Spring Cloud学习文档

# 相关网站

官网：<https://spring.io/projects/spring-cloud>

中文网：<https://springcloud.cc/>

当当书籍目录：<http://product.dangdang.com/25309061.html>

中文文档：<https://springcloud.cc/spring-cloud-dalston.html>

例子：<https://gitee.com/chrywhy/test/tree/master/spring-cloud>

# 简介

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线）。分布式系统的协调导致了样板模式, 使用Spring Cloud开发人员可以快速地支持实现这些模式的服务和应用程序。他们将在任何分布式环境中运行良好，包括开发人员自己的笔记本电脑，裸机数据中心，以及Cloud Foundry等托管平台。

版本：Dalston.RELEASE

# 特性

Spring Cloud专注于提供良好的开箱即用经验的典型用例和可扩展性机制覆盖。

* 分布式/版本化配置
* 服务注册和发现
* 路由
* service - to - service调用
* 负载均衡
* 断路器
* 分布式消息传递

# [服务注册](https://www.cnblogs.com/chry/p/7248947.html)

<https://www.cnblogs.com/chry/p/7248947.html>

Spring Cloud官网的定义比较抽象，我们可以从简单的东西开始。Spring Cloud是基于Spring Boot的， 最适合用于管理Spring Boot创建的各个微服务应用。要管理分布式环境下的各个Spring Boot微服务，必然存在服务的注册问题。所以我们先从服务的注册谈起。既然是注册，必然有个管理注册中心的服务器，各个在Spring Cloud管理下的Spring Boot应用就是需要注册的client。

Spring Cloud使用erureka [jʊ'ri:kə] server,  然后所有需要访问配置文件的应用都作为一个erureka client注册上去。eureka是一个高可用的组件，它没有后端缓存，每一个实例注册之后需要向注册中心发送心跳，在默认情况下erureka server也是一个eureka client ,必须要指定一个 server。

**2.  创建Eureka Server**

1）.创建一个Maven工程helloworld.eureka.server， pom.xml内容如下：

<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>org.lys</groupId>

<artifactId>helloworld.eureka.server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.eureka.server</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

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

</properties>

<dependencies>

<!--eureka server -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- spring boot test-->

<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>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

2）. 用Spring Boot创建一个服务类EurekaServerApplication，需要一个注解@EnableEurekaServer加在springboot工程的启动类上

**package** org.lys.helloworld.eureka.server;

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

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

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

@EnableEurekaServer

@SpringBootApplication

**public** **class** EurekaServerApplication {

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

SpringApplication.*run*(EurekaServerApplication.**class**, args);

}

}

3).eureka server的配置文件application.yml，其中registerWithEureka：false和fetchRegistry：false表明自己是一个eureka server

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

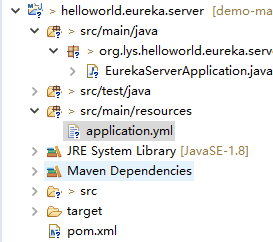
registerWithEureka: false

fetchRegistry: false

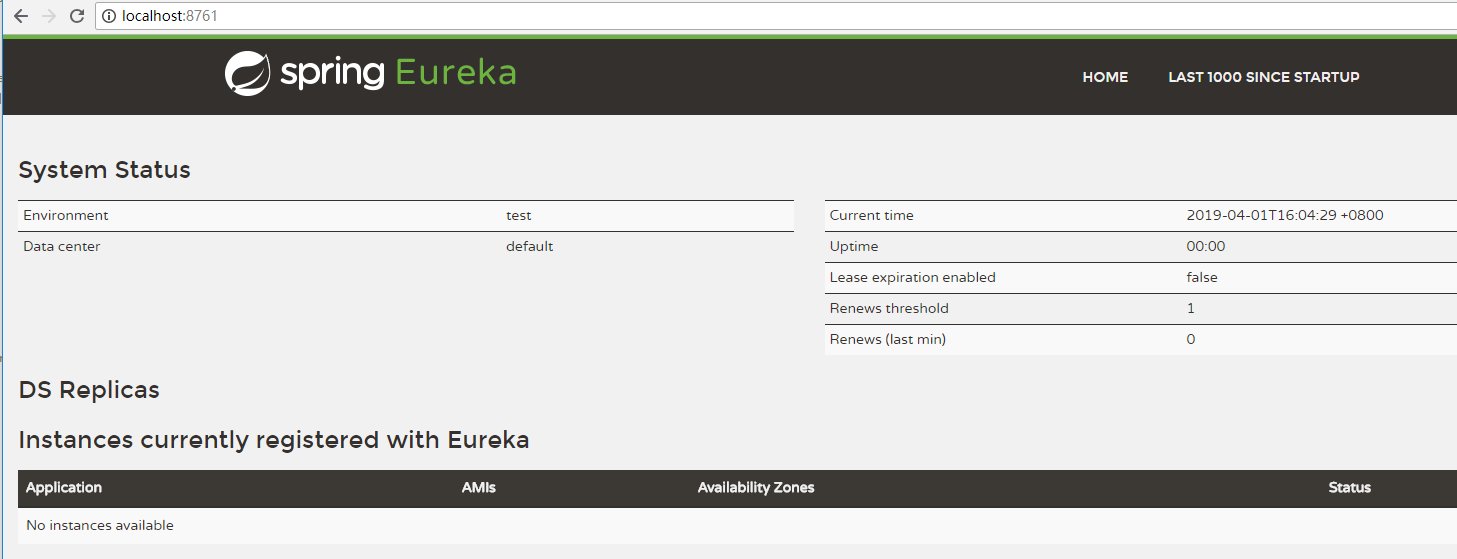
serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

4) eureka server的工程结构如下



5）启动eureka server，然后访问http://localhost:8761, 界面如下， "No instances available" 表示无client注册



**3.  创建Eureka Client**

1). 创建一个Maven工程helloworld.eureka.client， pom.xml内容如下：

<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>org.lys</groupId>

<artifactId>helloworld.eureka.client</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.eureka.client</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<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-web</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>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

2).  创建主类EurekaClientApplication

使用@EnableEurekaClient注解表明是client

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

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

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

**import** org.springframework.cloud.netflix.eureka.EnableEurekaClient;

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

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

@SpringBootApplication

@EnableEurekaClient

@RestController

**public** **class** EurekaClientApplication {

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

SpringApplication.*run*(EurekaClientApplication.**class**, args);

}

@Value("${server.port}")

String port;

@RequestMapping("/")

**public** String home() {

**return** "hello world from port " + port;

}

}

3) eureka client的配置文件application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

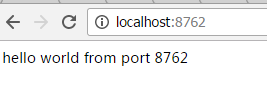
port: 8762

spring:

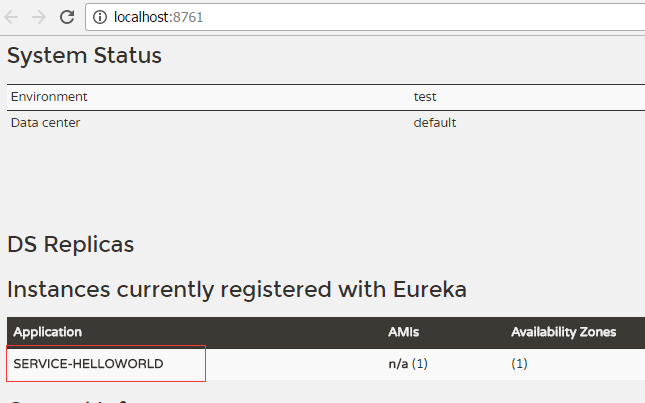
application:

name: service-helloworld

4). Client启动后， 可以访问<http://localhost:8762>



5). 再次访问服务器端口, 可以看到Service Helloworld已经自动注册到之前的server中http://localhost:8761/



# [配置管理](https://www.cnblogs.com/chry/p/7250584.html)

<http://www.cnblogs.com/chry/p/7250584.html>

使用Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念映射与Spring *Environment*和*PropertySource*抽象相同，因此它们与Spring应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。很容易添加替代实现，并使用Spring配置将其插入。

以上是Spring Cloud官网对配置服务的描述， 简单阐述一下我的理解。比如我们要搭建一个网站，需要配置数据库连接，指定数据库服务器的IP地址，数据库名称，用户名和口令等信息。通常的方法， 我们可以在一个配置文件中定义这些信息，或者开发一个页面专门配置这些东西。只有一个web服务器的时候, 很方便。但假如需要搭建同多台服务器时，当然可以每台服务器做同样配置，但维护和同步会很麻烦。我理解的配置服务至少有两种不同场景：

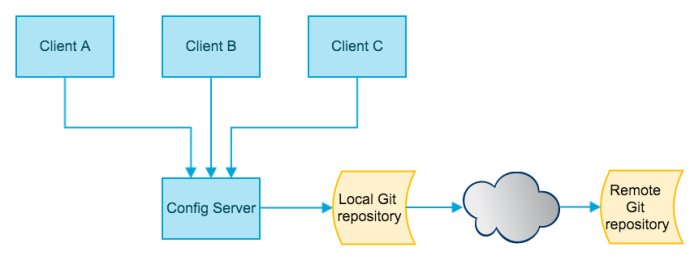
1）.  多个客户使用同一配置： 比如，多台服务器组成的集群，假如后端使用同一数据库，那么每台服务器都是用相同的配置。

2）.  不同客户使用不同的配置： 比如典型的场景是，开发，测试，生产使用相同的系统，但使用不同的数据库

如果有个统一的根本配置，是不是就很方便，一个可行的办法是，把这些配置文件放到一个共享存储（比如网络共享盘）中。这样只需要在共享存储修改一个或多个配置文件就可以了。但共享文件的方式受到具体布署环境的限制，很多时候很难达到多台Web服务器共享同一个存储硬盘。

共享盘的缺点是资源定位比较困难，Spring Cloud的解决方案是， 将这些配置文件放到版本管理服务器里面，Spring Cloud缺省配置使用GIT中。所有Web服务均从GIT中获取这些配置文件。由于GIT服务器与具体Web服务器之间不需要共享存储， 只要网络可达就行，从而可以实现Web服务于配置信息的存放位置的解耦。

Spring Cloud统一控制应用和GIT服务的交互，应用只需要按照Spring Cloud的规范配置GIT的URL即可。 使用GIT后，场景2和场景1的区别仅仅是，场景2中不同的client使用不同版本的配置文件，但应用但访问的文件看起来是会是同一个。Spring Cloud的配置服务结构入下图



下面我们继续上一节的例子[Spring Cloud 入门之一. 服务注册](http://www.cnblogs.com/chry/p/7248947.html) 继续展开, 让“Hello World”从配置文件helloworld.properties读出，内容格式如下

hello=Hello World

其中关键字hello的值“Hello World”，就是我们要输出的内容。

**一. 创建config Server**

 1.  创建Config Server：helloworld.config.server， maven工程里面配置spring-cloud-config-server

<dependency>

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

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

</dependency>

完整配置如下：

<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>org.lys</groupId>

<artifactId>helloworld.config.server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.config.server</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

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

</properties>

<dependencies>

<!--eureka server -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- spring boot test-->

<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>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

2. 创建Config Server，它也是一个Spring Boot应用，@EnableConfigServer注解说明了一个Config Server。同样我们使用@EnableEurekaClient将它注册到服务中心。

**package** org.lys.helloworld.config.server;

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

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

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

//@EnableEurekaClient

@EnableConfigServer

@SpringBootApplication

**public** **class** ConfigServerApplication {

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

SpringApplication.*run*(ConfigServerApplication.**class**, args);

}

}

3. Config server的配置文件application.yml , 注意配置文件的url是GIT服务器的仓库地址， searchPaths配置文件所在的文件夹在仓库中的路径, 在server端不需要指定具体配置文件名， 因为具体的配置文件是什么有应用（也就是client）决定。

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8888

spring:

cloud:

config:

server:

git:

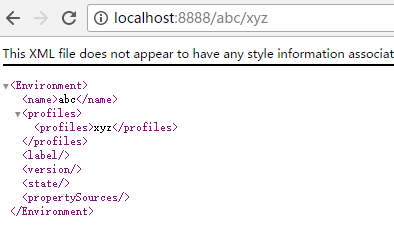
uri: https://git.oschina.net/chrywhy/test

searchPaths: spring-cloud/helloworldConfig

application:

name: config-server

4. 启动config server后，访问http://localhost:8888/abc/xyz, 可见如下响应。这个是输出是并没有包括具体配置文件的内容， 这个响应说明，config server可以正常访问我们配置在application.yml中的GIT服务



这个URL是啥意思， 需要解释一下。我们从输出就可以看到 abc 就是application的名字，xyz是profile的名字， 注意这里的abc, xyz均是随便输入的名字， 并不需要真实存在，config server这个REST接口返回的只是应用名为abc, profile名为xyz时，GIT配置环境的结构。

config server提供的REST接口，Spring Cloud官方文档提供了几个可选URL可以是如下几个：

1. /{application}/{profile}[/{label}]
2. /{application}-{profile}.yml
3. /{label}/{application}-{profile}.yml
4. /{application}-{profile}.properties
5. /{label}/{application}-{profile}.properties

比如 第三个格式，如果我们在GIT版本库中有一个配置文件 spring-cloud/helloworldConfig/config-client-dev.properties. 那么访问http://localhost:8888/config-client-dev.properties就可以显示配置文件内容。这个例子中， application的名字是"config-client"(也是下面我们即将创建的client), profile名字是dev, 文件后缀是.properties



本例由于配置了eureka服务中心，所以这个config server作为一个eureka client注册到了 eureka server中， 可以从http://localhost:8761看到我们启动的config server, 如果不需要注册到服务中心， 也可把这个配置去掉

  二**. 创建config client:** **helloworld.config.client**

1.  创建maven工程， pom.xml如下：

<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>org.lys.demo</groupId>

<artifactId>helloworld.config.client</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.config.client</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<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-web</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-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

2. 创建一个spring boot应用作为client

**package** org.lys.demo.helloworld.config.client;

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

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

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

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

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

@SpringBootApplication

@RestController

**public** **class** ConfigClientApplication {

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

SpringApplication.*run*(ConfigClientApplication.**class**, args);

}

@Value("${hello}")

String hello;

@RequestMapping(value = "/hello")

**public** String hello(){

**return** hello;

}

}

这个应用非常简单，就是从Config Server中获取配置项hello的值，Client Server向Config Server提交REST请求后，Config Server将访问GIT服务器，并将取得的配置项hello的值返回给client.

3. Config client需要一个应用配置文件， 定义config Server的URL，以及要访问的GIT具体分支。这个配置文件是bootstrap.yml (或者bootstrap.properties)

spring:

application:

name: config-client

cloud:

config:

label: master

profile: dev

uri: http://localhost:8888/

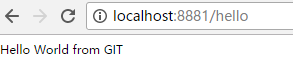
server:

port: 8881

这个配置定义了应用的名字是config-client(这就是将要用于组装前面Config Server一节中题到的application), profile采用dev, GIT分支用master。url是config server的地址。那么问题来了，我们似乎没定义配置文件名， 那配置文件名是什么呢？ 这点又体现了约定优于配置的思路， 这里Spring Cloud约定， 应用的配置文件名以如下方式组成：{application}-{profile}.properties（或者{application}-{profile}.yml）。比如我们这个应用的配置文件就是config-client-dev.properties. 所以只需要在GIT的中创建配置文件spring-cloud/helloworldConfig/config-client-dev.properties就可以了， 内容如下：

hello=Hello World from GIT

 4. 启动config-client应用后， 可以访问http://localhost:8881/hello, 可以看到，应用本身并没有直接配置hello的具体内容， 也没指定具体配置文件，所欲这些都由spring cloud框架提交给config server了。

**

5.  配置的更新

至此，spring cloud的配置管理简单示例已经完成，但client 不能自动感知服务端的变化。 比如，我们修改了GIT中的文件内容，但无论如何刷新client端的页面，都不能反映配置的变化

# [配置自动刷新](https://www.cnblogs.com/chry/p/7260778.html)

<http://www.cnblogs.com/chry/p/7260778.html>

之前讲的配置管理， 只有在应用启动时会读取到GIT的内容， 之后只要应用不重启，GIT中文件的修改，应用无法感知， 即使重启Config Server也不行。

比如上一单元（Spring Cloud 入门教程(二)： 配置管理）中的Hello World 应用，手动更新GIT中配置文件config-client-dev.properties的内容（别忘了用GIT push到服务器）

hello=Hello World from GIT version 1

刷新 http://localhost:8881/hello，页面内容仍然和之前一样，并没有反映GIT中最新改变， 重启config-server也一样，没有任何变化。要让客户端应用感知到这个变哈， Spring Cloud提供了解决方案是，客户端用POST请求/refresh方法就可以刷新配置内容。

**1. 让客户端支持/refresh方法**

要让/refresh生效，客户端需要增加一些代码支持：

1). 首先，在pom.xml中添加以下依赖。spring-boot-starter-actuator是一套监控的功能，可以监控程序在运行时状态，其中就包括/refresh的功能。

<dependency>

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

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

</dependency>

并修改bootstrap.yml

spring:

application:

name: config-client

cloud:

config:

label: master

profile: dev

uri: http://localhost:8888/

server:

port: 8881

management:

security:

enabled: false

增加红色部分，springboot 1.5.X 以上默认开通了安全认证，所以需要在配置文件添加以下配置management.security.enabled=false

2). 其次，开启refresh机制， 需要给加载变量的类上面加载@RefreshScope注解，其它代码可不做任何改变，那么在客户端执行/refresh的时候就会更新此类下面的变量值，包括通过config client从GIT获取的配置。

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.context.config.annotation.RefreshScope;

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

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

@SpringBootApplication

@RestController

@RefreshScope

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

@Value("${hello}")

String hello;

@RequestMapping(value = "/hello")

public String hello(){

return hello;

}

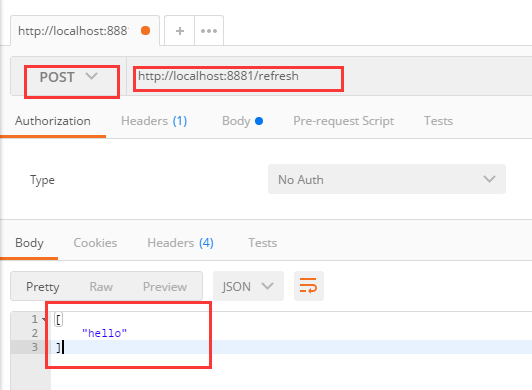
}

3). 启动应用， 查看http://localhost:8881/hello

4). 再次修改config-client-dev.properties的内容

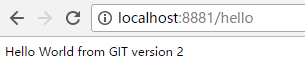
hello=Hello World from GIT version 2

5). 用postman发送POST请求：<http://localhost:8881/refresh>



可以从POST的结果看到，此次refresh刷新的配置变量有hello

6). 再次访问http://localhost/hello，可见到配置已经被刷新



**2. 通过Webhook自动触发/refresh方法刷新配置**

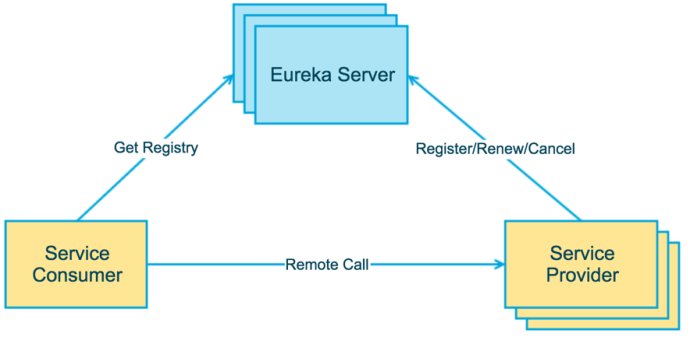
以上每当GIT中配置文件被修改，仍然需要我们主动调用/refresh方法（手动调用或者写代码调用）， 有没有办法让GIT中配置有改动就自动触发客户端的refresh机制呢？ 答案是：有。可以通过GIT提供的githook来监听push命令，如果项目中使用了Jenkins等持续集成工具（也是利用githook来监听的），就可以监听事件处理中直接调用/refresh方法就可以了。

# [分布式环境下自动发现配置服务](https://www.cnblogs.com/chry/p/7262088.html)

<http://www.cnblogs.com/chry/p/7262088.htm>

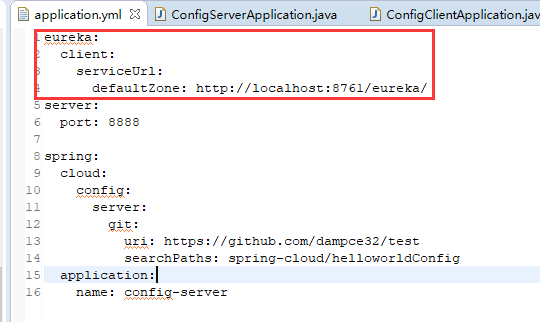
前一章， 我们的Hello world应用服务，通过配置服务器Config Server获取到了我们配置的hello信息“hello world”. 但自己的配置文件中必须配置config server的URL（http://localhost:8888）, 如果把config server搬到另外一个独立IP上， 那么作为一个client的hello world应用必须修改自己的bootstrap.yml中的config server的URL地址。这明显是不够方便的。

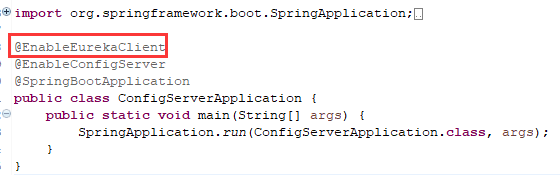
既然config server已经注册到了eureka服务中心，能否让服务中心自动帮hello world应用找到它需要的config server呢？ 答案是肯定的。我们的hello world应用只需要提供它需要的配置所在在的config server的名字就可以了， 在前面例子中，配置服务的名字就是“config-server”。那我们现在就把之前的服务和应用稍作修改， 来达到自动发现服务的方案。下图是Spring Cloud提供的服务发现机制。Config-server是其中的Service Provider, Config-client是Service Consumer， 它们都注册到服务中心Eureka Server。

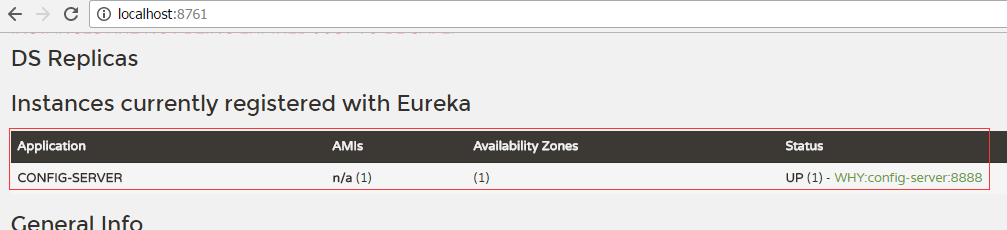


**1. 将config-server注册到服务中心**

config-server本身就是一个Spring Boot应用， 可以直接参考[Spring Cloud 入门教程(一): 服务注册](http://www.cnblogs.com/chry/p/7248947.html), 将config-server注册到eureka server中。访问http://localhost:8761, 可以看到我们的config-server已经注册。

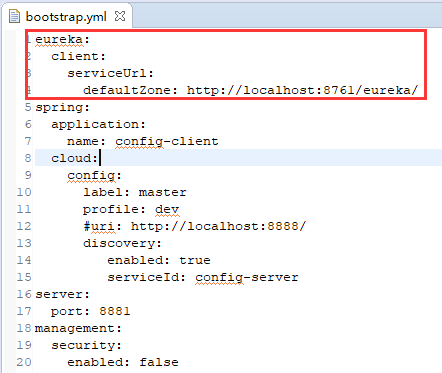


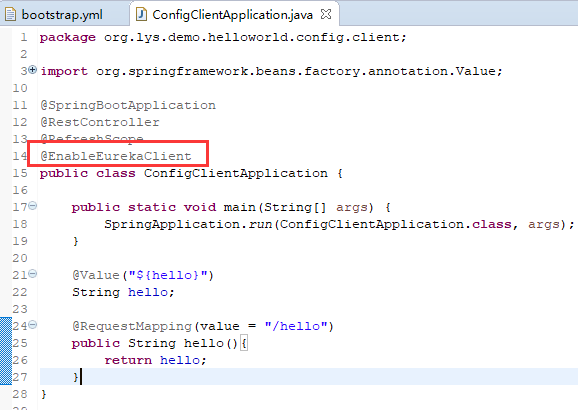




**2. 修改hello world应用的配置**

1）.同样，需要将Hello 我让你的应用注册到eureka 服务中心, 配置方法同前面一样， 不在赘述。





2）.修改配置文件，将config-server的URL硬编码机制改成，通过服务中心根据名字自动发现机制， 修改bootstrap.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

spring:

application:

name: config-client

cloud:

config:

label: master

profile: dev

#uri: http://localhost:8888/

discovery:

enabled: true

serviceId: config-server

server:

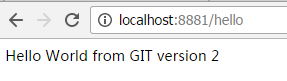
port: 8881

management:

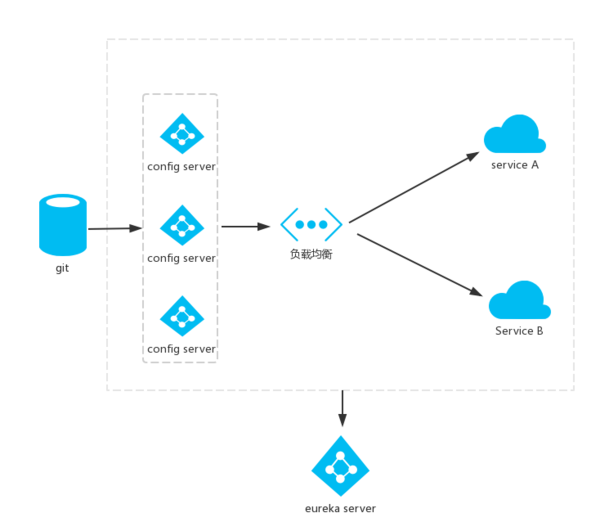
security:

enabled: false

我们注释掉了硬编码的config-server的URL配置， 取而代之的是服务注册中心的地址http://localhost:8761/eureka/以及配置服务的名字“config-server”, 同时打开自动发现机制discovery.enable = true. 我们在运行一下hello world应用， 可以发现， GIT里面的内容依然可以访问。此时我们的hello world应用已经完全不知道配置服务的地址，也不知道配置的内容， 所有这些都通过服务注册中心自动发现。



3.  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：

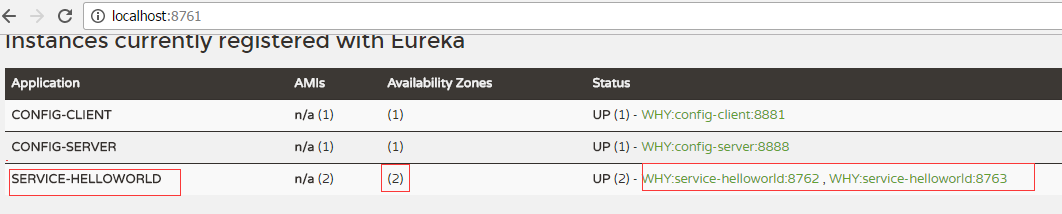


# [Ribbon实现客户端的负载均衡](https://www.cnblogs.com/chry/p/7263281.html)

<http://www.cnblogs.com/chry/p/7263281.html>

假如我们的Hello world服务的访问量剧增，用一个服务已经无法承载， 我们可以把Hello World服务做成一个集群。

很简单，我们只需要复制Hello world服务，同时将原来的端口8762修改为8763。然后启动这两个Spring Boot应用， 就可以得到两个Hello World服务。这两个Hello world都注册到了eureka服务中心。这时候再访问http://localhost:8761, 可以看到两个hello world服务已经注册。（服务与注册参见[Spring Cloud 入门教程(一): 服务注册](http://www.cnblogs.com/chry/p/7248947.html)）。



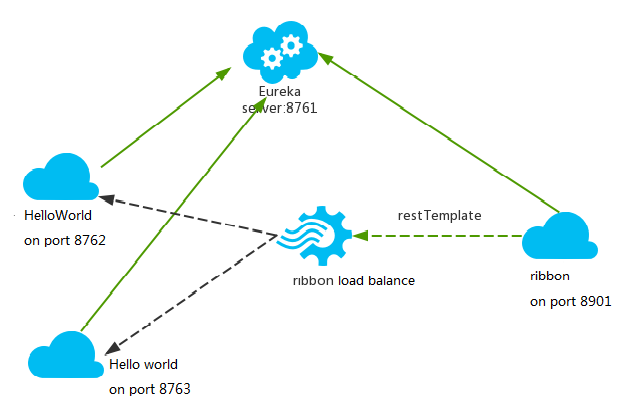
**1.  客户端的负载均衡**

负载均衡可分为服务端负载均衡和客户端负载均衡，服务端负载均衡完全由服务器处理，客户端不需要做任何事情。而客户端负载均衡技术，客户端需要维护一组服务器引用，每次客户端向服务端发请求的时候，会根据算法主动选中一个服务节点。常用的负载均衡算法有： Round Robbin,  Random，Hash，StaticWeighted等。

Spring 提供两种服务调度方式：Ribbon+restful和Feign。Ribbon就是一个基于客户端的负载均衡器， Ribbon提供了很多在HTTP和TCP客户端之上的控制.

Feign内部也已经使用了Ribbon, 所以只要使用了@FeignClient注解，那么这一章的内容也都是适用的。

下面就看看如何Spring Cloud如何用Ribbon来实现两个Hello World服务的负载均衡。以下是Spring cloud的ribbon客户端负载均衡架构图。



hello world服务和ribbon均注册到服务中心

service-Hellowworld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， 当sercvice-ribbon通过restTemplate调用service-Hellowworld的接口时，利用用ribbon进行负载均衡，会轮流的调用处于两个不同端口的Hello world服务

**2. 创建一个Ribbon服务**

**1) 创建一个maven工程，取名叫service-ribbon**, pom.xml文件如下：

<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>org.lys.demo</groupId>

<artifactId>helloworld.ribbon.service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.ribbon.service</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<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.cloud</groupId>

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

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

**2). 创建主类ServiceRibbonApplication**

@SpringBootApplication

@EnableDiscoveryClient//向服务中心注册

**public** **class** ServiceRibbonApplication {

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

SpringApplication.*run*(ServiceRibbonApplication.**class**, args);

}

@Bean

@LoadBalanced//注册表明，这个restTemplate是需要做负载均衡的

RestTemplate restTemplate() {

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

}

}

@EnableDiscoveryClient向服务中心注册，并且注册了一个叫restTemplate的bean。

@ LoadBalanced注册表明，这个restRemplate是需要做负载均衡的。

3**). 创建获取一个获取Hello内容的controler类**

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

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

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

@RestController

**public** **class** HelloControler {

@Autowired HelloService helloService;

@RequestMapping(value = "/")

**public** String hello(){

**return** helloService.getHelloContent();

}

}

**4). 创建获取一个获取Hello内容的service类**

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

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

public class HelloService {

@Autowired

RestTemplate restTemplate;

public String getHelloContent() {

//通过ribbon负载均衡机制， 自动选择一个Hello word服务

return restTemplate.getForObject("http://SERVICE-HELLOWORLD/",String.class);

}

}

这里关键代码就是, restTemplate.getForObject方法会通过ribbon负载均衡机制， 自动选择一个Hello word服务，

这里的URL是“http://SERVICE-HELLOWORLD/"，其中的SERVICE-HELLOWORLD是Hello world服务的名字，而注册到服务中心的有两个SERVICE-HELLOWORLD。 所以，这个调用本质是ribbon-service作为客户端根据负载均衡算法自主选择了一个作为服务端的SERVICE-HELLOWORLD服务。然后再访问选中的SERVICE-HELLOWORLD来执行真正的Hello world调用。

5）配置文件application.yml

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

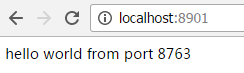
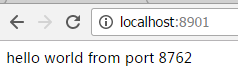
port: 8901

spring:

application:

name: service-ribbon

3. 按顺序启动应用：eureka-server、eureka-client、eureka-client2、ribbon-service应用，我们就可以访问http://localhost:8901/, 然后每次刷新可以看到以下两种结果交替出现，表明实际调用的是在不同端口的不同的SERVICE-HELLOWORLD。

# [用声明式REST客户端Feign调用远端HTTP服务](https://www.cnblogs.com/chry/p/7266916.html)

<http://www.cnblogs.com/chry/p/7266916.html>

首先简单解释一下什么是声明式实现？

要做一件事， 需要知道三个要素，where, what, how。即在哪里（ where）用什么办法（how）做什么（what）。什么时候做（when）我们纳入how的范畴。

1）编程式实现： 每一个要素（where，what，how）都需要用具体代码实现来表示。传统的方式一般都是编程式实现，业务开发者需要关心每一处逻辑

2）声明式实现： 只需要声明在哪里（where ）做什么（what），而无需关心如何实现（how）。Spring的AOP就是一种声明式实现，比如网站检查是否登录，开发页面逻辑的时候，只需要通过AOP配置声明加载页面（where）需要做检查用户是否登录（what），而无需关心如何检查用户是否登录（how)。如何检查这个逻辑由AOP机制去实现， 而AOP的登录检查实现机制与正在开发页面的逻辑本身是无关的。

在Spring Cloud Netflix栈中，各个微服务都是以HTTP接口的形式暴露自身服务的，因此在调用远程服务时就必须使用HTTP客户端。Feign就是Spring Cloud提供的一种声明式REST客户端。可以通过Feign访问调用远端微服务提供的REST接口。现在我们就用Feign来调用SERVICE-HELLOWORLD暴露的REST接口，以获取到“Hello World”信息。在使用Feign时，Spring Cloud集成了Ribbon和Eureka来提供HTTP客户端的负载均衡。

下面我们就采用Feign的方式来调用Hello World服务集群。

**1. 创建Maven工程，加入spring-cloud-starter-feign依赖**

<dependency>

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

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

</dependency>

完整的pom文件如下：

<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>org.lys.demo</groupId>

<artifactId>helloworld.feign.client</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.feign.client</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<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.cloud</groupId>

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

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

**2. 创建启动类，需呀加上@EnableFeignClients注解以使用Feign**， 使用@EnableDiscoveryClient开启服务自动发现

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

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

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

**import** org.springframework.cloud.netflix.feign.EnableFeignClients;

@SpringBootApplication

@EnableDiscoveryClient

@EnableFeignClients

**public** **class** ServiceFeignApplication {

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

SpringApplication.*run*(ServiceFeignApplication.**class**, args);

}

}

**3. 添加配置文件application.yml, 使用端口8902, 名字定义为service-feign, 并注册到eureka服务中心**

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

server:

port: 8902

spring:

application:

name: service-feign

**4. 定义Feign：一个用@FeignClient注解的接口类,**

@FeignClient用于通知Feign组件对该接口进行代理(不需要编写接口实现)，使用者可直接通过@Autowired注入; 该接口通过value定义了需要调用的SERVICE-HELLOWORLD服务（通过服务中心自动发现机制会定位具体URL）; @RequestMapping定义了Feign需要访问的SERVICE-HELLOWORLD服务的URL（本例中为根“/”）

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

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

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

@FeignClient(value = "SERVICE-HELLOWORLD")

**public** **interface** HelloWorldService {

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

String sayHello();

}

Spring Cloud应用在启动时，Feign会扫描标有@FeignClient注解的接口，生成代理，并注册到Spring容器中。生成代理时Feign会为每个接口方法创建一个RequetTemplate对象，该对象封装了HTTP请求需要的全部信息，请求参数名、请求方法等信息都是在这个过程中确定的，Feign的模板化就体现在这里

**5. 定义一个WebController。**

注入之前通过@FeignClient定义生成的bean,

sayHello()映射到http://localhost:8902/hello, 在这里，我修改了Hello World服务的映射，将根“/”， 修改成了“/hello”。

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

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

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

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

@RestController

**public** **class** WebController {

@Autowired HelloWorldService helloWorldFeignService;

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

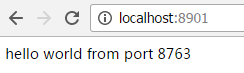
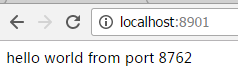
**public** String sayHello(){

**return** helloWorldFeignService.sayHello();

}

}

6. 启动Feign应用， 访问http://localhost:8902/hello， 多次刷新，可以看到和前一章Ribbon里面的应用一样， 两个Hello World服务的输出交替出现。说明通过Feign访问服务， Spring Cloud已经缺省使用了Ribbon负载均衡。

**6. 在Feign中使用Apache HTTP Client**

Feign在默认情况下使用的是JDK原生的URLConnection发送HTTP请求，没有连接池，但是对每个地址gwai会保持一个长连接，即利用HTTP的persistence connection 。我们可以用Apache的HTTP Client替换Feign原始的http client, 从而获取连接池、超时时间等与性能息息相关的控制能力。Spring Cloud从Brixtion.SR5版本开始支持这种替换，首先在项目中声明Apache HTTP Client和feign-httpclient依赖：

[复制代码](javascript:void(0);)

1 <!-- 使用Apache HttpClient替换Feign原生httpclient -->

2 <dependency>

3 <groupId>org.apache.httpcomponents</groupId>

4 <artifactId>httpclient</artifactId>

5 </dependency>

6 <dependency>

7 <groupId>com.netflix.feign</groupId>

8 <artifactId>feign-httpclient</artifactId>

9 <version>${feign-httpclient}</version>

10 </dependency>

[复制代码](javascript:void(0);)

然后在application.properties中添加：

feign.httpclient.enabled=true

**7. Feign的Encoder、Decoder和ErrorDecoder**

Feign将方法签名中方法参数对象序列化为请求参数放到HTTP请求中的过程，是由编码器(Encoder)完成的。同理，将HTTP响应数据反序列化为Java对象是由解码器(Decoder)完成的。默认情况下，Feign会将标有@RequestParam注解的参数转换成字符串添加到URL中，将没有注解的参数通过Jackson转换成json放到请求体中。注意，如果在@RequetMapping中的method将请求方式指定为POST，那么所有未标注解的参数将会被忽略，例如：

@RequestMapping(value = "/group/{groupId}", method = RequestMethod.GET)

void update(@PathVariable("groupId") Integer groupId, @RequestParam("groupName") String groupName, DataObject obj);

此时因为声明的是GET请求没有请求体，所以obj参数就会被忽略。

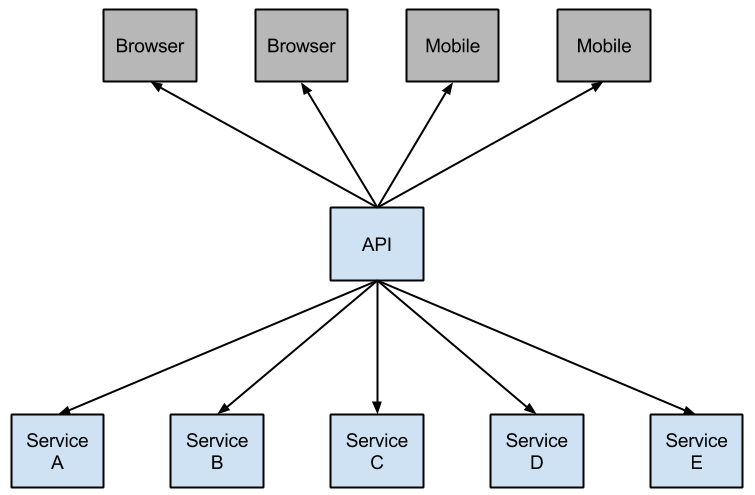
在Spring Cloud环境下，Feign的Encoder只会用来编码没有添加注解的参数。如果你自定义了Encoder, 那么只有在编码obj参数时才会调用你的Encoder。对于Decoder, 默认会委托给SpringMVC中的MappingJackson2HttpMessageConverter类进行解码。只有当状态码不在200 ~ 300之间时ErrorDecoder才会被调用。ErrorDecoder的作用是可以根据HTTP响应信息返回一个异常，该异常可以在调用Feign接口的地方被捕获到。我们目前就通过ErrorDecoder来使Feign接口抛出业务异常以供调用者处理。

# [熔断机制 -- 断路器](https://www.cnblogs.com/chry/p/7279856.html)

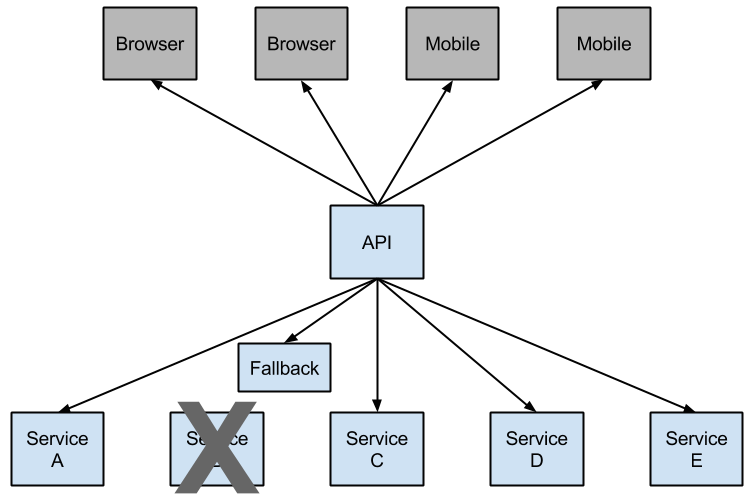
<http://www.cnblogs.com/chry/p/7279856.html>

对断路器模式不太清楚的话，可以参看另一篇博文：[断路器（Curcuit Breaker）模式](http://www.cnblogs.com/chry/p/7278853.html)，下面直接介绍Spring Cloud的断路器如何使用。

SpringCloud Netflix实现了断路器库的名字叫Hystrix. 在微服务架构下，通常会有多个层次的服务调用. 下面是微服架构下, 浏览器端通过API访问后台微服务的一个示意图：



一个微服务的超时失败可能导致瀑布式连锁反映，下图中，Hystrix通过自主反馈实现的断路器， 防止了这种情况发生。



 图中的服务B因为某些原因失败，变得不可用，所有对服务B的调用都会超时。当对B的调用失败达到一个特定的阀值(5秒之内发生20次失败是Hystrix定义的缺省值), 链路就会被处于open状态， 之后所有所有对服务B的调用都不会被执行， 取而代之的是由断路器提供的一个表示链路open的Fallback消息.  Hystrix提供了相应机制，可以让开发者定义这个Fallbak消息.

open的链路阻断了瀑布式错误， 可以让被淹没或者错误的服务有时间进行修复。这个fallback可以是另外一个Hystrix保护的调用, 静态数据，或者合法的空值. Fallbacks可以组成链式结构，所以，最底层调用其它业务服务的第一个Fallback返回静态数据.

下面，进入正题，在之前的两HELLO WORLD服务集群中加入断路器， 防止其中一个Hello world挂掉后， 导致系统发生连锁超时失败。

**1. 在maven工程（前面章节中介绍的Ribbon或者Feign工程）的pom.xml中添加hystrix库支持断路器**

<dependency>

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

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

</dependency>

**2. 在Ribbon应用中使用断路器**

**1). 在Spring Boot启动类上添加@EnableCircuitBreaker注解**

[复制代码](javascript:void(0);)

1 @SpringBootApplication

2 @EnableDiscoveryClient

3 @EnableCircuitBreaker

4 public class ServiceRibbonApplication {

5

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

7 SpringApplication.run(ServiceRibbonApplication.class, args);

8 }  
。。。 。。。

[复制代码](javascript:void(0);)

2). 用@HystrixCommand注解标注访问服务的方法

[复制代码](javascript:void(0);)

1 @Service

2 public class HelloService {

3 @Autowired RestTemplate restTemplate;

4

5 @HystrixCommand(fallbackMethod = "serviceFailure")

6 public String getHelloContent() {

7 return restTemplate.getForObject("http://SERVICE-HELLOWORLD/",String.class);

8 }

9

10 public String serviceFailure() {

11 return "hello world service is not available !";

12 }

13 }

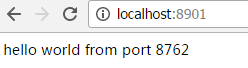
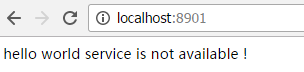
[复制代码](javascript:void(0);)

@HystrixCommand注解定义了一个断路器，它封装了getHelloContant()方法， 当它访问的SERVICE-HELLOWORLD失败达到阀值后，将不会再调用SERVICE-HELLOWORLD， 取而代之的是返回由fallbackMethod定义的方法serviceFailure（）。@HystrixCommand注解定义的fallbackMethod方法，需要特别注意的有两点：

第一，  fallbackMethod的返回值和参数类型需要和被@HystrixCommand注解的方法完全一致。否则会在运行时抛出异常。比如本例中，serviceFailure（）的返回值和getHelloContant()方法的返回值都是String。

第二，  当底层服务失败后，fallbackMethod替换的不是整个被@HystrixCommand注解的方法（本例中的getHelloContant), 替换的只是通过restTemplate去访问的具体服务。可以从中的system输出看到， 即使失败，控制台输出里面依然会有“call SERVICE-HELLOWORLD”。

启动eureka服务，只启动两个Helloworld服务，然后中断其中一个（模拟其中一个微服务挂起），访问http://localhost:8901/然后刷新, 由于有负载均衡可以看到以下两个页面交替出现。可以看到第二个被挂起的服务，被定义在Ribbon应该里面的错误处理方法替换了。

**4. 在Feign应用中使用断路器**

1）. Feign内部已经支持了断路器，所以不需要想Ribbon方式一样，在Spring Boot启动类上加额外注解

2）. 用@FeignClient注解添加fallback类， 该类必须实现@FeignClient修饰的接口。

1 @FeignClient(name = "SERVICE-HELLOWORLD", fallback = HelloWorldServiceFailure.class)

2 public interface HelloWorldService {

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

4 public String sayHello();

5 }

3). 创建HelloWorldServiceFailure类， 必须实现被@FeignClient修饰的HelloWorldService接口。注意添加@Component或者@Service注解，在Spring容器中生成一个Bean

[复制代码](javascript:void(0);)

1 @Component

2 public class HelloWorldServiceFailure implements HelloWorldService {

3 @Override

4 public String sayHello() {

5 System.out.println("hello world service is not available !");

6 return "hello world service is not available !";

7 }

8 }

[复制代码](javascript:void(0);)

4）. Spring Cloud之前的Brixton版本中，Feign是缺省是自动激活了断路器的，但最近的Dalston版本已经将缺省配置修改为禁止。

原因参见：  <https://github.com/spring-cloud/spring-cloud-netflix/issues/1277>， 这一点要注意。所以要在Feign中使用断路器， 必须在application.yml中添加如下配置：

feign:

hystrix:

enabled: true

5）. 启动Feign应用， 访问http://localhost：8902/hello， 可以一看到和Ribbon一样的效果。

# [断路器指标数据监控Hystrix Dashboard 和 Turbine](https://www.cnblogs.com/chry/p/7286601.html)

<http://www.cnblogs.com/chry/p/7286601.html>

**1. Hystrix Dashboard (断路器：hystrix 仪表盘)**

Hystrix一个很重要的功能是，可以通过HystrixCommand收集相关数据指标. Hystrix Dashboard可以很高效的显示每个断路器的健康状况。

1）. 在Ribbon服务和Feign服务的Maven工程的pom.xml中都加入依赖

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

spring-boot-starter-actuator用于手机metric， 支持hystrix.stream。spring-cloud-starter-hystrix-dashboard支持dashboard的UI

2）在Spring Boot启动类上用@EnableHystrixDashboard注解和@EnableCircuitBreaker注解。需要特别注意的是我们之前的Feign服务由于内置断路器支持， 所以没有**@EnableCircuitBreaker注解，但要使用Dashboard则必须加**，如果不加，Dashboard无法接收到来自Feign内部断路器的监控数据，会报“Unable to connect to Command Metric Stream”错误

[复制代码](javascript:void(0);)

1 @SpringBootApplication

2 @EnableDiscoveryClient

3 @EnableFeignClients

@EnableCircuitBreaker

@EnableHystrixDashboard

6 public class ServiceFeignApplication {

7

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

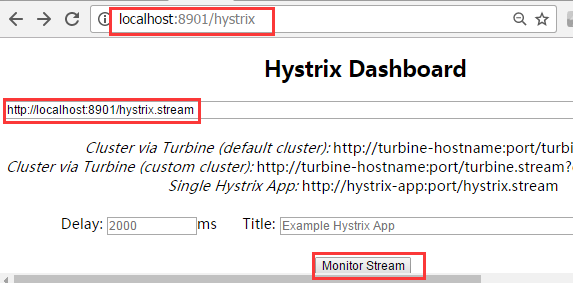
9 SpringApplication.run(ServiceFeignApplication.class, args);

10 }

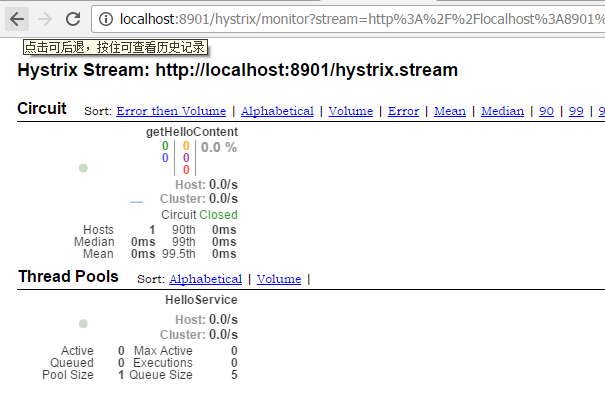
11 }

[复制代码](javascript:void(0);)

3）然后就可以访问http://localhost:8901/hystrix，这个URL将dashboard指向定义在Hystrix客户端应用中的http://localhost:8901/hystrix.stream



在dashboard中输入服务的URL：点击 monitor后进入监控界面，访问我们之前创建的Ribbon服务localhost:8901/, 或者Feign服务localhost:8902/可以看到监控UI动态变化



**2. 利用Turbine在一个Dashboard上监控多个流**

以上例子只能监控一个，要同时监控多个流怎么办？ 答案是， 可以单独做一个Turbine服务，专门监控所有断路器状态，从而掌握整个系统中所有微服务的状态。下面我们就来创建一个Turbine服务，来监控我们之前做的Feign服务和Ribbon服务

**1）.  创建一个maven工程， 在pox.xml添加以下依赖**

[复制代码](javascript:void(0);)

1 <dependency>

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

3 <artifactId>spring-cloud-starter-turbine</artifactId>

4 </dependency>

5 <dependency>

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

7 <artifactId>spring-cloud-netflix-turbine</artifactId>

8 </dependency>

9 <dependency>

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

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

12 </dependency>

13 <dependency>

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

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

16 </dependency>

[复制代码](javascript:void(0);)

整个个pox.xml文件如下：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif pom.xml

**2）. 创建Turbine Dashboard启动类:**

用@EnableHystrixDashboard和@EnableTurbine修饰主类， 分别用于支持Hystrix Dashboard和Turbine

[复制代码](javascript:void(0);)

1 package spring.helloworld.turbine.service;

2

3 import org.springframework.boot.SpringApplication;

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

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

6 import org.springframework.cloud.netflix.turbine.EnableTurbine;

7

8 @SpringBootApplication

9 @EnableHystrixDashboard

10 @EnableTurbine

11 public class DashboardApplication {

12

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

14 SpringApplication.run(DashboardApplication.class, args);

15 }

16 }

[复制代码](javascript:void(0);)

**3). 在application.yml中配置turbine参数**

[复制代码](javascript:void(0);)

1 eureka:

2 client:

3 serviceUrl:

4 defaultZone: http://localhost:8761/eureka/

5 server:

6 port: 8903

7 spring:

8 application:

9 name: hystrix-dashboard-turbine

10 turbine:

11 appConfig: service-feign, service-ribbon

12 aggregator:

13 clusterConfig: default

14 clusterNameExpression: new String("default")

[复制代码](javascript:void(0);)

turbine.appConfig定义了要监控的服务，这里是我们在前面章节创建的service-feign和sercice-ribbon; aggregator.clusterConfig定义了聚合方式， 此处为default.

turbine.appConfig ：配置Eureka中的serviceId列表，表明监控哪些服务

turbine.aggregator.clusterConfig ：指定聚合哪些集群，多个使用”,”分割，默认为default。可使用http://.../turbine.stream?cluster={clusterConfig之一}访问

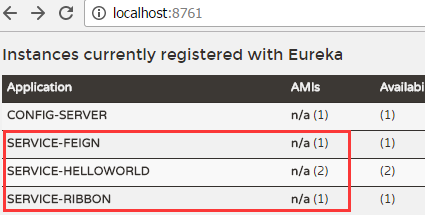
turbine.clusterNameExpression ：指定集群名称，可以是三种类型的值

         - 默认表达式为appName；此时turbine.aggregator.clusterConfig需要配置想要监控的应用名称；

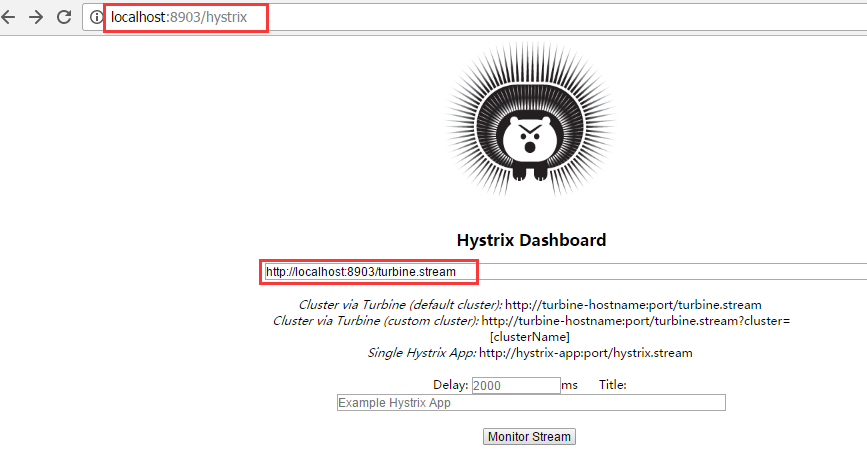
         - 当为default时，turbine.aggregator.clusterConfig可以不写，因为默认就是default；

         - 当为metadata[‘cluster’]时，假设想要监控的应用配置了eureka.instance.metadata-map.cluster: ABC，则需要配置，同时turbine.aggregator.clusterConfig: ABC

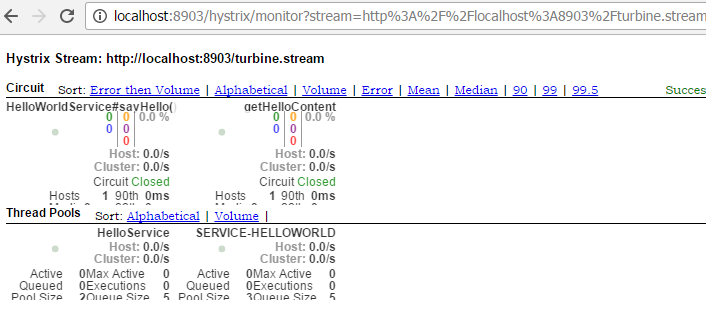
4）. 依次启动eureka服务， 2个Helloworld服务， Feign服务，ribbon服务和刚创建turbine服务。从eureka服务中我们可以看到



5）通过Turbine服务访问HystrixDashborad， http:localhost:8903/hystrix



 监控流的URL填http://localhost:8903/turbine.stream, 点击monitor stream, 进入监控页面， 随便刷新下feign和ribbon服务（http://localhost:8902/hello和http://localhost:8901）， 可以看到监控页面的变化。如下图， 两个服务的监控都会显示在dashboard上



参考：

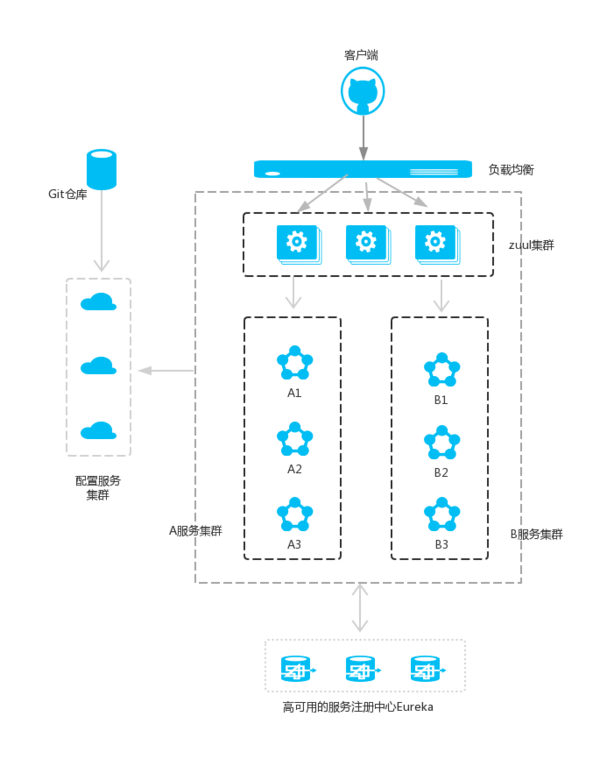
http://projects.spring.io/spring-cloud/spring-cloud.html#\_circuit\_breaker\_hystrix\_dashboard

<http://blog.csdn.net/ityouknow/article/details/72625646>

# [路由网关zuul](https://www.cnblogs.com/chry/p/7299844.html)

<http://www.cnblogs.com/chry/p/7299844.html>

在微服务架构中，需要几个关键的组件，服务注册与发现、服务消费、负载均衡、断路器、智能路由、配置管理等，由这几个组件可以组建一个简单的微服务架构。客户端的请求首先经过负载均衡（zuul、Ngnix），再到达服务网关（zuul集群），然后再到具体的服务，服务统一注册到高可用的服务注册中心集群，服务的所有的配置文件由配置服务管理（下一篇文章讲述），配置服务的配置文件放在Git仓库，方便开发人员随时改配置。



1. Zuul介绍

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。以下是微服务结构中，Zuul的基本流程。在接下来的步骤中，我们来创建一个zuul服务， 将/api-feign/\*\*映射到我们之前创建feign-service, 将/api-ribbon/\*\*映射到之前的ribbon-service服务。

 2. 创建Zuul的Maven工程，其中关于zuul的依赖是

<dependency>

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

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

</dependency>

 完整pom.xml如下：

<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>org.lys.demo</groupId>

<artifactId>helloworld.zuul.service</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>helloworld.zuul.service</name>

<url>http://maven.apache.org</url>

<parent>

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

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

<version>1.5.3.RELEASE</version>

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

</parent>

<properties>

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

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<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-web</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-zuul</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>Dalston.RC1</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>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

3. 创建启动类: 使用@EnableZuulProxy注解

[复制代码](javascript:void(0);)

1 package spring.helloworld.zuul.service;

2

3 import org.springframework.boot.SpringApplication;

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

5 import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

6 import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

7

8 @EnableZuulProxy

9 @EnableEurekaClient

10 @SpringBootApplication

11 public class ServiceZuulApplication {

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

13 SpringApplication.run(ServiceZuulApplication.class, args);

14 }

15 }

[复制代码](javascript:void(0);)

4. 编写zuul服务配置：

简单配置两个路由， 一个路由到ribbon，一个路由到feign； 由于都注册到eureka服务中心，所以都用通过serviceId来发现服务具体地址， path是路由的地址映射关系

[复制代码](javascript:void(0);)

1 eureka:

2 client:

3 serviceUrl:

4 defaultZone: http://localhost:8761/eureka/

5 server:

6 port: 8904

7 spring:

8 application:

9 name: service-zuul

10 zuul:

11 routes:

12 ribbo:

13 path: /api-ribbon/\*\*

14 serviceId: service-ribbon

15 feign:

16 path: /api-feign/\*\*

17 serviceId: service-feign

[复制代码](javascript:void(0);)

这时启动zuul服务， 然后访问http://localhost:8904/api-ribbon可直接路由到http://localhost:8901/.

http://localhost:8904/api-feign/hello可路由到http://localhost:8902/hello

5. Zuul过滤器

zuul还提供了过滤功能， 只要实现接口ZuulFilter即可对请求先进行筛选和过滤之后再路由到具体服务。

[复制代码](javascript:void(0);)

1 package spring.helloworld.zuul.service;

2

3 import javax.servlet.http.HttpServletRequest;

4

5 import org.slf4j.Logger;

6 import org.slf4j.LoggerFactory;

7 import org.springframework.stereotype.Component;

8

9 import com.netflix.zuul.ZuulFilter;

10 import com.netflix.zuul.context.RequestContext;

11

12 @Component

13 public class DemoFilter extends ZuulFilter {

14 private static Logger log = LoggerFactory.getLogger(DemoFilter.class);

15 @Override

16 public String filterType() {

17 return "pre";

18 }

19

20 @Override

21 public int filterOrder() {

22 return 0;

23 }

24

25 @Override

26 public boolean shouldFilter() {

27 return true;

28 }

29

30 @Override

31 public Object run() {

32 RequestContext ctx = RequestContext.getCurrentContext();

33 HttpServletRequest request = ctx.getRequest();

34 String s = String.format("%s >>> %s", request.getMethod(), request.getRequestURL().toString());

35 log.info(s);

36 return null;

37 }

38 }

[复制代码](javascript:void(0);)

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

* pre：路由之前
* routing：路由之时
* post： 路由之后
* error：发送错误调用

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，这里只是将请求的URL简单些到日志中

# 学习小结

SpringCloud是基于SpringBoot基础上搭建，了解到的概念有：服务注册（eureka）、配置管理（config、git）、负载均衡（ribbon、feign）、断路器（hystrix、hystrix-dashboard）、路由网关（zuul）