2019-3-11

# 第一章 mianshi题

1.题目：

1）微服务是什么？

1. 微服务间是如何独立通讯的？

3）springCloud 和Dubbo有哪些区别？\*

4）SpringBoot 和SpringCloud，请你谈谈对他们的理解？

5）什么是服务熔断？什么是服务降级？

6）微服务的优缺点是什么？说下你在项目中遇到的坑

7）你所知道的微服务技术栈有哪些？请列举一二

8）eureka 和zookeeper都可以提供服务注册与发现的功能，说说两个的区别？

# 第二章 微服务概述

1.是什么？

微服务是一种架构或者风格，**它提倡将单一应用程序划分成一组小的服务**，每个服务运行在其独立的自己的进程中，服务直接互相协调、互相配合，为用户提供最终价值。服务之间采用轻量级的通信机制互相沟通（通常是基于HTTP的RESTful API，注意：Dubbo服务间调用是基于RPC）。每个服务都围绕着具体业务进行构建，并且能够独立地部署到生成环境、类生成环境等。另外，应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建，可以有一个非常轻量级的集中式管理来协调这些服务，可以使用不同的语言来编写服务，也可以使用不同的数据存储。

技术维度理解：

微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地取耦合，每一个微服务提供一个业务功能的服务，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。

Ps:阅读：微服务架构提出者马丁福勒论文

2.微服务与微服务架构

》微服务

强调的是服务的大小，它关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用，狭意的看，可以看作eclipse里面一个个微服务工程/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。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。

3.微服务优缺点

》优点：

高内聚，代码容易理解，这样能聚焦一个指定的业务功能或业务需求；

开发简单、开发效率提高，一个服务可能就是专一的只干一件事；

单独开发；

松耦合，无论开发还是部署都是独立的；

微服务能使用不同的语言开发；

易于与第三方集成，微服务允许容易且灵活的方式自动部署，通过持续集成的工具：如jenkins；

易于开发和维护，小团队能够专注于自己的工作成果，无需通过合作才能体现价值；

微服务允许利用融合新技术；

微服务只是业务逻辑的代码，不会和html等界面组件混合；

每个微服务都有自己的存储能力，可以有自己的数据库，也可以统一数据库。

》缺点：

开发人员要处理分布式系统的复杂性；

多服务运维难度，随着服务的增加，运维压力也在增大；

系统部署依赖；

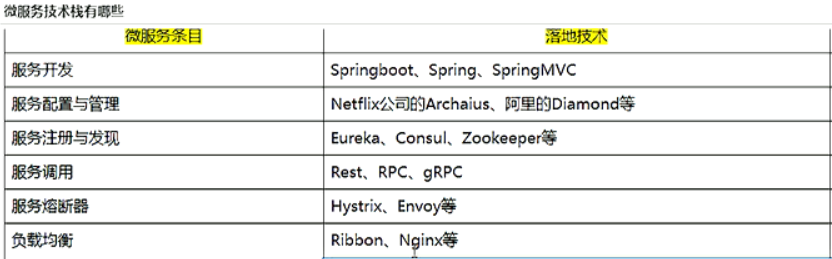
服务间通信成本；

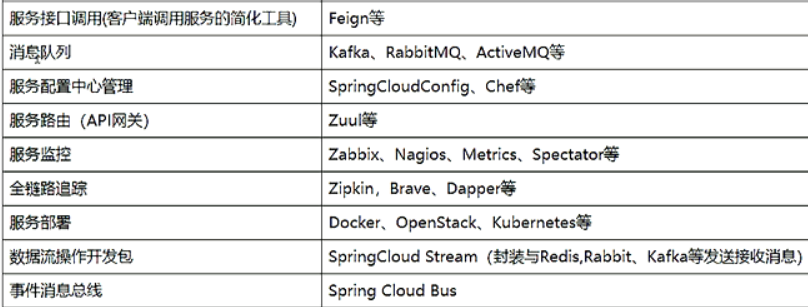
数据一致性；

系统集成测试；

性能监控...

4.微服务技术栈有哪些？





# 第三章 SpringCloud入门概述

1.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=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。

2.SpringCloud和springBoot是什么关系？

Springboot专注于快速的开发单个个体微服务。

springCloud关注全局的服务治理框架。

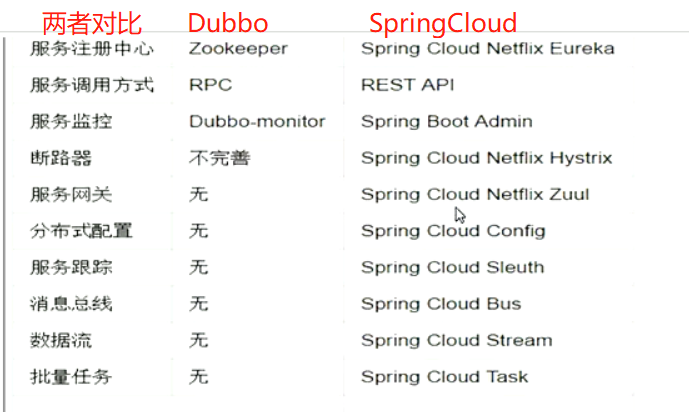
SpringCloud是关注全局的微服务协调治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来。为各个微服务之间提供，配置管理、服务发现、断路器、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务。

Spring boot可以离开springCloud独立开发使用，但是Springcloud离不开springboot，属于依赖关系。

3.Dubbo是怎么到springCloud的？

对比：

社区活跃度：springcloud更活跃

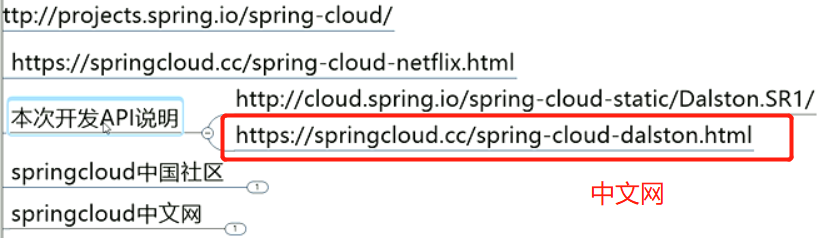


最大区别：SpringCloud采用基于HTTP的REST方式，dubbo是RPC。

品牌机和组装机的区别。

4.Springcloud介绍

哪里下载：官网：projects.spring.io/spring-cloud/

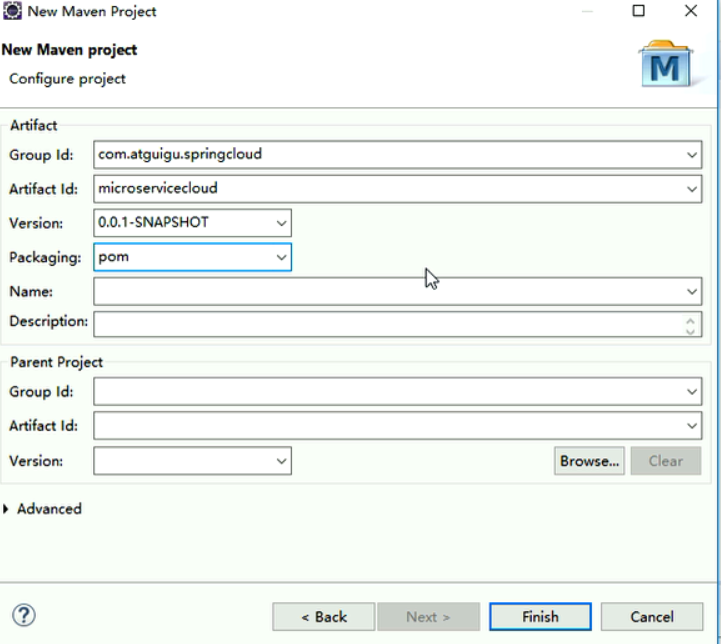




# 第四章REST微服务构建案例工程模块

1.新建microservicecloud父类工程，new->maven project->

定义pom文件，将后续各个子模块公用的jar包等统一提取出来。



Pom.xml文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<junit.version>4.12</junit.version>

<log4j.version>1.2.17</log4j.version>

<lombok.version>1.16.18</lombok.version>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-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>

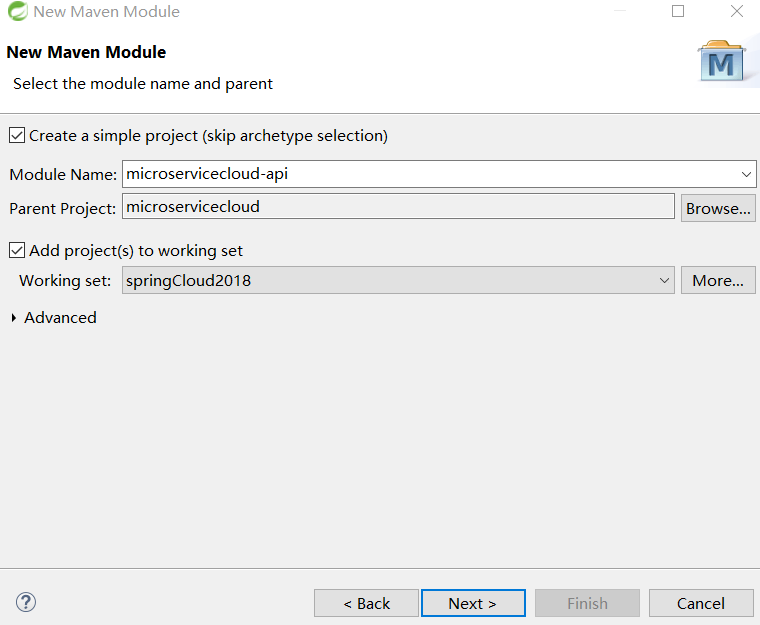
</dependencies>

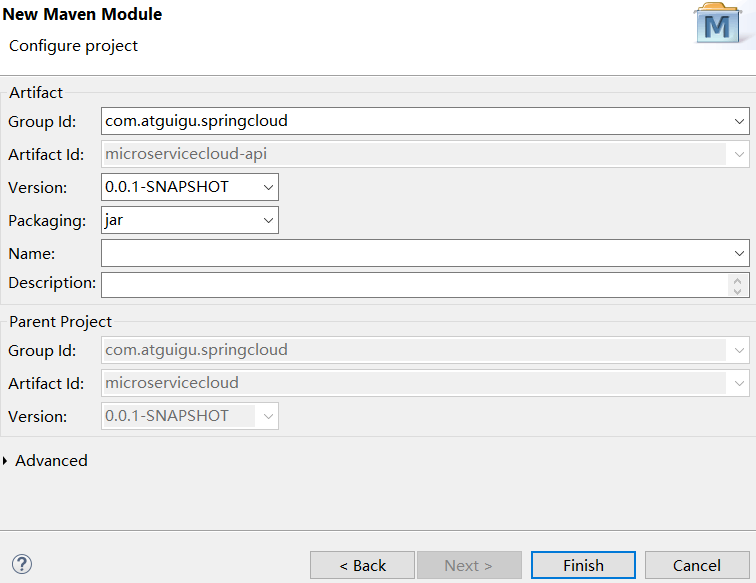
</dependencyManagement>

</project>

2.创建microservicecloud-api子工程

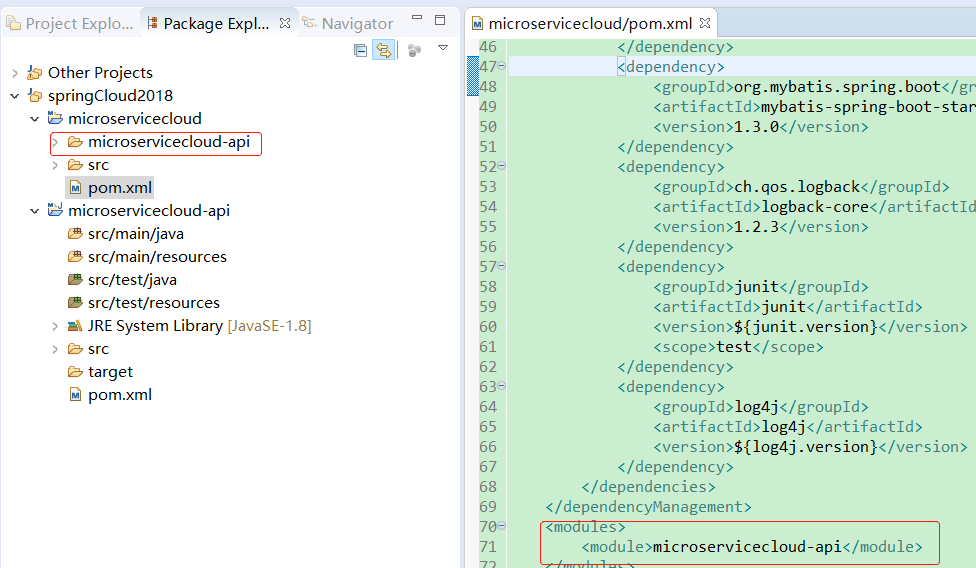
右键microservicecloud项目->New->other->maven module->





Finish

这时回头看父工程：依赖成功



子类pom文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-api</artifactId>

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

</dependencies>

</project>

创建entity类

**import** java.io.Serializable;

**import** lombok.Data;

**import** lombok.NoArgsConstructor;

/\*\*

\*

\* **@author** dada

\* **@date** 2019年3月12日

\*/

@Data

@NoArgsConstructor

**public** **class** Dept **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Long deptno; // 主键

**private** String dname; // 部门名称

**private** String db\_source;// 来自那个数据库，因为微服务架构可以一个服务对应一个数据库，同一个信息被存储到不同数据库

**public** Dept(String dname) {

**super**();

**this**.dname = dname;

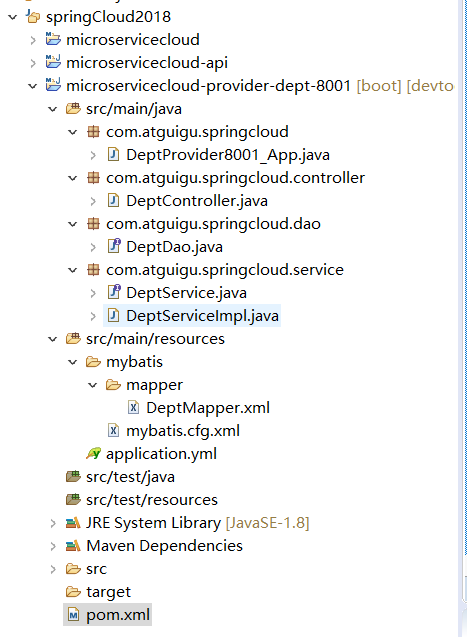
}

}

创建完成后，右键microservicecloud-api工程->run as->maven clear,再maven install

==================

3.创建子工程microservicecloud-provider-dept-8001



右键microservicecloud->new->maven module->……

1）

Pom.xml文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-provider-dept-8001</artifactId>

<dependencies>

<!-- 引入自己定义的api通用包，可以使用Dept部门Entity -->

<dependency>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<!-- actuator监控信息完善 -->

<dependency>

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

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

</dependency>

<!-- 将微服务provider侧注册进eureka -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-core</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- 修改后立即生效，热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

</project>

2）application.yml文件

server:

port: 8001

mybatis:

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

type-aliases-package: com.atguigu.springcloud.entities # 所有Entity别名类所在包

mapper-locations:

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

spring:

application:

name: microservicecloud-dept

datasource:

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

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

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

username: root

password: root

dbcp2:

min-idle: 5 # 数据库连接池的最小维持连接数

initial-size: 5 # 初始化连接数

max-total: 5 # 最大连接数

max-wait-millis: 200

3）mybatis.cfg.xml文件

<?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"* /><!-- 二级缓存开启 -->

</settings>

</configuration>

4）DeptMapper.xml文件

<?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.atguigu.springcloud.dao.DeptDao"*>

<select id=*"findById"* resultType=*"Dept"* parameterType=*"Long"*>

select deptno,dname,db\_source from dept where deptno=#{deptno};

</select>

<select id=*"findAll"* resultType=*"Dept"*>

select deptno,dname,db\_source from dept;

</select>

<insert id=*"addDept"* parameterType=*"Dept"*>

INSERT INTO dept(dname,db\_source) VALUES(#{dname},DATABASE());

</insert>

</mapper>

4.0）数据库脚本

DROP DATABASE IF EXISTS cloudDB01;

CREATE DATABASE cloudDB01 CHARACTER SET UTF8;

USE cloudDB01;

CREATE TABLE dept(

deptno BIGINT NOT NULL PRIMARY KEY AUTO\_INCREMENT,

dname VARCHAR(60),

db\_source VARCHAR(60));

INSERT INTO dept(dname,db\_source)VALUES('开发部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('人事部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('财务部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('市场部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('运维部',DATABASE());

SELECT \* FROM dept;

4.1）Dao层

**package** com.atguigu.springcloud.dao;

**import** java.util.List;

**import** org.apache.ibatis.annotations.Mapper;

**import** com.atguigu.springcloud.entities.Dept;

@Mapper

**public** **interface** DeptDao {

**public** **boolean** addDept(Dept dept);

**public** Dept findById(Long id);

**public** List<Dept> findAll();

}

5）service层接口

**package** com.atguigu.springcloud.service;

**import** java.util.List;

**import** com.atguigu.springcloud.entities.Dept;

**public** **interface** DeptService {

**public** **boolean** add(Dept dept);

**public** Dept get(Long id);

**public** List<Dept> list();

}

6）impl实现类

**package** com.atguigu.springcloud.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.atguigu.springcloud.dao.DeptDao;

**import** com.atguigu.springcloud.entities.Dept;

@Service

**public** **class** DeptServiceImpl **implements** DeptService {

@Autowired

**private** DeptDao deptDao;

@Override

**public** **boolean** add(Dept dept) {

**return** deptDao.addDept(dept);

}

@Override

**public** Dept get(Long id) {

**return** deptDao.findById(id);

}

@Override

**public** List<Dept> list() {

**return** deptDao.findAll();

}

}

7）controller层

**package** com.atguigu.springcloud.controller;

/\*\*

\*

\* **@author** dada

\* **@date** 2019年3月14日

\*/

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestBody;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

**import** org.springframework.web.bind.annotation.RestController;

**import** com.atguigu.springcloud.entities.Dept;

**import** com.atguigu.springcloud.service.DeptService;

@RestController

**public** **class** DeptController {

@Autowired

**private** DeptService service;

@RequestMapping(value = "/dept/add", method = RequestMethod.***POST***)

**public** **boolean** add(@RequestBody Dept dept) {

**return** service.add(dept);

}

@RequestMapping(value = "/dept/get/{id}", method = RequestMethod.***GET***)

**public** Dept get(@PathVariable("id") Long id) {

**return** service.get(id);

}

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

**public** List<Dept> list() {

**return** service.list();

}

}

8)启动类

**package** com.atguigu.springcloud;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** DeptProvider8001\_App {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(DeptProvider8001\_App.**class**, args);

}

}

9）启动8001项目测

http://localhost:8001/dept/list

================================

4、创建消费者consumer

Ps:RestTemplate

提供了多种便捷访问远程Http服务的方法，是一种简单便捷的访问restful服务模板类，是spring提供的用户访问Rest服务的客户端模板工具集。

使用：

(url,requestMap,ResponseBean.class)这三个参数分别代表REST请求地址、请求参数、HTTP响应转换被转换成的对象类型。

创建子工程：

microservicecloud-consumer-dept-80

1）pom.xml文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-consumer-dept-80</artifactId>

<description>部门微服务消费者</description>

<dependencies>

<dependency><!-- 自己定义的api -->

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<!-- Ribbon相关 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- 修改后立即生效，热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

</project>

2）application.yml文件

server:

port: 80

3）创建配置类

**package** com.atguigu.springcloud.cfgbeans;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.client.RestTemplate;

@Configuration

**public** **class** ConfigBean // boot -->spring applicationContext.xml --- @Configuration配置 ConfigBean =

// applicationContext.xml

{

@Bean

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

}

//@Bean

//public UserServcie getUserServcie()

//{

// return new UserServcieImpl();

//}

// applicationContext.xml == ConfigBean(@Configuration)

//<bean id="userServcie" class="com.atguigu.tmall.UserServiceImpl">

4）创建controller层

package com.atguigu.springcloud.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

import com.atguigu.springcloud.entities.Dept;

@RestController

public class DeptController\_Consumer

{

//private static final String REST\_URL\_PREFIX = "http://localhost:8001";

private static final String REST\_URL\_PREFIX = "http://MICROSERVICECLOUD-DEPT";

/\*\*

\* 使用 使用restTemplate访问restful接口非常的简单粗暴无脑。 (url, requestMap,

\* ResponseBean.class)这三个参数分别代表 REST请求地址、请求参数、HTTP响应转换被转换成的对象类型。

\*/

@Autowired

private RestTemplate restTemplate;

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

public boolean add(Dept dept)

{

return restTemplate.postForObject(REST\_URL\_PREFIX + "/dept/add", dept, Boolean.class);

}

@RequestMapping(value = "/consumer/dept/get/{id}")

public Dept get(@PathVariable("id") Long id)

{

return restTemplate.getForObject(REST\_URL\_PREFIX + "/dept/get/" + id, Dept.class);

}

@SuppressWarnings("unchecked")

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

public List<Dept> list()

{

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

}

}

//官网资料：https://docs.spring.io/spring-framework/docs/4.3.7.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html

5）启动类

**package** com.atguigu.springcloud;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** DeptConsumer80\_App {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(DeptConsumer80\_App.**class**, args);

}

}

6）测试：

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

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

<http://localhost/consumer/dept/add?dname=dada>

# 第五章：Eureka服务注册与发现

第一节：Eureka是什么

1.概念：Eureka是Netflix的一个子模块，也是核心模块之一。Eureka是一个基于REST的服务，用于定位服务，以实现云端中层服务发现和故障转移。服务注册于发现对于微服务框架来说是非常重要的，有了服务发现于注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件。功能类似于dubbo的注册中心，比如Zookeeper。

SpringCloud封装了NetFlix公司开发的Eureka的模块来实现服务注册和发现（请对比Zookeeper）。

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

而系统中的其他为服务，使用Eureka的客户连接到Eureka Server并维持心跳连接。这样系统的维护人员就可以通过EurekaServer 来监控系统中各个微服务是否正常运行。SpringCloud的一些其他模块（比如Zuul）就可以通过EurekaServer来发现系统中的其他微服务，并执行相关的逻辑。

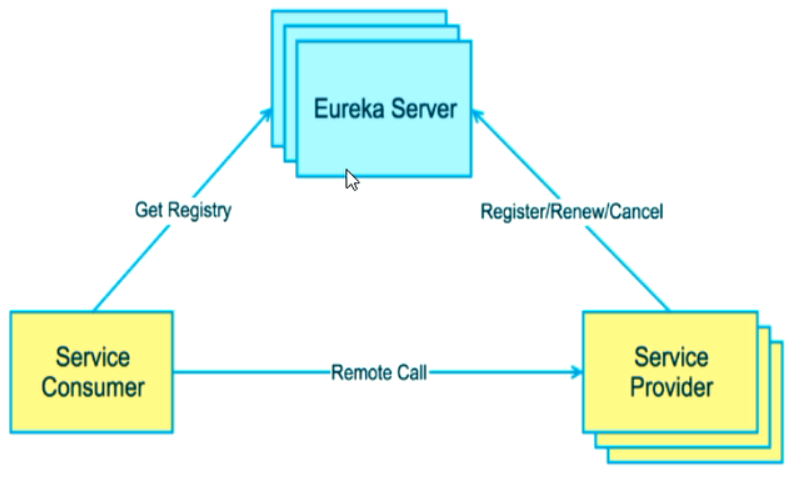
Eureka包含俩个组件：Eureka Server 和Eureka Client

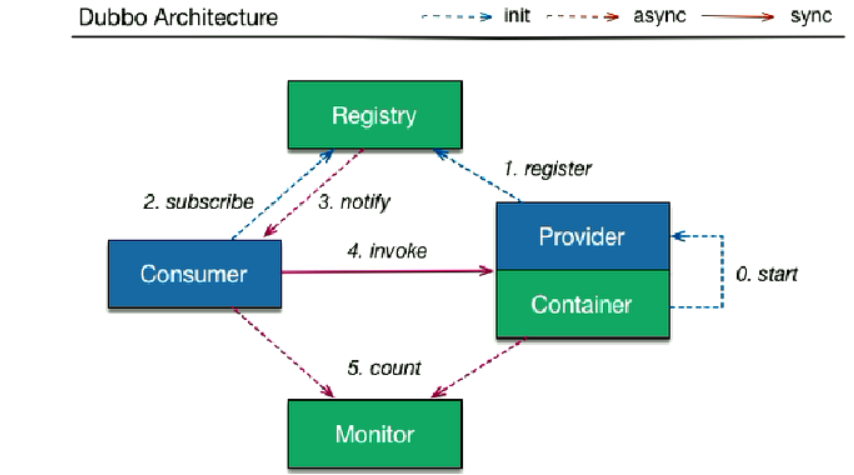
Eureka Server 提供服务注册服务

各个节点启动后，会在EurekaServer中进行注册，这样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的客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询（round-robin）复制算法的负载均衡器。在应用启动后，将会想Eureka Server发送心跳（默认周期为30秒）.如果Eureka Server 的多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除（默认90秒）。

对比：





第二节：创建服务中心Eureka

创建子项目microservicecloud-eureka-7001

1)pom.xml文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-eureka-7001</artifactId>

<dependencies>

<!--eureka-server服务端 -->

<dependency>

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

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

</dependency>

<!-- 修改后立即生效，热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

</project>

2）配置文件

application.yml

server:

port: 7001

eureka:

instance:

hostname: eureka7001.com #eureka服务端的实例名称

client:

register-with-eureka: **false** #false表示不向注册中心注册自己。

fetch-registry: **false** #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址（单机）。

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

3)启动类：

package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

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

public class EurekaServer7001\_App {

public static void main(String[] args) {

SpringApplication.run(EurekaServer7001\_App.class, args);

}

}

4）启动测试：

<http://localhost:7001/>

看到图标，测试正常

第三节：将已经有的注册进Eureka

1.引两个包

如修改8001工程，pom.xml 添加这两个包：

<!-- 将微服务provider侧注册进eureka -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

2.修改8001 的yml，后面追加

eureka:

client: #客户端注册进eureka服务列表内

service-url:

defaultZone: <http://localhost:7001/eureka>

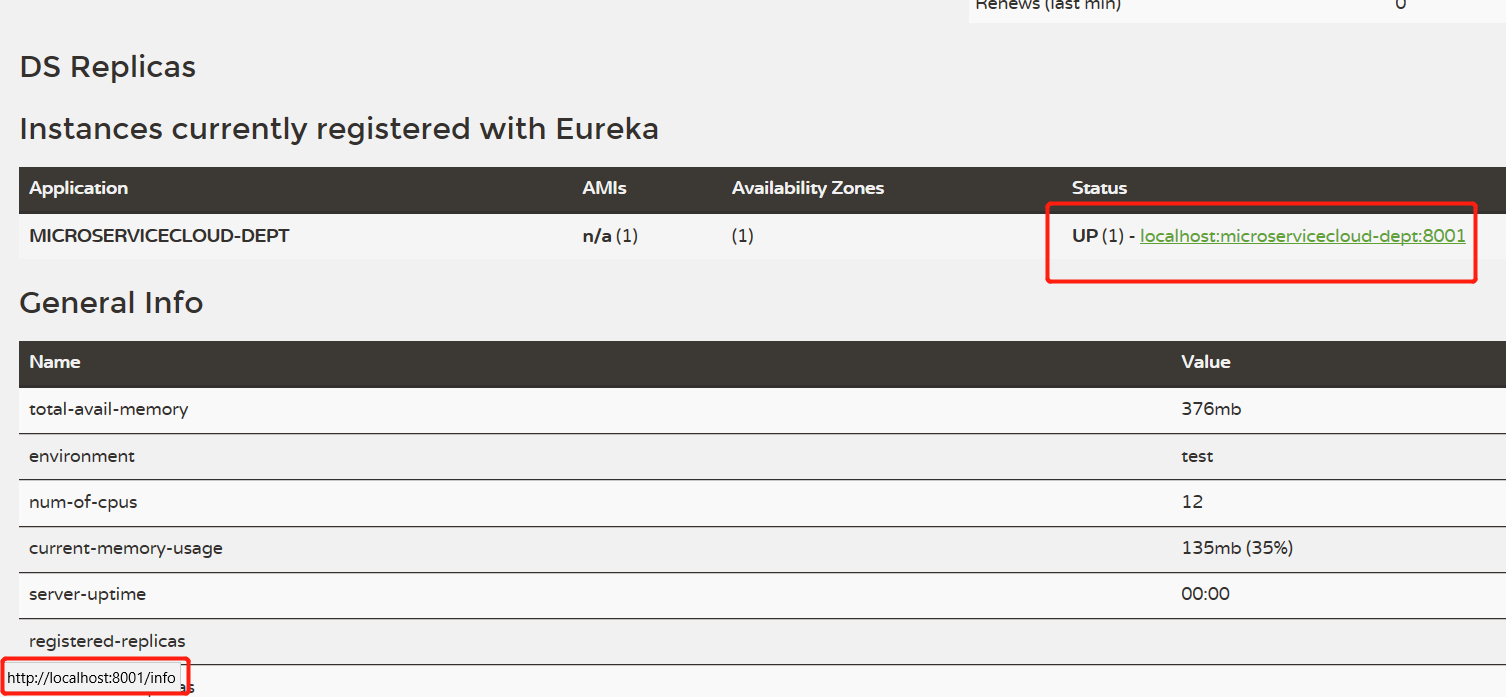
3.修改8001启动类，添加注解

@EnableEurekaClient//表明是Eureka客户端

Ps：@EnableEurekaServer //是Eureka的服务端

4.测试<http://localhost:7001/>

就可以看到服务名了。

第四节：服务注册信息完善， 

主机名称修改名称、访问信息有ip显示：

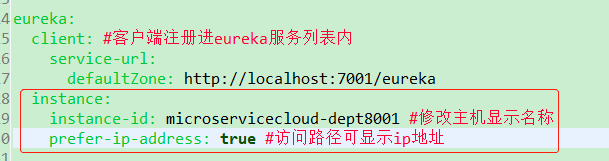
如在8001 yml文件中添加信息：

instance:

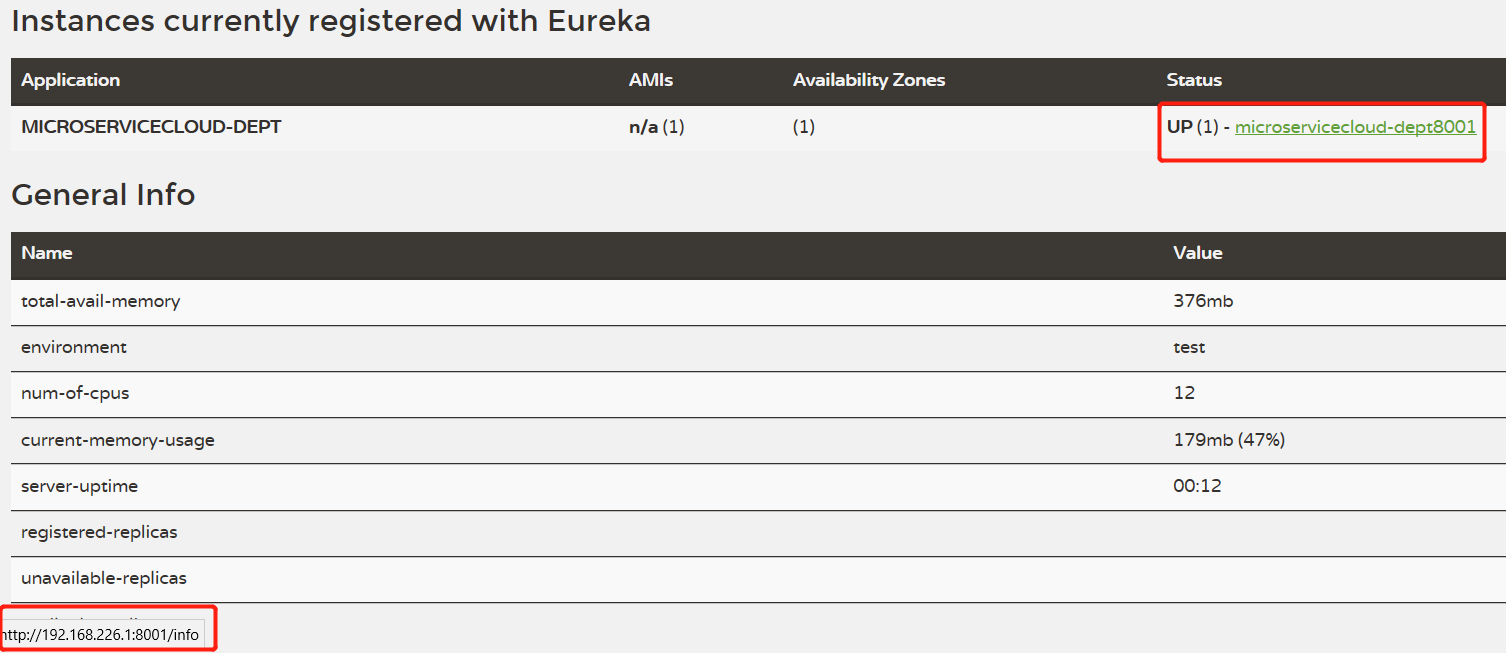
instance-id: microservicecloud-dept8001 #修改主机显示名称

prefer-ip-address: **true** #访问路径可显示ip地址

即：



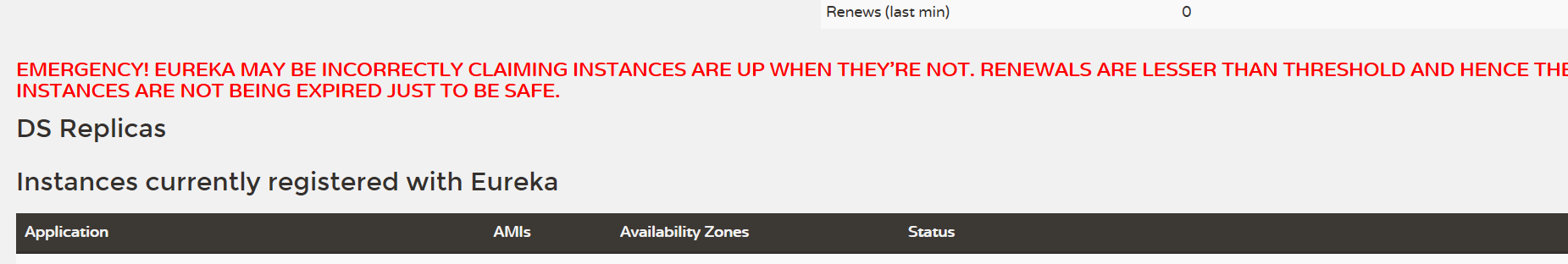
再刷新服务端7001



：

Eureka的自我保护机制

如：故障现象：



2.修改8001 pom文件，添加：（已经添加）

<!-- actuator监控信息完善 -->

<dependency>

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

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

</dependency>

3.给父工程添加信息，可以在Eureka页面查看接口信息，pom文件追加

<build>

<finalName>microservicecloud</finalName>

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<delimiters>

<delimit>$</delimit>

</delimiters>

</configuration>

</plugin>

</plugins>

</build>

4.再给8001 yml文件追加信息：

info:

app.name: atguigu-microservicecloud

company.name: www.atguigu.com

build.artifactId: $project.artifactId$

build.version: $project.version$

5.测试

启动

C:\Users\dada\AppData\Local\Temp\1552659964(1).png

<http://localhost:7001/>

点击微服务，查看info信息

第五节 Eureka自我保护

一句话：某时刻某一个微服务不可用了，Eureka不会立刻清理，依旧会对该微服务的消息进行保存。

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，Eureka Server将会注销实例（默认90秒）。但是当网络分区故障发生时，微服务于EurekaServer之间无法正常通信，以上行为可能变得非常危险了—因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题—当Eureka Server节点短时间内丢失过多客户端时（可能发送了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，EurekaServer就会保护服务注册表中的消息，不再删除服务注册表的数据（也就是不会注销任何微服务）。当网络故障恢复后，该EurekaServer节点会自动退出自我保护模式。

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

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加健壮、稳定！

在SpringCloud中，可以使用

eureka.server.enable-self-preservation=false 禁用自我保护模式

第六节 Eureka服务发现

第七节 Eureka的集群配置

1.原理：把功能多部署几份应当高并发的请求提供服务。

2.搭建eureka-7002，eureka-7003，

复制eureka-7001的pom文件、启动类、yml文件，改相应的名称；

修改C:\Windows\System32\drivers\etc 下hosts文件映射：

添加

127.0.0.1 eureka7001.com

127.0.0.1 eureka7002.com

127.0.0.1 eureka7003.com

注意相应yml文件也要改：如7001的，注意最后一行改为其他两个eureka地址

server:

port: 7001

eureka:

instance:

hostname: eureka7001.com #eureka服务端的实例名称

client:

register-with-eureka: **false** #false表示不向注册中心注册自己。

fetch-registry: **false** #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务

service-url:

#defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址（单机）。

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

============================

修改dept-8001 yml文件，

eureka:

client: #客户端注册进eureka服务列表内

service-url:

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

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

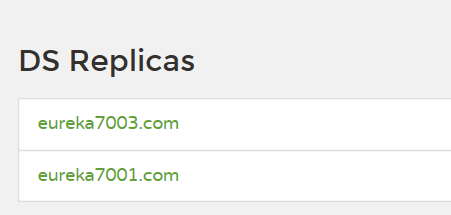
启动7001 、7002、7003、8001测试：

<http://eureka7002.com:7002/>

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

http://eureka7003.com:7003/

可以看到集群信息：



第八节：作为服务注册中心 Eureka比zookeeper好在哪？

1.

Ps：CAP原则，只能满足其二，

C：Consistency强一致性

A：Availability可用性

P：Partition tolerance 分区容错性

一个分布式系统不可能同时满足CAP，由于分区容错性p是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。

Eureka 遵守AP原则

Zookeeper遵守CP原则

Zookeeper保证CP：

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用，也就是说，无法注册功能对可用性的要求高于一致性。但是zk会出现这样的一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长了，30-120s，且选举期间整个zk集群是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

Eureka保证AP

先保证可用性，Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时如果发生连接失败，则会自动切换至其他节点，只要有一台Eureka还在，就能保证注册服务可用，只不过查询到的信息可能部署最新的（不保证强一致性）。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

>Eureka不再从注册列表中移除因为长时间没有收到心跳而应该过期服务；

>Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点依然可用）；

>当网络稳定时，当前实例新的注册信息会被同步到其他节点中。

结论：**Eureka可用很好的应对网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪！**

# 第六章 Ribbon负载均衡

第一节 概念

基于NetFlix Ribbon实现的一套客户端 负载均衡的工具。

简单的说，Ribbon是NetFlix开发的开源项目，主要功能是提供客户端的软件负载均衡算法，将NetFlix的中间层服务连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer（LB：即负载均衡，在微服务或分布式集群中经常用的一种应用。负载均衡简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA。）后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。

第二节 Ribbon的配置初步

1.修改consumer-dept-80 的pom文件添加信息

Ribbon需要跟Eureka整合

<!-- Ribbon相关 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

2.修改consumer-dept-80 yml，添加信息

eureka:

client: #客户端注册进eureka服务列表内

register-with-eureka: **false** #排除自己不注册

service-url:

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

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

3.在consumer-dept-80工程的ConfigBean类 getRestTemplate方法添加注解

@LoadBalanced

4.在consumer-dept-80 启动类添加注解@EnableEurekaClient

5. consumer-dept-80 修改DeptController\_Consumer

微服务名称（这样才符合规定）：

private static final String REST\_URL\_PREFIX = "http://MICROSERVICECLOUD-DEPT";

6. 测试

先启动三个Eureka7001 7002 7003集群后，再启动 provide-dept-8001并注册进eureka

再启动consumer-dept-80

http://localhost/consumer/dept/get/1

http://eureka7001.com:7001/

7.小结：

Ribbon和Eureka整合后Consumer可用直接调用服务而不用关心地址和端口号。

**第三节 Ribbon的负载均衡算法**

1.架构策略

Ribbon工作分为两步：

第一步先选择Eureka Server，它优先选择在同一个区域负载较少的server；

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

其中Ribbon提供了多种策略：比如轮询、随机和根据响应时间加权。

2.拷贝两份microservicecloud-provider-dept-8001 的pom

新建子项目为8002 8003

复制pom文件依赖 、各个类、配置文件等，修改相应的端口

创建各自的数据库，

如8002脚本

DROP DATABASE IF EXISTS cloudDB02;

CREATE DATABASE cloudDB02 CHARACTER SET UTF8;

USE cloudDB02;

CREATE TABLE dept(

deptno BIGINT NOT NULL PRIMARY KEY AUTO\_INCREMENT,

dname VARCHAR(60),

db\_source VARCHAR(60));

INSERT INTO dept(dname,db\_source)VALUES('开发部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('人事部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('财务部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('市场部',DATABASE());

INSERT INTO dept(dname,db\_source)VALUES('运维部',DATABASE());

SELECT \* FROM dept;

8003脚本类似。

3.修改yml文件的端口、连接的数据库、主机显示名称，

注意服务名称三个都是统一一样的，不能改！

4.启动测试

依次启动Eureka 7001 7002 7003

再启动3个Dept-8001 8002 8003，各自测试

localhost:8001/dept/listlocalhost:8002/dept/listlocalhost:8003/dept/list

再启动80，测试负载均衡、轮询策略

localhost/consumer/dept/list

结果显示会不断切换访问各自数据库信息

5.结论：

Ribbon其实就是一个软负载均衡的客户端组件，他可以和其他所需请求的客户端结合使用，和Eureka结合只是其中的一个实例。

第四节： Ribbon负载均衡 核心组件算法IRule

在dept-80子工程 修改ConfigBean类，添加自定义方法

//如果自定义了负载均衡策略，就会覆盖默认的策略（默认是轮询）

@Bean

**public** IRule myRule(){

**return** **new** RandomRule();//随机算法

// **return** **new** RetryRule();//重试

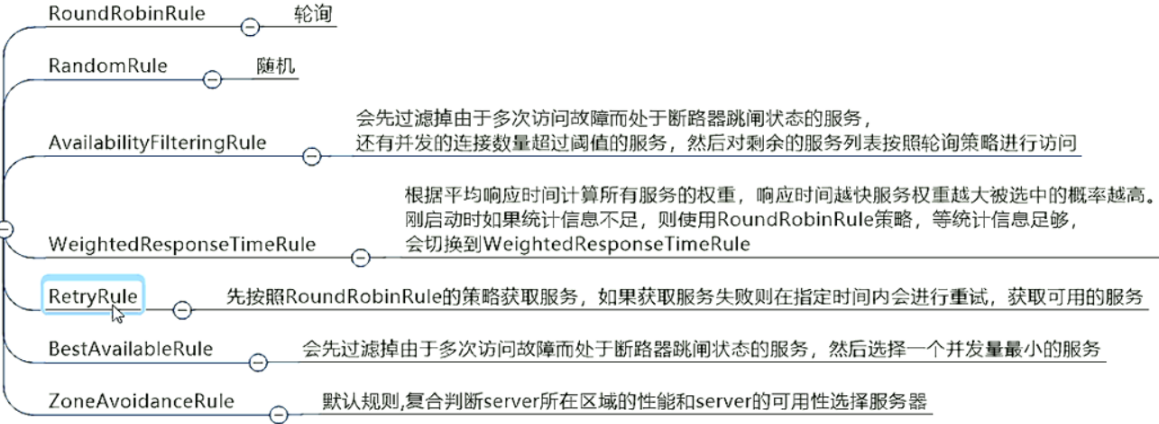
}

再重启 dept8001 8002 8003 ,80

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

会不规则访问

还可以试试其他算法：



第五节 自定义Ribbon算法

1.修改dept-80项目启动类添加注解

@RibbonClient(name="microservicecloud-dept",configuration=MySelfRule.**class**)

官方文档警告：

这个自定义配置类不能放在@ComponentScan所扫描的当前包及子包下，否则我们自定义配置类就会被所有的Ribbon客户端所共享，也就是说我们达不到特殊化定制的目的了。

在com.atguigu.myrule下新建类：

@Configuration

**public** **class** MySelfRule {

@Bean

**public** IRule myRule() {

**return** **new** RandomRule();//Ribbon默认是轮询，我自定义随机

}

}

然后重启7001 7002 7003 8001 8002 8003 80服务，测试效果

2.修改Ribbon默认算法源码，自定义负载均衡的算法。

Dept-80添加类RandomRule\_ZY

package com.atguigu.myrule;

import java.util.List;

import com.netflix.client.config.IClientConfig;

import com.netflix.loadbalancer.AbstractLoadBalancerRule;

import com.netflix.loadbalancer.ILoadBalancer;

import com.netflix.loadbalancer.Server;

public class RandomRule\_ZY extends AbstractLoadBalancerRule

{

// total = 0 // 当total==5以后，我们指针才能往下走，

// index = 0 // 当前对外提供服务的服务器地址，

// total需要重新置为零，但是已经达到过一个5次，我们的index = 1

// 分析：我们5次，但是微服务只有8001 8002 8003 三台，OK？

//

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

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

public Server choose(ILoadBalancer lb, Object key)

{

if (lb == null) {

return null;

}

Server server = null;

while (server == null) {

if (Thread.interrupted()) {

return null;

}

List<Server> upList = lb.getReachableServers();

List<Server> allList = lb.getAllServers();

int serverCount = allList.size();

if (serverCount == 0) {

/\*

\* No servers. End regardless of pass, because subsequent passes only get more

\* restrictive.

\*/

return null;

}

// int index = rand.nextInt(serverCount);// java.util.Random().nextInt(3);

// server = upList.get(index);

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

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

if(total < 5)

{

server = upList.get(currentIndex);

total++;

}else {

total = 0;

currentIndex++;

if(currentIndex >= upList.size())

{

currentIndex = 0;

}

}

if (server == null) {

/\*

\* The only time this should happen is if the server list were somehow trimmed.

\* This is a transient condition. Retry after yielding.

\*/

Thread.yield();

continue;

}

if (server.isAlive()) {

return (server);

}

// Shouldn't actually happen.. but must be transient or a bug.

server = null;

Thread.yield();

}

return server;

}

@Override

public Server choose(Object key)

{

return choose(getLoadBalancer(), key);

}

@Override

public void initWithNiwsConfig(IClientConfig clientConfig)

{

// TODO Auto-generated method stub

}

}

在修改ConfigBean类方法

@Bean

**public** IRule myRule(){

//return new RandomRule();//随机算法

//return new RetryRule();

**return** **new** RandomRule\_ZY();//自定义的算法 每个微服务询问五次 跳动

}

重启服务测试:

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

# 第七章 Feign负载均衡

第一节：概念

**Feign是一个声明式的Web服务客户端，**使得编写Web服务客户端变得非常容易，**只需要创建一个接口，然后上面添加注解即可。**

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

Feign能干什么？

Feign旨在使编写Java http客户端变得更容易。

前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成一套模板化的调用方法。但是在时间开发中，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，所以通常会针对每个微服务自行进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封装，由他来帮助我门定义和实现依赖服务接口的定义。在feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它（以前是Dao接口上面标注Mapper注解，现在是一个微服务**接口上面标注一个feign注解即可**），即可完成对服务提供方的接口绑定，简化了使用SpringCloud Ribbon时，自动封装服务调用客户端的开发量。

**Ribbo 与feign调用服务的区别：**

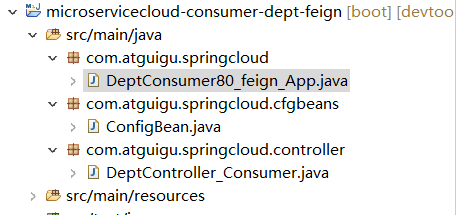
Ribbon：微服务名称+地址

Feign：接口+注解

第二节 Springcloud-feign的构建

1.复制microservicecloud-consumer-dept-80 相关文件，修改相应配置

新建子工程：microservicecloud-consumer-dept-feign



2.修改microservicecloud-api的pom文件，添加接口及方法

Pom文件追加依赖

<dependency>

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

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

</dependency>

---------------------

接口：

package com.atguigu.springcloud.service;

import java.util.List;

import org.springframework.cloud.netflix.feign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import com.atguigu.springcloud.entities.Dept;

/\*\*

\*

\* @Description: 修改microservicecloud-api工程，根据已经有的DeptClientService接口

新建

一个实现了FallbackFactory接口的类DeptClientServiceFallbackFactory

\* @author zzyy

\* @date 2018年4月21日

\*/

//@FeignClient(value = "MICROSERVICECLOUD-DEPT")

@FeignClient(value = "MICROSERVICECLOUD-DEPT")

public interface DeptClientService

{

@RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)

public Dept get(@PathVariable("id") long id);

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

public List<Dept> list();

@RequestMapping(value = "/dept/add", method = RequestMethod.POST)

public boolean add(Dept dept);

}

Api改动过，需要重新maven clean、install

3.dept-feign工程修改Controller，添加上一步新增的DeptClientService接口

@RestController

**public** **class** DeptController\_Consumer {

@Autowired

**private** DeptClientService service;

@RequestMapping(value = "/consumer/dept/get/{id}")

**public** Dept get(@PathVariable("id") Long id) {

**return** **this**.service.get(id);

}

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

**public** List<Dept> list() {

**return** **this**.service.list();

}

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

**public** Object add(Dept dept) {

**return** **this**.service.add(dept);

}

}

4.在dept-feign启动类中添加注解

@EnableFeignClients(basePackages= {"com.atguigu.springcloud"})

@ComponentScan("com.atguigu.springcloud")

5.测试

依次启动：7001 7002 7003 8001 8002 8003 dept-feign服务

http://localhost/consumer/dept/list

依次轮询访问

6.小结：

两者区别：

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

改请求发送给Eureka服务器（<http://MICROSERVICECLOUD-DEPT/dept/list>）

通过Feign直接找到服务接口，由于在进行服务调用的时候融合了Rinbbon技术，所以也支持负载均衡技术。

# 第八章 Hystrix断路器

第一节 .目前分布式系统存在的问题

服务雪崩

多个微服务a b c直接调用的时候，形成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钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，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。

2.概念：

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，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。

3.服务熔断

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回‘错误’的响应信息。当检查到该节点微服务调用响应正常后恢复调用链路。在spring Cloud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

第二节：

3.1.创建子工程microservicecloud-provider-dept-hystrix-8001，

仿microservicecloud-provider-dept-8001项目。

Pom文件追加依赖

<!-- hystrix -->

<dependency>

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

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

</dependency>

修改yml

instance:

instance-id: microservicecloud-dept8001-hystrix

prefer-ip-address: **true**

复制resources下文件 ，复制com包下代码，并修改DeptController

package com.atguigu.springcloud.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import com.atguigu.springcloud.entities.Dept;

import com.atguigu.springcloud.service.DeptService;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@RestController

public class DeptController

{

@Autowired

private DeptService service = null;

@RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)

//一旦调用服务方法失败并抛出了错误信息后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中的指定方法

@HystrixCommand(fallbackMethod = "processHystrix\_Get")

public Dept get(@PathVariable("id") Long id)

{

Dept dept = this.service.get(id);

if (null == dept) {

throw new RuntimeException("该ID：" + id + "没有没有对应的信息");

}

return dept;

}

public Dept processHystrix\_Get(@PathVariable("id") Long id)

{

return new Dept().setDeptno(id).setDname("该ID：" + id + "没有没有对应的信息,null--@HystrixCommand")

.setDb\_source("no this database in MySQL");

}

}

-------------------

Hystrix-8001启动类添加熔断注解

@EnableCircuitBreaker//对hystrix熔断机制支持

@EnableDiscoveryClient//服务发现

3.2.测试

启动7001 7002 7003 hystrix-8001 dept-80

当访问数据库没有的信息时会进熔断

http://localhost/consumer/dept/get/11

3.3.小结：

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法无法处理的异常会进行服务的降级，进而熔断该节点微服务的调用，快速返回‘错误’的响应信息。

第三节 服务降级

1.概念：

整体资源块不够了，忍痛将某些服务先关掉，度过难关，再开启回来。

服务的降级处理是在客户端实现完成的，与服务端没有关系。

2.修改api工程

根据已经有的DeptClientService接口新建一个实现了FallBackFactory接口的类DeptClientServiceFallbackFactory

2.1新建类

**package** com.atguigu.springcloud.service;

**import** java.util.List;

**import** org.springframework.stereotype.Component;

**import** com.atguigu.springcloud.entities.Dept;

**import** feign.hystrix.FallbackFactory;

@Component //不要忘记加

**public** **class** DeptClientServiceFallbackFactory **implements** FallbackFactory<DeptClientService> {

@Override

**public** DeptClientService create(Throwable cause) {

**return** **new** DeptClientService() {

@Override

**public** Dept get(**long** id) {

**return** **new** Dept().setDeptno(id).setDname("该ID：" + id + "没有没有对应的信息,Consumer客户端提供的降级信息,此刻服务Provider已经关闭")

.setDb\_source("no this database in MySQL");

}

@Override

**public** List<Dept> list() {

**return** **null**;

}

@Override

**public** **boolean** add(Dept dept) {

**return** **false**;

}

};

}

}

2.2修改**interface** DeptClientService

添加回调函数

@FeignClient(value = "MICROSERVICECLOUD-DEPT",

fallbackFactory=DeptClientServiceFallbackFactory.**class**)

打包api

3.2修改dept-feign 的yml文件，追加熔断降级开关

feign:

hystrix:

enabled: true

3.3启动测试

启动3个Eureka dept-8001 dept-feign

<http://localhost/consumer/dept/get/2>

故意关闭dept-8001

再测试

第四节 服务熔断与服务降级区别

服务熔断：一般是某个服务故障或者异常引起，类似现实世界的中的保险丝，当某个异常条件被触发，直接熔断整个服务，而不是一直等到此服务超时！

服务降级：一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然服务水平下降了，但好歹可用，比直接挂掉要强。

第五节 服务监控hystrixDashboard

除了隔离依赖服务的调用以外，Hystrix还提供了准实时的调用监控HystrixDashboard，Hystrix会持续地记录所有通过NetFlix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。SpringCloud也提供了HystrixDashboard的整合对监控内容转化成可视化界面。

1.新建microservicecloud-consumer-hystrix-dashboard子工程

Pom文件：

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-consumer-hystrix-dashboard</artifactId>

<dependencies>

<!-- 自己定义的api -->

<dependency>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

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

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

</dependency>

<!-- 修改后立即生效，热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<!-- Ribbon相关 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- feign相关 -->

<dependency>

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

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

</dependencies>

</project>

Yml文件application.yml

server:

port: 9001

新建启动类

package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

@SpringBootApplication

@EnableHystrixDashboard

public class DeptConsumer\_DashBoard\_App

{

public static void main(String[] args)

{

SpringApplication.run(DeptConsumer\_DashBoard\_App.class, args);

}

}

所有provider微服务提供类dept-8001 8002 8003 都需要依赖配置：

<!-- actuator监控信息完善 -->

<dependency>

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

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

</dependency>

2.测试

启动dashboard服务，

<http://localhost:9001/hystrix>

看到logo即正常

再启动三个Eureka集群

再启动hystrix-8001

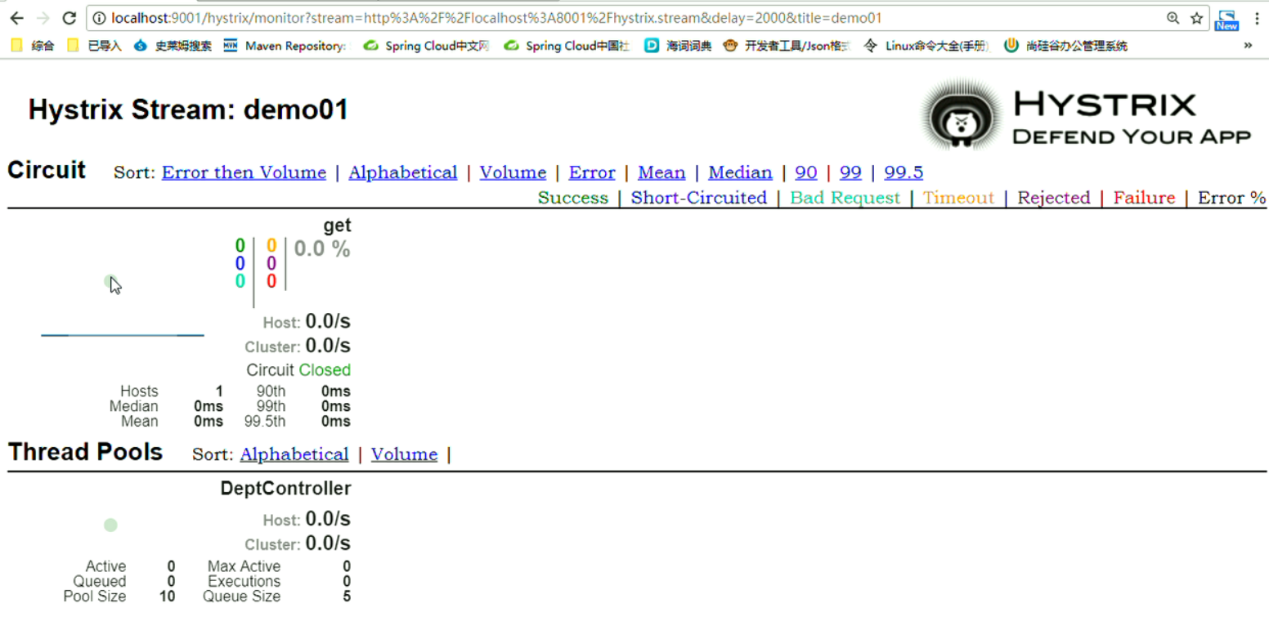
再重启dashboard

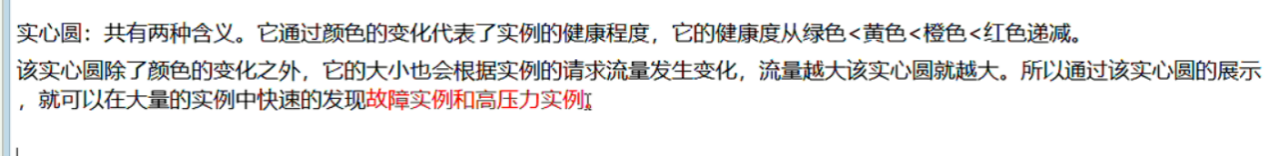
测试：

<http://localhost:8001/dept/get/1> 多刷新几次，查看监控图

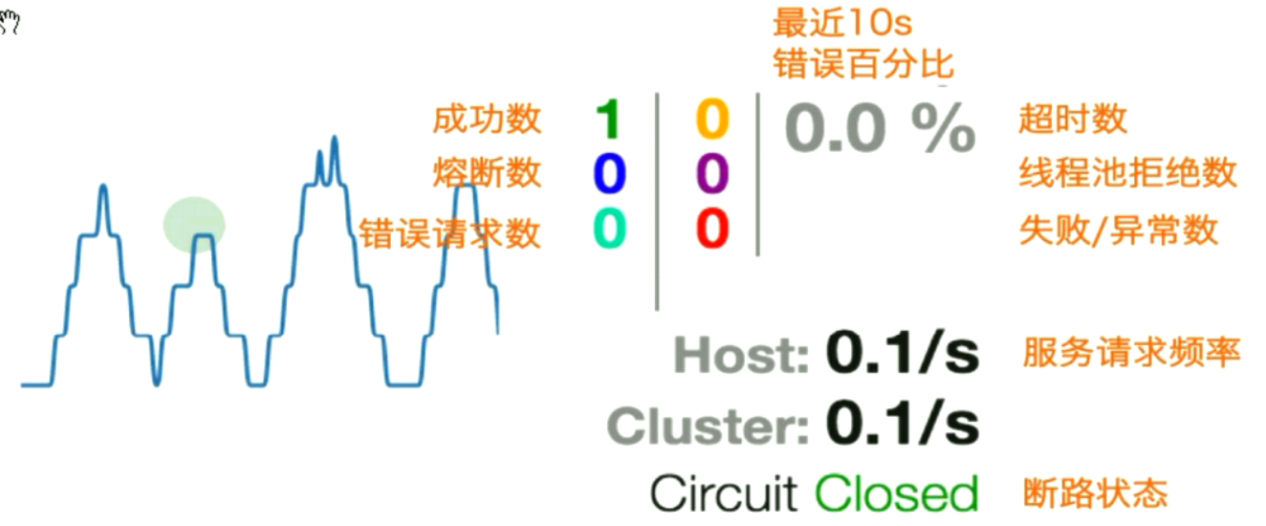
http://localhost:8001/hystrix.stream







C:\Users\dada\AppData\Local\Temp\1552735382(1).png



# 第九章 路由网关zuul

第一节：概述

Zuul包含了对请求的路由和过滤两个最主要的功能：

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础而过滤器功能则复制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

第二节：路由的基本配置

1.创建子工程：microservicecloud-zuul-gateway-9527

修改pom文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-zuul-gateway-9527</artifactId>

<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>

<!-- actuator监控 -->

<dependency>

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

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

</dependency>

<!-- hystrix容错 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- 日常标配 -->

<dependency>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- 热部署插件 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

</project>

Yml文件application.yml

server:

port: 9527

spring:

application:

name: microservicecloud-zuul-gateway

eureka:

client:

service-url:

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

instance:

instance-id: gateway-9527.com

prefer-ip-address: **true**

info:

app.name: atguigu-microcloud

company.name: www.atguigu.com

build.artifactId: $project.artifactId$

build.version: $project.version$

启动类

SpringBootApplication

@EnableZuulProxy

**public** **class** Zuul\_9527\_StartSpringCloudApp {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Zuul\_9527\_StartSpringCloudApp.**class**, args);

}

}

修改hosts文件

127.0.0.1 myzuul.com

3.启动 三个eureka集群 dept-8001 9527

4.测试

不用路由

<http://localhost:8001/dept/get/2>

用路由

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

第三节 路由映射规则

修改yml文件

添加

zuul:

routes:

mydept.serviceId: microservicecloud-dept

mydept.path: /mydept/\*\*

解析：



测试：

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

问题：修改前的网址还可以访问！

再把刚才yml文件内容改为：

zuul:

ignored-services: microservicecloud-dept

routes:

mydept.serviceId: microservicecloud-dept

mydept.path: /mydept/\*\*

测试：

这时原<http://myzuul.com:9527/microservicecloud-dept/dept/get/2>

就不能访问了！

问题：这样改只能禁一个microservicecloud-dept

改进，禁用批量的，再测试

zuul:

ignored-services: "\*"

routes:

mydept.serviceId: microservicecloud-dept

mydept.path: /mydept/\*\*

改进，在所有访问路径前面添加前缀

把yml文件改为

zuul:

prefix: /atguigu

ignored-services: "\*"

routes:

mydept.serviceId: microservicecloud-dept

mydept.path: /mydept/\*\*

再测试：myzuul.com:9527/atguigu/mydept/dept/get/1

# 第十章 SpringCloudConfig分布式配置中心

第一节 分布式系统面临的问题

配置问题

微服务意味着要将单体应用中的业务分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务，由于单个服务都需要必要的配置消息才能运行，所以一套集中式的、动态的配置管理设施是必不可少的。Springcloud提供了configServer来解决这个问题，我们每一个微服务自己带着一个application.yml,上百个呢？

概念：

是什么

Springcloud Config 为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

怎么玩

分为服务端和客户端

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未客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息配置服务默认采用git来存储配置信息，这样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 的管理和访问配置内容。

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；

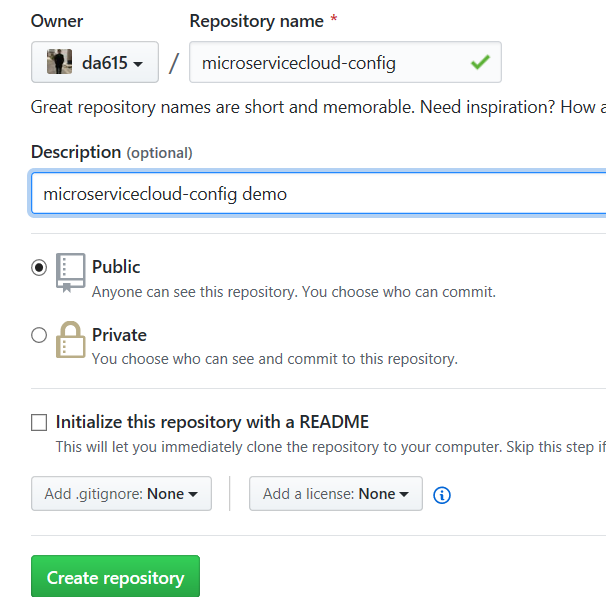
当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置；

将配置信息以rest接口形式暴露；

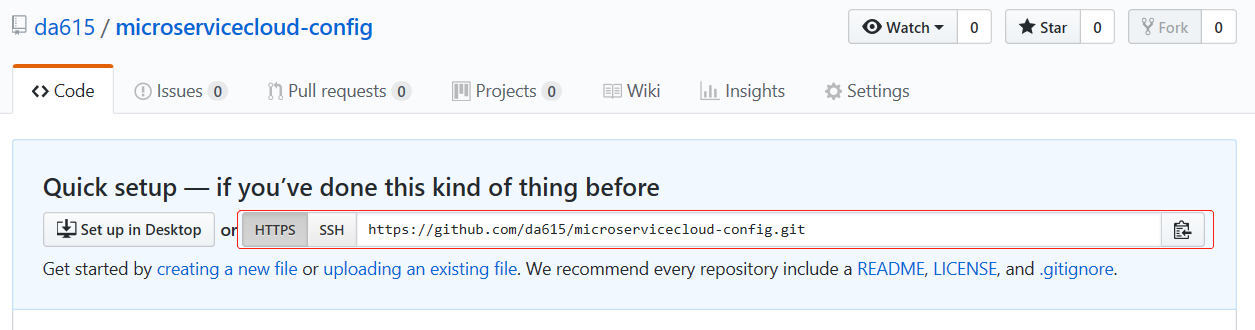
于github整合配置，而且使用的是http/https访问形式

第二节 spring Cloud 的github配置

现在github新建repository



复制远程库地址：



Clone到本地库

关联

>git clone https://github.com/da615/microservicecloud-config.git

在本地路径下新建application.yml,且必须以UTF-8格式保存！！！





创建好文件后，上传

>git status

>git add .

>git commit -m “init file”

>git push origin master

第三节 创建子工程

microservicecloud-config-3344

pom文件

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.atguigu.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-config-3344</artifactId>

<dependencies>

<!-- springCloud Config -->

<dependency>

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

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

</dependency>

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

<dependency>

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

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

<version>4.10.0.201712302008-r</version>

</dependency>

<!-- 图形化监控 -->

<dependency>

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

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

</dependency>

<!-- 熔断 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- 热部署插件 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

</project>

Application.yml文件

server:

port: 3344

spring:

application:

name: microservicecloud-config

cloud:

config:

server:

git:

uri: https://github.com/da615/microservicecloud-config.git

#GitHub上面的git仓库名字

启动类

package com.atguigu.springcloud;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class Config\_3344\_StartSpringCloudApp {

public static void main(String[] args) {

SpringApplication.run(Config\_3344\_StartSpringCloudApp.class, args);

}

}

Hosts文件映射

127.0.0.1 config-3344.com

启动3344，测试：

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

第四节：创建子工程3355

Ps：application.yml是用户级的资源配置项

Bootstrap.yml是系统级的，优先级更高，负责从外部源加载配置属性并解析配置。

# 第十一章SpringCloud Sleuth

微服务分布式链路追踪

第一节概念：

由于微服务是一个分布式架构，服务多，业务复杂，如果出现了错误和异常，很难去定位。主要体现在，一个请求可能需要调用很多个服务，而内部服务的调用复杂性，决定了问题难于定位。所以微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎么样的，从而达到每个请求的步骤清晰可见，出了问题，很快定位。

Springcloud中集成了Zipkin。在SpringCloud中集成Zipkin非常简单，只需加入响应的依赖和做相关的配置即可。

项目地址参考：<https://www.cnblogs.com/xingzc/p/9413860.html>