

# 过去和未来

学完Spring Boot之后，我们有了这么一个疑问：当公司的服务器撑不住了怎么办？

以前有一个简单的解决方法：加服务器，实现负载均衡。

这样做十分简单粗暴，属于横向解决问题。

负载均衡就是将一个项目完整的复制一份，到另一台服务器上  
努力实现两台服务器的负载均衡

但是假如一个公司有很多业务：签到，订单，支付，物流....

假如签到用的资源非常少，而其他资源用的非常多，那么虽然加服务器实现负载均衡可以解决这个问题，但是有非常多的资源会被浪费

我们想给签到少一点资源，给其他业务多一点资源

在这种情况下，出现了微服务架构体系，但是微服务架构存在问题：

**分布式架构会遇到的问题：**

1. 这么多服务分布在不同的服务器上，用户该如何去访问？

解决：给用户一个相同的接口，一个共同的网管。然后用这个接口去分发到具体的服务器上

2. 这么多服务，如何进行通信？
3. 这么多服务，如何进行统一的管理？

只要一个统一的管理平台即可

4. 服务器崩了怎么办？

熔断机制

**解决方案：**Spring Cloud

Spring Cloud是一套生态，而不是一个具体的解决方案  
生态中的解决方案用来解决分布式的这些问题

1. Spring Cloud **Netflix**：一站式解决方案
  1. 用户访问：**Api** 网关， **zuul** 组件
  2. 服务通信：**Feign**

Feign 基于 HttpClient

HttpClient 基于 Http

Http, 同步并阻塞

3. 统一管理：服务注册与发现：Eureka

4. 服务器崩了：熔断机制：Hystrix

虽然这个解决方案很好，但是在2018年年底，Netflix 宣布无限期停止维护

但是虽然不维护了，还是有很多东西还在用，比如熔断机制

2. Apache Dubbo zookeeper：半自动，需要整合别人

1. Api 网关，没有！所以要么找第三方，要么自己写

2. 服务通信：RPC 框架：Dubbo

3. 服务注册与发现：Zookeeper (包含 Hadoop, Hive)等

4. 熔断机制：没有，借助了 Hystrix

这个解决方案不太完善，但是正在完善

比如 Dubbo3.0, 正在进行升级，将会成为 Apache 的顶级项目，是一种碾压性的有事

3. Spring Cloud Alibaba：一站式解决方案

还没有孵化完成

---

比Spring Cloud更新一代的解决方案：服务网格

下一代微服务标准，Server Mess

代表解决方案：istio

---

## 常见面试题

---

1. 什么是微服务？
2. 微服务之间是如何进行通信的？
3. Spring Cloud和Dubbo的区别？
4. Spring Boot和Spring Cloud的理解？
5. 什么是服务熔断？什么是服务降级？
6. 微服务的优缺点？微服务有什么缺点？
7. 微服务的技术栈？
8. Eureka 和 zookeeper 都可以进行注册与发现，他们的区别？

.....

## 微服务概述

# 什么是微服务

---

微服务是近几年流行的一种服务器架构。

通常而言，微服务是一种架构模式或者说是一种架构风格。

他提倡将单一的应用程序划分为一组组小的服务，每个服务运行在单独的 **进程** 内，相互协调，提供最终价值。

服务之间 **采用轻量级的通信机制进行互通**，每个服务围绕着具体的业务进行构建。

避免统一的，集中式的管理机制。

**可以选择不同的语言编写服务，也可以使用不同的数据存储。**

---

- 说白了，彻底解耦

## 微服务与微服务架构

---

### 微服务

强调的是服务的大小，它关注的是某一个点，解决某一个问题的服务应用

一个小小的组件

### 微服务架构

一种架构模式，一种思想

## 微服务优缺点

---

优点：

1. 单一职责
2. 每个服务足够内聚，足够小，代码容易理解，能够聚焦一个指定的业务或者需求
3. 开发简单，效率高，一个服务可能只做一件事
4. 可以被小团队开发，比如2~5个人
5. 松耦合，有功能意义的服务，无论是开发阶段还是部署阶段都是独立的
6. 可以使用不同语言进行开发
7. 容易和第三方进行集成，允许容易且灵活地方式集成自动部署，通过持续继承工具，比如 `jenkins`，`Hudson`，`bamboo`
8. 容易被一个开发人员理解，修改和维护。
9. 允许融合最新技术
10. 只是纯业务逻辑代码，不用和 `html`，`css` 或者其他界面混合
11. **每个服务都有自己的存储能力，可以有自己的数据库，也可以有统一数据库**

缺点：

- 1. 开发人员要处理分布式系统复杂性
- 2. 多服务运维难度随着服务的增加而增加
- 3. 系统部署依赖
- 4. 服务通信成本
- 5. 数据一致性
- 6. 系统集成测试
- 7. 性能监控

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	Neflix 的 Archaius , Alibaba 的 Diamond 等
服务注册与发现	Eureka , Consul , Zookeeper 等
服务调用	Rest , RPC , gRPC (Google)
服务熔断器	Hystrix , Envoy 等
负载均衡	Ribbon , Nginx 等

微服务条目	落地技术
服务接口调用（客户端调用服务的简化工具）	Feign 等
消息队列	Kafka , RabbitMQ , ActiveMQ 等
服务配置中心管理	SpringCloud Config , Chef 等
服务路由（API网关）	Zuul 等
服务监控	Zabbix , Nagios , Metrics , Specatator 等
全链路追踪	Zipkin , Brave , Dapper 等
服务部署	Docker , OpenStack , Kubernetes 等
数据流操作开发包	SpringCloud Stream （封装 Redis, Rabbit, Kafka 等发送接收消息）
事件消息总线	SpringCloud Bus

## 什么是Spring Cloud

Spring Cloud，基于Spring Boot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的开源组件之外，还有一些选型中立的开源组建。

Spring Cloud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系列工具，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。他们都可以使用SpringBoot的开发风格做到一键启动和部署。

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结合体，俗称微服务全家桶。

## SpringCloud和SpringBoot的关系

- SpringBoot专注于快速开发单个个体服务
- 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，分布式会话等集成服务
- SpringBoot可以离开SpringCloud独立使用，但是SpringCloud离不开Springboot
- SpringBoot专注于快速开发，方便的开发单体微服务，SpringCloud关注全局的服务之力框架

# Dubbo和SpringCloud的选择

---

## Dubbo和SpringCloud的对比

	Dubbo	SpringCloud
服务注册中心	Zookeeper	SpringCloud 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	SpringCloud Netflix Hystrix
服务网关	无	SpringCloud Netflix Zuul
分布式配置	无	SpringCloud Config
服务跟踪	无	SpringCloud Sleuth
消息总线	无	SpringCloud Bus
数据流	无	SpringCloud Stream
批量任务	无	SpringCloud Task

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

严格来说，这两种方式各有优劣，虽然从一定程度上来说，后者牺牲了服务调用的性能，但是也避免原生RPC带来的问题。

而且REST比RPC更加灵活。

### 成品和组装的区别

SpringCloud 比 Dubbo更加强大，涵盖面积更加广泛，而且作为Spring的拳头产品，他也可以与SpringFramework，SpringBoot，SpringData，SpringBatch等其他Spring项目相结合。

使用SpringCloud就像是使用品牌机，简单易上手。

使用Dubbo就像是使用组装机，更加灵活，对于新手更难。

### 社区支持与更新力度

Dubbo停止了5年左右的更新，虽然2017.7重启了，但是对于新技术的发展需要开发者自己拓展升级。比如当当网的DubboX。

但是这明显不现实，中小型企业没有能力去修改Dubbo的源码+一整套生态结构。

### 解决的问题不同

Dubbo的定位是一款RPC框架，而SpringCloud是微服务架构下的一站式解决方案

---

# SpringCloud的版本号

OVERVIEW	LEARN	SAMPLES
----------	-------	---------

## Documentation

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

Hoxton SR3	CURRENT GA	Reference Doc.	API Doc.
Hoxton	SNAPSHOT	Reference Doc.	API Doc.
Greenwich SR5	GA	Reference Doc.	API Doc.
Greenwich	SNAPSHOT	Reference Doc.	API Doc.

上面的文档命名看起来很奇怪，所以我们需要了解：

SpringCloud是由众多独立子项目组成的大型综合项目，每一个子项目都有着不同的发行节奏，都有自己的版本号。

所以为了避免混淆，没有通过版本号的方式，而是通过命名的方式来进行。

这些版本的命名方式使用了伦敦地铁站的名称，同时根据字母的顺序来进行版本时间排序。

比如最早的Release版本：Angel。第二个Release版本：Brixton。然后是：Camden、Dalston、Edgware。

现在已经到了H版本了

## 参考书

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

SpringCloud中文API：<https://www.springcloud.cc/spring-cloud-dalston.html>

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

SpringCloud中文网：<https://www.springcloud.cc/>

## 总体开始

SpringCloud的学习就打一个整体的项目，然后在这个项目下建立分项目即可

- 我们使用一个 Dept 部门模块做一个微服务通用案例 Consumer 消费者(浏览器)通过REST调用 Provider 提供者(服务器)提供的服务
- Maven的分包分模块架构复习：

```
1  一个简单的Maven项目是这样的：
2
3  -- app-parent: 一个父项目(app-parent)，聚合了很多子项目(app-util, app-dao, app-web....)
4      |-- pom.xml
5      |
6      |-- app-core
7          |--pom.xml
8      |
9      |-- app-web
10         |--pom.xml
11     ....
```

一个父工程带有多个子 **Module** 模块

## SpringCloud版本如何选择

SpringBoot	SpringCloud	关系
1.2.x	Angel版本	兼容SpringBoot 1.2.x
1.3.x	Brixton版本	兼容SpringBoot 1.3.x和1.4.x
1.4.x	Camden版本	兼容SpringBoot 1.4.x和1.5.x
1.5.x	Dalston版本	兼容SpringBoot 1.5.x
1.5.x	Edgware版本	兼容SpringBoot 1.5.x
2.0x	Finchley版本	兼容SpringBoot 2.0x
2.1x	Greenwich版本	兼容Spring Boot 2.1.x

更详细的版本对应：

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

使用最后的这两个



# 搭建环境

## 总项目搭建

1. 创建一个普通的maven项目：springcloud，然后干掉src目录
2. 总pom文件

```
1      <!--打包方式现在改为pom方式打包-->
2      <packaging>pom</packaging>
3
4      <!--包版本的管理-->
5      <properties>
6          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
7          <maven.compiler.source>1.8</maven.compiler.source>
8          <maven.compiler.target>1.8</maven.compiler.target>
9          <junit.version>4.12</junit.version>
10         <log4j.version>1.2.17</log4j.version>
11         <lombok.version>1.16.18</lombok.version>
12     </properties>
13
14     <!--坐标管理：这个dependencyManagement是管理者，子项目可以使用-->
15     <dependencyManagement>
16         <dependencies>
17             <dependency>
18                 <groupId>org.springframework.cloud</groupId>
19                 <artifactId>spring-cloud-alibaba-dependencies</artifactId>
20                 <version>0.2.0.RELEASE</version>
21                 <type>pom</type>
22                 <scope>import</scope>
23             </dependency>
24             <!--springCloud的依赖-->
25             <dependency>
26                 <groupId>org.springframework.cloud</groupId>
27                 <artifactId>spring-cloud-dependencies</artifactId>
28                 <version>Greenwich.SR1</version>
29                 <type>pom</type>
30                 <scope>import</scope>
31             </dependency>
32             <!--SpringBoot-->
33             <dependency>
34                 <groupId>org.springframework.boot</groupId>
35                 <artifactId>spring-boot-dependencies</artifactId>
36                 <version>2.1.4.RELEASE</version>
37                 <type>pom</type>
38                 <scope>import</scope>
39             </dependency>
40             <!--数据库-->
41             <dependency>
42                 <groupId>mysql</groupId>
43                 <artifactId>mysql-connector-java</artifactId>
44                 <version>5.1.47</version>
45             </dependency>
```

```

46         <dependency>
47             <groupId>com.alibaba</groupId>
48             <artifactId>druid</artifactId>
49             <version>1.1.10</version>
50         </dependency>
51         <!--SpringBoot启动器-->
52         <dependency>
53             <groupId>org.mybatis.spring.boot</groupId>
54             <artifactId>mybatis-spring-boot-starter</artifactId>
55             <version>1.3.2</version>
56         </dependency>
57         <!--日志测试-->
58         <dependency>
59             <groupId>ch.qos.logback</groupId>
60             <artifactId>logback-core</artifactId>
61             <version>1.2.3</version>
62         </dependency>
63         <dependency>
64             <groupId>junit</groupId>
65             <artifactId>junit</artifactId>
66             <version>${junit.version}</version>
67         </dependency>
68         <dependency>
69             <groupId>log4j</groupId>
70             <artifactId>log4j</artifactId>
71             <version>${log4j.version}</version>
72         </dependency>
73         <!--lombok-->
74         <dependency>
75             <groupId>org.projectlombok</groupId>
76             <artifactId>lombok</artifactId>
77             <version>${lombok.version}</version>
78         </dependency>
79     </dependencies>
80 </dependencyManagement>

```

## 子项目实体类

### 对应pojo

1. 新建一个maven项目： `springcloud-api`，对应实体类，进去之后干掉测试
2. `pom.xml`

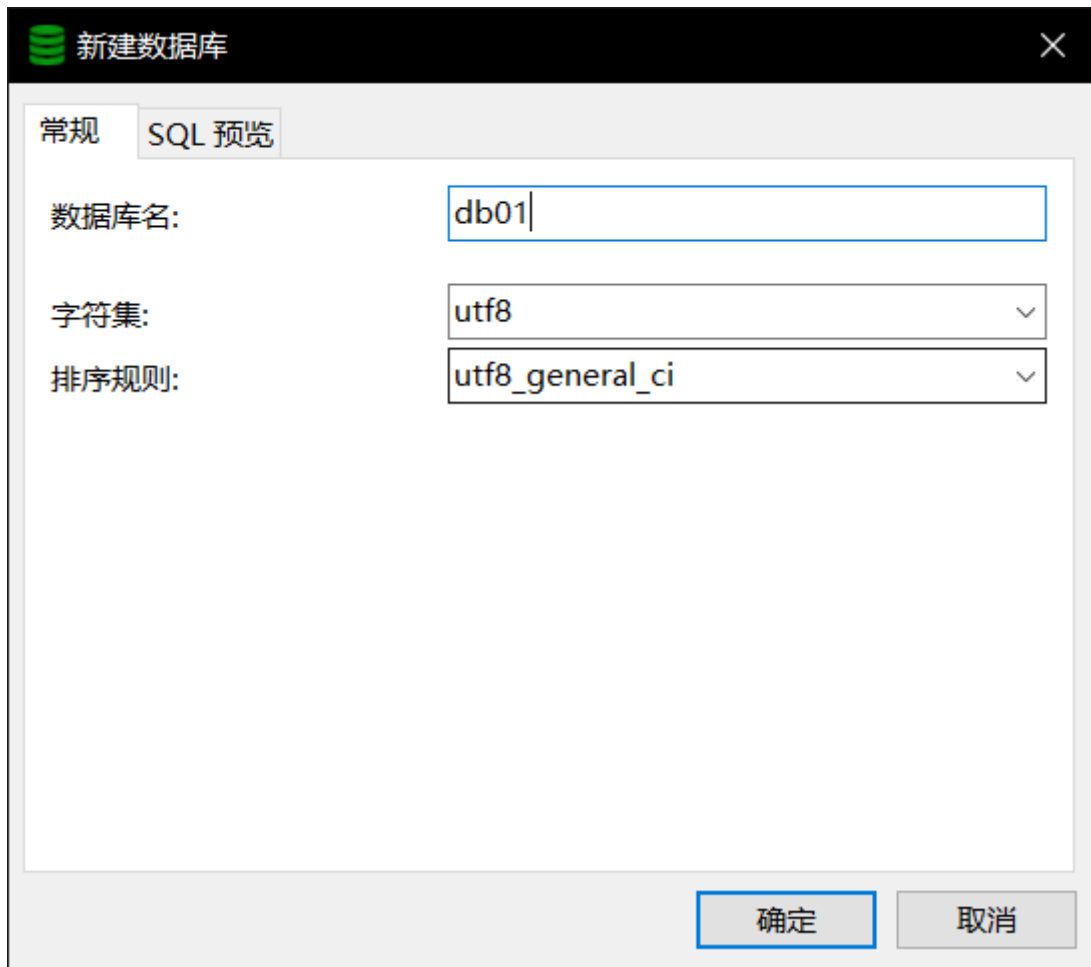
```

1     <dependencies>
2         <dependency>
3             <groupId>org.projectlombok</groupId>
4             <artifactId>lombok</artifactId>
5         </dependency>
6     </dependencies>

```

从这里开始引入的坐标都是父项目的坐标

3. 建立一个新的 **数据库**：就叫db01



新建数据库

常规 SQL 预览

数据库名: db01

字符集: utf8

排序规则: utf8\_general\_ci

确定 取消

4. 在db01这个数据库上新建数据表

Table: dept Comment: 部门表

```
deptno    bigint -- part of primary key
dname     varchar(60)
db_source varchar(60)
```

```
SQL Script
```

```
create table dept
(
    deptno bigint auto_increment,
    dname varchar(60) null,
    db_source varchar(60) null,
    constraint dept_pk
        primary key (deptno)
)
comment '部门表';
```

Action: Execute in database ? Execute Cancel

```
1 create table dept
2 (
3     deptno bigint auto_increment,
4     dname varchar(60) null,
5     db_source varchar(60) null,
6     constraint dept_pk
7         primary key (deptno)
8 )
9 comment '部门表';
```

- `deptno` : 主键
- `dname` : 部门名称
- `db_source` : 数据库的名字, 因为是分布式, 所以有数据库的名字, 我们要知道这个是从哪个数据库里出来的

## 5. 插入数据

```

1  # DATABASES()会读取当前数据库的名字
2  insert into dept (dname, db_source) values ('开发部',DATABASE());
3  insert into dept (dname, db_source) values ('人事部',DATABASE());
4  insert into dept (dname, db_source) values ('财务部',DATABASE());
5  insert into dept (dname, db_source) values ('市场部',DATABASE());
6  insert into dept (dname, db_source) values ('运维部',DATABASE());

```

	deptno	dname	db_source
1	1	开发部	db01
2	2	人事部	db01
3	3	财务部	db01
4	4	市场部	db01
5	5	运维部	db01

6. 新建实体类: `com.bean.pojo.Dept`

```

1  package com.bean.pojo;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Data;
5  import lombok.NoArgsConstructor;
6  import lombok.experimental.Accessors;
7
8  import java.io.Serializable;
9
10 //实现序列化接口
11 @Data
12 @AllArgsConstructor
13 @NoArgsConstructor
14 @Accessors(chain = true)//这个注解代表着链式写法，说明Dept现在支持链式写法了
15 public class Dept implements Serializable {
16     private Long deptno;
17     private String dname;
18
19     /**
20      * 这个数据存在哪个数据库的字段
21      * 因为微服务，一个服务对应一个数据库，同一个信息可能存在不同的数据库
22      */
23     private String db_source;
24 }

```

## 子项目生产者

对应提供者，8001是端口号

对应dao, server, controller。作为服务提供

1. 新建一个 maven 项目：springcloud-provider-8001，干掉test

2. pom.xml

```
1      <dependencies>
2          <!--我们需要拿到实体类，所以需要 api这个模块-->
3          <dependency>
4              <groupId>com.bean</groupId>
5              <artifactId>springcloud-api</artifactId>
6              <version>1.0-SNAPSHOT</version>
7          </dependency>
8
9          <!--junit-->
10         <dependency>
11             <groupId>junit</groupId>
12             <artifactId>junit</artifactId>
13         </dependency>
14
15         <!--mysql-->
16         <dependency>
17             <groupId>mysql</groupId>
18             <artifactId>mysql-connector-java</artifactId>
19         </dependency>
20         <!--数据源-->
21         <dependency>
22             <groupId>com.alibaba</groupId>
23             <artifactId>druid</artifactId>
24         </dependency>
25         <!--logback, 日志-->
26         <dependency>
27             <groupId>ch.qos.logback</groupId>
28             <artifactId>logback-core</artifactId>
29         </dependency>
30         <!--mybatis-springboot启动器-->
31         <dependency>
32             <groupId>org.mybatis.spring.boot</groupId>
33             <artifactId>mybatis-spring-boot-starter</artifactId>
34         </dependency>
35         <!--test-->
36         <dependency>
37             <groupId>org.springframework.boot</groupId>
38             <artifactId>spring-boot-test</artifactId>
39         </dependency>
40         <!--web-->
41         <dependency>
42             <groupId>org.springframework.boot</groupId>
43             <artifactId>spring-boot-starter-web</artifactId>
44         </dependency>
45         <!--jetty, 类似tomcat-->
46         <dependency>
47             <groupId>org.springframework.boot</groupId>
48             <artifactId>spring-boot-starter-jetty</artifactId>
49         </dependency>
50         <!--热部署-->
51         <dependency>
52             <groupId>org.springframework.boot</groupId>
53             <artifactId>spring-boot-devtools</artifactId>
54         </dependency>
```

## 生产者要链接数据库

## 3. application.yaml

```

1  # 端口号
2  server:
3    port: 8001
4
5  # mybatis的配置，其中配置别名时虽然是一个项目的包名，但是我们通过坐标拿到了这些
6  mybatis:
7    type-aliases-package: com.bean.pojo
8    mapper-locations: classpath:mybatis/mapper/*.xml
9    config-location: classpath:mybatis/mybatis-config.xml
10
11 # spring配置
12 spring:
13   application:
14     name: springcloud-provider-dept
15   datasource:
16     type: com.alibaba.druid.pool.DruidDataSource
17     driver-class-name: org.gjt.mm.mysql.Driver
18     url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
19         8&serverTimezone=Asia/Shanghai
20     username: root
21     password: root

```

## 4. mybatis-config.xml

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration
3    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4    "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6  <configuration>
7    <settings>
8      <!--开启二级缓存-->
9      <setting name="cacheEnabled" value="true"/>
10    </settings>
11  </configuration>

```

## 5. DeptMapper.xml

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5
6  <mapper namespace="com.bean.dao.DeptDao">
7
8    <insert id="addDept" parameterType="Dept">
9      insert into dept (dname, db_source)
10     values (#{dname}, DATABASE())
11    </insert>
12
13    <select id="queryById" resultType="Dept" parameterType="Long">

```

```

13         select * from dept where deptno = #{deptno};
14     </select>
15
16     <select id="queryAll" resultType="Dept">
17         select * from dept;
18     </select>
19
20 </mapper>

```

## 6. DeptDao

```

1     package com.bean.dao;
2
3     import com.bean.pojo.Dept;
4     import org.apache.ibatis.annotations.Mapper;
5     import org.springframework.stereotype.Repository;
6
7     import java.util.List;
8
9     @Mapper
10    @Repository
11    public interface DeptDao {
12
13        public boolean addDept(Dept dept);
14
15        public Dept queryById(Long id);
16
17        public List<Dept> queryAll();
18    }

```

## 7. DeptService

```

1     package com.bean.service;
2
3     import com.bean.pojo.Dept;
4
5     import java.util.List;
6
7     public interface DeptService {
8
9        public boolean addDept(Dept dept);
10
11        public Dept queryById(Long id);
12
13        public List<Dept> queryAll();
14
15    }

```

## 8. DeptServiceImpl

```

1     package com.bean.service;
2
3     import com.bean.dao.DeptDao;
4     import com.bean.pojo.Dept;
5     import org.springframework.beans.factory.annotation.Autowired;
6     import org.springframework.stereotype.Service;
7
8     import java.util.List;

```



```

9
10 @Service
11 public class DeptServiceImpl implements DeptService {
12
13     @Autowired
14     private DeptDao deptDao;
15
16     @Override
17     public boolean addDept(Dept dept) {
18         return deptDao.addDept(dept);
19     }
20
21     @Override
22     public Dept queryById(Long id) {
23         return deptDao.queryById(id);
24     }
25
26     @Override
27     public List<Dept> queryAll() {
28         return deptDao.queryAll();
29     }
30 }

```

## 9. DeptController

```

1 package com.bean.controller;
2
3 import com.bean.pojo.Dept;
4 import com.bean.service.DeptService;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.PostMapping;
9 import org.springframework.web.bind.annotation.RestController;
10
11 import java.util.List;
12
13 //提供restful服务
14 @RestController
15 public class DeptController {
16
17     @Autowired
18     private DeptService deptService;
19
20     //添加
21     @PostMapping("/dept/add")
22     public boolean addDept(Dept dept){
23         return deptService.addDept(dept);
24     }
25
26     @GetMapping("/dept/get/{id}")
27     public Dept get(@PathVariable("id")Long id){
28         return deptService.queryById(id);
29     }
30
31     @GetMapping("/dept/list")
32     public List<Dept> queryAll() {
33         return deptService.queryAll();

```

```
34     }
35 }
```

#### 10. 新建一个主启动类来测试一下

```
1  package com.bean;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class DeptProvider_8001 {
8      public static void main(String[] args) {
9          SpringApplication.run(DeptProvider_8001.class, args);
10     }
11 }
```

#### 11. 测试完毕

```
▼ [
  ▼ {
    "deptno": 1,
    "dname": "开发部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 2,
    "dname": "人事部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 3,
    "dname": "财务部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 4,
    "dname": "市场部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 5,
    "dname": "运维部",
    "db_source": "db01"
  }
]
```

# 子项目消费者

对应消费者，一般消费者进去的都是默认端口，所以是80端口

1. 新建一个项目： `springcloud-consumer-dept-80`

2. `pom.xml`

```
1      <dependencies>
2          <!--实体类-->
3          <dependency>
4              <groupId>com.bean</groupId>
5              <artifactId>springcloud-api</artifactId>
6              <version>1.0-SNAPSHOT</version>
7          </dependency>
8          <!--web-->
9          <dependency>
10             <groupId>org.springframework.boot</groupId>
11             <artifactId>spring-boot-starter-web</artifactId>
12         </dependency>
13         <!--热部署-->
14         <dependency>
15             <groupId>org.springframework.boot</groupId>
16             <artifactId>spring-boot-devtools</artifactId>
17         </dependency>
18     </dependencies>
```

消费者显然不用链接数据库

3. `application.yaml`

```
1     server:
2         port: 80
```

4. `controller`

```
1     package com.bean.controller;
2
3     import com.bean.pojo.Dept;
4     import org.springframework.beans.factory.annotation.Autowired;
5     import org.springframework.stereotype.Controller;
6     import org.springframework.web.bind.annotation.PathVariable;
7     import org.springframework.web.bind.annotation.RequestMapping;
8     import org.springframework.web.bind.annotation.ResponseBody;
9     import org.springframework.web.client.RestTemplate;
10
11     import java.util.List;
12
13     @Controller
14     public class DeptConsumerController {
15
16         //消费者不应该存在service层，所以我们要想办法拿到service的东西
17         //使用Restful方式，用http进行通信
18         //RestTemplate，这个模版里面有很多方法供我们调用
```

```

19      //(url, 实体:Map, 返回值类型: Class<T> responseType)
20
21      @Autowired
22      private RestTemplate restTemplate;
23
24      //去这个地址拿到service
25      private static final String REST_URL_PREFIX="http://localhost:8001";
26
27      @RequestMapping("/consumer/dept/get/{id}")
28      @ResponseBody
29      public Dept get(@PathVariable("id")Long id){
30          //去这个地址拿到对应的值, 注意这里使用的是get方式
31          //还有postForObject, patchForObject, delete, ...
32          return
33      restTemplate.getForObject(REST_URL_PREFIX+"/dept/get"+id, Dept.class);
34      }
35
36      @RequestMapping("/consumer/dept/add")
37      @ResponseBody
38      public Dept add(Dept dept){
39          return
40      restTemplate.postForObject(REST_URL_PREFIX+"/dept/add", dept, Dept.class);
41      }
42
43      @RequestMapping("/consumer/dept/list")
44      @ResponseBody
45      public List<Dept> list(){
46          return restTemplate.getForObject(REST_URL_PREFIX+"/dept/list", List.class);
47      }

```

注意, 这里的RestTemplate是在config中的

## 5. config

```

1      package com.bean.config;
2
3      import org.springframework.context.annotation.Bean;
4      import org.springframework.context.annotation.Configuration;
5      import org.springframework.web.client.RestTemplate;
6
7      @Configuration
8      public class ConfigBean {
9
10         @Bean
11         public RestTemplate getRestTemplate(){
12             return new RestTemplate();
13         }
14
15     }

```

## 6. 写一个主启动类, 测试一下

```
1 package com.bean;
2
3
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6
7 @SpringBootApplication
8 public class DeptConsumer_80 {
9     public static void main(String[] args) {
10         SpringApplication.run(DeptConsumer_80.class, args);
11     }
12 }
```

注意了，这里要开启消费者和提供者两个服务

---

# Eureka

## 什么是Eureka

---

- 服务注册与发现
- Netflix再设计Eureka时，遵循的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

## 原理

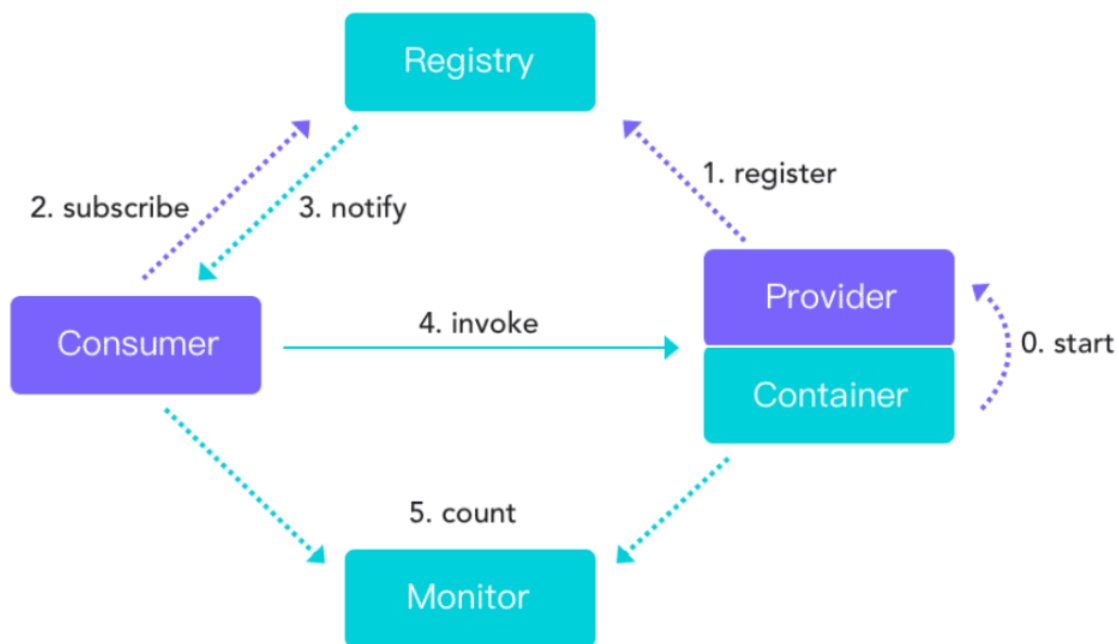
---

### 回顾Dubbo

首先我们来回顾一下Dubbo

# Dubbo Architecture

.....▶ init    .....▶ async    —▶ sync



## Eureka的基本架构

- SpringCloud封装了Netflix公司开发的Eureka模块来实现服务的注册与发现（对比zookeeper）
- Eureka采用了C/S架构的模式设计，EurekaServer作为服务注册功能的服务器，他是服务注册中心
- 系统中的其他微服务，使用Eureka的客户端连接到EurekaSever并维持心跳链接。这样系统的维护人员就可以通过EurekaServer来监控系统中各个微服务是否正常运行，并执行相关逻辑

## Eureka的两个组件

- Eureka Server：服务的注册。节点启动之后会在Eureka中进行注册。
- Eureka Client：一个JAVA客户端，用于简化EurekaSever的交互。有一个内置的，具有轮询负载算法的负载均衡器。应用启动后，回想EurekaServer发送心跳(默认周期30秒)。假如EurekaServer在多个周期内没有接受到某个节点的心跳，EurekaServer会从服务注册表中把这个服务节点移除（默认周期90秒）

## 三大角色

- Eureka Server：提供服务的注册与发现，代替Zookeeper的功能
- Service Provider：将自身服务注册到Eureka中，从而使消费者能够找到。环境搭建中的生产者。
- Service Consumer：服务消费方从Eureka中获取注册服务列表，从而找到消费服务。环境搭建中的消费者

## Eureka环境搭建

1. 新建一个子项目：springcloud-eureka-7001

2. pom.xml

```
1      <dependencies>
2          <!--Eureka Server-->
3          <dependency>
4              <groupId>org.springframework.cloud</groupId>
5              <artifactId>spring-cloud-starter-eureka-server</artifactId>
6              <version>1.4.6.RELEASE</version>
7          </dependency>
8          <!--热部署-->
9          <dependency>
10             <groupId>org.springframework.boot</groupId>
11             <artifactId>spring-boot-devtools</artifactId>
12         </dependency>
13     </dependencies>
```

这里导入的Eureka的包是服务端的

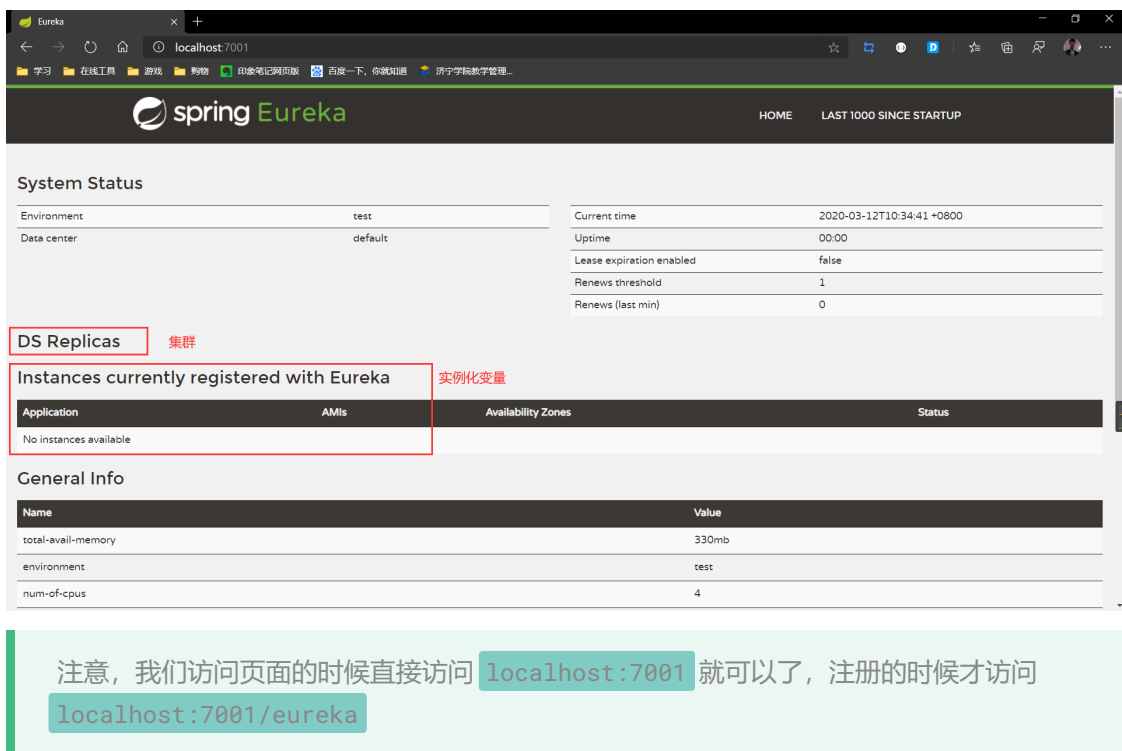
3. application.yaml

```
1     server:
2         port: 7001
3
4     # Eureka配置
5
6     eureka:
7         instance:
8             hostname: localhost # Eureka服务端的实例名字
9         client:
10             register-with-eureka: false # 表示是否向eureka注册中心注册自己，因为我们编写的本来就是
Eureka注册中心服务器，所以这个不用注册
11             fetch-registry: false # 假如为False，表示自己为注册中心
12             service-url: # 别人连他如何链接，监控页面，点击去发现默认是8761端口，但是我们要重写
13             defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #
http://localhost:7001/eureka/
```

4. 主启动类

```
1     package com.bean;
2
3     import org.springframework.boot.SpringApplication;
4     import org.springframework.boot.autoconfigure.SpringBootApplication;
5     import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7     @SpringBootApplication
8     @EnableEurekaServer //EurekaServer服务端主启动类
9     public class EurekaServer_7001 {
10         public static void main(String[] args) {
11             SpringApplication.run(EurekaServer_7001.class, args);
12         }
13     }
```

5. 启动测试



System Status

Environment	test	Current time	2020-03-12T10:34:41 +0800
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas 集群

Instances currently registered with Eureka 实例化变量

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	330mb
environment	test
num-of-cpus	4

注意，我们访问页面的时候直接访问 `localhost:7001` 就可以了，注册的时候才访问 `localhost:7001/eureka`

## Eureka提供者

## 使用Eureka启动注册

1. 给服务生产者 `springcloud-provider-8001` 加入 `Eureka` 依赖

```
1      <!--Eureka-->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-eureka</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
```

注意，我们刚才用的依赖是Eureka的服务端，这次要更改一下，作为生产者就不能导入服务端的包了

导入正常的Eureka包即可，也就是客户端的坐标

2. `application.yaml`

```
1      # 端口号
2      server:
3          port: 8001
4
5      # mybatis的配置，其中配置别名的时候虽然是上一个项目的包名，但是我们通过坐标拿到了这些
6      mybatis:
7          type-aliases-package: com.bean.pojo
8          mapper-locations: classpath:mybatis/mapper/*.xml
9          config-location: classpath:mybatis/mybatis-config.xml
10
11     # spring配置
```



```

12     spring:
13         application:
14             name: springcloud-provider-dept
15         datasource:
16             type: com.alibaba.druid.pool.DruidDataSource
17             driver-class-name: org.gjt.mm.mysql.Driver
18             url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
            8&serverTimezone=Asia/Shanghai
19             username: root
20             password: root
21
22     # Eureka生产者端的配置
23     eureka:
24         client:
25             service-url:
26                 defaultZone: http://localhost:7001/eureka/ # 注册中心地址
27         instance:
28             instance-id: springcloud-provider-dept8001 # 修改Eureka中默认描述信息

```

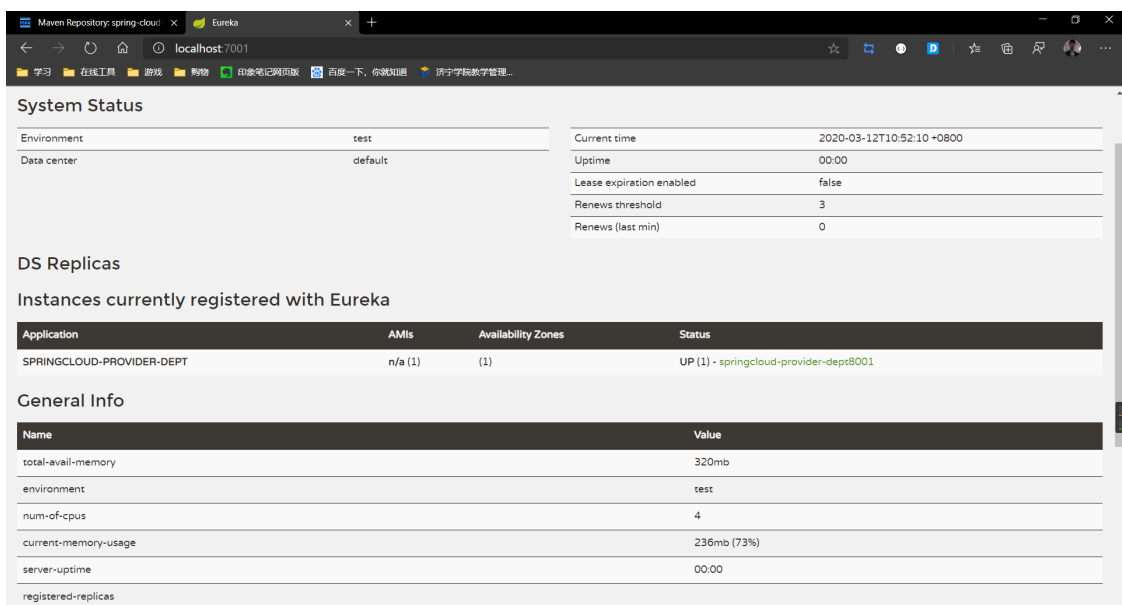
### 3. 主启动类

```

1     package com.bean;
2
3     import org.springframework.boot.SpringApplication;
4     import org.springframework.boot.autoconfigure.SpringBootApplication;
5     import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
6
7     @SpringBootApplication
8     @EnableEurekaClient//在服务启动后自动注册到Eureka中
9     public class DeptProvider_8001 {
10         public static void main(String[] args) {
11             SpringApplication.run(DeptProvider_8001.class,args);
12         }
13     }

```

### 4. 测试：启动7001和8001



The screenshot shows the Eureka web interface in a browser. The address bar indicates the URL is localhost:7001. The page displays the following information:

- System Status:**
  - Environment: test
  - Data center: default
  - Current time: 2020-03-12T10:52:10+0800
  - Uptime: 00:00
  - Lease expiration enabled: false
  - Renews threshold: 3
  - Renews (last min): 0
- DS Replicas:** (Empty table)
- Instances currently registered with Eureka:**

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (1)	(1)	UP (1) - springcloud-provider-dept8001
- General Info:**

Name	Value
total-avail-memory	320mb
environment	test
num-of-cpus	4
current-memory-usage	236mb (73%)
server-uptime	00:00
registered-replicas	

我们看到，服务已经注册进去了，其中

- Application栏中的名字：spring.application.name: springcloud-provider-dept
- Status：eureka.instance.instance-id: springcloud-provider-dept8001

## 添加监控信息

1. 还是服务生产者

2. pom.xml

```
1      <!-- 监控信息 -->
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-actuator</artifactId>
5      </dependency>
```

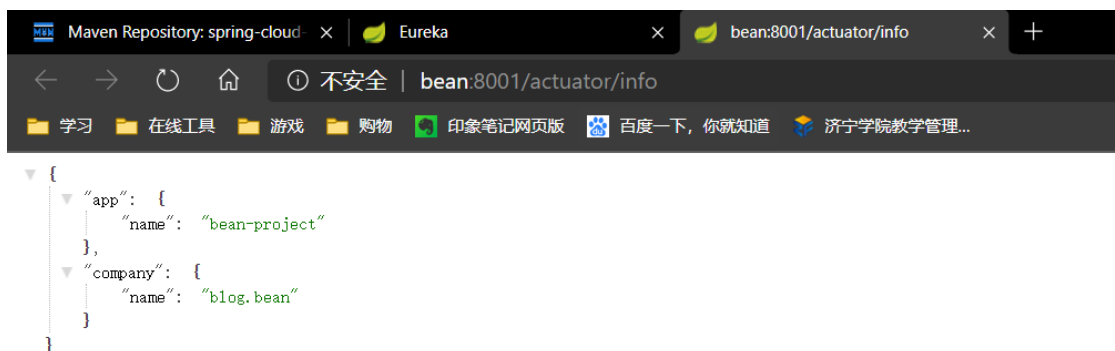
3. 配置监控信息

```
1      # 端口号
2      server:
3          port: 8001
4
5      # mybatis的配置，其中配置别名时虽然用的是上一个项目的包名，但是我们通过坐标拿到了这些
6      mybatis:
7          type-aliases-package: com.bean.pojo
8          mapper-locations: classpath:mybatis/mapper/*.xml
9          config-location: classpath:mybatis/mybatis-config.xml
10
11     # spring配置
12     spring:
13         application:
14             name: springcloud-provider-dept
15         datasource:
16             type: com.alibaba.druid.pool.DruidDataSource
17             driver-class-name: org.gjt.mm.mysql.Driver
18             url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
19                 8&serverTimezone=Asia/Shanghai
20             username: root
21             password: root
22
23     # Eureka生产者端的配置
24     eureka:
25         client:
26             service-url:
27                 defaultZone: http://localhost:7001/eureka/ # 注册中心地址
28         instance:
29             instance-id: springcloud-provider-dept8001 # 修改Eureka中默认描述信息
30
31     # 监控信息配置，可以监控我们整个服务
32     info:
33         app.name: bean-project # 项目的名字，随便起
34         company.name: blog.bean # 公司名字
```

4. 开启7001和8001

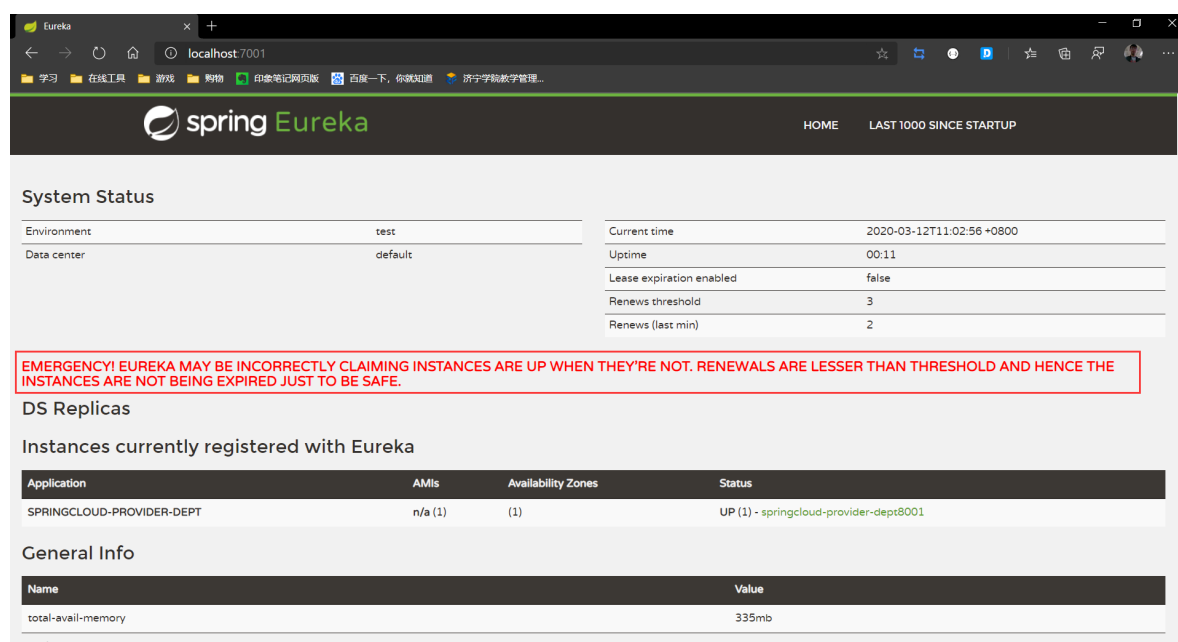
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (1)	(1)	UP (1) <a href="#">springcloud-provider-dept8001</a> 点击



## Eureka自我保护机制

有的时候可能爆红了：



这个意思是好死不如赖活着，有的时候一个微服务崩了，Eureka不会立刻把他清理，会把信息保存着，直到过了周期或者能够重新链接之后，等到很长时间才会关闭

关闭自我保护机制：（不建议关闭）

```
1 eureka:
2   server:
3     enable-self-preservation: false
```

注意，我们不建议关闭，关闭之后会立刻清理注册表

## 获取微服务的信息

1. 我们在8001新建一个controller

```
1 import org.springframework.cloud.client.discovery.DiscoveryClient;
```

```

2
3 @Autowired
4 private DiscoveryClient client;
5 //注册的进来的微服务，可以获取一些消息
6 @GetMapping("/dept/discovery")
7 public Object discovery(){
8     //获得微服务列表的清单
9     List<String> services = client.getServices();
10    System.out.println("service==>" + services);
11
12    //通过一个具体的微服务id获得这个微服务的信息
13    List<ServiceInstance> instances = client.getInstances("SPRINGCLOUD-PROVIDER-DEPT");
14    for (ServiceInstance instance : instances) {
15        System.out.println((instance.getHost() + instance.getPort() + instance.getUri() +
16        instance.getServiceId()));
17    }
18    return this.client;
19 }

```

SPRINGCLOUD-PROVIDER-DEPT:

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (1)	(1)	UP (1) - springcloud-provider-dept8001

## 2. 在主启动类中

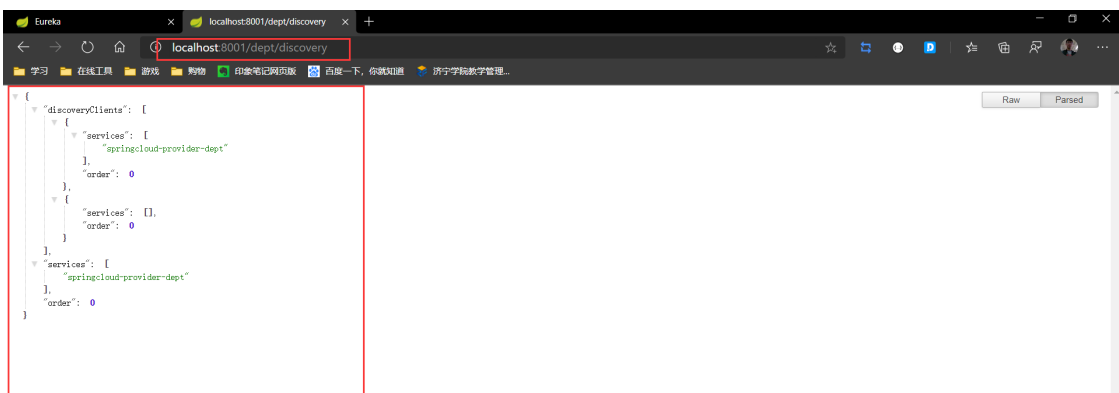
```

1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableDiscoveryClient //服务发现
4 public class DeptProvider_8001 {
5     public static void main(String[] args) {
6         SpringApplication.run(DeptProvider_8001.class, args);
7     }
8 }

```

@EnableDiscoveryClient使用这个信息

## 3. 测试



这个功能在团队开发中非常有用

## Eureka服务集群

集群可以保证：当一个机子崩了之后，另外的机子可以正常使用

### 开始之前

集群的配置不太麻烦，但是如果都用一个localhost在逻辑上有点难理解，下面我们来欺骗一下自己的大脑：

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

在这里面新添加：

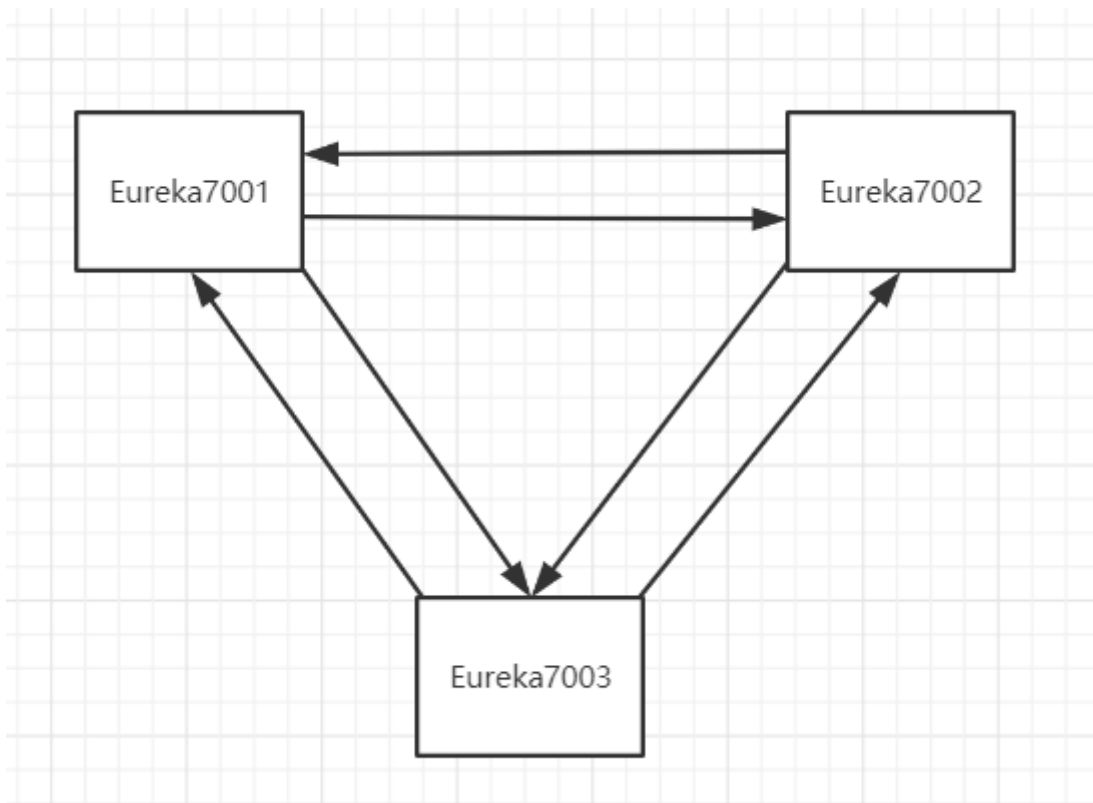
1	127.0.0.1	eureka7001.com
2	127.0.0.1	eureka7002.com
3	127.0.0.1	eureka7003.com

```
127.0.0.1      localhost
127.0.0.1      eureka7001.com
127.0.0.1      eureka7002.com
127.0.0.1      eureka7003.com
```

这个意思是：localhost我们可以使用 `eureka7001.com`，`eureka7002.com`，`eureka7003.com` 来代替了

这样做的目的是欺骗一下自己的大脑，模拟真实开发环境（在不同主机上）

Eureka服务集群的配置就是让一个主机链接另外几台主机，实现共同成为注册中心，防止一台损坏其他崩溃



- 7001 注册 7002, 7003
- 7002 注册 7001, 7003
- 7003 注册 7001, 7002

下面我们来进行注册中心集群的配置：

在之前，我们进行Eureka环境搭建的时候就已经配置了一个端口为7001的注册中心，下面放一下配置：

```
1      <dependencies>
2          <!--Eureka Server-->
3          <dependency>
4              <groupId>org.springframework.cloud</groupId>
5              <artifactId>spring-cloud-starter-eureka-server</artifactId>
6              <version>1.4.6.RELEASE</version>
7          </dependency>
8          <!--热部署-->
9          <dependency>
10             <groupId>org.springframework.boot</groupId>
11             <artifactId>spring-boot-devtools</artifactId>
12         </dependency>
13     </dependencies>
```

注意这里导入的Eureka坐标是服务端的坐标，不是客户端的坐标

```

1  server:
2      port: 7001
3
4  # Eureka配置
5
6  eureka:
7      instance:
8          hostname: localhost # Eureka服务端的实例名字
9      client:
10         register-with-eureka: false # 表示是否向eureka注册中心注册自己，因为我们编写的本来就是Eureka服务
            器，所以这个不用注册
11         fetch-registry: false # 假如为False，表示自己为注册中心
12         service-url: # 别人连他如何链接，监控页面，点击去发现默认是8761端口，但是我们要重写
13         defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #
            http://localhost:7001/eureka/

```

下面我们按照这个配置，新增两个新的模块 `springcloud-eureka-7002`，`springcloud-eureka-7003` 端口分别为 **7002** 和 **7003**：

然后将7001的pom文件中的依赖复制到7002和7003

然后修改配置文件

1. 将配置文件复制到7002和7003（别忘记改变端口）
2. 以7001举例，7001、7002、7003的hostname和注册中心都要改

```

1  server:
2      port: 7001
3
4  # Eureka配置
5
6  eureka:
7      instance:
8          hostname: eureka7001.com # Eureka服务端的实例名字，这里改变了
9      client:
10         register-with-eureka: false
11         fetch-registry: false
12         service-url:
13         defaultZone:
            http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
14         # 这里注册另外的两个注册中心，这里一定要注意空格别多写

```

3. 修改8001服务提供者的配置文件

```

1
2  server:
3      port: 8001
4
5  mybatis:
6      type-aliases-package: com.bean.pojo
7      mapper-locations: classpath:mybatis/mapper/*.xml
8      config-location: classpath:mybatis/mybatis-config.xml
9
10 spring:

```

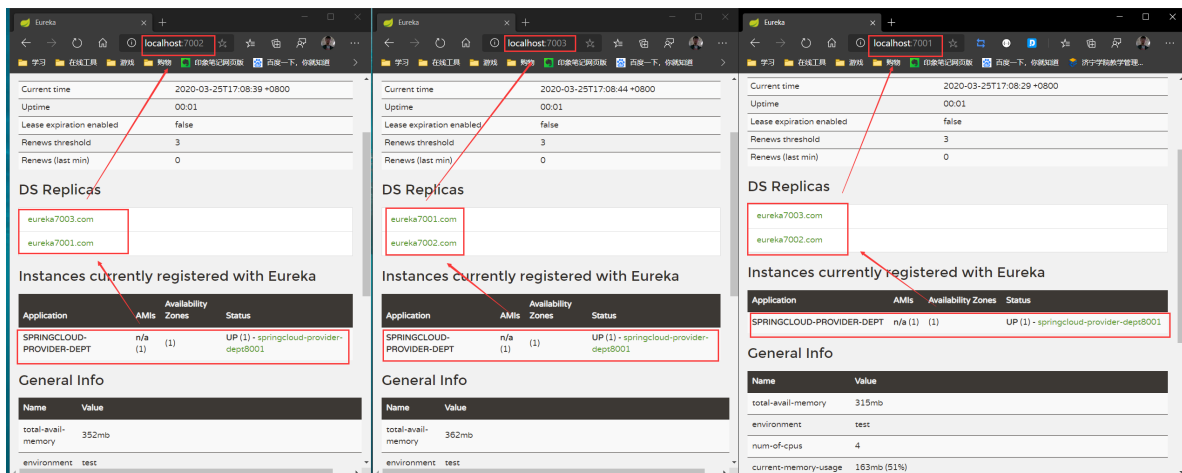
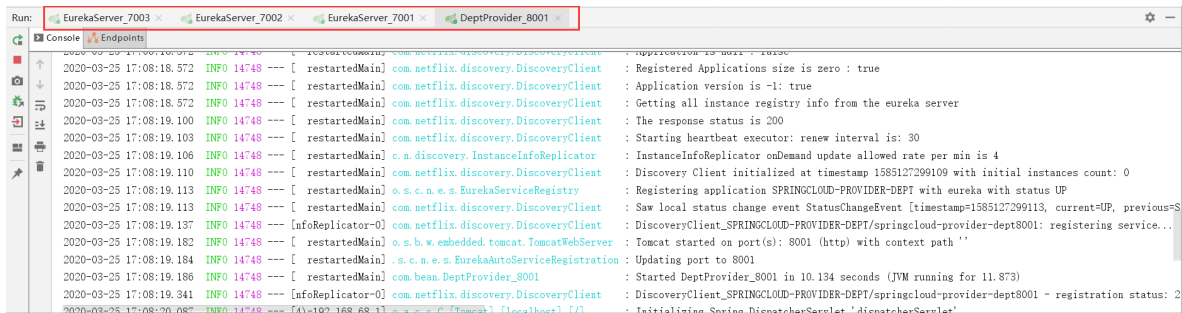
```

11     application:
12         name: springcloud-provider-dept
13     datasource:
14         type: com.alibaba.druid.pool.DruidDataSource
15         driver-class-name: org.gjt.mm.mysql.Driver
16         url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
            8&serverTimezone=Asia/Shanghai
17         username: root
18         password: root
19
20     # Eureka生产者端的配置
21     eureka:
22         client:
23             service-url:
24                 defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7
003.com:7003/eureka/
25
26         # 注册中心地址把7001,7002,7003都加上
27     instance:
28         instance-id: springcloud-provider-dept8001
29
30     info:
31         app.name: bean-project
32         company.name: blog.bean

```

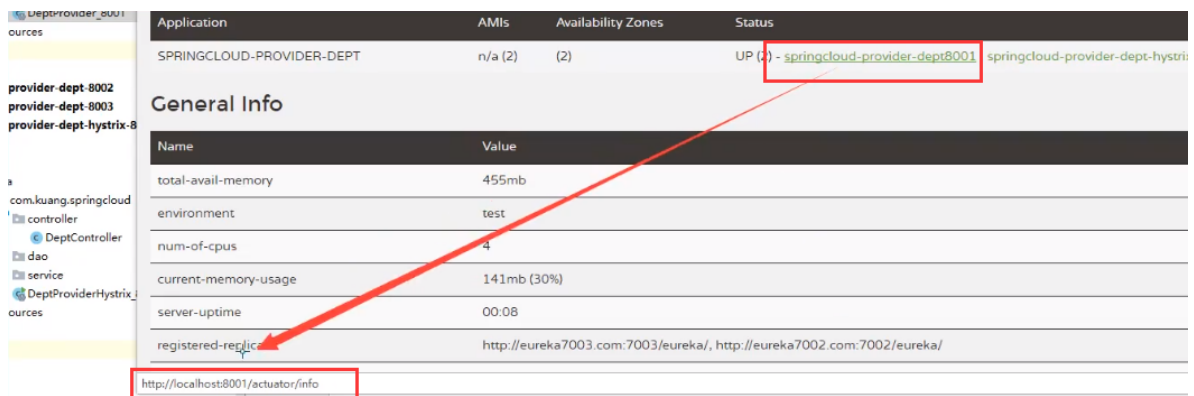
然后我们将主启动类在7002和7003写一份

启动7001,7002,7003,8001，查看服务集群和服务是否注册上





# 显示IP

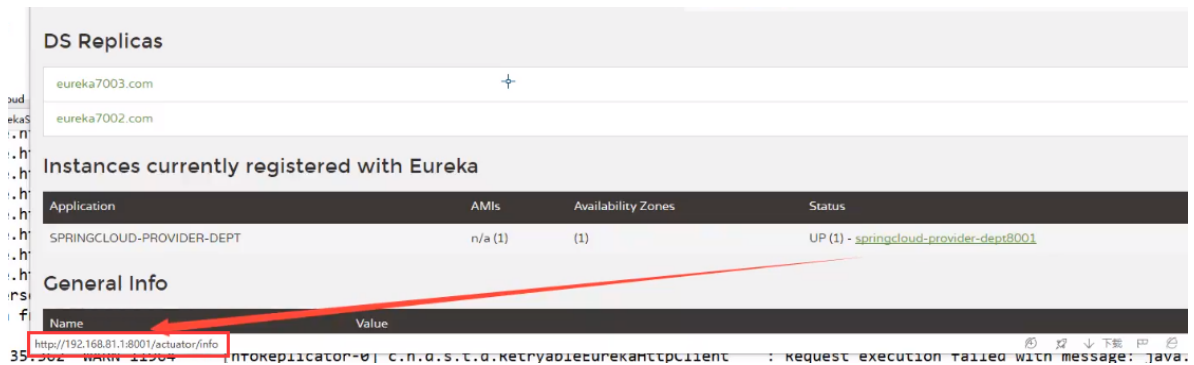


Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (2)	(2)	UP (2) - <a href="#">springcloud-provider-dept8001</a> <a href="#">springcloud-provider-dept-hystrix</a>

Name	Value
total-avail-memory	455mb
environment	test
num-of-cpus	4
current-memory-usage	141mb (30%)
server-up-time	00:08
registered-replicas	<a href="http://eureka7003.com:7003/eureka/">http://eureka7003.com:7003/eureka/</a> , <a href="http://eureka7002.com:7002/eureka/">http://eureka7002.com:7002/eureka/</a>

```
1 # Eureka配置
2 eureka:
3   instance:
4     prefer-ip-address: true # 显示ip
```



DS Replicas

- [eureka7003.com](#)
- [eureka7002.com](#)

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (1)	(1)	UP (1) - <a href="#">springcloud-provider-dept8001</a>

Name	Value
http://192.168.81.1:8001/actuator/info	

## CAP原则及对比Zookeeper

### 回顾CAP原则

关系型数据库 --> ACID

非关系型数据库 --> CAP

### ACID

- A: 原子性
- C: 一致性
- I: 隔离性
- D: 持久性

### CAP

- C: 强一致性
- A: 可用性
- P: 分区容错性

## CAP原则

指在一个分布式系统中，CAP三个要素中只能同时存在两点，而不可能三点同时存在

- CA：单点集群，满足一致性，可用性的系统，通常可扩展性较差
- CP：满足一致性，分区容错性的系统，通常性能不是很高
- AP：满足可用性，分区容错性的系统，通常对一致性要求低一些

---

**Zookeeper中保证的是CP，Eureka保证的是AP**

### Zookeeper中的CP

当向注册中心查询服务列表的时候，注册中心有两种情况：

- 直接挂掉
- 返回几分钟之前的注册信息

很明显，挂掉是我们更加不能容忍的，也就是说，服务注册功能可用性的要求要高于一致性。

但是Zookeeper会出现这样一种情况：当主节点挂掉之后，其余节点不能够立刻顶上，而是要先选举一个新的主节点，然后顶上。

在整个选取期间Zookeeper集群都是不可用的。

### Eureka的AP

Eureka的AP原则首先保证了可用性。Eureka各个节点之间都是平等的，几个节点挂掉不会影响整体的工作，只不过查询到的信息可能不是最新的。

另外Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有心跳，那么Eureka就认为客户端与注册中心之间出现了故障，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时间过长没有收到心跳而应该过期的服务
2. Eureka仍然可以接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点仍然可用）
3. 当网络稳定时，当前实例新的注册信息会被同步到其他节点中

---

# 负载均衡和Ribbon

## Ribbon是什么

- Spring Cloud Ribbon是基于Netflix Ribbon实现的一套 **客户端负载均衡的工具**
- 简单的来说，Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层服务连接到一起。
- Ribbon的客户端组件提供一系列完整的配置项，如
  - 连接超时
  - 重试

简单的来说，就是在配置文件中列出LoadBalancer（简称LB：负载均衡）后面的所有机器，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

简单轮询的意思是：用户第一次请求服务，那么就去第一个服务端，第二次去第二个，...，第n次去第n个

简单轮询这个算法是一种最低级的算法

随机连接这个算法类似HashMap的算法

## Ribbon能干嘛

- LB，即负载均衡（Load Balance），在微服务或则好分布式集群中经常用到的一种应用
- 负载均衡简单地来说就是将用户的请求平摊地分派到多个服务上，从而达到系统的HA（高可用）
- 常见的负载均衡软件：Nginx，Lvs(中国人开发)，Apache+Tomcat等
- Dubbo和SpringCloud中都给我们提供了负载均衡，SpringCloud的负载均衡算法可以自定义
- 负载均衡简单分类
  - 集中式LB
    - 即在服务的消费方和提供方之间使用独立的LB设施，比如Nginx，由该设施负责把访问请求通过某种策略转发到服务的提供方
  - 进程式LB
    - 将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选出一个合适的服务器
    - **Ribbon属于进程内LB**，它只是一个类库，集成与消费方进程，消费方通过它来获取到服务提供方的地址

## 环境搭建和消费者配置

从上面的介绍我们知道，Ribbon是消费方的进程，所以我们在 `springcloud-consumer-dept-80` 消费者中操作

### 1. 首先导入pom依赖

```
1      <!-- Ribbon -->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-ribbon</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
```

我们取搜索Ribbon Maven的时候要注意，出来了很多，注意别导入Netflix的，要导入SpringCloud的

5. **Ribbon Eureka**  
`com.netflix.ribbon » ribbon-eureka`  
ribbon-eureka  
Last Release on May 29, 2019

6. **Spring Cloud Starter Netflix Ribbon**  
`org.springframework.cloud » spring-cloud-starter-netflix-ribbon`  
Spring Cloud Starter Netflix Ribbon  
Last Release on Mar 5, 2020

7. **Spring Cloud Starter Ribbon**  
`org.springframework.cloud » spring-cloud-starter-ribbon`  
Spring Cloud Starter Ribbon (deprecated, please use spring-cloud-starter-netflix-ribbon)  
Last Release on May 23, 2019

Eureka配置的也是客户端的依赖，也就是8001的那个依赖

## 2. 配置 `application.yaml`

```
1  server:
2    port: 80
3
4  #Eureka配置
5  eureka:
6    client:
7      register-with-eureka: false # 我们不向注册中心注册自己，因为是消费者不是提供者
8      service-url:
9        defaultZone:
10         http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

## 3. 在主启动类上加上Eureka的注解

```
1  @SpringBootApplication
2  @EnableEurekaClient //注解
3  public class DeptConsumer_80 {
4    public static void main(String[] args) {
5      SpringApplication.run(DeptConsumer_80.class, args);
6    }
7  }
```

## 4. 我们之前在 `com.bean.config` 中编写了ConfigBean，下面再次改造这个类，在方法上实现负载均衡

```
1  @Configuration
2  public class ConfigBean {
3
4    @Bean
5    @LoadBalanced //Ribbon, 负载均衡实现
6    public RestTemplate getRestTemplate(){
7      return new RestTemplate();
8    }
9  }
```

因为我们的Controller都是使用的RestHttp的方式进行通信，而使用Rest就要使用RestTemplate，那么我们直接在根源上实现负载均衡

## 5. 改造Controller

```
1  @Controller
2  public class DeptConsumerController {
3
4      @Autowired
5      private RestTemplate restTemplate;
6
7      /*
8          private static final String REST_URL_PREFIX="http://localhost:8001";
9          去这个地址拿到service，但是我们使用Ribbon的时候这个地址应该是一个变量而不应该每次都从这
          个地址拿取
10         所以我们应该从配置文件中的三种选择中获取：
11         http://eureka7001.com:7001/eureka/
12         http://eureka7002.com:7002/eureka/
13         http://eureka7003.com:7003/eureka/
14         那么这个变量是什么呢？这个变量就是服务名字，也就是之前在8001服务提供者配置的
          spring.application.name: springcloud-provider-dept
15         */
16         private static final String REST_URL_PREFIX="http://SPRINGCLOUD-PROVIDER-DEPT";
17
18         @RequestMapping("/consumer/dept/list")
19         @ResponseBody
20         public List<Dept> list(){
21             return restTemplate.getForObject(REST_URL_PREFIX+"/dept/list",List.class);
22         }
23
24
25     }
```

6. 启动7001, 7002, 7003, 8001, 80

7. 访问 <http://localhost/consumer/dept/list>

```
▼ [
  ▼ {
    "deptno": 1,
    "dname": "开发部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 2,
    "dname": "人事部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 3,
    "dname": "财务部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 4,
    "dname": "市场部",
    "db_source": "db01"
  },
  ▼ {
    "deptno": 5,
    "dname": "运维部",
    "db_source": "db01"
  }
]
```

成功

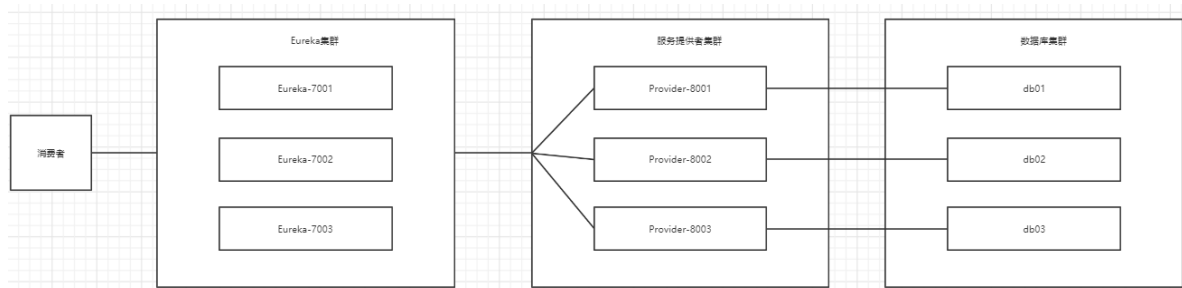
结论：

Eureka和Ribbon结合之后，客户端可以不用关心地址了，因为都是从Application中拿取

## Ribbon实现负载均衡

在这之前，其实已经实现了负载均衡，也就是上一步在消费者上我们在代码上配置的，但是结果很不明显，为了让结果清晰明了，我们要扩增几个服务提供者和对应的数据库

我们考虑，要另外新增两个消费者意思一下，那么我们的思想是这样的：



## 首先是数据库

我们以db01作为参考，来新建另外的两个数据库db02和db03

下面放上db01中dept的表：

```
1  create table dept
2  (
3      deptno bigint auto_increment,
4      dname varchar(60) null,
5      db_source varchar(60) null,
6      constraint dept_pk
7          primary key (deptno)
8  )
9  comment '部门表';
```

```
1  # DATABASES()会读取当前数据库的名字
2  insert into dept (dname, db_source) values ('开发部', DATABASE());
3  insert into dept (dname, db_source) values ('人事部', DATABASE());
4  insert into dept (dname, db_source) values ('财务部', DATABASE());
5  insert into dept (dname, db_source) values ('市场部', DATABASE());
6  insert into dept (dname, db_source) values ('运维部', DATABASE());
```

以这张表为基础，另外新建两个数据库，数据库中同样建立这张表，表的字段要求一模一样，这样才可以实现负载均衡

```
1
2  CREATE DATABASE db02;
3
4  USE db02;
5
6  CREATE TABLE dept
7  (
8      deptno BIGINT AUTO_INCREMENT,
9      dname VARCHAR(60) NULL,
10     db_source VARCHAR(60) NULL,
11     CONSTRAINT dept_pk
12         PRIMARY KEY (deptno)
13 )
14 COMMENT '部门表';
15
16 # DATABASES()会读取当前数据库的名字
17 INSERT INTO dept (dname, db_source) VALUES ('开发部', DATABASE());
18 INSERT INTO dept (dname, db_source) VALUES ('人事部', DATABASE());
19 INSERT INTO dept (dname, db_source) VALUES ('财务部', DATABASE());
20 INSERT INTO dept (dname, db_source) VALUES ('市场部', DATABASE());
21 INSERT INTO dept (dname, db_source) VALUES ('运维部', DATABASE());
```

```

22
23
24 CREATE DATABASE db03;
25
26 USE db03;
27
28 CREATE TABLE dept
29 (
30     deptno BIGINT AUTO_INCREMENT,
31     dname VARCHAR(60) NULL,
32     db_source VARCHAR(60) NULL,
33     CONSTRAINT dept_pk
34         PRIMARY KEY (deptno)
35 )
36 COMMENT '部门表';
37
38 # DATABASES() 会读取当前数据库的名字
39 INSERT INTO dept (dname, db_source) VALUES ('开发部', DATABASE());
40 INSERT INTO dept (dname, db_source) VALUES ('人事部', DATABASE());
41 INSERT INTO dept (dname, db_source) VALUES ('财务部', DATABASE());
42 INSERT INTO dept (dname, db_source) VALUES ('市场部', DATABASE());
43 INSERT INTO dept (dname, db_source) VALUES ('运维部', DATABASE());

```

## 下面是新增服务提供者

我们要新增服务提供者8002和8003，下面放一张8001的配置作为参考：

- pom.xml

```

1     <dependencies>
2         <!--我们需要拿到实体类，所以需要 api这个模块-->
3         <dependency>
4             <groupId>com.bean</groupId>
5             <artifactId>springcloud-api</artifactId>
6             <version>1.0-SNAPSHOT</version>
7         </dependency>
8
9         <!--junit-->
10        <dependency>
11            <groupId>junit</groupId>
12            <artifactId>junit</artifactId>
13        </dependency>
14
15        <!--mysql-->
16        <dependency>
17            <groupId>mysql</groupId>
18            <artifactId>mysql-connector-java</artifactId>
19        </dependency>
20        <!--数据源-->
21        <dependency>
22            <groupId>com.alibaba</groupId>
23            <artifactId>druid</artifactId>
24        </dependency>
25        <!--logback, 日志-->
26        <dependency>

```



```

27         <groupId>ch.qos.logback</groupId>
28         <artifactId>logback-core</artifactId>
29     </dependency>
30     <!--mybatis-springboot启动器-->
31     <dependency>
32         <groupId>org.mybatis.spring.boot</groupId>
33         <artifactId>mybatis-spring-boot-starter</artifactId>
34     </dependency>
35     <!--test-->
36     <dependency>
37         <groupId>org.springframework.boot</groupId>
38         <artifactId>spring-boot-test</artifactId>
39     </dependency>
40     <!--web-->
41     <dependency>
42         <groupId>org.springframework.boot</groupId>
43         <artifactId>spring-boot-starter-web</artifactId>
44     </dependency>
45     <!--jetty, 类似tomcat-->
46     <dependency>
47         <groupId>org.springframework.boot</groupId>
48         <artifactId>spring-boot-starter-jetty</artifactId>
49     </dependency>
50     <!--热部署-->
51     <dependency>
52         <groupId>org.springframework.boot</groupId>
53         <artifactId>spring-boot-devtools</artifactId>
54     </dependency>
55     <!--Eureka-->
56     <dependency>
57         <groupId>org.springframework.cloud</groupId>
58         <artifactId>spring-cloud-starter-eureka</artifactId>
59         <version>1.4.6.RELEASE</version>
60     </dependency>
61     <!--监控信息-->
62     <dependency>
63         <groupId>org.springframework.boot</groupId>
64         <artifactId>spring-boot-starter-actuator</artifactId>
65     </dependency>
66 </dependencies>

```

- application.yaml

```

1
2 server:
3     port: 8001
4
5 mybatis:
6     type-aliases-package: com.bean.pojo
7     mapper-locations: classpath:mybatis/mapper/*.xml
8     config-location: classpath:mybatis/mybatis-config.xml
9
10 spring:
11     application:
12         name: springcloud-provider-dept
13     datasource:
14         type: com.alibaba.druid.pool.DruidDataSource
15         driver-class-name: org.gjt.mm.mysql.Driver

```

```

16     url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
      8&serverTimezone=Asia/Shanghai
17     username: root
18     password: root
19
20     # Eureka生产者端的配置
21     eureka:
22       client:
23         service-url:
24           defaultZone:
25             http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka700
26             3.com:7003/eureka/
27           # 注册中心地址把7001,7002,7003都加上
28
29     instance:
30       instance-id: springcloud-provider-dept8001
31
32     info:
33       app.name: bean-project
34       company.name: blog.bean

```

- resources/mybatis/mybatis-config.xml

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
4      config.dtd">
5
6  <configuration>
7      <settings>
8          <!--开启二级缓存-->
9          <setting name="cacheEnabled" value="true" />
10     </settings>
11 </configuration>

```

- resources/mybatis/mapper/DeptMapper.xml

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
4      mapper.dtd">
5
6  <mapper namespace="com.bean.dao.DeptDao">
7
8      <insert id="addDept" parameterType="Dept">
9          insert into dept (dname, db_source)
10         values (#{dname},DATABASE())
11     </insert>
12
13     <select id="queryById" resultType="Dept" parameterType="Long">
14         select * from dept where deptno = #{deptno};
15     </select>
16
17     <select id="queryAll" resultType="Dept">
18         select * from dept;
19     </select>
20 </mapper>

```

- `com.bean.controller.DeptController`

```
1  @RestController
2  public class DeptConrtoller {
3
4      @Autowired
5      private DeptService deptService;
6
7      //添加
8      @PostMapping("/dept/add")
9      public boolean addDept(Depth dept){
10         return deptService.addDept(dept);
11     }
12
13     @GetMapping("/dept/get/{id}")
14     public Dept get(@PathVariable("id")Long id){
15         return deptService.queryById(id);
16     }
17
18     @GetMapping("/dept/list")
19     public List<Dept> queryAll() {
20         return deptService.queryAll();
21     }
22
23     @Autowired
24     private DiscoveryClient client;
25
26     //注册的进来的微服务，可以获取一些消息
27     @GetMapping("/dept/discovery")
28     public Object discovery(){
29         //获得微服务列表的清单
30         List<String> services = client.getServices();
31         System.out.println("service==>" + services);
32
33         //通过一个具体的微服务id获得这个微服务的信息
34         List<ServiceInstance> instances = client.getInstances("SPRINGCLOUD-PROVIDER-DEPT");
35         for (ServiceInstance instance : instances) {
36             System.out.println((instance.getHost() + instance.getPort() +
37 instance.getUri() + instance.getServiceId()));
38         }
39         return this.client;
40     }
```

- `com.bean.dao.DeptDao`

```

1  @Mapper
2  @Repository
3  public interface DeptDao {
4
5      boolean addDept(Dept dept);
6
7      Dept queryById(Long id);
8
9      List<Dept> queryAll();
10
11 }

```

- `com.bean.service.DeptService`

```

1  public interface DeptService {
2      public boolean addDept(Dept dept);
3
4      public Dept queryById(Long id);
5
6      public List<Dept> queryAll();
7  }
8

```

- `com.bean.service.DeptServiceImpl`

```

1  @Service
2  public class DeptServiceImpl implements DeptService {
3      @Autowired
4      private DeptDao deptDao;
5
6      @Override
7      public boolean addDept(Dept dept) {
8          return deptDao.addDept(dept);
9      }
10
11     @Override
12     public Dept queryById(Long id) {
13         return deptDao.queryById(id);
14     }
15
16     @Override
17     public List<Dept> queryAll() {
18         return deptDao.queryAll();
19     }
20 }

```

有几个要点注意一下：

1. `application.yaml` 中的端口号和描述信息要修改，但是应用名称不应该修改，数据库也需要修改为对应的数据库，下面放一张配置作为对比

```

1
2  server:
3      port: 8002 # 端口号需要修改一下
4

```

```

5   mybatis:
6       type-aliases-package: com.bean.pojo
7       mapper-locations: classpath:mybatis/mapper/*.xml
8       config-location: classpath:mybatis/mybatis-config.xml
9
10  spring:
11      application:
12          name: springcloud-provider-dept # 注意，服务的名称必须要一个，因为我们在Controller中获取的
                                           是这里的名称
13      datasource:
14          type: com.alibaba.druid.pool.DruidDataSource
15          driver-class-name: org.gjt.mm.mysql.Driver
16          url: jdbc:mysql://localhost:3306/db02?useUnicode=true&characterEncoding=utf-
17              8&serverTimezone=Asia/Shanghai # 数据库也需要修改一下
18          username: root
19          password: root
20
21  eureka:
22      client:
23          service-url:
24              defaultZone:
25                  http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka700
26                      3.com:7003/eureka/
27      instance:
28          instance-id: springcloud-provider-dept8002 #描述信息应该修改一下
29
30  info:
31      app.name: bean-project
32      company.name: blog.bean

```

## 2. 启动类名称修改，放一张配置

```

1   @SpringBootApplication
2   @EnableEurekaClient
3   @EnableDiscoveryClient
4   public class DeptProvider_8003 {
5
6       public static void main(String[] args) {
7           SpringApplication.run(DeptProvider_8003.class, args);
8       }
9   }
10

```

考虑到电脑性能的问题，我们这样启动：

启动 7001注册中心

启动8001,8002,8003服务提供者

启动80消费者

最终结果：

```

▼ [
  ▼ {
    "deptno": 1,
    "dname": "开发部",
    "db_source": "db03"
  },
  ▼ {
    "deptno": 2,
    "dname": "人事部",
    "db_source": "db03"
  },
  ▼ {
    "deptno": 3,
    "dname": "财务部",
    "db_source": "db03"
  },
  ▼ {
    "deptno": 4,
    "dname": "市场部",
    "db_source": "db03"
  },
  ▼ {
    "deptno": 5,
    "dname": "运维部",
    "db_source": "db03"
  }
]

```

这里放的是负载到8003的时刻，其实Ribbon是使用1,3,2这样的轮询来进行负载均衡的

Ribbon最基础的算法就是轮询

## Ribbon的负载均衡算法自定义

### 前期准备

首先我们要了解 `IRule` 是一切的源头，这就是负载均衡算法的接口，它有一些实现类

- `RoundRobinRule`：默认的轮询算法
- `RandomRule`：随机
- `AvailabilityFilteringRule`：先过滤掉跳闸和访问故障的服务，然后从剩下活着的服务里进行轮询
- `RetryRule`：先进行轮询，假如服务获取失败会在指定的时间内进行重试

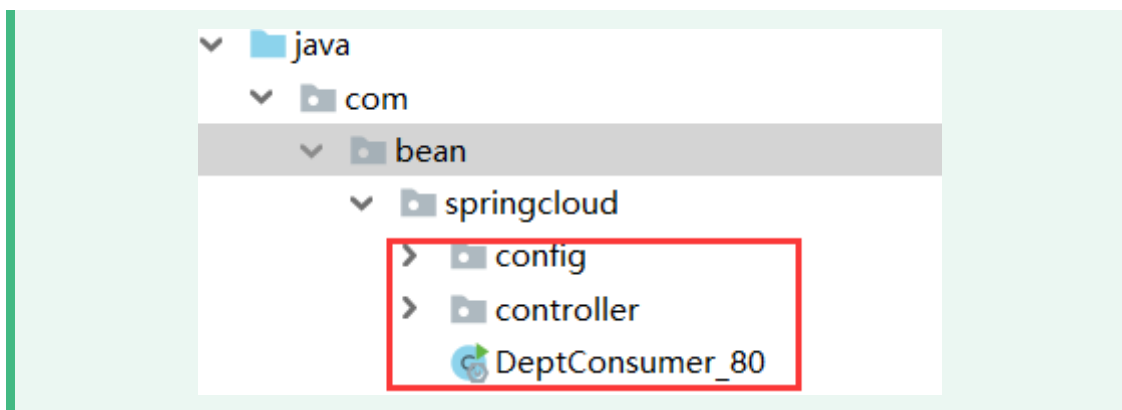
然后我们要知道，这些配置都是放到Ribbon下面来做的，也就是消费者Consumer-80上做的

所谓自定义算法，就是把别人的算法拿来改一下，就成了自己的算法。比如我们要根据 `RandomRule` 进行修改

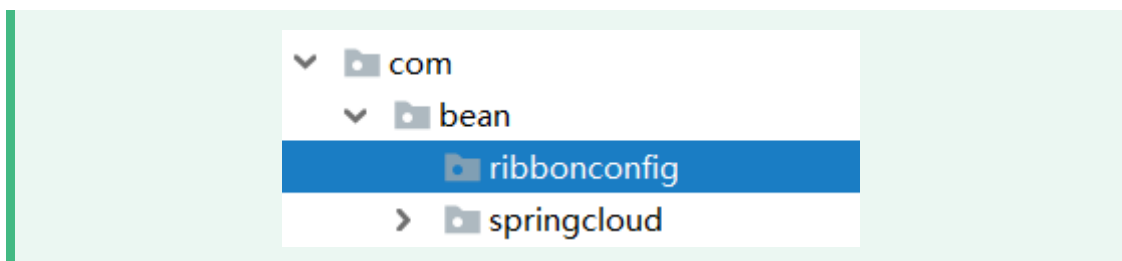
我们从官网上查看自定义Ribbon算法的时候，会发现他说对应的实现类应该放到主启动类包的外面  
虽然不知道为什么，但是我们还是要尊重官网，免得出现什么不必要的错误

## 算法的分析

1. 新建一个springcloud的包，将刚才的类都放到这个包下



2. 新建一个包 `com.bean.ribbonconfig`，我们将在这里进行Ribbon算法的配置



3. 将 `RandomRule` 复制一份到 `ribbonconfig` 包下，然后把没用的去了，下面放一个精简版的：

```
1 package com.bean.ribbonconfig;
2
3 import com.netflix.client.config.IClientConfig;
4 import com.netflix.loadbalancer.AbstractLoadBalancerRule;
5 import com.netflix.loadbalancer.ILoadBalancer;
6 import com.netflix.loadbalancer.Server;
7
8 import java.util.List;
9 import java.util.concurrent.ThreadLocalRandom;
10
11 public class RandomRule extends AbstractLoadBalancerRule {
12
13
14     /*这个代码这么多，一看就是实现算法的地方*/
15     public Server choose(ILoadBalancer lb, Object key) {
16         if (lb == null) { //假如没有负载均衡，就返回null
17             return null;
18         }
19     }
20 }
```

```

19         Server server = null;
20
21         while (server == null) {           //当没有服务的时候
22             if (Thread.interrupted()) {
23                 return null;
24             }
25             List<Server> upList = lb.getReachableServers(); //获取还活着的服务
26             List<Server> allList = lb.getAllServers(); //获取所有的服务
27
28
29             /*-----*/
30             /*我们知道，这下面就是具体的算法，上边的都是前期的准备工作*/
31
32             int serverCount = allList.size();
33             if (serverCount == 0) {
34                 return null;
35             }
36
37             int index = chooseRandomInt(serverCount); //这个是索引，用于指向当前用哪个
服务
38             server = upList.get(index); //让服务 = 从活着的服务里面获取新的服务
39
40
41             /*-----*/
42             /*上面的就是具体的算法，下面就是一些收尾*/
43
44             if (server == null) { //假如服务没有获取到，还为null
45                 Thread.yield(); //进行处理，咋处理的也看不懂....
46                 continue;
47             }
48
49             if (server.isAlive()) { //假如这个服务还活着
50                 return (server); //返回
51             }
52
53             server = null; //服务置空
54             Thread.yield(); //处理
55         }
56
57         return server;
58     }
59
60
61
62     /*下面这三个方法代码这么少，一看就不是实现算法的地方*/
63
64
65     protected int chooseRandomInt(int serverCount) {
66         return ThreadLocalRandom.current().nextInt(serverCount);
67     }
68
69     @Override
70     public Server choose(Object key) {
71         return choose(getLoadBalancer(), key);
72     }
73
74     @Override

```



```

75     public void initWithNiwsConfig(IClientConfig clientConfig) { //这个方法都是空的，说
      明没啥用
76
77     }
78 }

```

对代码分析完之后，我们知道我们要修改的也就是中间的那几行，其余的没什么要改的

## 算法改造

来实现这样一个需求：

- A服务提供五次服务之后，假如服务B活着，跳转服务B
- B服务提供五次服务之后，假如服务C活着，跳转服务C
- C服务提供五次服务之后，假如服务A活着，跳转服务A

1. 第一步就是要把类名改了，改成我们自己的类名，比如 `MyRule`
2. 改造算法

```

1  package com.bean.ribbonconfig;
2
3  import com.netflix.client.config.IClientConfig;
4  import com.netflix.loadbalancer.AbstractLoadBalancerRule;
5  import com.netflix.loadbalancer.ILoadBalancer;
6  import com.netflix.loadbalancer.Server;
7
8  import java.util.List;
9  import java.util.concurrent.ThreadLocalRandom;
10
11  public class MyRule extends AbstractLoadBalancerRule {
12
13
14      /*
15          需求：
16          - A服务提供五次服务之后，假如服务B活着，跳转服务B
17          - B服务提供五次服务之后，假如服务C活着，跳转服务C
18          - C服务提供五次服务之后，假如服务A活着，跳转服务A
19      */
20
21      private int total = 0; //我们设置一下当前服务被调用的次数
22      private int currentIndex = 0; //设置一下现在在哪个服务
23
24      /*这个代码这么多，一看就是实现算法的地方*/
25      public Server choose(ILoadBalancer lb, Object key) {
26          if (lb == null) { //假如没有负载均衡，就返回null
27              return null;
28          }
29          Server server = null;
30
31          while (server == null) { //当没有服务的时候
32              if (Thread.interrupted()) {
33                  return null;
34              }
35              List<Server> upList = lb.getReachableServers(); //获取还活着的服务

```

```

36         List<Server> allList = lb.getAllServers(); //获取所有的服务
37
38
39         /*-----*/
40         /*我们知道，这下面就是具体的算法，上边的都是前期的准备工作*/
41
42         //         int serverCount = allList.size();
43         //         if (serverCount == 0) {
44         //             return null;
45         //         }
46         //
47         //         int index = chooseRandomInt(serverCount); //这个是索引，用于指向当前用
哪个服务
48         //         server = upList.get(index); //让服务 = 从活着的服务里面获取新的服务
49
50         //2. 我们把上面的全部都注释掉
51         if (total<5){ //假如当前服务的调用次数小于5次
52             server = upList.get(currentIndex); //让服务为当前还活着的服务
53             /*
54                 这里就是优化了，我完全可以让server = allList.get(currentIndex)，但是
这样的话假如服务崩了就没治了
55                 */
56             total++; //让总数++
57         }else {
58             total = 0; //假如超过了5次，那么对总数进行重置
59             currentIndex++; //我们的游标也往前一位
60             if (currentIndex>=upList.size()){
61                 /*
62                     假如游标 >= 还活着的总个数
63                     这里应该是游标 >= 还活着的总数量，不要忘记我们的游标是从0开始的
64                 */
65                 currentIndex = 0; //重置游标
66             }
67             server = upList.get(currentIndex); //从活着的服务中，获取指定的服务来进
行操作
68         }
69
70         /*-----*/
71         /*上面的就是具体的算法，下面就是一些收尾*/
72
73         if (server == null) { //假如服务没有获取到，还为null
74             Thread.yield(); //进行处理，咋处理的也看不懂....
75             continue;
76         }
77
78         if (server.isAlive()) { //假如这个服务还活着
79             return (server); //返回
80         }
81
82         server = null; //服务置空
83         Thread.yield(); //处理
84     }
85
86     return server;
87
88 }
89
90

```

```

91      /*下面这三个方法代码这么少，一看就不是实现算法的地方*/
92
93
94      protected int chooseRandomInt(int serverCount) {
95          return ThreadLocalRandom.current().nextInt(serverCount);
96      }
97
98      @Override
99      public Server choose(Object key) {
100          return choose(getLoadBalancer(), key);
101      }
102
103      @Override
104      public void initWithNiwsConfig(IClientConfig clientConfig) { //这个方法都是空的，
        说明没啥用
105
106      }
107  }

```

这个算法很low，但是毕竟也是改造了

### 3. 配置配置类 `com.bean.ribbonconfig.RibbonConfig`

按照官网说的，配置类要有 `@Configuration`

```

1  @Configuration
2  public class RibbonConfig {
3
4      @Bean
5      public IRule myRule(){
6          return new MyRule();
7      }
8
9  }

```

### 4. 修改主启动类，让它扫描到具体的包

```

1  @SpringBootApplication
2  @EnableEurekaClient //注解
3  @RibbonClient(name = "SPRINGCLOUD-PROVIDER-DEPT", configuration = RibbonConfig.class)
4
5      //name指的是应用名称，也就是服务提供者的spring.application.name = springcloud-provider-
    dept
6  public class DeptConsumer_80 {
7      public static void main(String[] args) {
8          SpringApplication.run(DeptConsumer_80.class, args);
9      }
10 }

```

### 5. 开启7001,8001,8002,8003,80，来测试一下

成功

# Feign

## 简介

Feign是声明式的web service客户端，它让微服务之间的调用变得更简单了，类似controller调用service。SpringCloud集成了Ribbon和Eureka，可在使用Feign时提供负载均衡的http客户端

只需要创建一个接口，然后添加注解即可！

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

- 微服务名字(Ribbon)
- 接口和注释(Feign)

### Feign能干嘛

- 旨在使编写Java Http客户端变得容易
- 前面在使用Ribbon+RestTemplate时，利用RestTemplate对Http请求的封装处理，形成了一套模板化的调用方法。

但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用

所以通常都会针对每一个微服务自行封装一些客户端类来包装这些依赖服务的调用。

所以，Feign在此基础上做了进一步的封装

**在Feign的实现下，我们只需要创建一个接口并加上注释来配置它(类似于以前Dao接口上标注Mapper注解)**

现在是在一个微服务接口上标注一个Feign注解

Feign也可以实现负载均衡，不过和Ribbon实现的方式不同，两者都可以用，两者都要会

## 使用步骤

要使用Feign，我们要进行改造

按理说我们应该对 `springcloud-consumer-80` 进行改造，但是随后一想，代码不能全丢了，所以我们新建一个模块，将80的所有代码全都移过去，然后对它进行改造

新建的模块叫做 `springcloud-consumer-feign`，端口还是80，主启动类名字改为 `FeignDeptConsumer_80`

1. 将Ribbon相关的全部代码删掉，包括 `com.bean.ribbonconfig.*`，主启动类的注解配置
2. 因为我们是客户端服务，所以我们要在 `springcloud-api` 进行改造
  1. 导入Feign的依赖

Feign的依赖和Ribbon的依赖区别就是里面的ribbon改为feign，比如：

```
1      <!-- Ribbon -->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-ribbon</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
```

```
1      <!-- Feign -->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-feign</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
```

2. 添加一个service包，里面加上服务，服务里面的方法都是服务提供者里面的方法，没有任何区别

- `com.beat.service.DeptClientService`

```
1      @FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT")//value:微服务的名字，也就是提供者
      的spring.application.name = springcloud-provider-dept
2      public interface DeptClientService {
3
4          @PostMapping("/dept/add")
5          public boolean addDept(Dept dept);
6
7          @GetMapping("/dept/get/{id}")
8          public Dept queryById(@PathVariable("id")Long id);
9
10         @GetMapping("/dept/list")
11         public List<Dept> queryAll();
12
13     }
```

注意加入的注解

- >
- > 我们以前在controller中写的东西全都移动到这里来了，而且请求的路径也一样
- >

3. 修改 `springcloud-consumer-feign` 注意先导入pom依赖

1. 修改controller

```
1      @RestController
2      public class DeptConsumerController {
3
4          //      @Autowired
5          //      private RestTemplate restTemplate;
6          //      RestTemplate也不要了
7
8          @Autowired
9          private DeptClientService service = null;
10
```

```

11 // private static final String REST_URL_PREFIX="http://SPRINGCLOUD-
    PROVIDER-DEPT";
12 //这个我们不要了，因为我们要使用接口方式去调用
13
14 @RequestMapping("/consumer/dept/get/{id}")
15 public Dept get(@PathVariable("id")Long id){
16 // return
    restTemplate.getForObject(REST_URL_PREFIX+"/dept/get"+id,Dept.class);
17 //这个我们也不要了
18 return this.service.queryById(id);
19 }
20
21 @RequestMapping("/consumer/dept/add")
22 public boolean add(Dept dept){
23 // return
    restTemplate.postForObject(REST_URL_PREFIX+"/dept/add",dept,Dept.class);
24 return this.service.addDept(dept);
25 }
26
27 @RequestMapping("/consumer/dept/list")
28 public List<Dept> list(){
29 // return
    restTemplate.getForObject(REST_URL_PREFIX+"/dept/list",List.class);
30 return this.service.queryAll();
31 }
32
33 }

```

## 2. 修改主启动类

```

1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableFeignClients(basePackages = "com.bean.springcloud")//扫描哪个地方的包
4 public class FeignDeptConsumer_80 {
5     public static void main(String[] args) {
6         SpringApplication.run(FeignDeptConsumer_80.class,args);
7     }
8 }

```

视频上还要加上一个@Component注解，但是其实不用，因为  
 @SpringBootApplication内包含@Component注解  
 再加上@Component注解会报错  
 不知道视频上为什么没有报错

## 4. 测试

成功  
 关于Ribbon和Feign，其实用哪个都可以

## 5. 假如报错：

1. **com.netflix.client.ClientException: Load balancer does not have available server for client**，可以在 `feign-80` 配置文件上加入：

```
1 ribbon:
2 eureka:
3   enabled: true
```

2. 等待一会就好

---

## Hystrix（豪猪）

### 服务熔断

---

#### 理论准备

##### 分布式系统面临的问题

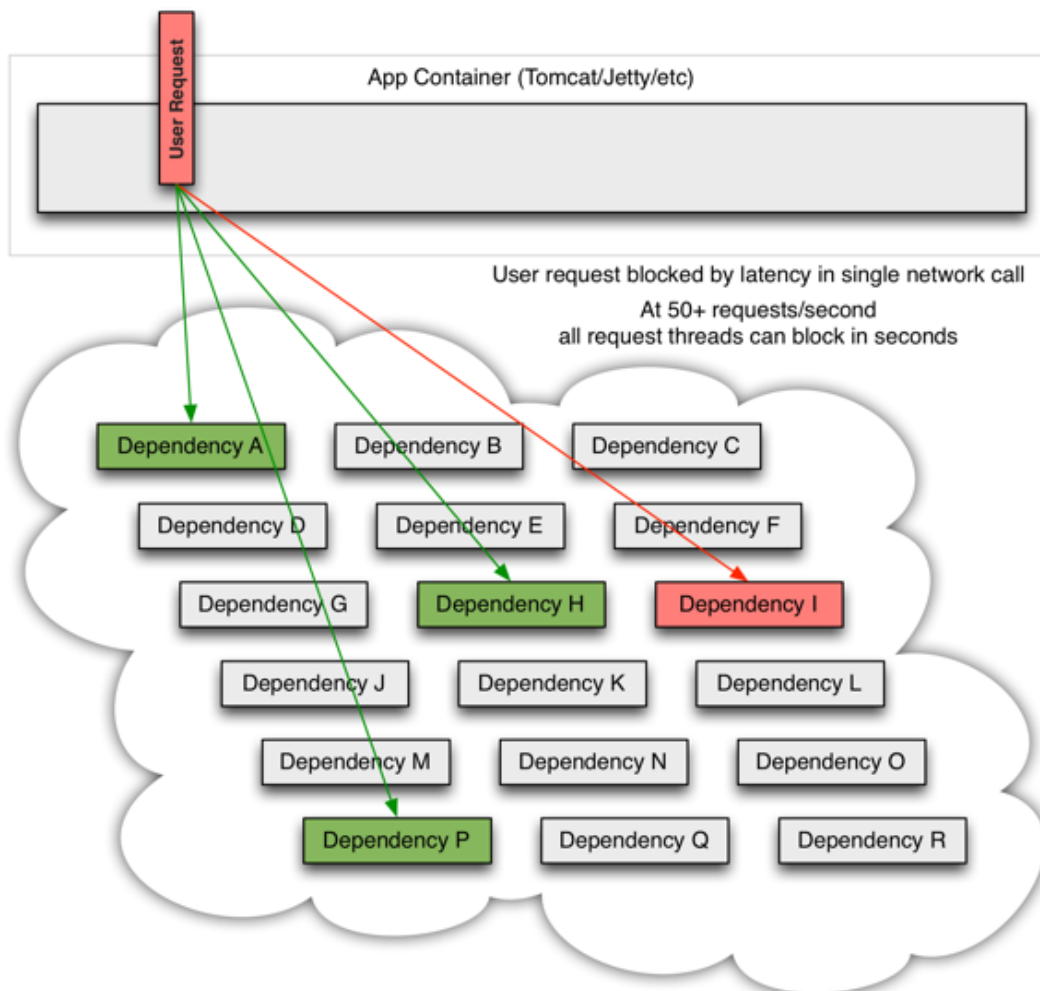
复杂分布式体系结构中的应用程序有数十个依赖关系，每一个依赖在某些时候将不可避免的失败

##### 服务雪崩

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C,微服务B和微服务C又调用其他的微服务，这就是所谓的“扇出”、如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃,所谓的“雪崩效应”。

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

我们需要：弃车保帅



比如这个时候，A调用H，H调用I，I调用P

假如某一个服务调用了别的的服务的时候调用失败，比如H调用I失败

那么服务雪崩

因为这个时候服务崩了，这个时候我们使用I的备份CFROR等等，只对客户端返回一个消息：服务崩了

## 什么是Hystrix

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

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



## Hystrix能干嘛

- 服务降级
- 服务熔断
- 服务限流
- 服务监控

## 服务熔断是什么

熔断机制是对应雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，**进而熔断该节点微服务的调用，快速返回错误的响应信息**。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制。

熔断机制的注解是 `@HystrixCommand`。

## 前期准备、熔断编写及测试

1. 把8001的项目拷贝成一个一模一样的模块，叫做 `springcloud-provider-dept-hystrix-8001`
2. 把启动类改一下，把application中的描述信息改一下

```
1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableDiscoveryClient
4  public class HystrixDeptProvider_8001 {
5
6      public static void main(String[] args) {
7          SpringApplication.run(HystrixDeptProvider_8001.class, args);
8      }
9  }
```

```
1
2  server:
3      port: 8001
4
5  mybatis:
6      type-aliases-package: com.bean.pojo
7      mapper-locations: classpath:mybatis/mapper/*.xml
8      config-location: classpath:mybatis/mybatis-config.xml
9
10 spring:
11     application:
12         name: springcloud-provider-dept
13     datasource:
14         type: com.alibaba.druid.pool.DruidDataSource
15         driver-class-name: org.gjt.mm.mysql.Driver
```

```

16     url: jdbc:mysql://localhost:3306/db01?useUnicode=true&characterEncoding=utf-
      8&serverTimezone=Asia/Shanghai
17     username: root
18     password: root
19
20     # Eureka生产者端的配置
21     eureka:
22       client:
23         service-url:
24           defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7
003.com:7003/eureka/
25       instance:
26         instance-id: springcloud-provider-hystirx_dept8001 # 描述改一下
27
28     info:
29       app.name: bean-project
30       company.name: blog.bean

```

3. 导入依赖，依赖类似ribbon改为feign，也是把stater后面的字改为hystrix

```

1     <dependency>
2       <groupId>org.springframework.cloud</groupId>
3       <artifactId>spring-cloud-starter-hystrix</artifactId>
4       <version>1.4.6.RELEASE</version>
5     </dependency>

```

前面的pom文件都是复制8001的，这里只放出hystrix的

4. 重新写一下Controller，让Controller结构更加清楚，添加熔断机制

```

1     /*把没用的Conctroller都删了*/
2     @RestController
3     public class DeptConrtoller {
4
5         @Autowired
6         private DeptService deptService;
7
8         @GetMapping("/dept/get/{id}")
9         @HystrixCommand(fallbackMethod = "hystrixGet")//假如服务崩了，会自动调用hystrixGet
方法
10        public Dept get(@PathVariable("id")Long id){
11            Dept dept = deptService.queryById(id);
12            /*
13                这里有点问题：假如查找不存在的id，那么查出来就是null
14                这么返回前端肯定爆出500
15                那么这个时候就需要服务熔断
16            */
17            if (dept==null){
18                //假如dept为null，那么直接抛出异常，模拟服务崩溃
19                throw new RuntimeException();
20            }
21
22            return dept;
23        }
24
25        //服务熔断机制，服务崩了之后的备选方案

```

```

26     public Dept hystrixGet(@PathVariable("id")Long id){
27         return new Dept()
28             .setDname("id=>"+id+" 不存在, 或者无法查询")
29             .setDb_source("no this database in MySQL");
30     }
31
32
33
34 }

```

## 5. 在启动类上添加对熔断的支持

```

1     @SpringBootApplication
2     @EnableEurekaClient
3     @EnableDiscoveryClient
4     @EnableCircuitBreaker//注意不是这个@EnableHystrix, 这个注解是另一个注解, 我们要用这个注解, 把
    断路器打开
5     public class HystrixDeptProvider_8001 {
6
7         public static void main(String[] args) {
8             SpringApplication.run(HystrixDeptProvider_8001.class, args);
9         }
10    }
11

```

## 6. 启动7001, hystrix-8001, feign-80并测试

成功

```

▼ {
    "deptno": null,
    "dname": "id=>10 不存在, 或者无法查询",
    "db_source": "no this database in MySQL"
}

```

# 服务降级

资源不够用了, 忍痛把一些服务关掉

其实熔断和降级其实是差不多的, 区别之一就是服务熔断是针对某一个方法, 而服务降级是针对某一个类

我们对 `springcloud-api` 进行操作

1. 新建类 `DeptClientServiceFallbackFactory`, 继承 `FallbackFactory`

```

1  @Component
2  public class DeptClientServiceFallbackFactory implements FallbackFactory {
3      @Override
4      public Object create(Throwable throwable) {
5          return null;
6      }
7  }

```

## 2. 对这个类进行改造

```

1  @Component
2  public class DeptClientServiceFallbackFactory implements FallbackFactory {
3      @Override
4      public DeptClientService create(Throwable throwable) {
5          //返回值为DeptClientService这个类，也就是说对这个类进行服务降级
6
7          return new DeptClientService() {
8              @Override
9              public boolean addDept(Dept dept) {
10                 return false;
11             }
12
13             @Override
14             public Dept queryById(Long id) {
15                 return new Dept()
16                     .setDname("进行了服务降级")
17                     .setDb_source("no this database in MySQL");
18             }
19
20             @Override
21             public List<Dept> queryAll() {
22                 return null;
23             }
24         };
25     }
26 }
27

```

为了看的清楚，这里只填写一个方法

## 3. 对 DeptClientService改造

```

1  //value:微服务的名字，也就是提供者的spring.application.name = springcloud-provider-dept
2  //fallbackFactory: 进行服务降级的类
3  @FeignClient(value = "SPRINGCLOUD-PROVIDER-DEPT", fallbackFactory =
DeptClientServiceFallbackFactory.class)
4  public interface DeptClientService {
5
6      @PostMapping("/dept/add")
7      public boolean addDept(Dept dept);
8
9      @GetMapping("/dept/get/{id}")
10     public Dept queryById(@PathVariable("id")Long id);
11
12     @GetMapping("/dept/list")
13     public List<Dept> queryAll();
14

```

#### 4. 对 `springcloud-consumer-dept-feign` 中的 `application.yaml` 进行改造

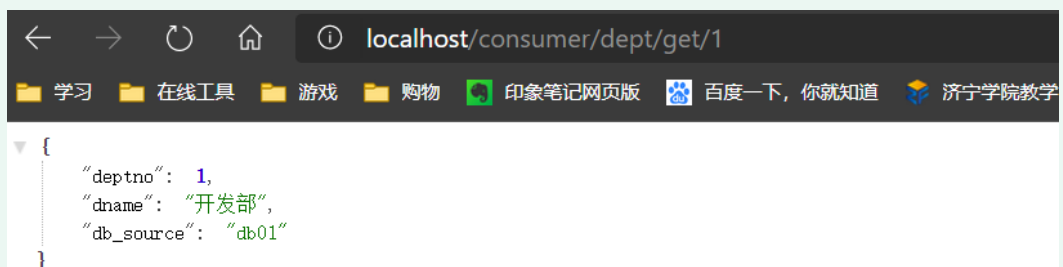
```

1  server:
2      port: 80
3
4  #Eureka配置
5  eureka:
6      client:
7          register-with-eureka: false # 我们不向注册中心注册自己，因为是消费者不是提供者
8          service-url:
9              defaultZone:
1             http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7
1             003.com:7003/eureka/
10     ribbon:
11         eureka:
12             enabled: true
13     #feign-hystrix 服务降级
14     feign:
15         hystrix:
16             enabled: true
17

```

#### 5. 启动7001, `hystirx-8001`, `feign-80` 进行测试

一开始我们可以访问（正常情况下）

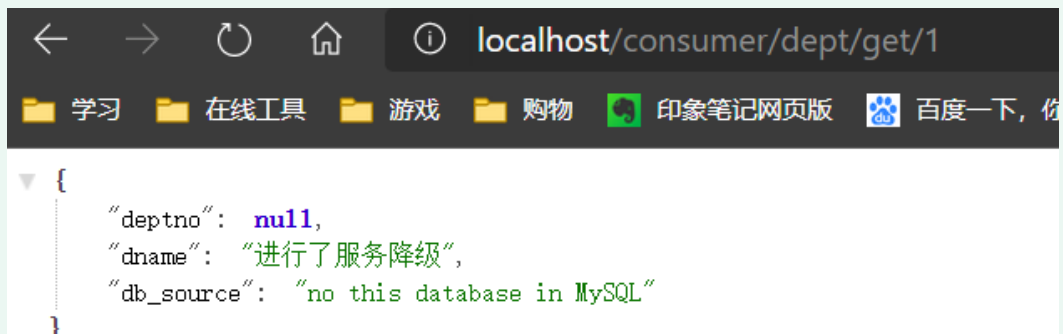


```

{
  "deptno": 1,
  "dname": "开发部",
  "db_source": "db01"
}

```

后来突发情况（关闭8001），出现了服务降级的提示



```

{
  "deptno": null,
  "dname": "进行了服务降级",
  "db_source": "no this database in MySQL"
}

```

# 服务熔断和服务降级的联系和区别

## 服务熔断

在服务端，某个服务超时或者异常，引起熔断

## 服务降级

在客户端，从整体网站的负载进行考虑，当某个服务熔断或者关闭之后，服务将不再进行调出

此时在客户端我们准备一个FallbackFactory，返回一个缺省值，整体服务水平下降了，但是好歹能用，比直接挂掉强

# 服务监控

## Dashboard服务监控

这个服务监控与消费者相关，与服务端无关了

## 环境搭建

1. 新建一个模块 `springcloud-consumer-hystrix-dashboard`
2. 从80导入依赖，并且增加 `hystrix` 的依赖，增加 `hystrix-dashboard` 的依赖

`hystrix` 依赖就是将ribbon的依赖中的ribbon字符串改为hystrix

`hystrix-dashboard` 依赖就是hystrix依赖中的hystrix改为hystrix-dashboard

```
1      <!--Hystrix-->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-hystrix</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
7      <!-- hystrix-dashboard -->
8      <dependency>
9          <groupId>org.springframework.cloud</groupId>
10         <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
11         <version>1.4.6.RELEASE</version>
12     </dependency>
```

3. 改一下端口为 `9001`
4. 新建包 `com.bean`，新建启动类，使用注解 `@EnableHystrixDashboard`

```
1  @SpringBootApplication
2  @EnableHystrixDashboard
3  public class HystrixDashboard {
4      public static void main(String[] args) {
5          SpringApplication.run(HystrixDashboard.class, args);
6      }
7  }
```

## 5. 检查一遍服务端是否有监控的依赖

分别去 8001, 8002, 8003 查看是否有 `spring-boot-starter-actuator` 依赖, 没有就加上

```
1      <!--监控信息-->
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-actuator</artifactId>
5      </dependency>
```

## 6. 启动9001, 尝试访问 `localhost:9001/hystrix`



### Hystrix Dashboard

Cluster via Turbine (default cluster): `http://turbine-hostname:port/turbine.stream`

Cluster via Turbine (custom cluster): `http://turbine-hostname:port/turbine.stream?cluster=[clusterName]`

Single Hystrix App: `http://hystrix-app:port/actuator/hystrix.stream`

Delay:  ms    Title:

这个页面有:

1. 监控的地址
2. 轮询事件
3. 监控实例的名字
4. 要配置一个 `http://hystrix-app:port/actuator/hystrix.stream`

## 使用

假如我们想注册 `hystrix-8001`, 那么需要在启动类上增加一个Bean对象

1. 首先将 `hystrix` 的包导入 `hystrix-8001`, 不过经过之前的编写应该已经有了
2. 在启动类上加上新的Bean对象

```
1      @SpringBootApplication
2      @EnableEurekaClient
3      @EnableDiscoveryClient
4      @EnableCircuitBreaker
5      public class HystrixDeptProvider_8001 {
6
```


```

7      public static void main(String[] args) {
8          SpringApplication.run(HystrixDeptProvider_8001.class, args);
9      }
10
11      @Bean
12      public ServletRegistrationBean registrationBean(){
13          ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
HystrixMetricsStreamServlet());
14          registrationBean.addUrlMappings("/actuator/hystrix.stream");
15          //这个地方，我们访问localhost:9001/hystrix的时候已经提示过了
16          return registrationBean;
17      }
18  }

```

### 3. 开启 hystrix-dashboard-9001 , 7001 , hystrix-8001

按照图中的分别填入



## Hystrix Dashboard

<http://localhost:8001/actuator/hystrix.stream>

*Cluster via Turbine (default cluster):* <http://turbine-hostname:port/turbine.stream>  
*Cluster via Turbine (custom cluster):* [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])  
*Single Hystrix App:* <http://hystrix-app:port/actuator/hystrix.stream>

Delay:  ms    Title:

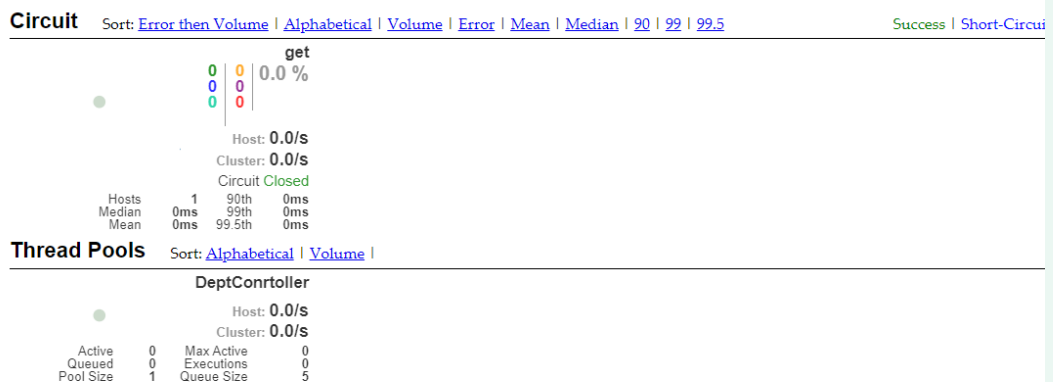
[Monitor Stream](#)

注意，我们监控的是8001服务的地址：

<http://localhost:8001/actuator/hystrix.stream>

点击按钮跳转

### Hystrix Stream: demo





#### 4. 监控不到:

1. 首先要监控的服务要启用熔断 `@EnableCircuitBreaker` , 我们启动的就是 `hystrix-8001` 这个带熔断的
2. 注意要加上Bean对象

```
1  @Bean
2  public ServletRegistrationBean registrationBean(){
3      ServletRegistrationBean registrationBean = new ServletRegistrationBean(new
4      HystrixMetricsStreamServlet());
5      registrationBean.addUrlMappings("/actuator/hystrix.stream");
6      //这个地方, 我们访问localhost:9001/hystrix的时候已经提示过了
7      return registrationBean;
8  }
```

3. 该服务只能监控有熔断注解 `@HystrixCommand` 的方法

监控中所代表的意义:

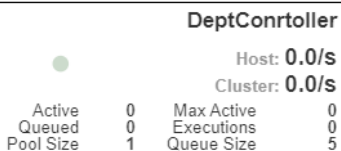
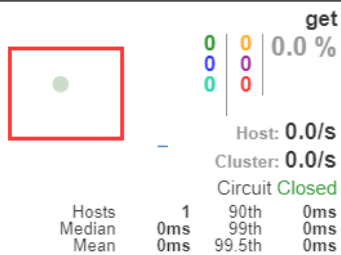
#### 七色



#### 一圈

实心圆: 两种含义, 通过颜色的变化代表着实例的健康程度, 通过大小代表流量变化

1. 绿——黄——橙——红 递减 (虽然看着像灰色, 但确实是绿色)
2. 流量越大, 实心圆越大



## Zuul

### 概述

`http://localhost:9001/hystrix` , 其中 `localhost:9001` 不应该被显示出来, 而是应该使用 `www.xxx` 来代替

但是这样出来了一个问题: 不同的微服务有不同的地址, 比如8001,8002,8003等等

我们要配置的这个路径是一个路径, 那么也需要分发的概念

#### 什么是Zuul?

Zuul包含了对请求的路由和过滤两个最主要的功能:

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

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

注意: Zuul服务最终还是会注册进Eureka

提供:代理+路由+过滤三大功能!

#### Zuul能干嘛?

- 路由

- 过滤

官网文档: <https://github.com/Netflix/zuul>

## 环境搭建

1. 新建项目 `springcloud-zuul-9527`
2. 导入依赖, 从 `springcloud-consumer-hystrix-dashboard` 中导入依赖
3. 新增依赖 `zuul`, 也是像之前一样, 把hystrix的依赖的hystrix字符串改为zuul

```
1      <!--zuul-->
2      <dependency>
3          <groupId>org.springframework.cloud</groupId>
4          <artifactId>spring-cloud-starter-zuul</artifactId>
5          <version>1.4.6.RELEASE</version>
6      </dependency>
```

4. 配置 `application.yaml`, 设置端口为9527, 设置服务名称为 `springcloud-zuul`, 设置eureka, 信息, zuul

```
1      server:
2          port: 9527
3
4      #设置服务名字
5      spring:
6          application:
7              name: springcloud-zuul
8
9
10     # Eureka生产者端的配置
11     eureka:
12         client:
13             service-url:
14                 defaultZone:
15                     http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
16
17         # 注册中心地址把7001,7002,7003都加上
18         instance:
19             instance-id: springcloud-zuul # 描述信息
20
21     # 配置信息
22     info:
23         app.name: bean-project
24         company.name: blog.bean
25
26     zuul:
27         routes:
28             #zuul.routes是一个map类型的, 所以我们下面的随便起名字就好, 这个map的用法是将所有用到的服务名字都用设定的路径来代替
29             dept.serviceId: springcloud-provider-dept # 这个配置的是服务名字
```

```
29     dept.path: /dept/** # 我们使用/dept/**来代替
30
```

5. 更改host文件, 新增一个本地域名为 `www.zuul.com`

```
127.0.0.1    localhost
127.0.0.1    eureka7001.com
127.0.0.1    eureka7002.com
127.0.0.1    eureka7003.com
127.0.0.1    www.zuul.com|
```

6. 写一个包 `com.bean.springcloud`, 新增启动类, 增加注解 `@EnableZuulProxy`

```
1  @SpringBootApplication
2  @EnableZuulProxy
3  public class ZuulProxy {
4      public static void main(String[] args) {
5          SpringApplication.run(ZuulProxy.class, args);
6      }
7  }
8
```

7. 启动 `7001`, `hystrix-8001`, `9527` 测试



但是这样有一个问题, 这样访问之后直接输入 `xxx/项目名字/xxx` 也能访问:

`http://www.zuul.com:9527/springcloud-provider-dept/dept/get/1`

我们要配置让原路径不能访问:

```
1  server:
2      port: 9527
3
4      #设置服务名字
5  spring:
6      application:
7          name: springcloud-zuul
8
9
10     # Eureka生产者端的配置
11     eureka:
12         client:
```

```

13     service-url:
14     defaultZone:
        http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:
        7003/eureka/
15     instance:
16         instance-id: springcloud-zuul # 描述信息
17
18
19 # 配置信息
20 info:
21     app.name: bean-project
22     company.name: blog.bean
23
24 zuul:
25     routes:
26         dept.serviceId: springcloud-provider-dept
27         dept.path: /mydept/**
28         ignored-services: "*" # 忽略这个服务下的原路径，单个服务可以使用服务名，比如：springcloud-provider-
        dept

```



## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Mar 27 17:56:45 CST 2020

There was an unexpected error (type=Not Found, status=404).

No message available

果然不能访问了，项目名字也隐藏了



```

{
  "deptno": 1,
  "dname": "开发部",
  "db_source": "db01"
}

```

现在只能用这个路径来进行访问

### 配置前缀

配置前缀之后首先要写这个前缀，然后是项目路径

```

1 server:
2     port: 9527

```

```

3
4   #设置服务名字
5   spring:
6     application:
7       name: springcloud-zuul
8
9
10  # Eureka生产者端的配置
11  eureka:
12    client:
13      service-url:
14        defaultZone:
15          http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:
16          7003/eureka/
17
18    instance:
19      instance-id: springcloud-zuul # 描述信息
20
21  # 配置信息
22  info:
23    app.name: bean-project
24    company.name: blog.bean
25
26  zuul:
27    routes:
28      dept.serviceId: springcloud-provider-dept
29      dept.path: /mydept/**
30      ignored-services: "*"
31      prefix: /bean # 配置前缀

```



## SpringCloud Config

微服务A, B, C有三个配置文件

假如我们能有一个具体的配置中心, 要什么配置就去配置中心读, 那么就好了

Spring Cloud Config为微服务架构中的微服务提供集中化的外部配置支持, 配置服务器为各个不同微服务应用的所有环节都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

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

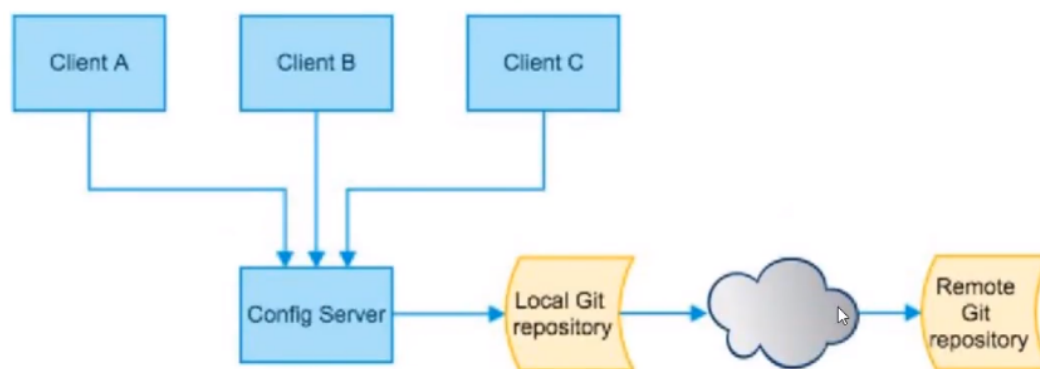
## 客户端

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，并且在启动的时候从配置中心获取和加载配置信息。

配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理。

并且可以通过Git客户端工具来方便地进行管理和访问配置内容

下面这张图很清晰了：用户链接服务，服务链接远程仓库



## SpringCloud config分布式配置中心能干嘛

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

## SpringCloud config分布式配置中心与github整合

由于Spring Cloud Config默认使用Git来存储配置文件(也有其他方式，比如支持SVN和本地文件)

但是最推荐的还是Git，而且使用的是http / https访问的形式;

还可以使用码云，coding，建议使用码云

---

## 服务端

在码云上新建一个仓库springcloudConfig, 然后往里面添加一个配置文件 `application.yaml`

```
1  spring:
2    profiles:
3      active: dev
4
5  ---
6
7  spring:
8    profiles: dev
9    application:
10     name: springcloud-cofnig-dev
11
12  ---
13  spring:
14    profiles: test
15    application:
16     name: springcloud-cofnig-test
```

关于Git的不说, 下面说一下服务

新建一个模块 `springcloud-config-server-3344`, 在这里面配置

### 1. 导入依赖

```
1      <dependencies>
2        <!--Web-->
3        <dependency>
4          <groupId>org.springframework.boot</groupId>
5          <artifactId>spring-boot-starter-web</artifactId>
6        </dependency>
7
8        <!--ConfigServer-->
9        <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-
cloud-config-server -->
10       <dependency>
11         <groupId>org.springframework.cloud</groupId>
12         <artifactId>spring-cloud-config-server</artifactId>
13         <version>2.1.6.RELEASE</version>
14       </dependency>
15
16     </dependencies>
```

一定要有一个springcloudconfigserver, 其余的有需求就再加

### 2. `application.yaml`



```

1  server:
2    port: 3344
3  spring:
4    application:
5      name: springcloud-config-server
6    cloud: # 链接远程仓库
7    config:
8      server:
9        git:
10         uri: https://gitee.com/BEANGITEE/springcloudConfig.git
11         # 具体的位置, 比如码云, 为此我单独创建了一个仓库, 用https的

```

### 3. 主启动类开启, 加上那个注解

```

1  @SpringBootApplication
2  @EnableConfigServer //加上注解
3  public class Config_Server_3344 {
4      public static void main(String[] args) {
5          SpringApplication.run(Config_Server_3344.class,args);
6      }
7  }

```

### 4. 测试

#### 后缀版

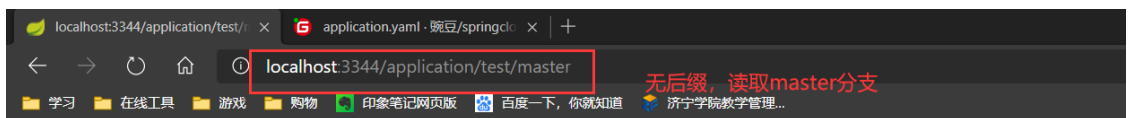


```

spring:
  application:
    name: springcloud-cofnig-dev
  profiles:
    active: dev

```

#### 无后缀版



```

{
  "name": "application",
  "profiles": [
    "test"
  ],
  "label": "master",
  "version": "c3eb5a92d4b3c1dead9042870ed6348c031e3c80",
  "state": null,
  "propertySources": [
    {
      "name": "https://gitee.com/BEANGITEE/springcloudConfig.git/application.yaml (document #2)",
      "source": {
        "spring.profiles": "test",
        "spring.application.name": "springcloud-cofnig-test"
      }
    },
    {
      "name": "https://gitee.com/BEANGITEE/springcloudConfig.git/application.yaml (document #0)",
      "source": {
        "spring.profiles.active": "dev"
      }
    }
  ]
}

```

# 客户端

在git文件中新建一个 `config-client.yaml` 作为客户端的配置

```
1  spring:
2    profiles:
3      active: dev
4
5  ---
6  server:
7    port: 8201
8
9  spring:
10   profiles: dev
11   application:
12     name: springcloud-cofnig-dev
13
14   # Eureka配置
15   eureka:
16     client:
17       service-url:
18         defaultZone: http://eureka7001.com:7001/eureka/
19
20   ---
21   server:
22     port: 8202
23
24   spring:
25     profiles: test
26     application:
27       name: springcloud-cofnig-test
28
29   # Eureka配置
30   eureka:
31     client:
32       service-url:
33         defaultZone: http://eureka7001.com:7001/eureka/
```

1. 写一个模块: `springcloud-config-client-3355`
2. 依赖

```
1      <dependencies>
2        <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-web</artifactId>
5        </dependency>
6        <!--config, 这个是客户端的config-->
7        <dependency>
8          <groupId>org.springframework.cloud</groupId>
9          <artifactId>spring-cloud-starter-config</artifactId>
10         <version>2.1.1.RELEASE</version>
11       </dependency>
12       <dependency>
13         <groupId>org.springframework.boot</groupId>
14         <artifactId>spring-boot-starter-actuator</artifactId>
```

```

15         </dependency>
16     </dependencies>

```

### 3. 编写配置 bootstrap.yaml

以前学过 `application.yaml`，这里使用 `bootstrap.yaml`

`bootstrap.yaml` 是系统级别的配置，而 `application.yaml` 是用户级别的配置

一般情况下使用用户级别的就够了，但是这里有可能会和远程的冲突，所以使用系统级别的加载器，更高优先级

```

1     spring:
2         cloud:
3             config:
4                 uri: http://localhost:3344
5                 # 这里连接的是我们刚才编写服务端的地址，而服务端链接远程仓库。这就是用户连接服务，服务连接远程
           仓库
6                 name: config-client #要从git上读取的资源，不用后缀
7                 profile: dev #使用dev开发环境
8                 label: master # 去master分支拿

```

### 4. 编写Controller

```

1     @RestController
2     public class ConfigController {
3
4         @Value("${spring.application.name}")
5         private String applicationName;
6         @Value("${eureka.client.service-url.defaultZone}")
7         private String eurekaServer;
8         @Value("${server.port}")
9         private String port;
10
11         @RequestMapping("/config")
12         public String getConfig(){
13             return "applicationName: "+applicationName+",
           eurekaServer: "+eurekaServer+", port: "+port;
14         }
15
16
17     }

```

### 5. 编写启动类

```

1     @SpringBootApplication
2     public class ConfigClient_3355 {
3         public static void main(String[] args) {
4             SpringApplication.run(ConfigClient_3355.class,args);
5         }
6     }

```

### 6. 启动服务端，然后启动客户端

因为我们在远程的配置上写的是8201，所以读取到的端口号自动配置成了8201，现在我们进行访问：



applicationName: springcloud-cofnig-dev, eurekaServer: http://eureka7001.com:7001/eureka/, port: 8201

成功

---

从此之后别人看不懂了你的配置，实现了配置与编码解耦