[Spring Cloud简介 1](#_Toc721)

[微服务架构 1](#_Toc5545)

[一、服务注册与发现（Eureka Server & Service A/B) 3](#_Toc30884)

[(1)创建服务注册中心(Eureka Server) 3](#_Toc21402)

[(2)创建服务提供方 5](#_Toc29618)

[二、实现负载均衡（Ribbon/Feign） 9](#_Toc2670)

[（1）再创建一个Service-B,并修改端口 9](#_Toc864)

[(2)使用Ribbon实现客户端负载均衡的消费者 10](#_Toc14980)

[三、分布式配置中心 12](#_Toc25310)

[(1)构建Config Server 13](#_Toc9585)

[（2）服务端验证 16](#_Toc31745)

[(3)微服务端映射配置 16](#_Toc27112)

[四、服务网关（api-gateway/Zuul) 19](#_Toc9757)

[(1)准备工作 19](#_Toc4014)

[(2)开始使用Zuul 20](#_Toc10399)

[(3)Zuul配置 21](#_Toc3150)

[五、服务间调用 23](#_Toc1902)

## Spring Cloud简介

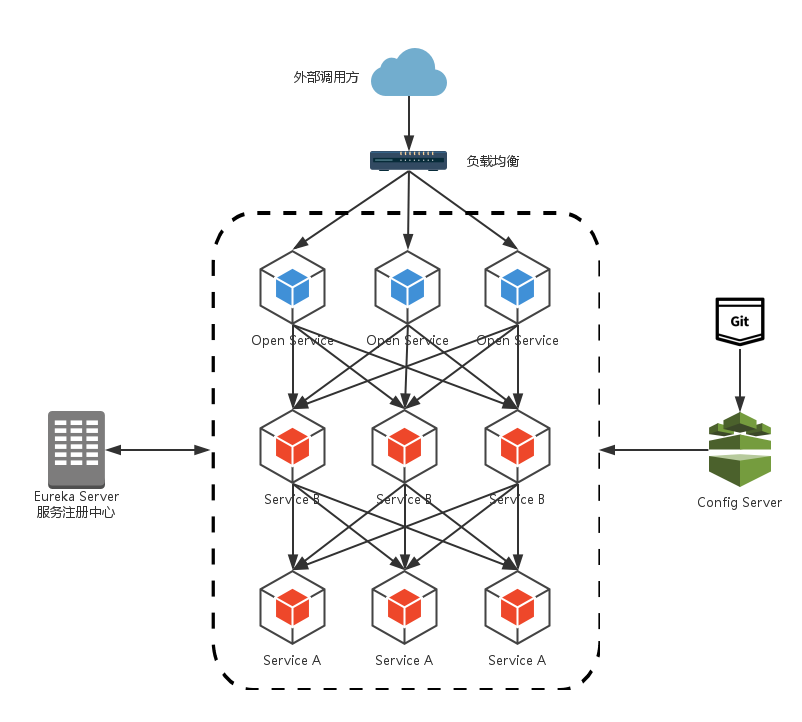
Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

Spring Cloud包含了多个子项目（针对分布式系统中涉及的多个不同开源产品），比如：Spring Cloud Config、Spring Cloud Netflix、Spring Cloud CloudFoundry、Spring Cloud AWS、Spring Cloud Security、Spring Cloud Commons、Spring Cloud Zookeeper、Spring Cloud CLI等项目。

## 微服务架构

“微服务架构”在这几年非常的火热，以至于关于微服务架构相关的产品社区也变得越来越活跃（比如：netflix、dubbo），Spring Cloud也因Spring社区的强大知名度和影响力也被广大架构师与开发者备受关注。

那么什么是“微服务架构”呢？简单的说，微服务架构就是将一个完整的应用从数据存储开始垂直拆分成多个不同的服务，每个服务都能独立部署、独立维护、独立扩展，服务与服务间通过诸如RESTful API的方式互相调用。

最终实现的架构图：  


## 一、服务注册与发现（Eureka Server & Service A/B)

### (1)创建服务注册中心(Eureka Server)

创建一个基础的Spring Boot工程，并在pom.xml中引入需要的依赖内容：

<parent>

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

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

<version>1.3.5.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

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

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常的简单，只需要在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：

package com.didispace;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.builder.SpringApplicationBuilder;

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

@EnableEurekaServer

@SpringBootApplication

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需要在application.properties中问增加如下配置：

server.port=1111

#eureka.instance.hostname=localhost

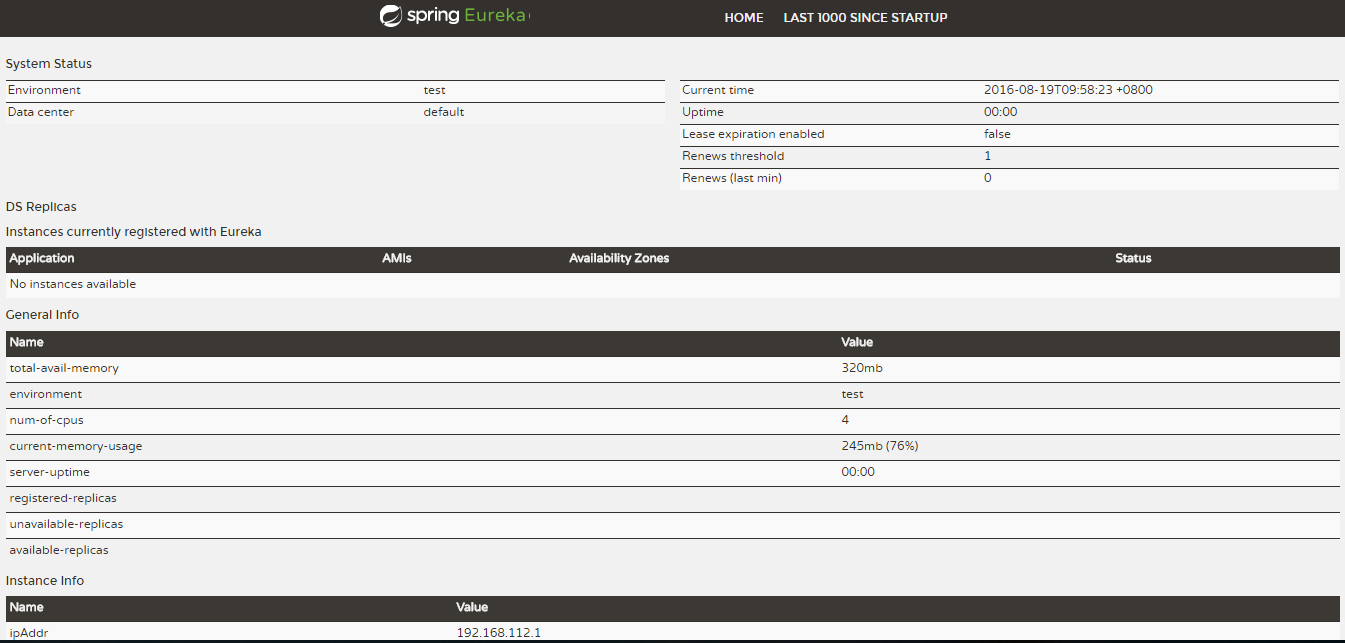
eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/

启动工程后，访问：<http://localhost:1111/>

可以看到下面的页面，其中还没有发现任何服务



### (2)创建服务提供方

下面我们创建提供服务的客户端，并向服务注册中心注册自己。

假设我们有一个提供计算功能的微服务模块，我们实现一个RESTful API，通过传入两个参数a和b，最后返回a + b的结果。

首先，创建一个基本的Spring Boot应用，在pom.xml中，加入如下配置：

<?xml version="1.0" encoding="UTF-8"?>

<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.didispace</groupId>

<artifactId>service-B</artifactId>

<version>1.0.0</version>

<packaging>jar</packaging>

<name>service-B</name>

<description>Spring Cloud project</description>

<parent>

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

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

<version>1.3.5.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

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

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

其次，实现/add请求处理接口，通过DiscoveryClient对象，在日志中打印出服务实例的相关内容。

@RestController

public class ComputeController {

private final Logger logger = Logger.getLogger(getClass());

@Autowired

private DiscoveryClient client;

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

public String add(@RequestParam Integer a, @RequestParam Integer b) {

ServiceInstance instance = client.getLocalServiceInstance();

Integer r = a + b;

logger.info("/add, host:" + instance.getHost() + ", service\_id:" + instance.getServiceId() + ", result:" + r);

return "From Service-B, Result is " + r;

}

}

最后在主类中通过加上@EnableDiscoveryClient注解，该注解能激活Eureka中的DiscoveryClient实现，才能实现Controller中对服务信息的输出。

@EnableDiscoveryClient

@SpringBootApplication

public class ComputeServiceApplication {

public static void main(String[] args) {

new SpringApplicationBuilder(ComputeServiceApplication.class).web(true).run(args);

}

}

在完成了服务内容的实现之后，再继续对application.properties做一些配置工作，具体如下：

spring.application.name=service-B

server.port=3333

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

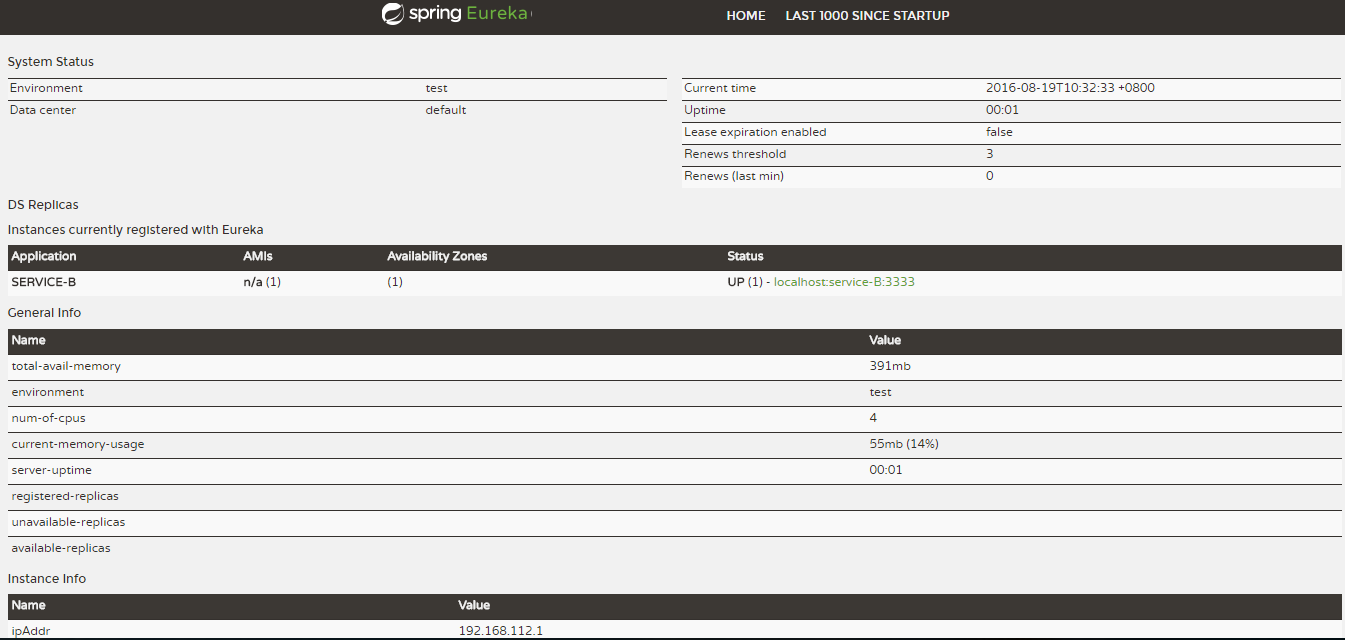
通过spring.application.name属性，我们可以指定微服务的名称后续在调用的时候只需要使用该名称就可以进行服务的访问。

eureka.client.serviceUrl.defaultZone属性对应服务注册中心的配置内容，指定服务注册中心的位置。

为了在本机上测试区分服务提供方和服务注册中心，使用server.port属性设置不同的端口。

启动该工程后，再次访问：<http://localhost:1111/>

可以看到，我们定义的服务被注册了。



## 二、实现负载均衡（Ribbon/Feign）

### （1）再创建一个Service-B,并修改端口

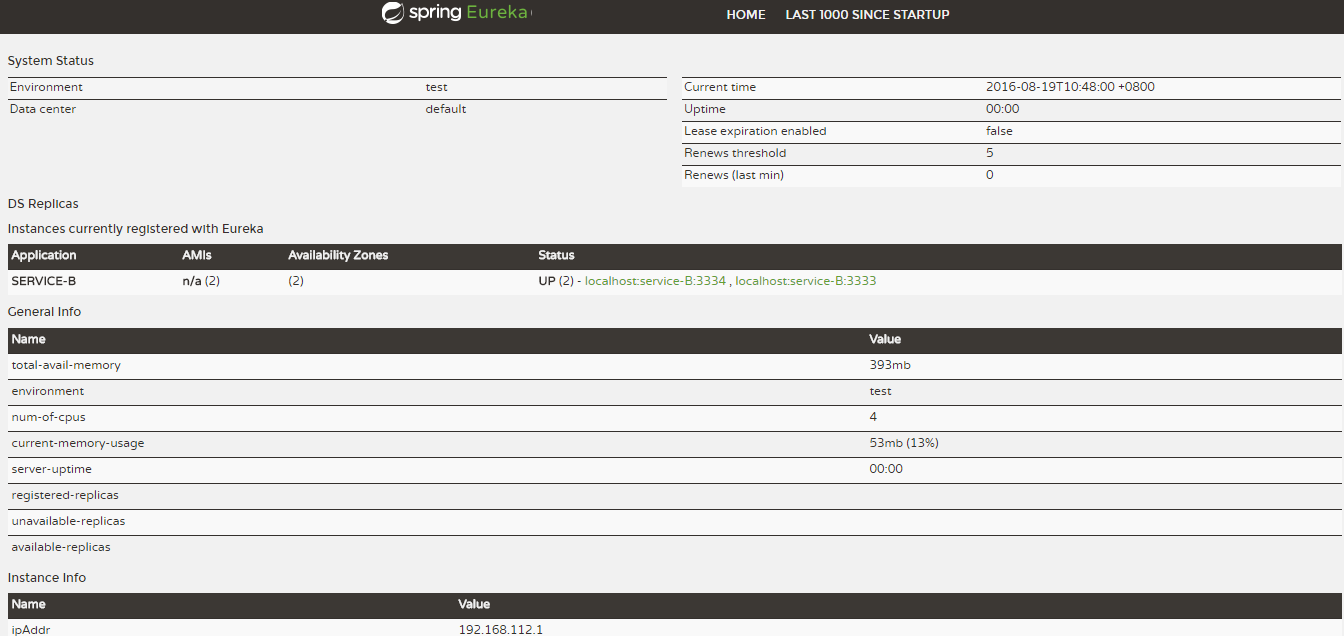
application.properties:

spring.application.name=service-B

server.port=3334

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

启动两个服务，再次访问：<http://localhost:1111/>



可以看到Service-B有两个单元正在运行

### (2)使用Ribbon实现客户端负载均衡的消费者

构建一个基本Spring Boot项目，并在pom.xml中加入如下内容：

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

在应用主类中，通过@EnableDiscoveryClient注解来添加发现服务能力。创建RestTemplate实例，并通过@LoadBalanced注解开启均衡负载能力。

@SpringBootApplication

@EnableDiscoveryClient

public class RibbonApplication {

@Bean

@LoadBalanced

RestTemplate restTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(RibbonApplication.class, args);

}

}

创建ConsumerController来消费SERVICE-B的add服务。通过直接RestTemplate来调用服务，计算10 + 20的值。

@RestController

public class ConsumerController {

@Autowired

RestTemplate restTemplate;

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

public String add() {

return restTemplate.getForEntity("http://SERVICE-B/add?a=10&b=20", String.class).getBody();

}

}

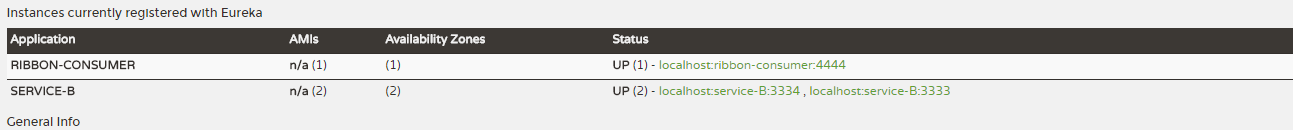
application.properties中配置eureka服务注册中心

spring.application.name=ribbon-consumer

server.port=4444

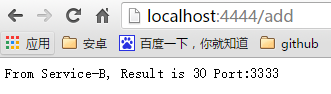
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

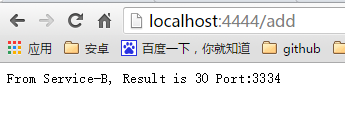
启动Ribbon服务



访问几次[http://localhost:4444/add](http://localhost:3333/add)

可以看到几次访问分别由不同的单元执行的：





## 三、分布式配置中心

Spring Cloud Config为服务端和客户端提供了分布式系统的外部化配置支持。配置服务器为各应用的所有环境提供了一个中心化的外部配置。它实现了对服务端和客户端对Spring Environment和PropertySource抽象的映射，所以它除了适用于Spring构建的应用程序，也可以在任何其他语言运行的应用程序中使用。作为一个应用可以通过部署管道来进行测试或者投入生产，我们可以分别为这些环境创建配置，并且在需要迁移环境的时候获取对应环境的配置来运行。

### (1)构建Config Server

通过Spring Cloud构建一个Config Server，非常简单，只需要三步：

* pom.xml中引入spring-cloud-config-server依赖，完整依赖配置如下：

<parent>

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

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

<version>1.3.5.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<properties>

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

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

* 创建Spring Boot的程序主类，并添加@EnableConfigServer注解，开启Config Server

@EnableConfigServer

@SpringBootApplication

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

application.properties中配置服务信息

spring.application.name=config-server

server.port=7001

# git远程配置

#spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringBoot-Learning/

#spring.cloud.config.server.git.searchPaths=Chapter9-1-4/config-repo

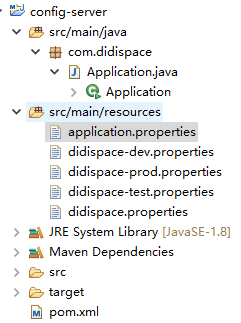
#spring.cloud.config.server.git.username=username

#spring.cloud.config.server.git.password=password

# 激活本地配置

spring.profiles.active=native

可以使用git远程配置或本地配置，这里使用本地配置，文件目录结构：



* spring.cloud.config.server.git.uri：配置git仓库位置
* spring.cloud.config.server.git.searchPaths：配置仓库路径下的相对搜索位置，可以配置多个
* spring.cloud.config.server.git.username：访问git仓库的用户名
* spring.cloud.config.server.git.password：访问git仓库的用户密码

到这里，使用一个通过Spring Cloud Config实现，并使用git管理内容的配置中心已经完成了，启动该应用，成功后开始下面的内容。

Spring Cloud Config也提供本地存储配置的方式。我们只需要设置属性spring.profiles.active=native**，**Config Server会默认从应用的src/main/resource目录下检索配置文件。也可以通过spring.cloud.config.server.native.searchLocations=file:F:/properties/属性来指定配置文件的位置。

### （2）服务端验证

给每个properties文件配置一个测试属性，如didispace-dev.properties

from=local-dev

完成了这些准备工作之后，我们就可以通过浏览器或POSTMAN等工具直接来访问到我们的配置内容了。

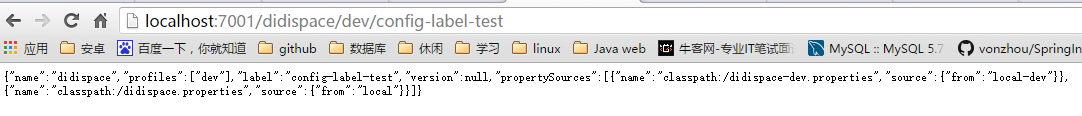
URL与配置文件的映射关系如下：

* /{application}/{profile}[/{label}]
* /{application}-{profile}.yml
* /{label}/{application}-{profile}.yml
* /{application}-{profile}.properties
* /{label}/{application}-{profile}.properties

上面的url会映射{application}-{profile}.properties对应的配置文件，{label}对应git上不同的分支，默认为master。

启动应用后访问<http://localhost:7001/didispace/dev/config-label-test>

可以看到JSON数据：



### (3)微服务端映射配置

在完成并验证了配置服务中心之后，下面看看我们如何在微服务应用中获取配置信息。

* 创建一个Spring Boot应用，在pom.xml中引入spring-cloud-starter-config依赖，完整依赖关系如下：

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

* 创建最基本的Spring Boot启动主类

@SpringBootApplication

public class ComputeServiceApplication {

public static void main(String[] args) {

new SpringApplicationBuilder(ComputeServiceApplication.class).web(true).run(args);

}

}

* 创建bootstrap.properties配置，来指定config server，例如：

spring.application.name= didispace

spring.cloud.config.profile=dev

spring.cloud.config.label=master

spring.cloud.config.uri=http://localhost:7001/

server.port=2222

* spring.application.name：对应前配置文件中的{application}部分
* spring.cloud.config.profile：对应前配置文件中的{profile}部分
* spring.cloud.config.label：对应前配置文件的git分支
* spring.cloud.config.uri：配置中心的地址

这里需要格外注意：上面这些属性必须配置在bootstrap.properties中，config部分内容才能被正确加载。因为config的相关配置会先于application.properties**，**而bootstrap.properties的加载也是先于application.properties**。**

@RefreshScope

@RestController

class TestController {

@Value("${from}")

private String from;

@RequestMapping("/from")

public String from() {

return this.from;

}

public void setFrom(String from) {

this.from = from;

}

public String getFrom() {

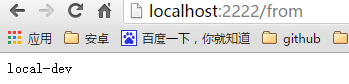
return from;

}

}

* ：

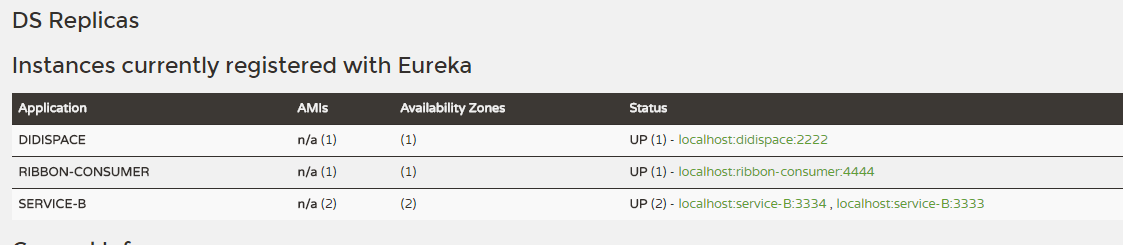
启动该应用，并访问：[http://localhost:2222/from](http://localhost:7002/from) ，我们就可以根据配置内容输出对应环境的from内容了。



## 四、服务网关（api-gateway/Zuul)

### (1)准备工作

先启动eureka-server,config-server,Service-A和两个Service-B，eureka-ribbon此时访问<http://localhost:1111/>



注意：由于Service-A要访问Config-Server中的配置文件配置文件名为{application}-{profile}.properties，因此Service-A的服务名要改为{application}此处是didispace.

### (2)开始使用Zuul

* + 引入依赖spring-cloud-starter-zuul、spring-cloud-starter-eureka，如果不是通过指定serviceId的方式，eureka依赖不需要。

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

* + 应用主类使用@EnableZuulProxy注解开启Zuul

@EnableZuulProxy

@SpringCloudApplication

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

@Bean

public AccessFilter accessFilter() {

return new AccessFilter();

}

}

这里用了@SpringCloudApplication注解，之前没有提过，通过源码我们看到，它整合了@SpringBootApplication、@EnableDiscoveryClient、@EnableCircuitBreaker，主要目的还是简化配置。这几个注解的具体作用这里就不做详细介绍了，之前的文章已经都介绍过。

* + application.properties中配置Zuul应用的基础信息，如：应用名、服务端口等。

spring.application.name=api-gateway

server.port=5555

### (3)Zuul配置

通过服务路由的功能，我们在对外提供服务的时候，只需要通过暴露Zuul中配置的调用地址就可以让调用方统一的来访问我们的服务，而不需要了解具体提供服务的主机信息了。

在Zuul中提供了两种映射方式：

* + 通过url直接映射，我们可以如下配置：

zuul.routes.api-a-url.path=/api-a-url/\*\*

zuul.routes.api-a-url.url=http://localhost:2222/

该配置，定义了，所有到Zuul的中规则为：/api-a-url/\*\*的访问都映射到http://localhost:2222/上，也就是说当我们访问http://localhost:5555/api-a-url/add?a=1&b=2的时候，Zuul会将该请求路由到：http://localhost:2222/add?a=1&b=2上。

其中，配置属性zuul.routes.api-a-url.path中的api-a-url部分为路由的名字，可以任意定义，

但是一组映射关系的path和url要相同，下面讲serviceId时候也是如此。

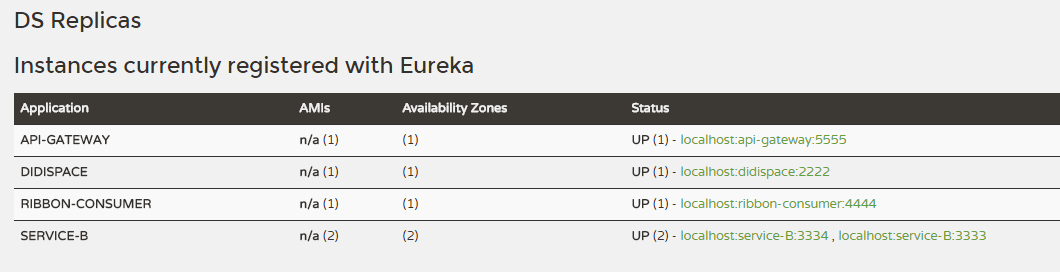
* + 通过url映射的方式对于Zuul来说，并不是特别友好，Zuul需要知道我们所有为服务的地址，才能完成所有的映射配置。而实际上，我们在实现微服务架构时，服务名与服务实例地址的关系在eureka server中已经存在了，所以只需要将Zuul注册到eureka server上去发现其他服务，我们就可以实现对serviceId的映射。例如，我们可以如下配置：

# routes to serviceId

zuul.routes.api-b.path=/api-b/\*\*

zuul.routes.api-b.serviceId=service-B

启动应用，再次访问<http://localhost:1111/>



尝试通过服务网关来访问service-A和service-B，根据配置的映射关系，分别访问下面的url

* + http://localhost:5555/api-b/add?a=1&b=2：通过serviceId映射访问service-B中的add服务
  + http://localhost:5555/api-a-url/add?a=1&b=2：通过url映射访问service-A中的add服务

至此，已经实现了基本架构

## 五、服务间调用

如何实现Service-A调用Service-B的服务

使用RestTemplate实现

例如：在Service-A中ComputeController中新增如下服务：

//A服务调用B服务

@RequestMapping(value="testServiceB",method=RequestMethod.GET)

public String testServiceB(@RequestParam Integer a,@RequestParam Integer b){

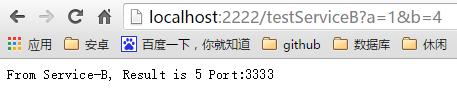
RestTemplate restTemplate=new RestTemplate();

return restTemplate.getForObject("http://localhost:3333/add?a="+a+"&b="+b, String.class);

}

启动应用，访问<http://localhost:2222/testServiceB?a=1&b=4>

得到如下结果：



说明服务调用成功。