署的频率也会大幅提升（业务系统 70% 的工作日都有部署操作），综合计算下来，微服务部署的次数是大一统系统部署次数的几十倍。这么大量的部署操作，如果继续采用人工手工处理，需要投入大量的人力，且容易出错，因此需要自动化部署的系统来完成部署操作。

自动化部署系统包括版本管理、资源管理（例如，机器管理、虚拟机管理）、部署操作、回退操作等功能。

#### 配置中心

微服务的节点数量非常多，通过人工登录每台机器手工修改，效率低，容易出错。特别是在部署或者排障时，需要快速增删改查配置，人工操作的方式显然是不行的。除此以外，有的运行期配置需要动态修改并且所有节点即时生效，人工操作是无法做到的。综合上面的分析，微服务需要一个统一的配置中心来管理所有微服务节点的配置。

配置中心包括配置版本管理（同样的微服务，有 10 个节点是给移动用户服务的，有 20 个节点给联通用户服务的，配置项都一样，配置值不一样）、增删改查配置、节点管理、配置同步、配置推送等功能。

#### 接口框架

微服务提倡轻量级的通信方式，一般采用 HTTP/REST 或 RPC 方式统一接口协议。但在实践过程中，光统一接口协议还不够，还需要统一接口传递的数据格式。例如指定接口协议HTTP/REST，还需要指定 HTTP/REST 的数据格式采用JSON，并且JSON的数据都遵循规范。

如果只是简单指定了HTTP/REST协议，而不指定JSON和JSON的数据规范，那么就会出现混乱：有的微服务采用XML，有的采用JSON，有的采用键值对；即使同样都是JSON，JSON数据格式也不一样。这样每个微服务都要适配几套甚至几十套接口协议，相当于把曾经由 ESB 做的事情转交给微服务自己做了，这样做的效率显然是无法接受的，因此需要统一接口框架。

接口框架不是一个可运行的系统，一般以库或者包的形式提供给所有微服务调用。例如，针对上面的 JSON 样例，可以由某个基础技术团队提供多种不同语言的解析包（Java 包、Python 包、C 库等）。

#### API网关

系统拆分为微服务后，内部的微服务之间是互联互通的，相互之间的访问都是点对点的。如果外部系统想调用系统的某个功能，也采取点对点的方式，则外部系统会非常“头大”。因为在外部系统看来，它不需要也没办法理解这么多微服务的职责分工和边界，它只会关注它需要的能力，而不会关注这个能力应该由哪个微服务提供。

除此以外，外部系统访问系统还涉及安全和权限相关的限制，如果外部系统直接访问某个微服务，则意味着每个微服务都要自己实现安全和权限的功能，这样做不但工作量大，而且都是重复工作。

综合上面的分析，微服务需要一个统一的 API 网关，负责外部系统的访问操作。

API网关是外部系统访问的接口，所有的外部系统接入系统都需要通过API网关，主要包括接入鉴权（是否允许接入）、权限控制（可以访问哪些功能）、传输加密、请求路由和流量控制等功能。

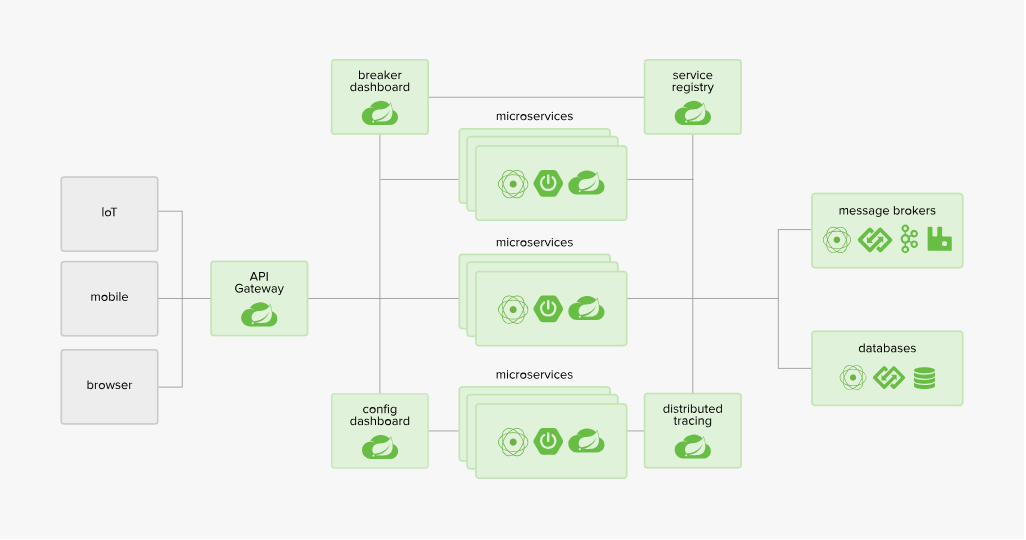
## Spring Cloud概述

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

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

SpringBoot并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考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，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包

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



## Spring Cloud和Spring Boot

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系。SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架； Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现，必须基于Spring Boot开发。

可以单独使用Spring Boot开发项目，但是Spring Cloud离不开 Spring Boot。

| Cloud代号 | Boot版本(train) | Boot版本(tested) | lifecycle |
| --- | --- | --- | --- |
| Angle | 1.2.x | incompatible with 1.3 | EOL in July 2017 |
| Brixton | 1.3.x | 1.4.x | 2017-07卒 |
| Camden | 1.4.x | 1.5.x | - |
| Dalston | 1.5.x | not expected 2.x | - |
| Edgware | 1.5.x | not expected 2.x | - |
| Finchley | 2.x | not expected 1.5.x | - |

**Spring Cloud 小版本分为:**

SNAPSHOT： 快照版本，随时可能修改

M： MileStone，M1表示第1个里程碑版本，一般同时标注PRE，表示预览版版。

SR： Service Release，SR1表示第1个正式版本，一般同时标注GA：(GenerallyAvailable),表示稳定版本。

## Spring Cloud和Dubbo

### 背景

Dubbo，是阿里巴巴服务化治理的核心框架，并被广泛应用于阿里巴巴集团的各成员站点。

Spring Cloud是Spring Source的产物，Spring社区的强大背书可以说是Java企业界最有影响力的组织了，除了Spring Source之外，还有Pivotal和Netfix是其强大的后盾与技术输出。其中Netflix开源的整套微服务架构套件是Spring Cloud的核心。

### 架构完整度

或许很多人会说Spring Cloud和Dubbo的对比有点不公平，Dubbo只是实现了服务治理，而Spring Cloud下面有17个子项目（可能还会新增）分别覆盖了微服务架构下的方方面面，服务治理只是其中的一个方面，一定程度来说，Dubbo只是Spring Cloud Netflix中的一个子集。但是在选择框架上，方案完整度恰恰是一个需要重点关注的内容。

根据微服务架构在各方面的要素，看看Spring Cloud和Dubbo都提供了哪些支持。

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

以上列举了一些核心部件，大致可以理解为什么之前说Dubbo只是类似Netflix的一个子集了吧。当然这里需要申明一点，Dubbo对于上表中总结为“无”的组件不代表不能实现，而只是Dubbo框架自身不提供，需要另外整合以实现对应的功能，比如：

分布式配置：可以使用淘宝的diamond、百度的disconf来实现分布式配置管理。但是Spring Cloud中的Config组件除了提供配置管理之外，由于其存储可以使用git，因此它天然的实现了配置内容的版本管理，可以完美的与应用版本管理整合起来。

服务跟踪：可以使用京东开源的Hydra

批量任务：可以使用当当开源的Elastic-Job

……

虽然，Dubbo自身只是实现了服务治理的基础，其他为保证集群安全、可维护、可测试等特性方面都没有很好的实现，但是几乎大部分关键组件都能找到第三方开源来实现，这些组件主要来自于国内各家大型互联网企业的开源产品。

### RPC和REST

最大区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。

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

**使用Dubbo的RPC来实现服务间调用的一些痛点：**

**服务提供方与调用方接口依赖方式太强**：每个微服务定义了各自的service抽象接口，并通过持续集成发布到私有仓库中，调用方应用对微服务提供的抽象接口存在强依赖关系，因此不论开发、测试、集成环境都需要严格的管理版本依赖，才不会出现服务方与调用方的不一致导致应用无法编译成功等一系列问题，这也会直接影响本地开发的环境要求，往往一个依赖很多服务的上层应用，每天都要更新很多代码并install之后才能进行后续的开发。若没有严格的版本管理制度或开发一些自动化工具，这样的依赖关系会成为开发团队的一大噩梦。

而REST接口相比RPC更为轻量化，服务提供方和调用方的依赖只是依靠一纸契约，不存在代码级别的强依赖，当然REST接口也有痛点，因为接口定义过轻，很容易导致定义文档与实际实现不一致导致服务集成时的问题，但是该问题很好解决，只需要通过每个服务整合swagger，让每个服务的代码与文档一体化，就能解决。所以在分布式环境下，REST方式的服务依赖要比RPC方式的依赖更为灵活。

**服务对平台敏感，难以简单复用**：在提供对外服务时，都会以REST方式提供，这样可以实现跨平台的特点，任何一个语言的调用方都可以根据接口定义来实现。在Dubbo中提供REST接口时，不得不实现一层代理，用来将RPC接口转换成REST接口进行对外发布。若每个服务本身就以REST接口方式存在，当对外提供服务时，主要在API网关中配置映射关系和权限控制就可实现服务的复用了。

### 文档质量

Dubbo文档在国内开源框架中算是一流的，非常全，并且讲解的也非常深入，由于版本已经稳定不再更新，所以也不太会出现不一致的情况，另外提供了中文与英文两种版本，对于国内开发者来说，阅读起来更加容易上手。

Spring Cloud由于整合了大量组件，文档在体量上自然要比dubbo多很多，文档内容上还算简洁清楚，但是更多的是偏向整合，更深入的使用方法还是需要查看其整合组件的详细文档。另外由于Spring Cloud基于Spring Boot，很多例子相较于传统Spring应用要简单很多（因为自动化配置，很多内容都成了约定的默认配置），这对于刚接触的开发者可能会有些不适应。

虽然Spring Cloud的文档量大，但是如果使用Dubbo去整合其他第三方组件，实际也是要去阅读大量第三方组件文档的，所以在文档量上区别不大。对于文档质量，由于Spring Cloud的迭代很快，难免会出现不一致的情况，所以在质量上Dubbo更好一些。而对于文档语言上，Dubbo自然对国内开发团队来说更有优势。

### 总结

Dubbo构建的微服务架构就像组装电脑，各环节选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了；

Spring Cloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

从目前Spring Cloud的被关注度和活跃度上来看，很有可能将来会成为微服务架构的标准框架。

# Eureka

SpringCloud环境下的一个注册中心，功能类似于Zookeeper。Eureka是一个SpringBoot的工程，不需要额外安装应用程序。

Eureka是Netflix开发的服务发现框架，SpringCloud将它集成在自己的子项目 spring-cloud-netflix中，实现SpringCloud的服务发现功能。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Eureka Server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。

Eureka Client是一个java客户端，用于简化与Eureka Server的交互，客户端同时也包含一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会 向Eureka Server发送心跳,默认周期为30秒，如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中把这个服务节点移除(默认90 秒)。

Eureka Server之间通过复制的方式完成数据的同步，Eureka还提供了客户端缓存机制，即使所有的Eureka Server都挂掉，客户端依然可以利用缓存中的信息消费其他服务的API。

综上，Eureka通过心跳检查、集群、客户端缓存等机制，确保了系统的高可用性、灵活性和可伸缩性。

## 自我保护机制



Renews threshold：server期望在每分钟中收到的心跳次数   
Renews (last min)：上一分钟内收到的心跳次数。

这是因为Eureka进入了自我保护机制，默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳时，EurekaServer将会注销该实例（默认90s）。但是当网络发生故障时，微服务与EurekaServer之间无法通信，这样就会很危险了，因为微服务本身是很健康的，此时就不应该注销这个微服务，而Eureka通过自我保护机制来预防这种情况，当网络健康后，该EurekaServer节点就会自动退出自我保护模式；

Eureka Server在运行期间，会统计心跳失败的比例在15分钟之内是否低于85%，如果低于，就会将当前实例注册信息保护起来，让实例不会过期，尽可能保护这些注册信息。

这时再次将客户端微服务启动，刷新服务注册中心会发现，自我保护状态已取消。

综上所述，我们可以看出来Eureka的两个组件EurekaServer和EurekaClient的作用：

1、EurekaServer 提供服务发现的能力，各个微服务启动时，会向EurekaServer注册自己的信息（例如：ip、端口、微服务名称等），EurekaServer会存储这些信息；

2、EurekaClient是一个Java客户端，用于简化与EurekaServer的交互；

3、微服务启东后，会定期性（默认30s）的向EurekaServer发送心跳以续约自己的“租期”；

4、如果EurekaServer在一定时间内未接收某个微服务实例的心跳，EurekaServer将会注销该实例（默认90s）；

5、默认情况下，EurekaServer同时也是EurekaClient。多个EurekaServer实例，互相之间通过复制的方式，来实现服务注册表中数据的同步；

6、EurekaClient也会缓存服务注册表中的信息；

综上，Eureka通过心跳检查、客户端缓存等机制，提高了系统的灵活性、可伸缩性和可用性，所以作为一个微服务架构，需要一个服务注册中心来统筹管理服务。

### 关闭自我保护机制

如果在保护期间，实例出现问题，那么客户端很容易拿到实际已经不存在的服务实例，会出现调用失败。这个时候客户端的容错机制就很重要了。（重新请求，断路器）

保护机制，可能会导致服务实例不能够被正确剔除。我们在开发过程中不希望Eureka的自我保护机制开启，不利于开发。我们可以关闭Eureka的自我保护机制（实际生产环境不建议关闭）。

在本地开发时，可使用：eureka.server.enable-self-preservation=false关闭保护机制，使不可用实例能够正常下线。

Eureka服务端配置

|  |
| --- |
| #关闭保护机制，以确保注册中心将不可用的实例正确剔除 eureka.server.enable-self-preservation=false #（代表是5秒，单位是毫秒,清理失效服务的间隔 ） eureka.server.eviction-interval-timer-in-ms=5000 |

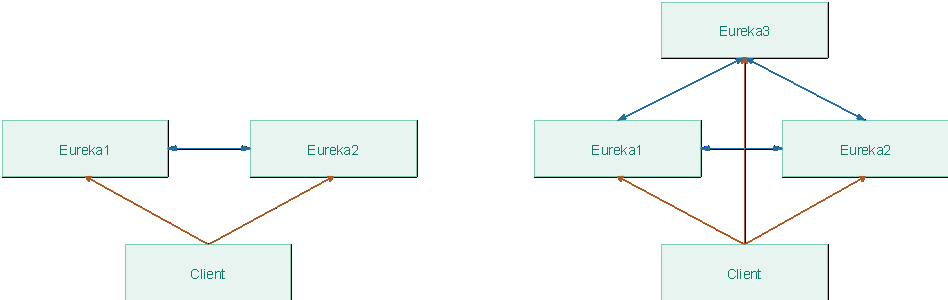
Euraka客户端配置

|  |
| --- |
| # 心跳检测检测与续约时间，测试时将值设置设置小些，保证服务关闭后注册中心能及时踢出服务 # 每间隔5s，向服务端发送一次心跳，证明自己依然”存活“ eureka.instance.lease-renewal-interval-in-seconds=5 # 告诉服务端，如果我10s之内没有给你发心跳，就代表我“死”了，将我踢出掉。 eureka.instance.lease-expiration-duration-in-seconds=10 |

重新启动Eureka服务，此时红色警告变成了自我保护被关闭的警告，然后关闭edu客户端进行测试，发现隔20秒左右，edu实例被注册中心剔除，表明此时自我保护机制被关闭。

## 服务集群

Eureka服务器两两互相注册，其它微服务注册到集群中所有的服务器上。一般生产环境中建议最少配置两台或以上的Eureka服务器集群。

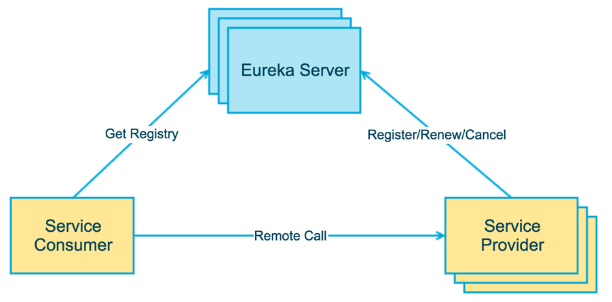


编辑Spring Boot启动配置，将一个Eureka服务端分别以三个不同的端口号启动，启动时尽可能使Eureka服务端同时启动。

多个Eureka服务端需要分别将自身注册到除自身外的Eureka服务端上，此时Eureka客户端只要注册到集群中的任意一个Eureka服务端上，其他Eureka服务端都可以获取到该Eureka客户端的注册信息。但是为了确保Eureka客户端任何时候都能注册到集群中的可用的Eureka服务器上，建议在Eureka客户端中配置注册到集群中所有的Eureka服务端上。

# Ribbon

SpringCloud环境中的客户端负载均衡工具。Ribbon向Eureka询问微服务名称的具体实例信息，根据自己的负载均衡策略决定具体访问哪一个实例。因为Ribbon是工作在Consumer中，而Consumer相对于Provider是客户端，所以Ribbon称为客户端负载均衡。



SpringCloud技术栈中的Nginx。但是Ribbon和Nginx有一个重大区别：Ribbon是一个客户端负载均衡解决方案。

**Nginx和Ribbon对比**

Nginx：服务器端负载均衡——给Tomcat做代理

Ribbon：客户端负载均衡——给Consumer做代理

Ribbon在工作时分成两步：

第一步：先确定Eureka集群中的一个EurekaServer，它优先选择在同一个区域内负载较少的。

第二步：再根据用户指定的策略，在从EurekaServer取到的服务注册列表中选择一个微服务地址。

Ribbon提供了多种负载均衡策略：比如轮询、随机和根据响应时间加权。

# Feign

SpringCloud环境下实现远程方法声明式调用，Feign是对Ribbon的进一步封装，微服务模块中加入了Feign后将同时具备Ribbon和RestTemplate的功能，进一步简化了代码。

获取单个参数，要么使用@PathVariable注解，要么使用@RequestParam注解，不能什么都不写，不写就获取不到这个参数。注解不能省略。

@RequestMapping("/find/by/id")

public ResultEntity<Dept> findById(@RequestParam("deptNo") Integer deptNo);

如果接收实体类类型的数据，那么必须写@RequestBody，而且不论是接口中的方法还是Controller类中的方法都必须写。

@RequestMapping("/add/dept")

public ResultEntity<String> addDept(@RequestBody Dept dept);

# Hystrix

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

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

Hytrix能够提供服务降级、服务熔 断、服务限流、接近实时的监控等方面的功能。

## 服务雪崩

一个微服务局部锁死，蔓延到整个系统，导致整个系统不可用。服务雪崩也称之为扇出。

在微服务架构体系下，服务间的调用错综复杂，交织成一张大网。如果其中某个节点突然无法正常工作，则访问它的众多服务都会被卡住，进而有更多服务被卡住，系统中的线程、CPU、内存等资源有可能被迅速耗尽，最终整个服务体系崩溃。

这样的现象叫服务雪崩。

## 服务熔断

熔断机制发生在Provider端，指的是原本要执行的方法如果出现故障，Consumer就会调用起熔断作用的方法，获取到预期的数据，从而不会影响到其他操作。

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，快速响应错误信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。

## 服务降级

服务降级处理是在客户端(Consummer端)实现完成的，与服务端(Provider端)没有关系。当某个Consummer访问一个Provider却迟迟得不到响应时执行预先设定好的一个解决方案，而不是一直等待。

## 服务监控

# Zuul

网关机制，让项目可以将Zuul微服务作为统一入口。

不同的微服务一般有不同的网络地址，而外部的客户端可能需要调用多个服务的接口才能完成一个业务需求。比如一个电影购票的手机APP，可能会调用电影分类微服务，用户微服务，支付微服务等。如果客户端直接和微服务进行通信，会存在以下问题：

1、客户端会多次请求不同微服务，增加客户端的复杂性

2、存在跨域请求，在一定场景下处理相对复杂

3、认证复杂，每一个服务都需要独立认证

4、难以重构，随着项目的迭代，可能需要重新划分微服务，如果客户端直接和微服务通信，那么重构会难以实施

5、某些微服务可能使用了其他协议，直接访问有一定困难

**Zuul包含了对请求的路由和过滤两个最主要的功能**

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

总体来说，Zuul提供了代理、路由和过滤的功能。

## Zuul相关类

### ZuulFilter抽象类

全类名：com.netflix.zuul.ZuulFilter

开发自定义过滤逻辑，需要继承这个类，并实现如下方法：

**filterType**()方法，返回当前Filter的类型。

pre：在目标微服务方法前执行过滤，可以用来实现登录检查

route：路由到原始目标微服务

post：在目标微服务方法后面执行过滤

error：前面过滤器发生问题调用

**filterOrder**()方法

通过返回的数值，设定Filter执行的顺序。

### IZuulFilter接口

全类名：com.netflix.zuul.IZuulFilter

这是ZuulFilter抽象类实现的一个接口

shouldFilter()

返回true：执行过滤（调用后面的run()方法）

返回false：不过滤（不调用后面的run()方法）

run()

真正执行过滤逻辑。

### RequestContext类

全类名：com.netflix.zuul.context.RequestContext

该类用于获取当前拦截到的请求的响应对象。

RequestContext requestContext = RequestContext.getCurrentContext();

HttpServletRequest request = requestContext.getRequest();

HttpServletResponse response = requestContext.getResponse();

# 实例

## 基础实例

### springcloud-parent

该工程是各个模块的聚合工程，也是父工程。

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <junit.version>4.12</junit.version>  <log4j.version>1.2.17</log4j.version> </properties> <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.SR1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.9.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.0.4</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.31</version>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.2.3</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  </dependencies> </dependencyManagement> |

### springcloud-api

该工程是公共API的模块。

----com.springcloud.entity.Dept.--------------------------------------------------------------

|  |
| --- |
| public class Dept {  private Integer deptNo;  private String deptName;  // 仅仅是为了体现测试效果，并没有业务功能方面的作用  private String dbSource;  } |

### springcloud-provider

服务提供者，通过8001端口访问。

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <dependencies>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jetty</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <!--添加对springcloud-api的依赖-->  <dependency>  <groupId>com.springcloud</groupId>  <artifactId>springcloud-api</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency> </dependencies> |

**创建数据库**

----dept.sql----------------------------------------------------------------------------------

|  |
| --- |
| drop database if exists `cloud\_db\_one`; create database `cloud\_db\_one` character set utf8 collate utf8\_bin; use cloud\_db\_one; drop table if exists `dept`; create table `dept` (  `dept\_no` int(11) not null auto\_increment,  `dept\_name` varchar(500) default null,  `db\_source` varchar(500) default null,  primary key (`dept\_no`) ) engine=innodb default charset=utf8; insert into `dept` values ('1', 'IT', *database*()); insert into `dept` values ('2', 'HR', *database*()); insert into `dept` values ('3', 'MIS', *database*()); insert into `dept` values ('4', 'FUN', *database*()); insert into `dept` values ('5', 'WM', *database*()); insert into `dept` values ('6', 'EI', *database*()); insert into `dept` values ('7', 'UU', *database*()); |

#### 整合MyBatis

----com.springcloud.mapper.DeptMapper---------------------------------------------------------

|  |
| --- |
| public interface DeptMapper {  Dept findById(Integer deptNo);  List<Dept> findAll();  boolean addDept(Dept dept); } |

----mybatis/mapper/DeptMapper.xml-------------------------------------------------------------

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <mapper namespace="com.springcloud.mapper.DeptMapper">  <select id="findById" resultType="Dept" parameterType="Integer">  select dept\_no deptNo,dept\_name deptName,db\_source dbSource from dept where dept\_no=#{deptNo};  </select>  <select id="findAll" resultType="Dept">  select dept\_no deptNo,dept\_name deptName,db\_source dbSource from dept;  </select>  <insert id="addDept" parameterType="Dept">  INSERT INTO dept(dept\_name,db\_source) VALUES(#{deptName},*DATABASE*());  </insert> </mapper> |

----mybatis/mybatis-config.xml----------------------------------------------------------------

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <settings>  <setting name="cacheEnabled" value="true" />  </settings> </configuration> |

----com.springcloud.service.api.DeptService---------------------------------------------------

|  |
| --- |
| public interface DeptService {  Dept findById(Integer deptNo);  List<Dept> findAll();  boolean addDept(Dept dept); } |

----com.springcloud.service.impl.DeptServiceImpl----------------------------------------------

|  |
| --- |
| @Service @Transactional(readOnly=true) public class DeptServiceImpl implements DeptService {  @Autowired  private DeptMapper deptMapper;  @Override  public Dept findById(Integer deptNo) {  return deptMapper.findById(deptNo);  }  @Override  public List<Dept> findAll() {  return deptMapper.findAll();  }  @Override  @Transactional(readOnly=false, propagation=Propagation.*REQUIRES\_NEW*,  rollbackFor=Exception.class)  public boolean addDept(Dept dept) {  return deptMapper.addDept(dept);  } } |

----com.springcloud.controller.DeptController-------------------------------------------------

|  |
| --- |
| @RestController public class DeptController {  @Autowired  private DeptService deptService;  @RequestMapping("/find/by/{deptNo}")  public ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo) {  Dept dept = deptService.findById(deptNo);  return ResultEntity.*successWithData*(dept);  }  @RequestMapping("/find/all")  public ResultEntity<List<Dept>> findAll() {  List<Dept> deptList = deptService.findAll();  return ResultEntity.*successWithData*(deptList);  }  @RequestMapping("/add/dept")  public ResultEntity<String> addDept(@RequestBody Dept dept) {  deptService.addDept(dept);  return ResultEntity.*successWithoutData*();  } } |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| @MapperScan("com.springcloud.mapper") @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 8001 mybatis:  config-location: classpath:mybatis/mybatis-config.xml  type-aliases-package: com.springcloud.entity  mapper-locations:  - classpath:mybatis/mapper/\*\*/\*.xml spring:  application:  name: springcloud-Dept  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: org.gjt.mm.mysql.Driver  url: jdbc:mysql://localhost:3306/cloud\_db\_one  username: root  password: root  dbcp2:  min-idle: 5  initial-size: 5  max-total: 5  max-wait-millis: 200 |

以Spring Boot App的方式运行工程，通过浏览器访问http://localhost:8001/dept/get/all。

### springcloud-consumer

服务消费者，通过80端口访问。

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>com.springcloud</groupId>  <artifactId>springcloud-api</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency> </dependencies> |

----com.springcloud.config.ConfigBean---------------------------------------------------------

|  |
| --- |
| @Configuration public class ConfigBean {  @Bean  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

----com.springcloud.controller.DeptConsumerController-----------------------------------------

|  |
| --- |
| @RestController public class DeptConsumerController {  @Autowired  private RestTemplate restTemplate;  private static final String *URL\_PREFIX* = "http://localhost:8001";  @RequestMapping("/consumer/find/by/{deptNo}")  public ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo) {  return restTemplate.getForObject(*URL\_PREFIX* + "/find/by/" + deptNo, ResultEntity.class);  }  @RequestMapping("/consumer/find/all")  public ResultEntity<List<Dept>> findAll() {  return restTemplate.getForObject(*URL\_PREFIX* + "/find/all", ResultEntity.class);  }  @RequestMapping("/consumer/add/dept")  public ResultEntity<String> addDept(Dept dept) {  return restTemplate.postForObject(*URL\_PREFIX* + "/add/dept", dept, ResultEntity.class);  } } |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 80 |

以Spring Boot App方式运行程序，访问http://localhost/consummer/dept/get/all。

## Eureka单机版

### springcloud-eurek-7000

创建该maven项目并配置依赖。

----pom.xml-----------------------------------------------------------------------------------

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

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 7000 eureka:  instance:  *#eureka服务端的实例名称* hostname: localhost   client:  *#false表示不向注册中心注册自己。* register-with-eureka: false   *#false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务* fetch-registry: false   service-url:  *#指定客户端访问Eureka服务端的URL地址* defaultZone: http://localhost:7000/eureka |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| // 启用Eureka注册中心功能 @EnableEurekaServer @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

以Spring Boot App方式运行工程，访问http://localhost:7000。

### springcloud-provider

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <!--添加对springcloud-api的依赖--> <dependency>  <groupId>com.springcloud</groupId>  <artifactId>springcloud-api</artifactId>  <version>1.0-SNAPSHOT</version> </dependency> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId> </dependency> |

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| eureka:  client:  service-url:  defaultZone: http://localhost:7000/eureka |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| //开启作为Eureka客户端的功能 @EnableEurekaClient @MapperScan("com.springcloud.mapper") @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

### springcloud-consumer

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <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> |

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| eureka:  client:  register-with-eureka: false *#当前是微服务的调用端、消费端，不进行注册* service-url:  defaultZone: http://localhost:7000/eureka |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| @EnableEurekaClient @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

#### 用微服务名替代IP地址加端口号

----com.springcloud.config.ConfigBean---------------------------------------------------------

|  |
| --- |
| @Configuration public class ConfigBean {  @Bean  @LoadBalanced//添加负载均衡  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

这里其实已经用到了Ribbon，这是SpringCloud技术栈中的负载均衡工具，Eureka需要和Ribbon配合起来才能实现服务名替代效果。

----com.springcloud.controller.DeptConsumerController-----------------------------------------

|  |
| --- |
| @SuppressWarnings("unchecked") @RestController public class DeptConsumerController {  @Autowired  private RestTemplate restTemplate;  @SuppressWarnings("unused")  private static final String *URL\_PREFIX* = "http://localhost:8001";  private static final String *URL\_PREFIX\_SERVER\_NAME* = "http://SCMS-DEPT";  @RequestMapping("/consumer/ribbon/find/by/{deptNo}")  public ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo) {  return restTemplate.getForObject(*URL\_PREFIX\_SERVER\_NAME* + "/find/by/" + deptNo, ResultEntity.class);  }  @RequestMapping("/consumer/ribbon/find/all")  public ResultEntity<List<Dept>> findAll() {  return restTemplate.getForObject(*URL\_PREFIX\_SERVER\_NAME* + "/find/all", ResultEntity.class);  }  @RequestMapping("/consumer/ribbon/add/dept")  public ResultEntity<String> addDept(Dept dept) {  return restTemplate.postForObject(*URL\_PREFIX\_SERVER\_NAME* + "/add/dept", dept, ResultEntity.class);  } } |

## Eureka集群版

修改hosts文件，指定URL访问Eureka。如下：

springcloud-eureka-7000:通过域名www.eureka7000.com访问:127.0.0.1 www.eureka7000.com

springcloud-eureka-7001:通过域名www.eureka7001.com访问:127.0.0.1 www.eureka7001.com

springcloud-eureka-7002:通过域名www.eureka7002.com访问:127.0.0.1 www.eureka7002.com

### springcloud-eureka集群工程

通过复制的方式创建springcloud-eureka-7001和springcloud-eureka-7002工程，并添加如下依赖。

----pom.xml-----------------------------------------------------------------------------------

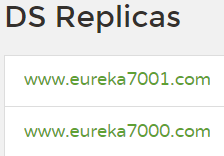
|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId> </dependency> |

然后配置eureka集群工程的springboot的yml配置文件。三个eureka集群工程的配置中port、defaultZone属性不同，需要根据具体工程作出对应的修改。

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 7002 eureka:  instance:hostname: localhost  client:register-with-eureka: falsefetch-registry: false  service-url:defaultZone: http://www.eureka7001.com:7001/eureka/,http://www.eureka7000.com:7000/eureka/ |

分别以SpringBoot App方式运行三个工程，访问http://www.eureka7000.com:7000/、http://www.eureka7001.com:7001/ 和http://www.eureka7002.com:7002/。如果DS Replicas处各自显示的是另外两个工程的地址，则表示集群eureka配置成功。



### springcloud-provider

修改application.yml配置文件中的defaultZone，将其设置为三个eureka集群的地址，防止其中一个eureka微服务宕机导致整个系统无法运行。

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| eureka:  client:  service-url: *# defaultZone: http://localhost:7000/eureka* defaultZone: http://www.eureka7000.com:7000/eureka,http://www.eureka7001.com:7001/eureka,http://www.eureka7002.com:7002/eureka |

配置完成后运行springcloud-provider工程，刷新三个eureka页面均出现注册信息。



## Ribbon

### springcloud-provider集群工程

首先搭建Provider集群，根据集群Provider创建对应的数据库，这里创建两个数据库，将springcloud-provider工程中的创建数据库的SQL语句中的cloud\_db\_one分别改为cloud\_db\_two和cloud\_db\_three。

----springcloud-provider-two的dept.sql--------------------------------------------------------

|  |
| --- |
| drop database if exists `cloud\_db\_two`; create database `cloud\_db\_one` character set utf8 collate utf8\_bin; use cloud\_db\_two;  ... |

----springcloud-provider-three的dept.sql------------------------------------------------------

|  |
| --- |
| drop database if exists `cloud\_db\_three`; create database `cloud\_db\_one` character set utf8 collate utf8\_bin; use cloud\_db\_three;  ... |

通过复制springcloud-provider的方式分别创建springcloud-provider-two和springcloud-provider-three工程，这三个工程要求pom.xml配置文件一致，Java代码也一致。

然后修改新建的两个工程的application.yml配置文件，按照如下方式修改

----springcloud-provider-two------------------------------------------------------------------

server.port: 8002

spring.datasource.url: jdbc:mysql://localhost:3306/cloud\_db\_two

---- springcloud-provider -three--------------------------------------------------------------

server.port: 8003

spring.datasource.url: jdbc:mysql://localhost:3306/cloud\_db\_three

springcloud-provider-two和springcloud-provider-three要想成为一个集群，分担负载，那么对外暴露的微服务名称必须一致。也就是说，spring.application.name配置项要一致。

分别访问如下网址，如果均能获取到对应数据库中的结果，则说明provider集群配置成功。

http://localhost:8003/find/all

http://localhost:8003/find/all

http://localhost:8003/find/all

### springcloud-consumer

前面为了Consummer80工程能够通过微服务名称调用微服务其实已经完成Ribbon的配置了，这里再重申一下Ribbon的配置：

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <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> |

确认ConfigBean中getRestTemplate()方法的@LoadBalanced注解。

----com.springcloud.config.ConfigBean---------------------------------------------------------

|  |
| --- |
| @Configuration public class ConfigBean {  @Bean  @LoadBalanced//添加负载均衡  public RestTemplate getRestTemplate() {  return new RestTemplate();  } } |

在浏览器中输入http://localhost/consumer/ribbon/find/all后多次刷新访问，每次获取数据的数据库都不同。

## Feign

### springcloud-api

----pom.xml-----------------------------------------------------------------------------------

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId> </dependency> |

----com.springcloud.api.DeptRemoteService-----------------------------------------------------

|  |
| --- |
| //指定要调用的微服务名称 @FeignClient(value="SCMS-Dept") public interface DeptRemoteService {  @RequestMapping("/find/by/{deptNo}")  ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo);  @RequestMapping("/find/all")  ResultEntity<List<Dept>> findAll();  @RequestMapping("/add/dept")  ResultEntity<String> addDept(@RequestBody Dept dept); } |

### springcloud-consumer-feign

通过复制springcloud-consumer工程来创建springcloud-consumer-feign工程，两者的pom.xml配置相同。在springcloud-consumer-feign工程中不再需要RestTemplate，而且内置了Ribbon。

将DeptConsumerController类删除，并创建FeignDeptController类。

----com.springcloud.controller.FeignDeptController--------------------------------------------

|  |
| --- |
| @RestController public class FeignDeptController {  @Autowired  private DeptRemoteService deptRemoteService;  @RequestMapping("/consumer/ribbon/find/by/{deptNo}")  public ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo) {  return deptRemoteService.findById(deptNo);  }  @RequestMapping("/consumer/ribbon/find/all")  public ResultEntity<List<Dept>> findAll() {  return deptRemoteService.findAll();  }  @RequestMapping("/consumer/ribbon/add/dept")  public ResultEntity<String> addDept(Dept dept) {  return deptRemoteService.addDept(dept);  } } |

在主程序的类名上添加@EnableFeignClients注解。

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| //启用Feign客户端功能 @EnableFeignClients @EnableEurekaClient @SpringBootApplication public class SpringCloudMainType {  ...  } |

在浏览器中输入http://localhost/consumer/ribbon/find/all后多次刷新访问，每次获取数据的数据库都不同。

## Hystrix

### 服务熔断

#### springcloud-provider-hystrix

通过复制springcloud-provider工程来创建springcloud-provider-hystrix工程，并在新工程的pom.xml配置文件中添加hystrix依赖。

----pom.xml-----------------------------------------------------------------------------------

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

修改新工程的application.yml配置文件中的server.port和spring.application.name的值。

|  |
| --- |
| server:  port: 8004 spring:  application:  name: SCMS-Dept-Hystrix |

修改com.springcloud.controller.DeptController类，在其中添加发生熔断时调用的方法。@HystrixCommand注解的fallbackMethod属性指定的方法入参和返回值都和原方法一致，这保证了原方法出现问题时可以由fallbackMethod方法来接替，从而返回一个可以让调用者继续执行其他任务的响应信息。

----com.springcloud.controller.DeptController-------------------------------------------------

|  |
| --- |
| // @HystrixCommand用于指定熔断时调用的备份方法，fallbackMethod属性指定方法名 @HystrixCommand(fallbackMethod="findByIdBackup") @RequestMapping("/find/by/{deptNo}") public ResultEntity<Dept> findById(@PathVariable("deptNo") Integer deptNo) throws InterruptedException {  if(deptNo == 555) {  throw new RuntimeException();  }  if(deptNo == 999) {  Thread.*sleep*(5000);  }  Dept dept = deptService.findById(deptNo);  return ResultEntity.*successWithData*(dept); } // 备份方法 public ResultEntity<Dept> findByIdBackup(@PathVariable("deptNo") Integer deptNo) {  return ResultEntity.*failed*("Get dept by "+deptNo+" failed"); } @HystrixCommand(fallbackMethod="findAllBackup") @RequestMapping("/find/all") public ResultEntity<List<Dept>> findAll() {  List<Dept> deptList = deptService.findAll();  return ResultEntity.*successWithData*(deptList); } public ResultEntity<List<Dept>> findAllBackup() {  return ResultEntity.*failed*("Get Dept List Failed"); } |

在主程序的类上添加@EnableCircuitBreaker注解启用对Hystrix熔断机制的支持。

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| // 启用断路器功能 @EnableCircuitBreaker @EnableEurekaClient @MapperScan("com.springcloud.mapper") @SpringBootApplication public class SpringCloudMainType {  ... } |

这时启动Eureka集群和springcloud-provider-hystrix工程，在浏览器中分别输入如下地址：

http://localhost:8004/find/by/555

结果：{"data":null,"result":"FAILED","message":"Get dept by 555 failed"}

http://localhost:8004/find/by/999

结果：{"data":null,"result":"FAILED","message":"Get dept by 999 failed"}

http://localhost:8004/find/all

### 服务降级

#### springcloud-api

创建DeptRemoteServiceFallBackFactory类实现FallbackFactory接口。

----com.springcloud.factory.DeptRemoteServiceFallBackFactory----------------------------------

|  |
| --- |
| /\*关于@Component注解，当前类也必须加入IOC容器，而加入IOC容器有两个要求：  1.当前类使用@Component注解标记  2.当前类所在包能够被扫描（例如：主启动类所在包或其子包）  关于FallbackFactory接口，当前类必须实现FallbackFactory接口才能提供降级机制中的备用解决方案  1.create()方法要创建和远程接口类型一致的对象。  2.如果调用远程方法失败，那么降级机制会调用create()方法返回值对象的相同方法拿到相同类型的返回值  3.程序在降级后可以继续执行，不会受到其他服务故障影响 \*/ @Component public class DeptRemoteServiceFallBackFactory implements FallbackFactory<DeptRemoteService> {  @Override  public DeptRemoteService create(Throwable cause) {  return new DeptRemoteService() {  @Override  public ResultEntity<Dept> findById(Integer deptNo) {  return ResultEntity.*failed*("remote call failed by deptNo="+deptNo);  }  @Override  public ResultEntity<List<Dept>> findAll() {  return ResultEntity.*failed*("remote call failed by find all");  }  @Override  public ResultEntity<String> addDept(Dept dept) {  return ResultEntity.*failed*("remote call failed by add dept");  }  };  } } |

修改DeptRemoteService接口上的@FeignClient注解。

|  |
| --- |
| // 使用fallbackFactory属性指定了具体降级工厂类，降级机制才能够生效 //@FeignClient(value="SCMS-Dept") @FeignClient(value="SCMS-Dept", fallbackFactory= DeptRemoteServiceFallBackFactory.class)  public interface DeptRemoteService {...} |

#### springcloud-consumer-feign

在application.yml配置文件中添加启用hystrix的配置。

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| feign:  hystrix:  enabled: true |

先启动Eureka集群，再启动一个没有加熔断机制的Provider，最后启动Feign Consummer。先测试正常访问，然后关掉Provider后再访问。

### 服务监控

#### springcloud-hystrix-hashboard

创建springcloud-hystrix-dashboard工程，并配置依赖。

----pom.xml-----------------------------------------------------------------------------------

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

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 9001 |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| // 启用服务监控仪表盘功能 @EnableHystrixDashboard @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

#### springcloud-provider-hystrix

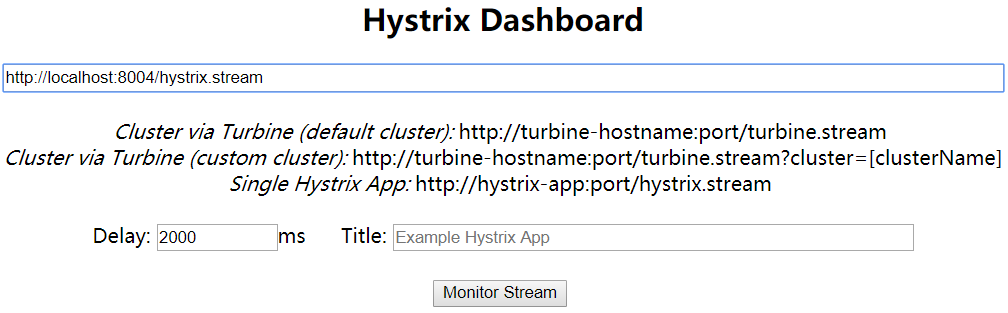
给被监控的微服务添加依赖，被监控的微服务也需要配置好熔断机制才可以，目前springcloud-provider-hystrix满足这个要求。

----pom.xml---------------------------------------------------------------------------

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId> </dependency> |

打开监控首页：http://localhost:9001/hystrix

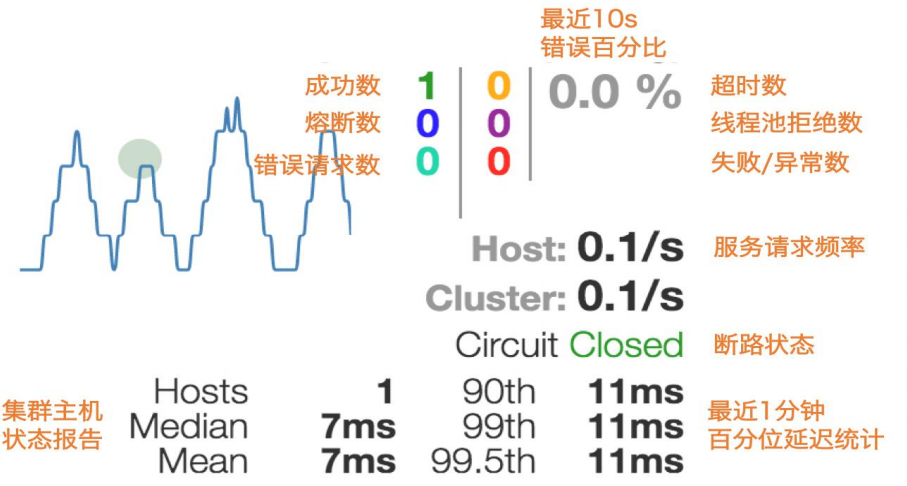
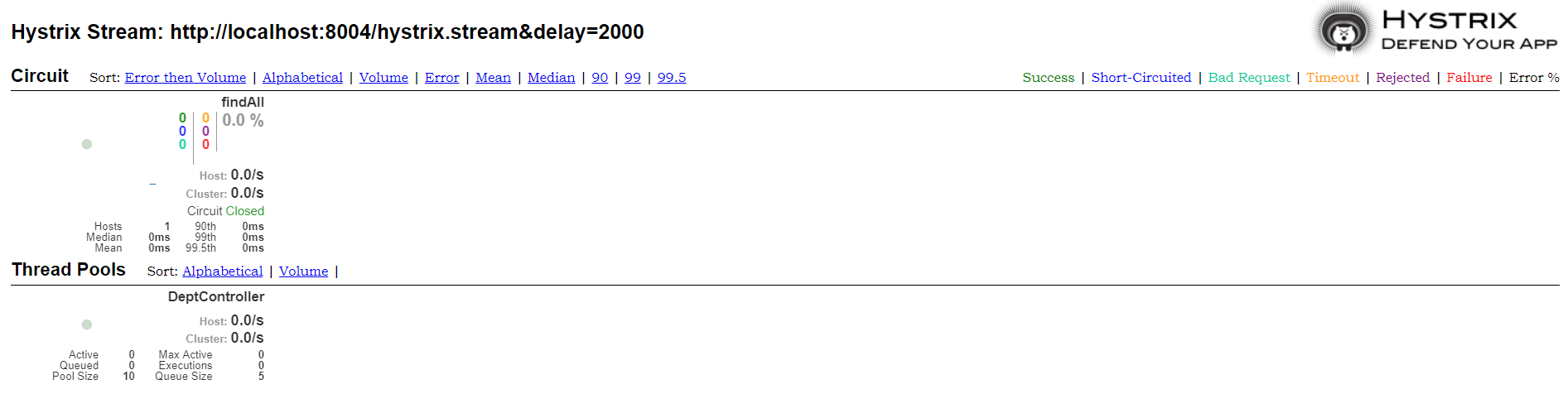
直接查看监控数据：http://localhost:8004/hystrix.stream，其中8004是被监控的微服务端口



图形化界面方式查看监控数据：将http://localhost:8004/hystrix.stream填入文本框中并单机Monitor Stream按钮进入图形化界面。

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

Title：该参数对应了头部标题Hystrix Stream之后的内容，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。



## Zuul

### springcloud-zuul

----pom.xml---------------------------------------------------------------------------

|  |
| --- |
| <dependencies>  <!-- 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> </dependencies> |

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| server:  port: 9002 spring:  application:  name: scms-zuul-gateway eureka:  client:  service-url:  defaultZone: http://www.eureka7000.com:7000/eureka,http://www.eureka7001.com:7001/eureka,http://www.eureka7002.com:7002/eureka |

----com.springcloud.SpringCloudMainType-------------------------------------------------------

|  |
| --- |
| // 启用Zuul网关代理功能 @EnableZuulProxy @SpringBootApplication public class SpringCloudMainType {  public static void main(String[] args) {  SpringApplication.*run*(SpringCloudMainType.class, args);  } } |

### 测试

启动eureka集群，启动springcloud-provider-hystrix，启动springcloud-zuul。查看eureka上注册的服务。



不通过路由访问：http://localhost:8004/find/all

通过路由访问：http://localhost:9002/scms-dept-hystrix/find/all

**注意**：在Eureka中注册的微服务SCMS-DEPT-HYSTRIX名称全为大写，但是在使用zuul访问时必须要全部是小写，否则无法访问。

### 路由访问映射规则

#### 隐藏微服务名称

在springcloud-zuul工程的application.yml中添加如下配置。

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| zuul:routes:  DIYname.serviceId: scms-dept-hystrix  DIYname.path: /demo/\*\* |

此时可以使用两种方式进行访问，分别为：http://localhost:9002/scms-dept-hystrix/find/all和http://localhost:9002/demo/find/all来进行访问。

#### 原真实服务名忽略

经过上面的配置后，既可以使用别名访问，又可以使用原服务名访问。如果希望只能通过别名访问，可以做这样的配置：

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| zuul:ignored-services: scms-dept-hystrix  routes:  DIYname.serviceId: scms-dept-hystrix  DIYname.path: /demo/\*\* |

此时就无法通过http://localhost:9002/scms-dept-hystrix/find/all来访问了，就只能通过别名的方式，即http://localhost:9002/demo/find/all来访问。

#### 统一设置公共前缀

设置公共前缀后所有微服务名前面都需要再加上前缀才可以访问。

----application.yml---------------------------------------------------------------------------

|  |
| --- |
| zuul:  prefix: /good  ignored-services: scms-dept-hystrix  routes:  DIYname.serviceId: scms-dept-hystrix  DIYname.path: /demo/\*\* |

此时就只能通过http://localhost:9002/good/demo/find/all来访问了。