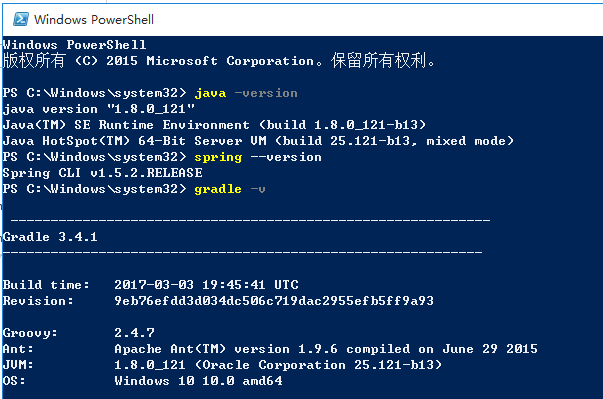
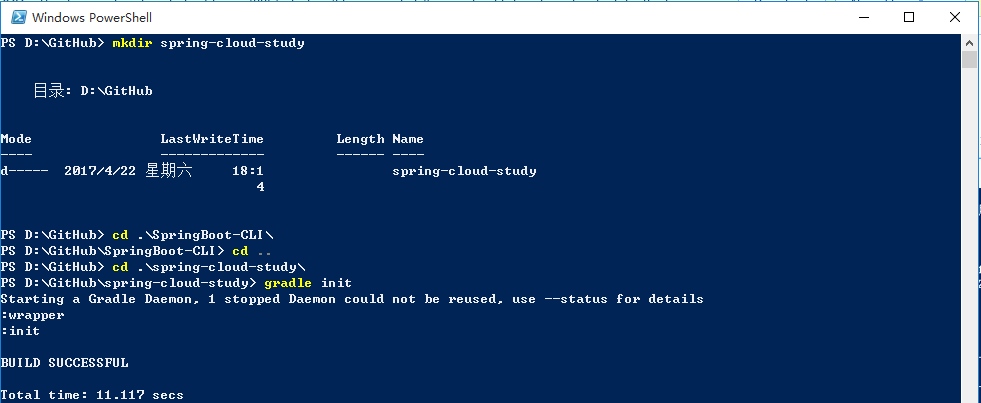
Spring Cloud学习

先看一下我的环境， JDK1.8+Spring CLI 1.5.2+Gradle 3.4.1



# 主项目

这是一系列的学习项目，所以先使用gradle init 创建主项目，也就用Gradle构建一个多模块的项目。



修改build.gradle文件，使用IDEA插件。因为一些其他原因，也将Eclipse插件也加上。然后用IDEA打开项目。

# Eureka

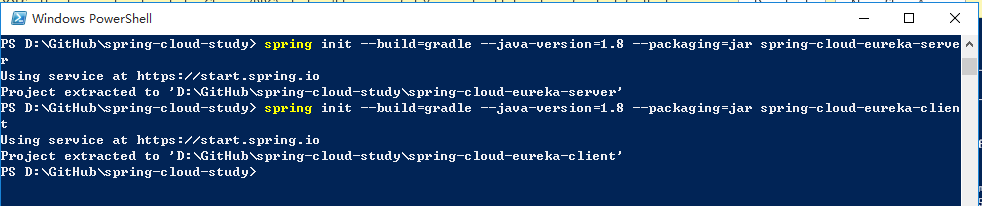
Eureka主要用于注册和发现服务的。她是一个高可用的组件，没有后端缓存，每一个实例注册之后需要向注册中心发送心跳（因此可以在内存中完成）

## Step 1 spring init命令

在spring-cloud-study目录下面使用Spring CLI创建两个项目作为eureka的server和client。

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-eureka-server

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-eureka-client



## Step 2 更改gradle编译文件

同样我们更改一下build.gradle文件, 加入

apply **plugin**: **'idea'**apply **plugin**: **'eclipse'**

插件，即可用IDEA打开了，可是很明显，这个build.gradle跟前面主项目的build.gradle长得何其相似。多年的经验告诉我们相同的代码在同一个项目中多次出现就应该重构了，不发明重复无意义的轮子是衡量一个开发人员的等级的一个标杆。所以我们将eureka-server和eureka-client的build.gradle里相同的部分提取到主项目的build.gradle里的subprojects 节中。

接下来，我们更改主项目下的setting.gradle文件，表示这个一个多模块的方案，其实很简单，现在只需简单的加入include **'spring-cloud-eureka-server'**,**'spring-cloud-eureka-client'**即可。

移除多余配置后的spring-cloud-eureka-server端的build.gradle将简洁成这样：

version = **'0.0.1-SNAPSHOT'**dependencies {  
 compile(**'org.springframework.cloud:spring-cloud-starter-eureka-server'**)  
}

而spring-cloud-eureka-client的build.gradle是这样子的：

version = **'0.0.1-SNAPSHOT'**dependencies {  
 compile(**'org.springframework.cloud:spring-cloud-starter-eureka'**)  
}

## Step 3 Java代码的更改

比较简单,就是在Spring Boot的启动程序加上相关的声明即可。

在EurekaServerApplication.java类上加入声明@EnableEurekaServer

在EurekaServerApplication.java类上加入声明@EnableEurekaClient

而作为测试，可简单写个RestController.

@RestController  
**public class** HelloController {  
  
 @Value(**"${server.port}"**)  
 String **port**;  
 @RequestMapping(**"/hi"**)  
 **public** String home(@RequestParam String name) {  
 **return "hi "**+name+**",i am from port:"** +**port**;  
 }  
}

## Step 4 更改application.yml文件

eureka-server 的application.yml中指明该程序占用的端口号为8761,本次的详细配置如下：

**server:  
 port:** 8761  
  
**eureka:  
 instance:  
 hostname:** localhost  
 **client:  
 registerWithEureka:** false  
 **fetchRegistry:** false  
 **serviceUrl:  
 defaultZone:** http://${eureka.instance.hostname}:${server.port}/eureka/

默认情况下erureka server也是一个eureka client ,必须要指定一个 server。

通过

eureka.client.registerWithEureka = false

eureka.client.fetchRegistry = false

来表明自己是一个eureka server.

而eureka-client的application.yml则指定程序所占商品为8762.

**server:  
 port:** 8762  
  
**eureka:  
 client:  
 serviceUrl:  
 defaultZone:** http://localhost:8761/eureka/  
  
**spring:  
 application:  
 name:** service-hi

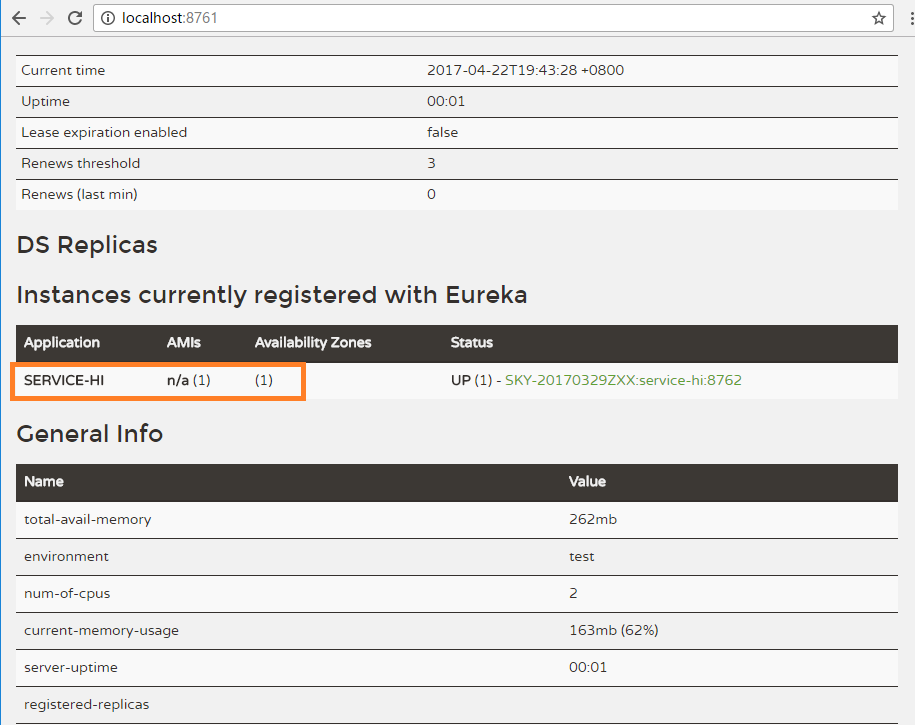
当client向server注册时，它会提供一些元数据，例如主机和端口，URL，主页等。Eureka server 从每个client实例接收心跳消息。 如果心跳超时，则通常将该实例从注册server中删除。需要指明spring.application.name,这个很重要，这在以后的服务与服务之间相互调用一般都是根据这个name 。

## Step 5 测试

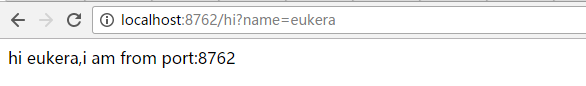
最后分别在spring-cloud-eureka-server和spring-cloud-eureka-client目录下运行

.\gradlew bootRun

然后在Chrome中输入<http://localhost:8761/>



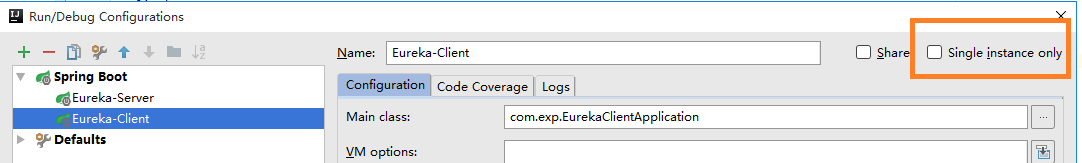
同样在另一个tab输入<http://localhost:8762/hi?name=eukera>



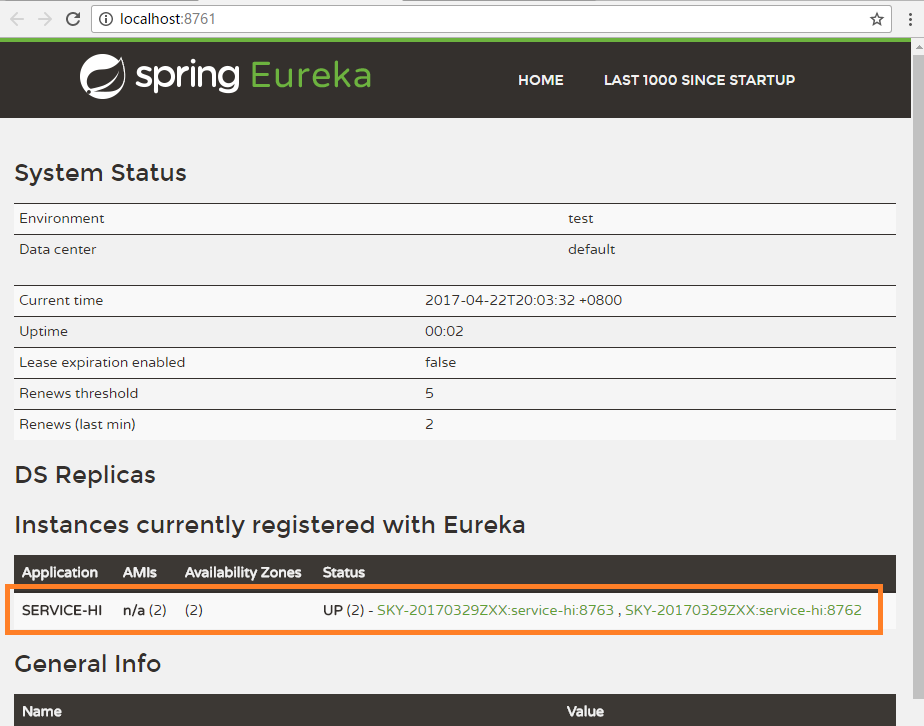
# Rest + Ribbon

ribbon是一个负载均衡客户端，可以很好的控制htt和tcp的一些行为。Feign也用到ribbon，当你使用@ FeignClient，ribbon自动被应用。

将spring-cloud-eureka-client的springboot配置改成可运行多个实例。



然后在application.yml中将端口改为8763,然后运行，就会启动两个服务，并且体现在eureka服务注册页面。



新建一个spring-boot项目

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-ribbon

修改build.gradle以及父项目的setting.gradle。

修改后的build.gradle

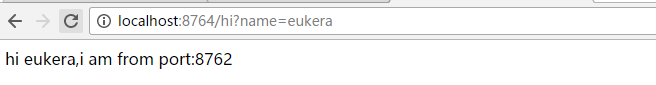
version = **'0.0.1-SNAPSHOT'**dependencies {  
 compile(**'org.springframework.cloud:spring-cloud-starter-eureka'**)  
 compile(**'org.springframework.cloud:spring-cloud-starter-ribbon'**)  
}  
  
dependencyManagement {  
 imports {  
 mavenBom **"org.springframework.cloud:spring-cloud-dependencies:Dalston.RELEASE"** }  
}

至于settings.gradle只是在后面加上’spring-cloud-ribbon’

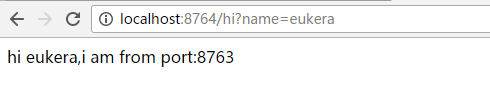
在application.yml中指定本服务的端口为8764，并建一个service和一个controller进行测试。

分别启动所有项目，然后在Chrome中输入<http://localhost:8764/hi?name=eukera>

会发现交替出现



和



这说明当我们通过调用restTemplate.getForObject(“http://SERVICE-HI/hi?name=“+name,String.class)，获取service-hi的方法时，已经做了负载均衡，访问了不同的端口的服务。

此时的架构是这样子的



* 一个服务注册中心，eureka server,端口为8761
* service-hi工程跑了两个副本，端口分别为8762,8763，分别向服务注册中心注册
* sercvice-ribbon端口为8764,向服务注册中心注册
* 当sercvice-ribbon通过restTemplate调用service-hi的hi接口时，因为用ribbon进行了负载均衡，会轮流的调用service-hi：8762和8763 两个端口的hi接口；

# Feign

Feign是一个声明式的web服务客户端，它使得写web服务变得更简单。使用Feign,只需要创建一个接口并注解。它具有可插拔的注解特性，包括Feign 注解和JAX-RS注解。Feign同时支持可插拔的编码器和解码器。spring cloud对Spring mvc添加了支持，同时在spring web中次用相同的HttpMessageConverter。当我们使用feign的时候，spring cloud 整和了Ribbon和eureka去提供负载均衡。

简而言之：

* feign采用的是接口加注解
* feign 整合了ribbon

创建一个新的spring-boot服务

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-feign

同ribbon一样的操作之后，我们在Chrome中输入<http://localhost:8765/hi?name=eukera>

同样发现8762和8763两个端口的数据交替出现。

# 熔断器Hystrix

在微服务架构中，我们将业务拆分成一个个的服务，服务与服务之间可以相互调用（RPC）。为了保证其高可用，单个服务又必须集群部署。由于网络原因或者自身的原因，服务并不能保证服务的100%可用，如果单个服务出现问题，调用这个服务就会出现网络延迟，此时若有大量的网络涌入，会形成任务累计，导致服务瘫痪，甚至导致服务“雪崩”。

为了解决这个问题，就出现熔断器模型。Netflix已经创建了一个名为Hystrix的库来实现熔断器模式。

改造前面的ribbon，使其支持熔断器。

首先引入引用compile(**'org.springframework.cloud:spring-cloud-starter-hystrix'**)

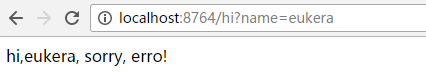
在启动RobbinApplicaiton中声明加上@EnableHystrix

并且在HelloService里加上@HystrixCommand(fallbackMethod = **"hiError"**)

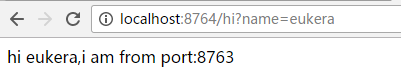
**public** String hiError(String name) {  
 **return** String.*format*(**"hi,%s,sorry,error!"**,name);  
}

启动之后，正常情况下，交替出现8762和8763

停掉一个8762后，就会出现



和



说明熔断器起作用了。

由于feign自带断路模式，而且默认是打开的(也就是feign.hystrix.enabled=false,这个理解起来有点怪异)，所以如果我们停掉一个spring-cloud-eureka-client，就会发现8762始终不会出现了。当然如果不想使用默认的，也想和我们上面的ribbon一样的行为的话，首先要在application.yml中停止使用默认的熔断器。

**feign:  
 hystrix:  
 enabled:** true

不过，经实验发现，这个并不会如ribbon一样交替出现错误和正常显示。而是有时还是会持续出现8763，偶尔出现错误显示而已。

查看Dashboard. 加入引用compile(**'org.springframework.cloud:spring-cloud-starter-hystrix-dashboard'**)

compile(**'org.springframework.boot:spring-boot-starter-actuator'**)

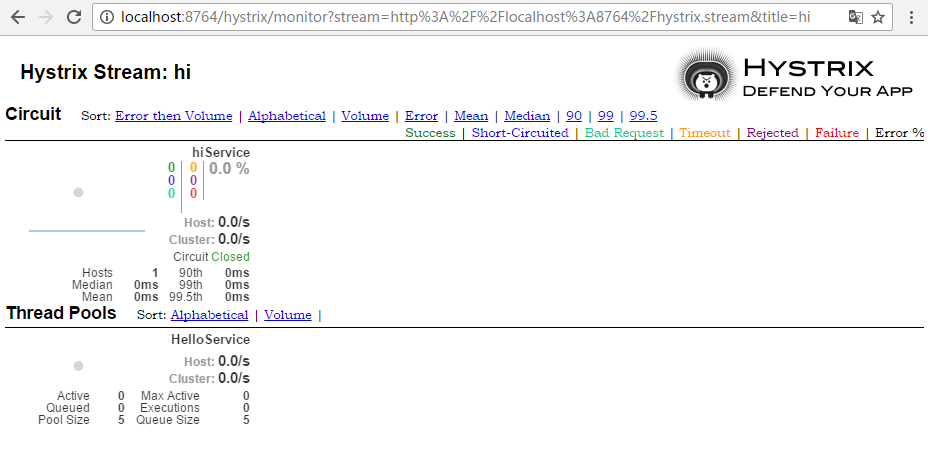
这两个都必须引用，如果第二个不引用，就会出现

Unable to connect to Command Metric Stream

并在启动程序中加上：@EnableHystrixDashboard

然后在Chrome中输入<http://localhost:8764/hystrix>

或者<http://localhost:8765/hystrix>



当然你在Chrome的另一个tab继续输入刷新<http://localhost:8764/hi?name=eukera>

这里也会同步更新的。

# 路由网关zuul

zuul的主要功能是路由和过滤器。路由功能是微服务的一部分，比如／api/user映射到user服务，/api/shop映射到shop服务。zuul实现了负载均衡

创建一个新的spring-boot服务

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-zuul

更改build.gradle和settings.gradle

添加引用compile(**'org.springframework.cloud:spring-cloud-starter-zuul'**)

并在ZuulApplication.java里加入声明

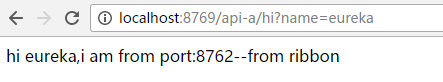
@EnableEurekaClient  
@EnableZuulProxy

添加application.yml

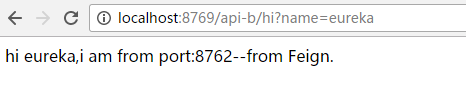
**eureka:  
 client:  
 serviceUrl:  
 defaultZone:** http://localhost:8761/eureka/  
**server:  
 port:** 8769  
**spring:  
 application:  
 name:** service-zuul  
**zuul:  
 routes:  
 api-a:  
 path:** /api-a/\*\*  
 **serviceId:** service-ribbon  
 **api-b:  
 path:** /api-b/\*\*  
 **serviceId:** service-feign

首先向eureka注册自己，端口为8769，服务名为service-zuul；以/api-a/ 开头的请求都指向service-ribbon；以/api-b/开头的请求都指向service-feign；

依次启动这5个项目之后，在Chrome里运行<http://localhost:8769/api-a/hi?name=eureka>



输入<http://localhost:8769/api-b/hi?name=eureka>



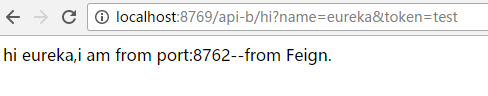
如果zuul单单只是路由功能的话，似乎也不值得如此宣扬，她还可以用于过滤，做一些安全验证之类的工作。

我们可以让上面的两个方法必须要有一个token之类的安全验证。有了zuul, 只需要添加一个Filter就可以达到目的了。

@Component  
**public class** SecurityFilter **extends** ZuulFilter {  
 **private static** Logger *log* = LoggerFactory.*getLogger*(SecurityFilter.**class**);  
  
 @Override  
 **public** String filterType() {  
 **return "pre"**;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** 0;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run() {  
 RequestContext ctx = RequestContext.*getCurrentContext*();  
 HttpServletRequest request = ctx.getRequest();  
 *log*.info(String.*format*(**"%s >>> %s"**, request.getMethod(), request.getRequestURL().toString()));  
 Object accessToken = request.getParameter(**"token"**);  
 **if**(accessToken == **null**) {  
 *log*.warn(**"token is empty"**);  
 ctx.setSendZuulResponse(**false**);  
 ctx.setResponseStatusCode(401);  
 **try** {  
 ctx.getResponse().getWriter().write(**"token is empty"**);  
 }**catch** (Exception e){}  
  
 **return null**;  
 }  
 *log*.info(**"ok"**);  
 **return null**;  
 }  
}

再次运行上面的代码就会输出token is empty. 我们需要在后面加上&token=test参数了。

<http://localhost:8769/api-b/hi?name=eureka&token=test>



关于上面这段SecurityFilter.java的简要解释

* filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：
  + - pre：路由之前
    - routing：路由之时
    - post： 路由之后
    - error：发送错误调用
* filterOrder：过滤的顺序
* shouldFilter：这里可以写逻辑判断，是否要过滤，本文true,永远过滤。
* run：过滤器的具体逻辑。可用很复杂，包括查sql，nosql去判断该请求到底有没有权限访问。

# Config

在分布式系统中，spring cloud config 提供一个服务端和客户端去提供可扩展的配置服务。我们可用用配置服务中心区集中的管理所有的服务的各种环境配置文件。配置服务中心采用Git的方式存储配置文件，因此我们很容易部署修改，有助于对环境配置进行版本管理。

创建一个新的spring-boot程序作为配置服务器，spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-config-server

修改build.gradle和settings.gradle

dependencies {  
 compile(**'org.springframework.cloud:spring-cloud-config-server'**)  
}

这次我们使用bootstrap.yml来配置git相关的配置。

bootstrap.yml 和application.yml 都可以用来配置参数

bootstrap.yml可以理解成系统级别的一些参数配置，这些参数一般是不会变动的

application.yml 可以用来定义应用级别的，如果搭配spring-cloud-config使用 application.yml里面定义的文件可以实现动态替换

在application.yml中只是指定程序的名字以及程序所占的端口号。

**server:  
 port:** 8888  
**spring:  
 application:  
 name:** config-server

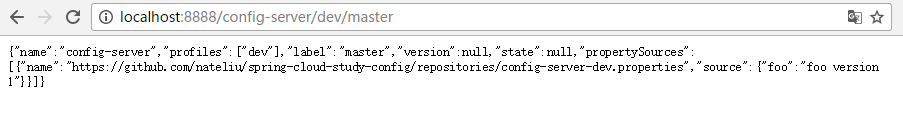
bootstrap.yml的内容：

**spring:  
 cloud:  
 config:  
 label:** master  
 **server:  
 git:  
 uri:** https://github.com/nateliu/spring-cloud-study-config  
 **searchPaths:** repositories  
 **username:** nateliu  
 **password: ""**

password这里我使用了双引号是以防有些用户的密码包含特殊字符。

最后在仓库<https://github.com/nateliu/spring-cloud-study-config> 里新建一个目录叫repositories。里面存放一个文件叫config-server-dev.properties，注意这个名字一定要和application.yml中定义的name一致。

Chrome中输入<http://localhost:8888/config-server/dev/master>



证明配置服务中心可以从远程程序获取配置信息。

http请求地址和资源文件映射如下:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

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

/{application}-{profile}.properties

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

翻译成本示例，也就是可以这样子访问，虽然我们并没有定义什么yml文件在这个远程仓库里:

<http://localhost:8888/config-server/dev/master>

<http://localhost:8888/config-server-dev.yml>

<http://localhost:8888/master/config-server-dev.yml>

<http://localhost:8888/config-server-dev.properties>

<http://localhost:8888/master/config-server-dev.properties>

创建一个新的spring-boot程序作为配置客户端

spring init --build=gradle --java-version=1.8 --packaging=jar spring-cloud-config-client

修改build.gradle和settings.gradle

dependencies {  
 compile(**'org.springframework.cloud:spring-cloud-starter-config'**)  
 compile(**'org.springframework.boot:spring-boot-starter-web'**)  
}

同config server一样，我们在application.yml中指定程序的名称和占用的端口号。

**server:  
 port:** 8881  
**spring:  
 application:  
 name:** config-client

而在bootstrap.yml中指定配置服务的一些信息:

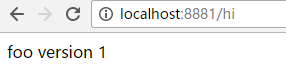
**spring:  
 cloud:  
 config:  
 label:** master  
 **profile:** dev  
 **uri:** http://localhost:8888/

写到这里，我们可以看出我们的远程git配置仓库似乎需要更改一下了。我们试着将config-server-dev.properties改名为config-client-dev.properties。内容不变.

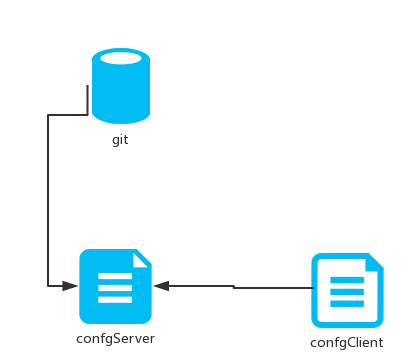
最后新增一个RestController供测试。

@RestController  
**public class** HelloController {  
 @Value(**"${foo}"**)  
 String **foo**;  
  
 @RequestMapping(value = **"/hi"**)  
 **public** String hi(){  
 **return foo**;  
 }  
}

Chrome里输入<http://localhost:8881/hi>

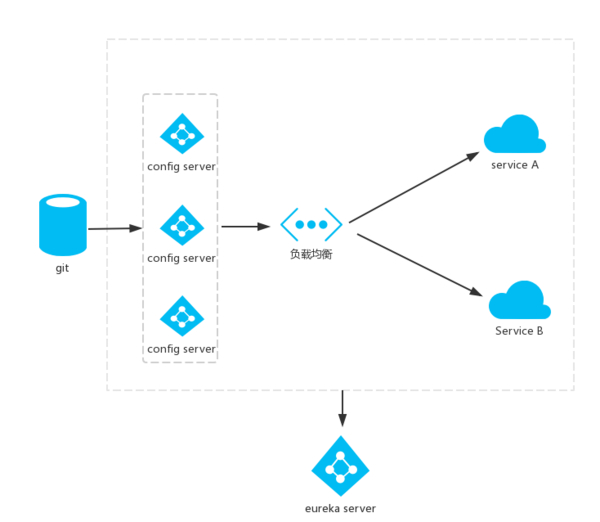


这也就是说，config-client从config-server获取了foo的属性，而config-server是从git仓库读取的,如图：



Eureka + Config

当服务很多时，都需要同时从配置中心读取文件的时候，这时我们可以考虑将配置中心做成一个微服务，并且将其集群化，从而达到高可用，架构图如下：



将config server加入eureka的引用，并注册进eureka server，即在启动Application中加入@EnableEurekaClient

Config server的application.yml中加上

**eureka:  
 client:  
 serviceUrl:  
 defaultZone:** http://localhost:8761/eureka/

config client的application.yml中也加上这面片断

并将bootstrap.yml改成：

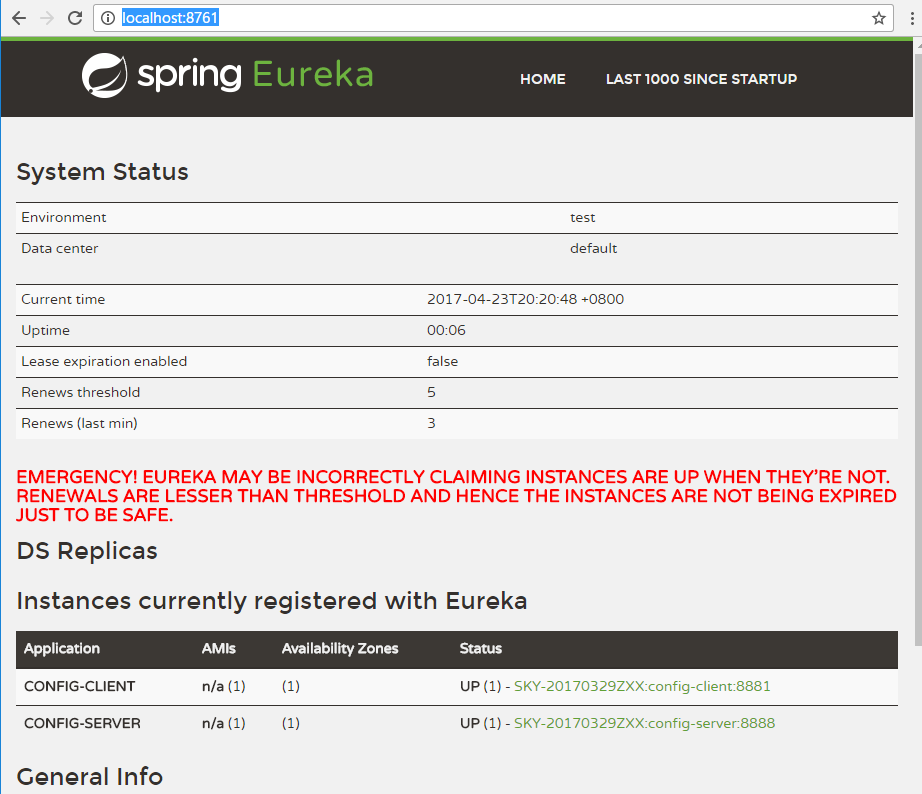
**spring:  
 cloud:  
 config:  
 label:** master  
 **profile:** dev  
 *#uri: http://localhost:8888/* **discovery:  
 enabled:** true  
 **serviceId:** config-server

spring.cloud.config.discovery.enabled 是否从配置中心读取文件

spring.cloud.config.discovery.serviceId 配置中心的servieId，即服务名

这时发现，在读取配置文件不再写ip地址，而是服务名，这时如果配置服务部署多份，通过负载均衡，从而高可用。

Chrome中输入<http://localhost:8761/>



<http://localhost:8881/hi> 仍然取到了foo属性的值。

# 附录

本文主要参考<http://blog.csdn.net/forezp/article/details/70148833>